第九章 Android Hook框架

在前面的章节中，我们简要介绍了如何将开发的模块内置到系统中，并将其注入到应用程序中执行。而内置并注入第三方开发的工具，与之前介绍的简单内置注入过程没有太大区别。关键步骤是加载工具所依赖的动态库，然后再加载核心业务组件。本章将以几个典型的Hook框架作为例子，展示如何将它们内置在系统中。

通过这些例子，我们可以更深入地理解和学习Hook技术，并掌握如何将其集成到系统中。

9.1 Xposed

Xposed是一个Android Hook框架，它可以在不修改APK文件的情况下改变系统和应用程序的行为。通过开发模块，我们能够对目标进程的Java函数调用进行Hook拦截。然而，要使用该框架中的模块功能，需要将其安装在Root权限的Android设备上。

根据Xposed框架原理衍生出了很多类似的框架，比如Edxposed、Lsposed等等。

在Xposed的架构中，主要包含三个部分：Xposed Installer、Xposed Bridge和Xposed Module。其中，Xposed Installer是用户安装和管理Xposed模块的应用程序；Xposed Bridge是实现系统与模块之间相互通信的核心组件；而Xpoed Module则是开发者使用Xpose API编写并且加载到目标进程中来实现对目标进程函数调用拦截和修改。

运行时, Xpose Installer会通过Android的PackageManager查询已经安装好了那些 app 并将相关信息传递给 Xpose Bridge. 当目标app启动时, Xposd bridge就会load相关module到target process中，并建立起相应管道以便后续操作.

编写一个xposed模块主要涉及两个方面:

1. 继承 IXpodeHookLoadPackage 接口来完成启动事件监听;
2. 使用 xposd API 来进行函数调用拦截和修改.

这些API包括XposedHelpers.findAndHookMethod和XposedHelpers.callMethed等，它们可以帮助我们定位到目标进程中的函数，并对其进行拦截和修改。

本章将详细解析Xposed的原理，学习Xposed如何利用Android的运行机制来实现注入。

9.2 Xposed实现原理

在开始分析Xposed源码前，首先回顾一下第三章中，讲解Android启动流程时，最后根据AOSP的源码得到的以下结论。

1. zygote进程启动是通过app_process执行程序启动的
2. 由init进程解析init.rc时启动的第一个zygote
3. 在第一个zygote进程中创建的ZygoteServer，并开始监听消息。
4. zygote是在ZygoteServer这个服务中收到消息后，再去fork出新进程的
5. 所有进程均来自于zygote进程的fork而来，所以zygote是进程的始祖

从上面的结论中可以看到，app_process执行程序在其中占据着非常重要的位置，而Xposed的核心原理，就是将app_process替换为Xposed修改过的app_process，这样就会让所有进程都会通过它的业务逻辑处理。首先找到项目https://github.com/rovo89/Xposed。查看文件Android.mk。

```
ifeq (1,$(strip $(shell expr $(PLATFORM_SDK_VERSION) \>= 21)))
  LOCAL_SRC_FILES := app_main2.cpp
  LOCAL_MULTILIB := both
  LOCAL_MODULE_STEM_32 := app_process32_xposed
  LOCAL_MODULE_STEM_64 := app_process64_xposed
else
  LOCAL_SRC_FILES := app_main.cpp
  LOCAL_MODULE_STEM := app_process_xposed
endif
```

可以看到这里是用来编译一个Xposed专用的app_process。当Android版本大于21（Android 5）时，使用app_main2.cpp来编译。接下来查看入口函数的实现。

```
#define XPOSED_CLASS_DOTS_TOOLS
"de.robv.android.xposed.XposedBridge$ToolEntryPoint"

int main(int argc, char* const argv[])
{
    ...
    // 检测Xposed的参数
    if (xposed::handleOptions(argc, argv)) {
        return 0;
    }
    ...
    if (zygote) {
        // Xposed 框架的初始化，为后续的 Hook 操作和代码注入操作提供支持。
        isXposedLoaded = xposed::initialize(true, startSystemServer, NULL, argc,
argv);
        runtimeStart(runtime, isXposedLoaded ? XPOSED_CLASS_DOTS_ZYGOTE :
"com.android.internal.os.ZygoteInit", args, zygote);
    } else if (className) {
        isXposedLoaded = xposed::initialize(false, false, className, argc, argv);
        runtimeStart(runtime, isXposedLoaded ? XPOSED_CLASS_DOTS_TOOLS :
"com.android.internal.os.RuntimeInit", args, zygote);
    } else {
        fprintf(stderr, "Error: no class name or --zygote supplied.\n");
        app_usage();
        LOG_ALWAYS_FATAL("app_process: no class name or --zygote supplied.");
        return 10;
    }
}
```

在这个特殊的app_process中，首先是对启动进程的参数进行检查，然后初始化Xposed框架，如果初始化成功了，则使用Xposed的入口de.robv.android.xposed.XposedBridge$ToolEntryPoint来替换系统原本的com.android.internal.os.ZygoteInit入口。

xposed::initialize是一个非常关键的函数，它完成了 Xposed 框架的初始化工作。查看实现代码如下。

```
bool initialize(bool zygote, bool startSystemServer, const char* className, int
argc, char* const argv[]) {
#if !defined(XPOSED_ENABLE_FOR_TOOLS)
    ...
    // 将参数保存
    xposed->zygote = zygote;
    xposed->startSystemServer = startSystemServer;
    xposed->startClassName = className;
    xposed->xposedVersionInt = xposedVersionInt;

#if XPOSED_WITH_SELINUX
    xposed->isSELinuxEnabled   = is_selinux_enabled() == 1;
    xposed->isSELinuxEnforcing = xposed->isSELinuxEnabled && security_getenforce()
== 1;
#else
    xposed->isSELinuxEnabled   = false;
    xposed->isSELinuxEnforcing = false;
#endif  // XPOSED_WITH_SELINUX

    ...

    if (startSystemServer) {
        if (!determineXposedInstallerUidGid() || !xposed::service::startAll()) {
            return false;
        }
        // 启动Xposed框架的日志记录，将xposed框架日志写入logcat中。
        xposed::logcat::start();
// SELinux启用的情况
#if XPOSED_WITH_SELINUX
    } else if (xposed->isSELinuxEnabled) {
        // 用于启动Xposed框架的 membased 服务，该服务实现hooking功能
        if (!xposed::service::startMembased()) {
            return false;
        }
#endif  // XPOSED_WITH_SELINUX
    }
// SELinux启用的情况
#if XPOSED_WITH_SELINUX
    // 限制内存继承，以确保Xposed服务只能被当前进程和其子进程使用，而不能被其他进程使用
    if (xposed->isSELinuxEnabled) {
        xposed::service::membased::restrictMemoryInheritance();
    }
#endif  // XPOSED_WITH_SELINUX

    // 是否禁用xposed
    if (zygote && !isSafemodeDisabled() &&
detectSafemodeTrigger(shouldSkipSafemodeDelay()))
        disableXposed();

    if (isDisabled() || (!zygote && shouldIgnoreCommand(argc, argv)))
        return false;
    // 将Xposed JAR文件添加到应用程序或服务的类路径中
    return addJarToClasspath();
```

```
}
```

在启用SELinux的情况下，Xposed需要使用membased服务来实现hooking功能。但是，为了确保安全性，Xposed需要限制将Xposed服务复制到其他进程中的能力。通过调用restrictMemoryInheritance函数，Xposed会防止任何进程继承Zygote进程的内存，这将确保Xposed服务只能被当前进程和其子进程使用。

初始化完成时，将XposedBridge.jar文件添加到了CLASSPATH环境变量中，查看addJarToClasspath的实现。

```
#define XPOSED_JAR               "/system/framework/XposedBridge.jar"

bool addJarToClasspath() {
    ALOGI("-----------------");
    if (access(XPOSED_JAR, R_OK) == 0) {
        if (!addPathToEnv("CLASSPATH", XPOSED_JAR))
            return false;

        ALOGI("Added Xposed (%s) to CLASSPATH", XPOSED_JAR);
        return true;
    } else {
        ALOGE("ERROR: Could not access Xposed jar '%s'", XPOSED_JAR);
        return false;
    }
}
```

初始化成功后，接着继续追踪替换后的入口点de.robv.android.xposed.XposedBridge$ToolEntryPoint，该入口点的实现是在XposedBridge.jar中。查看项目https://github.com/rovo89/XposedBridge，文件XposedBridge.java的实现代码如下。

```
package de.robv.android.xposed;

public final class XposedBridge {
    protected static final class ToolEntryPoint {
        protected static void main(String[] args) {
            isZygote = false;
            XposedBridge.main(args);
        }
    }

    protected static void main(String[] args) {
        // 初始化Xposed框架和模块
        try {
            if (!hadInitErrors()) {
                initXResources();
                SELinuxHelper.initOnce();
                SELinuxHelper.initForProcess(null);

                runtime = getRuntime();
```

```
                XPOSED_BRIDGE_VERSION = getXposedVersion();

                if (isZygote) {
                    // hook Android 资源系统
                    XposedInit.hookResources();
                    // 初始化 Xposed 框架的 zygote 进程，创建用于跨进程通信的 Binder
对象，并注册相关的 Service。这样就能够实现跨进程的 Hook 功能
                    XposedInit.initForZygote();
                }
                // 加载Xposed模块
                XposedInit.loadModules();
            } else {
                Log.e(TAG, "Not initializing Xposed because of previous errors");
            }
        } catch (Throwable t) {
            Log.e(TAG, "Errors during Xposed initialization", t);
            disableHooks = true;
        }

        // 调用原始应用的入口
        if (isZygote) {
            ZygoteInit.main(args);
        } else {
            RuntimeInit.main(args);
        }
    }
}
```

到这里，Xposed的启动流程基本完成了，Xposed首先替换原始的app_process，让每个进程启动时使用自己的app_process_xposed，在执行zygote入口函数前，先初始化了自身的环境，然后每个进程后是先进入的XposedBridge，在完成自身的逻辑后，才调用zygote的入口函数，进入应用正常启动流程。这也意味着，对于系统定制者来说，所谓的Root权限才能使用Xposed并不是必须的。最后看看loadModules的实现，是如何加载Xposed模块的。

```
private static final String INSTANT_RUN_CLASS =
"com.android.tools.fd.runtime.BootstrapApplication";

// 加载模块列表
static void loadModules() throws IOException {

    final String filename = BASE_DIR + "conf/modules.list";
    BaseService service = SELinuxHelper.getAppDataFileService();
    if (!service.checkFileExists(filename)) {
        Log.e(TAG, "Cannot load any modules because " + filename + " was not
found");
        return;
    }
    // 拿到顶端的ClassLoader
    ClassLoader topClassLoader = XposedBridge.BOOTCLASSLOADER;
    ClassLoader parent;
    while ((parent = topClassLoader.getParent()) != null) {
```

```
            topClassLoader = parent;
        }
        // 读取模块列表
        InputStream stream = service.getFileInputStream(filename);
        BufferedReader apks = new BufferedReader(new InputStreamReader(stream));
        String apk;
        // 使用顶端ClassLoader加载每个模块
        while ((apk = apks.readLine()) != null) {
            loadModule(apk, topClassLoader);
        }
        apks.close();
    }

    private static void loadModule(String apk, ClassLoader topClassLoader) {
        Log.i(TAG, "Loading modules from " + apk);

        if (!new File(apk).exists()) {
            Log.e(TAG, "  File does not exist");
            return;
        }

        DexFile dexFile;
        try {
            dexFile = new DexFile(apk);
        } catch (IOException e) {
            Log.e(TAG, "  Cannot load module", e);
            return;
        }
        // 如果加载成功，说明该应用启用了 Instant Run
        if (dexFile.loadClass(INSTANT_RUN_CLASS, topClassLoader) != null) {
            Log.e(TAG, "  Cannot load module, please disable \"Instant Run\" in
    Android Studio.");
            closeSilently(dexFile);
            return;
        }
        // 尝试在目标模块中加载XposedBridge类，可以获取到说明已经成功注入XposedBridge
        if (dexFile.loadClass(XposedBridge.class.getName(), topClassLoader) != null) {
            Log.e(TAG, "  Cannot load module:");
            Log.e(TAG, "  The Xposed API classes are compiled into the module's
    APK.");
            Log.e(TAG, "  This may cause strange issues and must be fixed by the
    module developer.");
            Log.e(TAG, "  For details, see: http://api.xposed.info/using.html");
            closeSilently(dexFile);
            return;
        }

        closeSilently(dexFile);
        // 由于模块实际都是apk，而apk本质是压缩包，所以使用Zip来处理文件
        ZipFile zipFile = null;
        InputStream is;
        try {
            zipFile = new ZipFile(apk);
            // 解压出xposed_init文件，这里存放着模块启动的入口
```

```
        ZipEntry zipEntry = zipFile.getEntry("assets/xposed_init");
        if (zipEntry == null) {
            Log.e(TAG, "  assets/xposed_init not found in the APK");
            closeSilently(zipFile);
            return;
        }
        is = zipFile.getInputStream(zipEntry);
    } catch (IOException e) {
        Log.e(TAG, "  Cannot read assets/xposed_init in the APK", e);
        closeSilently(zipFile);
        return;
    }
    // 动态加载模块
    ClassLoader mcl = new PathClassLoader(apk, XposedBridge.BOOTCLASSLOADER);
    BufferedReader moduleClassesReader = new BufferedReader(new
InputStreamReader(is));
    try {
        String moduleClassName;
        while ((moduleClassName = moduleClassesReader.readLine()) != null) {
            moduleClassName = moduleClassName.trim();
            if (moduleClassName.isEmpty() || moduleClassName.startsWith("#"))
                continue;

            try {
                // 加载模块的入口类
                Log.i(TAG, "  Loading class " + moduleClassName);
                Class<?> moduleClass = mcl.loadClass(moduleClassName);
                // 检查该类是否有实现接口
                if (!IXposedMod.class.isAssignableFrom(moduleClass)) {
                    Log.e(TAG, "    This class doesn't implement any sub-interface
of IXposedMod, skipping it");
                    continue;
                } else if (disableResources &&
IXposedHookInitPackageResources.class.isAssignableFrom(moduleClass)) {
                    Log.e(TAG, "    This class requires resource-related hooks
(which are disabled), skipping it.");
                    continue;
                }
                // 使用该类初始化一个对象
                final Object moduleInstance = moduleClass.newInstance();
                if (XposedBridge.isZygote) {
                    // 不同的实现接口有各自对应的处理，这里是Zygote模块初始化时使用的模
块
                    if (moduleInstance instanceof IXposedHookZygoteInit) {
                        IXposedHookZygoteInit.StartupParam param = new
IXposedHookZygoteInit.StartupParam();
                        param.modulePath = apk;
                        param.startsSystemServer = startsSystemServer;
                        ((IXposedHookZygoteInit)
moduleInstance).initZygote(param);
                    }
                    // 普通应用的模块接口
                    if (moduleInstance instanceof IXposedHookLoadPackage)
                        // 调用了模块中的实现。
```

```
                        XposedBridge.hookLoadPackage(new
IXposedHookLoadPackage.Wrapper((IXposedHookLoadPackage) moduleInstance));

                    if (moduleInstance instanceof IXposedHookInitPackageResources)
                        XposedBridge.hookInitPackageResources(new
IXposedHookInitPackageResources.Wrapper((IXposedHookInitPackageResources)
moduleInstance));
                } else {
                    if (moduleInstance instanceof IXposedHookCmdInit) {
                        IXposedHookCmdInit.StartupParam param = new
IXposedHookCmdInit.StartupParam();
                        param.modulePath = apk;
                        param.startClassName = startClassName;
                        ((IXposedHookCmdInit) moduleInstance).initCmdApp(param);
                    }
                }
            } catch (Throwable t) {
                Log.e(TAG, "    Failed to load class " + moduleClassName, t);
            }
        }
    } catch (IOException e) {
        Log.e(TAG, "  Failed to load module from " + apk, e);
    } finally {
        closeSilently(is);
        closeSilently(zipFile);
    }
}
```

分析完加载模块的实现后，这时就明白模块开发时定义的入口是如何被调用的，以及被调用的时机在哪里。理解其中的原理后，同样可以自己进行修改，在其他的时机来选择注入。用自己的方式来定义模块。

9.3 常见的hook框架

根据Xposed的源码分析不难看出其关键在于XposedBridge.jar的注入，然后由XposedBridge.jar实现对函数Hook的关键逻辑，因为Xposed框架提供了非常方便和灵活的API，使得开发者可以快速地编写自己的Hook模块并且可以兼容大多数Android系统版本和设备。所以很多Hook框架都会兼容支持Xposed框架。

SandHook是作用在Android ART虚拟机上的Java层Hook框架，作用于进程内是不需要Root的，支持Android 4.4 - Android 10，该框架兼容Xposed Api调用。

除了支持常规的Java层Hook外，Sandhook还支持对Native层的函数进行Hook。它通过使用系统提供的符号表来获取函数地址，并将函数地址转换为可执行代码，从而实现Native Hook。

Sandhook本身是没有注入功能的，开发完模块功能后，需要自行重打包，或者使用其他工具将模块注入。从开发AOSP的角度，可以参考前文内置JAR包的做法，直接将Sandhook内置到AOSP系统中，并实现对任意进程自动注入。

pine是一个在虚拟机层面、以Java方法为粒度的运行时动态Hook框架，它可以拦截本进程内几乎所有的java方法调用。支持Android 4.4 - Android 12。同样该框架也兼容Xposed Api调用。Pine支持两种方案，一种是替换入口，即修改ArtMethod的entrypoint；另一种类似于native的inline hook，即覆盖掉目标方法的代码开始处的一段代码，用于弥补Android 8.0以下版本入口替换很有可能不生效的问题。

Dobby是一个基于Android NDK开发的Native Hook框架。它可以在Android应用程序中注入自定义代码段，从而实现函数替换、跳转、插桩等操作。Dobby主要使用了动态链接库和指令重写技术，通过Hook目标进程中的函数来达到修改目的。

相比Java层的Hook框架，Native Hook有一些优势。首先，Native Hook可以直接操作目标进程的内存空间，更加灵活；其次，Native Hook可以通过指令重写技术来控制执行流程，效果更加精准；最后，Native Hook避免了Java层Hook可能引起的兼容性问题，适用范围更广。

9.4 集成dobby

集成方式与pine相同，首先开发一个使用dobby的样例，然后将其中的依赖动态库集成到系统中，最后在进程启动的过程中，将其加载即可。由于dobby是对native函数进行hook的，所以Android Studio创建一个native c++的项目，然后使用git将dobby项目拉取下来。项目地址：https://github.com/jmpews/Dobby。然后修改项目中cpp目录下的CMakeLists.txt文件，将dobby加入其中。修改如下。

```
cmake_minimum_required(VERSION 3.18.1)
// 设置dobby源码的目录
set(DobbyHome ~/git_src/Dobby)
enable_language(C ASM)

include_directories(
        dlfc
        utils
)

project("mydobby")

add_library(
        mydobby
        SHARED
        native-lib.cpp)

find_library(
        log-lib
        log)

target_link_libraries(
        mydobby
        dobby
        ${log-lib})

# 使用设置的路径,引入Dobby
include_directories(
        ${DobbyHome}/include
        ${DobbyHome}/source
        ${DobbyHome}/builtin-plugin
        ${DobbyHome}/builtin-plugin/AndroidRestriction
        ${DobbyHome}/builtin-plugin/SymbolResolver
        ${DobbyHome}/external/logging
)
```

```
macro(SET_OPTION option value)
    set(${option} ${value} CACHE INTERNAL "" FORCE)
endmacro()

SET_OPTION(DOBBY_DEBUG ON)
SET_OPTION(DOBBY_GENERATE_SHARED ON)
SET_OPTION(Plugin.LinkerLoadCallback OFF)

add_subdirectory(~/git_src/Dobby dobby.build)

if(${CMAKE_ANDROID_ARCH_ABI} STREQUAL "arm64-v8a")
    add_definitions(-DCORE_SO_NAME="${LIBRARY_NAME}")
elseif(${CMAKE_ANDROID_ARCH_ABI} STREQUAL "armeabi-v7a")
    add_definitions(-DCORE_SO_NAME="${LIBRARY_NAME}")
endif()
```

将dobby的源码引入后，就可以在项目中使用dobby进行hook处理了。修改native-lib.cpp文件，添加测试的hook代码，内容如下。

```
#include <jni.h>
#include <string>
#include <android/log.h>
#include "dobby.h"

#define LOG_TAG "native-lib"

#define ALOGD(...) __android_log_print(ANDROID_LOG_DEBUG  , LOG_TAG, __VA_ARGS__)

int (*source_openat)(int fd, const char *path, int oflag, int mode) = nullptr;

// 替换后的新函数
int MyOpenAt(int fd, const char *pathname, int flags, int mode) {
    ALOGD("[ROM] MyOpenAt  pathname :%s",pathname);
    if (strcmp(pathname, "/sbin/su") == 0 || strcmp(pathname, "/system/bin/su") ==
0) {
        pathname = "/system/xbin/Mysu";
    }
    // 执行原来的openat函数
    return source_openat(fd, pathname, flags, mode);
}

void HookOpenAt() {
    // 找到函数对应的地址
    void *__openat =
            DobbySymbolResolver("libc.so", "__openat");

    if (__openat == nullptr) {
        ALOGD("__openat null ");
        return;
    }
```

```
    ALOGD("拿到 __openat 地址 ");
    //dobby hook 函数
    if (DobbyHook((void *) __openat,
                  (dobby_dummy_func_t) MyOpenAt,
                  (dobby_dummy_func_t*) &source_openat) == RT_SUCCESS) {
        ALOGD("DobbyHook __openat sucess");
    }
}

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

    ALOGD("Hello JNI_OnLoad 开始加载");
    JNIEnv *env = nullptr;
    //改变openat 指定函数 函数地址 替换成自己的
    HookOpenAt();

    if (vm->GetEnv((void **) &env, JNI_VERSION_1_6) == JNI_OK) {
        return JNI_VERSION_1_6;
    }
    return 0;
}
```

样例应用准备完毕，将该样例编译并运行后，就能成功看到对openat进行hook的输出如下。

```
...
D  [ROM] MyOpenAt  pathname
:/data/vendor/gpu/esx_config_cn.rom.devchangemodule.txt
D  [ROM] MyOpenAt  pathname :/data/vendor/gpu/esx_config.txt
D  [ROM] MyOpenAt  pathname :/data/misc/gpu/esx_config_cn.rom.devchangemodule.txt
D  [ROM] MyOpenAt  pathname :/data/misc/gpu/esx_config.txt
D  [ROM] MyOpenAt  pathname
:/data/vendor/gpu/esx_config_cn.rom.devchangemodule.txt
D  [ROM] MyOpenAt  pathname :/data/vendor/gpu/esx_config.txt
D  [ROM] MyOpenAt  pathname :/data/misc/gpu/esx_config_cn.rom.devchangemodule.txt
D  [ROM] MyOpenAt  pathname :/data/misc/gpu/esx_config.txt
...
```

接下来将该样例应用编译的apk文件进行解压，在lib目录中找到依赖的动态库，分别是libdobby.so和libmydobby.so，其中前者是hook框架的核心库，后者是刚刚对openat进行hook的业务代码。只需要在任何进程启动前，按顺序将依赖的核心动态库，和业务代码加载，即可完成集成的工作，libdobby.so可以选择集成到系统中，也可以选择跟业务代码动态库一起放同一个目录进行加载。下面看实现加载的代码。

```
private static void loadSoModule(String soName){
    String soPath="";
    if(System.getProperty("os.arch").indexOf("64") >= 0) {
        soPath = String.format("/data/data/cn.rom.dobbydemo/%s", soName);
    }else{
        soPath = String.format("/data/data/cn.rom.dobbydemo/%s", soName);
```

```
        }
        File file = new File(soPath);
        if (file.exists()){
            Log.e("[ROM]", "load so "+soPath);
            System.load(tmpPath);
            Log.e("[ROM]", "load over so "+soPath);
        }else{
            Log.e("[ROM]", "load so "+soPath+" not exist");
        }
    }

    private void handleBindApplication(AppBindData data) {
        ...
        app = data.info.makeApplication(data.restrictedBackupMode, null);
        // Propagate autofill compat state
        app.setAutofillOptions(data.autofillOptions);
        // Propagate Content Capture options
        app.setContentCaptureOptions(data.contentCaptureOptions);
        sendMessage(H.SET_CONTENT_CAPTURE_OPTIONS_CALLBACK, data.appInfo.packageName);
        mInitialApplication = app;
        // 非系统进程则注入jar包
        int flags = mBoundApplication == null ? 0 : mBoundApplication.appInfo.flags;
        if(flags>0&&((flags&ApplicationInfo.FLAG_SYSTEM)!=1)){
            loadSoModule("libdobby.so");
            loadSoModule("libmydobby.so");
        }
    }
```

这只是一个简单的加载样例演示。在安装目标应用后，还需要将两个动态库拷贝到相应的目录中。在实际运用场景中，我们尽量不要将动态库路径和要加载的库名称固定写入源代码中。最好通过配置的方式来管理这些需要加载的参数。

为了确保加载动态库成功，需要确保目录具有执行权限，并且最好将文件放在当前应用程序的私有目录中。

完成修改后，无论安装任何应用程序并打开它时，都会被hook `openat`函数。

9.5 本章小结

本章主要介绍了安卓系统上一些常见的Hook框架。以及讲解了如何修改系统代码，集成Hook框架到系统中。Hook框架由于其特性，检测与反检测技术也是安全领域时常讨论的话题，本书不深入讨论，有兴趣的朋友可以网络上搜索相应的文章与工具。这种方式最大的优点是，集成的自定义的框架具有较高的权限与隐蔽性，缺点也很明显，就是框架代码的升级比较麻烦，可能需要重新编译修改ROM与替换系统文件。