Android Java Hook学习笔记

前言

在Android上面实现了java hook的框架有xposed，cydia substrate和adbi，其中开源的只有xposed和adbi，网上有许多xposed 和 Cydia substrate的java hook教程，虽然能够实现java hook，但是我们没有弄明白hook的原理，没有学习到任何东西。

在看雪论坛看到一篇java hook的文章 注入安卓进程,并hook java世界的方法，我们来以这篇文章来学习如何进行java hook

代码编译

将文中提到的测试代码下载导入到eclipse中，最开始的还是进行ndk-build进行编译，查看Android.mk的内容发现编译了几个模块，将art的编译命令删除，执行ndk-build出现error

```
F:/密码/TestAR//jni/so.cpp:2:44: fatal error: android_runtime/AndroidRuntime.h:
No such file or directory
 #include "android_runtime/AndroidRuntime.h"
                                           ^
```

中提
发现

我们可以看到找不到andriod_runtime.h，但是在代码目录中我们可以看到这个头文件，在android.mk文件中我们也包含了这个文件的路径

```
LOCAL_CFLAGS:= -I./jni/include/ -I./jni/dalvik/vm/ -I./jni/dalvik -DHAVE_LITTLE_ENDIAN
```

当时我想编译成功也是弄了个把小时，一直提示找不到头文件，如果仔细看的话android.mk是在jni目录下的，所以需要修改Android.mk文件的内容，将前面的/jni去掉

```
LOCAL_CFLAGS:= -I./include/ -I./dalvik/vm/ -I./dalvik -DHAVE_LITTLE_ENDIAN
LOCAL_LDFLAGS   :=  -L./lib/  -L$(SYSROOT)/usr/lib -llog -ldvm -landroid_runtime
```

再执行ndk-build就可以成功了 同样地，我们可以将Android.mk命令改为这样

```
LOCAL_CFLAGS:= -DHAVE_LITTLE_ENDIAN
LOCAL_C_INCLUDES +=$(LOCAL_PATH)/include/  $(LOCAL_PATH)/dalvik/vm  $(LOCAL_PATH)/dalvik
```

代码分析

这里我将so.cpp的内容修改了一下，让libso.so被注入之后自动执行 InjectInterface函数

```
extern "C" void InjectInterface(char*arg) __attribute__((constructor));
extern "C" void InjectInterface(char*arg)
{

log_("*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*");
log_("*-*-*-*-*-* Injected so *-*-*-*-*-*-*-*");
log_("*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*");
Hook();
log_("*-*-*-*-*-*-*- End -*-*-*-*-*-*-*-*-*-*");

}
```

InjectInterface调用了MethodHooker.cpp中的Hook函数

```
int Hook(){
init();
void* handle = dlopen("/data/local/tmp/libTest.so",RTLD_NOW);
const char *dlopen_error = dlerror();
if(!handle){
    ALOG("Error","cannt load plugin :%s",dlopen_error);
    return -1;
}
```

```
SetupFunc setup = (SetupFunc)dlsym(handle,"getpHookInfo");
const char *dlsym_error = dlerror();
if (dlsym_error) {
    ALOG("Error","Cannot load symbol 'getpHookInfo' :%s" , dlsym_error);
    dlclose(handle);
    return 1;
}

HookInfo *hookInfo;
setup(&hookInfo);
ALOG("LOG","Target Class:%s",hookInfo[1].tClazz);
ALOG("LOG","Target Method:%s",hookInfo[1].tMethod);

ClassMethodHook(hookInfo[1]);
}
```

init函数获取当前进程的javaVM,然后加载libTest.so，执行getHookInfo()函数， Test.c中的 getpHookInfo函数

```
HookInfo hookInfos[] = {
        {"android/net/wifi/WifiInfo","getMacAddress","()Ljava/lang/String;",(void*)test},
        {"com/example/javahook/MainActivity","test","()Ljava/lang/String;",(void*)test},
        //{"android/app/ApplicationLoaders","getText","()Ljava/lang/CharSequence;",(void*)test},
};

int getpHookInfo(HookInfo** pInfo){
    *pInfo = hookInfos;
    return sizeof(hookInfos) / sizeof(hookInfos[0]);
}
```

从代码我们可以看到 hookInfos[]数组中的Hook函数信息，包括函数所在的类，函数名称，函数类型以及FakeHook函数地址，getpHookInfo函数就是将这些信息返回给HookInfo *hookInfo指针

获取到要hook的函数信息，执行ClassMethodHook()

```
bool ClassMethodHook(HookInfo info){

    JNIEnv *jenv = GetEnv();
    //寻找getMacAddress所在的类"android/net/wifi/WifiInfo"
    jclass clazzTarget = jenv->FindClass(info.tClazz);
    if (ClearException(jenv)) {
        ALOG("Exception","ClassMethodHook[Can't find class:%s in bootclassloader",info.tClazz);
    /*
      通常不是系统自带的类FindClass(info.tClazz)是找不到的，作者自己写了一个 findAppClass函数寻找自定义类
    */
        clazzTarget = findAppClass(jenv,info.tClazz);
        if(clazzTarget == NULL){
            ALOG("Exception","%s","Error in findAppClass");
            return false;
        }
        }

        ALOG("LOG","Find calss success");
        //获取getMacAddress函数的jMethodID
        jmethodID method = jenv->GetMethodID(clazzTarget,info.tMethod,info.tMeihodSig);
        if(method==NULL){
            ALOG("Exception","ClassMethodHook[Can't find method:%s",info.tMethod);
            return false;
        }
        else
            ALOG("LOG","Find Method ID success");
    /*
    if(isArt()){
        HookArtMethod(jenv,method);
    }else{
        HookDalvikMethod(method);
    }
```

```
    这里我们只关心Dalvik Hook HookDalvikMethod也就是java hook最重要的函数
*/
HookDalvikMethod(method);
JNINativeMethod gMethod[] = {
{info.tMethod, info.tMeihodSig, info.handleFunc},
};

//func为NULL时不自行绑定,后面扩展吧
if(info.handleFunc != NULL){
        //关键!!将目标方法关联到自定义的native方法
        if (jenv->RegisterNatives(clazzTarget, gMethod, 1) < 0) {
            ALOG("RegisterNatives","err");
            return false;
        }
}

    DetachCurrent();
    return true;
}
```

java类中的每个方法都对应一个jMethodID,在Android源码中Method结构体的定义如下：

```
struct Method {
487/* the class we are a part of */
488ClassObject*clazz;
489
490/* access flags; low 16 bits are defined by spec (could be u2?) */
491u4  accessFlags;
492
493/*
494 * For concrete virtual methods, this is the offset of the method
495 * in "vtable".
496 *
497 * For abstract methods in an interface class, this is the offset
498 * of the method in "iftable[n]->methodIndexArray".
499 */
500u2 methodIndex;
501
502/*
503 * Method bounds; not needed for an abstract method.
504 *
505 * For a native method, we compute the size of the argument list, and
506 * set "insSize" and "registerSize" equal to it.
507 */
508u2  registersSize;  /* ins + locals */
509u2  outsSize;
510u2  insSize;
511
512/* method name, e.g. "<init>" or "eatLunch" */
513const char* name;
514
515/*
516 * Method prototype descriptor string (return and argument types).
517 *
518 * TODO: This currently must specify the DexFile as well as the proto_ids
519 * index, because generated Proxy classes don't have a DexFile.  We can
520 * remove the DexFile* and reduce the size of this struct if we generate
521 * a DEX for proxies.
522 */
523DexProtoprototype;
524
525/* short-form method descriptor string */
526const char* shorty;
527
528/*
```

```
529 * The remaining items are not used for abstract or native methods.
530 * (JNI is currently hijacking "insns" as a function pointer, set
531 * after the first call.  For internal-native this stays null.)
532 */
533
534/* the actual code */
535const u2*   insns;  /* instructions, in memory-mapped .dex */
536
537/* JNI: cached argument and return-type hints */
538int jniArgInfo;
539
540/*
541 * JNI: native method ptr; could be actual function or a JNI bridge.  We
542 * don't currently discriminate between DalvikBridgeFunc and
543 * DalvikNativeFunc; the former takes an argument superset (i.e. two
544 * extra args) which will be ignored.  If necessary we can use
545 * insns==NULL to detect JNI bridge vs. internal native.
546 */
547DalvikBridgeFunc nativeFunc;
548
549/*
550 * JNI: true if this static non-synchronized native method (that has no
551 * reference arguments) needs a JNIEnv* and jclass/jobject. Libcore
552 * uses this.
553 */
554bool fastJni;
555
556/*
557 * JNI: true if this method has no reference arguments. This lets the JNI
558 * bridge avoid scanning the shorty for direct pointers that need to be
559 * converted to local references.
560 *
561 * TODO: replace this with a list of indexes of the reference arguments.
562 */
563bool noRef;
564
565/*
566 * JNI: true if we should log entry and exit. This is the only way
567 * developers can log the local references that are passed into their code.
568 * Used for debugging JNI problems in third-party code.
569 */
570bool shouldTrace;
571
572/*
573 * Register map data, if available.  This will point into the DEX file
574 * if the data was computed during pre-verification, or into the
575 * linear alloc area if not.
576 */
577const RegisterMap* registerMap;
578
579/* set if method was called during method profiling */
580boolinProfile;
581};
```

accessflags字段表示方法的属性，例如public，private，native等等， 这份代码的核心也就是修改accessflags字段，实现将java层的函数改为native层我们自己的FakeHook函数

```
bool HookDalvikMethod(jmethodID jmethod){
    Method *method = (Method*)jmethod;
    //关键!!将目标方法修改为native方法
    SET_METHOD_FLAG(method, ACC_NATIVE);

    //获取hook函数的原始参数
    int argsSize = dvmComputeMethodArgsSize(method);
    /*
    如果不是staticmethod，argSize加1的原因，不是staticmethod的函数需要多传入类的实例，也就是this
```

```
    */
    if (!dvmIsStaticMethod(method))
    argsSize++;
    /*
    Method结构体的注释中有这么一段话
    For a native method, we compute the size of the argument list, and set "insSize" and "registerSize" equal to it.
    */
    method->registersSize = method->insSize = argsSize;

    if (dvmIsNativeMethod(method)) {
    method->nativeFunc = dvmResolveNativeMethod;
    method->jniArgInfo = computeJniArgInfo(&method->prototype);
    }
}
```

DalvikMethodHook只是函数实现了将hook函数属性改为native函数，设置Method的insSize和registerSize，并没有将hook函数绑定到我们的native hook函数，RegisterNatives就实现了这个功能，至此java hook已经实现完成。

```
//func为NULL时不自行绑定,后面扩展吧
if(info.handleFunc != NULL){
        //关键!!将目标方法关联到自定义的native方法
        if (jenv->RegisterNatives(clazzTarget, gMethod, 1) < 0) {
            ALOG("RegisterNatives","err");
            return false;
        }
}
```

当我们在Android应用中执行getMacAddress()函数后，就会跳转到我们的native FakeHook函数

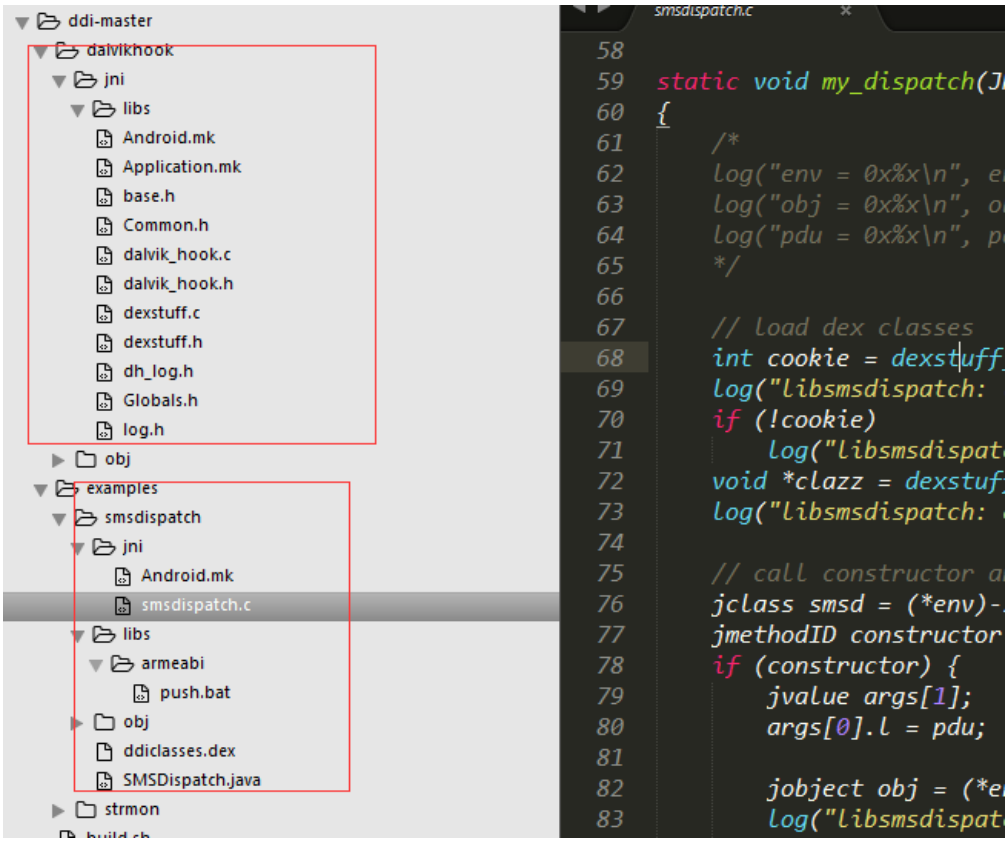
```
//FakeHook函数
JNIEXPORT jstring JNICALL test(JNIEnv *env, jclass clazz)
{
__android_log_print(ANDROID_LOG_VERBOSE, "Log", "call <native_printf> in java");
return (*env)->NewStringUTF(env,"haha ");;
}
```

ddi框架分析

在github上有个开源框架adbi实现了android so的inject和hook，adbi的作者再接再厉实现了java层的hook框架ddi，ddi框架目录如下

dalvikhook目录：实现了java层的hook

exampel目录:smsdispatch和strmon例子



编译

ddi框架的使用需要结合adbi框架，将ddi和adbi放在同一级目录

```
//编译libbase.a 实现so库的hook
cd D:\github\adbi\instruments\base\jni
ndk-build

//编译libdalvikhook.a
cd D:\github\ddi-master\dalvikhook\jni
ndk-build

//编译smsdispatch.so
cd D:\github\ddi-master\examples\smsdispatch\jni
ndk-build
```

ddi代码分析

我们需要将smsdispatch.so注入到要hook的android进程中 先分析smsdisptahc.c的入口

```
// set my_init as the entry point
void __attribute__ ((constructor)) my_init(void);

void my_init(void)
{
    log("libsmsdispatch: started\n")

    debug = 1;
    // set log function for  libbase (very important!)
    set_logfunction(my_log2);
    // set log function for libdalvikhook (very important!)
    dalvikhook_set_logfunction(my_log2);

    hook(&eph, getpid(), "libc.", "epoll_wait", my_epoll_wait, 0);
}
```

hook函数参数：

- arg1：hook_t结构体指针
- arg2:要hook函数所在so库的名称
- arg3:hook函数名称
- arg4:FakeHook函数
- arg5:timeout设置

入口调用hook函数，hook libc.so中的epoll*wait函数，跳转到my*epoll_wait函数

```
static int my_epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout)
{
    int (*orig_epoll_wait)(int epfd, struct epoll_event *events, int maxevents, int timeout);
    orig_epoll_wait = (void*)eph.orig;
    // remove hook for epoll_wait
        //恢复hook处函数原始指令
    hook_precall(&eph);

    // resolve symbols from DVM
        //解析libdvm.so中的函数地址 保存到dexstuff_t当中
    dexstuff_resolv_dvm(&d);

    //protected void dispatchPdus(byte[][] pdus)
    //{"android/net/wifi/WifiInfo","getMacAddress","()Ljava/lang/String;",(void*)test},
    dalvik_hook_setup(&dpdu, "Lcom/android/internal/telephony/SMSDispatcher;", "dispatchPdus", "([[B)V", 2, my_dispatch);
    //dalvik_hook_setup(&dpdu,"Landroid/net/wifi/WifiInfo;","getMacAddress","()Ljava/lang/String;",1,my_getmacaddress);
    dalvik_hook(&d, &dpdu);

    // call original function
    int res = orig_epoll_wait(epfd, events, maxevents, timeout);
    return res;
}
```

my*epoll*wait中实现java hook的函数是dalvik_hook*setup和dalvik*hook

我们先看下dalvik*hook*setup函数

```
int dalvik_hook_setup(struct dalvik_hook_t *h, char *cls, char *meth, char *sig, int ns, void *func)
{
    log("start call exec hook_setup\n")
    if (!h)
        return 0;
   //copy "Lcom/android/internal/telephony/SMSDispatcher;"
    strcpy(h->clname, cls);
    //copy "com/android/internal/telephony/SMSDispatcher"
    strncpy(h->clnamep, cls+1, strlen(cls)-2);
    //copy "dispatchPdus"
    strcpy(h->method_name, meth);
    strcpy(h->method_sig, sig);
    /*ns表示hook函数的参数个数是2
    至于为什么是2，我们看下smsdispatch函数的原型
    protected void dispatchPdus(byte[][] pdus)
    这里只有一个参数pdus 但是在dalvik中还需要传递类的实例参数，类似this
    */

    h->n_iss = ns;
    h->n_rss = ns;
    h->n_oss = 0;
    //FakeHook函数地址
    h->native_func = func;

    h->sm = 0; // set by hand if needed

    h->af = 0x0100; // native, modify by hand if needed

    h->resolvm = 0; // don't resolve method on-the-fly, change by hand if needed
```

```
    //为1，为后面的log输出信息
    h->debug_me = 1;

    return 1;
}
```

再来看下dalvik_hook函数，也是实现java hook的核心部分 Method结构资料

```
void* dalvik_hook(struct dexstuff_t *dex, struct dalvik_hook_t *h)
{
    if (h->debug_me)
        log("dalvik_hook: class %s\n", h->clname)
    //调用libdvm.so函数寻找hook的类
//"Lcom/android/internal/telephony/SMSDispatcher;"这个类
    void *target_cls = dex->dvmFindLoadedClass_fnPtr(h->clname);
    if (h->debug_me)
        log("class = 0x%x\n", target_cls)

    // print class in logcat
    if (h->dump && dex && target_cls)
        dex->dvmDumpClass_fnPtr(target_cls, (void*)1);

    if (!target_cls) {
        if (h->debug_me)
            log("target_cls == 0\n")
        return (void*)0;
    }

    //获取hook函数smsDispatch,返回一个jmethodID 也就是Method对象的指针
    更多Method结构体资料参考
    h->method = dex->dvmFindVirtualMethodHierByDescriptor_fnPtr(target_cls, h->method_name, h->method_sig);
    if (h->method == 0) {
        h->method = dex->dvmFindDirectMethodByDescriptor_fnPtr(target_cls, h->method_name, h->method_sig);
    }

    // constrcutor workaround, see "dalvik_prepare" below
    if (!h->resolvm) {
        h->cls = target_cls;//指向SMSDispatcher类对象
        h->mid = (void*)h->method;//dispatchPdus方法id
    }

    if (h->debug_me)
        log("%s(%s) = 0x%x\n", h->method_name, h->method_sig, h->method)

    if (h->method) {
        //保存原始的insns
        h->insns = h->method->insns;

        if (h->debug_me) {
            log("nativeFunc %x\n", h->method->nativeFunc)

            log("insSize = 0x%x  registersSize = 0x%x  outsSize = 0x%x\n", h->method->insSize, h->method->registersSize, h->method
        }


        /*
        举个例子，如果一个非静态方法有2个参数（没有long和double型的），
        其使用到了5个寄存器（v0-v4），那么参数将置于最后2个寄存器，即v3和v4中，
        而v2是这个方法所在对象的指针，v0和v1是函数自己所需要的本地寄存器。
        这时，registersSize的值是5，而insSize的值是3。
        */
//保存方法原先所需要的参数
        h->iss = h->method->insSize;
        h->rss = h->method->registersSize;
        h->oss = h->method->outsSize;
```

```
        /*
        h->n_iss = ns;ns=2
        h->n_rss = ns;
        h->n_oss = 0;
        h->native_func = func;
        */

        //修改jMethodID
        h->method->insSize = h->n_iss;
        h->method->registersSize = h->n_rss;
        h->method->outsSize = h->n_oss;

        if (h->debug_me) {
            log("shorty %s\n", h->method->shorty)
            log("name %s\n", h->method->name)
            log("arginfo %x\n", h->method->jniArgInfo)

        }
        /*
        jniArgInfo：这个变量记录了一些预先计算好的信息，
        从而不需要在调用的时候再通过方法的参数和返回值实时计算了，
        方便了JNI的调用，提高了调用的速度。如果第一位为1（即0x80000000），
        则Dalvik虚拟机会忽略后面的所有信息，强制在调用时实时计算
        */
        h->method->jniArgInfo = 0x80000000; // <--- also important
        if (h->debug_me) {
            log("noref %c\n", h->method->noRef)
            log("access %x\n", h->method->a)

        }
        //将需要hook的函数修改为native，important
        h->access_flags = h->method->a;
        h->method->a = h->method->a | h->af; // make method native
        if (h->debug_me){
            log("access %x\n", h->method->a)

        }
    //由于前面修改accessflag将要hook的java函数修改为native函数 dvmUseJNIBridge_fnPtr将hook函数绑定到FakeHook函数
        dex->dvmUseJNIBridge_fnPtr(h->method, h->native_func);

        if (h->debug_me){

            log("patched %s to: 0x%x\n", h->method_name, h->native_func)

        }

        return (void*)1;
    }
    else {
        if (h->debug_me){
            log("could NOT patch %s\n", h->method_name)

        }
    }

    return (void*)0;
}
```

至此，我们发现ddi框架实现方法和看雪帖子中实现java hook的核心思想是修改java函数为native 函数，即找到hook函数的jMethodID进行修改

不过ddi框架的FakeHook函数实现了对原函数的调用，

```
/*
    FakeHook函数的实现
    通常hook函数的话都是为了做一些额外的工作，如果想要通过java代码实现功能
    可以在FakeHook内加载dex文件，ddi正是这样做的
```

```
*/
static void my_dispatch(JNIEnv *env, jobject obj, jobjectArray pdu)
{
    /*
    log("env = 0x%x\n", env)
    log("obj = 0x%x\n", obj)
    log("pdu = 0x%x\n", pdu)
    */

    /*
        load dex classes 加载我们自定义的dex
        不过要让/data/dalvik-cache目录具有写权限，因为dex文件是释放在此目录中的
    */
    int cookie = dexstuff_loaddex(&d, "/data/local/tmp/ddiclasses.dex");
    log("libsmsdispatch: loaddex res = %x\n", cookie)
    if (!cookie)
        log("libsmsdispatch: make sure /data/dalvik-cache/ is world writable and delete data@local@tmp@ddiclasses.dex\n")
    //加载dex中的SMSDispatch类
    void *clazz = dexstuff_defineclass(&d, "org/mulliner/ddiexample/SMSDispatch", cookie);
    log("libsmsdispatch: clazz = 0x%x\n", clazz)

    // call constructor and passin the pdu
    jclass smsd = (*env)->FindClass(env, "org/mulliner/ddiexample/SMSDispatch");
    //获取构造函数的jMethodID
    jmethodID constructor = (*env)->GetMethodID(env, smsd, "<init>", "([[B)V");
    if (constructor) {
        jvalue args[1];
        args[0].l = pdu;
        //构造自定义的SMSDispatch实例，执行构造函数
        jobject obj = (*env)->NewObjectA(env, smsd, constructor, args);
        log("libsmsdispatch: new obj = 0x%x\n", obj)

        if (!obj)
            log("libsmsdispatch: failed to create smsdispatch class, FATAL!\n")
    }
    else {
        log("libsmsdispatch: constructor not found!\n")
    }

    // call original SMS dispatch method
    jvalue args[1];
    args[0].l = pdu;
    //恢复原始的Method结构体
    dalvik_prepare(&d, &dpdu, env);
    /*
    调用原始的smsDispatch函数 注意此处的obj是传过来的参数jobject obj,也就是调用smsDispatch函数的SMSDispatcher类的实例引用
    */
    (*env)->CallVoidMethodA(env, obj, dpdu.mid, args);
    log("success calling : %s\n", dpdu.method_name)
    //恢复被hook之后的Method结构体内容
    dalvik_postcall(&d, &dpdu);
}
```

java hook实战

这里我使用ddi实现hook中的WifiInfo.class类中的getMacAddress，并且在FakeHook函数中调用java函数，并且调用原始的getMacAddress函数

1.java层代码编写，生成dex文件

创建getMacAddressHook.java文件，目录位于..../com/example/javahook/下getMacAddressHook.java package com.example.javahook;

```
public class getMacAddressHook{
    public  getMacAddressHook(){
        System.out.println("this is a joke");
    }
}
```

```
用法:
javac -source 1.6 -target 1.6 getMacAddressHook.java
//切换到package目录 在src目录执行
dx --dex --output=getMacAddressHook.dex com/example/javahook/getMacAddressHook.class
由于我使用的jdk version 是1.8 使用dx工具会提示无效的class文件 这里将java强制编译为了1.6版本
-source 1.6表示java编译器版本为1.6 -target 1.6表示运行在1.6版本的jvm中
如果我们的java文件是在一个package中的话。例如package com.example.javahook;
则java目录看起来如这样.....\JavaHijack\src\com\example\javahook\getMacAddressHook.java
如果我们直接在javahook目录下用dx执行:dx --dex --output=getMacAddressHook.dex getMacAddressHook.class
会提示class name <.....> not match path error
参考http://stackoverflow.com/questions/15085602/android-javac-and-dx-trouble-processing-class-name-and-path-do-not-match
```

2.将自定义dex文件pull到/data/local/tmp/目录

1.native层代码编写

我将smsdispatch这个例子改写一下，实现getMacAddress的hook 在my*epoll*wait函数中，修改dalvik*hook*setup的参数

```
/*
"Landroid/net/wifi/WifiInfo;"hook函数所在的类
"getMacAddress" hook函数
()Ljava/lang/String;函数签名sig String getMacAddress()
1:将getMacAddress改为native函数后 的餐宿
my_getmacaddress：FakeHook函数
*/
dalvik_hook_setup(&dpdu,"Landroid/net/wifi/WifiInfo;","getMacAddress","()Ljava/lang/String;",1,my_getmacaddress);
```

当我们修改了上述代码之后，android程序调用getMacAddress()函数就会调用我们的FakeHook函数

```
static jstring my_getmacaddress(JNIEnv *env, jobject obj){
    log("having enter fakemacaddress\n");


    // load dex classes
    int cookie = dexstuff_loaddex(&d, "/data/local/tmp/getMacAddressHook.dex");
    log("libgetMacAddressHook: loaddex res = %x\n", cookie)
    if (!cookie)
        log("libsmsdispatch: make sure /data/dalvik-cache/ is world writable and delete data@local@tmp@ddiclasses.dex\n")
    //加载我们自己写的类getMacAddressHook
    void *clazz = dexstuff_defineclass(&d, "com/example/javahook/getMacAddressHook", cookie);
    log("libgetMacAddressHook: clazz = 0x%x\n", clazz)

    // call constructor and passin the pdu
    jclass smsd = (*env)->FindClass(env, "com/example/javahook/getMacAddressHook");
    //寻找构造函数
    jmethodID constructor = (*env)->GetMethodID(env, smsd, "<init>", "()V");
    if (constructor) {

        //调用构造函数，我们写的构造函数没有arg
        jobject fakeobj = (*env)->NewObject(env, smsd, constructor);
        log("libgetMacAddressHook: new obj = 0x%x\n", fakeobj)

        if (!fakeobj)
            log("libgetMacAddressHook: failed to create smsdispatch class, FATAL!\n")
    }
    else {
        log("libgetMacAddressHook: constructor not found!\n")
    }

    //恢复getMacAddress()函数的jMethodID信息
    dalvik_prepare(&d, &dpdu, env);
    //调用原始的getMacAddress函数
    jstring result=(jstring)((*env)->CallObjectMethod(env, obj, dpdu.mid));

    log("Mac address is 0x%x\n", result);
    log("success calling : %s\n", dpdu.method_name)
```

```
    dalvik_postcall(&d, &dpdu);
    return result;
}
```

既然完成了hook getMacAddressHook代码的编写，我们运行一下

运行结果测试

确保将getMacAddressHook.dex和libgetMacAddressHook.so放到/data/loal/tmp 以及/data/dalvik-cache目录具有写权限

开启adb logcat -s "System.out" 没hook之前点击HookMe按钮

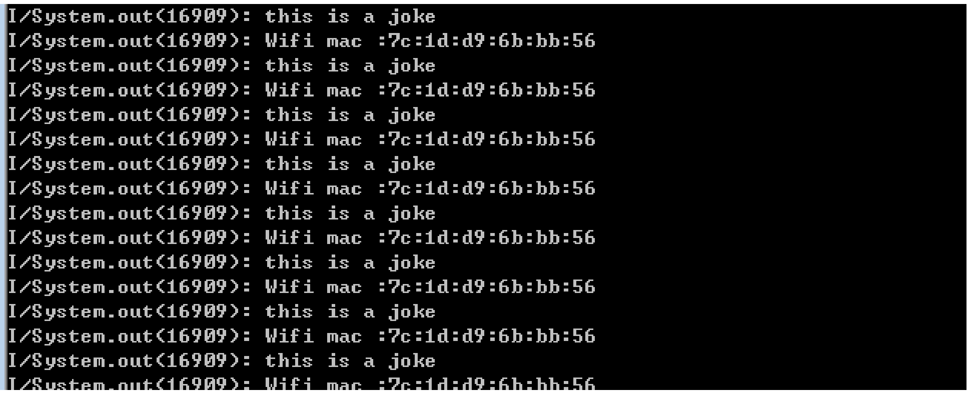
```
/System.out(15984): name=com.job.android  versioncode=520
/System.out(15725): Wifi mac :7c:1d:d9:6b:bb:56
/System.out(15725): Wifi mac :7c:1d:d9:6b:bb:56
/System.out(15725): Wifi mac :7c:1d:d9:6b:bb:56
```

```
adb shell
su
cd /data/local/tmp
ll
//注入libgetMacAddressHook.so进行hook
./inject pid /data/local/tmp/libgetMacAddressHook.so
```

```
-rw-rw-rw- shell    shell      559908 2014-10-23 03:39 ddiclasses.dex
-rw-rw-rw- shell    shell         692 2016-09-05 09:53 getMacAddressHook.dex
-rw------- system   system     170986 2016-09-06 01:26 glsl_shader_log.txt
-rwxrwxr-x shell    shell       21972 2016-08-24 09:24 hijack
-rwxrwxr-x shell    shell       13644 2016-08-29 21:30 inject
-rw-rw-rw- shell    shell     7962147 2016-08-24 13:47 libandroid_runtime.idb
-rw-rw-rw- shell    shell       21624 2016-08-29 23:28 libexample.so
-rw-rw-rw- shell    shell       29816 2016-09-07 12:26 libgetMacAddressHook.so
-rw-rw-rw- shell    shell       29816 2016-09-05 12:18 libsmsdispatch.so
-rw-rw-rw- shell    shell       17604 2016-08-27 23:27 libso.so
-rwxrwxrwx root     root         8708 2016-09-05 12:06 mtools
drwxrwxrwx root     root              2016-09-03 12:11 zgo
```

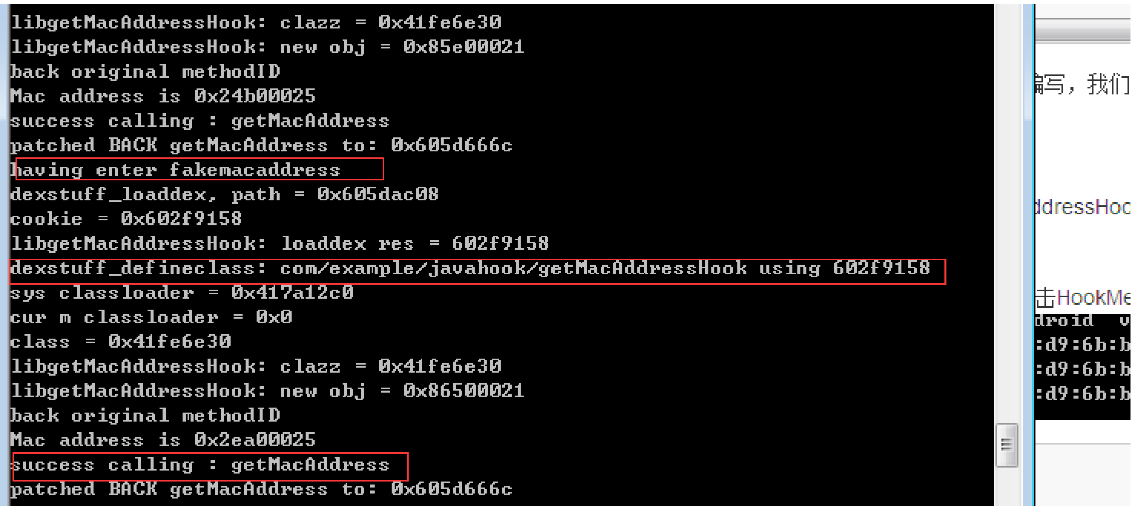
```
[+] Target process returned from dlopen, return value=0, pc=0
root@dior:/data/local/tmp # ps |grep hook
u0_a105   16909 227   534928 35332 ffffffff 4011f8a0 S com.example.ddihook
o/libgetMacAddressHook.so
target pid is 16909
inject so path is /data/local/tmp/libgetMacAddressHook.so
[+] Injecting process: 16909
[+] get_remote_addr: local[b6f83000], remote[400fe000]
[+] Remote mmap address: 40110dc5
[+] Calling mmap in target process.
[+] Target process returned from mmap, return value=605d1000, pc=0
[+] get_remote_addr: local[b6fe8000], remote[400e3000]
[+] get_remote_addr: local[b6fe8000], remote[400e3000]
[+] get_remote_addr: local[b6fe8000], remote[400e3000]
[+] get_remote_addr: local[b6fe8000], remote[400e3000]
[+] Get imports: dlopen: 400e3f31, dlsym: 400e3e81, dlclose: 400e3dfd, dlerror:
400e3dad
library path = /data/local/tmp/libgetMacAddressHook.so
[+] Calling dlopen in target process.
[+] Target process returned from dlopen, return value=5b955a48, pc=0
root@dior:/data/local/tmp # ./inject  16909 /data/local/tmp/libgetMacAddressHo>
```

再点击HookMe按钮 成功执行了我们加载的dex代码输出信息"This is a joke "并调用了原始的getMacAddress()函数



```
I/System.out(16909): this is a joke
I/System.out(16909): Wifi mac :7c:1d:d9:6b:bb:56
I/System.out(16909): this is a joke
I/System.out(16909): Wifi mac :7c:1d:d9:6b:bb:56
I/System.out(16909): this is a joke
I/System.out(16909): Wifi mac :7c:1d:d9:6b:bb:56
I/System.out(16909): this is a joke
I/System.out(16909): Wifi mac :7c:1d:d9:6b:bb:56
I/System.out(16909): this is a joke
I/System.out(16909): Wifi mac :7c:1d:d9:6b:bb:56
I/System.out(16909): this is a joke
I/System.out(16909): Wifi mac :7c:1d:d9:6b:bb:56
I/System.out(16909): this is a joke
I/System.out(16909): Wifi mac :7c:1d:d9:6b:bb:56
I/System.out(16909): this is a joke
I/System.out(16909): Wifi mac :7c:1d:d9:6b:bb:56
```

查看生成的smsdispatch.log信息



```
libgetMacAddressHook: clazz = 0x41fe6e30
libgetMacAddressHook: new obj = 0x85e00021
back original methodID
Mac address is 0x24b00025
success calling : getMacAddress
patched BACK getMacAddress to: 0x605d666c
having enter fakemacaddress
dexstuff_loaddex, path = 0x605dac08
cookie = 0x602f9158
libgetMacAddressHook: loaddex res = 602f9158
dexstuff_defineclass: com/example/javahook/getMacAddressHook using 602f9158
sys classloader = 0x417a12c0
cur m classloader = 0x0
class = 0x41fe6e30
libgetMacAddressHook: clazz = 0x41fe6e30
libgetMacAddressHook: new obj = 0x86500021
back original methodID
Mac address is 0x2ea00025
success calling : getMacAddress
patched BACK getMacAddress to: 0x605d666c
```

参考资料

jni函数积累

android ddi框架分析 important

android so的注入和hook(for x86 and arm)

老罗的博客:浅谈android中log的使用

Android so注入挂钩-Adbi 框架如何实现dalvik函数挂钩