# 关于apk加壳之动态加载dex文件

转载

JackxinXu2100 于 2017-12-10 08:41:32 发布 1128 收藏 2
分类专栏： 移动应用研发(iPAD/Android) 汇编以及逆向工程


移动应用研发(iPAD/Android) 同时被 2 个专栏收录
18 篇文章 0 订阅
订阅专栏


汇编以及逆向工程
16 篇文章 0 订阅
订阅专栏

由于自己之前做了一个关于手机令牌的APK软件，在实现的过程中尽管使用了native so进行一定的逻辑算法保护，但是在自己逆向破解的过程中发现我的手机令牌关键数据能够“轻易地”暴露出来，所以我就想进一步的对其进行加固。于是，我使用的网上常用的梆梆加固、爱加密和阿里的聚安全应用来对我的apk进行一个加固保护。加固后，出于好奇心，我想对这些加固的原理进行一个了解，便于我自己能够实现这个加固的方法。于是开始了网上关于这方面的学习，我将这些加固的大致原理进行了一个总结，发现它们实现的最主要的方法就是利用了dex文件动态加载，将主逻辑的dex文件经过加密隐藏在壳程序的dex中，并在运行时通过so进行解密，并从内存读取dex数据，直接在native层进行一个动态加载。这样的实现有几个关键点：

1. dex文件不存储在设备的物理存储区域而是将文件的数据加密存储在壳程序的dex数据区域（关于dex的结构就在此不再解释）；
2. 从内存中获取dex数据，动态加载到进程空间中；
3. 壳程序的application重定向加载到原程序的application对象；

下面我就对这几个问题进行一一的学习之旅。

关于第一个问题，其实经历了我很长时间的学习，主要是我在最开始学习的过程中，一直在dex的动态加载上面打转，由于关于dex的加载问题主要涉及到一个

`DexClassLoder(String dexPath, String optimizedDirectory,String libraryPath ,ClassLoader parent)`方法，所以我必须得有个dex的路径方法啊，这点让我真的很抓狂，所以只能硬着头皮写咯。

于是在http://blog.csdn.net/androidsecurity/article/details/8809542的帮助下完成了壳程序加载dex数据的方法。

DexClassLoder-> BaseDexClassLoader->DexPathList->makeDexElements-> loadDexFile-> loadDex->DexFile(String fileName)

```
1 package com.unshell.test;
2
3 import android.app.Application;
4 import java.io.BufferedInputStream;
5 import java.io.ByteArrayInputStream;
6 import java.io.ByteArrayOutputStream;
7 import java.io.DataInputStream;
8 import java.io.File;
```

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.lang.ref.WeakReference;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.zip.ZipEntry;
import java.util.zip.ZipInputStream;

import dalvik.system.DexClassLoader;
import android.app.Instrumentation;
import android.content.Context;
import android.content.pm.ApplicationInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.os.Bundle;

public class ProxyApplication extends Application{
    private static final String appkey = "APPLICATION_CLASS_NAME";
    private String apkFileName;
    private String odexPath;
    private String libPath;

    //这是context 赋值
    @Override
    protected void attachBaseContext(Context base) {
        super.attachBaseContext(base);
        try {
            //创建两个文件夹payload_odex，payload_lib 私有的，可写的文件目录
            File odex = this.getDir("payload_odex", MODE_PRIVATE);
            File libs = this.getDir("payload_lib", MODE_PRIVATE);
            odexPath = odex.getAbsolutePath();
            libPath = libs.getAbsolutePath();
            apkFileName = odex.getAbsolutePath() + "/payload.apk";
            File dexFile = new File(apkFileName);
            if (!dexFile.exists())
            {
                dexFile.createNewFile();  //在payload_odex文件夹内，创建payload.apk
                // 读取程序classes.dex文件
                byte[] dexdata = this.readDexFileFromApk();
                // 分离出解壳后的apk文件已用于动态加载
                this.splitPayLoadFromDex(dexdata);
            }
            // 配置动态加载环境
            Object currentActivityThread = RefInvoke.invokeStaticMethod(
                    "android.app.ActivityThread", "currentActivityThread",
                    new Class[] {}, new Object[] {});//获取主线程对象
http://blog.csdn.net/myarrow/article/details/14223493
            String packageName = this.getPackageName();//当前apk的包名
            //下面两句不是太理解
            HashMap mPackages = (HashMap) RefInvoke.getFieldOjbect(
                    "android.app.ActivityThread", currentActivityThread,
                    "mPackages");
            WeakReference wr = (WeakReference) mPackages.get(packageName);
            //创建被加壳apk的DexClassLoader对象    加载apk内的类和本地代码（c/c++代码）
            DexClassLoader dLoader = new DexClassLoader(apkFileName, odexPath,
                    libPath, (ClassLoader) RefInvoke.getFieldOjbect(
                            "android.app.LoadedApk", wr.get(), "mClassLoader"));
```

```
67              //base.getClassLoader(); 是不是就等同于 (ClassLoader) RefInvoke.getFieldOjbect()? 有空验证下//?
68              //把当前进程的DexClassLoader 设置成了被加壳apk的DexClassLoader  ----有点c++中进程环境的意思~~
69              RefInvoke.setFieldOjbect("android.app.LoadedApk", "mClassLoader",
70                      wr.get(), dLoader);
71
72
73          } catch (Exception e) {
74              // TODO Auto-generated catch block
75              e.printStackTrace();
76          }
77      }
78
79      @Override
80      public void onCreate() {
81          {
82              // 如果源应用配置有Appliction对象，则替换为源应用Applicaiton，以便不影响源程序逻辑。
83              String appClassName = null;
84              //获取xml文件里配置的被加壳apk的Applicaiton
85              try {
86                  ApplicationInfo ai = this.getPackageManager()
87                          .getApplicationInfo(this.getPackageName(),
88                                  PackageManager.GET_META_DATA);
89                  Bundle bundle = ai.metaData;
90                  if (bundle != null
91                          && bundle.containsKey("APPLICATION_CLASS_NAME")) {
92                      appClassName = bundle.getString("APPLICATION_CLASS_NAME");//className 是配置在xml文件
中的。
93                  } else {
94                      return;
95                  }
96              } catch (NameNotFoundException e) {
97                  // TODO Auto-generated catch block
98                  e.printStackTrace();
99              }
100             //有值的话调用该Applicaiton
101             Object currentActivityThread = RefInvoke.invokeStaticMethod(
102                     "android.app.ActivityThread", "currentActivityThread",
103                     new Class[] {}, new Object[] {});
104             Object mBoundApplication = RefInvoke.getFieldOjbect(
105                     "android.app.ActivityThread", currentActivityThread,
106                     "mBoundApplication");
107             Object loadedApkInfo = RefInvoke.getFieldOjbect(
108                     "android.app.ActivityThread$AppBindData",
109                     mBoundApplication, "info");
110             //把当前进程的mApplication 设置成了null
111             RefInvoke.setFieldOjbect("android.app.LoadedApk", "mApplication",
112                     loadedApkInfo, null);
113             Object oldApplication = RefInvoke.getFieldOjbect(
114                     "android.app.ActivityThread", currentActivityThread,
115                     "mInitialApplication");
117             ArrayList<Application> mAllApplications = (ArrayList<Application>) RefInvoke
118                     .getFieldOjbect("android.app.ActivityThread",
119                             currentActivityThread, "mAllApplications");
120             mAllApplications.remove(oldApplication);//删除oldApplication
121
122             ApplicationInfo appinfo_In_LoadedApk = (ApplicationInfo) RefInvoke
123                     .getFieldOjbect("android.app.LoadedApk", loadedApkInfo,
124                             "mApplicationInfo");
125             ApplicationInfo appinfo_In_AppBindData = (ApplicationInfo) RefInvoke
126                     .getFieldOjbect("android.app.ActivityThread$AppBindData",
```

```
                        mBoundApplication, "appInfo");
            appinfo_In_LoadedApk.className = appClassName;
            appinfo_In_AppBindData.className = appClassName;
            Application app = (Application) RefInvoke.invokeMethod(
                    "android.app.LoadedApk", "makeApplication", loadedApkInfo,
                    new Class[] { boolean.class, Instrumentation.class },
                    new Object[] { false, null });//执行 makeApplication（false,null）
            RefInvoke.setFieldOjbect("android.app.ActivityThread",
                    "mInitialApplication", currentActivityThread, app);


            HashMap mProviderMap = (HashMap) RefInvoke.getFieldOjbect(
                    "android.app.ActivityThread", currentActivityThread,
                    "mProviderMap");
            Iterator it = mProviderMap.values().iterator();
            while (it.hasNext()) {
                Object providerClientRecord = it.next();
                Object localProvider = RefInvoke.getFieldOjbect(
                        "android.app.ActivityThread$ProviderClientRecord",
                        providerClientRecord, "mLocalProvider");
                RefInvoke.setFieldOjbect("android.content.ContentProvider",
                        "mContext", localProvider, app);
            }
            app.onCreate();
        }
    }

    /**
     * 释放被加壳的apk文件，so文件
     * @param data
     * @throws IOException
     */
    private void splitPayLoadFromDex(byte[] data) throws IOException {

        int ablen = apkdata.length;
        //取被加壳apk的长度   这里的长度取值，对应加壳时长度的赋值都可以做些简化
        byte[] dexlen = new byte[4];
        System.arraycopy(apkdata, ablen - 4, dexlen, 0, 4);
        ByteArrayInputStream bais = new ByteArrayInputStream(dexlen);
        DataInputStream in = new DataInputStream(bais);
        int readInt = in.readInt();
        System.out.println(Integer.toHexString(readInt));
        byte[] newdex = new byte[readInt];
        //把被加壳apk内容拷贝到newdex中
        System.arraycopy(apkdata, ablen - 4 - readInt, newdex, 0, readInt);
        //这里应该加上对于apk的解密操作，若加壳是加密处理的话
        //?byte[] apkdata = decrypt(newdex);  //解壳程序的dex并没有加密，所以也不需要解密
        //写入apk文件
        File file = new File(apkFileName);
        try {
            FileOutputStream localFileOutputStream = new FileOutputStream(file);
            localFileOutputStream.write(newdex);
            localFileOutputStream.close();


        } catch (IOException localIOException) {
            throw new RuntimeException(localIOException);
        }

```

```
        //分析被加壳的apk文件
        ZipInputStream localZipInputStream = new ZipInputStream(
                new BufferedInputStream(new FileInputStream(file)));
        while (true) {
            ZipEntry localZipEntry = localZipInputStream.getNextEntry();//不了解这个是否也遍历子目录，看样子应该是遍历的
            if (localZipEntry == null) {
                localZipInputStream.close();
                break;
            }
            //取出被加壳apk用到的so文件，放到 libPath中（data/data/包名/payload_lib)
            String name = localZipEntry.getName();
            if (name.startsWith("lib/") && name.endsWith(".so")) {
                File storeFile = new File(libPath + "/"
                        + name.substring(name.lastIndexOf('/')));
                storeFile.createNewFile();
                FileOutputStream fos = new FileOutputStream(storeFile);
                byte[] arrayOfByte = new byte[1024];
                while (true) {
                    int i = localZipInputStream.read(arrayOfByte);
                    if (i == -1)
                        break;
                    fos.write(arrayOfByte, 0, i);
                }
                fos.flush();
                fos.close();
            }
            localZipInputStream.closeEntry();
        }
        localZipInputStream.close();


    }

    /**
     * 从apk包里面获取dex文件内容（byte）
     * @return
     * @throws IOException
     */
    private byte[] readDexFileFromApk() throws IOException {
        ByteArrayOutputStream dexByteArrayOutputStream = new ByteArrayOutputStream();
        ZipInputStream localZipInputStream = new ZipInputStream(
                new BufferedInputStream(new FileInputStream(
                        this.getApplicationInfo().sourceDir)));
        while (true) {
            ZipEntry localZipEntry = localZipInputStream.getNextEntry();
            if (localZipEntry == null) {
                localZipInputStream.close();
                break;
            }
            if (localZipEntry.getName().equals("classes.dex")) {
                byte[] arrayOfByte = new byte[1024];
                while (true) {
                    int i = localZipInputStream.read(arrayOfByte);
                    if (i == -1)
                        break;
                    dexByteArrayOutputStream.write(arrayOfByte, 0, i);
                }
            }
            localZipInputStream.closeEntry();
```

```
        }
        localZipInputStream.close();
        return dexByteArrayOutputStream.toByteArray();
    }


    // //直接返回数据，读者可以添加自己解密方法
    private byte[] decrypt(byte[] data) {
        return data;
    }
}
```

接着我们就要解决从内存dex的动态加载问题，于是根据我在看雪论坛上面学习的这几篇文章

http://blog.csdn.net/androidsecurity/article/details/9674251

http://www.kanxue.com/bbs/showthread.php?t=195865

实现了dex的读取数据

Jni关键代码基本都在译文博客中了，我们要做的是让它通过编译、得到so库。本地代码当然要有与之对应的java代码去加载才能用，通过上面对因为的总结，可以先这样定义　　本地方法：

static native int loadDex(byte[] dex,long dexlen);

生成好对应的.h、.c文件之后把译文中给出的核心代码填上，下面才是难题，许多类型都是unknown的，ndk编译器会告诉你它不认识这些乱七八糟的玩意儿。接下来就是挨个补充定义了。

看着u4、u1这些从java程序猿眼中怪怪的类型我不禁长出一口气——幸亏当年是C出身的。溯本清源，在源码 /dalvik/vm/Common.h 类中找到了这群货的宏定义，于是照葫芦画瓢，在jni目录里弄了一个伪造版的Common.h，搜刮了一下所有需要定义的类型之后，这个文件基本上是这个样子的：

```
#ifndef DALVIK_COMMON_H_
#define DALVIK_COMMON_H_

#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <assert.h>

static union { char c[4]; unsigned long mylong; }endian_test = {{ 'l', '?', '?', 'b' } };
#define ENDIANNESS  ((char)endian_test.mylong)

//#if ENDIANNESS == "l"
#define HAVE_LITTLE_ENDIAN
//#else
//#define HAVE_BIG_ENDIAN
//#endif

#if defined(HAVE_ENDIAN_H)
# include <endian.h>
#else /*not HAVE_ENDIAN_H*/
# define __BIG_ENDIAN 4321
# define __LITTLE_ENDIAN 1234
# if defined(HAVE_LITTLE_ENDIAN)
#  define __BYTE_ORDER __LITTLE_ENDIAN
```

```
25 # else
26 #  define __BYTE_ORDER __BIG_ENDIAN
27 # endif
28 #endif /*not HAVE_ENDIAN_H*/
29
30 #if !defined(NDEBUG) && defined(WITH_DALVIK_ASSERT)
31 # undef assert
32 # define assert(x) \
33 ((x) ? ((void)0) : (ALOGE("ASSERT FAILED (%s:%d): %s", \
34 __FILE__, __LINE__, #x), *(int*)39=39, (void)0) )
35 #endif
36
37 #define MIN(x,y) (((x) < (y)) ? (x) : (y))
38 #define MAX(x,y) (((x) > (y)) ? (x) : (y))
39
40 #define LIKELY(exp) (__builtin_expect((exp) != 0, true))
41 #define UNLIKELY(exp) (__builtin_expect((exp) != 0, false))
42
43 #define ALIGN_UP(x, n) (((size_t)(x) + (n) - 1) & ~((n) - 1))
44 #define ALIGN_DOWN(x, n) ((size_t)(x) & -(n))
45 #define ALIGN_UP_TO_PAGE_SIZE(p) ALIGN_UP(p, SYSTEM_PAGE_SIZE)
46 #define ALIGN_DOWN_TO_PAGE_SIZE(p) ALIGN_DOWN(p, SYSTEM_PAGE_SIZE)
47
48 #define CLZ(x) __builtin_clz(x)
49
50 /*
51  * If "very verbose" logging is enabled, make it equivalent to ALOGV.
52  * Otherwise, make it disappear.
53  *
54  * Define this above the #include "Dalvik.h" to enable for only a
55  * single file.
56  */
57 /* #define VERY_VERBOSE_LOG */
58 #if defined(VERY_VERBOSE_LOG)
59 # define LOGVV  ALOGV
60 # define IF_LOGVV() IF_ALOGV()
61 #else
62 # define LOGVV(...) ((void)0)
63 # define IF_LOGVV() if (false)
64 #endif
65
66
67 /*
68  * These match the definitions in the VM specification.
69  */
70 typedef uint8_t u1;
71 typedef uint16_tu2;
72 typedef uint32_tu4;
73 typedef uint64_tu8;
74 typedef int8_t  s1;
75 typedef int16_t s2;
76 typedef int32_t s4;
77 typedef int64_t s8;
78
79 /*
80  * Storage for primitive types and object references.
81  *
82  * Some parts of the code (notably object field access) assume that values
83  * are "left aligned", i.e. given "JValue jv", "jv.i" and "*((s4*)&jv)"
84  * yield the same result.  This seems to be guaranteed by gcc on big- and
```

```
 * yield the same result.  This seems to be guaranteed by gcc on big- and
 * little-endian systems.
 */

#define OFFSETOF_MEMBER(t, f) \
  (reinterpret_cast<char*>(   \
 &reinterpret_cast<t*>(16)->f) -  \
   reinterpret_cast<char*>(16))

#define NELEM(x) ((int) (sizeof(x) / sizeof((x)[0])))

union JValue {
#if defined(HAVE_LITTLE_ENDIAN)
    u1  z;
    s1  b;
    u2  c;
    s2  s;
    s4  i;
    s8  j;
    float   f;
    double  d;
    void* l;
#endif
#if defined(HAVE_BIG_ENDIAN)
    struct {
        u1_z[3];
        u1z;
    };
    struct {
        s1_b[3];
        s1b;
    };
    struct {
        u2_c;
        u2c;
    };
    struct {
        s2_s;
        s2s;
    };
    s4  i;
    s8  j;
    float   f;
    double  d;
    void*   l;
#endif
};

/*
 * Array objects have these additional fields.
 *
 * We don't currently store the size of each element.  Usually it's implied
 * by the instruction.  If necessary, the width can be derived from
 * the first char of obj->clazz->descriptor.
 */
typedef struct   {
   void*clazz;
   u4  lock;
   u4  length;
   u1*  contents;
```

```
144 }ArrayObject ;
145
146 #endif  // DALVIK_COMMON_H_
```

这里面还有个大小端的问题，不过为求实验先通过就先定义死，过了再说。

还有个值得一提的结构就是最后面的ArrayObject，这玩意定义在源码的/dalvik/vm/oo/Object.h 中，原本的定义是这样的：

```
1 struct Object {
2     ClassObject*clazz;
3     u4  lock;
4 };
5
6 struct ArrayObject : Object {
7     u4  length;
8     u8  contents[1];
9 };
```

如果还实实在在的去弄一个ClassObject，那就是java中毒已深的表现，根据看雪里面的相关讨论（就是文首提到的两篇），直接如上定义了。得到最后的C代码如下：

```
1 #include "com_android_dexunshell_NativeTool.h"
2 #include <stdlib.h>
3 #include <dlfcn.h>
4 #include <stdio.h>
5
6 JNINativeMethod *dvm_dalvik_system_DexFile;
7 void (*openDexFile)(const u4* args,union  JValue* pResult);
8
9 int lookup(JNINativeMethod *table, const char *name, const char *sig,
10    void (**fnPtrout)(u4 const *, union JValue *))
11 {
12     int i = 0;
13     while (table[i].name != NULL)
14     {
15         LOGI("lookup %d %s" ,i,table[i].name);
16         if ((strcmp(name, table[i].name) == 0)
17                 && (strcmp(sig, table[i].signature) == 0))
18         {
19             *fnPtrout = table[i].fnPtr;
20             return 1;
21            }
22            i++;
23     }
24      return 0;
25 }
26
27 /* This function will be call when the library first be load.
28  * You can do some init in the libray. return which version jni it support.
29  */
```

```
JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
    void *ldvm = (void*) dlopen("libdvm.so", RTLD_LAZY);
    dvm_dalvik_system_DexFile = (JNINativeMethod*) dlsym(ldvm,
           "dvm_dalvik_system_DexFile");
    if(0 == lookup(dvm_dalvik_system_DexFile, "openDexFile", "([B)I",
        &openDexFile))
     {
           openDexFile = NULL;
           LOGE("method does not found ");
    }else
    {
        LOGI("method found ! HAVE_BIG_ENDIAN");
     }
     LOGI("ENDIANNESS is %c" ,ENDIANNESS );
     void *venv;
     LOGI("dufresne----->JNI_OnLoad!");
     if ((*vm)->GetEnv(vm, (void**) &venv, JNI_VERSION_1_4) != JNI_OK)
    {
           LOGE("dufresne--->ERROR: GetEnv failed");
           return -1;
     }
     return JNI_VERSION_1_4;
}

JNIEXPORT jint JNICALL Java_com_android_dexunshell_NativeTool_loadDex(
   JNIEnv * env, jclass jv, jbyteArray dexArray, jlong dexLen)
{
    // header+dex content
    u1 * olddata = (u1*)(*env)-> GetByteArrayElements(env,dexArray,   NULL);
    char* arr;
     arr=(char*)malloc(16+dexLen);
     ArrayObject *ao=(ArrayObject*)arr;
     ao->length=dexLen;
     memcpy(arr+16,olddata,dexLen);
      u4 args[] = { (u4) ao };
    union JValue pResult;
    jint result;
    LOGI("call openDexFile 33..." );
    if(openDexFile != NULL)
    {
        openDexFile(args,&pResult);
    }
    else
    {
        result = -1;
    }

    result = (jint) pResult.l;
    LOGI("Java_com_android_dexunshell_NativeTool_loadDex %d" , result);
    return result;
}
```

Java层

底层代码基本了然，也就是说译文提供的思路基本实现，剩下其他加壳的事儿还要自己动脑筋补上。现在java层我们有一个可以使用的以byte数组为参数的加载dex的接口了：
static native int loadDex(byte[] dex,long dexlen);
要知道我们花这么大力气实现的这个方法，实际意义在于让源程序的dex数据在内存中传递，而不是必须保存在某个地方、以文件的方式。也就是说，我们需要一个新的DexClassLoader，去替换在上一篇提到的基础加壳方案中自定义Application—— ProxyApplication 类，通过反射设置到”android.app.LoadedApk”中mClassLoder属性的那个系统DexClassLoader，即至少那一段应该改成这样：

```
1 DynamicDexClassLoder dLoader = new DynamicDexClassLoder(base,srcdata,
2    libPath, (ClassLoader) RefInvoke.getFieldOjbect(
3  "android.app.LoadedApk", wr.get(), "mClassLoader"),
4  getPackageResourcePath(),getDir(".dex", MODE_PRIVATE).getAbsolutePath() );
5
6 RefInvoke.setFieldOjbect("android.app.LoadedApk", "mClassLoader",  wr.get(), dLoader);
```

没错，DynamicDexClassLoder 它的构造参数中应当去接收源程序的dex数据，以byte数组的形式，这样、相关把dex数组保存为文件那段代码可以删除，/data/data 中相关目录就找不到缓存dex文件的身影了；

替换DexClassLoader，要知道相对于系统版本的加载器我们的少了什么，又多出了什么，在一一对接上，就没问题了。少了什么呢？是dex文件路径、多出了什么呢？是dex byte数组，考虑到已经实现的jni库，那就是多了一个加载好的dex文件对应的cookie值。那么，这个
Cookie 是否能够完成替换呢？这需要到源码中找答案。
源码路径：libcore/dalvik/src/main/java/dalvik/system ，生成类图，取出DexClassLoader相关的一部分：

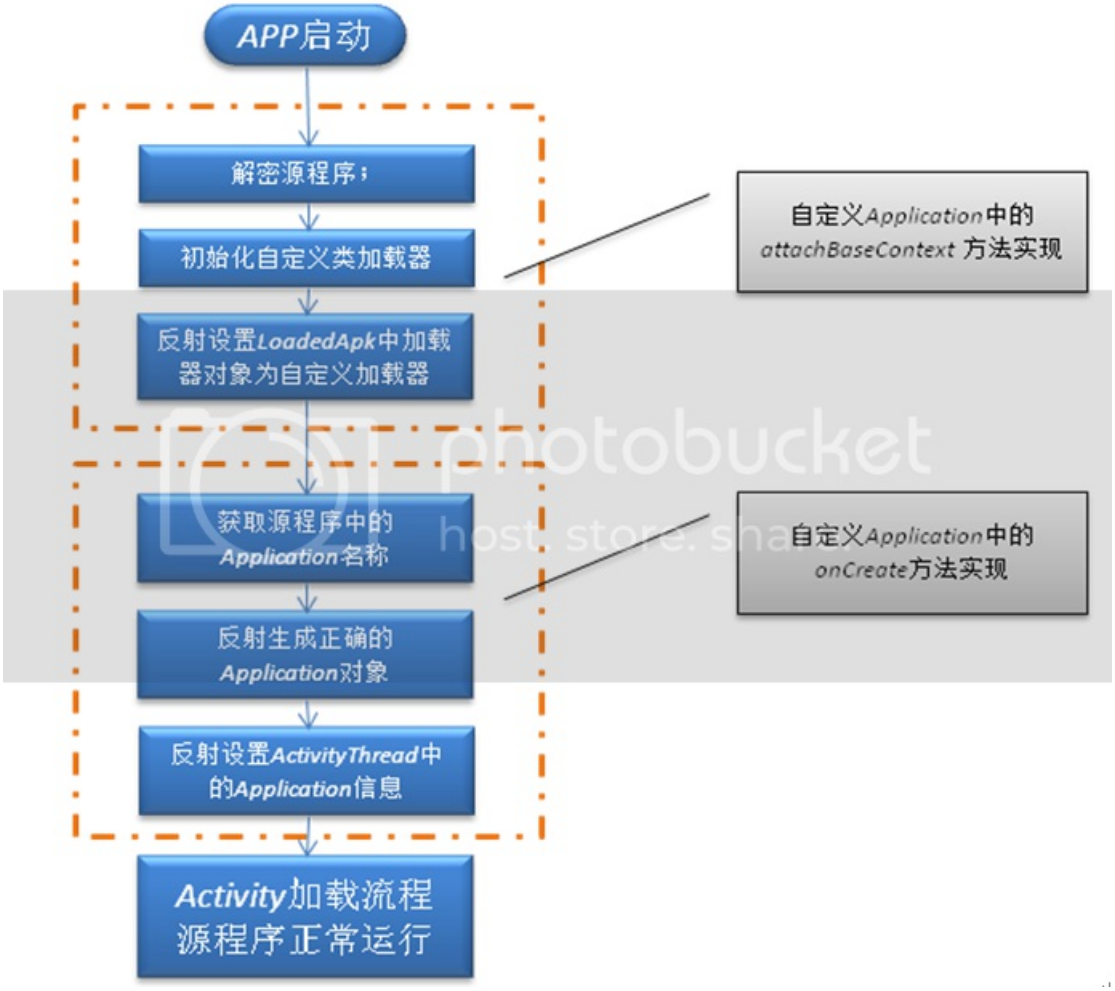
走读几遍代码基本就能了解，对于dex文件加载而言，DynamicDexClassLoder需要做的实际上只有一件事，复写findClass方法，使APK运行时能够找到和加载源程序dex中的类，至于如何实现，从类图上就可以看出，最后实际上追溯到DexFile类，可以利用到jni加载到的cookie，通过反射DexFile中的方法，实现我们的预期，具体实现如下：

```
package com.android.dexunshell;

import java.io.IOException;
import java.net.URL;
import java.util.Enumeration;

import com.eebbk.mingming.k7utils.ReflectUtils;

import android.content.Context;
import android.util.Log;
import android.view.LayoutInflater;

import dalvik.system.DexClassLoader;
import dalvik.system.DexFile;

public class DynamicDexClassLoder extends DexClassLoader {
     private static final String TAG = DynamicDexClassLoder.class.getName();
     private int cookie;
     private Context mContext;

     /**
      * 原构造
      *
      * @param dexPath
      * @param optimizedDirectory
      * @param libraryPath
```

```
     * @param parent
     */
    public DynamicDexClassLoder(String dexPath, String optimizedDirectory,
                        String libraryPath, ClassLoader parent) {
                super(dexPath, optimizedDirectory, libraryPath, parent);
    }

    /**
     * 直接从内存加载 新构造
     *
     * @param dexBytes
     * @param libraryPath
     * @param parent
     * @throws Exception
     */

    public DynamicDexClassLoder(Context context, byte[] dexBytes,
                        String libraryPath, ClassLoader parent, String oriPath,
                        String fakePath) {
                super(oriPath, fakePath, libraryPath, parent);
                setContext(context);
                setCookie(NativeTool.loadDex(dexBytes, dexBytes.length));
    }

    private void setCookie(int kie) {
                cookie = kie;
    }

    private void setContext(Context context) {
                mContext = context;
    }

    private String[] getClassNameList(int cookie) {
                return (String[]) ReflectUtils.invokeStaticMethod(DexFile.class,
                                    "getClassNameList", new Class[] { int.class },
                                    new Object[] { cookie });
    }

    private Class defineClass(String name, ClassLoader loader, int cookie) {
                return (Class) ReflectUtils.invokeStaticMethod(DexFile.class,
                                    "defineClass", new Class[] { String.class, ClassLoader.class,
                                                        int.class }, new Object[] { name, loader, cookie });
    }

    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
                Log.d(TAG, "findClass-" + name);
                Class<?> cls = null;

                String as[] = getClassNameList(cookie);
                for (int z = 0; z < as.length; z++) {
                        if (as[z].equals(name)) {
                                cls = defineClass(as[z].replace('.', '/'),
                                                    mContext.getClassLoader(), cookie);
                        } else {
                                defineClass(as[z].replace('.', '/'), mContext.getClassLoader(),
                                                    cookie);
                        }
                }
```

```
            if (null == cls) {
                    cls = super.findClass(name);
            }

            return cls;
    }

    @Override
    protected URL findResource(String name) {
            Log.d(TAG, "findResource-" + name);
            return super.findResource(name);
    }

    @Override
    protected Enumeration<URL> findResources(String name) {
            Log.d(TAG, "findResources ssss-" + name);
            return super.findResources(name);
    }

    @Override
    protected synchronized Package getPackage(String name) {
            Log.d(TAG, "getPackage-" + name);
            return super.getPackage(name);
    }

    @Override
    protected Class<?> loadClass(String className, boolean resolve)
                    throws ClassNotFoundException {
            Log.d(TAG, "loadClass-" + className + " resolve " + resolve);
            Class<?> clazz = super.loadClass(className, resolve);
            if (null == clazz) {
                    Log.e(TAG, "loadClass fail,maybe get a null-point exception.");
            }
            return clazz;
    }

    @Override
    protected Package[] getPackages() {
            Log.d(TAG, "getPackages sss-");
            return super.getPackages();
    }

    @Override
    protected Package definePackage(String name, String specTitle,
                    String specVersion, String specVendor, String implTitle,
                    String implVersion, String implVendor, URL sealBase)
                    throws IllegalArgumentException {
            Log.d(TAG, "definePackage" + name);
           return super.definePackage(name, specTitle, specVersion, specVendor,
                            implTitle, implVersion, implVendor, sealBase);
    }
```

它的启动过程就是：




但是有一个地方我确实很难理解就是为什么自己写了一个loadDex的方法为什么在它的前面还是要有

```
super(oriPath, fakePath, libraryPath, parent);
```

于是就有了我进一步的跟踪DexClassLoader的这一些方法的过程了，它的调用关系如下：
DexClassLoder-> BaseDexClassLoader->DexPathList->makeDexElements-> loadDexFile-> loadDex->DexFile(String fileName)
而且在

```
1 private static DexFile loadDexFile(File file, File optimizedDirectory)
2             throws IOException {
3         if (optimizedDirectory == null) {
4             return new DexFile(file);
5         } else {
6             String optimizedPath = optimizedPathFor(file, optimizedDirectory);
7             return DexFile.loadDex(file.getPath(), optimizedPath, 0);
8         }
9     }
```

我就明白了，原来DexLoader也实现了DexFile操作，但是这些操作的实现在这个过程中不仅仅只是完成了dex的一个加载，另外也实现了class的映射方便上层去调用底层的方法。