阿里早期Android加固代码的实现分析

原创

置顶 Fly20141201 于 2017-11-02 10:27:29 发布 3465 收藏 7
分类专栏： Android逆向学习 Android系统安全和逆向分析研究 文章标签： 阿里 阿里加固 andorid加固 dvm_dalvik_system_DexopenDexFile

本文链接：https://blog.csdn.net/QQ1084283172/article/details/78320445

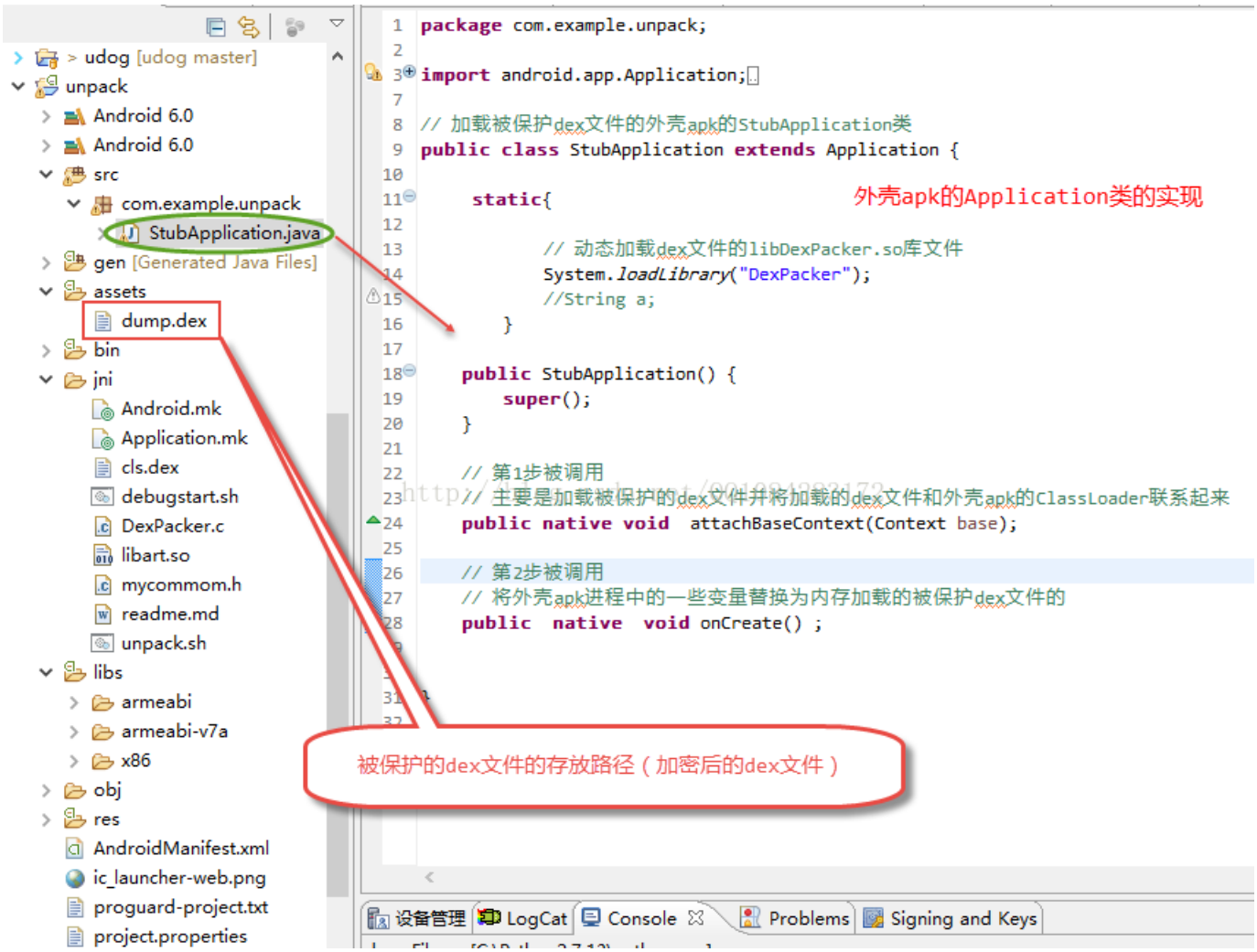

Android逆向学习 同时被 2 个专栏收录
58 篇文章 6 订阅
订阅专栏


Android系统安全和逆向分析研究
72 篇文章 59 订阅
订阅专栏

本文博客地址：http://blog.csdn.net/qq1084283172/article/details/78320445

看雪上有作者（寒号鸟二代）将阿里移动早期的Android加固进行了逆向分析，得到了能够运行成功的NDK代码，主要是基于第2界阿里移动破解大赛的apk逆向分析出来的，文章的原地址《阿里早期加固代码还原4.4-6.0》。周末有空仔细阅读和分析了一下作者寒号鸟二代给出的逆向还原的代码，发现阿里移动的Android加固在dalvik虚拟机模式下dex文件的加载方案和看雪上的一些作者给出的方案一样，具体可以参考桃园小七的博客《APK加壳【2】内存加载dex实现详解》；在Art虚拟机模式下，鉴于Art虚拟机的复杂和兼容性考虑，dex文件的加载方案做不是很底层。尽管Android加固的原理分析已经比较多了，但是为了进一步感受Android加固的产品化，还是有必要学习一下阿里移动加固早期Android加固的代码实现。

1. 阿里移动早期Andorid加固外壳apk应用java层代码的实现如下，为了增加apk应用的逆向难度，Android加固的外壳apk应用的Application类的代理实现子类StubApplication的成员函数attachBaseContext和成员函数onCreate的实现被放在Native层，也就是说原来在java层实现的Android加固代码在Native层实现啦。



2. 由于外壳apk应用类StubApplication的成员函数attachBaseContext和成员函数onCreate是在Native层实现的，因此需要对函数attachBaseContext和函数onCreate进行Jni调用的函数注册，这里采用的是基于JNI_OnLoad函数调用的动态方式的Jni函数注册。外壳apk应用类StubApplication的成员函数attachBaseContext和成员函数onCreate被动态注册以后，就可以实现Android加固java层代码到Native层代码的成功调用，外壳apk应用类StubApplication的java层实现成员函数attachBaseContext和成员函数onCreate分别对应Native层实现的native_attachContextBaseContext函数和native_onCreate函数。

```
// 进行jni函数注册的jni函数信息表
static JNINativeMethod method_table[] = {

  // 被注册的函数的名称（java函数名称）、被注册函数的签名（javah获取）、
  // 被注册函数native实现的函数指针
  { "attachBaseContext", "(Landroid/content/Context;)V",
     (void*) native_attachContextBaseContext },
  { "onCreate", "()V", (void*) native_onCreate },
};

static int registerNativeMethods(JNIEnv* env, const char* className,
  JNINativeMethod* gMethods, int numMethods) {
 jclass clazz;

 // 获取进行jni函数注册的目标类（"com/example/unpack/StubApplication"）
 clazz = (*env)->FindClass(env, className);
 if (clazz == 0) {
  return JNI_FALSE;
 }

 LOGI("gMethods  %s,%s,%p\n ", gMethods[0].name, gMethods[0].signature,
   gMethods[0].fnPtr);

 // 调用函数RegisterNatives为目标类注册jni函数
 if ((*env)->RegisterNatives(env, clazz, gMethods, numMethods) < 0) {
  return JNI_FALSE;
 }

 return JNI_TRUE;
}

int register_ndk_load(JNIEnv *env) {

 // 对类"com/example/unpack/StubApplication"的函数进行注册
 return registerNativeMethods(env, JNIREG_CLASS, method_table,
   NELEM(method_table));
}


JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* reserved) {

 JNIEnv* env = 0;
 jint result = -1;

 // LOGI("JNI_OnLoad is called");
 if ((*vm)->GetEnv(vm, (void**) &env, JNI_VERSION_1_4) != JNI_OK) {
  return result;
 }

 // 进行jni函数的动态注册
 int status = register_ndk_load(env);
 if (!status) {
  LOGI("register call failed");
 }

 return JNI_VERSION_1_4;
}
```

3. 外壳apk应用类StubApplication的成员函数attachBaseContext首先被调用，然后才是成员函数onCreate被调用，**注意**：这两个关键函数的调用顺序，不要搞错了。外壳apk应用类StubApplication的成员函数attachBaseContext对应的Native层实现函数native_attachContextBaseContext的代码实现如下所示：

```
// 1.首先被调用
void native_attachContextBaseContext(JNIEnv *env, jobject obj, jobject ctx) {

 // 保存传入的StubApplication类实例对象
 jobject application_obj = obj;

 // 通过反射调用，获取必要的函数调用id和成员变量获取id以备用以及当前Android虚拟机所处的运行模式
 init_class(env, obj, ctx);
 LOGI(
   "arg:application_obj:%p, myContextWrapper:%p, ContextWrapper_attachBaseContext:%p",
   application_obj, myContextWrapper,
   ContextWrapper_attachBaseContext);


 // 调用父类的函数android.content.ContextWrapper.attachBaseContext(ctx)
 (*env)->CallNonvirtualVoidMethod(env, application_obj, myContextWrapper,
   ContextWrapper_attachBaseContext, ctx);

 // 获取StubApplication类
 jclass v12 = (*env)->GetObjectClass(env, application_obj);
 // 获取StubApplication类的非静态成员方法getFilesDir的调用id
 jmethodID v13 = (*env)->GetMethodID(env, v12, "getFilesDir",
   "()Ljava/io/File;");
 // 调用函数com.example.unpack.StubApplication.getFilesDir
 // 获取文件路径/data/data/<packagename>/files的File实例对象
 jobject file_obj = (*env)->CallObjectMethod(env, obj, v13);

 //   LOGI("file_obj:%p",file_obj);
 // 获取File类
 jclass file_classz = (*env)->GetObjectClass(env, file_obj);
 // 获取类File的非静态成员函数getAbsolutePath的调用id
 jmethodID v18 = (*env)->GetMethodID(env, file_classz, "getAbsolutePath",
   "()Ljava/lang/String;");
 // 调用类File的函数getAbsolutePath获取文件路径/data/data/<packagename>/files
 jobject mAbsolutePath = (*env)->CallObjectMethod(env, file_obj, v18);

 // 6.0下为/data/user/0/packagename/files/目录
 // 将java字符串转换成C语言字符串
 mAbsolutePath_str = (*env)->GetStringUTFChars(env, mAbsolutePath, 0);
 LOGI("global files path is %s", mAbsolutePath_str);


 // 调用类StubApplication对象实例函数getApplicationInfo()获取ApplicationInfo信息
 jobject ApplicationInfo = (*env)->CallObjectMethod(env, application_obj,
   context_getApplicationInfo);
 // 获取类ApplicationInfo的实例成员变量nativeLibraryDir
 jobject v24 = (*env)->GetObjectField(env, ApplicationInfo,
   ApplicationInfo_nativeLibraryDir);
 // 得到当前apk的so库文件存放路径
 const char* mNativeLibraryDir = (*env)->GetStringUTFChars(env, v24, 0);
 //LOGI("mNativeLibraryDir is %s",mNativeLibraryDir);
```

```
// 调用类StubApplication的实例方法getPackageResourcePath()
jobject v32 = (*env)->CallObjectMethod(env, application_obj,
  context_getPackageResourePath);
// 得到当前APK应用存放资源的文件路径
const char* mPackageResourePath = (*env)->GetStringUTFChars(env, v32, 0);


// 设置环境变量"APKPATH"的值即修改为被加密apk的dex文件所在的路径
setenv("APKPATH", mPackageResourePath, 1);
//LOGI("APK Path is %s",mPackageResourePath);

// 调用类Context的实例方法getPackageName()
jobject packageName = (*env)->CallObjectMethod(env, ctx, context_getPackageName);
// 得到当前APK应用(即外壳apk应用)的包名
mPackageName = (*env)->GetStringUTFChars(env, packageName, 0);
LOGI("mPackageName:%s", mPackageName);


// public ClassLoader getClassLoader()
// 调用类Context的实例方法getClassLoader()获取到外壳apk应用的ClassLoader实例对象
jobject classLoader = (*env)->CallObjectMethod(env, ctx,
  context_getClassLoader);
LOGI("classLoader:%p", classLoader);


char szPath[260] = { 0 };
//   sprintf(szPath,"%s/dump.dex",mAbsolutePath_str);
// 拼接字符串得到文件路径
sprintf(szPath, "/data/data/%s/files/dump.dex", mPackageName);
LOGI("szPath:%s", szPath);

// 将外壳apk应用资源文件夹下（加密处理的dex文件dump.dex）释放
// 到szPath指定的文件路径/data/data/<PackageName>/files/dump.dex中
myExtractFile(env, application_obj, szPath);


// 根据当前外壳apk应用所处的android虚拟机环境进行dex文件的内存加载(important)
if (isDalvik) {

 // dalvik下内存加载指定文件路径szPath下的dex文件
 myLoadDex_dvm(env, szPath);

} else {

 // art下内存加载指定文件路径szPath下的dex文件
 myLoadDex_art(env, szPath);
}

// 在ClassLoader中用内存加载的dex文件的mCookie值替换掉外壳apk应用的mCookie值(important)
replace_classloader_cookie(env, classLoader);
LOGI("enter new application");


// 构建jni字符串"android.app.Application"
// 注意:准确来说这儿的Application应为被加固保护的dex文件的Application类
// 通过解析外壳apk应用的AndroidMainfest.xml文件或者配置文件中来获取的（需要注意一下）
jstring newapp = (*env)->NewStringUTF(env, "android.app.Application");
// 调用ClassLoader的非静态成员方法loadClass加载系统类"android.app.Application"
jobject findclass_classz = (*env)->CallObjectMethod(env, classLoader,
```

```
   classLoader_findClass, newapp);
 if (!findclass_classz) {
  LOGI("can't findClass realAppClass");
  return;
 }
 // 调用android系统类"android.app.Application"的构造函数创建Application实例对象
 jmethodID initMethod = (*env)->GetMethodID(env, findclass_classz, "<init>", "()V");
 // 创建android系统类"android.app.Application"的实例对象
 onCreateObj = (*env)->NewObject(env, findclass_classz, initMethod);

 // 调用父类即android系统类android.app.Application的attach()函数
 (*env)->CallVoidMethod(env, onCreateObj, Application_attach, ctx);

 // 创建android系统类"android.app.Application"实例对象的全局引用
 if (onCreateObj) {
  onCreateObj = (*env)->NewGlobalRef(env, onCreateObj);
 }
 LOGI("enter realAppClass");
}
```

外壳apk应用类StubApplication的成员函数attachBaseContext对应的Native层实现 **函数 native_attachContextBaseContext** 的具体实现流程梳理如下：

< 1 >. 通过类反射调用获取必要的类方法调用id和类成员变量的获取id以及获取当前Android虚拟机的运行模式是Dalvik虚拟机模式还是Art虚拟机模式，主要的代码实现是在函数init_class中完成的。

```
859
860 // 1.首先被调用
861 void native_attachContextBaseContext(JNIEnv *env, jobject obj, jobject ctx) {
862
863     // 保存传入的StubApplication类实例对象
864     jobject application_obj = obj;
865
866     // 通过类反射调用，获取必要的函数调用id和成员变量获取id以备用
867     init_class(env, obj, ctx);
868     LOGI(
869             "arg:application_obj:%p, myContextWrapper:%p, ContextWrapper_attachBaseContext:%p",
870             application_obj, myContextWrapper,
871             ContextWrapper_attachBaseContext);
872
```


init_class函数的代码实现分析如下，主要获取一些需要用到的Android系统函数的调用方法id和成员变量获取id，获取这些目标函数的调用id以及变量的获取id会因为Android系统的版本变化会稍有不同，在init_class函数最后还会获取外壳apk应用当前运行的Android虚拟机模式是Dalvik还是Art。

```
// 获取指定名称类的全局引用
jclass myFindClass(JNIEnv *env, jclass* ptr, char* name) {

 jobject g_cls_string;
 // 获取指定名称的目标类
 jclass clazz = (*env)->FindClass(env, name);
 if (clazz) {
  // 获取类的全局引用
```

```
  // 获取类的全局引用
  g_cls_string = (*env)->NewGlobalRef(env, clazz);
  // 保存返回值
  *ptr = g_cls_string;

  return g_cls_string;

 } else {
  return 0;
 }

}

// [ *Android系统版本不一样，一些系统函数的参数调用会有所不同 *]
void init_class(JNIEnv *env, jobject obj, jobject ctx) {

 // 获取类android.os.Build.VERSION的全局引用
 if (!myFindClass(env, &Build_version, "android/os/Build$VERSION")) {

  LOGI("ERROR:Build$VERSION");
  return;
 }
 // 获取类android.os.Build.VERSION的静态成员变量的jfieldID信息结构体
 jfieldID fieldID = ((*env)->GetStaticFieldID)(env, Build_version, "SDK_INT",
   "I");
 // 获取类android.os.Build.VERSION的静态成员变量的值SDK_INT
 // 即获取当前Android系统的版本号信息
 sdk_int = (*env)->GetStaticIntField(env, Build_version, fieldID);
 LOGI("sdk_int %d\n", sdk_int);

 // 获取类android.app.ActivityThread的全局引用
 if (!myFindClass(env, &ActivityThread, "android/app/ActivityThread")) {

  LOGI("ERROR:ActivityThread");
  return;
 }

 // 对当前Android系统的版本号进行判断，执行对应的代码流程
 if (sdk_int > 18) {

  // 获取类android.app.ActivityThread的非静态成员变量mPackages的jfieldID
  mPackages = (*env)->GetFieldID(env, ActivityThread, "mPackages",
    "Landroid/util/ArrayMap;");

  // 获取类android.util.ArrayMap的全局引用
  if (!myFindClass(env, &myArrayMap, "android/util/ArrayMap")) {

   LOGI("ERROR:myArrayMap");
   return;
  }
  // 获取类android.util.ArrayMap的非静态成员方法get函数
  // android.util.ArrayMap.get(Objetc )
  ArrayMap_get = (*env)->GetMethodID(env, myArrayMap, "get",
    "(Ljava/lang/Object;)Ljava/lang/Object;");

  // 获取类android.app.ActivityThread的非静态成员变量mBoundApplication的jfieldID
  // android.app.ActivityThread$AppBindData mBoundApplication
  mBoundApplication = (*env)->GetFieldID(env, ActivityThread,
    "mBoundApplication",
    "Landroid/app/ActivityThread$AppBindData;");
```

```
// 获取类android.app.ActivityThread的非静态成员变量mInitialApplication的jfieldID
// android.app.Application mInitialApplication
mInitialApplication = (*env)->GetFieldID(env, ActivityThread,
  "mInitialApplication", "Landroid/app/Application;");

// 获取类android.app.ActivityThread的非静态成员变量mAllApplicationsn的jfieldID
// java.util.ArrayList mAllApplications
mAllApplications = (*env)->GetFieldID(env, ActivityThread,
  "mAllApplications", "Ljava/util/ArrayList;");

// 获取类android.app.ActivityThread的静态成员方法currentActivityThread的MethodID
// android.app.ActivityThread currentActivityThread
currentActivityThread = (*env)->GetStaticMethodID(env, ActivityThread,
  "currentActivityThread", "()Landroid/app/ActivityThread;");
//LOGI("ActivityThread:%p,%p,%p,%p",mBoundApplication,mInitialApplication,mAllApplications,currentActivit

// 获取类android.app.ActivityThread$AppBindData的全局引用（内部类）
if (!myFindClass(env, &AppBindData,
  "android/app/ActivityThread$AppBindData")) {

 LOGI("ERROR:AppBindData");
 return;
}
// 获取类android.app.ActivityThread$AppBindData的非静态成员变量info的jfieldID（android.app.LoadedApk）
// android.app.LoadedApk info
AppBindData_info = (*env)->GetFieldID(env, AppBindData, "info",
  "Landroid/app/LoadedApk;");

if (!myFindClass(env, &myArrayList, "java/util/ArrayList")) {

 LOGI("ERROR:myArrayList");
 return;
}
// 获取类java.util.ArrayList的非静态成员方法size的调用id
arraylist_size = (*env)->GetMethodID(env, myArrayList, "size", "()I");
// 获取类java.util.ArrayList的非静态成员方法get的调用id
arraylist_get = (*env)->GetMethodID(env, myArrayList, "get",
  "(I)Ljava/lang/Object;");
// 获取类java.util.ArrayList的非静态成员方法set的调用id
arraylist_set = (*env)->GetMethodID(env, myArrayList, "set",
  "(ILjava/lang/Object;)Ljava/lang/Object;");

if (!myFindClass(env, &myContext, "android/content/Context")) {

 LOGI("ERROR:myContext");
 return;
}
// 获取类android.content.Context的非静态成员方法getPackageName的调用id
context_getPackageName = (*env)->GetMethodID(env, myContext,
  "getPackageName", "()Ljava/lang/String;");
// 获取类android.content.Context的非静态成员方法getApplicationInfo的调用id
context_getApplicationInfo = (*env)->GetMethodID(env, myContext,
  "getApplicationInfo", "()Landroid/content/pm/ApplicationInfo;");
// 获取类android.content.Context的非静态成员方法getClassLoader的调用id
context_getClassLoader = (*env)->GetMethodID(env, myContext,
  "getClassLoader", "()Ljava/lang/ClassLoader;");
// 获取类android.content.Context的非静态成员方法getAssets的调用id
context_getAssets = (*env)->GetMethodID(env, myContext, "getAssets",
  "()Landroid/content/res/AssetManager;");
```

```
  "()Landroid/content/res/AssetManager;");
// 获取类android.content.Context的非静态成员方法getPackageResourcePath的调用id
context_getPackageResourePath = (*env)->GetMethodID(env, myContext,
  "getPackageResourcePath", "()Ljava/lang/String;");

if (!myFindClass(env, &myWeakReference,
  "java/lang/ref/WeakReference")) {

 LOGI("ERROR:myWeakReference");
 return;
}
// 获取类java.lang.ref.WeakReference的非静态成员方法get的调用id
WeakReference_get = (*env)->GetMethodID(env, myWeakReference, "get",
  "()Ljava/lang/Object;");

if (!myFindClass(env, &myLoadedApk, "android/app/LoadedApk")) {

 LOGI("ERROR:myLoadedApk");
 return;
}
// 获取类android.app.LoadedApk的非静态成员mClassLoader的获取id
LoadedApk_mClassLoader = (*env)->GetFieldID(env, myLoadedApk,
  "mClassLoader", "Ljava/lang/ClassLoader;");
// 获取类android.app.LoadedApk的非静态成员mApplication的获取id
LoadedApk_mApplication = (*env)->GetFieldID(env, myLoadedApk,
  "mApplication", "Landroid/app/Application;");

if (!myFindClass(env, &myApplicationInfo,
  "android/content/pm/ApplicationInfo")) {

 LOGI("ERROR:myApplicationInfo");
 return;
}
// 获取类android.content.pm.ApplicationInfo的非静态成员变量dataDir的获取id
ApplicationInfo_dataDir = (*env)->GetFieldID(env, myApplicationInfo,
  "dataDir", "Ljava/lang/String;");
// 获取类android.content.pm.ApplicationInfo的非静态成员变量nativeLibraryDir的获取id
ApplicationInfo_nativeLibraryDir = (*env)->GetFieldID(env,
  myApplicationInfo, "nativeLibraryDir", "Ljava/lang/String;");
// 获取类android.content.pm.ApplicationInfo的非静态成员变量sourceDir的获取id
ApplicationInfo_sourceDir = (*env)->GetFieldID(env, myApplicationInfo,
  "sourceDir", "Ljava/lang/String;");

if (!myFindClass(env, &myApplication, "android/app/Application")) {
 LOGI("ERROR:myApplication");
 return;
}
// 获取类android.app.Application的非静态成员函数onCreate的调用id
Application_onCreate = (*env)->GetMethodID(env, myApplication,
  "onCreate", "()V");
// 获取类android.app.Application的非静态成员函数attach的调用id
Application_attach = (*env)->GetMethodID(env, myApplication, "attach",
  "(Landroid/content/Context;)V");

if (!myFindClass(env, &myContextWrapper,
  "android/content/ContextWrapper")) {

 LOGI("ERROR:myContextWrapper");
 return;
}
```

```
// 获取类android.content.ContextWrapper的非静态成员函数attachBaseContext的调用id
ContextWrapper_attachBaseContext = (*env)->GetMethodID(env,
  myContextWrapper, "attachBaseContext",
  "(Landroid/content/Context;)V");

LOGI("PathClassLoader start");
// 获取类dalvik.system.PathClassLoader的全局引用
if (!myFindClass(env, &myPathClassLoader,
  "dalvik/system/PathClassLoader")) {

 LOGI("ERROR:myPathClassLoader");
 return;
}

if (sdk_int > 13) {

 if (!myFindClass(env, &myBaseDexClassLoader,
   "dalvik/system/BaseDexClassLoader")) {

  LOGI("ERROR:myBaseDexClassLoader");
  return;
 }
 // 获取类dalvik.system.BaseDexClassLoader的非静态成员变量pathList的获取id
 BaseDexClassLoader_pathList = (*env)->GetFieldID(env,
   myBaseDexClassLoader, "pathList",
   "Ldalvik/system/DexPathList;");

 if (!myFindClass(env, &myDexPathList,
   "dalvik/system/DexPathList")) {
  LOGI("ERROR:myDexPathList");
  return;
 }
 // 获取类dalvik.system.DexPathList的非静态成员变量dexElements的获取id
 DexPathList_dexElements = (*env)->GetFieldID(env, myDexPathList,
   "dexElements", "[Ldalvik/system/DexPathList$Element;");

 if (!myFindClass(env, &myElement,
   "dalvik/system/DexPathList$Element")) {
  LOGI("ERROR:myElement");
  return;
 }
 // 获取类dalvik.system.DexPathList$Element的非静态成员变量dexFile的获取id
 DexPathList_Element_dexFile = (*env)->GetFieldID(env, myElement,
   "dexFile", "Ldalvik/system/DexFile;");

 if (sdk_int > 22) //6.0
  // Android 6.0版本,获取类dalvik.system.DexPathList$Element的非静态成员变量dir的获取id
  DexPathList_Element_file = (*env)->GetFieldID(env, myElement,
    "dir", "Ljava/io/File;");
 else
  // 其他版本,获取类dalvik.system.DexPathList$Element的非静态成员变量file的获取id
  DexPathList_Element_file = (*env)->GetFieldID(env, myElement,
    "file", "Ljava/io/File;");

 if (!myFindClass(env, &myFile, "java/io/File")) {

  LOGI("ERROR:myFile");
  return;
 }
// 获取类java.io.File的非静态成员方法getAbsolutePath的调用id
```

```
// 获取类java.io.File的非静态成员方法getAbsolutePath的调用id
myFile_getAbsolutePath = (*env)->GetMethodID(env, myFile,
  "getAbsolutePath", "()Ljava/lang/String;");
LOGI("PathClassLoader end");


// 获取类的dalvik.system.DexFile的全局引用
if (!myFindClass(env, &myDexFile, "dalvik/system/DexFile")) {
 LOGI("ERROR:myDexFile");
 return;
}
// mCookie值获取需要的函数
if (sdk_int > 22) {

 // 获取类dalvik.system.DexFile的非静态成员变量mCookie的获取id
 mCookie = (*env)->GetFieldID(env, myDexFile, "mCookie",
   "Ljava/lang/Object;");
 // 获取类dalvik.system.DexFile的静态成员方法openDexFile的调用id
 myOpenDexFile =
   (*env)->GetStaticMethodID(env, myDexFile, "openDexFile",
     "(Ljava/lang/String;Ljava/lang/String;I)Ljava/lang/Object;");

// 其它系统版本，获取类dalvik.system.DexFile的成员变量mCookie的获取id和函数openDexFile的调用id
} else if (sdk_int > 19) {

 mCookie = (*env)->GetFieldID(env, myDexFile, "mCookie", "J");
 myOpenDexFile = (*env)->GetStaticMethodID(env, myDexFile,
   "openDexFile",
   "(Ljava/lang/String;Ljava/lang/String;I)J");

// 主要不同的Android系统版本的openDexFile函数的函数返回值会有所不同
} else {

 mCookie = (*env)->GetFieldID(env, myDexFile, "mCookie", "I");
 myOpenDexFile = (*env)->GetStaticMethodID(env, myDexFile,
   "openDexFile",
   "(Ljava/lang/String;Ljava/lang/String;I)I");
}

// android 5+以上无法用findClass找到android.app.Application类
if (!myFindClass(env, &myClassLoader, "java/lang/ClassLoader")) {
 LOGI("ERROR:myClassLoader");
 return;
}
// 获取类java.lang.ClassLoader的非静态方法loadClass的调用id
classLoader_findClass = (*env)->GetMethodID(env, myClassLoader,
  "loadClass", "(Ljava/lang/String;)Ljava/lang/Class;");
LOGI("System start");


if (!myFindClass(env, &mySystem, "java/lang/System")) {
 LOGI("ERROR:myClassLoader");
 return;
}
// 获取类java.lang.System的静态成员方法getProperty的调用id
system_getProperty = (*env)->GetStaticMethodID(env, mySystem,
  "getProperty", "(Ljava/lang/String;)Ljava/lang/String;");

LOGI("SystemProperties start");
```

```
   status = myFindClass(env, &mySystemProperties,
     "android/os/SystemProperties");
   if (status) {

    // 获取类android.os.SystemProperties的静态成员方法get的调用id
    SystemProperties_get = (*env)->GetStaticMethodID(env,
      mySystemProperties, "get",
      "(Ljava/lang/String;)Ljava/lang/String;");

    jstring vmname = (*env)->NewStringUTF(env, "java.vm.name");
    // 调用函数java.lang.System.getProperty("java.vm.name")获取Android虚拟机的名称
    // 判断是否是dalvik虚拟机模式
    jobject tmp = (*env)->CallStaticObjectMethod(env, mySystem,
      system_getProperty, vmname);
    // 转换成C语言字符串
    const char* v22 = (*env)->GetStringUTFChars(env, tmp, 0);
    LOGI("------- vmNameStr:%s", v22);
    (*env)->ReleaseStringUTFChars(env, tmp, v22);

    // persist.sys.dalvik.vm.lib
    // persist.sys.dalvik.vm.lib.2
    jstring vmlibstr = (*env)->NewStringUTF(env,
      "persist.sys.dalvik.vm.lib.2");
    // 调用函数android.os.SystemProperties.get("persist.sys.dalvik.vm.lib.2")
    // 获取Android虚拟机libart.so字符串，判断是否是art虚拟机模式
    jobject runtime = (*env)->CallStaticObjectMethod(env,
      mySystemProperties, SystemProperties_get, vmlibstr);
    const char* v28 = (*env)->GetStringUTFChars(env, runtime, 0);
    (*env)->ReleaseStringUTFChars(env, runtime, v28);

    jstring vm_version = (*env)->NewStringUTF(env,
      "java.vm.version");
    // 调用函数java.lang.System.getProperty("java.vm.version")获取虚拟机的版本号
    jobject v32 = (*env)->CallStaticObjectMethod(env, mySystem,
      system_getProperty, vm_version);
    const char* runtime_version = (*env)->GetStringUTFChars(env, v32, 0);
    LOGI("---- vmVersionStr:%s", runtime_version);
    // 字符串转浮点型
    double d = atof(runtime_version);

    // 根据虚拟机版本号判断Android虚拟机的运行模式
    if (d > 2)
     // art 模式
     isDalvik = 0;
    else
     // dalvik 模式
     isDalvik = 1;
    (*env)->ReleaseStringUTFChars(env, v32, runtime_version);

    return;
   }

  }
 }
}
```

阿里移动Android加固中，判断当前Android系统所处的虚拟机模式是Dalvik模式还是Art模式的方法可以参考讨论《How can I detect the Android runtime (Dalvik or ART)?》。

At least, as early as June 2014 Google has released an official documentation on how to correctly verify the current runtime in use:

> You can verify which runtime is in use by calling System.getProperty("java.vm.version"). If ART is in use, the property's value is "2.0.0" or higher.

With that, now there is no need to go through reflection and simply check the corresponding system property:


```
private boolean getIsArtInUse() {
    final String vmVersion = System.getProperty("java.vm.version");
    return vmVersion != null && vmVersion.startsWith("2");
}
```

< 2 >. 在外壳apk应用类StubApplication成员函数attachContextBaseContext中，通过类反射调用父类android.content.ContextWrapper实例的类成员函数attachBaseContext。

```
/**
 * 在外壳apk应用的类StubApplication成员函数中attachContextBaseContext
 * 调用父类的方法super.attachBaseContext(ctx);
 */
// 调用父类android.content.ContextWrapper的实例成员函数attachBaseContext
if (!myFindClass(env, &myContextWrapper, "android/content/ContextWrapper")) {

    LOGI("ERROR:myContextWrapper");
    return;
}



// 获取类android.content.ContextWrapper的非静态成员函数attachBaseContext的调用id
ContextWrapper_attachBaseContext = (*env)->GetMethodID(env,
        myContextWrapper, "attachBaseContext", "(Landroid/content/Context;)V");

// 调用父类的函数android.content.ContextWrapper.attachBaseContext(ctx)
(*env)->CallNonvirtualVoidMethod(env, application_obj,
        myContextWrapper, ContextWrapper_attachBaseContext, ctx);
```

< 3 >. 获取外壳apk应用的files文件目录的全路径字符串 /data/data/<packagename>/files ，供后面释放被保护的dex文件使用。

```
/**
 * 获取外壳apk应用的files文件路径字符串 /data/data/<packagename>/files
 * String data_data_pgname_files = this.getFilesDir().getAbsolutePath();
 */
// 获取StubApplication类
jclass v12 = (*env)->GetObjectClass(env, application_obj);

// 获取StubApplication类的非静态成员方法getFilesDir的调用id
jmethodID v13 = (*env)->GetMethodID(env, v12, "getFilesDir", "()Ljava/io/File;");

// 调用函数com.example.unpack.StubApplication.getFilesDir
// 获取文件路径/data/data/<packagename>/files的File实例对象
jobject file_obj = (*env)->CallObjectMethod(env, obj, v13);

//   LOGI("file_obj:%p",file_obj);
// 获取File类
jclass file_classz = (*env)->GetObjectClass(env, file_obj);

// 获取类File的非静态成员函数getAbsolutePath的调用id
jmethodID v18 = (*env)->GetMethodID(env,
        file_classz, "getAbsolutePath", "()Ljava/lang/String;");

// 调用类File的函数getAbsolutePath获取文件路径/data/data/<packagename>/files
jobject mAbsolutePath = (*env)->CallObjectMethod(env, file_obj, v18);

// 6.0下为/data/user/0/packagename/files/目录
// 将java字符串转换成C语言字符串
mAbsolutePath_str = (*env)->GetStringUTFChars(env, mAbsolutePath, 0);
LOGI("global files path is %s", mAbsolutePath_str);
```


< 4 >. 获取外壳apk应用存放so库文件的文件目录 nativeLibraryDir 的路径字符串，供Android测试检查使用。注意：截图中java代码描述稍微有一点问题，应该是外壳apk应用的类com.example.unpack.StubApplication实例调用非静态成员函数getApplicationInfo()获取当前外壳apk应用的android.content.pm.ApplicationInfo信息。

```
/*
 * 获取外壳apk应用存放so库文件的文件路径字符串
 * android.content.pm.ApplicationInfo applicationInfo = ctx.getApplicationInfo();
 * String str_nativeLibraryDir = applicationInfo.nativeLibraryDir;
 */
if (!myFindClass(env, &myContext, "android/content/Context")) {

    LOGI("ERROR:myContext");
    return;
}
// 获取类android.content.Context的非静态成员方法getApplicationInfo的调用id
context_getApplicationInfo = (*env)->GetMethodID(env,
        myContext, "getApplicationInfo", "()Landroid/content/pm/ApplicationInfo;");

// 调用类StubApplication对象实例函数getApplicationInfo()获取ApplicationInfo信息
jobject ApplicationInfo = (*env)->CallObjectMethod(env,
        application_obj, context_getApplicationInfo);


if (!myFindClass(env, &myApplicationInfo, "android/content/pm/ApplicationInfo")) {

    LOGI("ERROR:myApplicationInfo");
    return;
}
// 获取类android.content.pm.ApplicationInfo的非静态成员变量nativeLibraryDir的获取id
ApplicationInfo_nativeLibraryDir = (*env)->GetFieldID(env,
        myApplicationInfo, "nativeLibraryDir", "Ljava/lang/String;");

// 获取类ApplicationInfo的实例成员变量nativeLibraryDir
jobject v24 = (*env)->GetObjectField(env, ApplicationInfo,
        ApplicationInfo_nativeLibraryDir);

// 得到当前外壳apk的so库文件存放路径
const char* mNativeLibraryDir = (*env)->GetStringUTFChars(env, v24, 0);
//LOGI("mNativeLibraryDir is %s", mNativeLibraryDir);
```

< 5 >. 获取外壳apk应用的资源Resource的存放文件目录并设置外壳apk应用进程APKPATH环境变量为外壳apk应用的资源存放文件目录的路径。

```
/*
 * 获取外壳apk应用的资源存放路径并设置外壳apk应用进程APKPATH为资源存放路径
 * String str_okgResourcePath = this.getPackageResourcePath();
 * setenv("APKPATH", mPackageResourePath, 1);
 */
if (!myFindClass(env, &myContext, "android/content/Context")) {

    LOGI("ERROR:myContext");
    return;
}
// 获取类android.content.Context的非静态成员方法getPackageResourcePath的调用id
context_getPackageResourePath = (*env)->GetMethodID(env,
        myContext, "getPackageResourcePath", "()Ljava/lang/String;");


// 调用类StubApplication的实例方法getPackageResourcePath()
jobject v32 = (*env)->CallObjectMethod(env, application_obj,
        context_getPackageResourePath);

// 得到当前APK应用存放资源的文件路径
const char* mPackageResourePath = (*env)->GetStringUTFChars(env, v32, 0);

// 设置环境变量"APKPATH"的值即修改为外壳apk所在的资源存放路径
setenv("APKPATH", mPackageResourePath, 1);
//LOGI("APK Path is %s", mPackageResourePath);
```

< 6 >. 获取外壳apk应用的包名packageName字符串，并调用外壳apk应用的当前类android.content.Context实例的实例成员函数getClassLoader，获取当前外壳apk应用程序对应的ClassLoader实例对象，供后面替换外壳apk应用的mCookie值为被保护dex文件内存加载成功后返回的mCookie值使用，也就是说外壳apk应用对应的ClassLoader实例将会被被保护的dex文件的ClassLoader实例所无缝替换，从实现外壳apk应用和被保护dex文件的无缝替换衔接。

```
/*
 * 获取外壳apk应用的包名packageName
 * String mPackageName = ctx.getPackageName();
 *
 * 获取外壳apk所对应的ClassLoader实例对象
 * ClassLoader classLoader = ctx.getClassLoader();
 */
if (!myFindClass(env, &myContext, "android/content/Context")) {

    LOGI("ERROR:myContext");
    return;
}
// 获取类android.content.Context的非静态成员方法getPackageName的调用id
context_getPackageName = (*env)->GetMethodID(env,
        myContext, "getPackageName", "()Ljava/lang/String;");

// 调用类Context的实例方法getPackageName()
jobject packageName = (*env)->CallObjectMethod(env, ctx, context_getPackageName);

// 得到当前APK应用(即外壳apk应用)的包名
mPackageName = (*env)->GetStringUTFChars(env, packageName, 0);
LOGI("mPackageName:%s", mPackageName);

// 获取类android.content.Context的非静态成员方法getClassLoader的调用id
context_getClassLoader = (*env)->GetMethodID(env,
        myContext, "getClassLoader", "()Ljava/lang/ClassLoader;");

// public ClassLoader getClassLoader()
// 调用类Context的实例方法getClassLoader()获取到外壳apk应用的ClassLoader实例对象
jobject classLoader = (*env)->CallObjectMethod(env,
        ctx, context_getClassLoader);

LOGI("classLoader:%p", classLoader);
```

< 7 >. 向外壳apk应用的files文件目录/data/data/<packagename>/files下释放被保护的dex文件dump.dex，这里直接将被保护的dex文件dump.dex放在了外壳apk应用的assets文件目录下，比较随意也没有对被保护的dex文件进行加密处理（这里主要是为了学习Android加固原理方便）；在正常的Android加固产品中一般会对被保护的dex文件进行加密处理。

```
/*
 * 向外壳apk应用的文件目录/data/data/<packagename>/files下释放dex文件dump.dex
 */
char szPath[260] = { 0 };
//    sprintf(szPath,"%s/dump.dex", mAbsolutePath_str);

// 拼接字符串得到释放dex文件的文件路径
sprintf(szPath, "/data/data/%s/files/dump.dex", mPackageName);
LOGI("szPath:%s", szPath);

// 将外壳apk应用资源文件夹下（加密处理的dex文件dump.dex）释放
// 到szPath指定的文件路径/data/data/<PackageName>/files/dump.dex中
myExtractFile(env, application_obj, szPath);
```

将外壳apk应用的资源文件夹assets下的被保护dex文件dump.dex(一般情况下，会对原始的dex文件进行加密处理，释放dex文件的时候进行dex文件的解密操作)释放到外壳apk应用的/data/data/<packagename>/files文件夹下，得到dex文件/data/data/<PackageName>/files/dump.dex，其中<PackageName>为外壳apk应用的包名。

```
/***
 * application_obj--外壳apk应用的Application实例对象
 * szPath--外壳apk应用资源文件夹下（加密的dex文件）释放后的dex文件路径/data/data/<PackageName>/files/dump.dex
 *
 * 将外壳apk应用资源文件夹下（加密处理的dex文件）释放到szPath指定的文件路径中
 */
void myExtractFile(JNIEnv* env, jobject application_obj, const char* szPath) {

 AAssetManager* mgr;

 // 判断文件路径szPath所指定的dex文件是否已存在
 if (access(szPath, R_OK)) {

  // 查找到类Application
  jclass application_cls = (*env)->GetObjectClass(env, application_obj);
  // 获取类Application的非静态类方法getAssets的调用id
  jmethodID getAssetsId = (*env)->GetMethodID(env, application_cls, "getAssets",
    "()Landroid/content/res/AssetManager;");
  // 调用Application类的非静态类方法getAssets函数得到外壳apk应用的java端资源管理对象
  jobject aassetMgr = (*env)->CallObjectMethod(env, application_obj, getAssetsId);

  // 将java端的资源管理对象转换成c端的资源管理对象AAssetManager
  mgr = AAssetManager_fromJava(env, aassetMgr);
  if (mgr == NULL) {

   LOGI(" %s", "AAssetManager==NULL");
   return;
  }

  FILE* file;
  void* buffer;
  int numBytesRead;

  // 打开外壳apk应用的资源文件夹下的文件dump.dex
  AAsset* asset = AAssetManager_open(mgr, "dump.dex", AASSET_MODE_STREAMING);
  if (asset) {

   // 以szPath指定的字符串为文件路径创建新文件
   file = fopen(szPath, "w");
   //       int bufferSize = AAsset_getLength(asset);
   //       LOGI("buffersize is %d",bufferSize);

   // 申请内存空间用于临时存放数据
   buffer = malloc(1024);
   while (1) {

    // 从外壳apk应用资源文件夹下的dump.dex文件读取数据到申请的内存空间中
    numBytesRead = AAsset_read(asset, buffer, 1024);
    // 文件数据读取完成，跳出循环
    if (numBytesRead <= 0) {
     break;
    }

    // 将从外壳apk应用资源文件夹下的dump.dex文件读取到数据写入到szPath为文件路径的文件中
    fwrite(buffer, numBytesRead, 1, file);
```

```
    fwrite(buffer, numBytesRead, 1, file);
   }

   // 资源的清理和释放
   free(buffer);
   fclose(file);
   AAsset_close(asset);

  } else {

   LOGI("Error AAssetManager_open");
   return;
  }

 } else {
  LOGI("dump.dex existed");
 }
}
```

< 8 >.根据当前外壳apk应用所处的android虚拟机环境进行dex文件的内存加载，Dalvik虚拟机模式下调用函数myLoadDex_dvm进行被保护dex文件的加载，Art虚拟机模式下调用函数myLoadDex_art进行被保护dex文件的加载。

```
// 根据当前外壳apk应用所处的android虚拟机环境进行dex文件的内存加载(important)
if (isDalvik) {
    // dalvik下内存加载指定文件路径szPath下的dex文件
    myLoadDex_dvm(env, szPath);
} else {
    // art下内存加载指定文件路径szPath下的dex文件
    myLoadDex_art(env, szPath);
}
```


获取Dalvik虚拟机模式下和Art虚拟机模式下dex文件加载后mCookie值的获取id以及获取Art虚拟模式下加载dex文件的函数openDexFile的调用id，供后面Art虚拟机模式下加载dex文件使用。

```
// 获取类的dalvik.system.DexFile的全局引用
if (!myFindClass(env, &myDexFile, "dalvik/system/DexFile")) {
 LOGI("ERROR:myDexFile");
 return;
}
// mCookie值获取需要的函数
if (sdk_int > 22) {

 // 获取类dalvik.system.DexFile的非静态成员变量mCookie的获取id
 mCookie = (*env)->GetFieldID(env, myDexFile, "mCookie", "Ljava/lang/Object;");

 // 获取类dalvik.system.DexFile的静态成员方法openDexFile的调用id
 myOpenDexFile = (*env)->GetStaticMethodID(env,
   myDexFile,
   "openDexFile",
   "(Ljava/lang/String;Ljava/lang/String;I)Ljava/lang/Object;");

// 其它系统版本，获取类dalvik.system.DexFile的成员变量mCookie的获取id和函数openDexFile的调用id
} else if (sdk_int > 19) {

 mCookie = (*env)->GetFieldID(env, myDexFile, "mCookie", "J");
 myOpenDexFile = (*env)->GetStaticMethodID(env,
   myDexFile,
   "openDexFile",
   "(Ljava/lang/String;Ljava/lang/String;I)J");

// 主要不同的Android系统版本的openDexFile函数的函数返回值会有所不同
} else {

 mCookie = (*env)->GetFieldID(env, myDexFile, "mCookie", "I");
 myOpenDexFile = (*env)->GetStaticMethodID(env,
   myDexFile,
   "openDexFile",
   "(Ljava/lang/String;Ljava/lang/String;I)I");
}
```

Dalvik虚拟机模式下，先加载动态库"libdvm.so"，获取导出的函数调用描述结构体数组dvm_dalvik_system_DexFile的地址，通过函数名称字符串匹配查找获取到函数名称openDexFile对应的函数Dalvik_dalvik_system_DexFile_openDexFile_bytearray的调用地址；打开外壳apk应用的files文件目录下的被保护dex文件 /data/data/<PackageName>/files/dump.dex (其中<PackageName>为外壳apk应用的包名),调用mmap函数将该dex文件映射到私有内存区域并将得到的结果作为调用函数Dalvik_dalvik_system_DexFile_openDexFile_bytearray的传入参数之一，从而实现调用Dalvik_dalvik_system_DexFile_openDexFile_bytearray函数进行Dalvik模式下dex文件的内存加载，得到dex文件加载成功后返回的mCookie值，详细的原理过程后面再逐一分析。

```
/*
 * private static int openDexFile(byte[] fileContents) throws IOException
 *
 * Open a DEX file represented in a byte[], returning a pointer to our
 * internal data structure.
 *
 * The system will only perform "essential" optimizations on the given file.
 *
 * TODO: should be using "long" for a pointer.
 */
```

```
/**
static void Dalvik_dalvik_system_DexFile_openDexFile_bytearray(const u4* args,
    JValue* pResult)
{
    ArrayObject* fileContentsObj = (ArrayObject*) args[0];
    u4 length;
    u1* pBytes;
    RawDexFile* pRawDexFile;
    DexOrJar* pDexOrJar = NULL;

    if (fileContentsObj == NULL) {
        dvmThrowNullPointerException("fileContents == null");
        RETURN_VOID();
    }

    // TODO: Avoid making a copy of the array. (note array *is* modified)
    length = fileContentsObj->length;
    pBytes = (u1*) malloc(length);

    if (pBytes == NULL) {
        dvmThrowRuntimeException("unable to allocate DEX memory");
        RETURN_VOID();
    }

    memcpy(pBytes, fileContentsObj->contents, length);

    if (dvmRawDexFileOpenArray(pBytes, length, &pRawDexFile) != 0) {
        ALOGV("Unable to open in-memory DEX file");
        free(pBytes);
        dvmThrowRuntimeException("unable to open in-memory DEX file");
        RETURN_VOID();
    }

    ALOGV("Opening in-memory DEX");
    pDexOrJar = (DexOrJar*) malloc(sizeof(DexOrJar));
    pDexOrJar->isDex = true;
    pDexOrJar->pRawDexFile = pRawDexFile;
    pDexOrJar->pDexMemory = pBytes;
    pDexOrJar->fileName = strdup("<memory>"); // Needs to be free()able.
    addToDexFileTable(pDexOrJar);

 // #define RETURN_PTR(_val)        do { pResult->l = (Object*)(_val); return; } while(0)
    RETURN_PTR(pDexOrJar);
}
**/

/**
 // jni函数注册的结构体DalvikNativeMethod
 const DalvikNativeMethod dvm_dalvik_system_DexFile[] = {
    { "openDexFileNative",  "(Ljava/lang/String;Ljava/lang/String;I)I",
        Dalvik_dalvik_system_DexFile_openDexFileNative },
    { "openDexFile",        "([B)I",
        Dalvik_dalvik_system_DexFile_openDexFile_bytearray },
    { "closeDexFile",       "(I)V",
        Dalvik_dalvik_system_DexFile_closeDexFile },
    { "defineClassNative",  "(Ljava/lang/String;Ljava/lang/ClassLoader;I)Ljava/lang/Class;",
        Dalvik_dalvik_system_DexFile_defineClassNative },
    { "getClassNameList",   "(I)[Ljava/lang/String;",
        Dalvik_dalvik_system_DexFile_getClassNameList },
    { "isDexOptNeeded",     "(Ljava/lang/String;)Z"
```

```
    { "isDexOptNeeded",     "(Ljava/lang/String;)Z",
        Dalvik_dalvik_system_DexFile_isDexOptNeeded },
    { NULL, NULL, NULL },
};
**/


// 在JNINativeMethod结构体中根据函数名称和函数签名字符串匹配查找到该函数对应的函数调用指针
int lookup(JNINativeMethod *table, const char *name, const char *sig,
  void (**fnPtrout)(u4 const *, union JValue *)) {

 int i = 0;
 while (table[i].name != NULL) {

  LOGI("lookup %d %s", i, table[i].name);

  // 根据函数名称和函数签名匹配查找
  if ((strcmp(name, table[i].name) == 0)
    && (strcmp(sig, table[i].signature) == 0)) {

   // 查找到该函数名称和函数签名对应的函数调用指针
   *fnPtrout = table[i].fnPtr;
   return 1;
  }
  i++;
 }

 return 0;
}


// szPath--dalvik下需要内存加载的dex文件的路径
void myLoadDex_dvm(JNIEnv* env, char* szPath) {

 // 加载动态库"libdvm.so"
 void *ldvm = (void*) dlopen("libdvm.so", 1);
 // 获取动态库"libdvm.so"的导出结构体dvm_dalvik_system_DexFile
 JNINativeMethod* dvm_dalvik_system_DexFile = (JNINativeMethod*) dlsym(ldvm, "dvm_dalvik_system_DexFile");

 // 在dvm_dalvik_system_DexFile结构体中查找到openDexFile函数对应的函数调用地址
 // static void Dalvik_dalvik_system_DexFile_openDexFile_bytearray(const u4* args, JValue* pResult)
 if (0 == lookup(dvm_dalvik_system_DexFile, "openDexFile", "([B)I", &openDexFile)) {

  openDexFile = NULL;
  LOGI("method does not found ");

  return;

 } else {

  LOGI("openDexFile method found ! HAVE_BIG_ENDIAN");
 }

 int handle;
 struct stat buf = { 0 };

 // 打开szPath文件路径指定的dex文件
 handle = open(szPath, 0);
 LOGI("handle:%X\n", handle);
 if (!handle) {
```

```
 LOGI("open dump.dex failed");
 return;
}

// 获取szPath文件路径指定的dex文件的大小
int status = fstat(handle, &buf);
if (status) {

 LOGI("fstat failed");
 return;
}

// 需要加载的szPath文件路径指定的dex文件的长度
int dexLen = buf.st_size;

LOGI("dexLen:%d, st_blksize:%d", dexLen, (int )buf.st_blksize);
//#define PROT_READ 0x1     /* Page can be read.  */
//#define PROT_WRITE  0x2     /* Page can be written.  */
//#define PROT_EXEC 0x4     /* Page can be executed.  */
//#define PROT_NONE 0x0     /* Page can not be accessed.  */

//#define MAP_SHARED  0x01    /* Share changes.  */
//#define MAP_PRIVATE 0x02    /* Changes are private.  */

// 根据szPath文件路径指定的dex文件的大小创建可读可写的私有内存映射
// 实际就是对szPath文件路径指定的dex文件读取操作
char* dexbase = (char*) mmap(0, dexLen, 3, 2, handle, 0);
LOGI("dex magic %c %c %c %c %c %c %c", *dexbase, *(dexbase + 1),
  *(dexbase + 2), *(dexbase + 3), *(dexbase + 4), *(dexbase + 5),
  *(dexbase + 6));

char* arr;
// 参考:http://bbs.pediy.com/thread-197274.htm
// 参考:https://segmentfault.com/a/1190000002578755
// 申请内存空间构建Dalvik_dalvik_system_DexFile_openDexFile_bytearray函数第一个传入参数args
arr = (char*) malloc(16 + dexLen);

ArrayObject *ao = (ArrayObject*) arr;
LOGI("sizeof ArrayObject:%d", sizeof(ArrayObject));

// dex文件的长度
ao->length = dexLen;
// 拷贝dex文件的数据内容到申请内存空间中
memcpy(arr + 16, dexbase, dexLen);
// 取消dex文件读取操作的内存映射
munmap(dexbase, dexLen);

u4 args[] = { (u4) ao };
union JValue pResult;
jint result;

// #define RETURN_PTR(_val)   do { pResult->l = (Object*)(_val); return; } while(0)
if (openDexFile != NULL) {

 // 调用Dalvik_dalvik_system_DexFile_openDexFile_bytearray函数实现dex文件的内存加载
 openDexFile(args, &pResult);
 // 得到dex文件内存加载后的DexOrJar结构体指针即mCookie
 result = (jint) pResult.l;
```

```
  // 保存dex文件内存加载后的mCookie值
  dvm_Cookie = result;

  LOGI("Dalvik Cookie :0x%08x", result);
 }
}
```

Art虚拟机模式下，鉴于Art虚拟机的执行过程比较复杂以及兼容性的考虑，阿里移动早期加固在Art虚拟机模式下实现不像Dalvik虚拟机模式下那么底层，通过Jni下的C++反射调用java层实现的类dalvik.system.DexFile的静态成员方法openDexFile实现Art虚拟机模式下dex文件的加载，函数openDexFile调用成功后返回，得到dex文件加载到内存后的mCookie值，具体的原理过程后面再逐一分析。

```
// szPath--art下需要内存加载的dex文件的路径
void myLoadDex_art(JNIEnv* env, char* szPath) {

 // 将c语言的字符串转换成jni的字符串
 jstring inPath = (*env)->NewStringUTF(env, szPath);

 // art_MarCookie保存dex文件内存加载后的mCookie值
 if (sdk_int > 22) {

  // 调用类dalvik.system.DexFile的静态成员方法openDexFile实现art下dex文件的内存加载
  art_MarCookie = (*env)->CallStaticObjectMethod(env, myDexFile, myOpenDexFile, inPath, 0, 0);

  LOGI("----MarCookie:%p", art_MarCookie);

 } else {

  // 调用类dalvik.system.DexFile的静态成员方法openDexFile实现art下dex文件的内存加载
  art_Cookie = (*env)->CallStaticLongMethod(env, myDexFile, myOpenDexFile, inPath, 0, 0);

  LOGI("----artCookie:%llx", art_Cookie);
 }

 // 加载动态库文件libart.so
 void* dlart = dlopen("libart.so", 1);

 // 获取动态库文件libart.so的导出函数art::DexFile::FindClassDef的调用地址
 // const DexFile::ClassDef* DexFile::FindClassDef(uint16_t type_idx) const
 pArtFun pArtDexFileFindClassDef = (pArtFun) dlsym(dlart, "_ZNK3art7DexFile12FindClassDefEt");

 LOGI("pArtDexFileFindClassDef:%p", pArtDexFileFindClassDef);

}
```

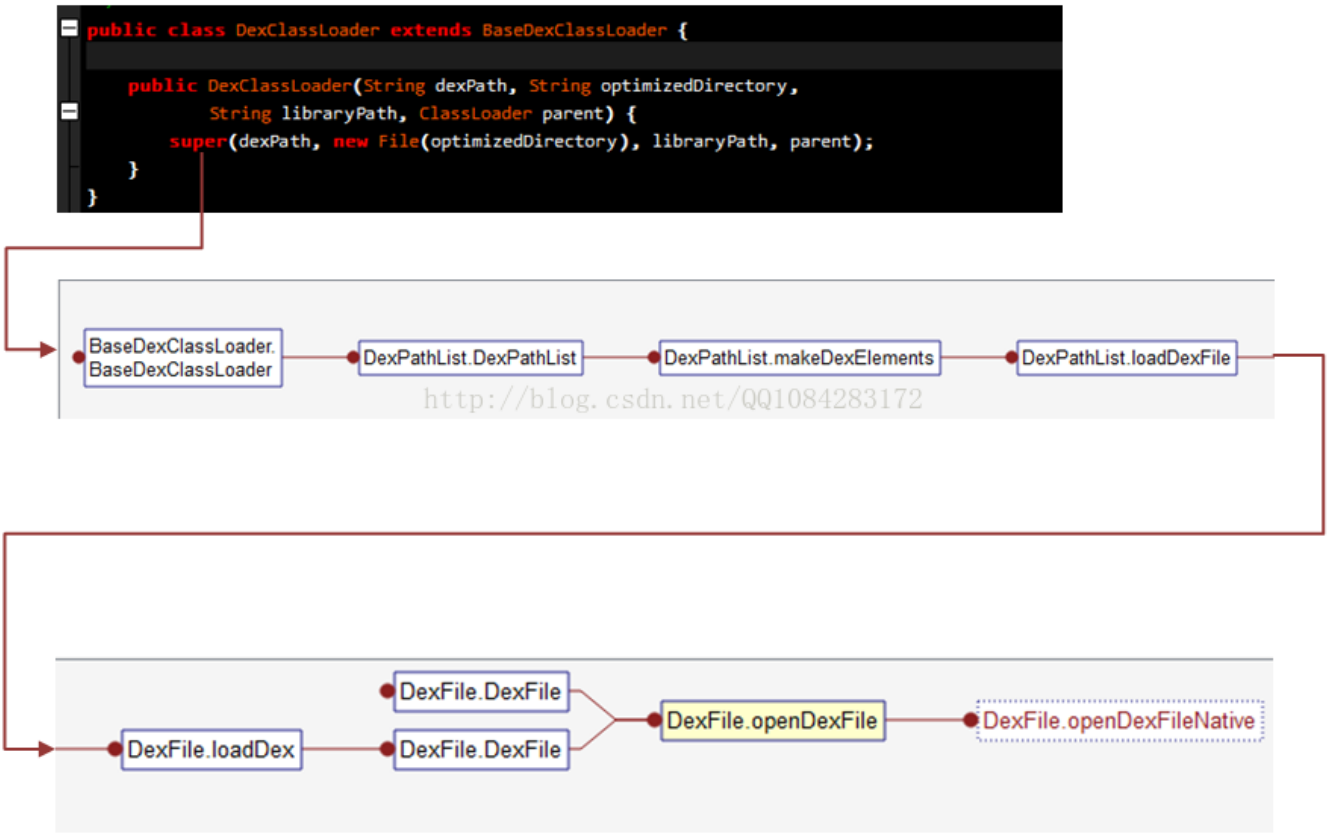
< 9>. 阿里移动早期Android加固Dalvik虚拟机模式和Art虚拟机模式下加载dex文件到内存中的实现细节梳理。

Android加固的原理是基于DexClassLoader加载Android dex文件到内存中实现的，DexClassLoader的java层代码执行流程如下图所示，DexClassLoader调用到Native型声明的**openDexFileNative函数**时，DexClassLoader的整个java层的代码实现已经基本调用完成了，再继续往下执行代码就会调用openDexFileNative函数Jni函数注册对应的Native层实现的本地方法，并且由于Android虚拟机有Dalvik和Art两种运行模式，因此openDexFileNative函数在Dalvik虚拟机模式下和Art虚拟机模式下分别有Jni函数注册对应的不同Native函数实现。

http://androidxref.com/4.4_r1/xref/libcore/dalvik/src/main/java/dalvik/system/DexClassLoader.java

http://androidxref.com/4.4.4_r1/xref/libcore/dalvik/src/main/java/dalvik/system/DexFile.java#301

DexClassLoader的java层代码执行流程示意图：



java层的openDexFileNative函数，在Dalvik虚拟机模式下和Art虚拟模式下分别对应不同的Native函数实现。在Dalvik虚拟机模式下，openDexFileNative函数对应的Native实现函数为遵从Jni函数注册命名规则的**Dalvik_dalvik_system_DexFile_openDexFileNative函数**；在Art虚拟机模式下，openDexFileNative函数对应的Native实现函数为Jni函数动态注册的**DexFile_openDexFileNative函数** 。

http://androidxref.com/4.4.4_r1/xref/dalvik/vm/native/dalvik_system_DexFile.cpp#Dalvik_dalvik_system_DexFile

http://androidxref.com/4.4.4_r1/xref/art/runtime/native/dalvik_system_DexFile.cc#87

openDexFileNative函数在Dalvik和Art下的不同Native实现

```
300
301     native private static int openDexFileNative(String sourceName, String outputName,
302         int flags) throws IOException;
303
```

Full Search openDexFileNative
Definition
Symbol
File Path
History

In Project(s) select all invert selection

abi
art
bionic
bootable
build
cts

Search Clear Help

Searched **full:opendexfilenative** (Results **1 - 3** of **3**) sorted by relevance

/libcore/dalvik/src/main/java/dalvik/system/

HAD DexFile.java
296 return openDexFileNative(new File(sourceName).getCanonicalPath(),
301 native private static int openDexFileNative(String sourceName, String outputName, met

/dalvik/vm/native/

HAD dalvik_system_DexFile.cpp
133 * private static int openDexFileNative(String sourceName, String outputName,
520 { "openDexFileNative", "(Ljava/lang/String;Ljava/lang/String;I)I",

/art/runtime/native/

HAD dalvik_system_DexFile.cc
314 NATIVE_METHOD(DexFile, openDexFileNative, "(Ljava/lang/String;Ljava/lang/String;I)I"),

Dalvik虚拟机模式下，openDexFileNative函数 对应的Native实现函数为**Dalvik_dalvik_system_DexFile_openDexFileNative函数**，并且该函数是遵从Jni函数注册命名规则进行注册的。

http://androidxref.com/4.4.4_r1/xref/dalvik/vm/native/dalvik_system_DexFile.cpp

Dalvik模式下openDexFileNative函数的Native实现

```
518
519 const DalvikNativeMethod dvm_dalvik_system_DexFile[] = {
520     { "openDexFileNative",  "(Ljava/lang/String;Ljava/lang/String;I)I",
521       Dalvik_dalvik_system_DexFile_openDexFileNative },
522     { "openDexFile",        "([B)I",
523       Dalvik_dalvik_system_DexFile_openDexFile_bytearray },
524     { "closeDexFile",       "(I)V",
525       Dalvik_dalvik_system_DexFile_closeDexFile },
526     { "defineClassNative",  "(Ljava/lang/String;Ljava/lang/ClassLoader;I)Ljava/lang/Class;",
527       Dalvik_dalvik_system_DexFile_defineClassNative },
528     { "getClassNameList",   "(I)[Ljava/lang/String;",
529       Dalvik_dalvik_system_DexFile_getClassNameList },
530     { "isDexOptNeeded",     "(Ljava/lang/String;)Z",
531       Dalvik_dalvik_system_DexFile_isDexOptNeeded },
532     { NULL, NULL, NULL },
533 };
```


```
150  */
151 static void Dalvik_dalvik_system_DexFile_openDexFileNative(const u4* args,
152     JValue* pResult)
153 {
154     StringObject* sourceNameObj = (StringObject*) args[0];
155     StringObject* outputNameObj = (StringObject*) args[1];
156     DexOrJar* pDexOrJar = NULL;
157     JarFile* pJarFile;
158     RawDexFile* pRawDexFile;
159     char* sourceName;
160     char* outputName;
161
162     if (sourceNameObj == NULL) {
163         dvmThrowNullPointerException("sourceName == null");
164         RETURN_VOID();
165     }
166
167     sourceName = dvmCreateCstrFromString(sourceNameObj);
168     if (outputNameObj != NULL)
169         outputName = dvmCreateCstrFromString(outputNameObj);
170     else
171         outputName = NULL;
172
```

Art虚拟机模式下，openDexFileNative函数对应的Native实现函数为 **DexFile_openDexFileNative函数** ，并且该函数是Jni动态注册的函数。

http://androidxref.com/4.4.4_r1/xref/art/runtime/native/dalvik_system_DexFile.cc

Art模式下openDexFileNative函数的Native实现

```
static JNINativeMethod gMethods[] = {
  NATIVE_METHOD(DexFile, closeDexFile, "(I)V"),
  NATIVE_METHOD(DexFile, defineClassNative, "(Ljava/lang/String;Ljava/lang/ClassLoader;I)Ljava/lang/Class;"),
  NATIVE_METHOD(DexFile, getClassNameList, "(I)[Ljava/lang/String;"),
  NATIVE_METHOD(DexFile, isDexOptNeeded, "(Ljava/lang/String;)Z"),
  NATIVE_METHOD(DexFile, openDexFileNative, "(Ljava/lang/String;Ljava/lang/String;I)I"),
};

void register_dalvik_system_DexFile(JNIEnv* env) {
  REGISTER_NATIVE_METHODS("dalvik/system/DexFile");
}

}  // namespace art
```

```
86
87 static jint DexFile_openDexFileNative(JNIEnv* env, jclass, jstring javaSourceName, jstring javaOutputName, jint) {
88   ScopedUtfChars sourceName(env, javaSourceName);
89   if (sourceName.c_str() == NULL) {
90     return 0;
91   }
92   std::string dex_location(sourceName.c_str());
93   NullableScopedUtfChars outputName(env, javaOutputName);
94   if (env->ExceptionCheck()) {
95     return 0;
96   }
97   ScopedObjectAccess soa(env);
98
99   uint32_t dex_location_checksum;
100  if (!DexFile::GetChecksum(dex_location, &dex_location_checksum)) {
101    LOG(WARNING) << "Failed to compute checksum: " << dex_location;
102    ThrowLocation throw_location = soa.Self()->GetCurrentLocationForThrow();
103    soa.Self()->ThrowNewExceptionF(throw_location, "Ljava/io/IOException;",
104                                   "Unable to get checksum of dex file: %s", dex_location.c_str());
105    return 0;
106  }
107
108  ClassLinker* linker = Runtime::Current()->GetClassLinker();
109  const DexFile* dex_file;
110  if (outputName.c_str() == NULL) {
111    dex_file = linker->FindDexFileInOatFileFromDexLocation(dex_location, dex_location_checksum);
112  } else {
113    std::string oat_location(outputName.c_str());
114    dex_file = linker->FindOrCreateOatFileForDexLocation(dex_location, dex_location_checksum, oat_location);
115  }
```


Dalvik虚拟机模式下，Android dex文件的加载没有使用openDexFileNative函数 对应的Native实现函数**Dalvik_dalvik_system_DexFile_openDexFileNative**，而选择了使用另外一个Android系统的保留**函数Dalvik_dalvik_system_DexFile_openDexFile_bytearray**来实现dex文件的加载，为什么呢？

在Android加固中，被保护dex文件的加载在任何一个环节中都不能暴露；如果调用函数Dalvik_dalvik_system_DexFile_openDexFileNative进行dex文件的加载就会存在这个问题，但是通过调用Android系统的保留函数Dalvik_dalvik_system_DexFile_openDexFile_bytearray进行dex文件的加载就不会有这种问题存在，具体的原因从这两个函数的调用参数就可以知道了，还可以继续向 函数Dalvik_dalvik_system_DexFile_openDexFile_bytearray 的底层实现进行研究，调用更底层的Native函数来实现dex文件的加载，并解决dex文件比较大时，Android加固的apk应用第一次启动很慢的问题，这里就要提到 梆梆的Android加固实现了，这些问题后面再讨论。

1>. 在调用函数Dalvik_dalvik_system_DexFile_openDexFileNative时，要求使用到dex文件的文件路径，否则容易导致被保护的dex文件的暴露。

```
/*
 * private static int openDexFileNative(String sourceName, String outputName,
 *     int flags) throws IOException
 *
 * Open a DEX file, returning a pointer to our internal data structure.
 *
 * "sourceName" should point to the "source" jar or DEX file.
 *
 * If "outputName" is NULL, the DEX code will automatically find the
 * "optimized" version in the cache directory, creating it if necessary.
 * If it's non-NULL, the specified file will be used instead.
 *
 * TODO: at present we will happily open the same file more than once.
 * To optimize this away we could search for existing entries in the hash
 * table and refCount them.  Requires atomic ops or adding "synchronized"
 * to the non-native code that calls here.
 *
 * TODO: should be using "long" for a pointer.
 */
static void Dalvik_dalvik_system_DexFile_openDexFileNative(const u4* args,
    JValue* pResult)
{
    StringObject* sourceNameObj = (StringObject*) args[0];
    StringObject* outputNameObj = (StringObject*) args[1];
    DexOrJar* pDexOrJar = NULL;
    JarFile* pJarFile;
    RawDexFile* pRawDexFile;
    char* sourceName;
    char* outputName;

    if (sourceNameObj == NULL) {
        dvmThrowNullPointerException("sourceName == null");
        RETURN_VOID();
    }

    sourceName = dvmCreateCstrFromString(sourceNameObj);
    if (outputNameObj != NULL)
        outputName = dvmCreateCstrFromString(outputNameObj);
    else
        outputName = NULL;
```


2>. 而在调用函数Dalvik_dalvik_system_DexFile_openDexFile_bytearray时，使用到的是映射到内存中的dex文件数据的字节数组，可以避免被保护dex文件的暴露。

http://androidxref.com/4.4.4_r1/xref/dalvik/vm/native/dalvik_system_DexFile.cpp#Dalvik_dalvik_system_DexFile

```
/*
 * private static int openDexFile(byte[] fileContents) throws IOException
 *
 * Open a DEX file represented in a byte[], returning a pointer to our
 * internal data structure.
 *
 * The system will only perform "essential" optimizations on the given file.
 *
 * TODO: should be using "long" for a pointer.
 */
static void Dalvik_dalvik_system_DexFile_openDexFile_bytearray(const u4* args,
    JValue* pResult)
{
    ArrayObject* fileContentsObj = (ArrayObject*) args[0];
    u4 length;
    u1* pBytes;
    RawDexFile* pRawDexFile;
    DexOrJar* pDexOrJar = NULL;

    if (fileContentsObj == NULL) {
        dvmThrowNullPointerException("fileContents == null");
        RETURN_VOID();
    }

    /* TODO: Avoid making a copy of the array. (note array *is* modified) */
    length = fileContentsObj->length;
    pBytes = (u1*) malloc(length);

    if (pBytes == NULL) {
        dvmThrowRuntimeException("unable to allocate DEX memory");
        RETURN_VOID();
    }
```


Art虚拟机模式下，鉴于没有一个像Dalvik虚拟机模式下Dalvik_dalvik_system_DexFile_openDexFile_bytearray的函数，并且Art虚拟机模式下只有DexFile_openDexFileNative函数可以实现dex文件的加载，dex文件暴露的问题是存在的，不像Dalvik虚拟机模式下有Dalvik_dalvik_system_DexFile_openDexFile_bytearray这样的内存加载dex文件的函数以及基于Art虚拟机模式的执行复杂，兼容性等考虑，最终**通过Jni的C++反射调用Java层的函数openDexFile** 来实现Art虚拟机模式下的dex文件的加载。关于此种情况，dex文件加载的暴露问题，应该可以通过继续挖掘Native函数DexFile_openDexFileNative的实现或者Native函数的Hook以及结合dex文件的加密处理来解决好。

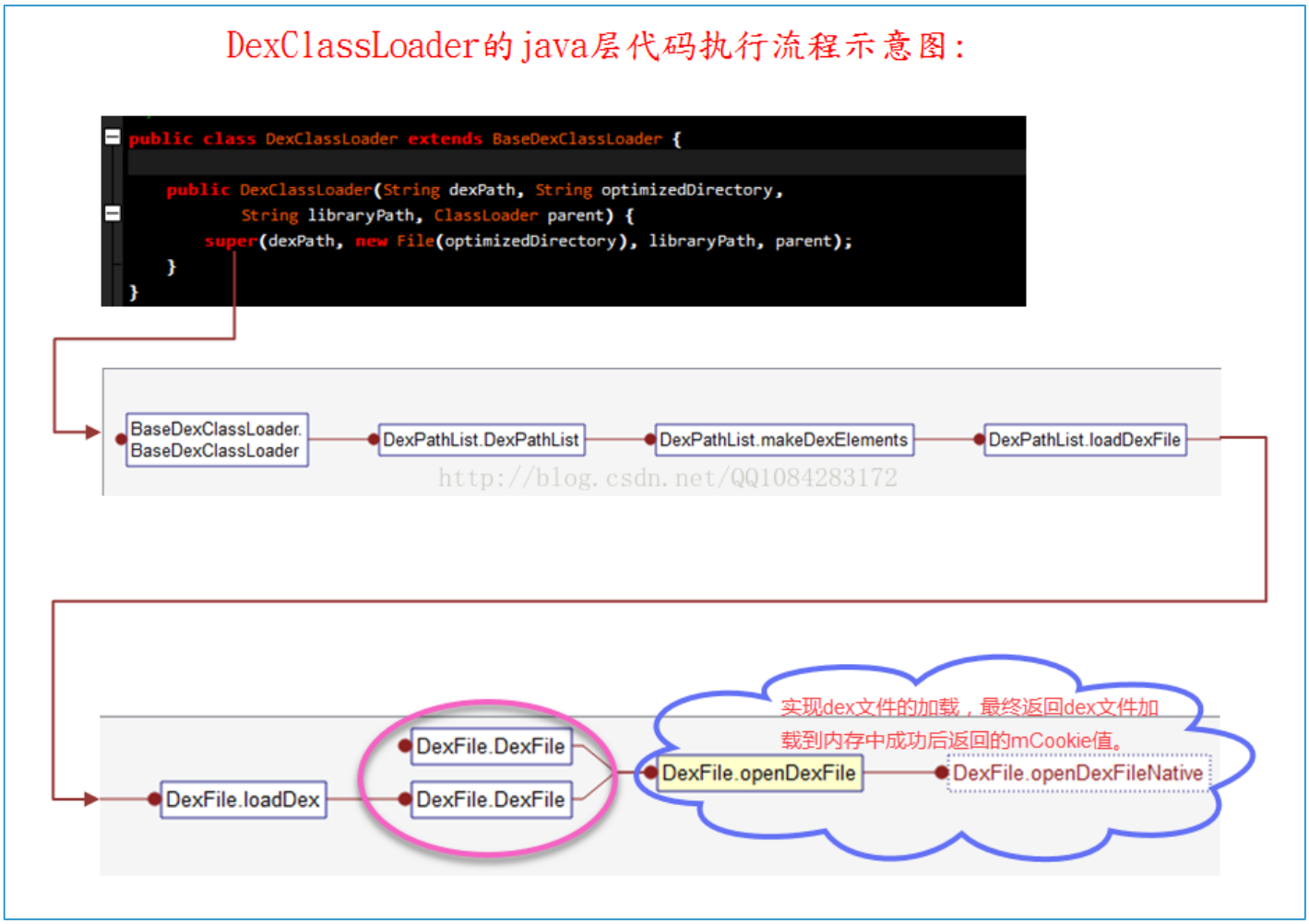
http://androidxref.com/4.4.4_r1/xref/libcore/dalvik/src/main/java/dalvik/system/DexFile.java#294

```
/*
 * Open a DEX file.  The value returned is a magic VM cookie.  On
 * failure, an IOException is thrown.
 */
private static int openDexFile(String sourceName, String outputName,
    int flags) throws IOException {
    return openDexFileNative(new File(sourceName).getCanonicalPath(),
                             (outputName == null) ? null : new File(outputName).getCanonicalPath(),
                             flags);
}

native private static int openDexFileNative(String sourceName, String outputName,
    int flags) throws IOException;
```


< 10 >.在ClassLoader中用加载到内存后的被保护dex文件的mCookie值替换掉外壳apk应用的dex文件的mCookie值。

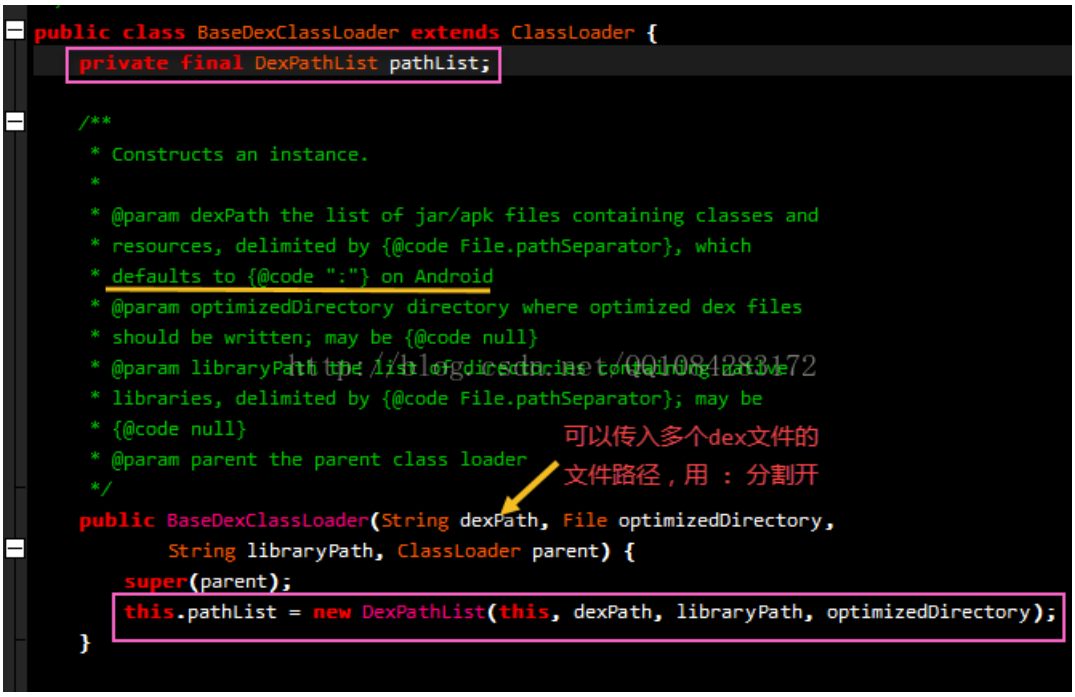
```
// 在ClassLoader中用内存加载的dex文件的mCookie值替换掉外壳apk应用的mCookie值(important)
replace_classloader_cookie(env, classLoader);
LOGI("enter new application");
```


在分析mCookie值替换的代码之前，先回顾一下前面整理的DexClassLoader的java层代码的执行流程示意图如下图所示，DexFile.openDexFile函数往下调用的底层代码的作用是实现对dex文件加载到内存中返回mCookie值，我们已经帮助Android系统实现了（下图蓝色标出的部分）。尽管被保护的dex文件已经加载到进程内存中，但是还没有和当前外壳apk进程进行连接起来，只有替换掉当前外壳apk应用进程的dex文件的mCookie值为被保护dex文件的mCookie值才能实现被保护的dex文件和当前外壳apk应用进程联系起来，当前外壳apk应用进程执行代码时才是执行的被保护的dex文件的代码。每一个Android应用在启动运行的时候，都会有一个对应的ActivityThread对象实例来管理，dex文件代码的执行都是有ActivityThread对象来管理、操作实现的，dex文件都是以ClassLoader对象实例的方式被ActivityThread对象实例进行管理、操作的，最终执行该dex文件的代码，经过对DexClassLoader的java层代码的执行流程的分析发现，每个DexClassLoader对象实例对dex文件的管理都是有一个维护的DexPathList的列表， DexPathList列表记录了当前进程加载的每一个dex文件相关的描述信息，其中描述dex文件加载到内存中的一个重要结构体就是DexFile，DexFile的成员变量之一就是mCookie值，因此只要将此处的外壳apk应用的DexFile的成员变量mCookie值修改为被保护dex文件的mCookie值就是实现当前外壳apk应用进程与被保护dex文件的代码联系起来被Android虚拟机执行。



说的明白一点，每一个Android应用进程是由一个对应的Activity对象实例来管理的，该Activity对象实例对Android应用代码执行的管理其实就是对Android应用的dex文件加载后的DexClassLoader对象实例的管理，Android应用代码的执行都是由DexClassLoader对象实例来提供的，然后交给Android虚拟机进行执行。DexClassLoader对象实例就是dex文件加载到内存后的管理描述信息结构体。

DexClassLoader对象实例的构造是通过调用父类BaseDexClassLoader的构造函数实现的，类BaseDexClassLoade的构造函数创建一个DexPathList类型的实例成员变量pathList，用于对当前apk应用进程加载的dex文件进行维护和管理。



类DexPathList中由实例成员变量dexElements来实现对加载到内存中的dex文件信息进行管理，类DexPathList的构造函数调用函数makeDexElements得到指定dex文件路径的dex文件加载到内存中的描述信息的Elements数组列表dexElements中。

```
public DexPathList(ClassLoader definingContext, String dexPath,
        String libraryPath, File optimizedDirectory) {
    if (definingContext == null) {
        throw new NullPointerException("definingContext == null");
    }

    if (dexPath == null) {
        throw new NullPointerException("dexPath == null");
    }

    if (optimizedDirectory != null) {
        if (!optimizedDirectory.exists())  {
            throw new IllegalArgumentException(
                    "optimizedDirectory doesn't exist: "
                    + optimizedDirectory);
        }

        if (!(optimizedDirectory.canRead()
                        && optimizedDirectory.canWrite())) {
            throw new IllegalArgumentException(
                    "optimizedDirectory not readable/writable: "
                    + optimizedDirectory);
        }
    }

    this.definingContext = definingContext;
    ArrayList<IOException> suppressedExceptions = new ArrayList<IOException>();

    // private final Element[] dexElements;
    // Makes an array of dex/resource path elements
    this.dexElements = makeDexElements(splitDexPath(dexPath), optimizedDirectory,
                                       suppressedExceptions);
    if (suppressedExceptions.size() > 0) {
        this.dexElementsSuppressedExceptions =
            suppressedExceptions.toArray(new IOException[suppressedExceptions.size()]);
    } else {
        dexElementsSuppressedExceptions = null;
    }

    // 获取当前进程运行dex文件所需要加载的so库文件所在的文件夹数组
    this.nativeLibraryDirectories = splitLibraryPath(libraryPath);
}
```


类DexPathList的非静态成员函数makeDexElements调用loadDexFile函数将dex文件加载到内存中，返回得到dex文件加载到内存中的描述结构体DexFile的对象实例，使用dex文件加载到内存中的描述结构体DexFile的对象实例作为传入参数之一调用 类Element的构造函数构建**Element** 元素的对象实例，然后将加载到内存中dex文件的这些信息存放到dexElements数组列表中。

```
private static Element[] makeDexElements(ArrayList<File> files, File optimizedDirectory,
                                         ArrayList<IOException> suppressedExceptions) {
    ArrayList<Element> elements = new ArrayList<Element>();
    /*
     * Open all files and load the (direct or contained) dex files
     * up front.
     */
    for (File file : files) {
        File zip = null;
        DexFile dex = null;
        String name = file.getName();

        // ".dex"文件
        if (name.endsWith(DEX_SUFFIX)) {
            // Raw dex file (not inside a zip/jar).
            try {

                dex = loadDexFile(file, optimizedDirectory);
            } catch (IOException ex) {
                System.logE("Unable to load dex file: " + file, ex);
            }
        } else if (name.endsWith(APK_SUFFIX) || name.endsWith(JAR_SUFFIX)
                || name.endsWith(ZIP_SUFFIX)) {
            zip = file;

            try {
                dex = loadDexFile(file, optimizedDirectory);
            } catch (IOException suppressed) {
                suppressedExceptions.add(suppressed);
            }
        } else if (file.isDirectory()) {
            // We support directories for looking up resources.
            // This is only useful for running libcore tests.
            elements.add(new Element(file, true, null, null));
        } else {
            System.logW("Unknown file type for: " + file);
        }

        if ((zip != null) || (dex != null)) {
            elements.add(new Element(file, false, zip, dex));
        }
    }

    return elements.toArray(new Element[elements.size()]);
}
```

dex文件加载到内存中返回的java层描述结构体

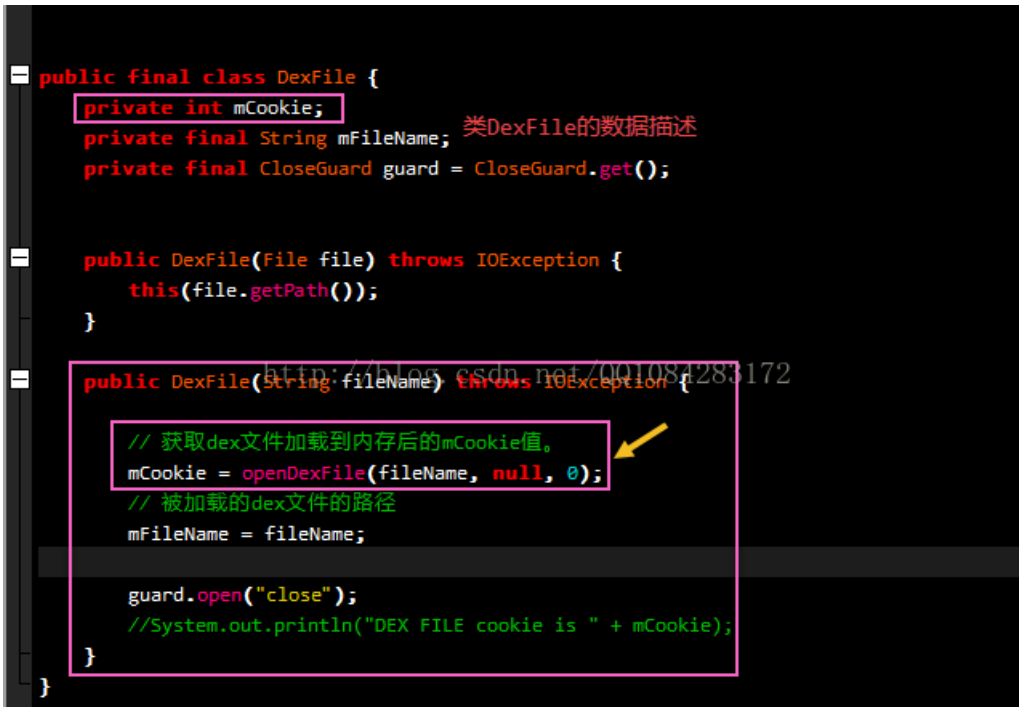
从上面的代码中可以了解到dex文件的真正加载是由 **loadDexFile函数** 开始的，并且dex文件加载到内存中成功以后是由DexFile结构体对象来描述的。在进行dex文件加载时，可以指定dex文件优化后的dex文件存放目录也可以不指定使用Android系统默认的优化后dex文件存放目录，但是这两种情况最终都是调用**openDexFile函数**实现dex文件的加载。下面以不指定优化后dex文件存放目录使用Android系统默认的这种情况来进行分析，loadDexFile函数调用类DexFile的构造函数实现dex文件的加载并获取dex文件加载到内存后的描述结构体DexFile的对象实例。

```java
/**
 * Constructs a {@code DexFile} instance, as appropriate depending
 * on whether {@code optimizedDirectory} is {@code null}.
 */
private static DexFile loadDexFile(File file, File optimizedDirectory)
        throws IOException {

    // 两种情况，最终都是调用的openDexFile函数
    if (optimizedDirectory == null) {
        // 没有指定dex文件优化后的dex文件的存放目录
        return new DexFile(file);
    } else {
        // 指定了dex文件优化后的dex文件的存放目录
        String optimizedPath = optimizedPathFor(file, optimizedDirectory);
        return DexFile.loadDex(file.getPath(), optimizedPath, 0);
    }
}
```


类Element 的数据属性描述和构造函数的实现如下图所示，类Element 的数据属性描述中比较重要的就是非静态成员变量dexFile和file，其中dexFile成员变量为dex文件加载到Android进程内存中后返回的Java层信息描述结构体DexFile的实例对象。

```java
/**
 * Element of the dex/resource file path
 */
/*package*/ static class Element {
    private final File file;
    private final boolean isDirectory;
    private final File zip;
    private final DexFile dexFile;

    private ZipFile zipFile;
    private boolean initialized;

    // 类Element的构造函数
    public Element(File file, boolean isDirectory, File zip, DexFile dexFile) {
        this.file = file;
        this.isDirectory = isDirectory;
        this.zip = zip;
        // dex文件加载到内存中的java层描述结构体对象
        this.dexFile = dexFile;
    }
```


类DexFile的构造函数最终调用openDexFile函数进行dex文件的加载，dex文件加载到内存成功返回得到dex文件加载到内存的信息描述mCookie值。Java层描述dex文件加载到内存中的描述结构体为类**DexFile** ，类DexFile的数据属性描述成员变量有3个，其中我们比较关注的是成员变量mCookie和mFileName，**mCookie** 是由dex文件加载到Android进程的内存中返回的信息描述结构体**DexOrJar**的指针，**mFileName** 为被加载的dex文件的文件路径字符串。因此，对于Android加固的外壳apk应用和被保护的dex文件而言，只要将外壳apk应用的java层描述结构体DexFile实例对象的成员变量mCookie值修改为被保护的dex文件的mCookie值就是实现被保护dex文件的代码执行和外壳apk应用的代码执行无缝的衔接，意思就是外壳apk应用进程将会执行被保护dex文件的代码。



替换外壳apk应用进程的mCookie值为被保护dex文件的mCookie值的步骤梳理：

```
// 替换外壳apk应用进程的mCookie值为被保护dex文件的mCookie值的步骤：

// 通过jni反射调用--C层调用java层代码的方法。

// 1.获取到外壳apk应用进程的ClassLoader实例对象的私有非静态成员变量pathList（dalvik.system.DexPathList类型）的实例对象；

// 2.获取到外壳apk应用进程的实例对象pathList的私有非静态成员变量dexElements（dalvik.system.DexPathList$Element Element[]）；

// 3.遍历外壳apk应用进程的实例对象Element数组dexElements的每一个Element实例对象，获取Element实例对象的私有非静态成员dexFile，
//   dexFile为dalvik.system.DexFile类型；

// 4.设置外壳apk应用进程的每一个Element实例对象（一个）的私有非静态成员dexFile对象实例的私有非静态成员mCookie值为被保护
// dex文件内存加载后的mCookie值，实现外壳apk应用与被保护dex文件的无缝衔接。
```

将外壳apk应用的ClassLoader对应的dex文件mCookie值修改为被保护的dex文件内存加载后的mCookie值。

```
if (sdk_int > 13) {

 if (!myFindClass(env, &myBaseDexClassLoader, "dalvik/system/BaseDexClassLoader")) {

  LOGI("ERROR:myBaseDexClassLoader");
  return;
 }
 // 获取类dalvik.system.BaseDexClassLoader的非静态成员变量pathList的获取id
 BaseDexClassLoader_pathList = (*env)->GetFieldID(env,
   myBaseDexClassLoader,
   "pathList",
   "Ldalvik/system/DexPathList;");

 if (!myFindClass(env, &myDexPathList, "dalvik/system/DexPathList")) {
```

```
 LOGI("ERROR:myDexPathList");
 return;
}
// 获取类dalvik.system.DexPathList的非静态成员变量dexElements的获取id
DexPathList_dexElements = (*env)->GetFieldID(env,
  myDexPathList,
  "dexElements",
  "[Ldalvik/system/DexPathList$Element;");

if (!myFindClass(env, &myElement, "dalvik/system/DexPathList$Element")) {

 LOGI("ERROR:myElement");
 return;
}
// 获取类dalvik.system.DexPathList$Element的非静态成员变量dexFile的获取id
DexPathList_Element_dexFile = (*env)->GetFieldID(env,
  myElement,
  "dexFile",
  "Ldalvik/system/DexFile;");

if (sdk_int > 22) //6.0

 // Android 6.0版本,获取类dalvik.system.DexPathList$Element的非静态成员变量dir的获取id
 DexPathList_Element_file = (*env)->GetFieldID(env,
   myElement,
   "dir",
   "Ljava/io/File;");
else
 // 其他版本,获取类dalvik.system.DexPathList$Element的非静态成员变量file的获取id
 DexPathList_Element_file = (*env)->GetFieldID(env,
   myElement,
   "file",
   "Ljava/io/File;");

if (!myFindClass(env, &myFile, "java/io/File")) {

 LOGI("ERROR:myFile");
 return;
}
// 获取类java.io.File的非静态成员方法getAbsolutePath的调用id
myFile_getAbsolutePath = (*env)->GetMethodID(env,
  myFile,
  "getAbsolutePath",
  "()Ljava/lang/String;");
LOGI("PathClassLoader end");

// 获取类的dalvik.system.DexFile的全局引用
if (!myFindClass(env, &myDexFile, "dalvik/system/DexFile")) {

 LOGI("ERROR:myDexFile");
 return;
}
// mCookie值获取需要的函数
if (sdk_int > 22) {

 // 获取类dalvik.system.DexFile的非静态成员变量mCookie的获取id
 mCookie = (*env)->GetFieldID(env, myDexFile, "mCookie", "Ljava/lang/Object;");

 // 获取类dalvik.system.DexFile的静态成员方法openDexFile的调用id
 myOpenDexFile = (*env)->GetStaticMethodID(env,
```

```
  myOpenDexFile = (*env)->GetStaticMethodID(env,
    myDexFile,
    "openDexFile",
    "(Ljava/lang/String;Ljava/lang/String;I)Ljava/lang/Object;");

 // 其它系统版本，获取类dalvik.system.DexFile的成员变量mCookie的获取id和函数openDexFile的调用id
 } else if (sdk_int > 19) {

  mCookie = (*env)->GetFieldID(env, myDexFile, "mCookie", "J");
  myOpenDexFile = (*env)->GetStaticMethodID(env,
    myDexFile,
    "openDexFile",
    "(Ljava/lang/String;Ljava/lang/String;I)J");

 // 主要不同的Android系统版本的openDexFile函数的函数返回值会有所不同
 } else {

  mCookie = (*env)->GetFieldID(env, myDexFile, "mCookie", "I");
  myOpenDexFile = (*env)->GetStaticMethodID(env,
    myDexFile,
    "openDexFile",
    "(Ljava/lang/String;Ljava/lang/String;I)I");
 }
} //

// classLoader--外壳apk应用的类加载器classLoader
void replace_classloader_cookie(JNIEnv *env, jobject classLoader) {

 if (sdk_int > 13) {

  // "java/lang/ClassLoader"-->pathList
  // 类dalvik.system.BaseDexClassLoader实例对象的非静态成员pathList
  // 成员pathList实例对象为dalvik.system.DexPathList类型
  jobject v28 = (*env)->GetObjectField(env, classLoader, BaseDexClassLoader_pathList);

  // pathList-->Element数组
  // 获取类dalvik.system.DexPathList实例对象的非静态成员dexElements
  // 成员dexElements为dalvik.system.DexPathList$Element类型数组
  jobject element_array = (*env)->GetObjectField(env, v28, DexPathList_dexElements);

  // 获取Element数组的长度
  int count = (*env)->GetArrayLength(env, element_array);
  //count一直是1
  LOGI("element count: %d", count);

  int i = 0;
  // 遍历Element数组的元素
  while (i < count) {

   // 获取Element数组的第i个数组元素
   jobject Element = (*env)->GetObjectArrayElement(env, element_array, i);

   // 获取类dalvik.system.DexPathList$Element的实例对象的非静态成员dir/file的值
   // 成员dir/file的类类型为java.io.File
   jobject fileclazz = (*env)->GetObjectField(env, Element, DexPathList_Element_file);

   // 调用类java.io.File实例对象的非静态成员方法getAbsolutePath()函数
   // 获取到外壳apk应用的dex文件路径jni字符串
   jobject v20 = (*env)->CallObjectMethod(env, fileclazz, myFile_getAbsolutePath);
   // 将jni类型的路径字符串转换成C语言的路径字符串
```

```
   const char* str = (*env)->GetStringUTFChars(env, v20, 0);

   //android 6.0下str为:/
   // 打印外壳apk应用的dex文件的加载路径
   LOGI("element is %s", str);

   /*int length = ((*env)->GetStringUTFLength)(env, v20);
    int cmpstatus = strncasecmp("apk", (length - 3 + str), 3);
    ((*env)->ReleaseStringUTFChars)(env, v20, str);*/

   // 获取Element实例对象的非静态成员dexFile
   // dexFile为dalvik.system.DexFile类型的
   jobject DexFile = (*env)->GetObjectField(env, Element, DexPathList_Element_dexFile);

   // 修改实例对象Element的非静态成员dexFile的mCookie值为dex文件内存加载返回的mCookie值
   // 到此替换外壳apk程序的mCookie值成功
   if (sdk_int <= 19) {

    LOGI("SetIntField > 19");
    LOGI("---dvm_Cookie:%X", dvm_Cookie);
    (*env)->SetIntField(env, DexFile, mCookie, dvm_Cookie);

   } else if (sdk_int > 22) {

    LOGI("SetObjectField > 22");
    LOGI("---art_MarCookie:%X", art_MarCookie);
    (*env)->SetObjectField(env, DexFile, mCookie, art_MarCookie);

   } else {

    LOGI("SetLongField others");
    LOGI("----artCookie:%llx", art_Cookie);
    (*env)->SetLongField(env, DexFile, mCookie, art_Cookie);

   }
   /*if(!cmpstatus)
    {
      break;
    }*/
   i++;
  }

  LOGI("exit replace_classloader_cookie");
 }

}
```

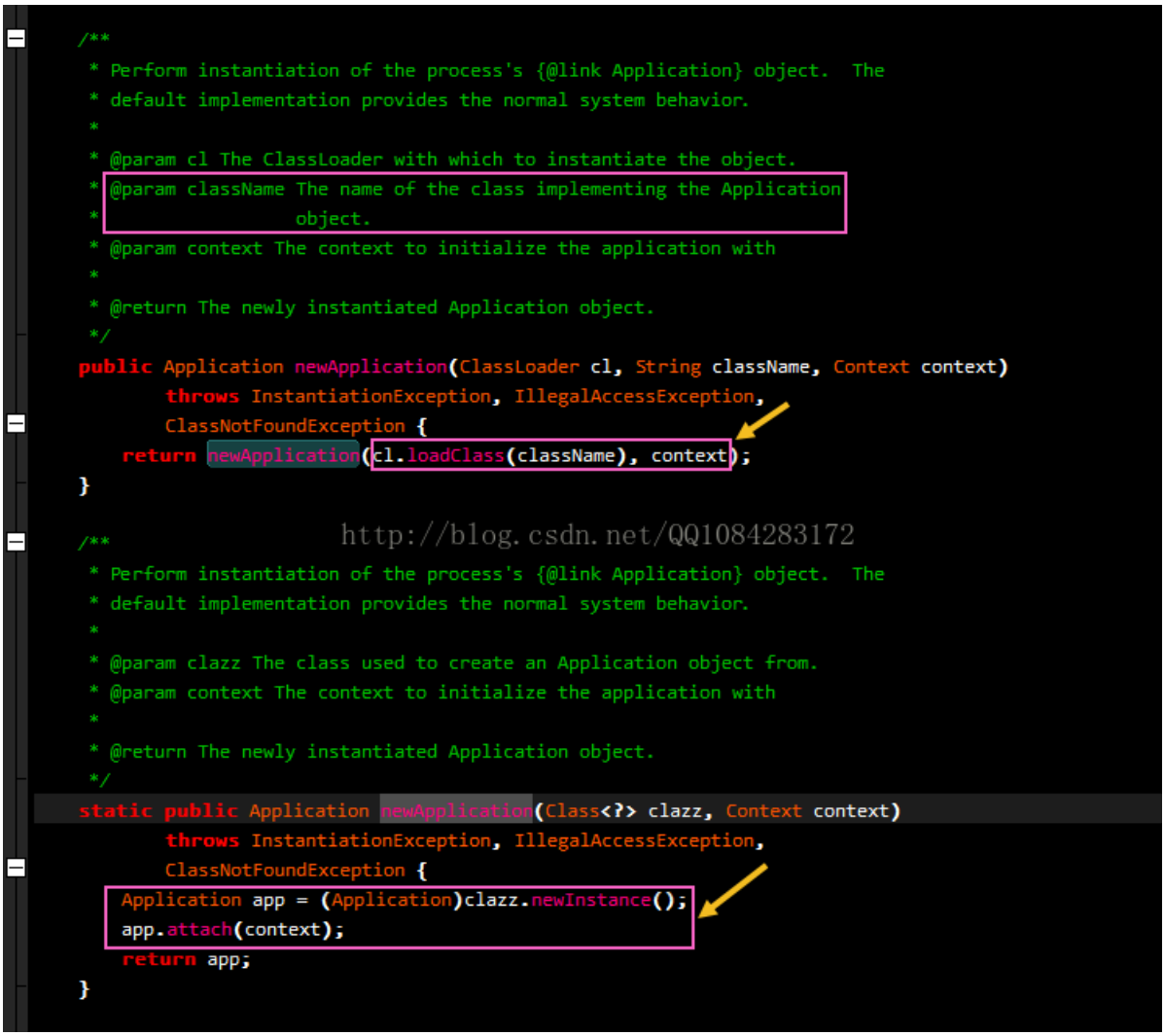
< 11 >. 将外壳apk应用的mCookie值替换为被保护dex文件的mCookie值之后，被保护dex文件的代码就会被执行。一般情况下，Android应用在执行代码时首先执行Applicaiton类里的代码，具体的原理需要参考Android源码中Android应用的启动过程部分；如果Android应用继承实现了Applicaiton类则执行Applicaiton的子类的成员方法，如果Android应用没有继承实现Applicaiton类则默认调用Android 系统提供的android.app.Application类的成员方法。在这里，**由于被保护的dex文件没有继承实现Android系统的类android.app.Application**，因此这里调用的是被保护dex文件的android.app.Application类的成员方法，如果被保护dex文件继承实现了android.app.Application类则调用的是Application子类的成员方法，例如：attach函数，onCreate函数。

调用外壳apk应用的ClassLoader对象实例的非静态成员方法loadClass函数（此时的ClassLoader对象实例已经被替换为被保护dex文件的了），加载和初始化被保护dex文件的android.app.Application类，调用被保护dex文件的android.app.Application类的构造函数<init>，创建Application类的对象实例，然后在外壳apk应用的StubApplication类实例方法attachBaseContext实现中，调用被保护dex文件的实例成员方法attach函数。

```
    /*
     * 构建java字符串"android.app.Application"即被保护dex文件的Application类；
     * 注意:这儿的Application类为被加固保护dex文件的Application类，由于当前被保护的dex文件没有
     * 继承实现android.app.Application类，因此这里使用的字符串是"android.app.Application"；如果
     * 被保护的dex文件有继承实现Android系统的android.app.Application类，则通过解析外壳apk应用的
     * AndroidMainfest.xml文件或者从其他配置文件中来获取被保护dex文件实际的Application类。
     */

    jstring newapp = (*env)->NewStringUTF(env, "android.app.Application");

    // android 5+以上无法用findClass找到android.app.Application类
    if (!myFindClass(env, &myClassLoader, "java/lang/ClassLoader")) {

        LOGI("ERROR:myClassLoader");
        return;
    }

    // 获取类java.lang.ClassLoader的非静态方法loadClass的调用id
    classLoader_findClass = (*env)->GetMethodID(env, myClassLoader,
            "loadClass", "(Ljava/lang/String;)Ljava/lang/Class;");

    // 调用ClassLoader的实例方法loadClass加载被保护dex文件的Application类
    jobject findclass_classz = (*env)->CallObjectMethod(env,
            // classLoader为外壳apk应用的ClassLoader实例对象
            classLoader,
            classLoader_findClass,
            // 加载被保护dex文件的Application类
            newapp);
    if (!findclass_classz) {

        LOGI("can't findClass realAppClass");
        return;
    }

    // 获取被保护dex文件的Application类的构造函数<init>的调用id
    jmethodID initMethod = (*env)->GetMethodID(env, findclass_classz, "<init>", "()V");

    // 调用被保护dex文件的Application类的构造函数创建Application类实例对象
    onCreateObj = (*env)->NewObject(env, findclass_classz, initMethod);
```

这儿一定要注意啊:如果被加固保护的dex文件继承实现了类android.app.Application，则这里需要修改为该继承Application子类的类名称字符串。

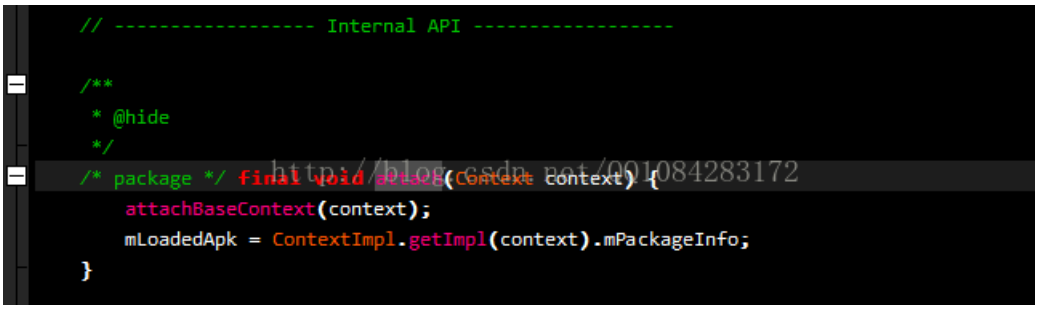
从Android系统源码中来查找被保护dex文件Applicaiton类这样创建和调用attach函数的原因，详细原因可以参考老罗的博客《Android系统默认Home应用程序（Launcher）的启动过程源代码分析》进行研究，后面有精力再分析一下Android应用的整个启动过程。



```
    /**
     * Perform instantiation of the process's {@link Application} object.  The
     * default implementation provides the normal system behavior.
     *
     * @param cl The ClassLoader with which to instantiate the object.
     * @param className The name of the class implementing the Application
     *                  object.
     * @param context The context to initialize the application with
     *
     * @return The newly instantiated Application object.
     */
    public Application newApplication(ClassLoader cl, String className, Context context)
            throws InstantiationException, IllegalAccessException,
            ClassNotFoundException {
        return newApplication(cl.loadClass(className), context);
    }

    /**
     * Perform instantiation of the process's {@link Application} object.  The
     * default implementation provides the normal system behavior.
     *
     * @param clazz The class used to create an Application object from.
     * @param context The context to initialize the application with
     *
     * @return The newly instantiated Application object.
     */
    static public Application newApplication(Class<?> clazz, Context context)
            throws InstantiationException, IllegalAccessException,
            ClassNotFoundException {
        Application app = (Application)clazz.newInstance();
        app.attach(context);
        return app;
    }
```


android.app.Application类的非静态成员方法attach函数的实现如下，在attach函数里已经调用了attachBaseContext函数。

http://androidxref.com/4.4.4_r1/xref/frameworks/base/core/java/android/app/Application.java



```
    // ------------------ Internal API ------------------

    /**
     * @hide
     */
    /* package */ final void attach(Context context) {
        attachBaseContext(context);
        mLoadedApk = ContextImpl.getImpl(context).mPackageInfo;
    }
```


具体的实现代码如下：

```
/*
 * 构建java字符串"android.app.Application"即被保护dex文件的Application类；
 * 注意:这儿的Application类为被加固保护dex文件的Application类，由于当前被保护的dex文件没有
 * 继承实现android.app.Application类，因此这里使用的字符串是"android.app.Application"；如果
 * 被保护的dex文件有继承实现Android系统的android.app.Application类，则通过解析外壳apk应用的
 * AndroidManifest.xml文件或者从其他配置文件中来获取被保护dex文件实际的Application类。
```

```
   AndroidManifest.xml文件或者从其他配置文件中来获取被保护dex文件类所的Application类。
 */

jstring newapp = (*env)->NewStringUTF(env, "android.app.Application");

// android 5+以上无法用findClass找到android.app.Application类
if (!myFindClass(env, &myClassLoader, "java/lang/ClassLoader")) {

 LOGI("ERROR:myClassLoader");
 return;
}

// 获取类java.lang.ClassLoader的非静态方法loadClass的调用id
classLoader_findClass = (*env)->GetMethodID(env, myClassLoader,
  "loadClass", "(Ljava/lang/String;)Ljava/lang/Class;");

// 调用ClassLoader的实例方法loadClass加载被保护dex文件的Application类
jobject findclass_classz = (*env)->CallObjectMethod(env,
  // classLoader为外壳apk应用的ClassLoader实例对象
  classLoader,
  classLoader_findClass,
  // 加载被保护dex文件的Application类
  newapp);
if (!findclass_classz) {

 LOGI("can't findClass realAppClass");
 return;
}

// 获取被保护dex文件的Application类的构造函数<init>的调用id
jmethodID initMethod = (*env)->GetMethodID(env, findclass_classz, "<init>", "()V");

// 调用被保护dex文件的Application类的构造函数创建Application类实例对象
onCreateObj = (*env)->NewObject(env, findclass_classz, initMethod);

if (!myFindClass(env, &myApplication, "android/app/Application")) {
   LOGI("ERROR:myApplication");
   return;
}
// 获取类android.app.Application的非静态成员函数onCreate的调用id
Application_onCreate = (*env)->GetMethodID(env,
  myApplication,
  "onCreate",
  "()V");

// 获取类android.app.Application的非静态成员函数attach的调用id
Application_attach = (*env)->GetMethodID(env,
  myApplication,
  "attach",
  "(Landroid/content/Context;)V");

// 调用被保护dex文件的Application类实例的attach()函数
(*env)->CallVoidMethod(env, onCreateObj, Application_attach, ctx);

// 创建被保护dex文件的Application类的实例对象的全局引用
if (onCreateObj) {

 onCreateObj = (*env)->NewGlobalRef(env, onCreateObj);
}
```

```
 LOGI("enter realAppClass");
}
```

4. 外壳apk应用类StubApplication的成员函数onCreate对应的Native层实现函数native_onCreate的实现代码如下，在函数native_onCreate中最主要的操作就是将外壳apk应用的Application类（外壳apk应用实现了Application类，其继承实现子类为StubApplication）对象实例替换为被保护dex文件的Application类对象实例（被保护dex文件没有实现Application类，如果有Application子类则替换为该Application子类）。因此这里就是将外壳apk应用的ActivityThread中使用到的外壳apk应用Application子类StubApplication对象实例全部修改替换为被保护dex文件的Application类对象实例，然后调用被保护dex文件的Application类对象实例的非静态成员onCreate函数。

http://androidxref.com/4.4.4_r1/xref/frameworks/base/core/java/android/app/ActivityThread.java

http://androidxref.com/4.4.4_r1/xref/frameworks/base/core/java/android/app/LoadedApk.java

```
// 2.然后被调用
// 参考源码: /frameworks/base/core/java/android/app/LoadedApk.java
// 参考源码: /frameworks/base/core/java/android/app/ActivityThread.java
// 此处的 obj 为类StubApplication的实例对象
void native_onCreate(JNIEnv *env, jobject obj) {

 // 调用类android.app.ActivityThread的静态成员方法currentActivityThread()
 // 获取当前外壳apk应用的android.app.ActivityThread currentActivityThread
 jobject theCurrentActivityThread = (*env)->CallStaticObjectMethod(env,
   ActivityThread, currentActivityThread);

 //final ArrayMap<String, WeakReference<LoadedApk>> mPackages= new ArrayMap<String, WeakReference<LoadedApk
 // 获取类android.app.ActivityThread实例对象currentActivityThread的非静态成员变量mPackages
 // mPackages为android.util.ArrayMap类型
 jobject arraymap_mPackages = (*env)->GetObjectField(env,
   theCurrentActivityThread, mPackages);

 // 将C语言形式的外壳apk应用的包名字符串转换成jni类型的字符串
 jstring thePackagename = (*env)->NewStringUTF(env, mPackageName);
 LOGI("mPackageName %s", mPackageName);

 // 调用类实例对象mPackages(android.util.ArrayMap类型)的非静态成员方法get()函数
 // 获取外壳apk应用的包名对应的LoadedApk类对象实例
 jobject v9 = (*env)->CallObjectMethod(env, arraymap_mPackages, ArrayMap_get,
   thePackagename);
 // 获取LoadedApk类对象实例的弱引用（WeakReference<LoadedApk>）
 // final ArrayMap<String, WeakReference<LoadedApk>> mPackages
 jobject v15 = (*env)->CallObjectMethod(env, v9, WeakReference_get);

 // 修改类对象LoadedApk实例的非静态成员 mApplication 的值为被保护dex文件的类Application实例对象
 (*env)->SetObjectField(env, v15, LoadedApk_mApplication, onCreateObj);

 // 修改类对象currentActivityThread实例非静态成员 mInitialApplication 的值
 // 为被保护dex文件的类Application实例对象
 (*env)->SetObjectField(env, theCurrentActivityThread, mInitialApplication,
   onCreateObj);

 // AppBindData mBoundApplication;
 // 获取类对象currentActivityThread实例的非静态成员mBoundApplication的值
 // 该成员mBoundApplication的类类型为android.app.ActivityThread$AppBindData
```

```
 jobject v16 = (*env)->GetObjectField(env, theCurrentActivityThread,
   mBoundApplication);

 // AppBindData_info=(*env)->GetFieldID(env, AppBindData, "info", "Landroid/app/LoadedApk;");
 // 获取类对象mBoundApplication的非静态成员info的值
 // 成员info的类型为android.app.LoadedApk
 jobject v17 = (*env)->GetObjectField(env, v16, AppBindData_info);
 // 修改mBoundApplication.info的成员mApplication为被保护dex文件的类Application实例对象
 // 其中mApplication为android.app.Application类型
 (*env)->SetObjectField(env, v17, LoadedApk_mApplication, onCreateObj);


 // 获取类对象currentActivityThread实例的非静态成员mAllApplications
 // 其中mAllApplications为Ljava/util/ArrayList类型
 // final ArrayList<Application> mAllApplications
 jobject allApplications = (*env)->GetObjectField(env, theCurrentActivityThread,
   mAllApplications);
 // 调用类对象mAllApplications的成员方法size函数获取数组链表的大小
 int count = (*env)->CallIntMethod(env, allApplications, arraylist_size);
 LOGI("array_size:%d", count);

 int index = 0;
 // 遍历类对象mAllApplications实例（ArrayList）
 while (index < count) {

  // 调用类对象mAllApplications实例（ArrayList）的get函数获取第i个元素
  jobject clazzObj = (*env)->CallObjectMethod(env, allApplications, arraylist_get, index);
  LOGI("compare: i=%d item=%p", index, clazzObj);

  // 在类对象mAllApplication实例中查找外壳apk应用的StubApplication类对象实例
  if (((*env)->IsSameObject)(env, obj, clazzObj) == 1) {

   LOGI("replace: find same replace");
   // 调用类对象mAllApplication实例的set函数将外壳apk的StubApplication类对象实例
   // 替换为被保护dex文件的类Application实例对象
   (*env)->CallObjectMethod(env, allApplications, arraylist_set, index,
     onCreateObj);
  }
  // _JNIEnv::DeleteLocalRef(env, v12);
  ++index;
 }

 // 调用被保护dex文件的类Application实例对象的onCreate()函数
 (*env)->CallVoidMethod(env, onCreateObj, Application_onCreate);
 // 删除对被保护dex文件的类Application实例对象的全局引用
 (*env)->DeleteGlobalRef(env, onCreateObj);

}
```

外壳apk应用类StubApplication的成员函数onCreate对应的Native层实现函数native_onCreate具体实现流程的梳理如下：

Native层实现函数native_onCreate中 4 处修改外壳apk应用的ActivityThread对象中的类Application实例对象为被保护dex文件的类Application实例对象的位置整理。

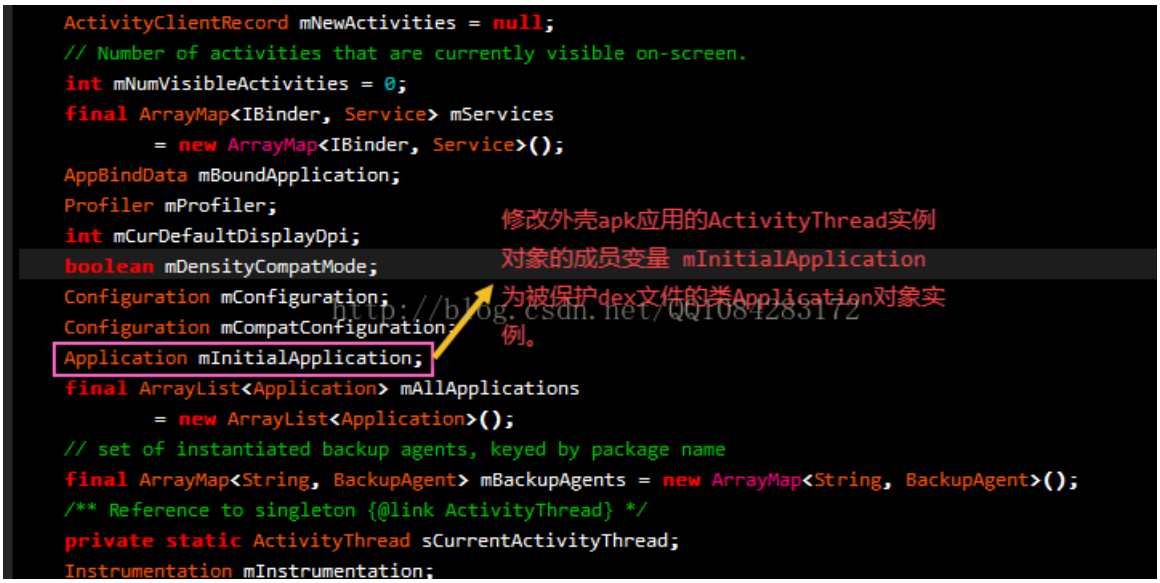
第 1 处--获取外壳apk应用进程的ActivityThread实例对象的非静态成员变量 **mPackages** （ArrayMap<String, WeakReference<LoadedApk>>类型）中外壳apk应用包名对应的类LoadedApk实例对象，然后修改类LoadedApk实例对象的非静态成员变量 **mApplication** 为被保护dex文件的类Application实例对象。

```
// These can be accessed by multiple threads; mPackages is the lock.
// XXX For now we keep around information about all packages we have
// seen, not removing entries from this map.
// NOTE: The activity and window managers need to call in to
// ActivityThread to do things like update resource configurations,
// which means this lock gets held while the activity and window managers
// holds their own lock.  Thus you MUST NEVER call back into the activity manager
// or window manager or anything that depends on them while holding this lock.
final ArrayMap<String, WeakReference<LoadedApk>> mPackages
        = new ArrayMap<String, WeakReference<LoadedApk>>();
final ArrayMap<String, WeakReference<LoadedApk>> mResourcePackages
        = new ArrayMap<String, WeakReference<LoadedApk>>();
final ArrayList<ActivityClientRecord> mRelaunchingActivities
        = new ArrayList<ActivityClientRecord>();
Configuration mPendingConfiguration = null;

private final ResourcesManager mResourcesManager;
```

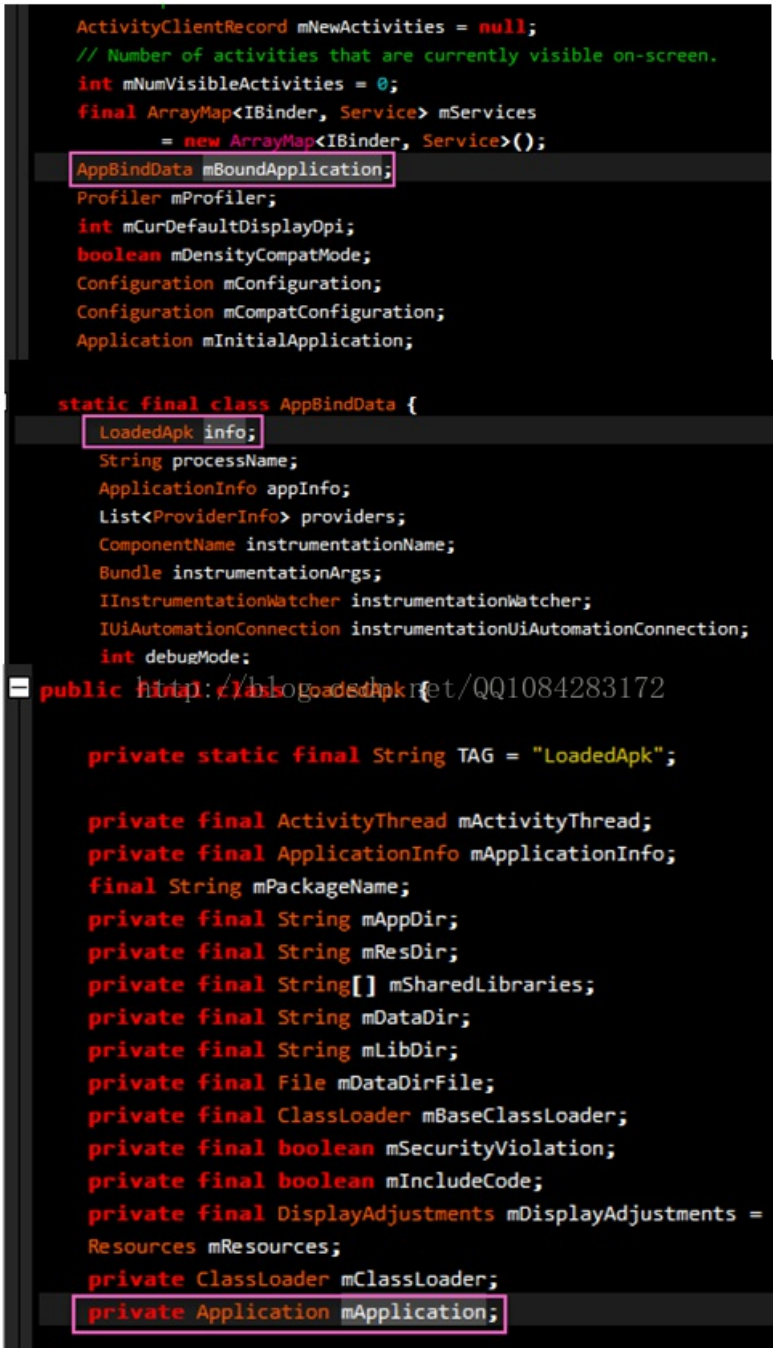
```
/**
 * Local state maintained about a currently loaded .apk.
 * @hide
 */
public final class LoadedApk {

    private static final String TAG = "LoadedApk";

    private final ActivityThread mActivityThread;
    private final ApplicationInfo mApplicationInfo;
    final String mPackageName;
    private final String mAppDir;
    private final String mResDir;
    private final String[] mSharedLibraries;
    private final String mDataDir;
    private final String mLibDir;
    private final File mDataDirFile;
    private final ClassLoader mBaseClassLoader;
    private final boolean mSecurityViolation;
    private final boolean mIncludeCode;
    private final DisplayAdjustments mDisplayAdjustments = new DisplayAdjustments();
    Resources mResources;
    private ClassLoader mClassLoader;
    private Application mApplication;
```

修改mPackages中，外壳apk应用包名对应LoadedApk实例对象的成员变量mApplication为被保护dex文件的类Application对象实例。

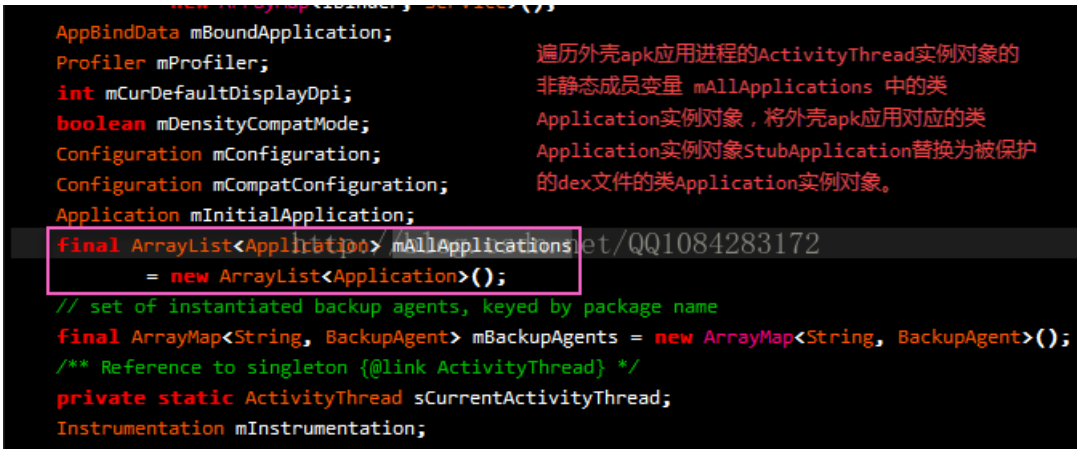
第 2 处--修改外壳apk应用的ActivityThread实例对象的成员变量 mInitialApplication（Application类型） 为被保护dex文件的类Application对象实例。



第 3 处--修改外壳apk应用进程的ActivityThread对象实例的非静态成员变量 **mBoundApplication** 中的LoadedApk类型的非静态成员变量**info** 实例对象中的实例成员变量**mApplication** 为被保护的dex文件的类Application对象实例。



第 4 处--遍历外壳apk应用进程的ActivityThread实例对象的非静态成员变量 mAllApplications 中的类Application实例对象，将外壳apk应用对应的类Application实例对象StubApplication替换为被保护的dex文件的类Application实例对象。



Native层实现函数native_onCreate的代码执行流程梳理：

< 1 >. 调用外壳apk应用进程的类ActivityThread的静态成员方法currentActivityThread，获取外壳apk应用进程的类ActivityThread实例对象。

```
// 2.然后被调用
// 参考源码：/frameworks/base/core/java/android/app/LoadedApk.java
// 参考源码：/frameworks/base/core/java/android/app/ActivityThread.java
// 此处的 obj 为类StubApplication的实例对象
void native_onCreate(JNIEnv *env, jobject obj) {

    /*
     * 调用外壳apk应用的ActivityThread.currentActivityThread()函数得到当前进程的ActivityThread实例
     * android.app.ActivityThread activityThread = ActivityThread.currentActivityThread();
     */

    // 获取类android.app.ActivityThread的全局引用
    if (!myFindClass(env, &ActivityThread, "android/app/ActivityThread")) {

        LOGI("ERROR:ActivityThread");
        return;
    }
    // 获取类android.app.ActivityThread的静态成员方法currentActivityThread的MethodID
    // android.app.ActivityThread currentActivityThread
    currentActivityThread = (*env)->GetStaticMethodID(env,
            ActivityThread,
            "currentActivityThread",
            "()Landroid/app/ActivityThread;");

    // 调用类android.app.ActivityThread的静态成员方法currentActivityThread()
    // 获取当前外壳apk应用的android.app.ActivityThread currentActivityThread
    jobject theCurrentActivityThread = (*env)->CallStaticObjectMethod(env,
            ActivityThread, currentActivityThread);
```

< 2 >. 获取外壳apk应用进程的ActivityThread实例对象的成员变量mPackages的值，并且成员变量mPackages为 **ArrayMap<String, WeakReference<LoadedApk>>**类型。

```
/*
 * 获取外壳apk应用进程的ActivityThread实例对象的成员变量mPackages的值
 * android.util.ArrayMap arrayMap = activityThread.mPackages;
 */
// 获取类android.app.ActivityThread的非静态成员变量mPackages的jfieldID
mPackages = (*env)->GetFieldID(env, ActivityThread, "mPackages",
        "Landroid/util/ArrayMap;");

//final ArrayMap<String, WeakReference<LoadedApk>> mPackages= new ArrayMap<String, WeakReference<LoadedApk>>();
// 获取类android.app.ActivityThread实例对象currentActivityThread的非静态成员变量mPackages
// mPackages为android.util.ArrayMap类型
jobject arraymap_mPackages = (*env)->GetObjectField(env,
        theCurrentActivityThread, mPackages);

// 将c语言形式的外壳apk应用的包名字符串转换成java类型的字符串
jstring thePackagename = (*env)->NewStringUTF(env, mPackageName);
LOGI("mPackageName %s", mPackageName);
```

< 3 >. 通过外壳apk应用的包名字符串，在mPackages实例对象中调用实例成员方法get函数查找、获取到对应的LoadedApk实例对象，然后获取到LoadedApk实例对象的弱引用。

```
/*
 * 获取到外壳apk应用的包名packageName对应的LoadedApk实例对象
 * LoadedApk objLoadedApk = arrayMap.get("thePackagename");
 */

// 获取类android.util.ArrayMap的全局引用
if (!myFindClass(env, &myArrayMap, "android/util/ArrayMap")) {

    LOGI("ERROR:myArrayMap");
    return;
}

// 获取类android.util.ArrayMap的非静态成员方法get函数
// android.util.ArrayMap.get(Objetc )
ArrayMap_get = (*env)->GetMethodID(env,
        myArrayMap,
        "get",
        "(Ljava/lang/Object;)Ljava/lang/Object;");

// 调用类实例对象mPackages(android.util.ArrayMap类型)的非静态成员方法get()函数
// 获取外壳apk应用的包名对应的LoadedApk类对象实例
jobject objLoadedApk = (*env)->CallObjectMethod(env,
        arraymap_mPackages,
        ArrayMap_get,
        // 外壳apk应用的包名
        thePackagename);

// 获取LoadedApk类对象实例的弱引用（WeakReference<LoadedApk>）
// final ArrayMap<String, WeakReference<LoadedApk>> mPackages
jobject v15 = (*env)->CallObjectMethod(env, objLoadedApk, WeakReference_get);
```


< 4 >. 设置外壳apk应用进程的LoadedApk实例对象的非静态成员变量 **mApplication** 为被保护dex文件的类Application对象实例，并设置ActivityThread实例对象的非静态成员变量**mInitialApplication** 为被保护dex文件的类Application对象实例。

```
/*
 * 设置外壳apk应用进程的LoadedApk实例对象的成员mApplication为被保护dex文件的类Application实例，
 * 并设置ActivityThread实例对象的成员mInitialApplication为被保护dex文件的类Application实例
 *
 * objLoadedApk.mApplication = onCreateObj;
 * activityThread.mInitialApplication = onCreateObj;
 */
if (!myFindClass(env, &myLoadedApk, "android/app/LoadedApk")) {

    LOGI("ERROR:myLoadedApk");
    return;
}
// 获取类android.app.LoadedApk的非静态成员mApplication的获取id
LoadedApk_mApplication = (*env)->GetFieldID(env,
        myLoadedApk,
        "mApplication",
        "Landroid/app/Application;");

// 修改类对象LoadedApk实例的非静态成员 mApplication 的值为被保护dex文件的类Application实例对象
(*env)->SetObjectField(env, v15, LoadedApk_mApplication, onCreateObj);

// 获取类android.app.ActivityThread的非静态成员变量mInitialApplication的jfieldID
// android.app.Application mInitialApplication
mInitialApplication = (*env)->GetFieldID(env,
        ActivityThread,
        "mInitialApplication",
        "Landroid/app/Application;");

// 修改类对象currentActivityThread实例非静态成员 mInitialApplication 的值
// 为被保护dex文件的Application实例对象
(*env)->SetObjectField(env, theCurrentActivityThread, mInitialApplication,
        onCreateObj);
```


< 5 >. 设置外壳apk应用进程的ActivityThread对象实例的非静态成员变量 **activityThread.mBoundApplication.info.mApplication** 为被保护的dex文件的类Application对象实例。

```
/*
 * 设置外壳apk应用进程的ActivityThread实例的activityThread.mBoundApplication.infor.mApplication
 * 为被保护的dex文件的类Application实例对象。
 *
 * android.app.ActivityThread.AppBindData boundApplication = activityThread.mBoundApplication;
 * android.app.LoadedApk _info = boundApplication.info;
 * _info.mApplication = onCreateObj;
 */

// 获取类android.app.ActivityThread的非静态成员变量mBoundApplication的jfieldID
// android.app.ActivityThread$AppBindData mBoundApplication
mBoundApplication = (*env)->GetFieldID(env,
        ActivityThread,
        "mBoundApplication",
        "Landroid/app/ActivityThread$AppBindData;");

// AppBindData mBoundApplication;
// 获取类对象currentActivityThread实例的非静态成员mBoundApplication的值
// 该成员mBoundApplication的类类型为android.app.ActivityThread$AppBindData
jobject boundApplication = (*env)->GetObjectField(env, theCurrentActivityThread,
        mBoundApplication);

// 获取类android.app.ActivityThread$AppBindData的全局引用（内部类）
if (!myFindClass(env, &AppBindData, "android/app/ActivityThread$AppBindData")) {

    LOGI("ERROR:AppBindData");
    return;
}
// 获取类android.app.ActivityThread$AppBindData的非静态成员变量info的jfieldID（android.app.LoadedApk）
// android.app.LoadedApk info
AppBindData_info = (*env)->GetFieldID(env,
        AppBindData,
        "info",
        "Landroid/app/LoadedApk;");

// AppBindData_info=(*env)->GetFieldID(env, AppBindData, "info", "Landroid/app/LoadedApk;");
// 获取类对象mBoundApplication的非静态成员info的值
// 成员info的类型为android.app.LoadedApk
jobject _info = (*env)->GetObjectField(env, boundApplication, AppBindData_info);

// 修改mBoundApplication.info的成员mApplication为被保护dex文件的类Application实例对象
// 其中mApplication为android.app.Application类型
(*env)->SetObjectField(env, _info, LoadedApk_mApplication, onCreateObj);
```

< 6 >. 获取外壳apk应用进程的ActivityThread实例对象的非静态成员变量 mAllApplications 中类Application实例对象的个数。

```
/*
 * 获取外壳apk应用进程的ActivityThread实例成员mAllApplications中类Application实例的个数
 *
 * ArrayList arrayListApplication = activityThread.mAllApplications;
 * int nApplicationSize = arrayListApplication.size();
 */

// 获取类android.app.ActivityThread的非静态成员变量mAllApplicationsn的jfieldID
// java.util.ArrayList mAllApplications
mAllApplications = (*env)->GetFieldID(env,
        ActivityThread,
        "mAllApplications",
        "Ljava/util/ArrayList;");

// 获取类对象currentActivityThread实例的非静态成员mAllApplications
// 其中mAllApplications为Ljava/util/ArrayList类型
// final ArrayList<Application> mAllApplications
jobject allApplications = (*env)->GetObjectField(env, theCurrentActivityThread,
        mAllApplications);

if (!myFindClass(env, &myArrayList, "java/util/ArrayList")) {

    LOGI("ERROR:myArrayList");
    return;
}
// 获取类java.util.ArrayList的非静态成员方法size的调用id
arraylist_size = (*env)->GetMethodID(env, myArrayList, "size", "()I");

// 调用类对象mAllApplications的成员方法size函数获取数组链表的大小
int count = (*env)->CallIntMethod(env, allApplications, arraylist_size);
LOGI("array_size:%d", count);
```

< 7 >. 遍历外壳apk应用进程的ActivityThread实例对象的非静态成员变量 mAllApplications 中的类Application实例对象，将外壳apk应用对应的类Application实例对象StubApplication替换为被保护的dex文件的类Application实例对象。

```
/*
 * 遍历外壳apk应用进程的ActivityThread实例成员mAllApplications中的类Application实例，
 * 将外壳apk应用的类Application实例StubApplication替换为被保护的dex文件的类Application实例
 * for (int i = 0; i < nApplicationSize; i++) {
 *
 *      Application objApplicaiton = arrayListApplication.get(i);
 *      if (IsSameObject(objStubApplication, objApplicaiton)){
 *
 *          arrayListApplication.set(i, objStubApplication);
 *      }
 * }
 */

// 获取类java.util.ArrayList的非静态成员方法get的调用id
arraylist_get = (*env)->GetMethodID(env,
        myArrayList,
        "get",
        "(I)Ljava/lang/Object;");

// 获取类java.util.ArrayList的非静态成员方法set的调用id
arraylist_set = (*env)->GetMethodID(env,
        myArrayList,
        "set",
        "(ILjava/lang/Object;)Ljava/lang/Object;");

int index = 0;
// 遍历外壳apk应用的类对象mAllApplications实例（ArrayList）中的所有类Application对象实例
while (index < count) {

    // 调用类对象mAllApplications实例（ArrayList）的get函数获取第i个元素
    jobject clazzObj = (*env)->CallObjectMethod(env, allApplications, arraylist_get, index);
    LOGI("compare: i=%d item=%p", index, clazzObj);

    // 在类对象mAllApplication实例中查找外壳apk应用的StubApplication类对象实例
    if (((*env)->IsSameObject)(env, obj, clazzObj) == 1) {

        LOGI("replace: find same replace");

        // 调用类对象mAllApplication实例的set函数将外壳apk的StubApplication类对象实例
        // 替换为被保护dex文件的类Application实例对象
        (*env)->CallObjectMethod(env, allApplications, arraylist_set, index,
                onCreateObj);
    }
    // _JNIEnv::DeleteLocalRef(env, v12);
    ++index;
}
```

< 8 >. 经过上面的操作已经全部将外壳apk应用的类Application实例对象替换为被保护dex文件的类Application实例对象，然后调用被保护的dex文件的类Application实例对象的非静态成员方法onCreate函数。

```
    /*
     * 经过前面的操作已经全部将外壳apk应用的类Application实例替换为被保护dex文件的类Application实例，
     * 然后调用被保护的dex文件的类Application实例的非静态成员方法onCreate()函数。
     * onCreateObj.onCreate();
     */

    if (!myFindClass(env, &myApplication, "android/app/Application")) {
        LOGI("ERROR:myApplication");
        return;
    }
    // 获取类android.app.Application的非静态成员函数onCreate的调用id
    Application_onCreate = (*env)->GetMethodID(env,
            myApplication,
            "onCreate",
            "()V");

    // 调用被保护dex文件的类Application实例对象的onCreate()函数
    (*env)->CallVoidMethod(env, onCreateObj, Application_onCreate);

    // 删除对被保护dex文件的类Application实例对象的全局引用
    (*env)->DeleteGlobalRef(env, onCreateObj);

}
```


5.在文章后面的讨论部分，作者繁华皆成空 又添加了对Android 7.0的Art虚拟机模式下的dex文件加载支持代码。

```
if(sdk_int > 22 &&sdk_int < 24)
{
 mCookie = (*env)->GetFieldID(env, myDexFile, "mCookie", "Ljava/lang/Object;");
 myOpenDexFile=(*env)->GetStaticMethodID(env, myDexFile, "openDexFile", "(Ljava/lang/String;Ljava/lang/Stri
}
else if(sdk_int >= 24) {

 //7.0+ openDexFile稍有不同
 mCookie = (*env)->GetFieldID(env, myDexFile, "mCookie", "Ljava/lang/Object;");
 myOpenDexFile=(*env)->GetStaticMethodID(env, myDexFile, "openDexFile", "(Ljava/lang/String;Ljava/lang/Stri
}


if(sdk_int > 22 &&sdk_int < 24)
{
 art_MarCookie=(*env)->CallStaticObjectMethod(env, myDexFile, myOpenDexFile, inPath,0,0);
 LOGI("----MarCookie:%p",art_MarCookie);

} else if(sdk_int>=24) {

    //7.0+此处需要5个参数
    jclass ApplicationClass = (*env)->GetObjectClass(env,new_ctx);
 jmethodID getClassLoader = (*env)->GetMethodID(env,ApplicationClass, "getClassLoader", "()Ljava/lang/Class
 jobject classLoader = (*env)->CallObjectMethod(env,new_ctx, getClassLoader);
 art_MarCookie=(*env)->CallStaticObjectMethod(env, myDexFile, myOpenDexFile, inPath,0,0,classLoader,0);
}
```

参考文章：

《阿里早期加固代码还原4.4-6.0》

《APK加壳【2】内存加载dex实现详解》

《DEX文件内存加载实现中的数据构造（C部分）》