导读

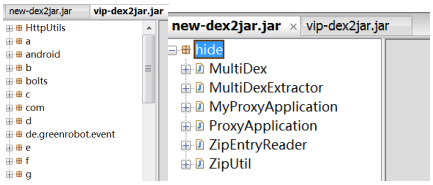
Android里的代码隐藏方案可以说有很多种实现方式，我在这里尝试了一种类似加壳的方式。在此方式上，可以作更多的扩展，如加密等。因为为了最大限度保持兼容性，很多地方没有作更深入地发掘，被逆向人员找出隐藏的代码不是难事。但此方法在某些场景还是有用的，例如规避超64k方法数限制，部分代码从网上加载再执行等。

此方法当作抛砖引玉，希望有更多人分享自己的想法。

0、作用概述
1、实现隐藏流程
2、壳作了哪些工作及原理
3、释放原代码的工作及原理
4、注意事项
5、参考资料

0、作用概述

唯品会Android客户端隐藏代码前与隐藏代码后的dex2jar对比 如图



1、实现隐藏流程

本人写了个脚本来跑整个隐藏代码的过程，脚本的流程就是实现隐藏的流程：

- 逆向原apk包获得AndroidManifest
- 修改AndroidManifest里的Application入口为壳
- 把原apk包的classes.dex经过压缩（加密）后放在assets文件夹（网络）里
- 把原apk包的classes.dex替换为壳的classes.dex
- 重打包

2、壳作了哪些工作及原理

壳主主要作了两个工作：

- 释放隐藏的原代码
- 还原运行环境给原代码
- 调用原代码的Application

这里感谢一下lucifazhang(张强)和darongchen(陈春荣)的指导，因本人之前没接触过这方面知识所以连google哪些关键词也不太懂。

还原运行环境的原理：
我们需要通过java反射修改的变量如下，分析下面说明

- ActivityThread.mInitialApplication Application
- ActivityThread.mAllApplications ArrayList<Application>
- ContextImpl.mOuterContext Context
- LoadedApk.mApplication Application

首先，我们需要对程序的Application何时创建有个完整的了解。

newApplication调用链一 如图

我们看到在SystemSever和ActivityThread的main函数中都会最终调用newApplication，但通过源码分析ActivityThread.main调用的是attach(false)并不会newApplication，而ActivityThread.systemMain调用attach(true)会调用newApplication，因此源头为SystemSever.main。

经过对SystemSever的一番搜索探究，可得出SystemSever为Zygote进程启动过程的其中一个步骤，因此整个调用的逻辑清晰浮现。

Zygote进程启动过程 如下
1.创建AppRuntime对象，并调用它的start函数；
2.调用startVm创建Java虚拟机；
3.调用startReg函数来注册JNI函数；
4.调用ZygoteInit类的main函数，从此就进入了Java世界；
5.调用registerZygoteSocket 注册一个服务端socket；
6.调用preloadClasses 函数加载类资源；
7.调用preloadResources函数加载系统资源；
8.调用startSystemServer函数创建SystemServer进程；
9.调用runSelectLoopMode函数进入服务端socket监听；

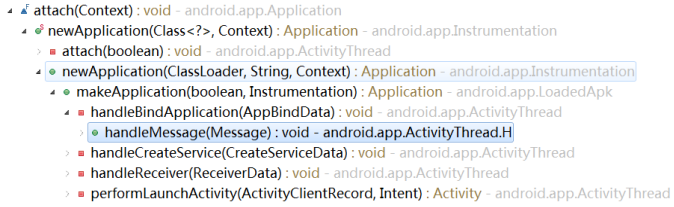
接下来我们可在上述调用链中看出有哪些需要修改的地方，在ActivityThread.attach中得到

```
1.                  mInstrumentation = new Instrumentation();
2.                ContextImpl context = ContextImpl.createAppContext(
3.                      this, getSystemContext().mPackageInfo);
4.                Application app = Instrumentation.newApplication(Application.class, context);
                    mAllApplications.add(app);
6.                  mInitialApplication = app;
7.                  app.onCreate();
```

直接可得ActivityThread实例对象中的mInitialApplication和mAllApplications这两个变量是记录Application需要修改。

---

newApplication调用链二 如图



如此类推，ActivityThread.H为ActivityThread的内部Handler类，用来处理各种消息。我们追踪第二条调用链时可看到在LoadApk.makeApplication中得到

```
1.                java.lang.ClassLoader cl = getClassLoader();
2.              ContextImpl appContext = ContextImpl.createAppContext(mActivityThread, this);
3.                app = mActivityThread.mInstrumentation.newApplication(
4.                        cl, appClass, appContext);
                  appContext.setOuterContext(app);
```

和

```
1.    mApplication = app;
```

LoadApk.mApplication直接看出是需要修改的，再通过
appContext.setOuterContext(app);得出ContextImpl.mOuterContext记录的其实是一个Application对象，因此也一并修改掉。

最后通过查看源码得出我们可通过反射context修改的地方如下：
context.mOuterContext
context.mPackageInfo.mApplication
context.mPackageInfo.mActivityThread.mInitialApplication
context.mPackageInfo.mActivityThread.mAllApplications

调用原代码的Application：
回头看看public Application newApplication(Class<?> clazz, Context context)函数得到

```
1.      static public Application newApplication(Class<?> clazz, Context context)
2.             throws InstantiationException, IllegalAccessException,
3.             ClassNotFoundException {
4.          Application app = (Application)clazz.newInstance();
             app.attach(context);
6.          return app;
7.      }
```

结合ActivityThread.attach代码，我们只需通过反射还原四个变量为原代码的Application后调用原Application的attach（context）和onCreate即可。

3、释放原代码的工作及原理

通常使用自定义的ClassLoader来加载，但我在开始尝试时发现google出了一个support包支持加载多个dex文件，因此转而研究了这个support包，发现其兼容性不错。
原理分析看我之前的文章：android.support.multidex.jar源码阅读分析—加载多个dex文件
但那个是官方的包，我们必须要改一下释放的代码，只要增加一个函数就可以。
MultiDexExtractor：

```
    static List<File> load2(Context context, ApplicationInfo applicationInfo, File dexDir,
           boolean forceReload) throws Exception {
      dexDir.mkdirs();

      String strHiddenFile="classes.zip";
      AssetManager am=context.getAssets();

    InputStream is=am.open(strHiddenFile);
    File fleClzZip = new File(dexDir,strHiddenFile);
    Inputstream2File(is, fleClzZip);

    List<File> files=new ArrayList<File>();
    files.add(fleClzZip);

        return files;
     }
```

MultiDex.install:
List<File> files = MultiDexExtractor.*load*(context, applicationInfo, dexDir, **false**); 改为
List<File> files = MultiDexExtractor.*load2*(context, applicationInfo, dexDir, **false**);

如果做了加密的话这里load2就加上解密的方法。

4、参考、注意事项

注意事项一　ContentProvider的处理
ContentProvider初始化在Application的attachBaseContext()之后，onCreate()之前。
我测试过在attachBaseContext()还原程序环境是可行的，这样下面的处理就不需要了，但如果要在onCreate()函数里还原程序环境，则需要作以下处理，先来看看原理。

四大组件中的Content Provider有getContext但其并不是从ContextWrapper继承下来的。从ContentProvider源码中可得知，内部有一个mContext变量记录着，而mContext的初始化如下。
从ActivityThread.installProvider得到

```
            Context c = null;
            ApplicationInfo ai = info.applicationInfo;
            if (context.getPackageName().equals(ai.packageName)) {
                  c = context;
            } else if (mInitialApplication != null &&
                      mInitialApplication.getPackageName().equals(ai.packageName)) {
                  c = mInitialApplication;
            } else {
                try {
                      c = context.createPackageContext(ai.packageName,
                          Context.CONTEXT_INCLUDE_CODE);
                } catch (PackageManager.NameNotFoundException e) {
                    // Ignore
                }
            }
```

如果context.getPackageName()不等于info.applicationInfo.packageName，c就等于mInitialApplication，而c最终赋值到ContentProvider的mContext上，因此只要在壳的代码加上

```
    @Override
    public String getPackageName() {
     return "";
    }
```

在壳的attachBaseContext加上

```
pName=super.getPackageName();
```

另还要在释放原代码的代码里改
MultiDex.getApplicationInfo加上

```
packageName = ProxyApplication.pName;
```

Done!

注意事项二 不释放未加密的odex文件在文件系统的扩展方案
这个方案可实现简单的加密解密，而且兼容性杠杠的，但最终还是会释放未加密的odex文件在文件系统里。所以可在此方案上扩展，仿照系统里的 **Dalvik_dalvik_system_DexFile_openDexFile_bytearray**函数直接构造为DexFile。**但，5.0之后完全取消了dalvik，利用此函数可能需要携带不少libdex的代码。因此再在上面扩展很可能要考虑各种麻烦的适配问题，兼容性也会下降。**

对于适配5.0之后的扩展方案，我有两个思路，大家有空可以一起探讨。
一、同时携带加密的odex、oat文件，直接在内存解密后根据运行环境让虚拟机用。
二、携带加密的dex文件，同时自己实现加密的dex文件在内存里转为odex或oat文件，按需转换并返回给虚拟机。
方案一运行速度比较快，而且实现起来比较简单，但等于携带了两份代码，安装包体积可能会飙升。方案二携带的代码大部分情况会比方案一少，但要经历转化和解密等，运行效率会较低，而且实现难度比较大。

解密后直接在内存构造dexfile可参考以下代码：
dalvik_system_DexFile.cpp中的Dalvik_dalvik_system_DexFile_openDexFile_bytearray函数（4.0以后才有）

```
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);

    RETURN_PTR(pDexOrJar);
}
```

类似功能的还有DexFile.cpp的dexFileParse函数

```
DexFile* dexFileParse(const u1* data, size_t length, int flags)
{
    DexFile* pDexFile = NULL;
    const DexHeader* pHeader;
```

```
    const u1* magic;
    int result = -1;

    if (length < sizeof(DexHeader)) {
        ALOGE("too short to be a valid .dex");
        goto bail;      /* bad file format */
    }

    pDexFile = (DexFile*) malloc(sizeof(DexFile));
    if (pDexFile == NULL)
        goto bail;      /* alloc failure */
    memset(pDexFile, 0, sizeof(DexFile));

    /*
     * Peel off the optimized header.
     */
    if (memcmp(data, DEX_OPT_MAGIC, 4) == 0) {
        magic = data;
        if (memcmp(magic+4, DEX_OPT_MAGIC_VERS, 4) != 0) {
            ALOGE("bad opt version (0x%02x %02x %02x %02x)",
                magic[4], magic[5], magic[6], magic[7]);
            goto bail;
        }

        pDexFile->pOptHeader = (const DexOptHeader*) data;
        ALOGV("Good opt header, DEX offset is %d, flags=0x%02x",
            pDexFile->pOptHeader->dexOffset, pDexFile->pOptHeader->flags);

        /* parse the optimized dex file tables */
        if (!dexParseOptData(data, length, pDexFile))
            goto bail;

        /* ignore the opt header and appended data from here on out */
        data += pDexFile->pOptHeader->dexOffset;
        length -= pDexFile->pOptHeader->dexOffset;
        if (pDexFile->pOptHeader->dexLength > length) {
            ALOGE("File truncated? stored len=%d, rem len=%d",
                pDexFile->pOptHeader->dexLength, (int) length);
            goto bail;
        }
        length = pDexFile->pOptHeader->dexLength;
    }

    dexFileSetupBasicPointers(pDexFile, data);
    pHeader = pDexFile->pHeader;

    if (!dexHasValidMagic(pHeader)) {
        goto bail;
    }

    /*
     * Verify the checksum(s).  This is reasonably quick, but does require
     * touching every byte in the DEX file.  The base checksum changes after
     * byte-swapping and DEX optimization.
     */
    if (flags & kDexParseVerifyChecksum) {
        u4 adler = dexComputeChecksum(pHeader);
        if (adler != pHeader->checksum) {
            ALOGE("ERROR: bad checksum (%08x vs %08x)",
                adler, pHeader->checksum);
            if (!(flags & kDexParseContinueOnError))
                goto bail;
        } else {
            ALOGV("+++ adler32 checksum (%08x) verified", adler);
        }

        const DexOptHeader* pOptHeader = pDexFile->pOptHeader;
        if (pOptHeader != NULL) {
            adler = dexComputeOptChecksum(pOptHeader);
            if (adler != pOptHeader->checksum) {
                ALOGE("ERROR: bad opt checksum (%08x vs %08x)",
                    adler, pOptHeader->checksum);
                if (!(flags & kDexParseContinueOnError))
                    goto bail;
            } else {
                ALOGV("+++ adler32 opt checksum (%08x) verified", adler);
            }
        }
    }

    /*
```

```
        * Verify the SHA-1 digest.  (Normally we don't want to do this --
        * the digest is used to uniquely identify the original DEX file, and
        * can't be computed for verification after the DEX is byte-swapped
        * and optimized.)
        */
       if (kVerifySignature) {
           unsigned char sha1Digest[kSHA1DigestLen];
           const int nonSum = sizeof(pHeader->magic) + sizeof(pHeader->checksum) +
                              kSHA1DigestLen;
            dexComputeSHA1Digest(data + nonSum, length - nonSum, sha1Digest);
           if (memcmp(sha1Digest, pHeader->signature, kSHA1DigestLen) != 0) {
               char tmpBuf1[kSHA1DigestOutputLen];
               char tmpBuf2[kSHA1DigestOutputLen];
                ALOGE("ERROR: bad SHA1 digest (%s vs %s)",
                    dexSHA1DigestToStr(sha1Digest, tmpBuf1),
                    dexSHA1DigestToStr(pHeader->signature, tmpBuf2));
               if (!(flags & kDexParseContinueOnError))
                   goto bail;
           } else {
                ALOGV("+++ sha1 digest verified");
           }
       }

       if (pHeader->fileSize != length) {
            ALOGE("ERROR: stored file size (%d) != expected (%d)",
               (int) pHeader->fileSize, (int) length);
           if (!(flags & kDexParseContinueOnError))
               goto bail;
       }

       if (pHeader->classDefsSize == 0) {
            ALOGE("ERROR: DEX file has no classes in it, failing");
           goto bail;
       }

       /*
        * Success!
        */
       result = 0;

   bail:
       if (result != 0 && pDexFile != NULL) {
            dexFileFree(pDexFile);
            pDexFile = NULL;
       }
       return pDexFile;
   }
```

5、参考资料

Context详解：Android中Context详解 ---- 你所不知道的Context

ActivityThread启动原理:Android应用程序进程启动过程的源代码分析

SystemServer启动原理：Zygote进程启动过程源代码分析

android.support.multidex官方介绍：Building Apps with Over 65K Methods

android.support.multidex我的原理分析：android.support.multidex.jar源码阅读分析—加载多个dex文件

壳的还原原理：Android的Proxy/Delegate Application框架

Dex动态加载:Android4.0内存Dex数据动态加载技术