**Android非ROOT环境下的动态库注入**

**设计背景**

一般来讲，研究某个app，分析其协议和算法，必须要有一台root好的手机。这样可以对手机系统进行全方位的访问和控制。比如安装javahook框架xposed，必须要有root权限；直接增、删、改app安装目录下的文件，需要root权限；抓包工具的系统证书设置也需要root权限等。另外，在root手机和正常收上app的行为也可能发生变化，比如设备信息的获取，在root过的手机上可能会得到一些非正常数据，导致app不能正常使用或提示安全风险警告等。

但手机想要获得root权限却并不容易，特别是现在各大手机厂商都陆续关闭了官方root通道的情况下，想要获取root权限只能通过重新刷机，第三方root工具等手段获取，不仅麻烦且root的手机多数只能是老旧机型，比如笔者的root手机还是Android8,而最新版本都已经Android12了。所以催生出了在非root的手机上进行app注入的想法，本文分享的正是笔者在这方面研究出的一些尚未成熟的个人经验。

**研究环境**

1. **手机环境**

Android10及以下版本的正常手机都可，笔者的研究手机是一台正常的小米公司的红米9手机，系统版本为Android10。

**二、测试App**

测试用的app是facebook最新版本v351(armeabi-v7a)。可以通过下面镜像地址下载：

<https://www.apkmirror.com/>

com.facebook.katana\_351.0.0.0.103-310211054\_minAPI23(armeabi-v7a)

(420,400,360,480dpi)\_apkmirror.com.apk

1. **依赖工具**

Apk包编译和反编译工具apktool，地址https://ibotpeaches.github.io/Apktool/。笔者使用的是2.6.0版本。

Apk重签名工具apksigner，这个是android sdk中自带的工具，可以在build-tools目录中找到，笔者用的版本是0.9版本。

Java层hook框架，用的是笔者自己开发的一个轻量级的框架。关于此框架的介绍文章

请在同程技术公众号中搜索《安卓ART虚拟机Java Hook框架JavaTweak的设计与实现》，github地址**<https://github.com/liaoguobao/javahook>**

**注入原理**

要让原生app加载我们开发的so或dex文件，重打包是必须的，因为必须要放置至少一个so或dex文件到原apk中。我的方案是放置一个通用的so(假定其名称为**libx.so**)到apk中，并静态更改某个一定会加载的so的依赖列表，让其依赖我们的libx.so，再重新打包回去。

这样，在正常手机上我们重新打包好的app安装就没有任何问题了，但由于我们重新打包签名了，原始app的签名证书势必遭到了破坏，在app本身防重打包，防重签名的校验下肯定会过不去，导致app无法运行。

app本身防重打包，防重签名的校验主要又分为两个种情形。一种是动态校验，主要用在包签名数据校验上，这种绕过思路是通过hook获取签名数据的java方法，将正确的签名数据返回即可；另一种是直接解析安装好的静态apk包，从包中获取原始数据，防止第三方重打包，这种方式防范能力很强，校验点也很多，检测可以放在java层也可以放在native层，如果挨个挨个去绕过显然费时费力，且肯定会有漏网之鱼。我的思路也很简单，直接hook fopen函数，因为要读取包数据，fopen调用是必须的，我在这里过滤一下打开的文件是否是当前apk包。如果是，我会重定向包路径到原始包路径上去，这样不管什么样的检测手段都失效了。

综上所述，要让我们的重打包的apk正常运行起来，至少还要一个so(fopen的hook，假定其名称为**libsotweak.so**)和一个dex(签名数据获取方法的hook,假定其名称为**javatweak.dex**)。而这两个文件，只能由libx.so加载了，并且这两个文件和app分析绑定紧密，需要经常变动，肯定不能打包到apk中，否每次改动都需要重新安装。我的方案是在sdcard目录下新建一个tweak目录，将原始apk、so、dex文件都放在此处，由libx.so统一管理。

另外，在android7及后续的版本中，不允许加载sdcard目录下的so。所以，每次app启动时，libx.so需要负责将/sdcard/tweak下的so文件拷贝到app私有的data目录下，然后再加载,为了便于管理，会在私有data目录下新建app\_tweak目录，用以放置libsotweak.so、javatweak.dex以及优化后的oat文件等。

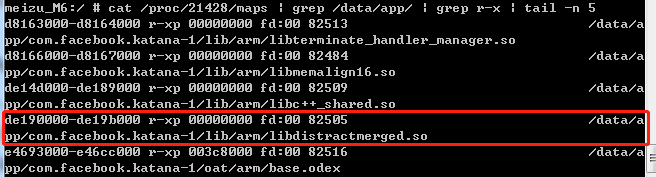
**注入过程**

**一、确定被编辑的so**

选取一个so，修改其依赖列表项。最好这个so是第一个被加载起来的so，因为签名校验可能发生在任何so中，如果不赶在校验之前注入我们的so从而修复校验问题的话，app就运行不起来或者发生某些意想不到的错误。第一个加载的so，一般是加固公司提供的，

通过如下命令可以大概定位出来

cat /proc/PID/maps | grep /data/app/ | grep r-x



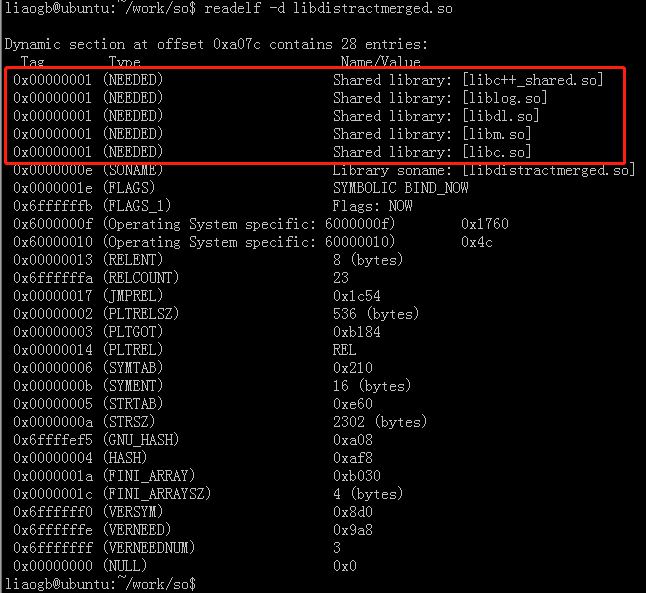
其中基址被映射的最大的一个so最先被加载，上图可以看出facebook的第一个被加载的so名叫libdistractmerged.so，接下来就在这个so上添加依赖项。

**二、添加依赖**

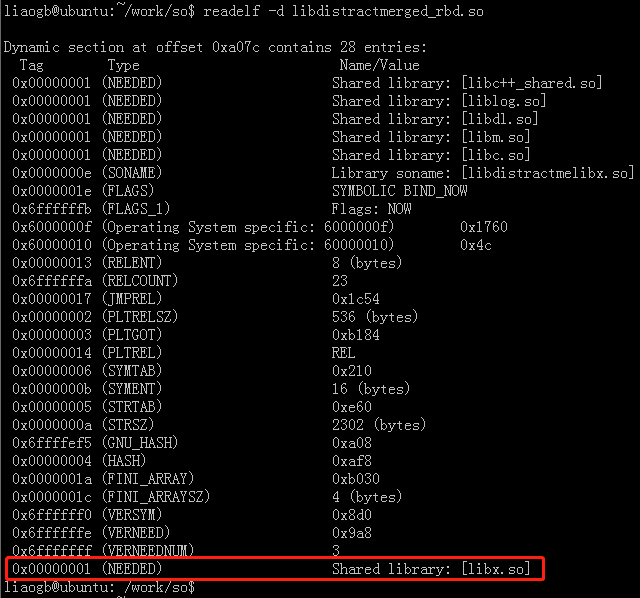
添加依赖项的工具可以在网上搜索的到，也可以自己编写一个，本文对应的github地址中也附带了一个附加小工具rbdso，只有附加so的简单功能,注入命令如下。

d:\rbdso.exe -f d:\libdistractmerged.so -i libx.so

注入之前，查看.dynamic段数据可以看到有5个依赖项，如下图所示：



注入之后可以看到.dynamic段数据有6个依赖项了，新增了libx.so的依赖，如下图。



1. **重新打包**

整个重打包过程请在linux下完成，windwos下会有文件名不符合规则冲突。

第一步，反编译apk,仅仅只是提取apk中的原始文件，不反编译任何dex和资源文件，命令如下,将提取文件保存到d:\decode，解包命令如下.

java -jar apktool\_2.6.0.jar decode -f -r -s -o ./decode ./com.facebook.katana.apk



解包后的内容如下所示。



注意，facebook包，解包后的文件与原始包相比，缺少了整个r目录，这里要收手动拷贝整个r目到unknown中【cp -rf com.facebook.katana/r decode/unknown/】

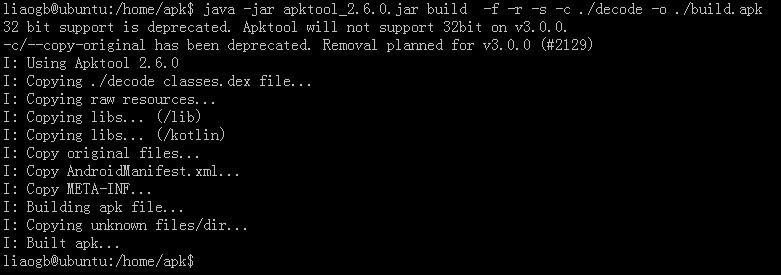
第二步，将libx.so和更改过的libdistractmerged.so放入到decode\lib目录中。libx.so是一个通用的so，其主要功能就是拷贝/sdcard/tweak目录下的libsotweak.so文件到app的私有/data/app\_tweak目录下，再将其加载起来。核心代码如下所示。



注意，必须在设置中为APP开启【手机存储】权限，否则复制文件会失败。

第三步，重新打包刚才的decode目录为build.apk文件，命令行如下：

java -jar apktool\_2.6.0.jar build -f -r -s -c ./decode -o ./build.apk

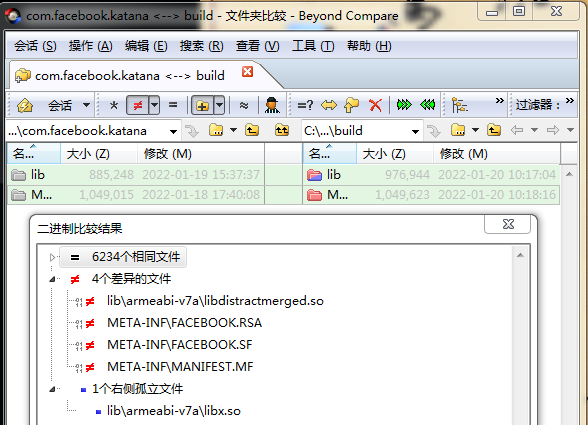


第四步,重签名，采用android默认的debug.keystore进行签名，命令行如下：

java -jar apksigner.jar sign --v3-signing-enabled=false -ks debug.keystore

--ks-pass pass:android --key-pass pass:android --in ./build.apk

将build.apk包的ANDROIDD.RSA、ANDROIDD.SF文件重命名为FACEBOOK.RSA、FACEBOOK.SF再和原始包进行二进制比较，结果如下：



可以看到，处包理签名相关的三个文件不同之外，就是多了一个libx.so，libdistractmerged.so被编辑过，也显示与原文件有差异。

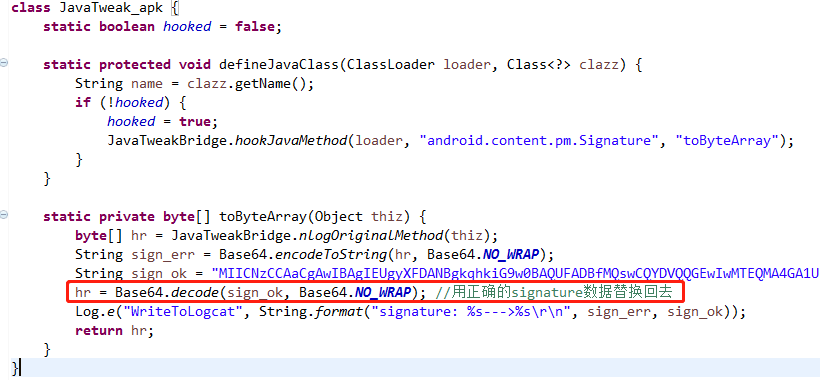
**四、Native层修正处理**

我们重建的build.apk经过签名之后，已经可以正常安装到手机上了，但现在还只是能安装上，并不能运行，还有几处重签名校验需要修复。libsotweak.so负责处理打开apk文件的路径修复，核心修复代码如下图所示。



**五、Java层修正处理**

java层只需要hook返回签名数据的函数并让其返回原始签名即可，核心代码如下。



将java代码打包为javatweak.dex。

如何打包，请参考**<https://github.com/liaoguobao/javahook>**中的文档说明。

**六、安装build.apk**

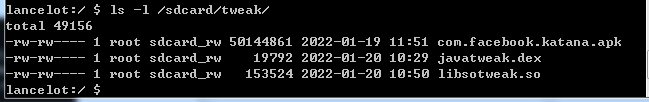
第一步，通过adb命令安装我们打包好的apk文件，安装命令入下：

adb install d:\build.apk

第二步，在/sdcard目录下新建一个tweak目录

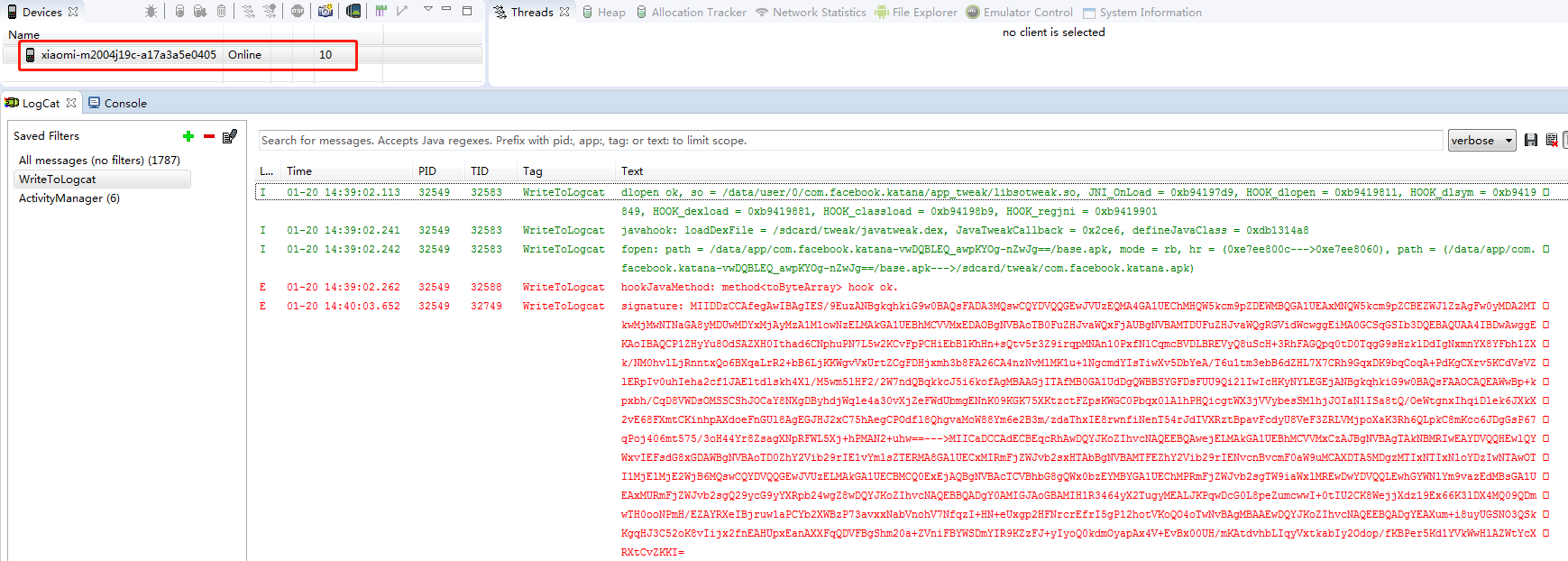
第三步，将原始apk文件，libsotweak.so，javatweak.dex三个文件push到

/sdcard/tweak目录中，apk文件名为{包名}.apk,如下图所示：



第四步，在【设置】中为app开启【读写手机存储】权限，允许app访问/sdcard/tweak目录下的所有文件。

经过上面四步操作之后，重打包的【com.facebook.katana.apk】就可以在非root的正常手机上运行起来了。小米手机运行截图如下：

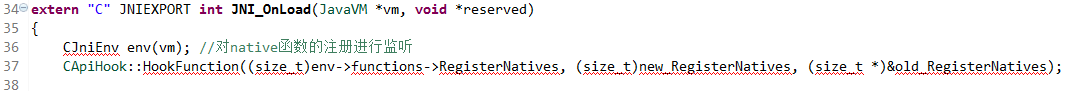


上图红色部分为java层的输出日志，绿色部分为native层输出日志。在这里可能会有一个疑问，javatweak.dex又是被谁加载的呢？答案肯定是被libsotweak.so啦，笔者将上一篇文章《安卓ART虚拟机Java Hook框架JavaTweak的设计与实现》中提供的native层源代码也编译进了libsotweak.so中，以支持java方法的hook。

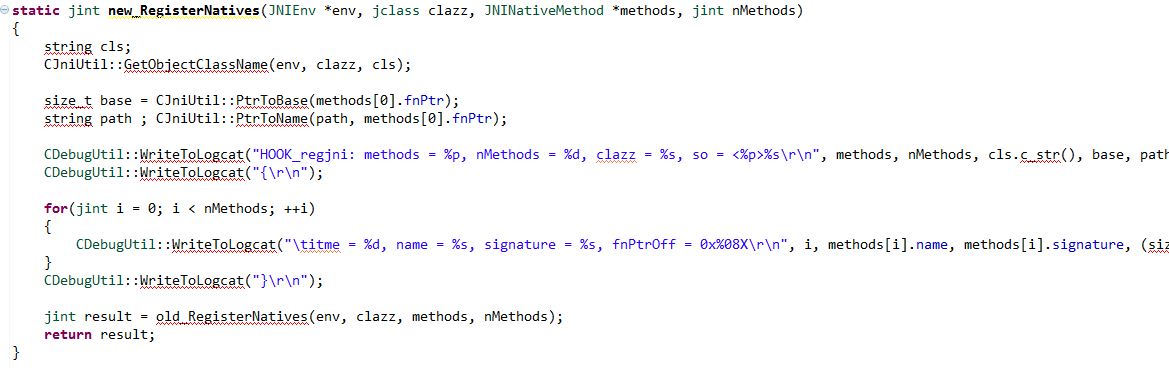
**扩展测试**

现在我们的build.apk包除了能正常运行之外，还没有其它功能。因为libsotweak.so这个so并没有打包进apk中，所以更新这个so不需要重新打包，只需要重启一下app即可生效。下面我们把libsotweak.so的功能做一下扩展，做一个JNI注册监听测试。

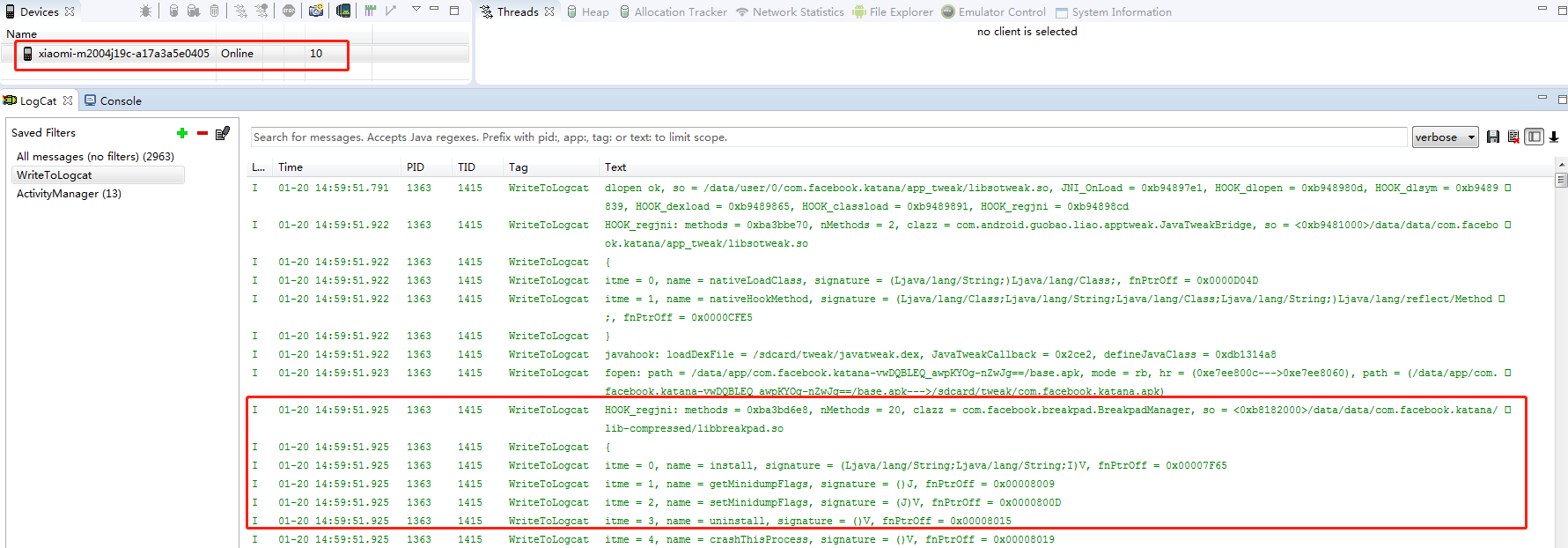
可以在libsotweak.so的JNI\_OnLoad中加入如下的核心代码：



在hook函数中，增加日志打印。



更新/sdcard/tweak/libsotweak.so，重启app，会有类似如下输出日志。



**总结分析**

以上就是笔者对非root手机动态库注入的一些粗浅看法，由于时间有限，只在小米Android10一台机器上验证过。理论上其它app依照这个思路应该也是可以成功的，之所以只支持到Android10，是因为javahook部分只研究到Android10。对于新的Android11和Android12，java方法的hook并没有去研究了，没跟上系统版本的更新速度。

一般来讲，如果要分析某个app的功能，基本上就是在root过的手机上进行。在正常手机上研究有很大限制，但优势是手机容易获取，没有root的麻烦。更重要一点的是，设备信息的采集真实准确，这对于绕过那些专门通过设备信息来阻止模拟请求的防范措施或封禁策略非常有用。这也是我研究这方面技术的原因之一吧！！！

文章中的源代码截图只供参考，完整demo请从GitHub下载。

<https://github.com/liaoguobao/buildapk>