Dalvik模式下基于Android运行时类加载的函数dexFindClass脱壳

原创

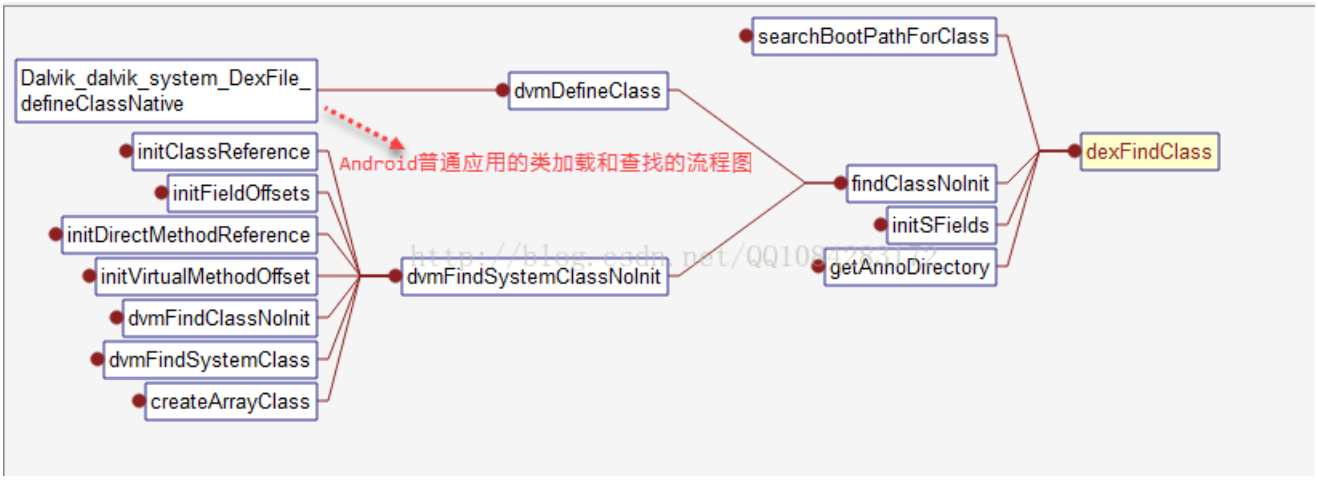
Fly20141201 于 2017-09-16 14:05:08 发布 4409 收藏 8
分类专栏： Android Hook学习 Android系统安全和逆向分析研究 文章标签： dexFindClass 360脱壳 android加固 Android Hook Android脱壳

本文链接：https://blog.csdn.net/QQ1084283172/article/details/78003184


Android Hook学习 同时被 2 个专栏收录
29 篇文章 3 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

前段时间在看雪论坛发现了《发现一个安卓万能脱壳方法》这篇文章，文章说的很简略，其实原理很简单也很有意思，说白了还是dalvik虚拟机模式下基于Android运行时的内存dex文件的dump，对一些免费版本的加固壳还是有效果的，dalvik模式下二代之后的加固壳就不行了。文章脱壳的原理涉及到dalvik模式下dex文件的类查找和加载的过程，下图是dalvik模式下dex文件的类查找和加载的流程示意图（native层的实现）：



Dalvik虚拟机模式下Android普通apk应用的类方法的调用过程:

先通过类方法所在类的类签名字符串查找到指定的目标类的 ClassObject描述对象，然后通过查找到的目标类对象 ClassObject查找获取到该类方法的描述结构体Method，再通过类方法描述结构体Method进行来方法的调用。dalvik模式下基于 dexFindClass 函数脱壳的原理就是在指定类签名字符串目标类的查找和加载过程中寻找脱壳点， 从 **Dalvik_dalvik_system_DexFile_defineClassNative函数-->dvmDefineClass函数-->findClassNoInit函数-->dexFindClass函数** 这整个流程是Android普通应用类查找和加载的流程，dexFindClass函数用于查找类的 DexClassDef 结构，Android普通应用目标类的查找和加载实现主要是在函数findClassNoInit里实现，http://androidxref.com/4.4.4_r1/xref/dalvik/vm/oo/Class.cpp#1473。

Android普通Apk应用类方法的查找流程梳理：

1.Android普通Apk应用类方法的查找和加载实现主要是从 函数findClassNoInit 开始的，很明显函数findClassNoInit调用完成之后返回是ClassObject类型的指针，ClassObject对象就是dex文件中类加载到内存之后的表现形式，类方法的调用也是先获取到类方法所在类的描述结构体ClassObject。

```
1459  /*
1460   * Find the named class (by descriptor). If it's not already loaded,
1461   * we load it and link it, but don't execute <clinit>. (The VM has
1462   * specific limitations on which events can cause initialization.)
1463   *
1464   * If "pDexFile" is NULL, we will search the bootclasspath for an entry.
1465   *
1466   * On failure, this returns NULL with an exception raised.
1467   *
1468   * TODO: we need to return an indication of whether we loaded the class or
1469   * used an existing definition.  If somebody deliberately tries to load a
1470   * class twice in the same class loader, they should get a LinkageError,
1471   * but inadvertent simultaneous class references should "just work".
1472   */
1473  static ClassObject* findClassNoInit(const char* descriptor, Object* loader,
1474      DvmDex* pDvmDex)
1475  {
1476      Thread* self = dvmThreadSelf();
1477      ClassObject* clazz;
1478      bool profilerNotified = false;
1479
1480      if (loader != NULL) {
1481          LOGVV("#### findClassNoInit(%s,%p,%p)", descriptor, loader,
1482              pDvmDex->pDexFile);
1483      }
```


2 .dalvik模式下Android类的查找和加载过程中，先调用 函数dvmLookupClass 进行类查找，如果查找不到指定类签名字符串的目标类，则进行目标类加载处理，意思就是说Android普通apk应用的在进行目标类的第一次查找时，目标类的ClassObject描述对象肯定是不存在的，需要先进行目标类的加载才会生成目标类的ClassObject描述对象，下次再查找该目标类就不用再进行类加载了。在进行目标类的加载时先调用 函数dexFindClass 获取到目标类的描述结构体 DexClassDef。

```
    if (dvmCheckException(self)) {
        ALOGE("Class lookup %s attempted with exception pending", descriptor);
        ALOGW("Pending exception is:");
        dvmLogExceptionStackTrace();
        dvmDumpAllThreads(false);
        dvmAbort();
    }

    // 先进行指定目标类的查找
    clazz = dvmLookupClass(descriptor, loader, true);
    if (clazz == NULL) {
        // 指定目标类查找不到（指定目标类还没有被加载）
        const DexClassDef* pClassDef;

        dvmMethodTraceClassPrepBegin();
        profilerNotified = true;

#if LOG_CLASS_LOADING
        u8 startTime = dvmGetThreadCpuTimeNsec();
#endif
                                                    http://blog.csdn.net/QQ1084283172

        if (pDvmDex == NULL) {
            assert(loader == NULL);     /* shouldn't be here otherwise */
            pDvmDex = searchBootPathForClass(descriptor, &pClassDef);
        } else {
            // 获取指定目标类的描述结构体DexClassDef
            pClassDef = dexFindClass(pDvmDex->pDexFile, descriptor);
        }

        if (pDvmDex == NULL || pClassDef == NULL) {
            if (gDvm.noClassDefFoundErrorObj != NULL) {
                /* usual case -- use prefabricated object */
                dvmSetException(self, gDvm.noClassDefFoundErrorObj);
            } else {
                /* dexopt case -- can't guarantee prefab (core.jar) */
                dvmThrowNoClassDefFoundError(descriptor);
            }
```

3. 将指定目标类的 DexClassDef传给函数loadClassFromDex进行目标类的加载，函数loadClassFromDex返回之后得到就是内存加载后的类 ClassObject。

```
        /* found a match, try to load it */
        // 传入指定目标类的DexClassDef描述结构体，获取加载后的类ClassObject
        clazz = loadClassFromDex(pDvmDex, pClassDef, loader);
        if (dvmCheckException(self)) {
            /* class was found but had issues */
            if (clazz != NULL) {
                dvmFreeClassInnards(clazz);
                dvmReleaseTrackedAlloc((Object*) clazz, NULL);
            }
            goto bail;
        }
                                                    http://blog.csdn.net/QQ1084283172

        /*
         * Lock the class while we link it so other threads must wait for us
         * to finish.  Set the "initThreadId" so we can identify recursive
         * invocation.  (Note all accesses to initThreadId here are
         * guarded by the class object's lock.)
         */
        dvmLockObject(self, (Object*) clazz);
        clazz->initThreadId = self->threadId;
```

函数loadClassFromDex先通过目标类的DexClassDef描述结构体，获取目标类的DexClassData信息结构体，接着根据DexClassData信息结构体获取到目标类的DexClassDataHeader描述结构体，然后将目标类的DexClassDef、DexClassData、DexClassDataHeader等类的描述结构体信息传给函数loadClassFromDex0，最终由函数loadClassFromDex0进行目标类的内存加载。

```
/*
 * Try to load the indicated class from the specified DEX file.
 *
 * This is effectively loadClass()+defineClass() for a DexClassDef.  The
 * loading was largely done when we crunched through the DEX.
 *
 * Returns NULL on failure.  If we locate the class but encounter an error
 * while processing it, an appropriate exception is thrown.
 */
static ClassObject* loadClassFromDex(DvmDex* pDvmDex,
    const DexClassDef* pClassDef, Object* classLoader)
{
    ClassObject* result;
    DexClassDataHeader header;
    const u1* pEncodedData;
    const DexFile* pDexFile;

    assert((pDvmDex != NULL) && (pClassDef != NULL));
    pDexFile = pDvmDex->pDexFile;

    if (gDvm.verboseClass) {
        ALOGV("CLASS: loading '%s'...",
            dexGetClassDescriptor(pDexFile, pClassDef));
    }

 // 通过目标类的DexClassDef获取到目标类的DexClassData
    pEncodedData = dexGetClassData(pDexFile, pClassDef);

    if (pEncodedData != NULL) {
  // 获取到目标类的DexClassDataHeader
        dexReadClassDataHeader(&pEncodedData, &header);
    } else {
        // Provide an all-zeroes header for the rest of the loading.
        memset(&header, 0, sizeof(header));
    }

 // 根据传入的目标类的DexClassDef、DexClassData以及DexClassDataHeader
 // 对目标类进行内存加载得到目标类内存加载之后的类描述结构体ClassObject
    result = loadClassFromDex0(pDvmDex, pClassDef, &header, pEncodedData,
            classLoader);

    if (gDvm.verboseClass && (result != NULL)) {
        ALOGI("[Loaded %s from DEX %p (cl=%p)]",
            result->descriptor, pDvmDex, classLoader);
    }

    return result;
}
```

4. 指定类签名字符串的目标类加载到内存以后，将其添加到普通apk应用进程的类Hash表中，方便下次该类的查找，以后查找该类的时候就不用再加载了，直接查找就能查找到。

```
    /*
     * Add to hash table so lookups succeed.
     *
     * [Are circular references possible when linking a class?]
     */
    assert(clazz->classLoader == loader);
    // 添加内存加载成功的类描述结构体ClassObject到类的Hash中
    if (!dvmAddClassToHash(clazz)) {
        /*
         * Another thread must have loaded the class after we
         * started but before we finished.  Discard what we've
         * done and leave some hints for the GC.
         *
         * (Yes, this happens.)
         */
        //ALOGI("WOW: somebody loaded %s simultaneously", descriptor);
        clazz->initThreadId = 0;
        dvmUnlockObject(self, (Object*) clazz);

        /* Let the GC free the class.
         */
        dvmFreeClassInnards(clazz);
        dvmReleaseTrackedAlloc((Object*) clazz, NULL);

        /* Grab the winning class.
         */
        clazz = dvmLookupClass(descriptor, loader, true);
        assert(clazz != NULL);
        goto got_class;
    }
```

```
/*
 * Add a new class to the hash table.
 *
 * The class is considered "new" if it doesn't match on both the class
 * descriptor and the defining class loader.
 *
 * TODO: we should probably have separate hash tables for each
 * ClassLoader. This could speed up dvmLookupClass and
 * other common operations. It does imply a VM-visible data structure
 * for each ClassLoader object with loaded classes, which we don't
 * have yet.
 */
bool dvmAddClassToHash(ClassObject* clazz)
{
    void* found;
    u4 hash;

 // 对指定签名字符串的类做Hash处理
    hash = dvmComputeUtf8Hash(clazz->descriptor);
 // 锁
    dvmHashTableLock(gDvm.loadedClasses);
 // 指定类的签名字符串hash和类加载后描述结构体ClassObject
 // 添加的进程的loadedClasses的哈希表中
    found = dvmHashTableLookup(gDvm.loadedClasses, hash, clazz,
                hashcmpClassByClass, true);
    dvmHashTableUnlock(gDvm.loadedClasses);

    ALOGV("+++ dvmAddClassToHash '%s' %p (isnew=%d) --> %p",
        clazz->descriptor, clazz->classLoader,
        (found == (void*) clazz), clazz);

    //dvmCheckClassTablePerf();

    /* can happen if two threads load the same class simultaneously */
    return (found == (void*) clazz);
}
```

5.指定签名字符串的目标类的ClassObject查找函数dvmLookupClass的实现。

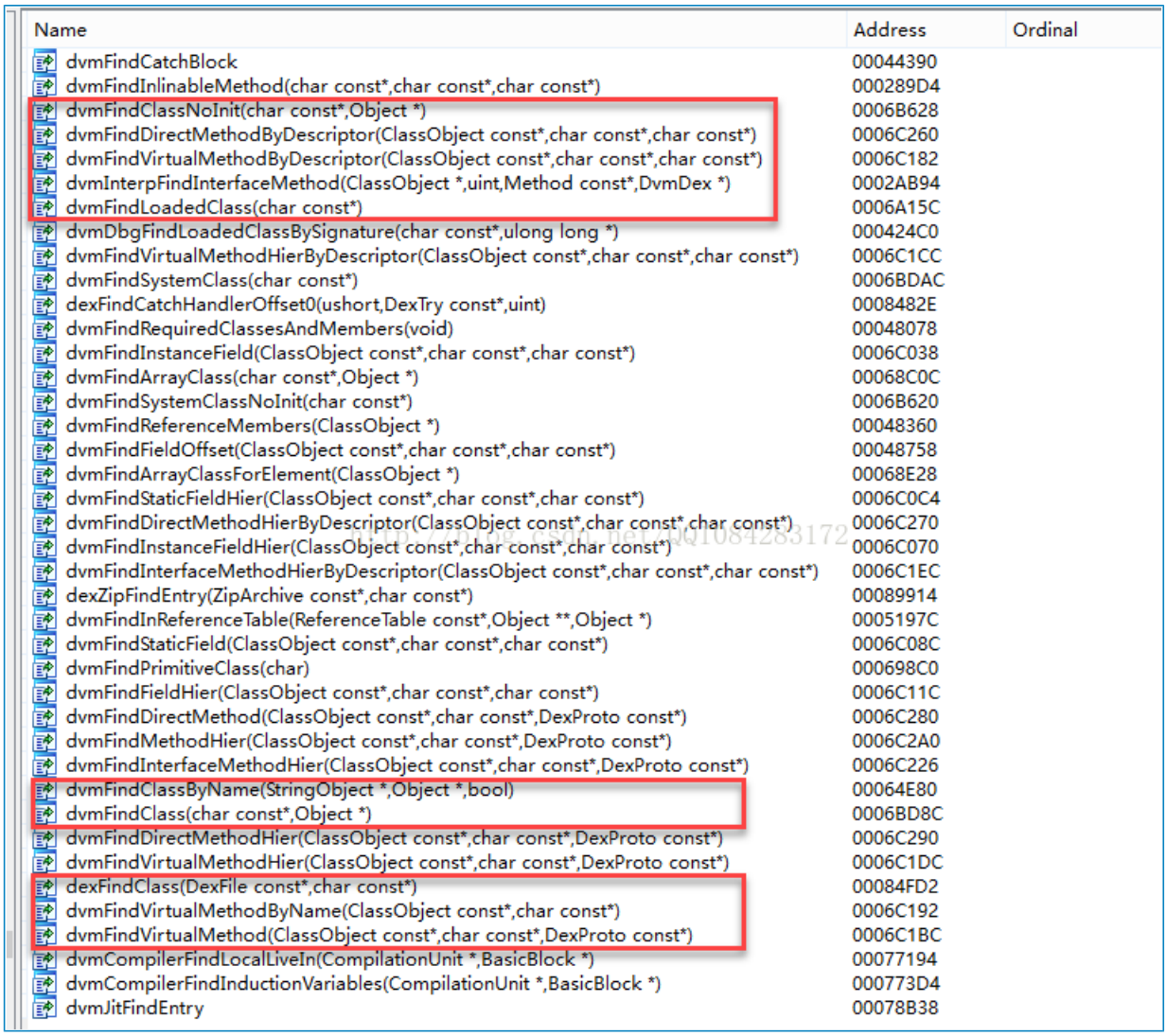
```
/*
 * Search through the hash table to find an entry with a matching descriptor
 * and an initiating class loader that matches "loader".
 *
 * The table entries are hashed on descriptor only, because they're unique
 * on *defining* class loader, not *initiating* class loader.  This isn't
 * great, because it guarantees we will have to probe when multiple
 * class loaders are used.
 *
 * Note this does NOT try to load a class; it just finds a class that
 * has already been loaded.
 *
 * If "unprepOkay" is set, this will return classes that have been added
 * to the hash table but are not yet fully loaded and linked.  Otherwise,
 * such classes are ignored.  (The only place that should set "unprepOkay"
 * is findClassNoInit(), which will wait for the prep to finish.)
 *
 * Returns NULL if not found.
 */
ClassObject* dvmLookupClass(const char* descriptor, Object* loader,
    bool unprepOkay)
{
    ClassMatchCriteria crit;
    void* found;
    u4 hash;

    crit.descriptor = descriptor;
    crit.loader = loader;
 // 对指定类的签名字符串做hash处理
    hash = dvmComputeUtf8Hash(descriptor);

    LOGVV("threadid=%d: dvmLookupClass searching for '%s' %p",
        dvmThreadSelf()->threadId, descriptor, loader);

    dvmHashTableLock(gDvm.loadedClasses);
 // 根据指定类的签名hash值查找该目标类的ClassObject
    found = dvmHashTableLookup(gDvm.loadedClasses, hash, &crit,
                hashcmpClassByCrit, false);
    dvmHashTableUnlock(gDvm.loadedClasses);

    /*
     * The class has been added to the hash table but isn't ready for use.
     * We're going to act like we didn't see it, so that the caller will
     * go through the full "find class" path, which includes locking the
     * object and waiting until it's ready.  We could do that lock/wait
     * here, but this is an extremely rare case, and it's simpler to have
     * the wait-for-class code centralized.
     */
    if (found && !unprepOkay && !dvmIsClassLinked((ClassObject*)found)) {
        ALOGV("Ignoring not-yet-ready %s, using slow path",
            ((ClassObject*)found)->descriptor);
        found = NULL;
    }

    return (ClassObject*) found;
}
```

6. Dalvik模式下基于Android运行时类加载的函数dexFindClass脱壳的要点: 由于一些免费版的apk加固基本都是dex文件的整体加载，粒度比较粗还没有细分到dex文件类方法的加固处理，在dalvik模式下Android运行时的类加载需要使用到内存加载的dex文件（此时一代的加固apk一般都已经在内存里解密完成，因此在解密后dex文件的类加载时，我们可以获取到解密后的dex文件的内存地址，这里选择在类加载的相关函数dexFindClass中进行内存dex文件的获取和dump处理），很多的普通apk应用都会调用dexFindClass函数，那么该怎么设置dex文件内存dump的过滤条件呢？被加固的dex虽然dex文件整体被加固了，类被隐藏了但是碍于加固的处理方法被加固dex文件的 主 **Activity** 还是暴露给我们了，故将被加固dex文件的主Activity类的签名字符串作为过滤条件。

```
436 /*
437  * Look up a class definition entry by descriptor.
438  *
439  * "descriptor" should look like "Landroid/debug/Stuff;".
440  */
441 const DexClassDef* dexFindClass(const DexFile* pDexFile,
442     const char* descriptor)
443 {
444     const DexClassLookup* pLookup = pDexFile->pClassLookup;
445     u4 hash;
446     int idx, mask;
447
448     hash = classDescriptorHash(descriptor);
449     mask = pLookup->numEntries - 1;
450     idx = hash & mask;
451
452     /*
453      * Search until we find a matching entry or an empty slot.
454      */
455     while (true) {
456         int offset;
457
458         offset = pLookup->table[idx].classDescriptorOffset;
459         if (offset == 0)
460             return NULL;
461
462         if (pLookup->table[idx].classDescriptorHash == hash) {
463             const char* str;
464
465             str = (const char*) (pDexFile->baseAddr + offset);
466             if (strcmp(str, descriptor) == 0) {
467                 return (const DexClassDef*)
468                     (pDexFile->baseAddr + pLookup->table[idx].classDefOffset);
469             }
470         }
471
472         idx = (idx + 1) & mask;
473     }
474 }
```

被加载类的内存dex文件的描述结构体

7. 以Android 4.4.4版本的系统为例，在dalvik虚拟机动态库文件libdvm.so 中，dvmFindClassNoInit、dvmFindDirectMethodByDescriptor、dvmFindVirtualMethodByDescriptor、dvmFindLoadedClass、dvmFindLoadedClass、dvmFindClass、**dexFindClass** 等函数是导出函数，可以被Hook掉。



Name	Address	Ordinal
dvmFindCatchBlock	00044390	
dvmFindInlinableMethod(char const*,char const*,char const*)	000289D4	
dvmFindClassNoInit(char const*,Object *)	0006B628	
dvmFindDirectMethodByDescriptor(ClassObject const*,char const*,char const*)	0006C260	
dvmFindVirtualMethodByDescriptor(ClassObject const*,char const*,char const*)	0006C182	
dvmInterpFindInterfaceMethod(ClassObject *,uint,Method const*,DvmDex *)	0002AB94	
dvmFindLoadedClass(char const*)	0006A15C	
dvmDbgFindLoadedClassBySignature(char const*,ulong long *)	000424C0	
dvmFindVirtualMethodHierByDescriptor(ClassObject const*,char const*,char const*)	0006C1CC	
dvmFindSystemClass(char const*)	0006BDAC	
dexFindCatchHandlerOffset0(ushort,DexTry const*,uint)	0008482E	
dvmFindRequiredClassesAndMembers(void)	00048078	
dvmFindInstanceField(ClassObject const*,char const*,char const*)	0006C038	
dvmFindArrayClass(char const*,Object *)	00068C0C	
dvmFindSystemClassNoInit(char const*)	0006B620	
dvmFindReferenceMembers(ClassObject *)	00048360	
dvmFindFieldOffset(ClassObject const*,char const*,char const*)	00048758	
dvmFindArrayClassForElement(ClassObject *)	00068E28	
dvmFindStaticFieldHier(ClassObject const*,char const*,char const*)	0006C0C4	
dvmFindDirectMethodHierByDescriptor(ClassObject const*,char const*,char const*)	0006C270	
dvmFindInstanceFieldHier(ClassObject const*,char const*,char const*)	0006C070	
dvmFindInterfaceMethodHierByDescriptor(ClassObject const*,char const*,char const*)	0006C1EC	
dexZipFindEntry(ZipArchive const*,char const*)	00089914	
dvmFindInReferenceTable(ReferenceTable const*,Object **,Object *)	0005197C	
dvmFindStaticField(ClassObject const*,char const*,char const*)	0006C08C	
dvmFindPrimitiveClass(char)	000698C0	
dvmFindFieldHier(ClassObject const*,char const*,char const*)	0006C11C	
dvmFindDirectMethod(ClassObject const*,char const*,DexProto const*)	0006C280	
dvmFindMethodHier(ClassObject const*,char const*,DexProto const*)	0006C2A0	
dvmFindInterfaceMethodHier(ClassObject const*,char const*,DexProto const*)	0006C226	
dvmFindClassByName(StringObject *,Object *,bool)	00064E80	
dvmFindClass(char const*,Object *)	0006BD8C	
dvmFindDirectMethodHier(ClassObject const*,char const*,DexProto const*)	0006C290	
dvmFindVirtualMethodHier(ClassObject const*,char const*,DexProto const*)	0006C1DC	
dexFindClass(DexFile const*,char const*)	00084FD2	
dvmFindVirtualMethodByName(ClassObject const*,char const*)	0006C192	
dvmFindVirtualMethod(ClassObject const*,char const*,DexProto const*)	0006C1BC	
dvmCompilerFindLocalLiveIn(CompilationUnit *,BasicBlock *)	00077194	
dvmCompilerFindInductionVariables(CompilationUnit *,BasicBlock *)	000773D4	
dvmJitFindEntry	00078B38	

8. Dalvik模式下基于Android运行时类加载的函数dexFindClass脱壳，对Android源码的修改（以Android 4.4.4 r1的源码为例），主要代码修改在源码文件 **/dalvik/libdex/DexFile.cpp** 中，修改如下：

```
// *******添加脱壳的代码***********************************************************
// http://androidxref.com/4.4.4_r1/xref/dalvik/libdex/DexFile.cpp

//#include <asm/siginfo.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

// 需要脱壳的apk的主activity的名称字符串
static char mainActivityName[128] = {0};
// 内存dex文件dump后文件夹
// /data/data/apk应用的包名/
static char dexDumpName[128] = {0};
// 记录脱壳配置文件是否读取的标记
static bool readable = true;
// 信号互斥量
static pthread_mutex_t read_mutex;

// 线程回调函数
void* ReadThread(void *arg)
{
    FILE *fp = NULL;
    while (mainActivityName[0] == 0 || dexDumpName[0] == 0)
    {
  // 读取脱壳的配置文件/data/local/tmp/dex_dump_ok的信息
        fp = fopen("/data/local/tmp/dex_dump_ok", "r");
        if (fp==NULL)
        {
            sleep(1);
            continue;
        }

  // 读取需要脱壳的apk的主activity的名称字符串（第1行）
        fgets(mainActivityName, 128, fp);
        mainActivityName[strlen(mainActivityName)-1] = 0;

        // 读取脱壳apk的内存dex文件dump后的文件名称字符串（第2行）
        fgets(dexDumpName, 128, fp);
        dexDumpName[strlen(dexDumpName)-1] = 0;

        fclose(fp);
        fp = NULL;
    }

 // 释放信号量
 pthread_mutex_lock(&read_mutex);
    return NULL;
}
// *******添加脱壳的代码***********************************************************
```

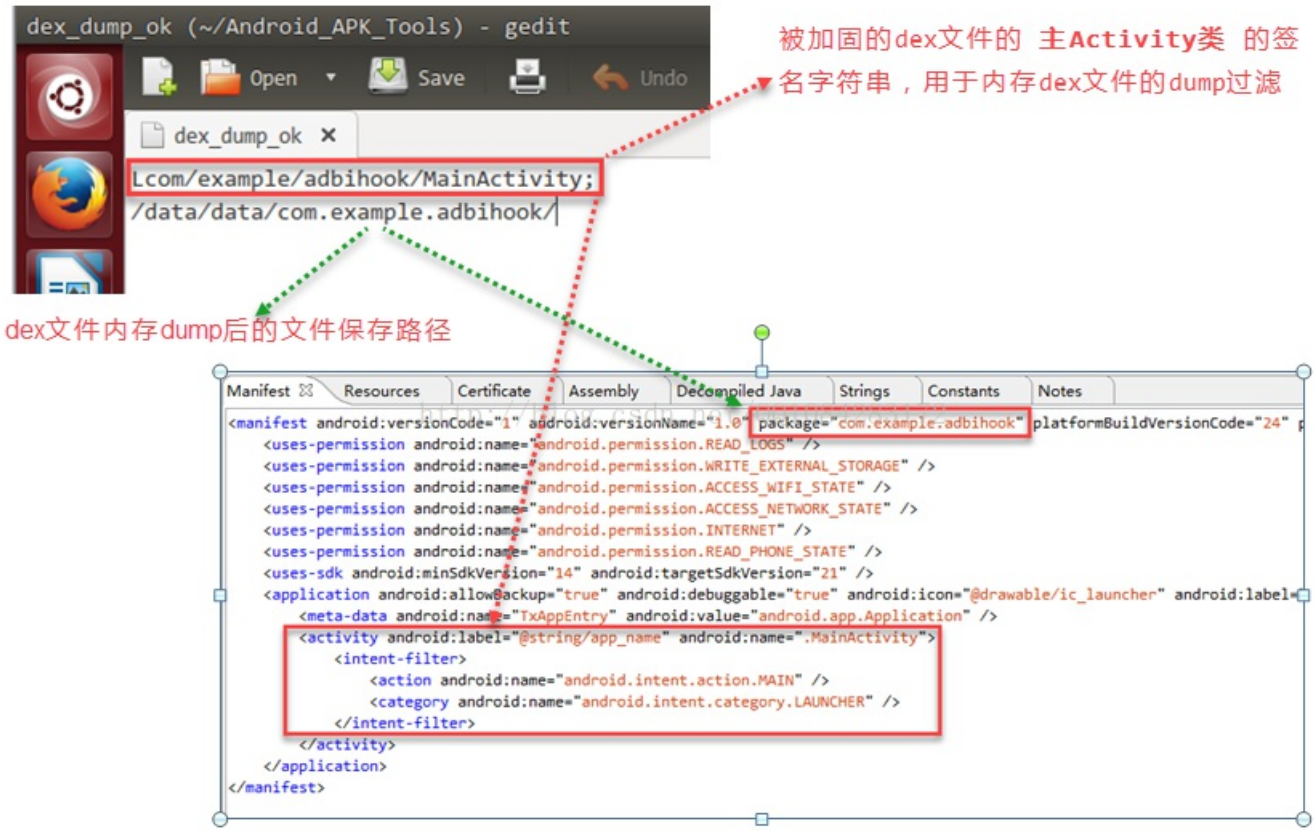
```
/*
 * Look up a class definition entry by descriptor.
 *
 * "descriptor" should look like "Landroid/debug/Stuff;".
 */
const DexClassDef* dexFindClass(const DexFile* pDexFile,
    const char* descriptor)
```

```
{
    const DexClassLookup* pLookup = pDexFile->pClassLookup;
    u4 hash;
    int idx, mask;

 // *******添加脱壳的代码*****************************************************
 // 1.读取配置文件(获取需要脱壳的apk的主activity类字符串)
 int uid = getuid();
 // 过滤掉系统进程
    if (uid)
    {
  // 打印当前被查找的类名称
  ALOGI("dexFindClass--DexFile addr: 0x%08x, Class descriptor: %s", (int)pDexFile, descriptor);

        if (readable)
        {
   // 创建互斥信号通量
            pthread_mutex_lock(&read_mutex);
            if (readable)
            {
                readable = false;

                // 释放互斥信号通量
                pthread_mutex_unlock(&read_mutex);
                pthread_t read_thread;
                // 创建线程，读取脱壳配置文件/data/local/tmp/dex_dump_ok的信息
                pthread_create(&read_thread, NULL, ReadThread, NULL);
            }
            else
            {
    // 释放互斥信号通量
                pthread_mutex_unlock(&read_mutex);
            }
        }
    }

 // 2，进行需要脱壳的apk的主activity类字符串的匹配
 // 格式:"Landroid/debug/Stuff;"
 if (strcmp(mainActivityName, descriptor) == 0) {

  // 3.匹配成功进行内存dex文件的dump处理
  char szBuffer[128] = {0};
  // 字符串拼接得到内存dex文件的dump路径
  // /data/data/com.example.seventyfour.tencenttest/
  strcat(szBuffer, dexDumpName);
  strcat(szBuffer, "dump_dex_over");
  // 打印dex文件的dump文件路径
  ALOGI("DEX_DUMP_PATH: %s", szBuffer);

  // 创建新文件保存dump的内存dex文件
  FILE* file = fopen(szBuffer, "wb+");
  if (file == NULL) {

   ALOGI("DEX_DUMP_PATH--fopen: %s error !", szBuffer);
  } else {

   // 保存三倍dex文件长度(比较暴力)
   // 暂时不考虑Hook系统函数write或者read反内存dump的情况
   fwrite(pDexFile->baseAddr, (pDexFile->pHeader->fileSize)*3, 1, file);
   // 关闭文件
```

```
   fclose(file);
  }
  // 打印内存dex文件的信息
  ALOGI("DEX_DUMP_PATH--addr: 0x%08x, lenth: %d", pDexFile->baseAddr, pDexFile->pHeader->fileSize);

 }
 // *******添加脱壳的代码*****************************************************

    hash = classDescriptorHash(descriptor);
    mask = pLookup->numEntries - 1;
    idx = hash & mask;

    /*
     * Search until we find a matching entry or an empty slot.
     */
    while (true) {
        int offset;

        offset = pLookup->table[idx].classDescriptorOffset;
        if (offset == 0)
            return NULL;

        if (pLookup->table[idx].classDescriptorHash == hash) {
            const char* str;

            str = (const char*) (pDexFile->baseAddr + offset);
            if (strcmp(str, descriptor) == 0) {
                return (const DexClassDef*)
                    (pDexFile->baseAddr + pLookup->table[idx].classDefOffset);
            }
        }

        idx = (idx + 1) & mask;
    }
}
```

9. 将Android 4.4.4 r1的源码文件 **/dalvik/libdex/DexFile.cpp** 按上面的修改以后，重新编译dalvik虚拟机的源码生成新的动态库文件libdvm.so，make snod 重新生成新的Android系统镜像文件system.img，重启Nexus 5手机进入刷机模式，使用新的Android系统镜像文件system.img进行刷机。加固apk脱壳的时候使用比较简单，先安装需要脱壳的apk应用到手机设备上，按下面的格式构建脱壳配置文件 **dex_dump_ok**，adb push 脱壳配置**文件 dex_dump_ok 到手机设备 /data/local/tmp 文件夹下，运行需要脱壳的加固apk，过一会儿在脱壳apk应用的包名路径下就会生成内存dump的dex文件 dump_dex_over**。

```
adb push  dex_dump_ok  /data/local/tmp
```



10. Dalvik模式下基于Android运行时类加载的函数dexFindClass脱壳，只针对第一代加壳的apk脱壳比较有效，我这里是通过修改Android源码的方式来进行内存dex文件dump脱壳操作，比较麻烦，由于dexFindClass函数 在动态库文件 libdvm.so 中是导出函数，因此也可以使用Hook dexFindClass函数的方式来进行dex文件内存dump的脱壳。整体来说，Dalvik模式下基于Android运行时类加载的函数dexFindClass脱壳的原理比较简单，主要是为了熟悉一下dalvik模式下dex文件类查找和加载的流程。

参考资料：

《发现一个安卓万能脱壳方法》