DexHunter在Dalvik虚拟机模式下的脱壳原理分析

原创

Fly20141201 于 2018-04-25 21:57:49 发布 1655 收藏 4
分类专栏： Android逆向学习 Android系统安全和逆向分析研究

本文链接：https://blog.csdn.net/QQ1084283172/article/details/78494671

在前面的博客《DexHunter的原理分析和使用说明（一）》、《DexHunter的原理分析和使用说明（二）》中已经将DexHunter工具的原理和使用需要注意的地方已经学习了一下，前面的博客中只讨论了DexHunter脱壳工具在Dalvik虚拟机模式下的脱壳原理和使用，一直想分析和研究一下DexHunter脱壳工具在ART虚拟机模式下的脱壳原理，终于有时间进行知识的消化了，特此整理一下基于Android运行时的脱壳工具DexHunter的脱壳原理，又将DexHunter工具的代码再看了几遍，对DexHunter脱壳工具的脱壳原理又有啦更进一步的理解和认识，并且也将DexHunter脱壳工具在ART虚拟机模式下的脱壳原理理解清楚啦。在DexHunter脱壳工具公布之后，类似变型的脱壳工具 Xdex 也出来了-详细的信息可以参考看雪文章《Xdex（百度版）脱壳工具基本原理》，基于类方法抽取的Android加固同样可以通过修改DexHunter脱壳工具进行脱壳，只是需要自己选择Android Dex文件的类方法加载点和类方法实现数据的dump点，最后进行dex文件的重组和还原。

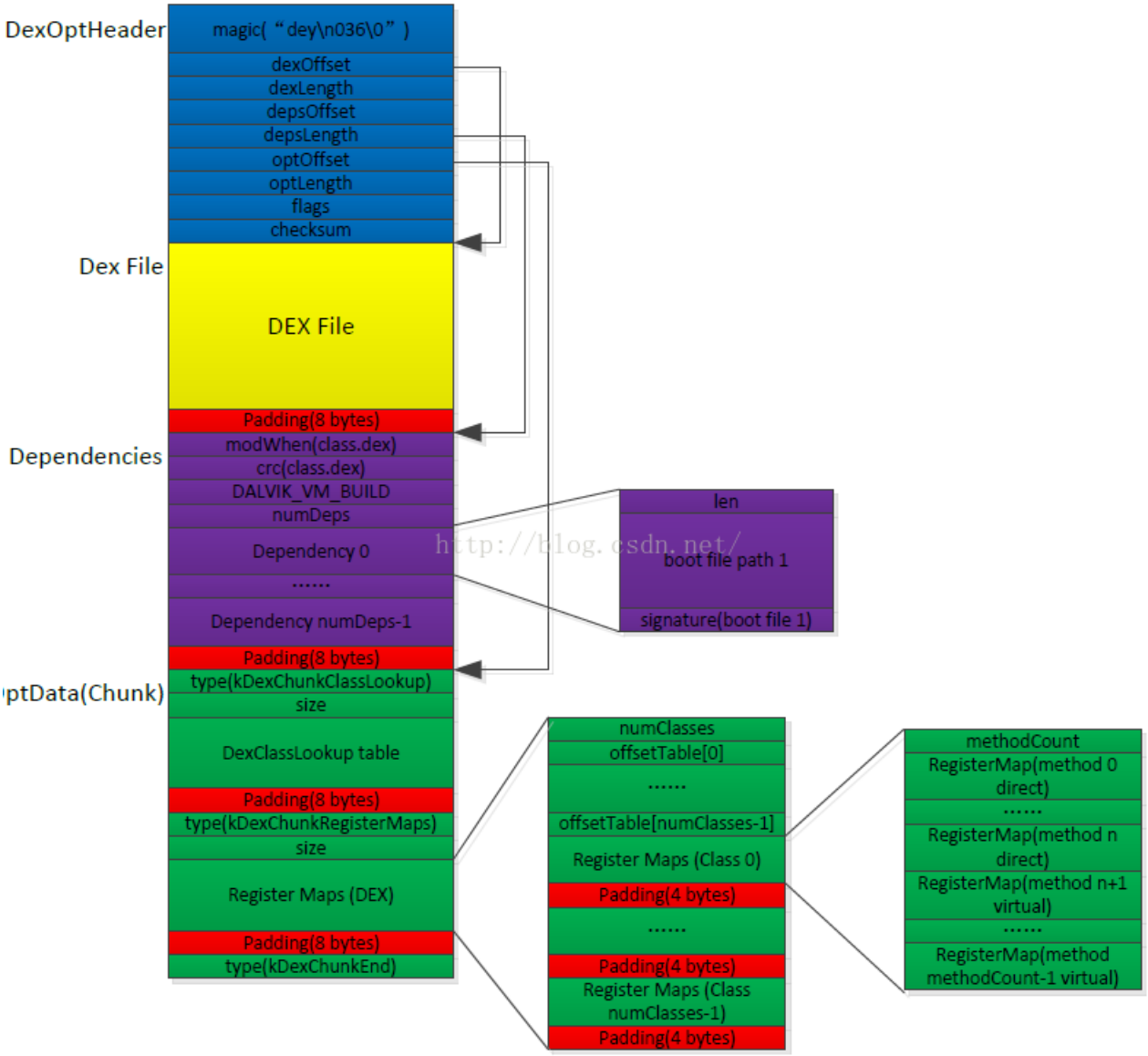
1. Dalvik虚拟机模式下，DexHunter脱壳工具dump出来的文件是odex文件

DexHunter脱壳工具是完全基于Android运行时进行dex文件的分步dump脱壳的，此时原始的Android Dex文件已经被Android系统优化为了odex文件进行执行。因此，DexHunter脱壳工具dump出来的dex文件是Android系统优化后的odex文件，并且DexHunter工具是分步进行odex文件的dump操作的，然后将dump出来odex文件的各部分进行重组得到一个能进行反编译分析的odex文件。

2. Dalvik虚拟机模式下Android Odex文件格式

在进行Dexhunter脱壳工具的原理分析之前，先了解一下Dalvik虚拟机模式下odex文件的格式。有关odex文件格式的详细信息，可以参考作者roland_sun的博文《Android系统ODEX文件格式解析》，写的比较详细也很不错，借用一下作者画的odex文件格式的示意图。

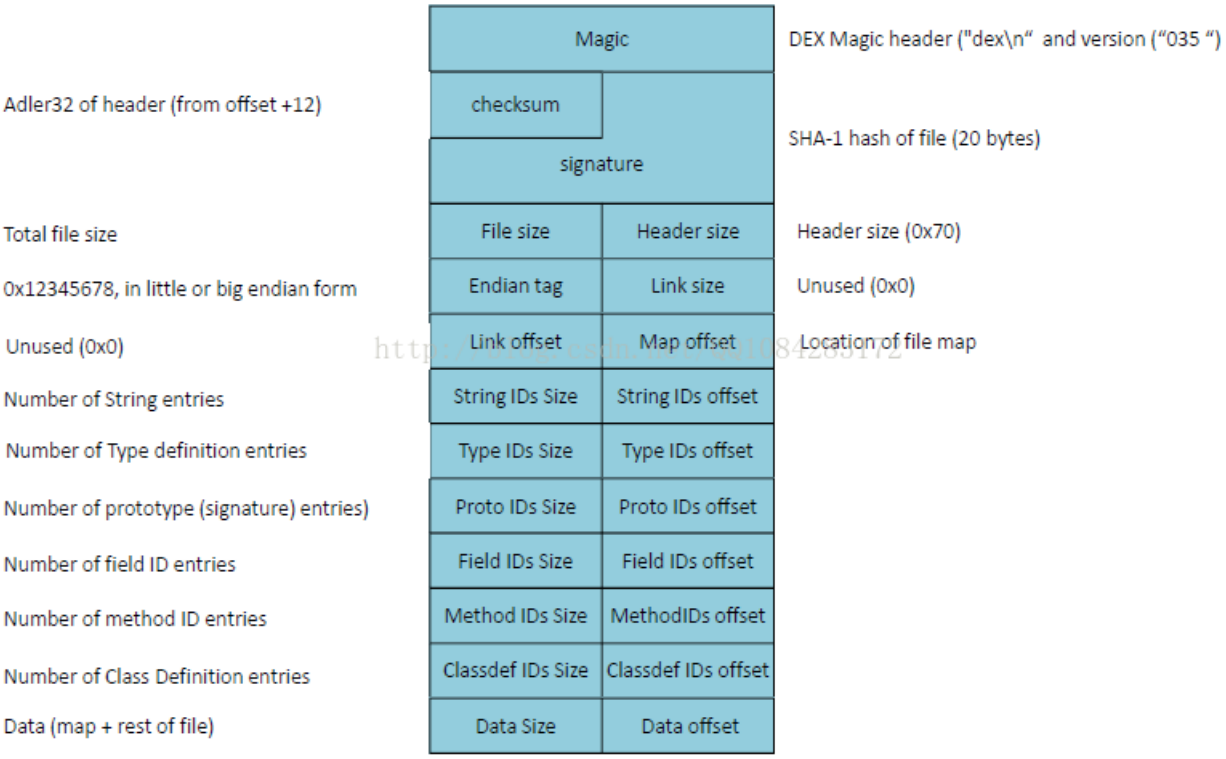
注意：flags域说明是用的大端字节序还是小端字节序，一般是小端，所以是0；最后是校验和的值，注意这个校验和不是算整个ODEX文件的，而是只算依赖库列表段和优化数据段的。



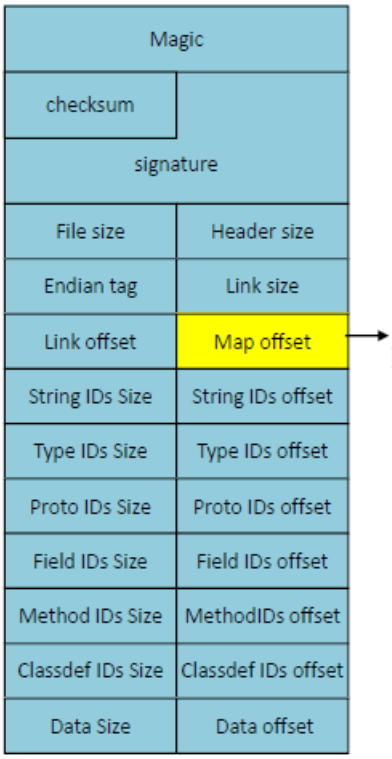
3. Dalvik虚拟机模式下Android Dex文件格式

Dalvik虚拟机模式下dex文件格式示意图来自于大牛 Jonathan Levin的paper《Andevcon-DEX.pdf》。

The DEX file format

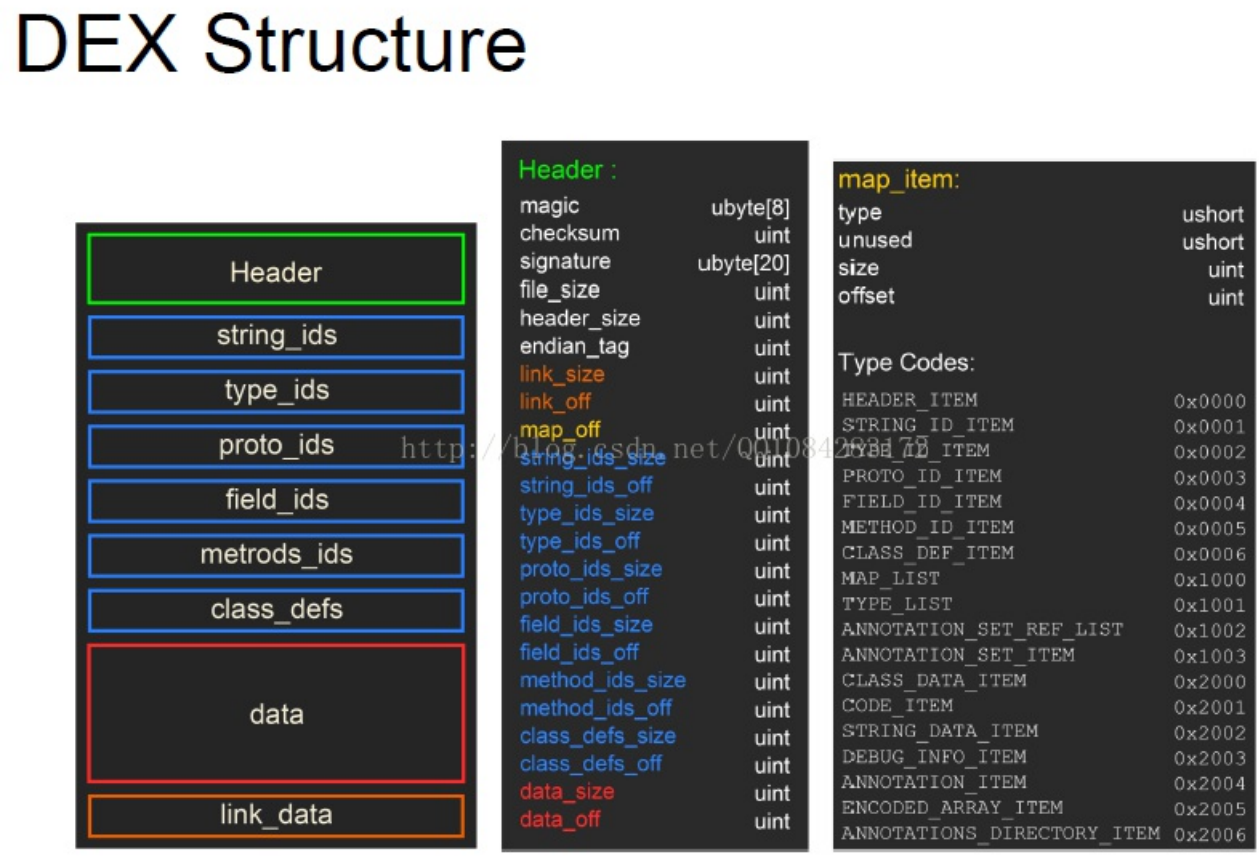


The DEX file format



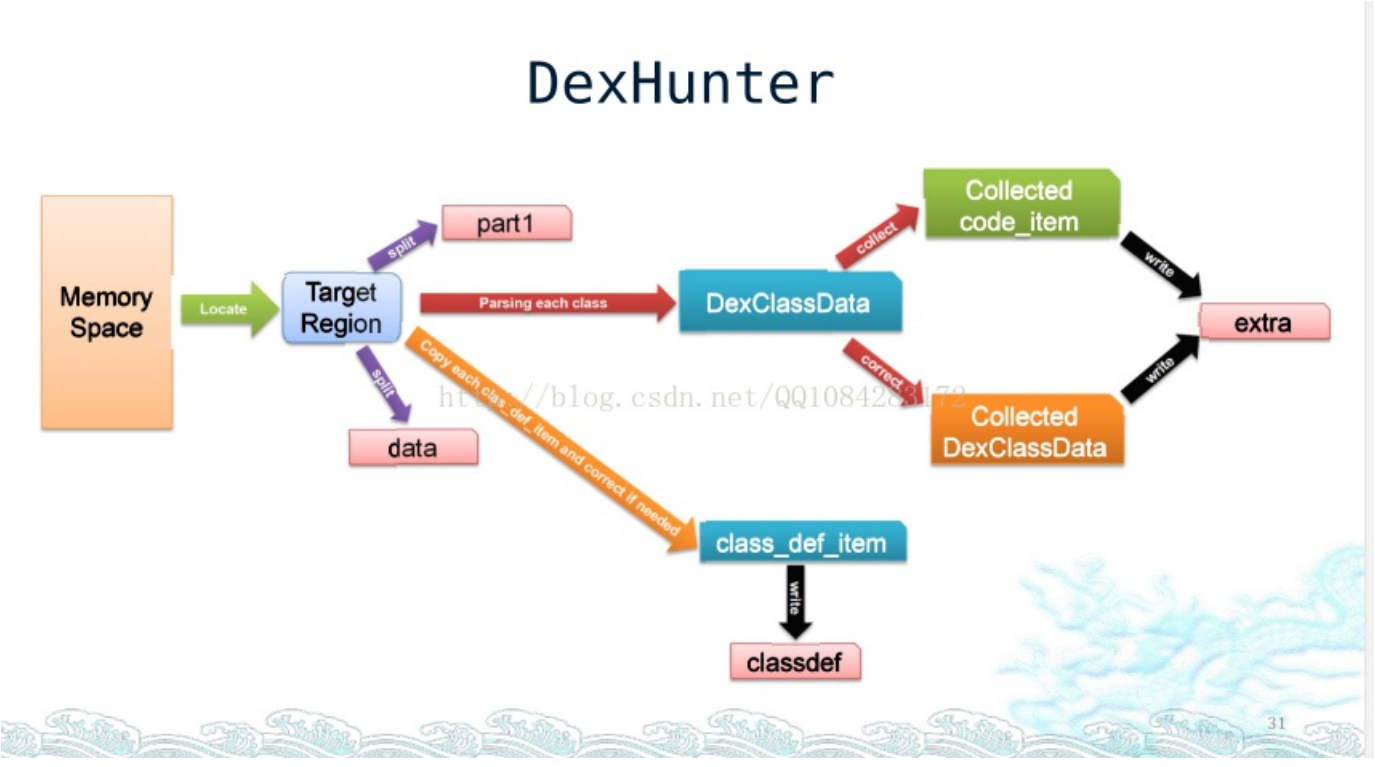
Type	Implies	Size	Offset
0x0	DEX Header	1 (implies Header Size)	0x0
0x1	String ID Pool	Same as String IDs size	Same as String IDs offset
0x2	Type ID Pool	Same as Type IDs size	Same as String IDs offset
0x3	Prototype ID Pool	Same as Proto IDs size	Same as ProtoIDs offset
0x4	Field ID Pool	Same as Field IDs size	Same as Field IDs offset
0x5	Method ID Pool	Same as Method IDs size	Same as Method IDs offset
0x6	Class Defs	Same as ClassDef IDs size	Same as ClassDef IDs offset
0x1000	Map List	1	Same as Map offset
0x1001	Type List	List of type indexes (from Type ID Pool)	
0x1002 0x1003	Annotation set Annotation Ref	Used by Class, method and field annotations	
0x2000	Class Data Item	For each class def, class/instance methods and fields	
0x2001	Code	DexCodeItems – contains the actual byte code	
0x2002	String Data	Pointers to actual string data	
0x2003	Debug Information	Debug_info_items containing line no and variable data)	
0x2004	Annotation	Field and Method annotations	
0x2005	Encoded Array	Used by static values	
0x2006	Annotations Directory	Annotations referenced from individual classdefs	

下面Android dex文件格式示的意图来自paper《A_deep_dive_into_dex_file_format》。

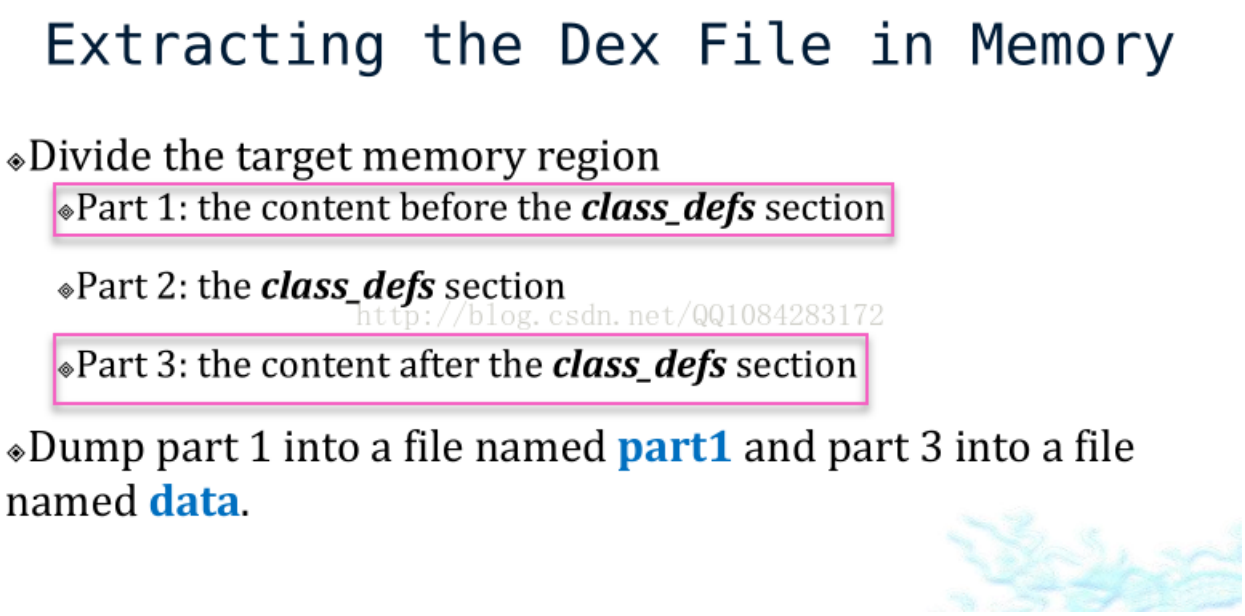


4. DexHunter脱壳工具的odex文件dump图解

DexHunter脱壳工具对被脱壳的Android应用的odex文件进行dump操作的原理图解（需要看懂DexHunter脱壳工具的代码，才能深刻的理解下面这幅图）



Dalvik虚拟机模式下，在目标被脱壳的Android进程内存中找到了需要dump的dex文件（实际是优化后的odex文件）所在的内存区域之后，将内存中dex文件分成3部分进行内存dump出来，然后进行dex文件的重组，最终得到能够进行逆向分析的dex文件。将目标被脱壳的Android进程内存中 odex文件开头到class_defs开始之前这段内存区域的数据 内存dump保存到 part1文件中；将从 class_defs的结尾开始到整个odex文件结尾之间的这段内存数据 内存dump操作保存到 data 文件中；dex文件中最关键的类信息描述结构体数据，以 Android运行时 的类填充数据为准，从内存中进行收集和整理，类定义的描述结构体DexClassDef的数据收集之后内存dump保存到classdef文件中，类实际的填充数据DexClassData和类方法的实际数据DexCode则按照偏移的格式保存到extra文件中，最后进行odex文件的重组。



5. DexHunter脱壳工具odex文件dump点的选择

Dalvik虚拟机模式下Android dex文件的dump点主要有4个地方：1. 打开dex文件的时候，判断dex文件是否优化为odex文件，如果已经优化为odex文件，则打开odex文件加载到内存中；2. 类Class加载的时候，dex文件加载到内存后的最终表现形式为ClassObject，在类方法被调用之前需要先加载该类Class并解析；3. 创建类Class对象初始化的时候，调用实例对象相关的操作时，需要先对类Class进行数据的初始化； 4. 类方法Method被调用的时候，类方法被调用的时需要获取Method方法的实际执行代码指令，这些代码指令从哪儿来，需要从加载到内存的dex文件中来获取。

Where to dump dex file?

- Four occasions
  - Opening a Dex file;
  - Loading a class;
  - Initializing a class;
  - Invoking a method;

Davik虚拟机模式下，为什么DexHunter脱壳工具选择类Class加载的时候进行dex文件的dump操作呢？

When to unpack the app?

- When the first class of the app is being loaded.
- Why?
  - Before a class is loaded, the content of the class should be available in the memory;
  - When the class is initialized, some content in memory may be modified dynamically;
  - Just before a method is invoked, its ***code_item*** or instructions should be available.
- How?
  - Load and initialize all classes proactively.

Dalvik虚拟机模式下，Android 类Class加载操作的步骤。

Loading a Class

- Operations
  - Form a class object from the data;
    构成、产生
  - Verify the legitimacy of access flags and the data;
    合法、合理
  - Populate all fields in the class object;
    填充
  - Deal with its super classes and/or interfaces;
  - Conduct some other checking.
    引导、组织

Dalvik虚拟机模式下，Android加载类Class的两种方法，调用类反射函数Class.forName进行类加载以及调用函数ClassLoader.loadClass进行类的主动加载。

Two Ways of Loading a Classes

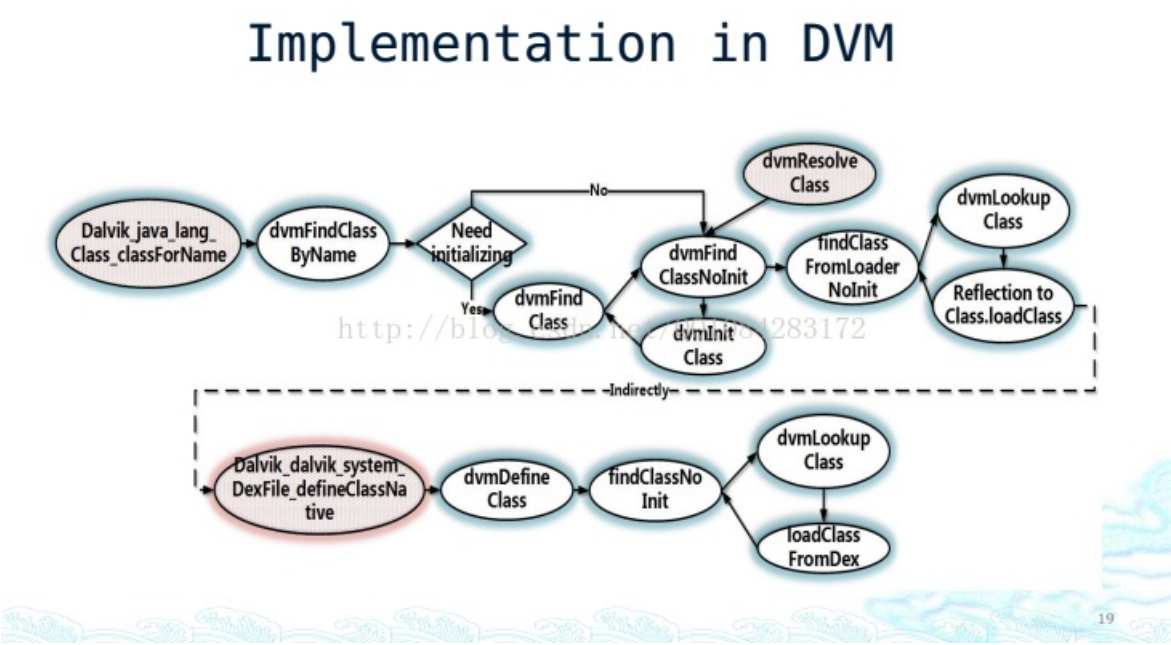
- Explicit approach
  - ***Class.forName()***, ***ClassLoader.loadClass()***.
- Implicit approach
  - E.g., ***new*** operation, accessing static members, etc.

Dalvik虚拟机模式下，java层类加载函数Class.forName和函数ClassLoader.loadClass对应的Native函数调用如下所示，在类对象创建时调用的dvmResolveClass函数获取ClassObject时也会涉及到Android 类的加载操作。

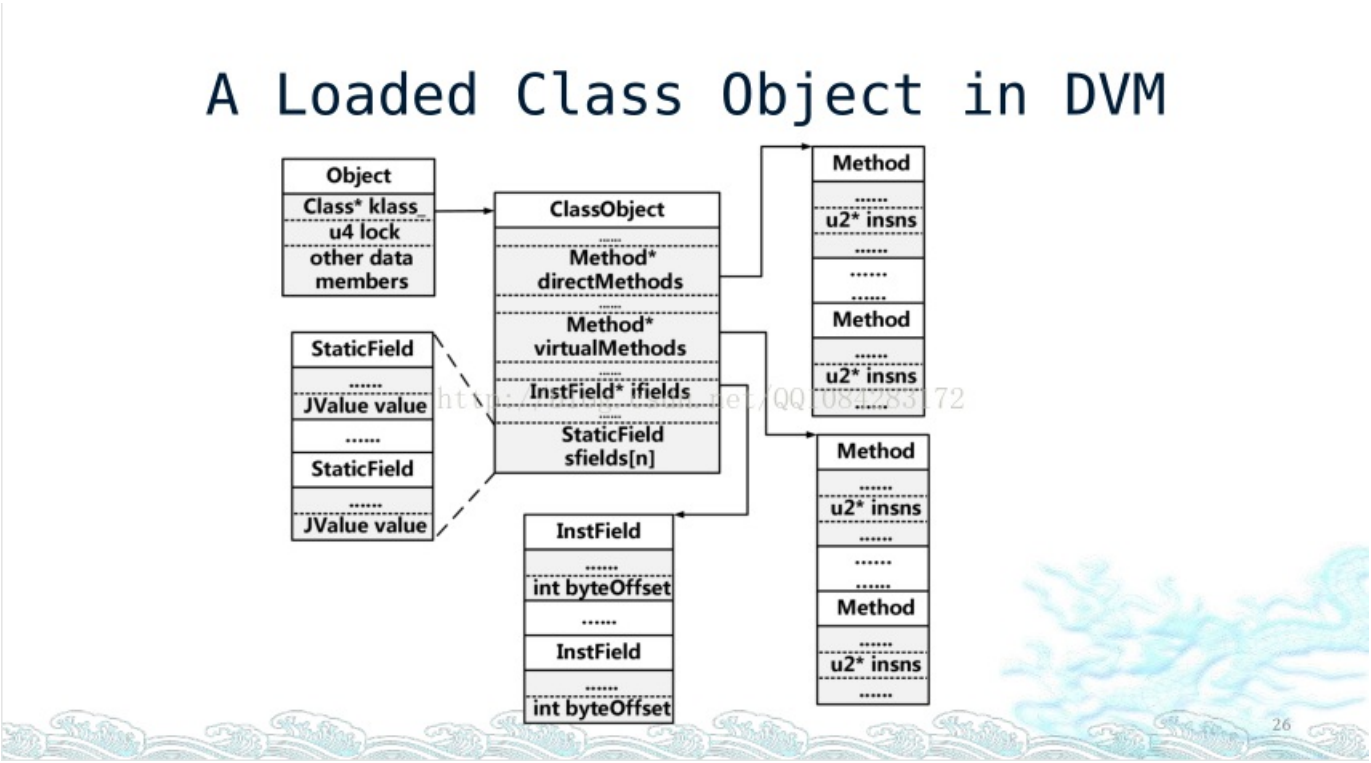
Implementation in DVM

- Explicit
  - ***ClassLoader.loadClass→Dalvik_dalvik_system_DexFile defineClassNative***
  - ***Class.forName →Dalvik_java_lang_Class_classForName***
- Implicit
  - ***new*** operations and so on→ ***dvmResolveClass***

Dalvik虚拟机模式下，Android类的加载最终是通过调用Native层的函数 Dalvik_dalvik_system_DexFile_defineClassNative 实现类的加载，因此我们在进行dex文件的内存dump时紧紧的卡在这个点的位置。

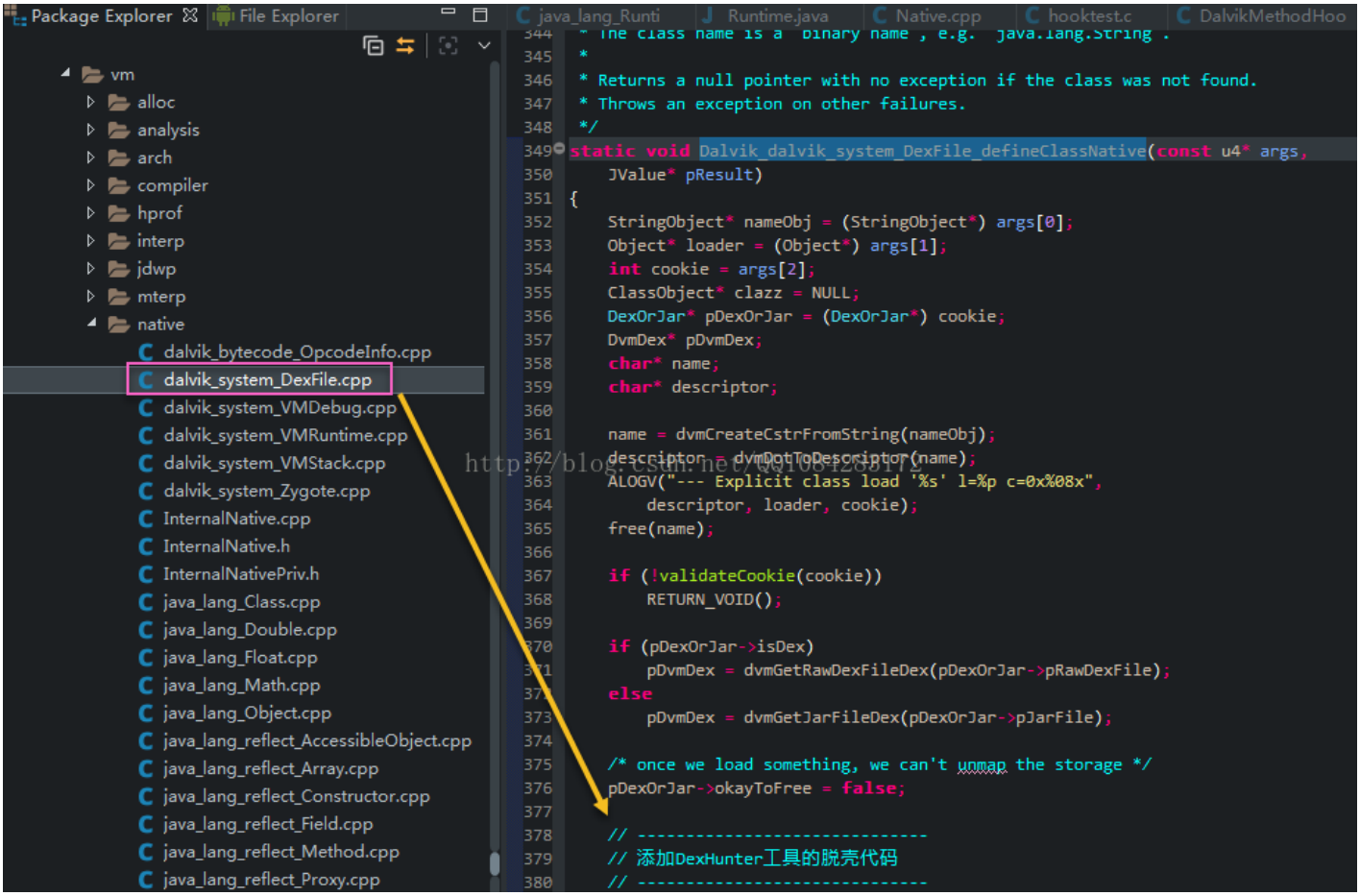


Dalvik虚拟机模式下，dex文件描述的类加载到内存后的ClassObject描述结构体示意图：



6.Dalvik虚拟机模式下，DexHunter脱壳工具对Android源码的修改位置（以Android 4.4.4 r1的源码为例）：对Android 4.4.4 r1 源码的文件路径 /dalvik/vm/native/dalvik_system_DexFile.cpp 中 函数**Dalvik_dalvik_system_DexFile_defineClassNative** 的实现代码进行修改。

http://androidxref.com/4.4.4_r1/xref/dalvik/vm/native/dalvik_system_DexFile.cpp#349



Dalvik虚拟机模式下，DexHunter脱壳工具内存dump被脱壳Android应用的odex文件的步骤详细分析。

7. Android系统进程的uid一般为0，普通应用的uid是大于0的，因此通过getuid获取当前Android进程的uid，使用uid进行简单的Android系统进程的过滤。创建原生线程，读取DexHunter脱壳工具需要的配置文件 /data/dexname 中的数据信息，顺便提示下：DexHunter脱壳工具出来以后，一些Android加固增加了对DexHunter脱壳工具的检测对抗，检测原理也比较简单--通过检测是否存在 **/data/dexname** 文件来检测DexHunter脱壳工具的存在， 因此在进行DexHunter脱壳工具的使用过程，最好修改一下DexHunter脱壳工具的配置文件路径。

```
//------------------------added begin----------------------//
    int uid = getuid();
    // 排除掉系统进程（因为系统进程的uid为0）
    if (uid)
    {
        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);
                http://blog.csdn.net/QQ1084283172
                pthread_t read_thread;
                // 创建线程，读取脱壳配置文件/data/dexname中数据信息以及设置等待定时器timer
                pthread_create(&read_thread, NULL, ReadThread, NULL);

            }
            else
            {
                // 释放互斥信号通量
                pthread_mutex_unlock(&read_mutex);
            }
        }
    }
```

DexHunter脱壳工具的配置文件 /data/dexname 中，有2行数据内容；第1行数据是Android加固对应的特征字符串 **dexname**--Android加固对应的被保护dex文件的加载路径字符串，不同Android加固的被保护dex文件的加载路径是不同的，同一种加固的被保护dex文件的加载路径也不是一直固定，脱壳操作之前需要自己先确定；第2行数据是DexHunter脱壳工具存放内存dump的odex文件数据的工作目录**dumppath**（要保证DexHunter脱壳工具在这个工作目录下有**文件的读写权限**），一般情况下这个工作目录设置为被脱壳Android应用的数据目录 **/data/data/pakegname(被脱壳的Android应用的包名)** 。创建并初始化定时器，主要是为了设置内存dump被脱壳Android应用的odex文件的等待时间，建议将定时器的等待时间设置的稍微长一点，因为现在的Android应用的dex文件都比较大，从内存中收集和dump被脱壳Android应用的odex文件还是比较耗费时间的；如果定时器的等待时间设置过短，会导致DexHunter脱壳失败的。

```
/*****读取配置文件/data/dexname中的数据信息并创建初始化定时器******/
void* ReadThread(void *arg)
{

    FILE *fp = NULL;
    while (dexname[0]==0 || dumppath[0]==0)
    {
        // 打开脱壳的配置文件/data/dexname
        // 脱壳的时，需要(adb push dexname /data)到Andrid系统里
        // 配置文件/data/dexname是可以自定义的，一些加固如梆梆可能会检测这个路径
        fp=fopen("/data/dexname", "r");
        if (fp==NULL)
        {
            sleep(1);
            continue;
        }

        // 读取文件中的第1行字符串---加固的特征字符串(即被加固的dex文件加载的文件路径)
        fgets(dexname, 99, fp);
        dexname[strlen(dexname)-1]=0;

        // 读取文件中的第2行字符串---被脱壳apk文件脱壳dump的dex文件数据的保存路径
        fgets(dumppath,99,fp);
        dumppath[strlen(dumppath)-1]=0;

        fclose(fp);
        fp=NULL;
    }

    struct sigevent sev;
    // 定时器事件类型为创建线程
    sev.sigev_notify=SIGEV_THREAD;
    // 设置事件的线程回调函数的传入参数
    sev.sigev_value.sival_ptr=&timerId;
    // 设置事件的线程回调函数--删除定时器
    sev.sigev_notify_function=timer_thread;
    // 设置事件的线程回调函数的属性
    sev.sigev_notify_attributes = NULL;

    // 创建定时器
    timer_create(CLOCK_REALTIME, &sev, &timerId);
```


DexHunter脱壳工具开源公布时，一些常见Android加固的特征字符串即被保护dex文件的加载路径特征字符串的格式，如下表所示：

String List

360	/data/data/XXX/.jiagu/classes.dex
Ali	/data/data/XXX/files/libmobisecy1.zip
Baidu	/data/data/XXX/.1/classes.jar
Bangcle	/data/data/XXX/.cache/classes.jar
Tencent	/data/app/XXX-1.apk (/data/app/XXX-2.apk)
ijiami	/data/data/XXX/cache/.

XXX stands for its package name.

8. 使用Android加固的特征字符串 dexname 即被保护dex文件的加载路径字符串，进行内存odex文件dump操作的准确过滤。

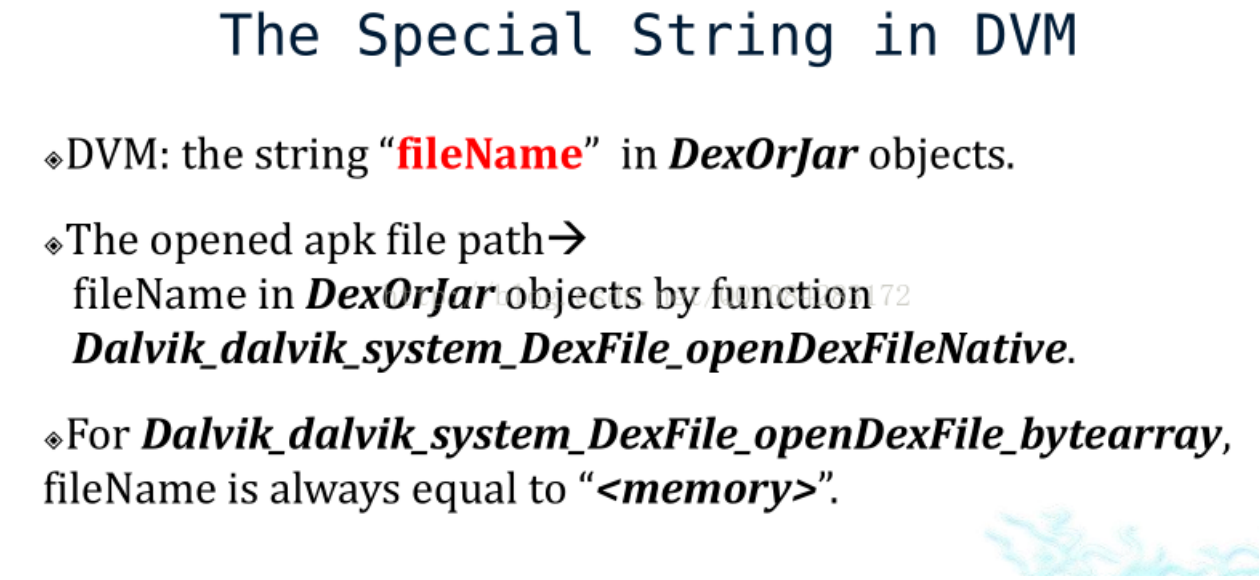
```
// 非Android系统进程且特征字符串不为空的情况下
if(uid && strcmp(dexname, ""))
{
    // 判断当前进程的apk是否是需要被脱壳的apk
    char * res = strstr(pDexOrJar->fileName, dexname);
    if (res&&flag)
    {
        // 创建互斥信号通量
        pthread_mutex_lock(&mutex);
        if (flag)
        {
            // 设置脱壳操作的开关为闭
            flag = false;
            // 释放互斥信号通量
            pthread_mutex_unlock(&mutex);

            // 获取内存中odex文件的结构信息
            DexFile* pDexFile = pDvmDex->pDexFile;
            // 获取odex文件在内存中的存放地址信息结构体
            MemMapping * mem = &pDvmDex->memMap;

            char * temp = new char[100];

            //-----------------第1步----------------------
```


Android加固对应的特征字符串即被保护dex文件的加载路径字符串，Dalvik虚拟机模式下是根据 cookie值 的描述结构体 DexOrJar 中的成员变量 **fileName** 来确定的，最终都是由加载dex文件到内存的 函数 Dalvik_dalvik_system_DexFile_openDexFileNative 和 函数 Dalvik_dalvik_system_DexFile_openDexFile_bytearray 来决定的。



Dalvik虚拟机模式下，DexOrJar 结构体中的成员变量 filename 描述了被加载的dex文件的路径。

```
33
34 /*
35  * Internal struct for managing DexFile.
36  */
37 struct DexOrJar {
38
39     // 被加载dex文件或者jar文件的路径字符串
40     char*       fileName;
41     bool        isDex;
42     bool        okayToFree;
43     RawDexFile* pRawDexFile;
44     JarFile*    pJarFile;
45     u1*         pDexMemory; // malloc()ed memory, if any
46 };
47
```

函数 Dalvik_dalvik_system_DexFile_openDexFileNative 加载dex文件到Android进程内存的实现（正常情况下，Android应用加载dex文件调用该函数）。

```
/*
 * 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;

    /*
     * We have to deal with the possibility that somebody might try to
     * open one of our bootstrap class DEX files.  The set of dependencies
     * will be different, and hence the results of optimization might be
     * different, which means we'd actually need to have two versions of
     * the optimized DEX: one that only knows about part of the boot class
     * path, and one that knows about everything in it.  The latter might
     * optimize field/method accesses based on a class that appeared later
     * in the class path.
     *
     * We can't let the user-defined class loader open it and start using
     * the classes, since the optimized form of the code skips some of
     * the method and field resolution that we would ordinarily do, and
     * we'd have the wrong semantics.
     *
     * We have to reject attempts to manually open a DEX file from the boot
     * class path.  The easiest way to do this is by filename, which works
     * out because variations in name (e.g. "/system/framework/./ext.jar")
     * result in us hitting a different dalvik-cache entry.  It's also fine
     * if the caller specifies their own output file.
     */
    if (dvmClassPathContains(gDvm.bootClassPath, sourceName)) {
        ALOGW("Refusing to reopen boot DEX '%s'", sourceName);
        dvmThrowIOException(
            "Re-opening BOOTCLASSPATH DEX files is not allowed");
        free(sourceName);
        free(outputName);
        RETURN_VOID();
    }

    /*
     * Try to open it directly as a DEX if the name ends with ".dex".
     * If that fails (or isn't tried in the first place), try it as a
     * Zip with a "classes.dex" inside.
     */
    if (hasDexExtension(sourceName)
```

```
            && dvmRawDexFileOpen(sourceName, outputName, &pRawDexFile, false) == 0) {
        ALOGV("Opening DEX file '%s' (DEX)", sourceName);

        pDexOrJar = (DexOrJar*) malloc(sizeof(DexOrJar));
        pDexOrJar->isDex = true;
        pDexOrJar->pRawDexFile = pRawDexFile;
        pDexOrJar->pDexMemory = NULL;
    } else if (dvmJarFileOpen(sourceName, outputName, &pJarFile, false) == 0) {
        ALOGV("Opening DEX file '%s' (Jar)", sourceName);

        pDexOrJar = (DexOrJar*) malloc(sizeof(DexOrJar));
        pDexOrJar->isDex = false;
        pDexOrJar->pJarFile = pJarFile;
        pDexOrJar->pDexMemory = NULL;
    } else {
        ALOGV("Unable to open DEX file '%s'", sourceName);
        dvmThrowIOException("unable to open DEX file");
    }

    if (pDexOrJar != NULL) {

        // 被加载的dex文件的路径描述
        pDexOrJar->fileName = sourceName;
        addToDexFileTable(pDexOrJar);
    } else {
        free(sourceName);
    }

    free(outputName);
    RETURN_PTR(pDexOrJar);
}
```

函数Dalvik_dalvik_system_DexFile_openDexFile_bytearray 加载dex文件到Android进程内存的实现（Android加固加载被保护dex文件时调用该函数）。

```
/*
 * 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;
    // 被加载的dex文件的路径描述
    pDexOrJar->fileName = strdup("<memory>"); // Needs to be free()able.
    addToDexFileTable(pDexOrJar);

    RETURN_PTR(pDexOrJar);
}
```

9.通过被脱壳Android应用进程dex文件的内存描述结构体 DvmDex 的成员变量 memMap 获取到该进程的odex文件所在的内存区域，然后将 **odex文件开头到class_defs数据起始偏移之间的数据** 从内存中dump出来保存到文件 **/data/data/pakegname/part1** 中，其中pakegname为被脱壳Android应用的包名。

```
//-----------------第1步----------------------

// 获取当前进程apk的dump数据的文件保存路径/data/data/xxxx/
strcpy(temp, dumppath);
// 拼接字符串
strcat(temp, "part1");
// 创建并打开文件/data/data/xxxx/part1
FILE *fp = fopen(temp, "wb+");

// 获取odex文件在内存中的存放指针
const u1 *addr = (const u1*)mem->addr;

// 获取odex文件的开头到dex文件classDefsOff起始地址之间的文件数据长度
int length = int(pDexFile->baseAddr+pDexFile->pHeader->classDefsOff - addr);

// 将这部分文件数据写入到文件/data/data/xxxx/part1中
fwrite(addr, 1, length, fp);
// 刷新文件流
fflush(fp);
// 关闭文件
fclose(fp);

//-----------------第2步----------------------
```


示意图：

10.将被脱壳Android应用的内存odex文件的 **class_defs数据结束偏移到odex文件结束的这段内存数据** 内存dump出来保存到文件 **/data/data/pakegname/data** 中，其中pakegname为被脱壳Android应用的包名。

```
//-----------------第2步----------------------

// 获取当前进程apk的数据目录路径/data/data/xxxx/
strcpy(temp, dumppath);
// 拼接字符串
strcat(temp,"data");

// 打开文件/data/data/xxxx/data
fp = fopen(temp, "wb+");

// 获取内存中dex文件存放DexClassDef结构体的结束内存地址
addr = pDexFile->baseAddr+pDexFile->pHeader->classDefsOff+sizeof(DexClassDef)*pDexFile->pHeader->classDefsSize;
// 获取内存中odex文件存放DexClassDef结构体的结尾到odex文件结尾的数据长度
length = int((const u1*)mem->addr+mem->length-addr);

// 将这部分数据写入到文件/data/data/xxxx/data中
fwrite(addr, 1, length, fp);
// 刷新文件流
fflush(fp);
// 关闭文件
fclose(fp);

// 删除申请的内存空间
delete temp;

//-----------------第3步----------------------
```


示意图：

11. 创建原生线程，遍历被脱壳Android应用odex文件的 DexClassDef 数据数组，基于Android运行时进行被脱壳Android应用类的描述数据 DexClassDef 、DexClassData 和类方法的实现数据 DexCode 的收集和内存dump，将类的定义描述数据 **DexClassDef**内存dump保存到文件**/data/data/pakegname/classdef** 中，将类的实际描述结构体数据**DexClassData** 和类方法的实现数据**DexCode** 内存dump保存到文件**/data/data/pakegname/extra** 中，然后将内存dump的odex文件的 4部分数据 进行重组，得到能够反编译odex文件，其中pakegname为被脱壳Android应用的包名。

```
            //-----------------第3步-----------------------

            // 用于保存线程id
            pthread_t dumpthread;

            // 构建线程的传入参数
            param.loader = loader;
            param.pDvmDex = pDvmDex;

            // 创建线程进行类class的内存dump
            dvmCreateInternalThread(&dumpthread, "ClassDumper", DumpClass, (void*)&param);
        }
        else
        {
            // 释放互斥信号通量
            pthread_mutex_unlock(&mutex);
        }
    }
}
//-----------------------added end----------------------//
```


12. 创建保存内存dump的类定义描述结构体 DexClassDef 数据的文件 /data/data/pakegname/classdef 以及创建保存内存dump的类数据 DexClassData 和 类方法的实际实现数据 DexCode 的文件 /data/data/pakegname/extra 其中pakegname为被脱壳Android应用的包名 。

```
void* DumpClass(void *parament)
{
  // 休眠一段时间
  while (timer_flag)
  {
      sleep(5);
  }

  // 获取odex文件的内存描述结构DvmDex
  DvmDex* pDvmDex=((struct arg*)parament)->pDvmDex;
  Object *loader=((struct arg*)parament)->loader;

  // 获取指向odex文件的信息结构体指针
  DexFile* pDexFile = pDvmDex->pDexFile;
  // 获取内存存放odex文件的内存地址信息描述结构体
  MemMapping * mem = &pDvmDex->memMap;

  // 获取当前时间
  u4 time = dvmGetRelativeTimeMsec();
  ALOGI("GOT IT begin: %d ms", time);
            http://blog.csdn.net/QQ1084283172
  // 申请内存空间
  char *path = new char[100];


  // 获取被脱壳apk的dump数据的文件保存路径
  strcpy(path, dumppath);
  // 拼接字符串（得到存放classdef的文件路径）
  // xxxx/classdef文件中保存着内存dump的类DexClassDef
  strcat(path, "classdef");
  // 创建并打开文件xxxx/classdef
  FILE *fp = fopen(path, "wb+");


  strcpy(path, dumppath);
  //  拼接字符串（得到存放extra数据的文件路径）
  // xxxx/extra文件中保存着内存dump的类DexCode和DexClassData数据
  strcat(path, "extra");
  // 打开文件xxxx/extra
  FILE *fp1 = fopen(path,"wb+");
```

被脱壳Android应用的odex文件中的类数据以Android运行时的 DexClassData 的实际填充数据 为基准进行内存dump，意思就是内存dump的类描述结构体数据以dex文件加载到内存后转换为**ClassObject** *中的类实际填充数据为基准进行内存dump，为什么要这样做呢？ 因为，Android加固会对被保护的dex文件进行加固处理，对被保护的dex文件中的类数据偏移 classDataOff 进行修改并将类数据 DexClassData 进行加密处理，在类被执行时先对被加密的类数据 DexClassData 进行解密处理，然后修正类数据的偏移classDataOff 使其指向解密后正确的DexClassData数据（一般情况，被解密的DexClassData数据会存放在odex文件文件头之前的内存区域 或者 在odex文件文件尾之后的内存区域）；还有一些Android加固采取的加固粒度更细，对被保护dex文件的codeOff 进行修改并对类方法实现 DexCode 的数据进行加密处理，在类方法执行时，先解密被加密的类方法实现 DexCode的数据，然后修正指向DexCode数据的偏移 codeOff （一般情况，解密后的DexCode数据会被存放在在odex文件文件头之前的内存区域 或者 在odex文件文件尾之后的内存区域）。由于当dex文件中类被执行时，类相关的描述数据都会被解密后正确填充，因此基于Android运行时进行dex文件类数据的dump。

How?

- Determine whether the ***class_data_off*** in ***class_def_item*** exists in the scope of the dex file.
  - Copy all ***class_def_item***s and write them into a file named **classdef**.
  - Collect the outside ***class_data_item***s into a file named **extra**.
- Correct the fields in selected ***DexClassData*** object according to the managed method object.

13. 鉴于Android进程中内存数据 4 字节对齐的要求（目的是为了提高内存数据访问的效率），因此需要对被脱壳Android进程的odex文件的内存数据进行 4 字节对齐处理，不足4字节进行 0 的数据填充，也是为了后面odex文件重组时正确的设置odex文件类数据DexClassData和DexCode的偏移。

```
uint32_t mask = 0x3ffff;
char padding = 0;
// Android系统的类库是以android开头的类（用于过滤）
const char* header = "Landroid";

// 获取dex文件的DexClassDef的数量classDefsSize
unsigned int num_class_defs = pDexFile->pHeader->classDefsSize;

// pDexFile->baseAddr为存放dex文件的内存地址，
// mem->addr为存放odex的文件的数据的内存地址,mem->length为存放的odex文件的长度
// 获取dex文件起始到odex文件结束在内存区域的长度
uint32_t total_pointer = mem->length - uint32_t(pDexFile->baseAddr-(const u1*)mem->addr);
// 保存dex文件起始到odex文件结束在内存区域的长度
uint32_t rec = total_pointer;

// 鉴于dex文件在内存中的4字节对齐问题（4字节对齐）
while (total_pointer&3)
{
    total_pointer++;
}

// 填充的内存字节数inc（4字节内存对齐导致的填充字节数）
int inc = total_pointer - rec;
```


14.为了判断被脱壳Android应用dex文件的DexClassData数据是否被Android加固所加固处理，因此需要对DexClassData数据的偏移 classDataOff 进行边界的判断，DexClassData数据保存的起始文件偏移是dex文件存放DexClassDef段数据结束的位置，DexClassData数据保存的结束文件偏移是整个odex文件结束的位置。遍历dex文件的每个类定义描述结构体DexClassDef，基于Android运行时的类数据DexClassData和DexCode的收集；在进行dex文件类数据收集时，排除过滤掉 "Landroid" 开头的Android系统类和空类的类数据DexClassData和DexCode的收集。

```
// 在内存中dex文件存放DexClassData的起始文件相对偏移地址（即存放DexClassDef的结束地址）
uint32_t start = pDexFile->pHeader->classDefsOff+sizeof(DexClassDef)*num_class_defs;
// 在内存中dex文件起始内存地址到整个odex文件结束在内存区域的长度
uint32_t end = (uint32_t)((const u1*)mem->addr+mem->length - pDexFile->baseAddr);

// 遍历dex文件的所有的DexClassDef
for (size_t i = 0; i < num_class_defs; i++)  //Loop start ----------
{
    const u1* data = NULL;
    DexClassData* pData = NULL;

    // 设置初始值
    bool need_extra = false;
    bool pass = false;

    // 获取dex文件的第i个DexClassDef的结构体信息
    const DexClassDef *pClassDef = dexGetClassDef(pDvmDex->pDexFile, i);
    // 获取类的描述符信息即类类型描述字符串如：Landroid/xxx/yyy;
    const char *descriptor = dexGetClassDescriptor(pDvmDex->pDexFile, pClassDef);

    // 判断该类是否是Landroid开头的Android系统类，是否是一个没数据的无效类
    if(!strncmp(header, descriptor, 8) || !pClassDef->classDataOff)
    {
        // 设置跳过过滤标签
        pass = true;

        // ******是系统类或者当前类不是有效的类，直接跳转******
        goto classdef;
    }
```


对于是Android系统（Landroid开头的）的类和空类，need_extra为 false 即不对 Landroid开头 的Android系统类和空类进行类实现数据DexClassData的收集，并设置这两种情况的类DexClassDef的成员变量 classDataOff 和 annotationsOff 的文件偏移值为 0 ,还有对于这两种情况的类，只保存 类定义的数据DexClassDef 到文件 /data/data/pakegname/classdef 中。这里还需要对标志 need_extra 和 pass 的意思进行说明一下，标志**need_extra** 的意思是是否保存类的实现数据 DexClassData 到文件/data/data/pakegname/extra 中；标志 **pass**的意思是 是否设置类定义DexClassDef 的成员变量 classDataOff 和 annotationsOff 的文件偏移值为 0，标志 need_extra 和 pass 的bool值总是相反的，其中pakegname为被脱壳Android应用的包名。

```
// 当Android系统类或者当前类是无数据的空类的情况下，need_extra = false，pass = true
// 针对非Android系统有效类的情况下，need_extra = true，pass = false
classdef:
        // 获取dex文件的第i个DexClassDef结构体的信息
        DexClassDef temp = *pClassDef;
        uint8_t *p = (uint8_t *)&temp;

        // 判断该类是否需要额外保存的类的数据DexClassData和类方法的DexCode数据信息
        // 这种情况下，需要保存DexClassData结构体的数据信息
        if (need_extra)
        {
            ALOGI("GOT IT classdata before");
            int class_data_len = 0;

            // pData = ReadClassData(&data);

            // 将DexClassData结构体的数据信息leb128编码后保存到申请的内存空间中
            // out为指向DexClassData结构体的数据存放指针，class_data_len为DexClassData结构体数据的长度
            uint8_t *out = EncodeClassData(pData, class_data_len);
            if (!out)
            {
                continue;
            }

            // 设置类的DexClassDef的成员classDataOff（指向DexClassData）的相对文件偏移
            temp.classDataOff = total_pointer;

            // 将类的DexClassData数据信息写入xxxx/extra文件中
            fwrite(out, 1, class_data_len, fp1);
            // 刷新文件流
            fflush(fp1);

            // 更新文件偏移地址
            total_pointer += class_data_len;

            // 处理dex文件在内存中4字节对齐的问题
            while (total_pointer&3)
            {
                // 写入填充的数据0
                fwrite(&padding, 1, 1, fp1);
                // 刷新文件流
                fflush(fp1);

                total_pointer++;
            }

            free(out);
            ALOGI("GOT IT classdata written");
        }
```

```
            free(out);
            ALOGI("GOT IT classdata written");
        }
        else
        {
            // pData = ReadClassData(&data);
            if (pData)
            {
                free(pData);
            }
        }

        // 对系统类或者当前类不是有效的类的情况，修改classDataOff和annotationsOff值为0
        if (pass)
        {
            // 设置类信息结构体的classDataOff和annotationsOff为0
            temp.classDataOff = 0;
            temp.annotationsOff = 0;
        }

        // 将类的DexCode和DexClassData数据信息保存到xxxx/extra文件中，保存格式如下：
        // |{[DexCode#DexCode....]DexClassData}|{[DexCode#DexCode....]DexClassData}|{[DexCode#DexCode....]DexClassData}...
        // xxxx/extra文件中DexCode的文件偏移保存在DexClassData结构体的pData->virtualMethods[i].codeOff和pData->directMethods[i].codeOff中
        // xxxx/extra文件中DexClassData的文件偏移保存在xxxx/classdef文件中的DexClassDef->classDataOff中

        ALOGI("GOT IT classdef");

        // 将dex文件的第i个DexClassDef结构体的信息写入到文件xxxx/classdef中
        fwrite(p, sizeof(DexClassDef), 1, fp);
        // 刷新文件流
        fflush(fp);

    } //Loop over ----------
```


15.重点描述（突出DexHunter脱壳工具的基于运行时的脱壳）

```
// 在内存中dex文件存放DexClassData的起始文件相对偏移地址（即存放DexClassDef的结束地址）
uint32_t start = pDexFile->pHeader->classDefsOff+sizeof(DexClassDef)*num_class_defs;
// 在内存中dex文件起始内存地址到整个odex文件结束在内存区域的长度
uint32_t end = (uint32_t)((const u1*)mem->addr+mem->length - pDexFile->baseAddr);

// 遍历dex文件的所有的DexClassDef
for (size_t i = 0; i < num_class_defs; i++)  //Loop start ----------
{
    const u1* data = NULL;
    DexClassData* pData = NULL;

    // 设置初始值
    bool need_extra = false;
    bool pass = false;

    // 获取dex文件的第i个DexClassDef的结构体信息
    const DexClassDef *pClassDef = dexGetClassDef(pDvmDex->pDexFile, i);
    // 获取类的描述符信息即类类型描述字符串如：Landroid/xxx/yyy;
    const char *descriptor = dexGetClassDescriptor(pDvmDex->pDexFile, pClassDef);

    // 判断该类是否是Landroid开头的Android系统类，是否是一个没数据的无效类
    if(!strncmp(header, descriptor, 8) || !pClassDef->classDataOff)
    {
        // 设置跳过过滤标签
        pass = true;

        // ******是系统类或者当前类不是有效的类，直接跳转******
        goto classdef;
    }

    // 调用函数dvmDefineClass加载当前类descriptor
    ClassObject * clazz=NULL;
    clazz = dvmDefineClass(pDvmDex, descriptor, loader);
    if (!clazz)
    {
       continue;
    }

    // 打印加载的类描述符信息
    ALOGI("GOT IT 加载class: %s", descriptor);

    // 判断加载的指定的类是否已经初始化完成
    if (!dvmIsClassInitialized(clazz))
    {
        if(dvmInitClass(clazz))
        {
            ALOGI("GOT IT init: %s", descriptor);
        }
    }
```

检查的边界

1

2

未完待续~