网上已经有很多360的脱壳教程了，看了好多后就想自己动手调试下。感谢各位大神无私分享的文章，本菜才能顺利过掉反调。

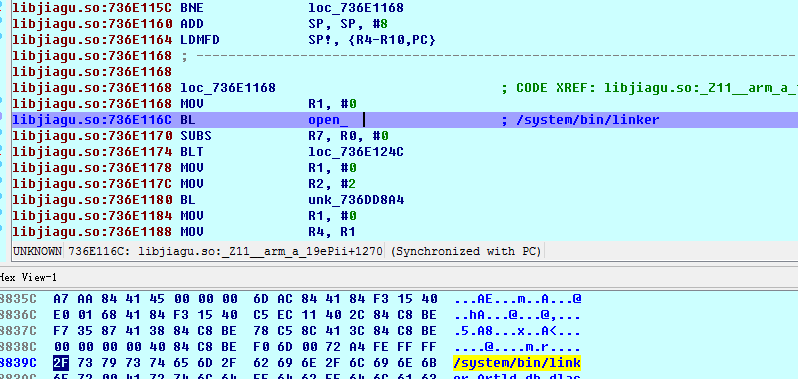
本人是初涉逆向，水平有限，秉着学习的目的，发帖分享自己的逆向学习过程，肯定会有些错误或分析不正确的地方，欢迎大牛指正，赐教，本菜鸟不胜感激。只是可以dump出dex，还无法进行还原、安装运行。onCreate函数被动态注册为native，可以在so中找到注册的地址，可以结合静态和动态调试，有时间的话会继续研究下去。

**第一  过反调试**

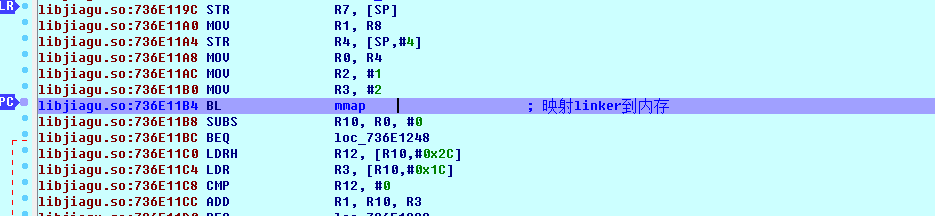
         主要在JNI\_Onload函数的case 29，case33上下断点

1、  第一处反调试，应用调试的特性

参考文档http://www.evil0x.com/posts/26301.html



2、映射linker



3、查看并修改符号rtld\_db\_dlactivity的值

         0x2B = open("/system/bin/linker");第一处反调试

         length = lseek(0x2B,0,SEEK\_END);

         lseek(0x2B,0,SEEK\_SET);

         mmap(NULL,0,1,2,0x2B,0)

         在程序头表中查找第一个PT\_LOAD段的p\_vaddr

         找到PT\_DYNAMIC段 = r3 = 0x0000FED4

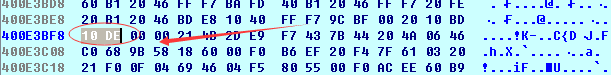
         r1 = 基址+0x0000FED4

         在动态段查找DT\_STRTAB 偏移为0x00000258 ，计算内存偏移                                DT\_SYMTAB 偏移为0x00000168 ，计算内存偏移

         找到符号 rtld\_db\_dlactivity ：基址+Elf32\_Sym->st\_value



查看0x400E3BF8内存地址的指令



将DE 10修改为nop指令 C0 46===>过掉第一处反调试

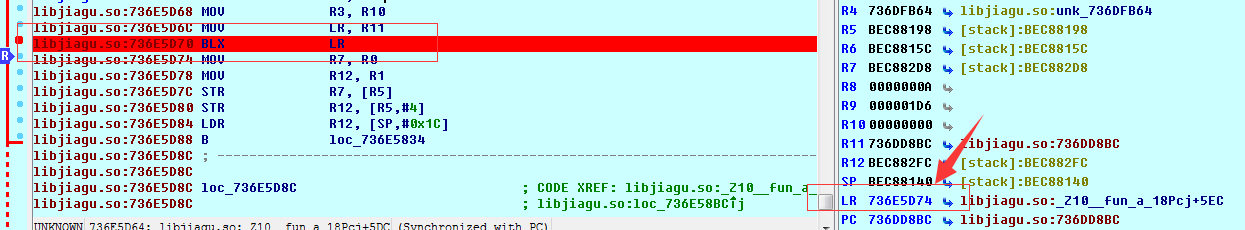
4、第二处反调，此处LR的值为strtol，将其返回值修改为0，过掉第二处反调。

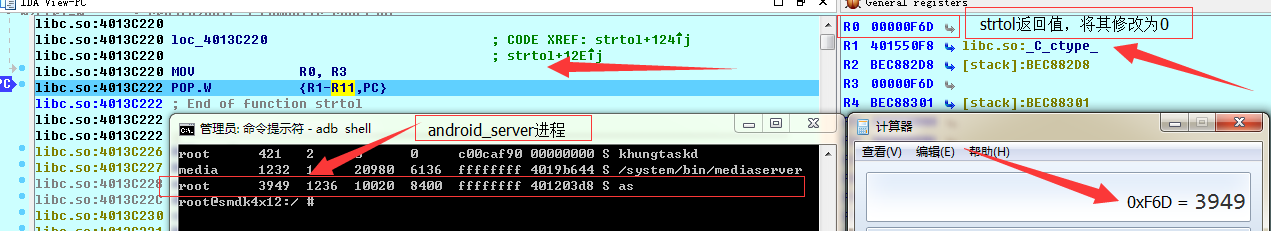
         0x2E = open("/proc/self/status")；                           ;

         strstr(read(),"TracePid");

         strtol();

         close(fd)





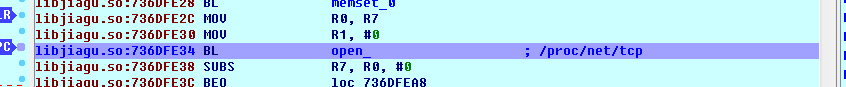
5、第三处反调

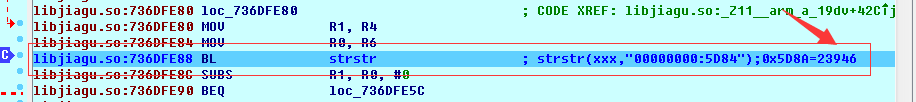
    fd =open("/proc/net/tcp");第三处反调试

    read();

    strstr(xxx,"00000000:5D84");0x5D8A=23946

   调试的时候可以不用远程端口号，或者修改内存中的值过掉反调试





6、第四处反调

       1）、文件监控线程， nop掉

                  将736E0DE8地址nop，过掉文件监控线程

                  pthread\_create ===>sub\_4B50 //第四处反调试

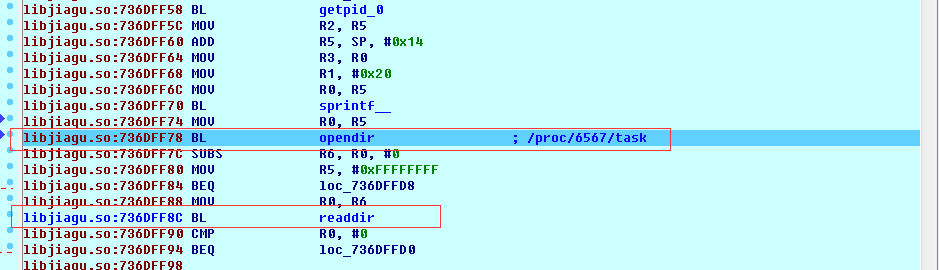
                  {

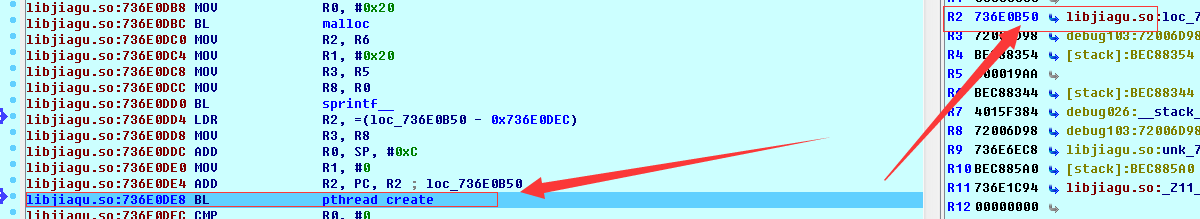
                            inotify\_init

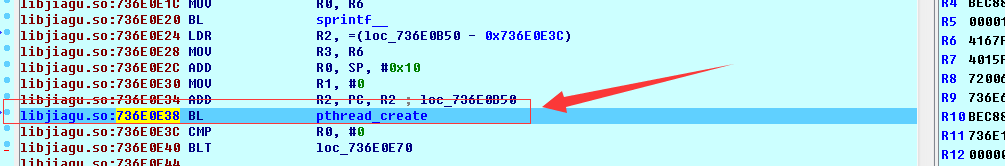
                            inotify\_add\_watch

                            select

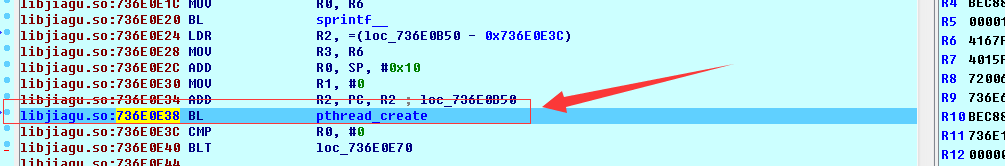
                  }



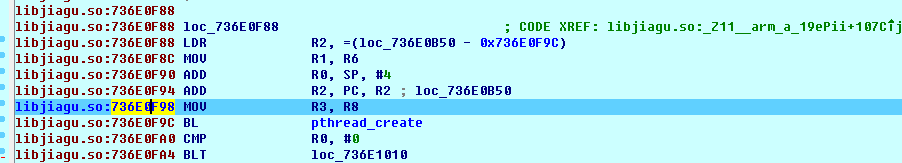
2）、文件监控线程， nop掉



 3）、文件监控线程， nop掉



4）、文件监控线程， nop掉



     总共有四处启动文件监控线程的点。

8. 第五处反调

360会time()函数，通过比较差值，判断程序是否被调试，简单点，可是在time()函数上下断点，让其返回相同的，固定的值。

刚开始调试可能会花费大量的时间在反调试上面，等熟悉了各个反调试点后，可以进行注入，hook这些函数，从而过掉反调试，会省去大量无畏的时间。

至此反调试全部去掉，后边就没有反调试了，进程开始原始dex的解密加载操作。

**第二  dex解密、加载**

1 、调试过程中，通过查看proc/self/maps中可以看出，反调试全部在libjiagu.so映射的段中，Dex的解密加载在debug\_xxx段中，将这款内存整个dump下来，可以发现为elf文件，Elf头被抹掉，与section相关的信息被修改为无效的或抹掉，防止静态分析，使用ThomasKing大神的so修复工具，ida就能正常打开，静态分析了，

    2、fopen(“/proc/self/maps”);进行匹配查找：

/dev/ashmem/dalvik-classes.dex

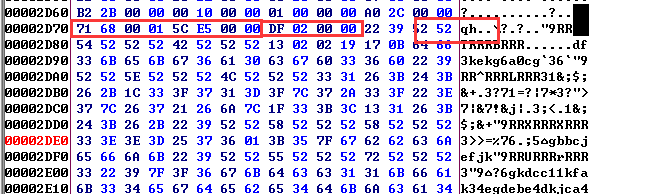
         apk@classes.dex

         .odex

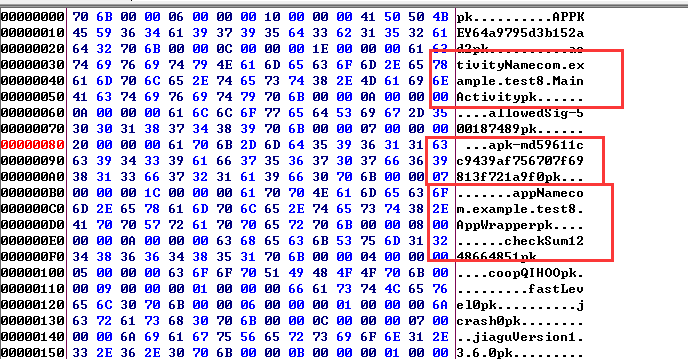
         分割出内存基地址，判断头是否为”dey”;

         根据pDexHeader->dataSize+ pDexHeader->dataSize找到qh开头的内存地址，

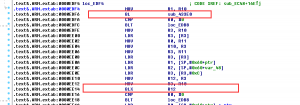
         0x2DF为数据块的长度，其中存储了，加固app的相关信息，包括，包名，Application、activity等和自身加载的一些信息。



解密后的文件



2、  解密dex数据



loc\_EDF4(){

                                     BL sub\_433E0

{

0x433FA j\_j\_tl\_mmap -->分配dex空间， 长度 0x3274

}

                                BLX  R12; --> //解密dex数据到前面分配的空间中//偏移0xEE14

                                     {

                                               blx sub\_36A88()                          ----------->0x353A4

                                               {

                                                        loc\_36ADA(){

                                                                 sub\_365A4(){

                                                                           sub\_358F0(){

                                                                                    loc\_36156 解密操作

                                                                           }

                                                                 }

                                                        }

                                               }

                                     }

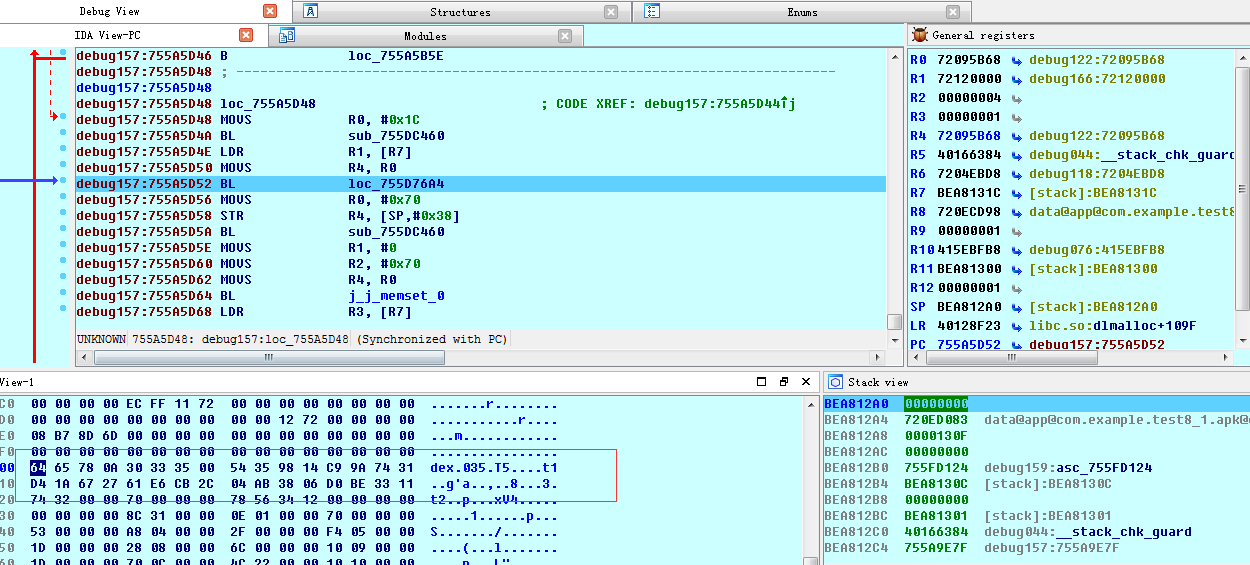
                            此时的dex头，0x70个字节还是加密的

                            sub\_46F6C（）

{

                                     //解密dex头，0x70个字节，

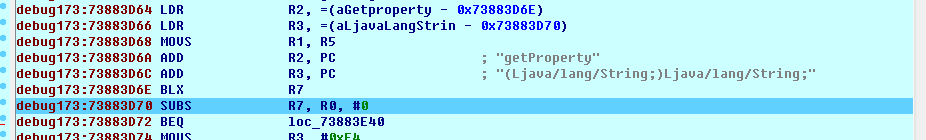
                            }



------------------------------------------------至此dex完全解密到内存中-------------------------------------------

                                               接下来自己构建，构建DexFile结构

进行系统版本属性的判断



getSystemContext();

getPackageInfo();

getPackageInfo(包名)

RegisterNatives(DexFile\_jcl,);                                                        {"getClassNameList","(I)[Ljava/lang/String;",0xD791}

RegisterNatives(StubApp2379654022);

                  {

{"interface9","()Ljava/lang/String;",0x73921d79},

                  {"getAppkey","()Ljava/lang/String;",0x73901ad1},

                  {"mark","()V",,0x739185dd},

              {"interface6","(Ljava/lang/String;)Ljava/lang/String;",0x738fde05},

              {"interface7","(Landroid/app/Application;Landroid/content/Context;)Z",0x73901e99},

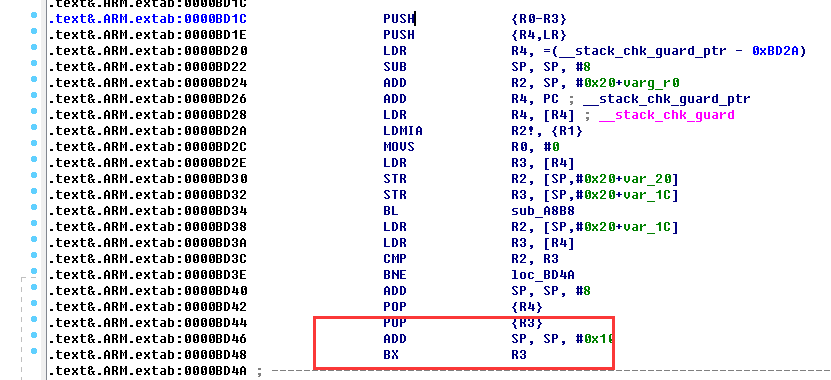
。。。。。。。。。。。。。。

}

RegisterNatives("onCreate","(Landroid/os/Bundle;)V",0x0000BD1D);

可见onCreate可定是注册为native函数的，在so的偏移0x0000BD1D处，

可以动态调试跟进去看看



在so偏移0xA9B8处为解释执行switch 开始的地方，猜测应该是逐条指令进行解释执行的



附件会上传脱壳过程中用到的app和记录的文档，dump的so和dex，供大家学习，相互交流，共同学习。