[插件开发之360 DroidPlugin源码分析（四）Activity预注册占坑](http://blog.csdn.net/hejjunlin/article/details/52258434)

在了解系统的activity，service，broadcastReceiver的启动过程后，今天将分析下360 DroidPlugin是如何预注册占坑的？本篇文章主要分析Activity预注册占坑，Activity占了坑后又是什么时候开始瞒天过海欺骗AMS的？先看下Agenda：

* **AndroidMainfest.xml中概览**
* **Activity中关键方法被hook时机**
* **startActivity被hook**
* **handelPerformActivity被hook**
* **Activity预注册占坑整体流程图**
* **瞒天过海，冒充真实身份，欺骗AMS**

## AndroidMainfest.xml中概览

我们知道所有能用的四大组件都要在Manifest中注册声明，第一步，先看AndroidManifest.xml，虽然在说hook机制时，也有提及过，但是毕竟没细致分析，话不多说，看代码上图：



然后看下各个属性的意思

IMG_257

表示一个activity1原来属于task1，但是如果task2启动起来的话，activity1可能不再属于task1了，转而投奔task2去了。

IMG_258

功能：启动硬件加速   
缺点：占用内存   
特点：可以在Application、Activity、Window、View四个级别进行硬件加速控制

从Android3.0（API Level 11）开始，Android 2D渲染管道能够更好的支持硬件加速。硬件加速执行的所有的绘图操作都是使用GPU在View对象的画布上来进行的。因为启用硬件加速会增加资源的需求，因此这样的应用会占用更多的内存。

启用硬件加速的最容易的方法是给整个应用程序都打开全局硬件加速功能。如果应用程序只使用标准的View和Drawable，那么打开全局硬件加速不会导致任何的不良的绘制效果。但是，因为硬件加速并不支持所有的2D图形绘制操作，所以对于那些使用定制的View和绘制调用的应用程序来说，打开全局硬件加速，可以会影响绘制效果。问题通常会出现在对那些不可见的元素进行了异常或错误的像素渲染。为了避免这种问题，Android提供以下级别，以便可选择性的启用或禁止硬件加速：

控制硬件加速，能够用以下级别来控制硬件加速：

1、Application级别

在应用的Android清单文件中，把下列属性添加到元素中，来开启整个应用程序的硬件加速。

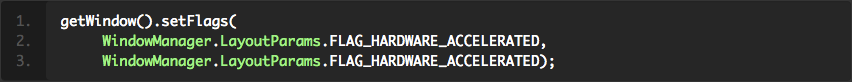
2、Activity级别

如果应用程序不能够正确的使用被打开的全局硬件加速，那么也可以对Activity分别进行控制。在元素中使用android:hardwareAccelerated属性，能够启用或禁止Activity级别的硬件加速。以下示例启用全局的硬件加速，但却禁止了一个Activity的硬件加速。

3、Window级别

如果需要更细粒度的控制，就可以使用下列代码来针对给定的窗口来启用硬件加速：

注意：当前不能在Window级别禁止硬件加速。



4、View级别

能够使用下列代码在运行时针对一个独立的View对象来禁止硬件加速：

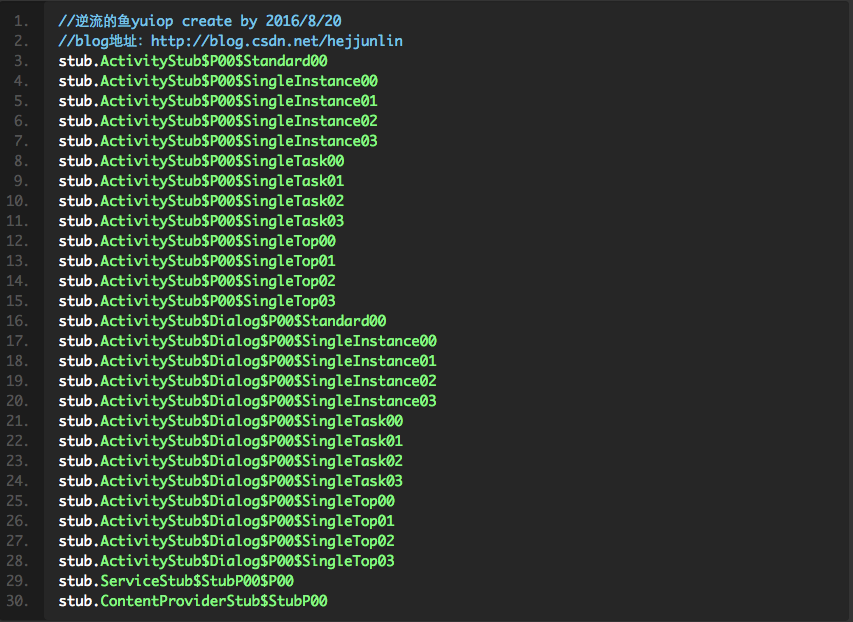
IMG_260

注意：当前不能在View级别开启硬件加速。View层除了禁止硬件加速以外，还有其他的功能，更多的相关信息请看本文的“View层”。



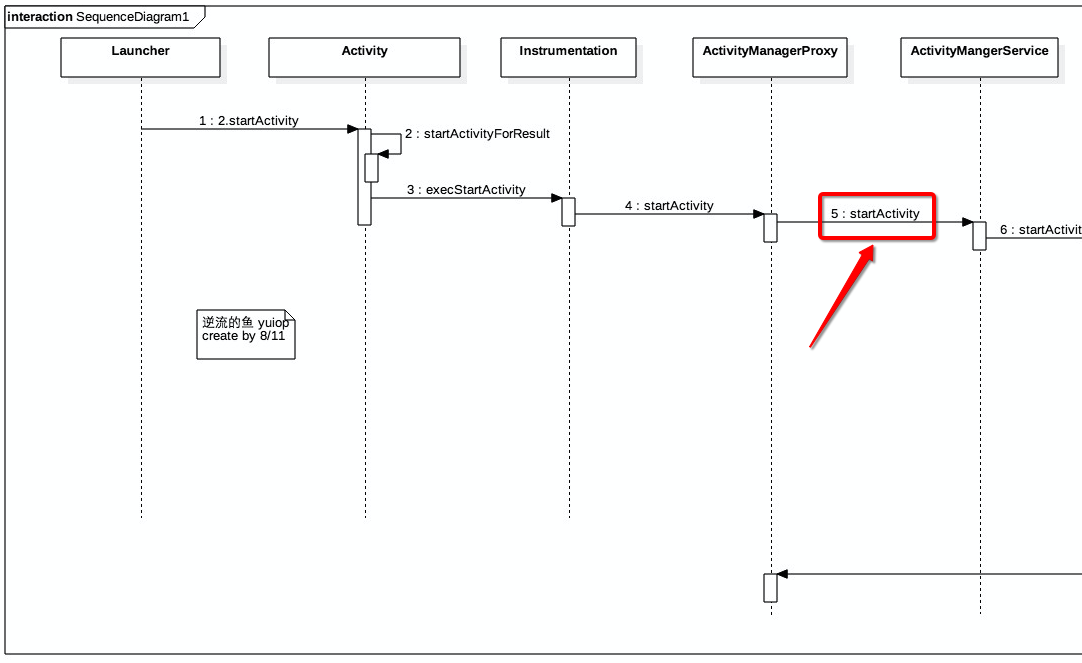


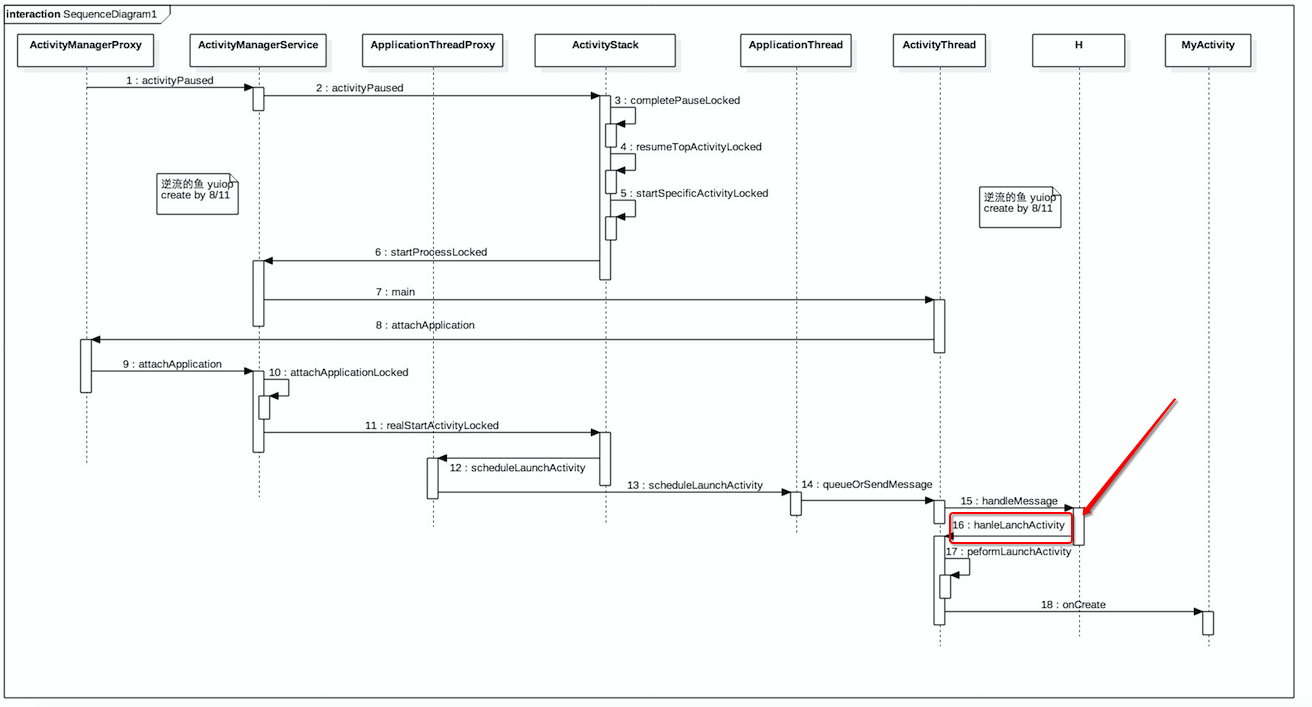
以上就是声明属性的含义说明，作为背景了解即可。重点看下面的分析



Manifest中注册了8个进程，加上主进程共9个   
然后第每个进程下面又有26个Activity注册，一个service，一个contentprovider,那么问题来了，搞这么多注册在manifest做什么用？仔细分类：就两类一类是Activity，一类是Dialog，我们知道Dialog是建立在Activity之上的，如果Activity被finish或destory后，就会报出异常：android.view.WindowManager$BadTokenException: Unable to add window — token android.os.BinderProxy@438e7108 is not valid; is your activity running?

附[《插件占坑，四大组件动态注册前奏（一） 系统Activity的启动流程》](http://blog.csdn.net/hejjunlin/article/details/52190050)（也可点击链接过去看详细过程）的activity的启动时序列图：

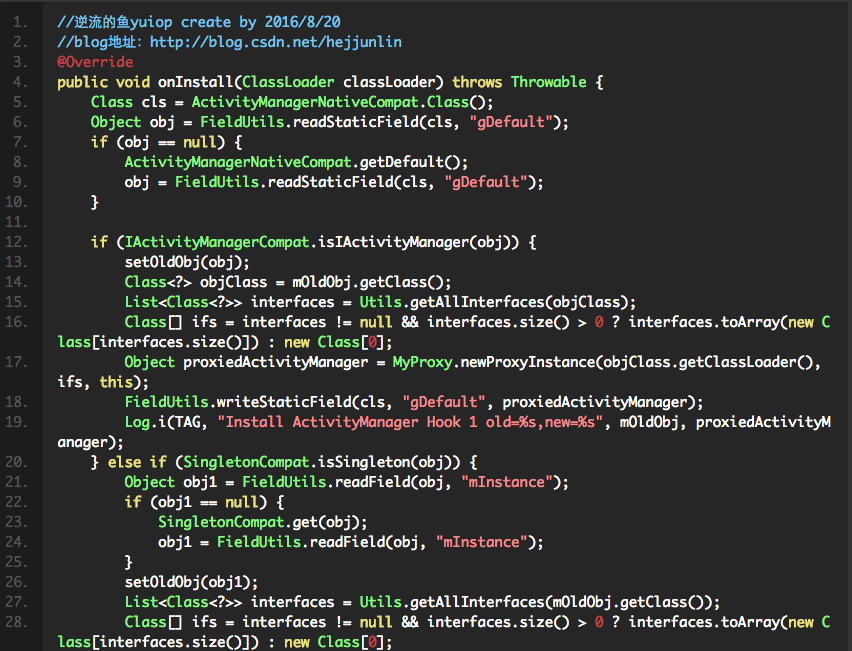
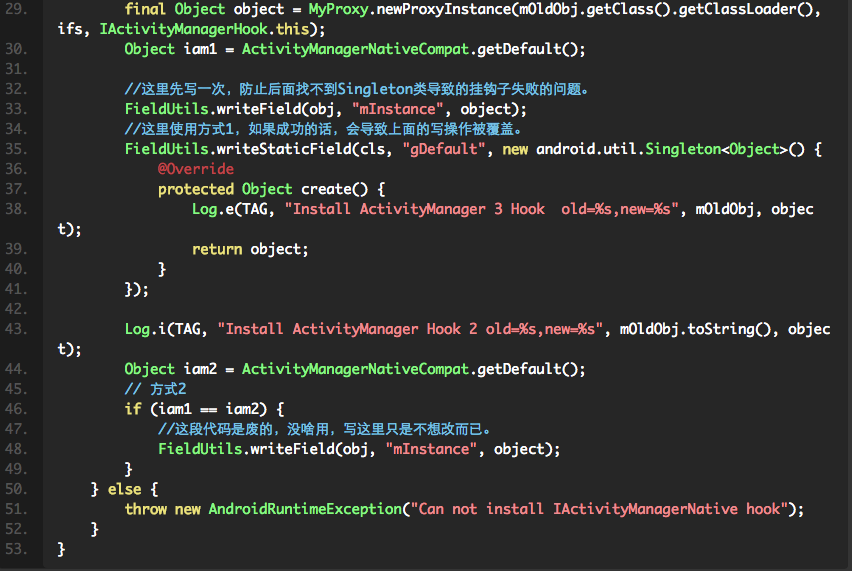




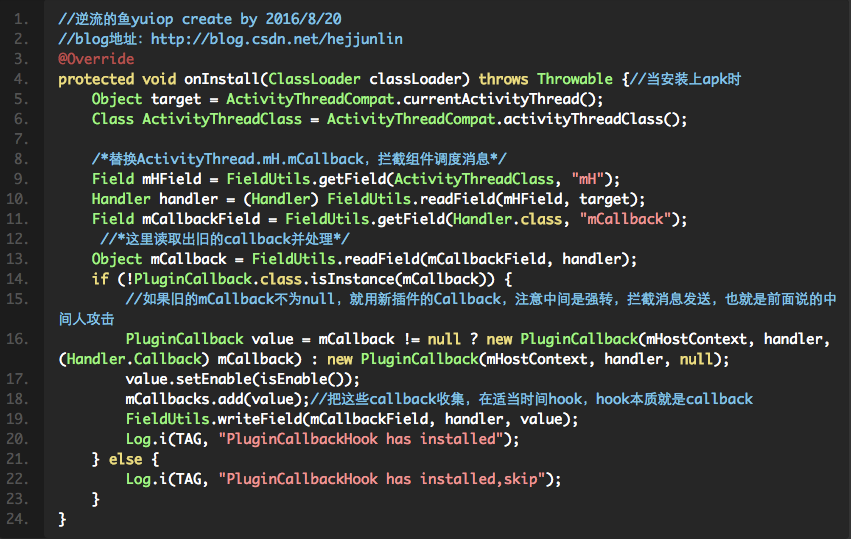
## Activity中关键方法被hook时机

其中startActivity()和handleLanchActivity()是被DroidPlugin 要欺骗系统的两个主要方法：

第一个方法是最被经常使用的startActivity()，hook机制见[《插件开发之360 DroidPlugin源码分析（二）Hook机制》](http://blog.csdn.net/hejjunlin/article/details/52124397)中分析，主要是通过Java的反射机制替换掉IActivityManager全局对象，具体IActivityManagerHookHandle的onInstall()方法如下：

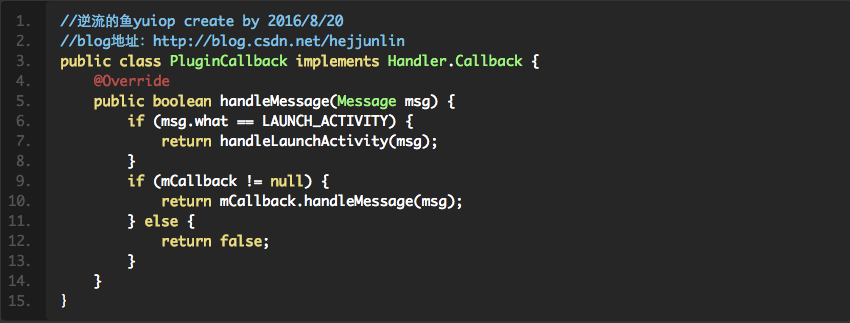
   


第二个方法是handleLaunchActivity()，这个方法属于ActivityThread的一个叫做H的内部类，前面讲Activity启动时，已埋下伏笔，可以参考[《插件占坑，四大组件动态注册前奏（一） 系统Activity的启动流程》](http://blog.csdn.net/hejjunlin/article/details/52190050)，如果学过中间人攻击协议的话，我们知道，现个通信双方发出消息后，进行消息验证，如果中间人拦截相关协议内容，通过一个代理进行转发出去，从而达到欺骗的目的，这里暂且理解handleLanchActivity被中间人攻击了，当启动handleLanchActivity时，被DroidPlugin hook后，那多人可能会想，你怎么hook住hanleLanchActivity呢？别忘了，我们可是在Manifest占了一堆坑的。要是不好理解，可以参看《插件前奏-android黑科技 hook介绍 ：<http://blog.csdn.net/hejjunlin/article/details/52091833>》，可能更直观些，哪在DroidPlugin中，如何在代码中瞒天过海的呢？   
我们可以看下：PluginCallbackHook.java，其中有一个onInstall()方法：



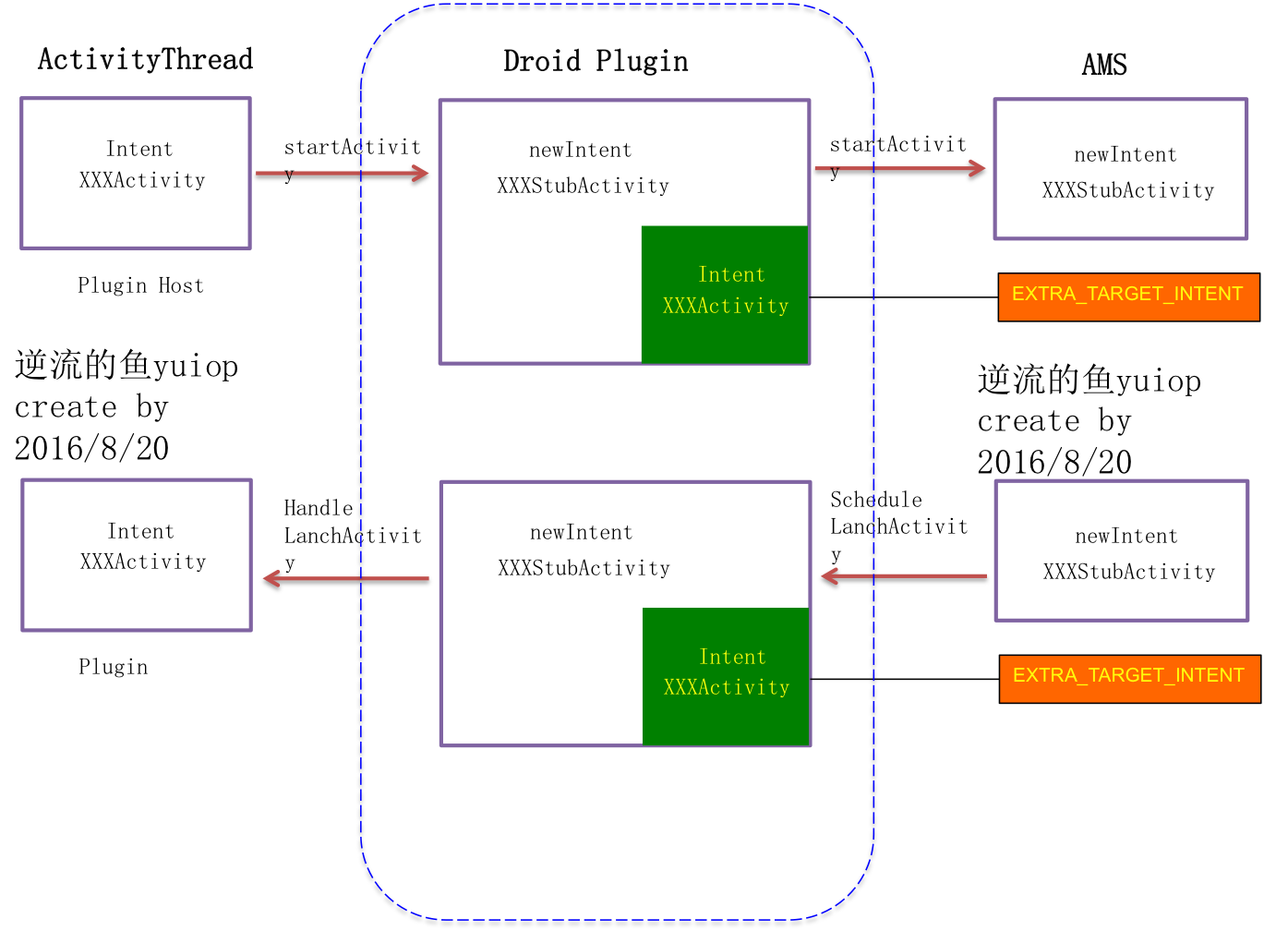
如上代码也有注释，可总结为：   
1.DroidPlugin不是完全攻击mH这个内部类，而是把mH的mCallback成员变量攻击了，然后替换成了一个PluginCallback对象，进行消息分发，那么就可以在在PluginCallback的handleMessage()里任意拦截想要分发的消息，来欺骗AMS。经过中间人这么一闹腾，那就能达到控制这个区域的目的了。

最狠的看下面：   
上面代码中，有一个mH的mCallback,是被拦截攻击了吧，既然被拦截后又要冒充一个担当mH的mCallback职责相关，（这里暂且夸张的说）那肯定得扒了它身上的某些特性放到冒充的mCallback吧，如身份验证相关之类，所以那原来的mCallback丢了它的身份相关，不就是废了，暂且理解为mCallback为null,实际上有mCallback在未赋值之前，初始化时，本身也是null。然后看下PluginCallback的handleMessage()方法：（ps: PluginCallback简直就是仿照ActivityThread中的H内部类写的，几乎逻辑一样，具体可看系统源码证实）

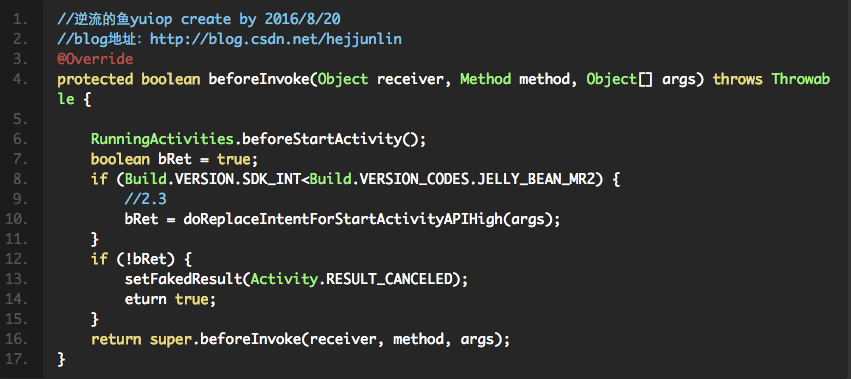


这里只拦截了一个消息类型为LAUNCH\_ACTIVITY（ActivityThread中的H内部类中会有很多消息类型，也包含LAUNCH\_ACTIVITY）。mCallback不为null，就是发一些伪装的消息（因为我们刚拦截了mCallback），如果为null,返回false，这样就会会再通过mH调用回真正ActivityThread的handleLaunchActivity()。（ps:系统发的消息没什么卵用时，我们就不去拦截，暂且理解为这样）

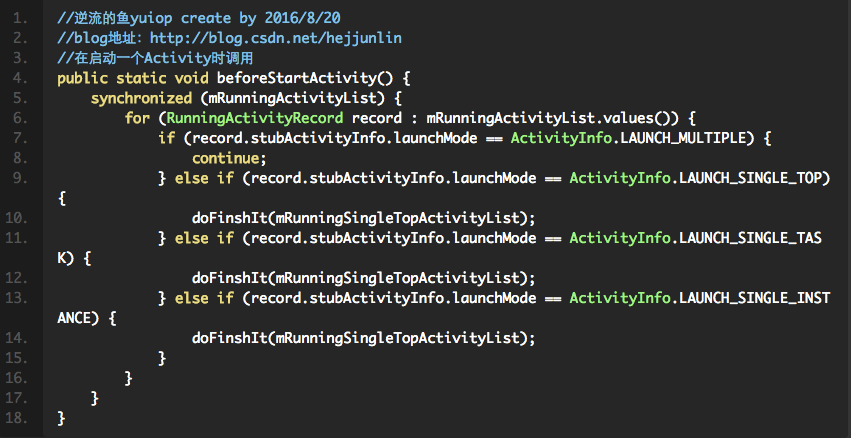
以上就是两个方法怎么被hook相关分析，也可参考在Hook机制举例为IPackageManager的验证签名被hook的过程，接下来就是要看hook中做了什么事？附一张Activity预注册占坑整体流程图：



前面我们都是在说DroidPlugin怎么用中间人攻击方式，拦截了startActivity()方法和hanleLanchActivity()方法，接下来要看下拦截了后，怎么把这种身份角色搞变化了（就是一个怎么扮演冒充的角色过程）：   
1.首先得找一个和真实的人像的角色，这时启用的我们备用人员，也就是事先占的坑stub.ActivityStubxxx。   
2.真实的角色身上有某些特定的特性，如爱抽烟，头发乱糟糟的，像Activity中Intent作为一个extra传到另一个Activity时，还有一些自己的lanchmode(启动方式)，theme(主题)，这些都是Activity的一些特性，所以stub.ActivityStubxxx也得有这些，否则怎么能达到冒充呢，这些我们在代码中早就写好了，所有的那些注册的26个备用人员都是继承ActivityStub，而ActivityStub是直接继承Activity，那还说啥呢，不就是天然的冒充么？说这么多了，直接看代码IActivityManagerHookHandle$startActvity：   
这是一个静态内部类，继承ReplaceCallingPackageHookedMethodHandler，重写了beforeInvoke方法，暂且理解为在冒充之前的一些准备工作，如下：



接着再看beforeStartActivity方法：



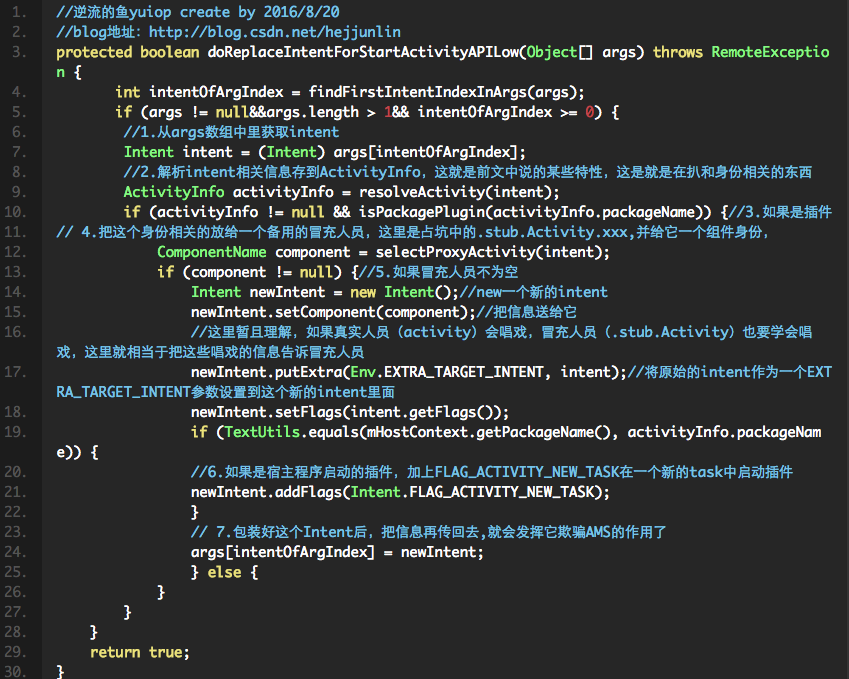
就是判断了下Activity的启动模式，接着都调入doFinshIt(mRunningXXXActivityList),继续看此方法：



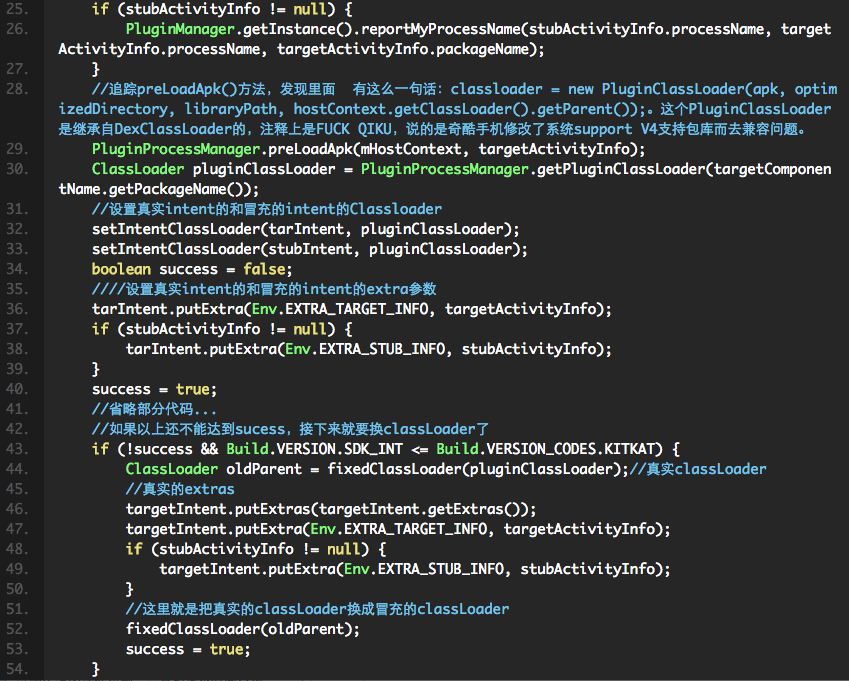
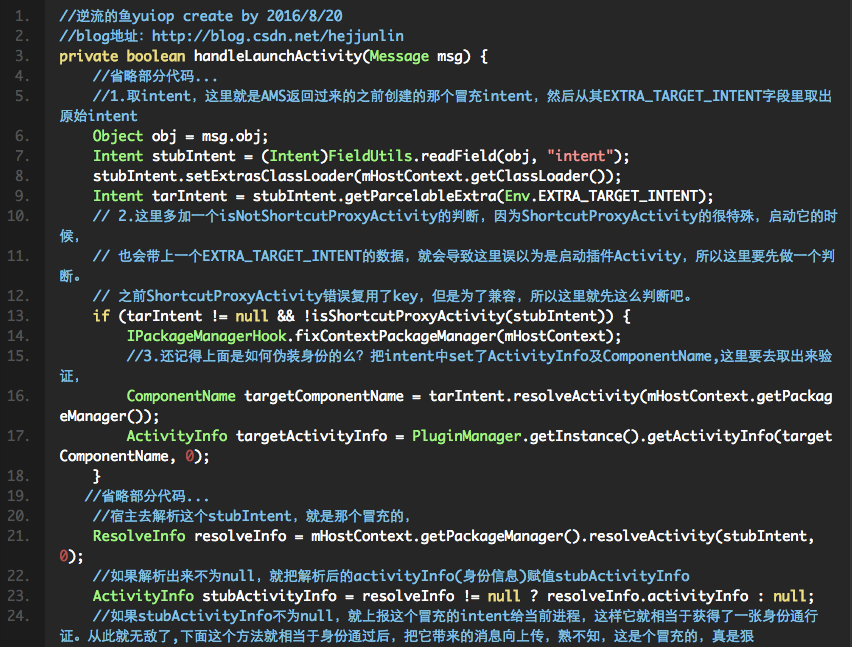
STUB\_NO\_ACTIVITY\_MAX\_NUM为4，上面方法总结为：runningActivityList（备用activity的list的总数）大于等于3时，就把最早进入activityRecord栈中的那个finish掉，因为坑就那么多，要是都占着不干活，那要它干嘛？

## 瞒天过海，冒充真实身份，欺骗AMS

前面这些，还是小打小闹，接下来看一个核心方法，就是beforeInvoke中的doReplaceIntentForStartActivityAPILow方法：



前面我们说了，欺骗了两个方法，上面说的都是startActivity（），接下来看另个一个方法handleLanchActivity(), 这个方法在哪呢？我们前面说过有一个仿照ActivityThread中H内部类的class叫PluginCallBack,对，handleLaunchActivity就是在接到handleMessage中消息类型为LANCH\_ACTIVITY时调用的：



以上就是Activity预注册占坑，并欺骗AMS的过程，下篇分析Service预注册占坑。