Xposed原理

这块知识本身是挺多的，网上有对应的源码分析，本文尽量从不分析代码的角度来把原理阐述清楚。

Xposed是一个在andoid平台上比较成熟的hook框架，可以完美的在dalvik虚拟机上做到hook任意java方法。在art虚拟机上仍然处在beta阶段，相信以后也会稳定支持。

Xposed在dalvik上的hook原理值得好好学习，这样才能改造它，或者开发类似的hook框架。

Dalvik虚拟机的代码分析参考这篇文章：  
<http://blog.csdn.net/innost/article/details/50377905>  
dex文件格式参考这篇文章：  
<http://www.blogfshare.com/dex-format.html>  
以上两篇文章写得非常好，当你对着源码来学习就会发现。

这里做一个总结，顺便说一些这些文章里面没有写的内容：

java源码经过编译后，得到很多个class文件， 考虑到手机的内存较小，google改进了字节码的组织形式，将一个app中的所有class文件合到了一起构成dex文件，当然并不是简单的拼接在一起，而是遵从dex的格式（参见上）重新组织。

dex文件最终会和资源文件等一起打包成为apk，签名后安装到手机上。

PackageManager在安装apk的时候，做了一件事：优化dex文件为odex，存放在/data/dalvik-cache目录下。

dex文件是遵从于dalvik虚拟机标准的文件，它具有跨dalvik虚拟机的特点，而odex是在特定dalvik虚拟机上优化得到的，通常不能跨dalvik虚拟机运行。

程序执行体现在方法的执行上，因为我们重点关注下方法的组织形式。

在dex文件中，方法体里面的内容最终存储在classData区域，方法体里面存储的是二进制的字节码。

在说字节码之前，先来说说什么是dalvik虚拟机，dalvik虚拟机说白了就是用c/c++写的一套复杂的程序，它定义了一堆的smali指令（256个），这些字节码指令高度抽象，组合这些指令可以完成我们想要的功能。你可以等同于汇编指令理解，但它们在语言级别要高于汇编（在虚拟机里面执行的，虚拟机又是c/c++写的，自然看出高于汇编）。

字节码的定义参考：  
<http://code.metager.de/source/xref/android/4.0.3/dalvik/libdex/DexOpcodes.h>  
字节码是用简单的二进制数字表示的，与可阅读的smali指令存在对应关系，具体对应方法参考google的官方文档：  
<http://source.android.com/devices/tech/dalvik/instruction-formats.html><http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html>

我们只要知道有这个对应关系就好了， 已经有人写了工具来做这个转化：smali/baksmali，这个工具可以分析dex文件，解析字节码为对应的smali语法（反汇编），同时也可以将smali语法的文件重新转换为字节码生成dex文件（汇编）。

dex优化过程，其实是将一些字节码替换为dalvik相关的， 优化后的等价字节码。smali/baksmali同时也支持将odex转化为dex，不过你需要从odex产生的手机上dump出对应的文件（framework.jar等），传递给它，才能正确转化。

dump（备份文件系统）

(1)在特定时刻，将整个[储存](https://baike.baidu.com/item/%E5%82%A8%E5%AD%98" \t "_blank)装置或储存装置之某部分的内容记录在另一储存装置中。存储的目的通常是为了防止发生错误。 (2) 将具备可读格式的数据从主要或辅助储存体复制至外部[媒体](https://baike.baidu.com/item/%E5%AA%92%E4%BD%93" \t "_blank)，如[磁带](https://baike.baidu.com/item/%E7%A3%81%E5%B8%A6)、[磁盘](https://baike.baidu.com/item/%E7%A3%81%E7%9B%98)或[打印机](https://baike.baidu.com/item/%E6%89%93%E5%8D%B0%E6%9C%BA)等媒体。 (3) 为收集错误信息而复制整个虚拟储存体或虚拟储存体之某部分的内容。

虚拟机又是如何执行这些字节码的呢？  
答案在AOSP中（开源就是好），原来虚拟机对于每一个字节码，都写了一段代码来解释执行（你可以等价理解为API调用一样，调用某个API，后面一堆逻辑来实现这个API），只是不同cpu结构，实现方式不一样，比如arm使用了arm汇编来实现，x86使用了x86的汇编来实现，google提供了一个c/c++的实现方式来兼容一些未知的cpu结构，见：  
<https://android.googlesource.com/platform/dalvik2/+/master/vm/mterp/out/InterpC-portable.cpp>

虚拟机在加载了odex（虚拟机总是使用odex文件，第一次使用时会先生成odex）, 会将整个odex文件的内容mmap到内存中，之后就和odex文件没有关系了。

Mmap：将一个文件或者其它对象映射进内存。文件被映射到多个页上，如果文件的大小不是所有页的大小之和，最后一个页不被使用的空间将会清零。munmap执行相反的操作，删除特定地址区域的对象映射。

虚拟机在load一个Class的时候（参见DexClassLoader源码），根据类的描述符，在内存中的odex区域，查询到对应的数据，构建出ClassObject对象，以及这个ClassObject关联的Method。

Method分为两种，dalvik虚拟机在处理的时候有区别，一种是directMethod，即Java世界里面实现的方法，一种是nativeMethod，即在c/c++里面实现的方法。

ClassObject里面有两个集合，分别存放了这个Class下定义的directMethods和nativeMethods。

Method中，有两个非常重要的指针：

|  |
| --- |
| const u2\* insns;  DalvikBridgeFunc nativeFunc; |

对于directMethod，insns存放了该方法的字节码指针(还记得odex被mmap到内存中了么，这个指针就是这段内存里面指向code区域的开始处的指针)。  
虚拟机在调用directMethod时，在构建好方法栈以后，pc指针指向了insns，于是可以从内存中取得字节码，然后解释执行。

一些apk加固厂商就是在这块做的手脚，这里衍生开来说一下（加密的两种方式）：  
1.梆梆加固的实现方式为：将原始dex中的内容加密处理，在app运行时，解密出dex，mmap到内存，还原了内存结构。  
2.爱加密的方法则是，将方法体里面的字节码从dex中抠出来，加密到了自己的so中，在app运行时，从so中解密出方法体，然后修改mmap对应的内存，还原内存结构。

这些加固方法说白了就是将原始dex做加固处理，在运行时还原内存结构。所以光从静态分析反编译加固后的dex文件，将得不到有用信息。

但有一个基于xposed的zjdroid脱壳工具，可以在运行时dump出内存（odex结构的内存），保存为本地odex文件，再利用smali/baksmali还原出原始dex文件。

虚拟机在处理native方法时，走的是另外一套逻辑。

我们在使用native方法时，首先得使用System.loadLibrary对so进行加载，其最终是使用dlopen函数加载了指定的so文件。

之后在我们调用nativeMehtod的时候，会根据方法描述符，通过特定的映射关系（是否主动进行了注册会有不同）得到一个native层的函数名，再从之前dlopen获得的句柄中使用dlsys去查找对应的函数，得到了函数指针后，将这个指针赋值给 insns。在nativeFunc这个桥接函数中，将insns解析为函数指针，然后进行调用。

Xposed hook原理

有前面这些知识后，再理解Xposed的hook原理就不难了。  
前面已经知道，一个java方法在虚拟机里面对应的Method为directMethod，其insns指向了字节码位置。

Xposed在对java方法进行hook时，先将虚拟机里面这个方法的Method改为nativeMethod(其实就是一个标识字段)，然后将该方法的nativeFunc指向自己实现的一个native方法，这样方法在调用时，就会调用到这个native方法，接管了控制权。

在这个native方法中，xposed直接调用了一个java方法，这个java方法里面对原方法进行了调用，并在调用前后插入了钩子，于是就hook住了这个方法。

Xposed的hook原理就是这么简单，但它有其他的问题要解决：如何将hook的代码注入到目标app的进程中？

Xposed的实现是依赖与root，重写android的zygote代码，加入自身的加载逻辑。zygote是android 系统最最初运行的程序，之后的进程都是通过它fork（你把它理解为复制吧）出来的。 于是zygote中加载的代码，在所有fork出来的子进程都含有（app进程也是fork出来的）。 所以xposed是一个可以hook android系统中任意一个java方法的 hook框架。

而淘宝根据xposed改造出来的dexposed，仅仅是注入hook代码的方式不同而已，hook逻辑完全一致。  
dexposed不依赖与root，但需要开发者主动集成进来（我们集合了别人的广告sdk，其实也是让别人的程序跑到我们的进程里面，所以得小心点，给我一个入口，我也能hook住你的任何方法）。所以其hook的范围仅仅是被集成的应用（对于淘宝的AOP框架定义，这个效果刚刚好）。

转载至：https://www.jianshu.com/p/b29a21a162ad