**Android Hook框架Xposed详解：从源代码分析到开发指南**

2016年11月27日 18:16:46 [双刃剑客](https://me.csdn.net/zhangmiaoping23) 阅读数：6249更多

个人分类： [Android注入与HOOK](https://blog.csdn.net/zhangmiaoping23/article/category/6439909)[XPosed](https://blog.csdn.net/zhangmiaoping23/article/category/6414377)[AndroidHook](https://blog.csdn.net/zhangmiaoping23/article/category/6414376)

转：http://www.sjsjw.com/kf\_mobile/article/1\_19087\_30637.asp

**1 Introduction**

**1.1 概述**

Xposed是GitHUB上rovo89大大设计的一个针对Android平台的动态劫持项目，通过替换/system/bin/app\_process程序控制zygote进程，使得app\_process在启动过程中会加载XposedBridge.jar这个jar包，从而完成对系统应用的劫持。   
Xposed框架的基本运行环境如下：

|  |  |
| --- | --- |
| Configuration | RequireMent |
| Root Access | 因为Xposed工作原理是在/system/bin目录下替换文件，在install的时候需要root权限，但是运行时不需要root权限。 |
| 版本要求 | 需要在Android 4.0以上版本的机器中 |

1. Xposed在线资源可以参照：http://forum.xda-developers.com/showthread.php?t=1574401

                                                                https://github.com/rovo89   
         2.  GitHub上的Xposed资源梳理一下，可以这么分类： 

* XposedBridge.jar：XposedBridge.jar是Xposed提供的jar文件，负责在Native层与FrameWork层进行交互。/system/bin/app\_process进程启动过程中会加载该jar包，其它的Modules的开发与运行都是基于该jar包的。
* Xposed：Xposed的C++部分，主要是用来替换/system/bin/app\_process，并为XposedBridge提供JNI方法。
* XposedInstaller：Xposed的安装包，负责配置Xposed工作的环境并且提供对基于Xposed框架的Modules的管理。
* XposedMods：使用Xposed开发的一些Modules，其中AppSettings是一个可以进行权限动态管理的应用

**1.2 Mechanism：原理**

**1.2.1 Zygote**

在Android系统中，应用程序进程都是由Zygote进程孵化出来的，而Zygote进程是由Init进程启动的。Zygote进程在启动时会创建一个Dalvik虚拟机实例，每当它孵化一个新的应用程序进程时，都会将这个Dalvik虚拟机实例复制到新的应用程序进程里面去，从而使得每一个应用程序进程都有一个独立的Dalvik虚拟机实例。   
Zygote进程在启动的过程中，除了会创建一个Dalvik虚拟机实例之外，还会将Java运行时库加载到进程中来，以及注册一些Android核心类的JNI方法来前面创建的Dalvik虚拟机实例中去。注意，一个应用程序进程被Zygote进程孵化出来的时候，不仅会获得Zygote进程中的Dalvik虚拟机实例拷贝，还会与Zygote一起共享Java运行时库。这也就是可以将XposedBridge这个jar包加载到每一个Android应用程序中的原因。XposedBridge有一个私有的Native（JNI）方法hookMethodNative，这个方法也在app\_process中使用。这个函数提供一个方法对象利用Java的Reflection机制来对内置方法覆写。具体的实现可以看下文的Xposed源代码分析。

**1.2.2 Hook/Replace**

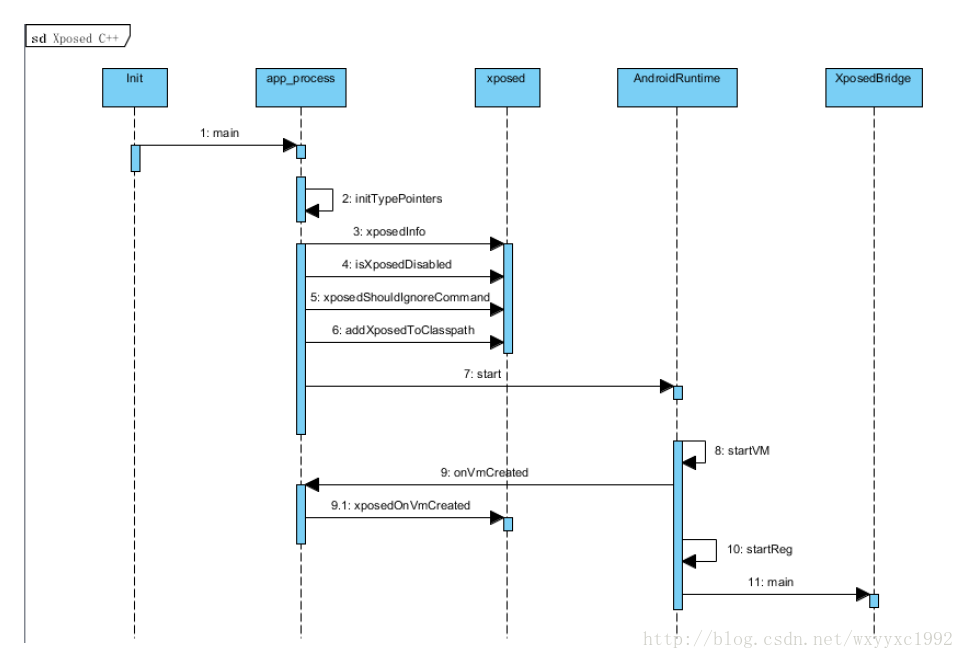
Xposed 框架中真正起作用的是对方法的hook。在Repackage技术中，如果要对APK做修改，则需要修改Smali代码中的指令。而另一种动态修改指令的技术需要在程序运行时基于匹配搜索来替换smali代码，但因为方法声明的多样性与复杂性，这种方法也比较复杂。   
在Android系统启动的时候，zygote进程加载XposedBridge将所有需要替换的Method通过JNI方法hookMethodNative指向Native方法xposedCallHandler，xposedCallHandler在转入handleHookedMethod这个Java方法执行用户规定的Hook Func。   
XposedBridge这个jar包含有一个私有的本地方法：hookMethodNative，该方法在附加的app\_process程序中也得到了实现。它将一个方法对象作为输入参数（你可以使用Java的反射机制来获取这个方法）并且改变Dalvik虚拟机中对于该方法的定义。它将该方法的类型改变为native并且将这个方法的实现链接到它的本地的通用类的方法。换言之，当调用那个被hook的方法时候，通用的类方法会被调用而不会对调用者有任何的影响。在hookMethodNative的实现中，会调用XposedBridge中的handleHookedMethod这个方法来传递参数。handleHookedMethod这个方法类似于一个统一调度的Dispatch例程，其对应的底层的C++函数是xposedCallHandler。而handleHookedMethod实现里面会根据一个全局结构hookedMethodCallbacks来选择相应的hook函数，并调用他们的before, after函数。   
当多模块同时Hook一个方法的时候，Xposed会自动根据Module的优先级来排序，调用顺序如下：   
A.before -> B.before -> original method -> B.after -> A.after 

**2 源代码分析**

**2.1 Cpp模块**

该部分的源代码地址是：https://github.com/rovo89/Xposed，其文件分类如下： 

* app\_main.cpp：类似AOSP中的frameworks/base/cmds/app\_process/app\_main.cpp，即/system/bin/app\_process这个zygote真实身份的应用程序的源代码。关于zygote进程的分析可以参照Android:AOSP&Core中的Zygote进程详解。
* xposed.cpp：提供给app\_main.cpp的调用函数以及XposedBridge的JNI方法的实现。主要完成初始化工作以及Framework层的Method的Hook操作。
* xposed.h，xposed\_offsets.h：头文件

  Xposed框架中的app\_main.cpp相对于AOSP的app\_main.cpp中修改之处主要为区分了调用runtime.start()函数的逻辑。Xposed框架中的app\_main.cpp在此处会根据情况选择是加载XposedBridge类还是ZygoteInit或者RuntimeInit类。而实际的加载XposedBridge以及注册JNI方法的操作发生在第四步：xposedOnVmCreated中。   
1．包含cutils/properties.h，主要用于获取、设置环境变量，xposed.cpp中需要将XposedBridge设置到ClassPath中。   
2．包含了dlfcn.h，用于对动态链接库的操作。   
3．包含了xposed.h，需要调用xposed.cpp中的函数，譬如在虚拟机创建时注册JNI函数。   
4．增加了initTypePointers函数，对于Android SDK大于等于18的会获取到atrace\_set\_tracing\_enabled函数指针，在Zygote启动时调用。   
5．AppRuntime类中的onVmCreated函数中增加xposedOnVmCreated函数调用。   
6．源代码中的Log\*全部重命名为ALog\*,所以Logv替换为Alogv，但是功能不变。   
7．Main函数开始处增加了大量的代码，但是对于SDK版本小于16的可以不用考虑。   


**2.1.1 Main函数：zygote入口**

1. int main(int argc, char\* const argv[])
2. {
3. ...
4. initTypePointers();
5. */\*该函数对于SDK>=18的会获取到atrace\_set\_tracing\_enabled的函数指针，获取到的指针会在Zygote初始化过程中调用，函数定义见代码段下方\*/*
6. ...
7. xposedInfo();
8. */\*xposedInfo函数定义在xposed.cpp中，该函数主要获取一些属性值譬如SDK版本，设备厂商，设备型号等信息并且打印到Log文件中\*/*
9. xposedEnforceDalvik();
10. keepLoadingXposed = !isXposedDisabled() && !xposedShouldIgnoreCommand(className, argc, argv) && addXposedToClasspath(zygote);
12. if (zygote) {
13. runtime.start(keepLoadingXposed ? XPOSED\_CLASS\_DOTS : "com.android.internal.os.ZygoteInit",
14. startSystemServer ? "start-system-server" : "");
15. } else if (className) {
16. *// Remainder of args get passed to startup class main()*
17. runtime.mClassName = className;
18. runtime.mArgC = argc - i;
19. runtime.mArgV = argv + i;
20. runtime.start(keepLoadingXposed ? XPOSED\_CLASS\_DOTS : "com.android.internal.os.RuntimeInit",
21. application ? "application" : "tool");
22. }
23. else
24. {
25. fprintf(stderr, "Error: no class name or --zygote supplied.\n");
26. app\_usage();
27. LOG\_ALWAYS\_FATAL("app\_process: no class name or --zygote supplied.");
28. return 10;
29. }
30. }
31. void initTypePointers()
32. {
33. char sdk[PROPERTY\_VALUE\_MAX];
34. const char \*error;
35. property\_get("ro.build.version.sdk", sdk, "0");
36. RUNNING\_PLATFORM\_SDK\_VERSION = atoi(sdk);
37. dlerror();
38. if (RUNNING\_PLATFORM\_SDK\_VERSION >= 18) {
39. \*(void \*\*) (&PTR\_atrace\_set\_tracing\_enabled) = dlsym(RTLD\_DEFAULT, "atrace\_set\_tracing\_enabled");
40. if ((error = dlerror()) != NULL) {
41. ALOGE("Could not find address for function atrace\_set\_tracing\_enabled: %s", error);
42. }
43. }
44. }

上述代码中的keepLoadingXposed 变量主要用于判断是否需要继续加载Xposed框架，其中isXposedDisabled、xposedShouldIgnoreCommand以及addXposedToClasspath都定义在xposed.cpp中。   
 

**2.1.2 keepLoadingXposed：判断是否需要加载XposedBridge**

    
1.isXposedDisabled

1. bool isXposedDisabled() {
2. *// is the blocker file present?*
3. if (access(XPOSED\_LOAD\_BLOCKER, F\_OK) == 0) {
4. ALOGE("found %s, not loading Xposed\n", XPOSED\_LOAD\_BLOCKER);
5. return true;
6. }
7. return false;
8. }

该函数通过读取/data/data/de.robv.android.xposed.installer/conf/disabled文件（Xposed框架通过XposedInstaller管理，需要安装此APK文件），来判断Xposed框架是否被禁用，如果该文件存在，则表示禁用Xposed。   
2.xposedShouldIgnoreCommand   
为了避免Superuser类似工具滥用Xposed的log文件，此函数会判断是否是SuperUser等工具的启动请求。

1. *// ignore the broadcasts by various Superuser implementations to avoid spamming the Xposed log*
2. bool xposedShouldIgnoreCommand(const char\* className, int argc, const char\* const argv[]) {
3. if (className == NULL || argc < 4 || strcmp(className, "com.android.commands.am.Am") != 0)
4. return false;
5. if (strcmp(argv[2], "broadcast") != 0 && strcmp(argv[2], "start") != 0)
6. return false;
7. bool mightBeSuperuser = false;
8. for (int i = 3; i < argc; i++) {
9. if (strcmp(argv[i], "com.noshufou.android.su.RESULT") == 0
10. || strcmp(argv[i], "eu.chainfire.supersu.NativeAccess") == 0)
11. return true;
12. if (mightBeSuperuser && strcmp(argv[i], "--user") == 0)
13. return true;
14. char\* lastComponent = strrchr(argv[i], '.');
15. if (!lastComponent)
16. continue;
17. if (strcmp(lastComponent, ".RequestActivity") == 0
18. || strcmp(lastComponent, ".NotifyActivity") == 0
19. || strcmp(lastComponent, ".SuReceiver") == 0)
20. mightBeSuperuser = true;
21. }
22. return false;
23. }

3.addXposedToClasspath   
若有新版本的XposedBridge，重命名为XposedBridge.jar并返回false;判断XposedBridge.jar文件是否存在，若不存在，返回false，否则将XposedBridge.jar添加到CLASSPATH环境变量中，返回true。

**2.1.3 runtime.start()：初始化Dalvik虚拟机**

一般情况下keepLoadingXposed值为true，以启动Zygote为例(zygote==true)，分析接下来的代码。   
runtime.start(keepLoadingXposed ? XPOSED\_CLASS\_DOTS : "com.android.internal.os.RuntimeInit",   
                application ? "application" : "tool");   
这一行代码是根据keepLoadingXposed 的值来判断是加载Xposed框架还是正常的ZygoteInit类。keepLoadingXposed值为true,则会加载XPOSED\_CLASS\_DOTS类，XPOSED\_CLASS\_DOTS值为de.robv.android.xposed.XposedBridge，即XposedBridge类。   
runtime是AppRuntime的实例，AppRuntime继承自AndroidRuntime。

1. ......
2. static AndroidRuntime\* gCurRuntime = NULL;
3. ......
4. AndroidRuntime::AndroidRuntime()
5. {
6. ......
7. assert(gCurRuntime == NULL); *// one per process*
8. gCurRuntime = this;
9. }

AndroidRuntime::start(const char\* className, const char\* options)函数完成Dalvik虚拟机的初始化和启动以及运行参数className指定的类中的main方法。当启动完虚拟机后，会调用onVmCreated(JNIEnv\* env)函数。该函数在AppRuntime类中被覆盖。因此直接看AppRuntime::onVmCreated(JNIEnv\* env)。

1. virtual void onVmCreated(JNIEnv\* env)
2. {
3. keepLoadingXposed = xposedOnVmCreated(env, mClassName);
4. if (mClassName == NULL) {
5. return; *// Zygote. Nothing to do here.*
6. }
7. char\* slashClassName = toSlashClassName(mClassName);
8. mClass = env->FindClass(slashClassName);
9. if (mClass == NULL) {
10. ALOGE("ERROR: could not find class '%s'\n", mClassName);
11. }
12. free(slashClassName);
13. mClass = reinterpret\_cast<jclass>(env->NewGlobalRef(mClass));
14. }

Xposed相对于AOSP就增加了如下代码：   
keepLoadingXposed = xposedOnVmCreated(env, mClassName);   
调用了xposed.cpp中的xposedOnVmCreated(JNIEnv\* env, const char\* className)函数。

**2.1.4 xposedOnVmCreated：加载Xposedbridge**

该函数的主要作用如下：   
    
1.根据JIT是否存在对部分结构体中的成员偏移进行初始化。   
xposedInitMemberOffsets();   
即时编译（Just-in-time Compilation，JIT），又称动态转译（Dynamic Translation），是一种通过在运行时将字节码翻译为机器码，从而改善字节码编译语言性能的技术。   
    
2.禁用部分访问检查

1. *// disable some access checks*
2. patchReturnTrue((uintptr\_t) &dvmCheckClassAccess);
3. patchReturnTrue((uintptr\_t) &dvmCheckFieldAccess);
4. patchReturnTrue((uintptr\_t) &dvmInSamePackage);
5. if (access(XPOSED\_DIR "conf/do\_not\_hook\_dvmCheckMethodAccess", F\_OK) != 0)
6. patchReturnTrue((uintptr\_t) &dvmCheckMethodAccess);

3.针对MIUI操作系统移除android.content.res.MiuiResources类的final修饰符   
    jclass miuiResourcesClass = env->FindClass(MIUI\_RESOURCES\_CLASS);   
    if (miuiResourcesClass != NULL) {   
        ClassObject\* clazz = (ClassObject\*)dvmDecodeIndirectRef(dvmThreadSelf(), miuiResourcesClass);   
        if (dvmIsFinalClass(clazz)) {   
            ALOGD("Removing final flag for class '%s'", MIUI\_RESOURCES\_CLASS);   
            clazz->accessFlags &= ~ACC\_FINAL;   
        }   
    }   
    
4. 获取XposeBridge类并new一个全局引用   
 xposedClass = env->FindClass(XPOSED\_CLASS);   
    xposedClass = reinterpret\_cast<jclass>(env->NewGlobalRef(xposedClass));   
       
    if (xposedClass == NULL) {   
        ALOGE("Error while loading Xposed class '%s':\n", XPOSED\_CLASS);   
        dvmLogExceptionStackTrace();   
        env->ExceptionClear();   
        return false;   
    }   
    
5. 注册JNI函数，xposed.cpp中定义了供XposedBridge类使用的JNI方法，此处进行注册，这样当XposeBridge中的main函数执行时，就可以调用xposed.cpp中定义的JNI方法   
 if (register\_de\_robv\_android\_xposed\_XposedBridge(env) != JNI\_OK) {   
        ALOGE("Could not register natives for '%s'\n", XPOSED\_CLASS);   
        return false;   
    }   
Xposed中JNI方法有：

1. static const JNINativeMethod xposedMethods[] = {
2. {"getStartClassName", "()Ljava/lang/String;", (void\*)de\_robv\_android\_xposed\_XposedBridge\_getStartClassName},
3. {"initNative", "()Z", (void\*)de\_robv\_android\_xposed\_XposedBridge\_initNative},
4. {"hookMethodNative", "(Ljava/lang/reflect/Member;Ljava/lang/Class;ILjava/lang/Object;)V", (void\*)de\_robv\_android\_xposed\_XposedBridge\_hookMethodNative},
5. };
6. static jobject de\_robv\_android\_xposed\_XposedBridge\_getStartClassName(JNIEnv\* env, jclass clazz) {
7. return env->NewStringUTF(startClassName);
8. }
9. static int register\_de\_robv\_android\_xposed\_XposedBridge(JNIEnv\* env) {
10. return env->RegisterNatives(xposedClass, xposedMethods, NELEM(xposedMethods));
11. }
12. static const JNINativeMethod xresourcesMethods[] = {
13. {"rewriteXmlReferencesNative", "(ILandroid/content/res/XResources;Landroid/content/res/Resources;)V", (void\*)android\_content\_res\_XResources\_rewriteXmlReferencesNative},
14. };
15. static int register\_android\_content\_res\_XResources(JNIEnv\* env) {
16. return env->RegisterNatives(xresourcesClass, xresourcesMethods, NELEM(xresourcesMethods));
17. }

注册的JNI方法见xposedMethods数组。   
至于这些JNI方法的用处，在后续XposedBridge的main函数调用中会继续分析。   
    
如果对Zygote启动过程熟悉的话，对后续XposedBridge的main函数是如何被调用的应该会很清楚。AndroidRuntime.cpp的start(const char\* className, const char\* options)函数完成环境变量的设置，Dalvik虚拟机的初始化和启动，同时Xposed在onVmCreated(JNIEnv\* env)中完成了自身JNI方法的注册。此后start()函数会注册Android系统的JNI方法，调用传入的className指定类的main方法，进入Java世界。

1. void AndroidRuntime::start(const char\* className, const bool startSystemServer)
2. {
3. ......
4. char\* slashClassName = NULL;
5. char\* cp;
6. JNIEnv\* env;
7. ......
8. */\* start the virtual machine \*/*
9. if (startVm(&mJavaVM, &env) != 0)
10. goto bail;
11. */\**
12. *\* Register android functions.*
13. *\*/*
14. if (startReg(env) < 0) {
15. LOGE("Unable to register all android natives\n");
16. goto bail;
17. }
18. */\**
19. *\* We want to call main() with a String array with arguments in it.*
20. *\* At present we only have one argument, the class name. Create an*
21. *\* array to hold it.*
22. *\*/*
23. jclass stringClass;
24. jobjectArray strArray;
25. jstring classNameStr;
26. jstring startSystemServerStr;
27. stringClass = env->FindClass("java/lang/String");
28. assert(stringClass != NULL);
29. strArray = env->NewObjectArray(2, stringClass, NULL);
30. assert(strArray != NULL);
31. classNameStr = env->NewStringUTF(className);
32. assert(classNameStr != NULL);
33. env->SetObjectArrayElement(strArray, 0, classNameStr);
34. startSystemServerStr = env->NewStringUTF(startSystemServer ?
35. "true" : "false");
36. env->SetObjectArrayElement(strArray, 1, startSystemServerStr);
37. */\**
38. *\* Start VM. This thread becomes the main thread of the VM, and will*
39. *\* not return until the VM exits.*
40. *\*/*
41. jclass startClass;
42. jmethodID startMeth;
43. slashClassName = strdup(className);
44. for (cp = slashClassName; \*cp != '\0'; cp++)
45. if (\*cp == '.')
46. \*cp = '/';
47. startClass = env->FindClass(slashClassName);
48. if (startClass == NULL) {
49. ......
50. } else {
51. startMeth = env->GetStaticMethodID(startClass, "main",
52. "([Ljava/lang/String;)V");
53. if (startMeth == NULL) {
54. ......
55. } else {
56. env->CallStaticVoidMethod(startClass, startMeth, strArray);
57. ......
58. }
59. }
60. ......
61. }

由于此时参数className为de.robv.android.xposed.XposedBridge，因此会调用XposedBridge类的main方法，XposedBridge源码https://github.com/rovo89/XposedBridge。 

**2.2 Java模块**

**2.2.1 Main函数：获取所启动的类名**

进入XposedBridge的main函数，首先获取所启动的类名。   
String startClassName = getStartClassName();   
getStartClassName()是一个JNI方法，定义在xposed.cpp中。   
static jobject de\_robv\_android\_xposed\_XposedBridge\_getStartClassName(JNIEnv\* env, jclass clazz) {   
    return env->NewStringUTF(startClassName);   
}   
startClassName变量在xposedOnVmCreated函数中被赋予需启动的类的名称。   
bool xposedOnVmCreated(JNIEnv\* env, const char\* className) {   
    startClassName = className;   
...   
}   
但是需要注意的是，若是启动Zygote，则此时startClassName值为null。如下代码（app\_main.cpp）所示，当zygote为true，即启动Zygote时，并没有给AppRuntime实例runtime的mClassName成员赋值。   
...    
if (zygote) {   
        runtime.start(keepLoadingXposed ? XPOSED\_CLASS\_DOTS : "com.android.internal.os.ZygoteInit",   
                startSystemServer ? "start-system-server" : "");   
    } else if (className) {   
        // Remainder of args get passed to startup class main()   
        runtime.mClassName = className;   
        runtime.mArgC = argc - i;   
        runtime.mArgV = argv + i;   
        runtime.start(keepLoadingXposed ? XPOSED\_CLASS\_DOTS : "com.android.internal.os.RuntimeInit",   
                application ? "application" : "tool");   
}   
...   
startClassName的赋值过程为：AppRuntime中mClassName成员初始值为NULL；在app\_main.cpp中的main函数中根据arg参数解析获得className，若是启动Zygote，则className值为NULL，否则若className有值，则赋值给AppRuntime实例runtime的mClassName成员变量；调用runtime.start(…)，进一步调用onVmCreated(…)，在onVmCreated函数中调用xposedOnVmCreated(…)，并传入mClassName值，xposedOnVmCreated函数将mClassName赋值给全局变量startClassName;   
jobject de\_robv\_android\_xposed\_XposedBridge\_getStartClassName(…)将此全局变量startClassName转换为Java字符串返回。 

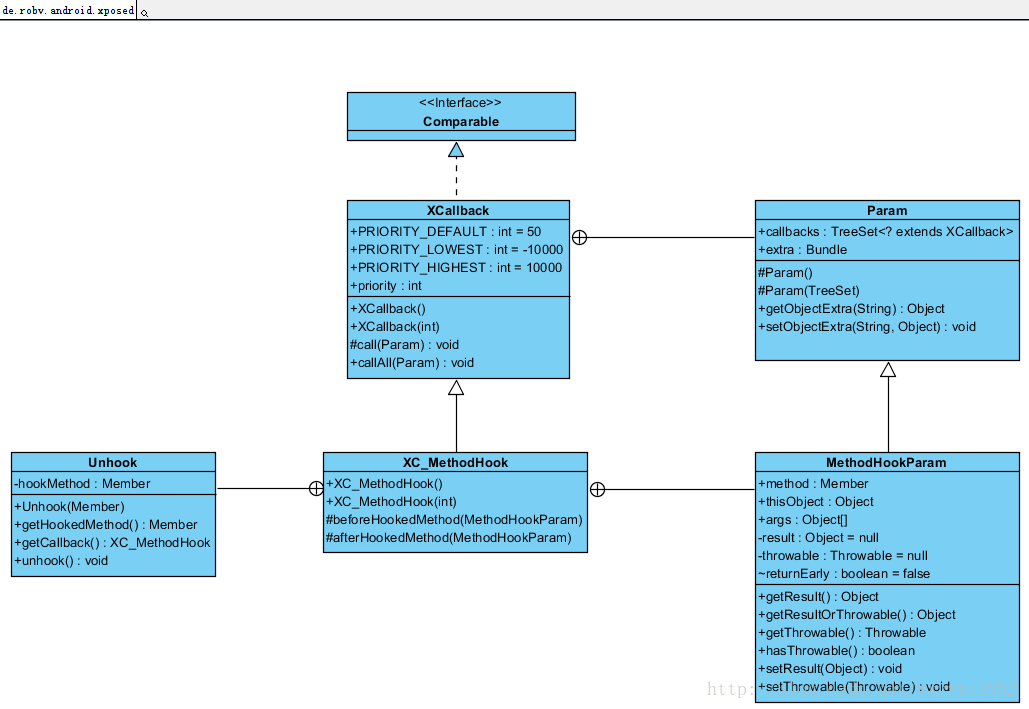
**2.2.2 初始化log文件**

XposedBridge会在XposedInstaller的目录下生成log文件，该log文件的路径为：/data/data/de.robv.android.xposed.installer/log/debug.log。log文件的初始化代码如下：   
 // initialize the Xposed framework and modules   
try {   
// initialize log file   
try {   
File logFile = new File(BASE\_DIR + "log/debug.log");   
if (startClassName == null && logFile.length() > MAX\_LOGFILE\_SIZE)   
logFile.renameTo(new File(BASE\_DIR + "log/debug.log.old"));   
logWriter = new PrintWriter(new FileWriter(logFile, true));   
logFile.setReadable(true, false);   
logFile.setWritable(true, false);   
} catch (IOException ignored) {}   
String date = DateFormat.getDateTimeInstance().format(new Date());   
determineXposedVersion();   
log("-----------------\n" + date + " UTC\n"   
+ "Loading Xposed v" + XPOSED\_BRIDGE\_VERSION   
+ " (for " + (startClassName == null ? "Zygote" : startClassName) + ")...");   
if (initNative()) {   
if (startClassName == null) {   
// Initializations for Zygote   
initXbridgeZygote();   
}   
loadModules(startClassName);   
} else {   
log("Errors during native Xposed initialization");   
}   
} catch (Throwable t) {   
log("Errors during Xposed initialization");   
log(t);   
disableHooks = true;   
}   
若startClassName==null并且log文件的长度超过阈值，会将debug.log重命名为debug.log.old。调用determineXposedVersion()获取XposedBridge的版本信息。版本信息存储在XposedBridge项目的assets/VERSION中。由于XposedBridge在Android设备上以Jar包的形式存在于XposedInstaller目录下，因此determineXposedVersion以读取zip文件的形式获取VERSION中的数据，并解析出其中的版本号，赋值给静态成员变量XPOSED\_BRIDGE\_VERSION。   
ZipInputStream is = new ZipInputStream(new FileInputStream(BASE\_DIR + "bin/XposedBridge.jar"));   
ZipEntry entry;   
try {   
while ((entry = is.getNextEntry()) != null) {   
if (!entry.getName().equals("assets/VERSION"))   
continue;   
BufferedReader br = new BufferedReader(new InputStreamReader(is));   
String version = br.readLine();   
br.close();   
XPOSED\_BRIDGE\_VERSION = extractIntPart(version);   
if (XPOSED\_BRIDGE\_VERSION == 0)   
throw new RuntimeException("could not parse XposedBridge version from \"" + version + "\"");   
return;   
}   
throw new RuntimeException("could not find assets/VERSION in " + BASE\_DIR + "bin/XposedBridge.jar");   
} finally {   
try {   
is.close();   
} catch (Exception e) { }   
}   
}

**2.2.3 获取对Java层函数的引用**

Xposed在进入XposedBridge.main函数之前，注册了4个JNI方法，其中一个是initNative()，这个函数负责获取XposedBridge中Java函数的引用。在完成log文件的初始化后，XposedBridge.main调用initNative函数。   
if (initNative()) {   
if (startClassName == null) {   
// Initializations for Zygote   
initXbridgeZygote();   
}   
loadModules(startClassName);   
} else {   
log("Errors during native Xposed initialization");   
}   
现在回到xposed.cpp中，看下initNative这个JNI方法的实现。   
static jboolean de\_robv\_android\_xposed\_XposedBridge\_initNative(JNIEnv\* env, jclass clazz) {   
...   
    xposedHandleHookedMethod = (Method\*) env->GetStaticMethodID(xposedClass, "handleHookedMethod",   
        "(Ljava/lang/reflect/Member;ILjava/lang/Object;Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;");   
...   
    xresourcesClass = env->FindClass(XRESOURCES\_CLASS);   
    xresourcesClass = reinterpret\_cast<jclass>(env->NewGlobalRef(xresourcesClass));   
...   
    if (register\_android\_content\_res\_XResources(env) != JNI\_OK) {   
        ALOGE("Could not register natives for '%s'\n", XRESOURCES\_CLASS);   
        return false;   
    }   
    xresourcesTranslateResId = env->GetStaticMethodID(xresourcesClass, "translateResId",   
        "(ILandroid/content/res/XResources;Landroid/content/res/Resources;)I");   
...   
    xresourcesTranslateAttrId = env->GetStaticMethodID(xresourcesClass, "translateAttrId",   
        "(Ljava/lang/String;Landroid/content/res/XResources;)I");   
    ...   
    return true;   
}   
该函数主要完成对XposedBridge类中函数的引用，这样可以实现在Native层对Java层函数的调用。譬如获取XposedBridge类中的handlHookedMethod函数的method id，同时赋值给全局变量xposedHandleHookedMethod。另外，initNative函数还会获取android.content.res.XResources类中的方法，完成对资源文件的处理；调用register\_android\_content\_res\_XResources注册rewriteXmlReferencesNative这个JNI方法。

**2.2.4 Hook：Java层获取hooked method与hook func**

在完成对Java层函数的引用赋值后，如果是启动Zygote，会接着执行对某些函数的hook处理。个人认为这部分是Xposed框架实现对函数hook的核心。代码如下：   
if (startClassName == null) {   
// Initializations for Zygote   
initXbridgeZygote();   
}   
initXbridgeZygote完成对一些函数的hook操作，主要是调用XposedHelpers类中的findAndHookMethod完成。   
private static void initXbridgeZygote() throws Exception {   
final HashSet<String> loadedPackagesInProcess = new HashSet<String>(1);   
// normal process initialization (for new Activity, Service, BroadcastReceiver etc.)    
findAndHookMethod(ActivityThread.class, "handleBindApplication", "android.app.ActivityThread.AppBindData", new XC\_MethodHook() {   
protected void beforeHookedMethod(MethodHookParam param) throws Throwable {   
...   
}   
}   
}   
以hook ActivityThread类的handleBindApplication函数为例来分析整个hook的过程。ActivityThread类定义在frameworks/base/core/java/android/app/ActivityThread.java文件中。ActivityThread的main函数是应用程序启动的入口。findAndHookMethod(Class<?> clazz, String methodName, Object... parameterTypesAndCallback)(另一个重载的findAndHookMethod最终也会调用前述findAndHookMethod函数)代码如下：   
public static XC\_MethodHook.Unhook findAndHookMethod(Class<?> clazz, String methodName, Object... parameterTypesAndCallback) {   
if (parameterTypesAndCallback.length == 0 || !(parameterTypesAndCallback[parameterTypesAndCallback.length-1] instanceof XC\_MethodHook))   
throw new IllegalArgumentException("no callback defined");   
XC\_MethodHook callback = (XC\_MethodHook) parameterTypesAndCallback[parameterTypesAndCallback.length-1];   
Method m = findMethodExact(clazz, methodName, parameterTypesAndCallback);   
return XposedBridge.hookMethod(m, callback);   
}   
findAndHookMethod函数参数的意义分别为：   
1. clazz: 需要hook的函数所在的类；   
2. methodName: 需要hook的函数名；   
3. parameterTypesAndCallback: 不定参数，包括methodName所指函数的参数，以及回调函数，主要是在执行methodName函数之前和之后调用的回调函数。   
其中   
XC\_MethodHook callback = (XC\_MethodHook) parameterTypesAndCallback[parameterTypesAndCallback.length-1];   
这段代码是用来parameterTypesAndCallback值为[“android.app.ActivityThread.AppBindData”, XC\_MethodHook实例]，因此callback值为其中的XC\_MethodHook实例。XC\_MethodHook类关系图如下：  
  
  
XC\_MethodHook类中的beforeHookedMethod函数会在被hook的函数调用之前调用，而afterHookedMethod函数会在被hook的函数调用之后调用。这两个函数的方法体为空，需要在实例化XC\_MethodHook时根据情况填充方法体。XC\_MethodHook的内部类MethodHookParam保存了相应的信息，如调用方法的参数，this对象，函数的返回值等。   
    
Method m = findMethodExact(clazz, methodName, parameterTypesAndCallback);   
本行代码根据需要hook的函数的类信息、函数名以及参数信息获取对应的Method实例，同时将其设置为可访问。第5行调用的findMethodExact函数的签名为Method  findMethodExact(Class<?> clazz, String methodName, Object... parameterTypes)，该函数最终会调用Method findMethodExact(Class<?> clazz, String methodName, Class<?>.parameterTypes)。在findMethodExact(Class<?> clazz, String methodName, Class<?>... parameterTypes)函数中，首先将类名、方法名以及参数信息构建成一个键值，以该键值从methodCache中查找是否存在Method实例，methodCache相当于缓存了对应的Method实例。如果没有找到，会调用Class类的getDeclaredMethod(String name,Class<?>... parameterTypes)方法获取Method实例，同时将该Method设置为可访问，加入到methodCache中。   
接下来调用XposedBridge类的静态方法hookMethod实现对函数的hook和回调函数的注册，代码如下：   
/\*\*   
 \* Hook any method with the specified callback   
 \*    
 \* @param hookMethod The method to be hooked   
 \* @param callback    
 \*/   
public static XC\_MethodHook.Unhook hookMethod(Member hookMethod, XC\_MethodHook callback) {   
if (!(hookMethod instanceof Method) && !(hookMethod instanceof Constructor<?>)) {   
throw new IllegalArgumentException("only methods and constructors can be hooked");   
}   
boolean newMethod = false;   
CopyOnWriteSortedSet<XC\_MethodHook> callbacks;   
synchronized (hookedMethodCallbacks) {   
callbacks = hookedMethodCallbacks.get(hookMethod);   
if (callbacks == null) {   
callbacks = new CopyOnWriteSortedSet<XC\_MethodHook>();   
hookedMethodCallbacks.put(hookMethod, callbacks);   
newMethod = true;   
}   
}   
callbacks.add(callback);   
if (newMethod) {   
Class<?> declaringClass = hookMethod.getDeclaringClass();   
int slot = (int) getIntField(hookMethod, "slot");   
Class<?>[] parameterTypes;   
Class<?> returnType;   
if (hookMethod instanceof Method) {   
parameterTypes = ((Method) hookMethod).getParameterTypes();   
returnType = ((Method) hookMethod).getReturnType();   
} else {   
parameterTypes = ((Constructor<?>) hookMethod).getParameterTypes();   
returnType = null;   
}   
           AdditionalHookInfo additionalInfo = new AdditionalHookInfo(callbacks, parameterTypes, returnType);   
hookMethodNative(hookMethod, declaringClass, slot, additionalInfo);   
}   
return callback.new Unhook(hookMethod);   
}   
HookedMethodCallbacks是一个hashMap的实例，存储每个需要hook的method的回调函数。首先查看hookedMethodCallbacks中是否有hookMethod对应的callbacks的集合，如果没有，则创建一个TreeSet，将该callbacks加入到hookedMethodCallbacks中，同时将newMethod标志设为true。接下来将传入的callback添加到callbacks集合中。如果是个新的需要hook的Method，则获取对应的Class对象，得到hookMethod的slot值，然后调用hookMethodNative这个JNI方法。最后实例化Unhook类，便于在后期需要对hook的方法进行unhook的操作。

**2.2.5 Hook：Native层hookMethodNative预处理Hook**

进入hookMethodNative这个JNI方法（xposed.cpp中），看下这个方法具体做了哪些操作。   
static void de\_robv\_android\_xposed\_XposedBridge\_hookMethodNative(JNIEnv\* env, jclass clazz,    
            jobject  reflectedMethodIndirect,   
            jobject declaredClassIndirect, jint slot, jobject additionalInfoIndirect) {   
    if (declaredClassIndirect == NULL || reflectedMethodIndirect == NULL) {   
        dvmThrowIllegalArgumentException("method and declaredClass must not be null");   
        return;   
    }   
    // Find the internal representation of the method   
    ClassObject\* declaredClass = (ClassObject\*) dvmDecodeIndirectRef(dvmThreadSelf(), declaredClassIndirect);   
    Method\* method = dvmSlotToMethod(declaredClass, slot);   
    if (method == NULL) {   
        dvmThrowNoSuchMethodError("could not get internal representation for method");   
        return;   
    }   
    if (xposedIsHooked(method)) {   
        // already hooked   
        return;   
    }   
    // Save a copy of the original method and other hook info   
    XposedHookInfo\* hookInfo = (XposedHookInfo\*) calloc(1, sizeof(XposedHookInfo));   
    memcpy(hookInfo, method, sizeof(hookInfo->originalMethodStruct));   
    hookInfo->reflectedMethod = dvmDecodeIndirectRef(dvmThreadSelf(), env->NewGlobalRef(reflectedMethodIndirect));   
    hookInfo->additionalInfo = dvmDecodeIndirectRef(dvmThreadSelf(), env->NewGlobalRef(additionalInfoIndirect));   
    // Replace method with our own code   
    SET\_METHOD\_FLAG(method, ACC\_NATIVE);   
    method->nativeFunc = &xposedCallHandler;   
    method->insns = (const u2\*) hookInfo;   
    method->registersSize = method->insSize;   
    method->outsSize = 0;   
    if (PTR\_gDvmJit != NULL) {   
        // reset JIT cache   
        MEMBER\_VAL(PTR\_gDvmJit, DvmJitGlobals, codeCacheFull) = true;   
    }   
　　}   
代码中首先获得Dalvik中对应的ClassObject以及Method，接下来判断需要hook的Method是否已经被hook处理过，若处理过，则直接返回。所有被hook的Method都会保存在xposedOriginalMethods这个list中。对于新的需要hook的函数，首先将其添加到xposedOriginalMethods的列表中。   
    SET\_METHOD\_FLAG(method, ACC\_NATIVE);   
    method->nativeFunc = &xposedCallHandler;   
    method->insns = (const u2\*) hookInfo;   
    method->registersSize = method->insSize;   
    method->outsSize = 0;   
如上几行代码是Xposed框架实现hook的关键。Dalvik中Method结构体定义在AOSP中/dalvik/vm/oo/Object.h中。首先将ACC\_NATIVE添加到Method的accessFlags标志位中,接下来将Method的nativeFunc设置为xposedCallHandler函数地址，然后将Method的registerSize设置为insSize，最后将outsSize设置为0。可参考Dalvik虚拟机的运行过程分析一文。Dalvik虚拟机在解释执行函数时，会调用dvmIsNativeMethod(const Method\* method)(定义于/dalvik/vm/oo/Object.h)判断Method是否为Native方法，若是，则直接调用Method->nativeFunc指向的函数。那dvmIsNativeMethod又是如何判断一个Method是否为Native方法呢？代码如下：   
INLINE bool dvmIsNativeMethod(const Method\* method)   
{   
return (method->accessFlags & ACC\_NATIVE) != 0;   
}   
正是通过比较Method的accessFlags与ACC\_NATIVE来判断的，这也就是为什么14行调用SET\_METHOD\_FLAG将ACC\_NATIVE添加到Method的accessFlags中。SET\_METHOD\_FLAG代码如下：   
#define SET\_METHOD\_FLAG(method,flag) \   
do{(method)\_>accessFlags |= (flag);} while(0)

**2.2.6 Native Hook：Native Function的dispatcher xposedCallHandler**

进入xposed.cpp中的xposedCallHandler函数，该函数会作为被hook的函数的Native方法调用，代码如下：   
static void xposedCallHandler(const u4\* args, JValue\* pResult, const Method\* method, ::Thread\* self) {   
    if (!xposedIsHooked(method)) {   
        dvmThrowNoSuchMethodError("could not find Xposed original method - how did you even get here?");   
        return;   
    }   
    XposedHookInfo\* hookInfo = (XposedHookInfo\*) method->insns;   
    Method\* original = (Method\*) hookInfo;   
    Object\* originalReflected = hookInfo->reflectedMethod;   
    Object\* additionalInfo = hookInfo->additionalInfo;   
     
    // convert/box arguments   
    const char\* desc = &method->shorty[1]; // [0] is the return type.   
    Object\* thisObject = NULL;   
    size\_t srcIndex = 0;   
    size\_t dstIndex = 0;   
       
    // for non-static methods determine the "this" pointer   
    if (!dvmIsStaticMethod(original)) {   
        thisObject = (Object\*) args[0];   
        srcIndex++;   
    }   
       
    ArrayObject\* argsArray = dvmAllocArrayByClass(objectArrayClass, strlen(method->shorty) - 1, ALLOC\_DEFAULT);   
    if (argsArray == NULL) {   
        return;   
    }   
       
    while (\*desc != '\0') {   
        char descChar = \*(desc++);   
        JValue value;   
        Object\* obj;   
        switch (descChar) {   
        case 'Z':   
        case 'C':   
        case 'F':   
        case 'B':   
        case 'S':   
        case 'I':   
            value.i = args[srcIndex++];   
            obj = (Object\*) dvmBoxPrimitive(value, dvmFindPrimitiveClass(descChar));   
            dvmReleaseTrackedAlloc(obj, self);   
            break;   
        case 'D':   
        case 'J':   
            value.j = dvmGetArgLong(args, srcIndex);   
            srcIndex += 2;   
            obj = (Object\*) dvmBoxPrimitive(value, dvmFindPrimitiveClass(descChar));   
            dvmReleaseTrackedAlloc(obj, self);   
            break;   
        case '[':   
        case 'L':   
            obj  = (Object\*) args[srcIndex++];   
            break;   
        default:   
            ALOGE("Unknown method signature description character: %c\n", descChar);   
            obj = NULL;   
            srcIndex++;   
        }   
        xposedSetObjectArrayElement(argsArray, dstIndex++, obj);   
    }   
       
    // call the Java handler function   
    JValue result;   
    dvmCallMethod(self, xposedHandleHookedMethod, NULL, &result,   
        originalReflected, (int) original, additionalInfo, thisObject, argsArray);   
           
    dvmReleaseTrackedAlloc(argsArray, self);   
    // exceptions are thrown to the caller   
    if (dvmCheckException(self)) {   
        return;   
    }   
    // return result with proper type   
    ClassObject\* returnType = dvmGetBoxedReturnType(method);   
    if (returnType->primitiveType == PRIM\_VOID) {   
        // ignored   
    } else if (result.l == NULL) {   
        if (dvmIsPrimitiveClass(returnType)) {   
            dvmThrowNullPointerException("null result when primitive expected");   
        }   
        pResult->l = NULL;   
    } else {   
        if (!dvmUnboxPrimitive(result.l, returnType, pResult)) {   
            dvmThrowClassCastException(result.l->clazz, returnType);   
        }   
    }   
}   
第4-7获得被hook函数的java.lang.reflect.Method对象实例。第9-52行完成Java本地类型到Java类型的转换，也就是将被hook函数的参数转换为Java类型，为后续在C++层调用Java层代码做准备。55行调用XposedBridge类中handleHookedMethod函数，参数分别为被hook的原始函数Method实例，this对象以及参数信息。这也就是为什么有9-52行的参数转换操作。58-72行完成对返回值的进一步处理，主要是将Java层的返回值类型转换为C++层的类型。74行将线程状态设置回调用handleHookedMethod之前的状态继续运行。

**2.2.7 XposedBridge：handleHookedMethod**

handleHookedMethod将被hook的代码又交还给java层实现。   
private static Object handleHookedMethod(Member method, Object thisObject, Object[] args) throws Throwable {   
if (disableHooks) {   
try {   
return invokeOriginalMethod(method, thisObject, args);   
} catch (InvocationTargetException e) {   
throw e.getCause();   
}   
}   
首先判断hook是否被禁用，若是，则直接调用invokeOriginalMethod函数，完成对原始函数的执行。关于如何执行原始函数的，可以继续跟踪下去分析。   
TreeSet<XC\_MethodHook> callbacks;   
synchronized (hookedMethodCallbacks) {   
callbacks = hookedMethodCallbacks.get(method);   
}   
if (callbacks == null || callbacks.isEmpty()) {   
try {   
return invokeOriginalMethod(method, thisObject, args);   
} catch (InvocationTargetException e) {   
throw e.getCause();   
}   
}   
synchronized (callbacks) {   
callbacks = ((TreeSet<XC\_MethodHook>) callbacks.clone());   
}   
根据method值，从hookedMethodCallbacks中获取对应的callback信息。hookedMethodCallbacks的分析可以参考之前对hookMethod的分析。callbacks中存储了所有对该method进行hook的beforeHookedMethod和afterHookedMethod。接着从callbacks中获取beforeHookedMethod和afterHookedMethod的迭代器。   
Iterator<XC\_MethodHook> before = callbacks.iterator();   
Iterator<XC\_MethodHook> after  = callbacks.descendingIterator();   
// call "before method" callbacks   
while (before.hasNext()) {   
try {   
before.next().beforeHookedMethod(param);   
} catch (Throwable t) {   
XposedBridge.log(t);   
// reset result (ignoring what the unexpectedly exiting callback did)   
param.setResult(null);   
param.returnEarly = false;   
continue;   
}   
if (param.returnEarly) {   
// skip remaining "before" callbacks and corresponding "after" callbacks   
while (before.hasNext() && after.hasNext()) {   
before.next();   
after.next();   
}   
break;   
}   
}   
// call original method if not requested otherwise   
if (!param.returnEarly) {   
try {   
param.setResult(invokeOriginalMethod(method, param.thisObject, param.args));   
} catch (InvocationTargetException e) {   
param.setThrowable(e.getCause());   
}   
}   
// call "after method" callbacks   
while (after.hasNext()) {   
Object lastResult =  param.getResult();   
Throwable lastThrowable = param.getThrowable();   
try {   
after.next().afterHookedMethod(param);   
} catch (Throwable t) {   
XposedBridge.log(t);   
// reset to last result (ignoring what the unexpectedly exiting callback did)   
if (lastThrowable == null)   
param.setResult(lastResult);   
else   
param.setThrowable(lastThrowable);   
}   
}   
// return   
if (param.hasThrowable())   
throw param.getThrowable();   
else   
return param.getResult();   
通过以上的分析，基本能够弄清楚Xposed框架实现hook的原理。Xposed将需要hook的函数替换成Native方法xposedCallHandler，这样Dalvik在执行被hook的函数时，就会直接调用xposedCallHandler，xposedCallHandler再调用XposedBridge类的handleHookedMethod完成注册的beforeHookedMethod以及afterHookedMethod的调用，这两类回调函数之间，会调用原始函数，完成正常的功能。   
 

**2.2.8 加载基于Xposed模块**

继续回到XposedBridge的main函数中，在处理完对hook函数的处理后会调用loadModules(String startClassName)加载基于Xposed框架的模块。   
if (initNative()) {   
if (startClassName == null) {   
// Initializations for Zygote   
initXbridgeZygote();   
}   
loadModules(startClassName);   
} else {   
log("Errors during native Xposed initialization");   
}   
loadModules读取/data/data/de.robv.android.xposed.installer/conf/modules.list文件，获得Android设备上安装的模块的APK具体路径，若设备上安装了XPrivacy，则modules.list文件内容为：/data/app/biz.bokhorst.xprivacy-1.apk。loadModules对每个模块调用loadMoudle，loadModule会根据提供的APK路径，实例化类，并根据实例的类型，进行一些初始化工作，主要的类型包括IXposedHookZygoteInit，IXposedHookLoadPackage，IXposedHookInitPackageResources，和IXposedHookCmdInit。以XPrivacy模块为例，XPrivacy类实现了IXposedHookLoadPackage和IXposedHookZygoteInit接口。如果是IXposedHookZygoteInit，loadModule会调用initZygote(StartupParam startupParam)函数。因此在分析基于Xposed框架的模块时，需要注意这点。   
private static void loadModules(String startClassName) throws IOException {   
BufferedReader apks = new BufferedReader(new FileReader(BASE\_DIR + "conf/modules.list"));   
String apk;   
while ((apk = apks.readLine()) != null) {   
loadModule(apk, startClassName);   
}   
apks.close();   
}

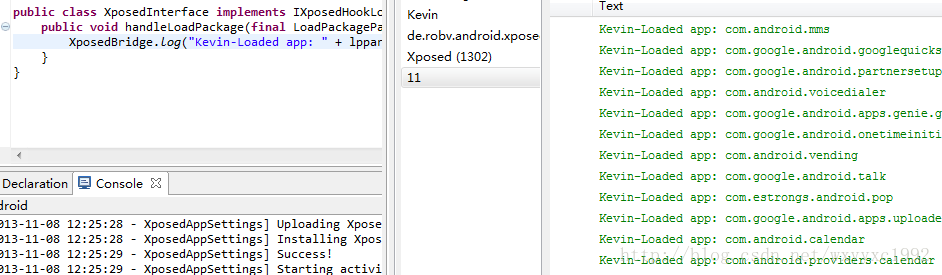
**2.2.9 调用ZygoteInit.main或RuntimeInit.main**

在Xposed的app\_main.cpp中，runtime.start调用了XposedBridge的main函数，对于Zygote启动过程来说，还必须完成对ZygoteInit.main函数的调用，完成类、资源的预加载以及对应用程序运行请求的处理。所以在XposedBridge完成自身的初始化之后，还需要完成对ZygoteInit.main的调用，如下代码所示。   
// call the original startup code   
if (startClassName == null)   
ZygoteInit.main(args);   
else   
RuntimeInit.main(args);   
 

**3 Developer Wiki**

**3.1 创建一个Xposed Module**

一个XposedModule本质上是设定了部分特殊元数据标志位的普通应用程序，需要在AndroidManifest.xml文件中添加如下设置：   
 AndroidManifest.xml => Application => Application Nodes (at the bottom) => Add => Meta Data   
添加节点：name = xposedmodule，value = true。name = xposedminiversion, value = API level。

<?xml version="1.0" encoding="utf-8"?>   
<manifest xmlns:android="http://schemas.android.com/apk/res/android"   
    package="de.robv.android.xposed.mods.tutorial"   
    android:versionCode="1"   
    android:versionName="1.0" >   
    <uses-sdk android:minSdkVersion="15" />   
    <application   
        android:icon="@drawable/ic\_launcher"   
        android:label="@string/app\_name" >   
        <meta-data android:value="true" android:name="xposedmodule"/>   
        <meta-data android:value="2.0\*" android:name="xposedminversion"/>   
        <meta-data android:value="Demonstration of the Xposed framework.\nMakes the status bar clock red." android:name="xposeddescription"/>   
    </application>   
</manifest>   
然后，将XposedBridge.jar这个引用导入到工程中，加入到reference path中。   
下面开始创建一个新的工程：   
package com.kevin.myxposed;   
import android.util.Log;   
import de.robv.android.xposed.IXposedHookLoadPackage;   
import de.robv.android.xposed.XposedBridge;   
import de.robv.android.xposed.callbacks.XC\_LoadPackage.LoadPackageParam;   
public class XposedInterface implements IXposedHookLoadPackage {   
public void handleLoadPackage(final LoadPackageParam lpparam) throws Throwable {   
        XposedBridge.log("Kevin-Loaded app: " + lpparam.packageName);   
    }   
}   
然后在assets目录下新建一个xposed\_init文件，这个文件声明了需要加载到XposedInstaller的入口类：   
com.kevin.myxposed.XposedInterface   
运行程序并在XposedInstaller的Module选项中激活，重启机器后可以得到如下数据：   
 

**3.2 HookedMethod：定位你要hook的方法**

在上一步中我们已经定位了需要Hook的方法以及所在的类，譬如：   
com.android.systemui.statusbar.policy.Clock 类   
中的updateClock方法。   
package de.robv.android.xposed.mods.tutorial;   
import static de.robv.android.xposed.XposedHelpers.findAndHookMethod;   
import de.robv.android.xposed.IXposedHookLoadPackage;   
import de.robv.android.xposed.XC\_MethodHook;   
import de.robv.android.xposed.callbacks.XC\_LoadPackage.LoadPackageParam;   
public class Tutorial implements IXposedHookLoadPackage {   
    public void handleLoadPackage(final LoadPackageParam lpparam) throws Throwable {   
     if (!lpparam.packageName.equals("com.android.systemui"))   
            return;   
       
     findAndHookMethod("com.android.systemui.statusbar.policy.Clock", lpparam.classLoader, "handleUpdateClock", new XC\_MethodHook() {   
     @Override   
     protected void beforeHookedMethod(MethodHookParam param) throws Throwable {   
     // this will be called before the clock was updated by the original method   
     }   
     @Override   
     protected void afterHookedMethod(MethodHookParam param) throws Throwable {   
     // this will be called after the clock was updated by the original method   
     }   
});   
    }   
}   
关于findAndHookMethod方法的说明见下面的API Reference。

**3.3 进行资源替换**

**3.3.1 简易资源替换**

下面所使用的方法可以适用于Boolean、Color、Integer、int[]、String与String[]。   
其中，对于Android框架层的资源（所有的APP都需要调用的资源）应该在initZygote这个方法中完成替换。而对于属于应用程序的资源，应该在hookInitPackageResources这个方法中完成替换。   
@Override   
public void initZygote(IXposedHookZygoteInit.StartupParam startupParam) throws Throwable {   
XResources.setSystemWideReplacement("android", "bool", "config\_unplugTurnsOnScreen", false);   
}   
@Override   
public void handleInitPackageResources(InitPackageResourcesParam resparam) throws Throwable {   
// replacements only for SystemUI   
if (!resparam.packageName.equals("com.android.systemui"))   
return;   
// different ways to specify the resources to be replaced   
resparam.res.setReplacement(0x7f080083, "YEAH!"); // WLAN toggle text. You should not do this because the id is not fixed. Only for framework resources, you could use android.R.string.something   
resparam.res.setReplacement("com.android.systemui:string/quickpanel\_bluetooth\_text", "WOO!");   
resparam.res.setReplacement("com.android.systemui", "string", "quickpanel\_gps\_text", "HOO!");   
resparam.res.setReplacement("com.android.systemui", "integer", "config\_maxLevelOfSignalStrengthIndicator", 6);   
　　resparam.res.setReplacement("com.android.systemui",    
　　                    "drawable", "status\_bar\_background",    
　　    new XResources.DrawableLoader() {   
　　 @Override   
　　 public Drawable newDrawable(XResources res, int id) throws Throwable {   
　　 return new ColorDrawable(Color.WHITE);   
　　 }   
　　});   
}

**3.3.2 复杂资源**

package de.robv.android.xposed.mods.coloredcirclebattery;   
import android.content.res.XModuleResources;   
import de.robv.android.xposed.IXposedHookInitPackageResources;   
import de.robv.android.xposed.IXposedHookZygoteInit;   
import de.robv.android.xposed.callbacks.XC\_InitPackageResources.InitPackageResourcesParam;   
public class ColoredCircleBattery implements IXposedHookZygoteInit, IXposedHookInitPackageResources {   
private static String MODULE\_PATH = null;   
@Override   
public void initZygote(StartupParam startupParam) throws Throwable {   
MODULE\_PATH = startupParam.modulePath;   
}   
@Override   
public void handleInitPackageResources(InitPackageResourcesParam resparam) throws Throwable {   
if (!resparam.packageName.equals("com.android.systemui"))   
return;   
XModuleResources modRes = XModuleResources.createInstance(MODULE\_PATH, resparam.res);   
resparam.res.setReplacement("com.android.systemui", "drawable", "stat\_sys\_battery", modRes.fwd(R.drawable.battery\_icon));   
resparam.res.setReplacement("com.android.systemui", "drawable", "stat\_sys\_battery\_charge", modRes.fwd(R.drawable.battery\_icon\_charge));   
}   
}

**3.3.3 修改layouts**

@Override   
public void handleInitPackageResources(InitPackageResourcesParam resparam) throws Throwable {   
if (!resparam.packageName.equals("com.android.systemui"))   
return;   
resparam.res.hookLayout("com.android.systemui", "layout", "status\_bar", new XC\_LayoutInflated() {   
@Override   
public void handleLayoutInflated(LayoutInflatedParam liparam) throws Throwable {   
TextView clock = (TextView) liparam.view.findViewById(   
liparam.res.getIdentifier("clock", "id", "com.android.systemui"));   
clock.setTextColor(Color.RED);   
}   
});    
}

**4 API Reference**

**关键类/API说明**

**IXposedHookLoadPackage**

|  |  |
| --- | --- |
| Method | Description |
| handleLoadPackage | 这个方法用于在加载应用程序的包的时候执行用户的操作。  l 调用示例  public class XposedInterface implements IXposedHookLoadPackage {  public void handleLoadPackage(final LoadPackageParam lpparam) throws Throwable {          XposedBridge.log("Kevin-Loaded app: " + lpparam.packageName);      }  }  l 参数说明   final LoadPackageParam lpparam这个参数包含了加载的应用程序的一些基本信息。 |
|  |  |

**XposedHelpers**

|  |  |
| --- | --- |
| Method | Description |
| findAndHookMethod | 这是一个辅助方法，可以通过如下方式静态导入：  import static de.robv.android.xposed.XposedHelpers.findAndHookMethod;  l 使用示例  findAndHookMethod("com.android.systemui.statusbar.policy.Clock", lpparam.classLoader, "handleUpdateClock", new XC\_MethodHook() {       @Override       protected void beforeHookedMethod(MethodHookParam param) throws Throwable {       // this will be called before the clock was updated by the original method       }       @Override       protected void afterHookedMethod(MethodHookParam param) throws Throwable {       // this will be called after the clock was updated by the original method       }  });  l 参数说明  v findAndHookMethod(Class<?> clazz,   //需要Hook的类名        ClassLoader，   //类加载器，可以设置为null        String methodName,    //需要Hook的方法名        Object... parameterTypesAndCallback  )  该函数的最后一个参数集，包含了：  （1）Hook的目标方法的参数,譬如：       "com.android.internal.policy.impl.PhoneWindow.DecorView"       是方法的参数的类。  （2）回调方法：       a.XC\_MethodHook        b.XC\_MethodReplacement |
|  | l |

**辅助项API**

Xposed框架也为我们提供了很多的辅助项来帮助我们快速开发XposedModule。

**XposedBridge 类**

|  |  |
| --- | --- |
| Method | Description |
| log | 该方法可以将log信息以及Throwable抛出的异常信息输出到标准的logcat以及/data/xposed/debug.log这个文件中。 |
| hookAllMethods / hookAllConstructors | 该方法可以用来hook某个类中的所有方法或者构造函数，但是不同的Rom（非Android原生Rom）会有不同的变种。 |

**XposedHelpers 类**

这个类用的也是比较多，可以使用   
Window => Preferences => Java => Editor => Content Assist => Favorites => New Type, enter de.robv.android.xposed.XposedHelpers   
这种方式将XposedHelpers这个类加入到Eclipse静态调用中方便查阅。

|  |  |
| --- | --- |
| Method | Description |
| findMethod / findConstructor / findField | 这是一组用于检索方法的方法。 |
| callMethod / callStaticMethod / newInstance |  |
| assetAsByteArray | 以字节数组的形式返回asset，可以以如下方式调用：  public class XposedTweakbox {  private static final String MODULE\_PATH = null;       // injected by XposedBridge  public static void init(String startClassName) throws Exception {  if (startClassName != null)  return;  Resources tweakboxRes =   XModuleResources.createInstance(MODULE\_PATH, null);  byte[] crtPatch = assetAsByteArray(tweakboxRes,   "crtfix\_samsung\_d506192d5049a4042fb84c0265edfe42.bsdiff");  ... |
|  |  |
| getMD5Sum | 返回对于一个文件的MD5校验值，需要root权限。 |
| getProcessPid | 获取一个进程的PID值，输入参数为/proc/[pid]/cmdline |