目录

[**1.** **防止工具反编译** 2](#_Toc443330710)

[1.1 伪加密 2](#_Toc443330711)

[1.2 APK压缩包故意破坏 2](#_Toc443330712)

[1.3 素材中的图片故意破坏 2](#_Toc443330713)

[**2.** **代码高级混淆** 2](#_Toc443330714)

[2.1 花指令在源码被窃后增加黑客阅读难度 2](#_Toc443330715)

[2.2 文件夹混淆(正式版本文件夹取名尽量不要太正规) 2](#_Toc443330716)

[2.3 文件名字混淆 3](#_Toc443330717)

[2.4 运行时验证 3](#_Toc443330718)

[**3.** **防止内存修改** 4](#_Toc443330719)

[4.1 如何防止“八门神奇”通过内存数据 4](#_Toc443330720)

[**4.** **进程注入** 4](#_Toc443330721)

[5.1 静态注入 4](#_Toc443330722)

[5.2 动态注入 4](#_Toc443330723)

[**5.** **附录** 4](#_Toc443330724)

1. **防止工具反编译**
   1. 伪加密

伪加密是Android4.2.x系统发布前最流行的加密方式之一，通过java代码对APK(压缩文件)进行伪加密，其修改原理是修改连续4位字节标记为“P K 01 02”的后第5位字节，奇数表示不加密偶数表示加密。伪加密后的APK不但可以防止PC端对它的解压和查看也同样能防止反编译工具编译。

但是伪加密对其APK加密后市场也无法对其进行安全检测，部分市场会拒绝这类APK上传市场。伪加密的加密方式和解密方式也早已公布导致它的安全程度也大大降低。Android4.2+系统由于修改了签名验证的方式导致无法安装伪加密的APK。

* 1. APK压缩包故意破坏

APK在PC上面也可以看作一个压缩文件，在Android系统里面它是一个手机系统软件文件。Android系统对APK的识别是从标志头到标志尾，其他多余数据都会无视。所以说在标志尾添加其他数据对把APK看做压缩文件的PC端来说这个文件被破坏了，所以你要对其进行解压或者查看都会提示文件已损坏，用反编译工具也会提示文件已损坏，但是它却不会影响在Android系统里面的正常运行和安装而且也能兼容到所有系统。  
 但是这种APK压缩包破坏，存在个别市场会不能识别导致不能上传市场。使用压缩文件修复工具也能把它修复好让我们做的保护消失。

* 1. 素材中的图片故意破坏

素材破坏和包破坏的原理其实差不多，这种破坏也只是针对视apk为压缩文件的pc来说的。具体的做法是：在开发工具中(例如：eclipse)在打包前将jpg格式的图片强行修改成png（由于jpg和png格式图片的识别格式不一样 强行修改后 压缩文件在被解压缩的时候会对任何格式的文件进行验证，在验证到图片格式的时候 会因为文件类型与格式不一样导致无法反编译）。。这种保护措施不能防止查看主要是防止工具反编译(例如:apktool)，前提是不会影响其apk在手机上的正常运行。

1. **代码高级混淆**
   1. 花指令在源码被窃后增加黑客阅读难度

花指令是程序中有一些指令，由设计者特别构思，希望使反汇编的时候出错，让破解者无法清楚正确地反汇编程序的内容，迷失方向。【花指令】这个词来源于汇编语言，它的思想是非常不错的。【花指令】另外目的就是利用反编译工具漏洞，来使工具无法使用。接下来 我们就在java代码处制造【花指令】，让反编译工具(jd-gui)无法反编译查询你的java代码。

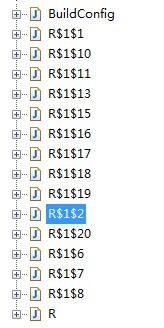
* 1. 文件夹混淆(正式版本文件夹取名尽量不要太正规)

文件夹混淆主要指的是 利用windows,linux,android 三个系统环境下的文件夹名的特殊性来对源码文件夹进行混淆，让混淆后的文件夹在window看起来失去原有的逻辑性，但是完全不影响其在android系统上的运行。

原理：在windows和linux下文件夹的名字是不区分大小写的，但是在android环境下它却要区分大小写。.2在linux算一个特殊符号，所以文件夹名字里面添加的.2会被忽略。。但是windows下 .2却是一个很普通的字符串。  
方法：反编译开发完成的apk，找到包目录下的最后一层文件夹(例如：包名是com.example.test2222,找到test2222所在的文件夹)，修改test2222文件夹名字为test2222.2并创建文件夹Test2222并随意存放一个有效的smali文件在Test2222里面，然后重新重写打包成apk,签名。

|  |
| --- |
| * 1. 文件名字混淆   大家一定知道proguard混淆，会对java的类名进行混淆，修改为a,b,c,d等等名字来混淆反编译的阅读，但是其混淆的类只能是开发者自己创建的类，对android原始类不能进行混淆。 原理：proguard混淆为基础，在开发完毕后统一修改自己程序的类名(包括主配文件也进行修改) 如下图：  这两类保护，不但能够使自己的apk在破解后很难被破解者阅读，还能防止一键反编译工具和检测工具，因为自己文件夹.2的方法已经破坏了windows下包路径的规律让其不能通过代码去根据主配文件里面的包名去寻找到类。 |

[**1.jpg**](http://www.apkbus.com/forum.php?mod=attachment&aid=MTI0MDM5fDNiMmU0NGZjfDE0NTU0Mzk0ODh8MHwxNDUyNDU%3D&nothumb=yes) (18.25 KB, 下载次数: 8)



* 1. 运行时验证

     运行时验证，主要是指在代码启动的时候本地获取签名信息然后对签名信息进行检验来判断自己的应用是否是正版，如果签名信息不是正版则提示盗版或者直接崩溃。  
 原理：APK的唯一识别是根据包名+签名，包名信息是写死在AndroidManifest.xml里面的，但是签名则是与APK绑定的，一旦APK被反编译后签名会自动消失。APK的签名需要签名文件，签名文件的md5值基本上是无法伪造成一样的。  
签名验证的方法也可以细分为3种：  
1) Java 层验证

获取签名信息和验证的方法都写在android 的java层。这种保护方法保护的意义并不大，因为反编译出源码后通过关键字搜索很快就能够找到验证的代码块，稍微一修改这验证保护就完全无效了。  
目前市场上使用此方法验证的应用：神庙逃亡2，qq电池管家，微信，360手机管家等  
2)服务器验证  
 在android 的java层获取签名信息，上传服务器在服务端进行签名然后返回验证结果。这种保护还不如在纯java层验证有用，一旦没有网络验证保护就无效了。用android方法获取的签名信息用java方法也可以获取，验证存放在服务器上也是为了把保护正确的签名信息值，但是保护的意义其实没有任何作用。同样破解后全局搜索关键字然后伪造一个正确的签名信息就可以完美破解了。  
目前市场上使用此方法验证的应用：地铁跑酷等  
3) NDK技术底层获取签名和验证

通过把Context,Activity,PackageManager,PackageInfo四个对象中的一个作为参数参入底层，在底层获取签名信息并验证。因为获取和验证的方法都封闭在更安全的so库里面，能够起到一定意义上的保护作用。不过通过java层的hook技术一样可以把这种保护完美破解。 但是相比于前两种，此保护的意义和价值就更大了。  
目前市场上使用此方法验证的应用：植物大战僵尸2等

1. **防止内存修改**
   1. 如何防止“八门神奇”通过内存数据
2. **进程注入**
   1. 静态注入

静态注入比较经典的案例就是二次打包了。比如说 登陆注册的输入框里面注入钩子，盗取账号密码。 防止静态注入就得防止二次打包了。防止二次打包的方式很多的。

* 1. 动态注入

动态注入的经典例子就是内存修改器比如说八门神奇。 防止内存修改就得编码的时候注意了，比如说：关键变量(当前分数，当前时间，当前血量，当前等级 等等) 写两份，一份加密的 一份未加密的，定时用加密那份来检验。

1. **附录**
   1. 一致性参考代码

@Override

public void onCreate(Bundle savedInstanceState)

{

super.onCreate(savedInstanceState);

getSingInfo();

boolean bCorrectSign = false;

if (bCorrectSign){

// Set by <content src="index.html" /> in config.xml

loadUrl(launchUrl);

}

else {

System.exit(0);

}

}

private String toHex(byte[] paramArrayOfByte) {

StringBuffer localStringBuffer = new StringBuffer();

for (int i = 0; i < paramArrayOfByte.length; i++) {

Object[] arrayOfObject = new Object[1];

arrayOfObject[0] = Byte.valueOf(paramArrayOfByte[i]);

localStringBuffer.append(String.format("%02x", arrayOfObject));

}

return localStringBuffer.toString();

}

private void parseSignature(byte[] signature) {

try{

CertificateFactory certFactory = CertificateFactory.getInstance("X.509");

X509Certificate cert = (X509Certificate) certFactory.generateCertificate(new ByteArrayInputStream(signature));

byte[] buffer = cert.getEncoded();

String pubKey = cert.getPublicKey().toString();

String signNumber = cert.getSerialNumber().toString();

String na = cert.getSigAlgName();

String dn = cert.getSubjectDN().toString();

buffer = cert.getEncoded();

} catch (CertificateException e) {

e.printStackTrace();

}

}

public void getSingInfo(){

try {

PackageInfo packageInfo = getPackageManager().getPackageInfo(getPackageName(), PackageManager.GET\_SIGNATURES);

android.content.pm.Signature[] signs = packageInfo.signatures;

Signature sign = signs[0];

MessageDigest localMessageDigest = MessageDigest.getInstance("MD5");

localMessageDigest.update(sign.toByteArray());

// md5 stringֵ

String str2 = toHex(localMessageDigest.digest());

//int h = sign.hashCode();

//parseSignature(sign.toByteArray());

} catch (Exception e){

e.printStackTrace();

}

}

* 1. 防止备份,防止其它程序调用

修改AndroidManifest.xml application 增加 android:allowBackup="false" android:exported="false"

* 1. https证书不校验，导致中间人攻击漏洞检测的问题

修改FileTransfer.java中 https.setHostnameVerifier(DO\_NOT\_VERIFY); 改为 https.setHostnameVerifier(STRICT\_HOSTNAME\_VERIFIER);

* 1. 增加防UI截持代码

修改MainActivity.java 中的onCreate方法