Hook梦幻旅途之Frida

原创雪狼别动队 酒仙桥六号部队
2020-09-04原文

这是 **酒仙桥六号部队** 的第 **75** 篇文章。

全文共计8297个字，预计阅读时长25分钟。

一、基础知识

Frida是全世界最好的Hook框架。在此我们详细记录各种各样常用的代码套路，它可以帮助逆向人员对指定的进程的so模块进行分析。它主要提供了功能简单的python接口和功能丰富的js接口，使得hook函数和修改so编程化，值得一提的是接口中包含了主控端与目标进程的交互接口，由此我们可以即时获取信息并随时进行修改。使用frida可以获取进程的信息（模块列表，线程列表，库导出函数），可以拦截指定函数和调用指定函数，可以注入代码，总而言之，使用frida我们可以对进程模块进行手术刀式剖析。

1.1 Frida安装

需要安装Python Frida库以及对应手机架构的Frida server，Frida如果安装极慢或者失败，原因在于国内网络状况。

1.1.1 启动进程

启动手机Frida server进程

```
adb shell


su

cd /data/local/tmp


chmod 777 frida-server


./frida-server
```

PS：/data/local/tmp是一个放置frida server的常见位置。

1.1.2 混合运行Frida

以Python+Javascript混合脚本方式运行Frida（两种模式）。

```
// 以附加模式启动（Attach）

// 要求待测试App正在运行
```

run.py文件

```
// 导入frida库，sys系统库用于让脚本持续运行

import sys

import frida

# 找寻手机frida server

device = frida.get_usb_device()

# 选择应用进程（一般为包名）

appPackageName =""
```

```
# 附加
session = device.attach(appPackageName)
# 加载脚本,填入脚本路径
with open("script.js", encoding="utf-8")as f:
    script = session.create_script(f.read())

script.load()
sys.stdin.read()    //也可以不依赖sys库，使用time.sleep(10000000);

script.js文件
setImmediate(function() {
    //prevent timeout
    console.log("[*] Starting script");
    Java.perform(function() {
        // 具体逻辑
    })
})
########################################################################
####################
// 启动新的进程（Spawn）
// 不要求待测试App正在运行，Frida会启动一个新的App进程并挂起
//
优点：因为是Frida启动的进程，在启动的同时注入frida代码，所以Hook的时机
很早。
```

```
// 适用于在进程启动前的一些hook，如hook
RegisterNative、较早进行的加解密等，注入完成后调用resume恢复进程。
//
缺点：会Hook到从App启动→想要分析的界面和逻辑的内容，干扰项多，且容易卡死。
```

run.py文件

```
import sys
import frida
# 找寻手机frida server
device = frida.get_usb_device()
# 选择应用进程（一般为包名）
appPackageName =""
# 启动新进程
pid = device.spawn([appPackageName])
device.resume(pid)
session = device.attach(pid)
# 加载脚本,填入脚本路径
with open("script.js", encoding="utf-8")as f:
    script = session.create_script(f.read())
script.load()
sys.stdin.read()//也可以不依赖sys库，使用time.sleep(10000000);
```

script.js文件

```
setImmediate(function() {
```

```
    //prevent timeout
    console.log("[*] Starting script");
    Java.perform(function() {
        // 具体逻辑
    })
})
```

PS:脚本的第一步总是通过get_usb_device用于寻找USB连接的手机设备，这是因为Frida是一个跨平台的Hook框架，它也可以Hook Windows、mac等PC设备，命令行输入frida-ls-devices可以展示当前环境所有可以插桩的设备，输入frida-ps展示当前PC所有进程（一个进程往往意味着一个应用），frida-ps -U即意味着展示usb所连接设备的进程信息。你可以通过Python+Js混合脚本的方式操作Frida，但其体验远没有命令行运行Frida Js脚本丝滑。

1.1.3 获取前端进程

获取最前端Activity所在的进程，进程名。

```
// 可以省去填写包名的困扰
device = frida.get_usb_device()
front_app = device.get_frontmost_application()
print(front_app)
front_app_name = front_app.identifier
print(front_app_name)
```

输出1：Application(identifier="com.xxxx.xxx", name="xxxx", pid=xxxx)

输出2: com.xxxx.xxxx

1.1.4 命令行调用

命令行方式使用：

Spawn方式

```
frida -U --no-pause -f packageName -l scriptPath
```

Attach方式

```
frida -U --no-pause packageName -l scriptPath
```

输出内容太多时，可以将输出导出至文件

```
frida -U --no-pause -f packageName -l scriptPath -o savePath
```

可以自行查看所有的可选参数。

```
➜  frida  frida -h
Usage: frida [options] target

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -D ID, --device=ID    connect to device with the given ID
  -U, --usb             connect to USB device
  -R, --remote          connect to remote frida-server
  -H HOST, --host=HOST  connect to remote frida-server on HOST
  -f FILE, --file=FILE  spawn FILE
  -n NAME, --attach-name=NAME
                        attach to NAME
  -p PID, --attach-pid=PID
                        attach to PID
  --debug               enable the Node.js compatible script debugger
  --enable-jit          enable JIT
  -l SCRIPT, --load=SCRIPT
                        load SCRIPT
  -e CODE, --eval=CODE  evaluate CODE
  -q                    quiet mode (no prompt) and quit after -l and -e
  --no-pause            automatically start main thread after startup
  -o LOGFILE, --output=LOGFILE
                        output to log file
```

通过CLI 进行hook有诸多优势，列举两个：

1） 当脚本出错时，会提供很好的错误提示；

```
{"moduleName":"libyoga.so","methodName":"jni_YGConfigFree","signature":"(J)V","address":"0x8f907281","IdaAddress":"0x628
1"}
{"moduleName":"libyoga.so","methodName":"jni_YGConfigSetExperimentalFeatureEnabled","signature":"(JIZ)V","address":"0x8f
9072c5","IdaAddress":"0x62c5"}
{"moduleName":"libyoga.so","methodName":"jni_YGConfigSetUseWebDefaults","signature":"(JZ)V","address":"0x8f907315","IdaA
ddress":"0x6315"}
{"moduleName":"libyoga.so","methodName":"jni_YGConfigSetPointScaleFactor","signature":"(JF)V","address":"0x8f907361","Id
aAddress":"0x6361"}
{"moduleName":"libyoga.so","methodName":"jni_YGConfigSetUseLegacyStretchBehaviour","signature":"(JZ)V","address":"0x8f90
73a9","IdaAddress":"0x63a9"}
{"moduleName":"libyoga.so","methodName":"jni_YGConfigSetLogger","signature":"(JLjava/lang/Object;)V","address":"0x8f9081
89","IdaAddress":"0x7189"}
[Xiaomi MI 4LTE::com.sankuai.moviepro]->
[Xiaomi MI 4LTE::com.sankuai.moviepro]->
[Xiaomi MI 4LTE::com.sankuai.moviepro]-> ReferenceError: identifier 'nativrAddress' undefined
    at [anon] (../../../frida-gum/bindings/gumjs/duktape.c:81144)
    at getNativeAddress (/repl2.js:25)
    at /repl2.js:14
    at frida/node_modules/frida-java-bridge/lib/vm.js:11
    at E (frida/node_modules/frida-java-bridge/index.js:346)
    at frida/node_modules/frida-java-bridge/index.js:298
    at frida/node_modules/frida-java-bridge/lib/vm.js:11
    at frida/node_modules/frida-java-bridge/index.js:278
    at /repl2.js:68
```

第25行出问题

2)Frida进程注入后和原JS脚本保持同步，只需要修改原脚本并保存，进程就会自动使用修改后的脚本，这会让出错→修复，调试→修改调试目标 的过程更迅捷。

1.2 Frida In Java

1.Frida hook 无重载Java方法；

2.Frida hook 有重载Java方法；

3.Frida hook Java方法的所有重载。

1.2.1 Hook导入导出表函数地址

对So的Hook第一步就是找到对应的指针（内存地址），Frida提供了各式各样的API帮助我们完成这一工作。

获得一个存在于导出表的函数的地址：

```
// 方法一
var so_name = "";
var function_name = "";
var this_addr = Module.findExportByName(so_name, function_name);
// 方法二
var so_name = "";
var function_name = "";
var this_addr = Module.getExportByName(so_name, function_name);
//
区别在于当找不到该函数时findExportByName返回null，而getExportByName抛出异常。
```

```
// 方法三
var so_name = "";
var function_name = "";
var this_addr = "";
var i = undefined;
var exports = Module.enumerateExportsSync(so_name);
for(i=0; i<exports.length; i++){
    if(exports[i].name == function_name){
        var this_addr = exports[i].address;
        break;
    }
}
```

1.2.2 枚举进程模块/导出函数

枚举某个进程的所有模块/某个模块的所有导出函数。

Frida与IDA交互：

1.内存地址和IDA地址相互转换；

```
function memAddress(memBase, idaBase, idaAddr) {
    var offset = ptr(idaAddr).sub(idaBase);
    var result = ptr(memBase).add(offset);
    return result;
}

function idaAddress(memBase, idaBase, memAddr) {
```

```
    var offset = ptr(memAddr).sub(memBase);
    var result = ptr(idaBase).add(offset);
    return result;
}
```

二、Hook JNI函数

JNI很多概念十分模糊，我们做如下定义，后续的阐述都依照此定义。

- native：特指Java语言中的方法修饰符native。
- Native方法：特指Java层中声明的、用native修饰的方法。
- JNI实现方法：特指Native方法对应的JNI层的实现方法。
- JNI函数：特指JNIEnv提供的函数。
- Native函数：泛指C/C++层的本地库/自写函数等。

2.1 JNI编程模型

如果对JNI以及NDK开发了解较少，务必阅读如下资料。（我不要你觉得，听我　的，下面都是精挑细选的。）

- 《深入理解Android　卷1》——第二章：深入理解JNI　作者邓凡平
- 《Android的设计与实现　卷1》——第二章：框架基础JNI　作者杨云君

除此之外，你可能还会想了解一些其他的知识，我们回顾一下JNI编程模型。

步骤1：Java层声明Native方法。

步骤2：JNI层实现Java层声明的Native方法，在JNI层可以调用底层库/回调Java方法。这部分将被编译为动态库（SO文件）供系统加载。

步骤3：加载JNI层代码编译后生成的SO文件。

这其中有一个额外的关键点，SO文件的架构。

C/C++等Native语言直接运行在操作系统上，由CPU执行代码，所以编译后的文件既和操作系统有关，也和CPU相关。So是C/C++代码在Linux系统中编译后的文件，Window系统中为dll格式文件。

Android手机的CPU型号千千万，但CPU架构主要有七种，Mips，Mips64位，x86，x86_64，armeabi，armv7-a，armv8，编译时我们需要生成这七种架构的so文件以适配各种各样的手机。

2.2 armv7a架构成因

在反编译过程中，我们需要选择某种CPU架构的so文件，得到特定架构的汇编代码。一般情况下我们选择armv7a架构，这涉及到一系列连环的原因。

2.2.1 通用情况

七种架构可以简单分为Mips，X86，ARM三家，前两者的在Android处理器市场占比极小。Arm架构几乎成为了Android处理器的行业标准，IOS和Android都采用ARM架构处理器。

2.2.2 Apk臃肿考虑

Apk的包体积对下载转化率、分发费直接挂钩，所以Apk一旦度过初创时期，就要考虑Apk的包体积优化，而So文件往往占据1/3-1/2的包体积，不提供市场占有率极小的Mips以及X86系列的So，可以瞬间解决Apk臃肿。

2.2.3 形势考虑

形势比人强，ARM如日中天，无奈之下Mips和X86都设计了用于转换ARM汇编的中间层，即使Apk只提供了ARM的So库文件，这两种CPU架构的手机也可以以较慢速度运行APK。

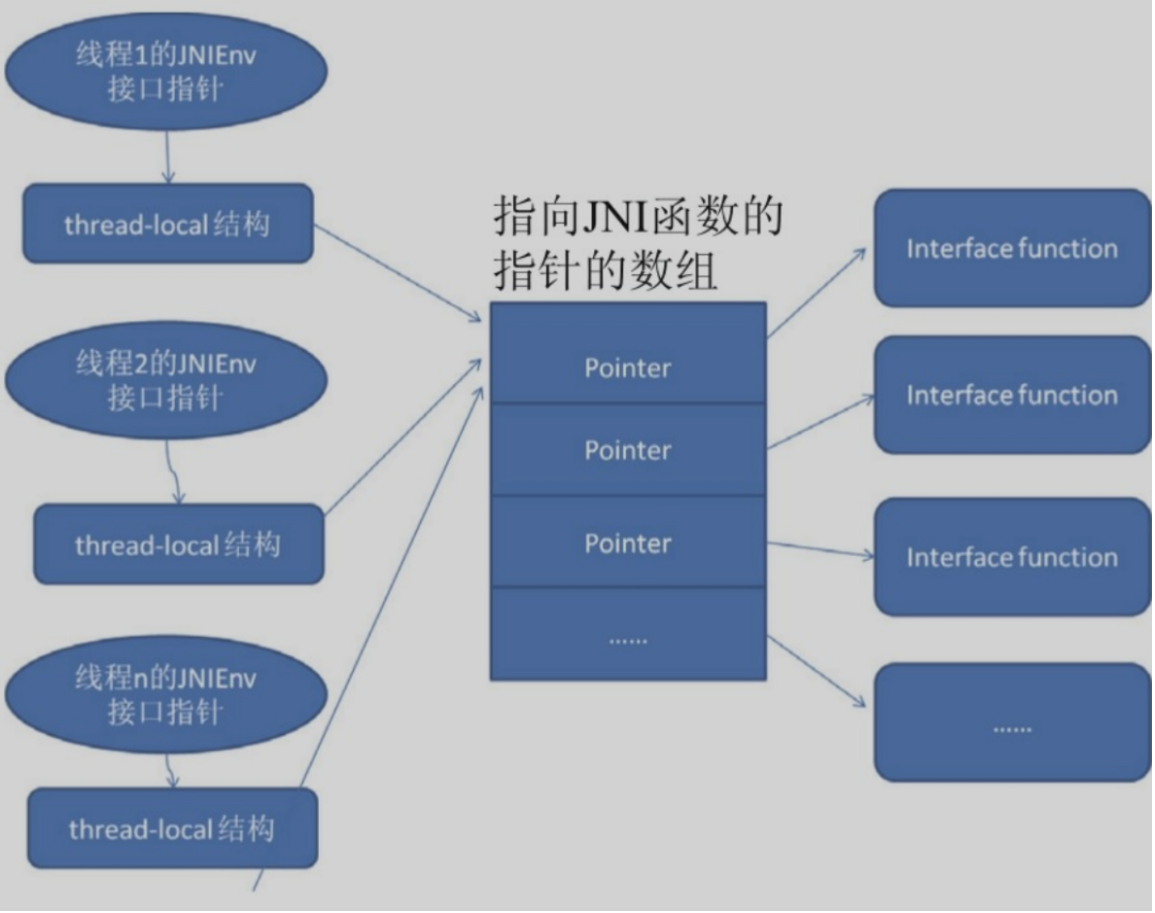
2.2.4 ARM兼容性

ARM有armeabi，armv7a，armv8a这三个系列，系列之间是不断发展和完善的升级关系。目前主流手机的CPU都是armv8a，即64位的ARM设备，而armeabi甚至只用在Android 4.0以下的手机，但好在Arm是向下兼容的，如果Apk不需要用到一些高性能的东西，完全可以只提供armeabi的So，这样几乎可以支持所有架构的手机。

2.3 Hook JNI函数

通过上述的学习我们了解到，JNIEnv提供给了我们两百多个函数，帮助我们将Java中的对象和数据转换成C/C++的类型，帮助我们调用Java函数、帮助我们将C中生成的结果转换回Java中的对象和数据并返回，因此，如果能Hook JNI函数，会对我们逆向与分析So产生帮助。

使用Frida Hook Native函数十分简单，只需要我们提供地址即可。



Frida提供了一种非常方便优雅的方式获得JNIEnv的地址，需要注意的是必须在Java.perform中调用。

```
var jnienv_addr = 0x0;

Java.perform(function(){

    jnienv_addr = Java.vm.getEnv().handle.readPointer();

});

console.log("JNIEnv base adress get by
Java.vm.getEnv().handle.readPointer():" + jnienv_addr);
```

JNIEnv指针指向JNINativeInterface这个数组，里面包含两百多个指针，即各种各样的JNI函数。

我们可以查看一下Jni.h头文件

```
/*
 * Table of interface function pointers.
 */
struct JNINativeInterface {
    void*       reserved0;
    void*       reserved1;
    void*       reserved2;
    void*       reserved3;

    jint        (*GetVersion)(JNIEnv *);

    jclass      (*DefineClass)(JNIEnv*, const char*, jobject, const jbyte*,
                        jsize);
    jclass      (*FindClass)(JNIEnv*, const char*);

    jmethodID   (*FromReflectedMethod)(JNIEnv*, jobject);
    jfieldID    (*FromReflectedField)(JNIEnv*, jobject);
    /* spec doesn't show jboolean parameter */
    jobject     (*ToReflectedMethod)(JNIEnv*, jclass, jmethodID, jboolean);

    jclass      (*GetSuperclass)(JNIEnv*, jclass);
    jboolean    (*IsAssignableFrom)(JNIEnv*, jclass, jclass);

    /* spec doesn't show jboolean parameter */
    jobject     (*ToReflectedField)(JNIEnv*, jclass, jfieldID, jboolean);

    jint        (*Throw)(JNIEnv*, jthrowable);
    jint        (*ThrowNew)(JNIEnv *, jclass, const char *);
    jthrowable  (*ExceptionOccurred)(JNIEnv*);
    void        (*ExceptionDescribe)(JNIEnv*);
    void        (*ExceptionClear)(JNIEnv*);
    void        (*FatalError)(JNIEnv*, const char*);

    jint        (*PushLocalFrame)(JNIEnv*, jint);
```

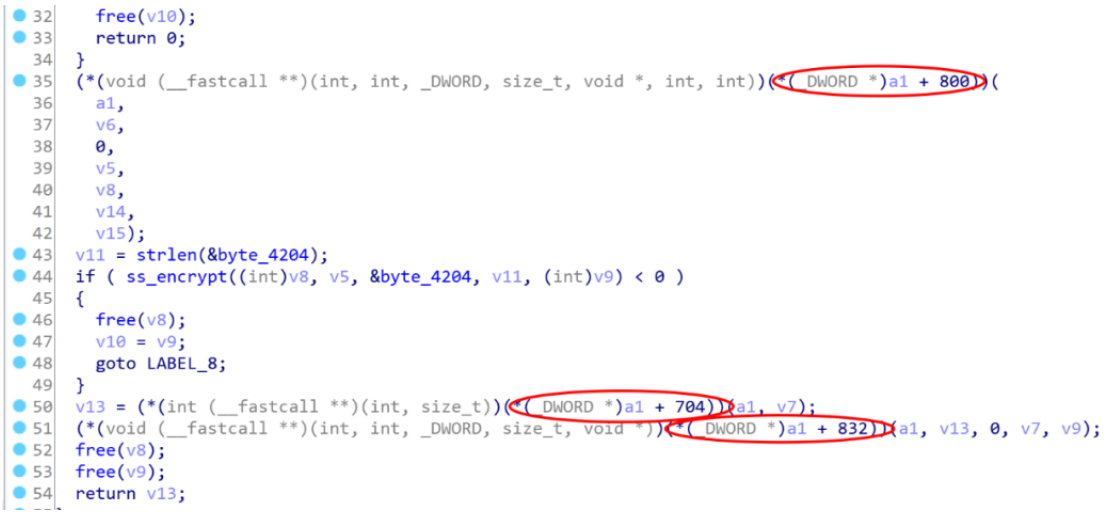
假设JNIEnv地址为0x1000，一个指针长4，那么reversed0地址即为0x1000，reversed1为0x1004，之后我们读取这个指针，就可以得到JNI函数的地址，从而实现Hook。

在我们上述的JNINativeInterface数组中，它排在第七个，那么偏移就是4*(7-1)=24。

```
function hook_native_findclass() {

    var jnienv_addr = Java.vm.getEnv().handle.readPointer();

    var FindClassPtr = Memory.readPointer(jnienv_addr.add(24));

    // 注意，Frida提供了add(+),sub(-
)等函数供我们做加减乘除，你也可以通过add(0x12)这种形式加一个十六进制数
。

    console.log("FindClassPtr addr: " + FindClassPtr);

    Interceptor.attach(FindClassPtr, {
```

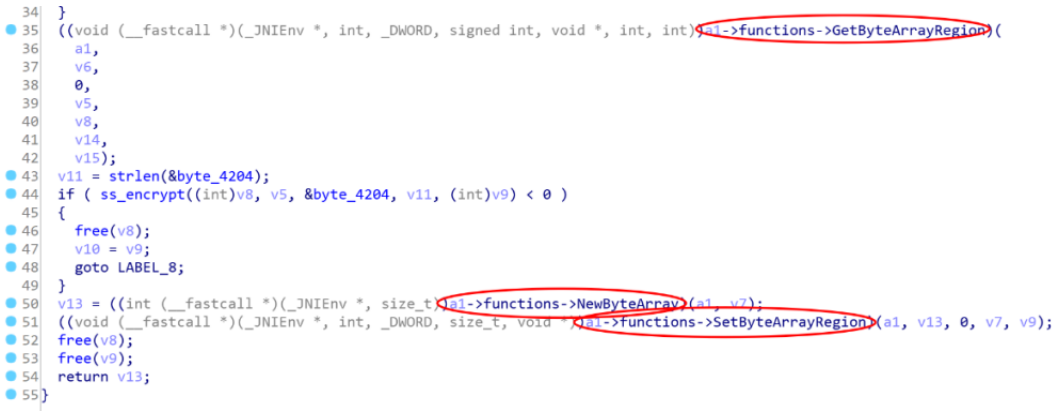
```
        onEnter: function (args) {
            ...
        }
    });
}
```

接下来我们以IDA为例，加深理解。在我们使用IDA逆向和分析SO时，如果单纯导入SO，会有大量“无法识别”的函数。



```
32    free(v10);
33    return 0;
34  }
35  (*(void (__fastcall **)(int, int, _DWORD, size_t, void *, int, int))(*(_DWORD *)a1 + 800))(
36    a1,
37    v6,
38    0,
39    v5,
40    v8,
41    v14,
42    v15);
43  v11 = strlen(&byte_4204);
44  if ( ss_encrypt((int)v8, v5, &byte_4204, v11, (int)v9) < 0 )
45  {
46    free(v8);
47    v10 = v9;
48    goto LABEL_8;
49  }
50  v13 = (*(int (__fastcall **)(int, size_t))(*(_DWORD *)a1 + 704))(a1, v7);
51  (*(void (__fastcall **)(int, int, _DWORD, size_t, void *))(*(_DWORD *)a1 + 832))(a1, v13, 0, v7, v9);
52  free(v8);
53  free(v9);
54  return v13;
```

所以惯例上，我们会导入Jni.h头文件，再设置方法的第一个参数为JNIEnv类型，这样IDA就能顺利将形如*（a1+xxx）这种指针识别为JNI函数 ，但可能很多人没有想过为什么这样可以成功。



```
34  }
35  ((void (__fastcall *)(_JNIEnv *, int, _DWORD, signed int, void *, int, int))(*a1)->functions->GetByteArrayRegion)(
36    a1,
37    v6,
38    0,
39    v5,
40    v8,
41    v14,
42    v15);
43  v11 = strlen(&byte_4204);
44  if ( ss_encrypt((int)v8, v5, &byte_4204, v11, (int)v9) < 0 )
45  {
46    free(v8);
47    v10 = v9;
48    goto LABEL_8;
49  }
50  v13 = ((int (__fastcall *)(_JNIEnv *, size_t))(*a1)->functions->NewByteArray)(a1, v7);
51  ((void (__fastcall *)(_JNIEnv *, int, _DWORD, size_t, void *))(*a1)->functions->SetByteArrayRegion)(a1, v13, 0, v7, v9);
52  free(v8);
53  free(v9);
54  return v13;
55}
```

事实上，导入Jni.h头文件是为了引入JNINativeInterface与JNIInvokeInterface结构体信息，而转换参数一为JNIEnv类型，就是在提醒IDA，将*(env+704)映射成对应的JNIEnv函数。

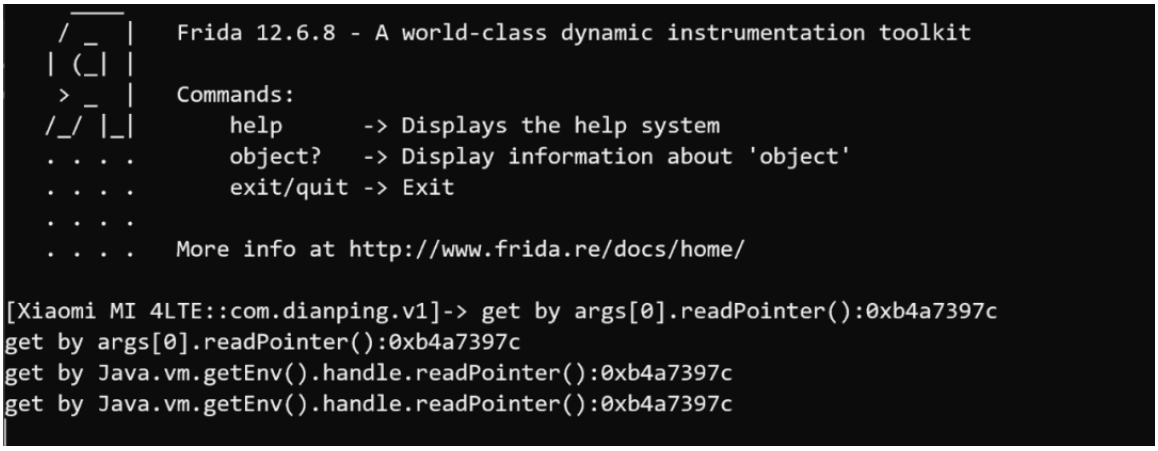
而我们现在所做的是一种相反的操作，已知各个JNI函数的名字和他们在数组中的位置，希望得到其地址。

不知道大家是否发现，由于JNI实现方法的第一个参数总是JNIEnv，所以我们也可以通过Hook一个JNI实现方法作为跳板，从而获得JNIEnv的地址。

```
function hook_jni(){
    var so_name = ""; // 请选择目标Apk SO
    var function_name = "";  //请选择目标SO中一个JNI实现方法
    var open_addr = Module.findExportByName(so_name,
function_name);
    Interceptor.attach(open_addr, {
        onEnter: function (args) {
            var jnienv_addr = 0x0;
            console.log("get by args[0].readPointer():" +
args[0].readPointer());
            Java.perform(function () {
                jnienv_addr =
Java.vm.getEnv().handle.readPointer();
            });
            console.log("get by
Java.vm.getEnv().handle.readPointer():" + jnienv_addr);
        },
        onLeave: function (retval) {
```

```
        }
    });
}

hook_jni();
```

结果完全正确，但这种方法流程明显更加复杂，不够优雅，不建议使用。



好了，我们回归到主线上来，上面我们Hook了FindClass这个函数，想一下我们Hook一个JNI函数需要做的工作，一是找到这个函数对应的偏移，二是在onEnter和onLeave中编写具体的逻辑，因为每个JNI函数的参数和返回值都不一样。

有没有办法简化这两个步骤呢？比如只需要输入JNI函数名，而不需要手动计算偏移？这个好办，我们看一下代码。

```
var jni_struct_array = [
    "reserved0",
    "reserved1",
    "reserved2",
```

```
    "reserved3",
    "GetVersion",
    "DefineClass",
    "FindClass",
    *******此处省略两百多个JNI函数**********
    "FromReflectedMethod",
    "FromReflectedField",
    "ExceptionCheck",
    "NewDirectByteBuffer",
    "GetDirectBufferAddress",
    "GetDirectBufferCapacity",
    "GetObjectRefType",
]

function getJNIFunctionAdress(jnienv_addr,func_name){
    var offset = jni_struct_array.indexOf(func_name) * 4;
    return Memory.readPointer(jnienv_addr.add(offset))
}
```

代码很简单，将JNI函数罗列在数组中，通过Js中indexOf这个数组处理函数得到目标数组的索引，乘4就是偏移了，除此之外，你可以选择乘Process.pointerSize，这是Frida提供给我们的Api，返回当前平台指针所占用的内存大小，这样做可以增加脚本的移植性（其实没啥区别）。

我们进一步希望，能不能不用在onEnter和onLeave中编写具体的逻辑，反正JNI函数的参数和返回值类型都在Jni.h中定义好了，也不会有什么更多的变化了。

需要注意的是，它在理论上实现了Hook所有JNI函数，并提供了人性化的筛选等功能，但在我的测试机上并没有很顺利或者正确的打印出全部JNI调用，更多精彩需要读者自己去挖掘喽。

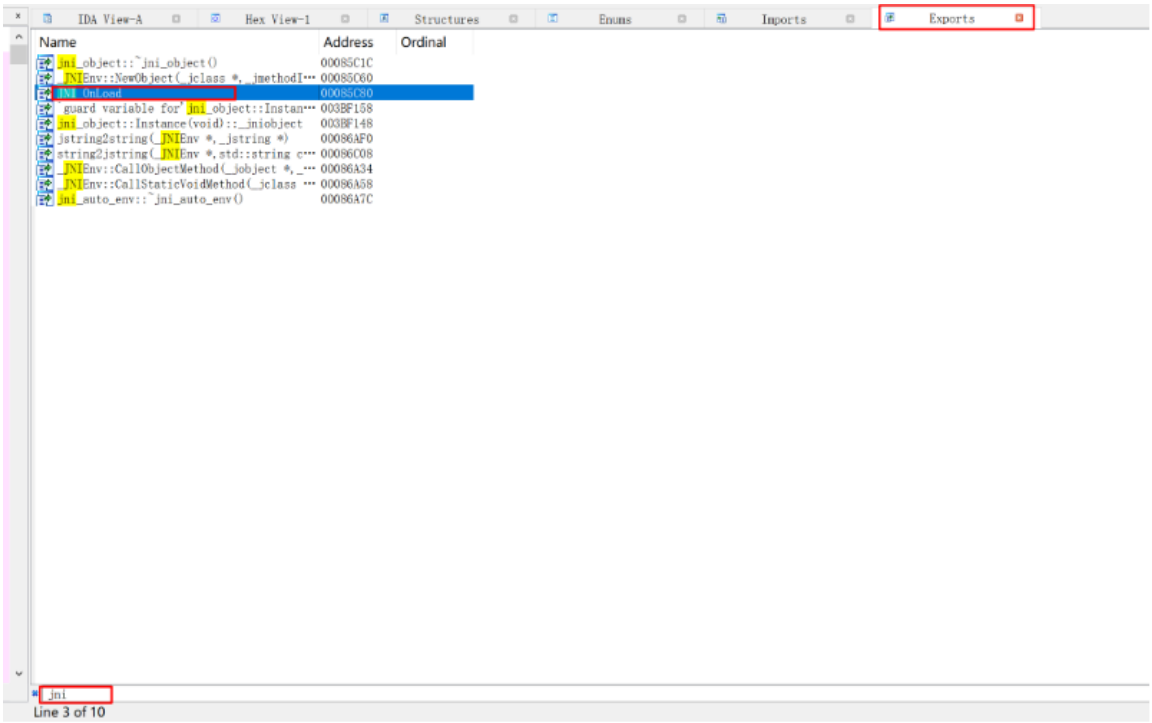
三、Hook动态注册函数

在第二部分我们将尝试Hook JNIEnv提供的RegisterNatives函数，在上面我们已经讲过JNI函数的Hook，为什么要花同样的篇幅去讲解呢？当然是因为这个函数比较常用，而且可以给分析带来很大帮助。

3.1 反编译so文件

在逆向时，静态注册的函数只需要找到对应的So，函数导出表中搜索即可定位。而动态注册的函数会复杂一些，下面列一下流程。

1.在导出函数中搜索JNI_OnLoad,点击进入。



2.Tab 或者 f5 键反汇编 arm 指令。

```
int __fastcall JNI_OnLoad(int a1)
{
  int v1; // r4
  int v2; // r5
  int v3; // r1
  int result; // r0
  int v5; // [sp+0h] [bp-10h]
  int v6; // [sp+4h] [bp-Ch]

  v5 = 0;
  v1 = -1;
  if ( !(*(int (**)(void))(*(_DWORD *)a1 + 24))() )
  {
    v2 = v5;
    v3 = (*(int (__fastcall **)(int, const char *))(*(_DWORD *)v5 + 24))(
           v5,
           "com/m4399/framework/helpers/AppNativeHelper");
    if ( v3 )
      v1 = ((*(int (__fastcall **)(int, int, char **, signed int))(*(_DWORD *)v2 + 860))(v2, v3, off_20044, 16) >> 31) | 0x10004;
  }
  result = _stack_chk_guard - v6;
  if ( _stack_chk_guard == v6 )
    result = v1;
  return result;
}
```

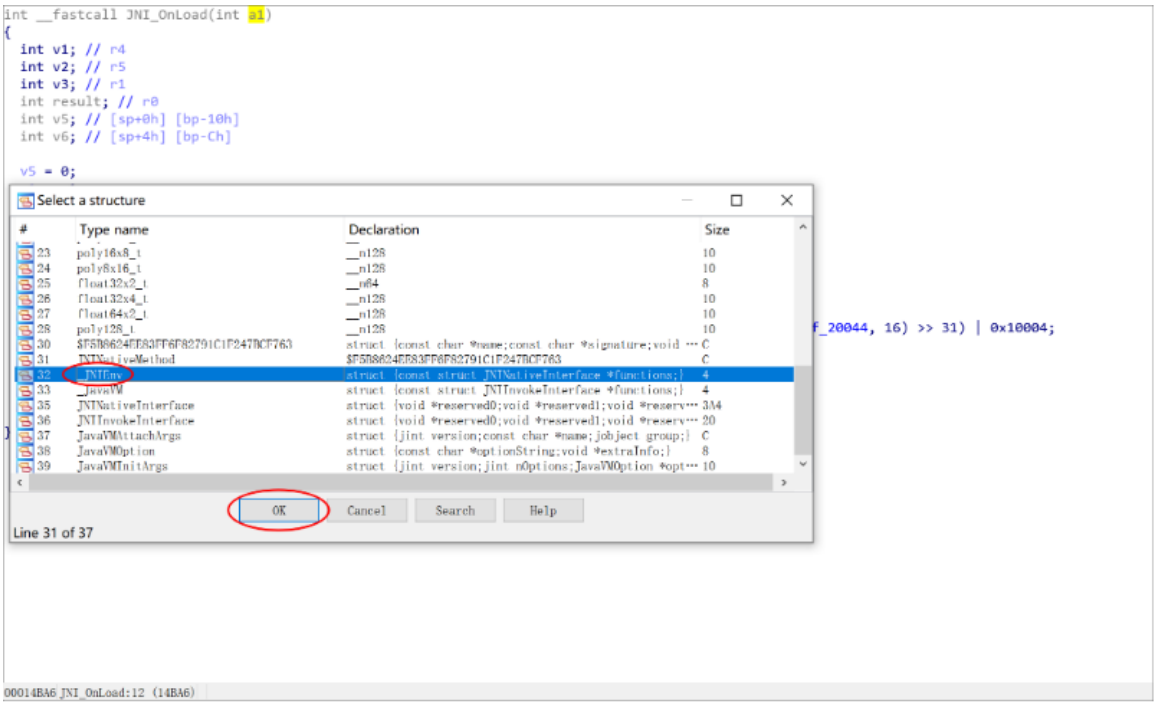
00014B9C JNI_OnLoad:10 (14B9C)

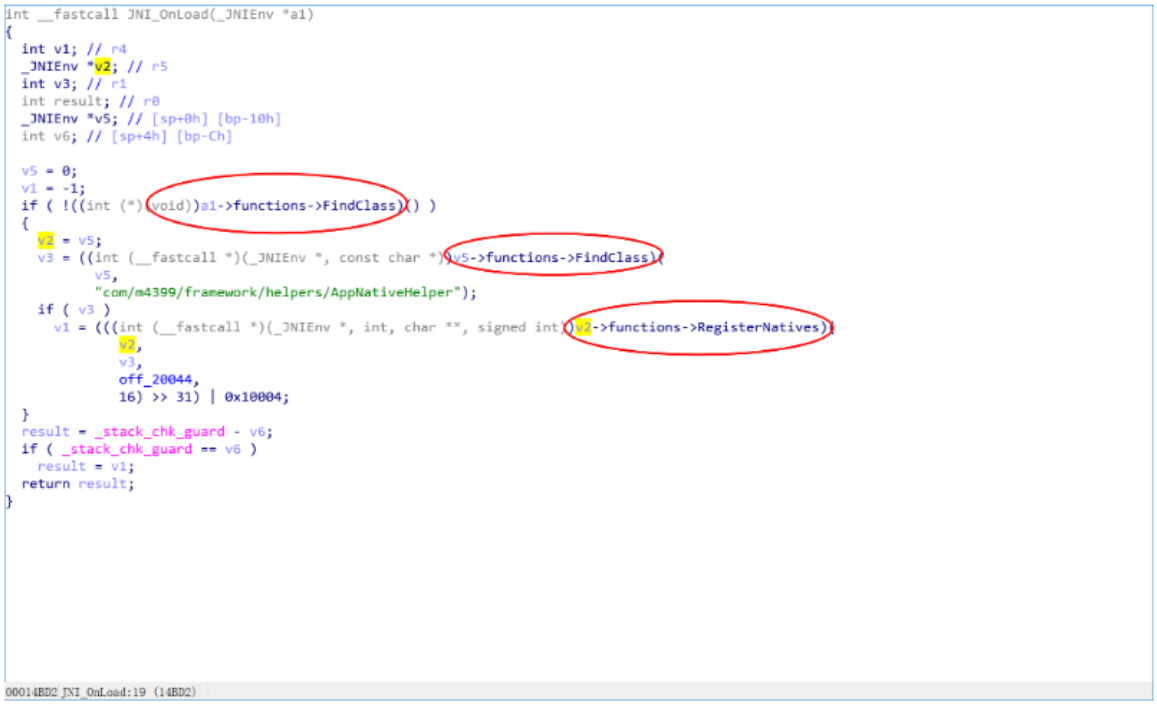
3.之前我们已经知道，凡是*(指针变量+xxx)这种形式都是在使用JNI函数，所以导入Jni.h头文件，在a1,v5,v2等变量上右键如图。

```
int __fastcall JNI_OnLoad(int a1)
{
  int v1; // r4
  int v2; // r5
  int v3; // r1
  int result; // r0
  int v5; // [sp+0h] [bp-10h]
  int v6; // [sp+4h] [bp-Ch]

  v5 = 0;
  v1 = -1;
  if ( !(*(int (**)(void))(*(_DWORD *)a1 + 24))() )
  {
    v2 = v5;
    v3 = (*(int (__fastcall **)(int, c
           v5,
           "com/m4399/framework/helpers
    if ( v3 )
      v1 = ((*(int (__fastcall **)(int,
  }
  result = _stack_chk_guard - v6;
  if ( _stack_chk_guard == v6 )
    result = v1;
  return result;
}
```

Rename lvar N
Set lvar type Y
Convert to struct *
Create new struct type
Jump to xref... X
Edit comment /
Edit block comment Ins
Hide casts \
Font...

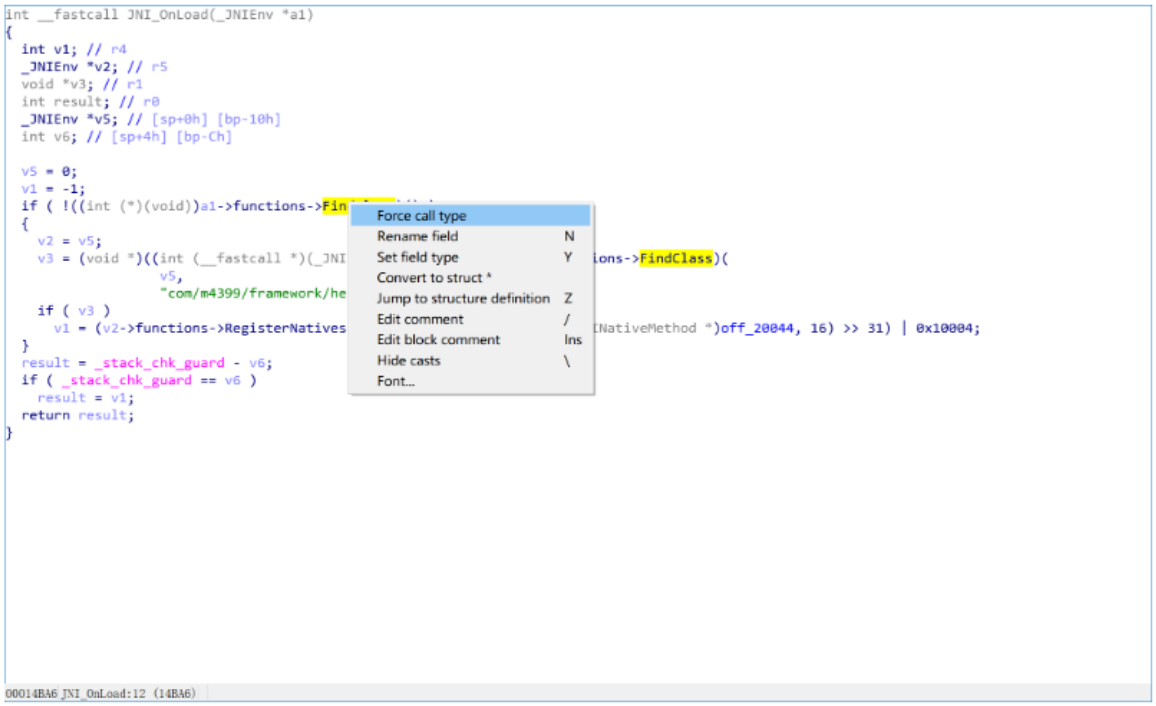
00014BA6 JNI_OnLoad:12 (14BA6)





这个时候JNI函数都正确展示出来，如果大家反编译的是自己的Apk，对照着看源码和反汇编代码，仍然会感觉“不太舒服”，我们还有一些额外的工作可以做。

4.IDA由于不确定参数的数目，常常会不显示函数的参数，用如下的方式强制展示参数（findclass显然不可能无参）。



在几个jni函数上都试一下，结果如下，需要注意的是，自己写的App可能不会有这些问题。

5. 接下来我们隐藏掉类型转换，这样代码会更加可读。

```
int __fastcall JNI_OnLoad(_JNIEnv *a1)
{
  int v1; // r4
  _JNIEnv *v2; // r5
  jclass v3; // r1
  int result; // r0
  _JNIEnv *v5; // [sp+0h] [bp-10h]
  int v6; // [sp+4h] [bp-Ch]

  v5 = 0;
  v1 = -1;
  if ( !a1->functions->FindClass(&a1->functions, &v5) )
  {
    v2 = v5;
    v3 = v5->functions->FindClass(&v5->functions, "com/m4399/framework/helpers/AppNativeHelper");
    if ( v3 )
      v1 = (v2->functions->RegisterNatives(&v2->functions, v3, off_20044, 16) >> 31) | 0x10004;
  }
  result = _stack_chk_guard - v6;
  if ( _stack_chk_guard == v6 )
    result = v1;
  return result;
}
```

00014B90 JNI_OnLoad:17 (14B90)

反编译的工作顺利完成了，接下来找动态注册的函数。

3.2 寻找关键函数

看一下RegisterNatives这个函数的原型。

```
jint RegisterNatives(JNIEnv *env,jclass clazz, const
JNINativeMethod *methods, jnint nMethods);
```

第一个参数是JNIEnv指针，所有的JNI函数第一个参数都是它。

第二个参数jclasss是类对象，通过 JNI FindClass函数得来。

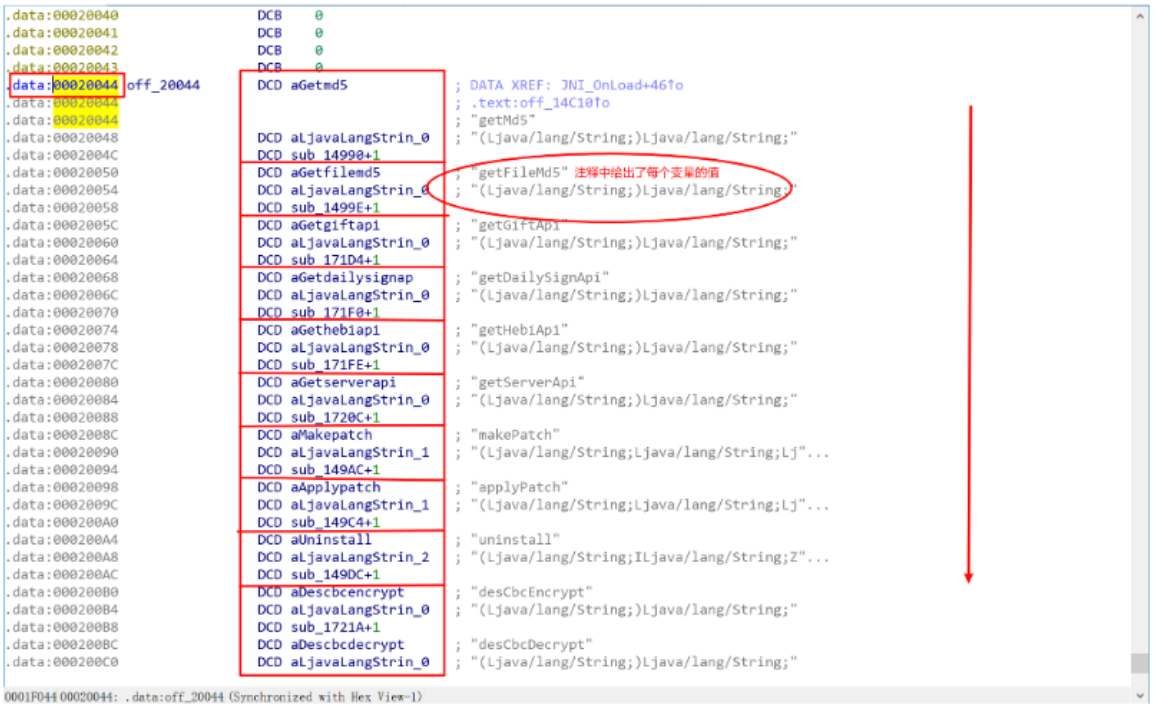
第三个参数是一个数组，数组中包含了若干个结构体，每个结构体存储了Java Native方法到JNI实现方法的映射关系。

第四个参数代表了数组中结构体的数量，或者可以说此次动态注册了多少个native方法。

我们仔细品一下这个结构体，内容为Java层方法名+签名+JNI层对应的函数指针，Java层方法名并不携带包的路径，包的信息由第二个参数，也就是jclass类对象提供。签名的写法和Smali语法类似，想必大家不陌生。JNI层对应的函数指针也似乎没啥问题。

接下来我们阅读一下截图中的RegisterNatives函数，v3即类对象，“com/m4399/……”即Java native函数所声明的类，第四个参数为16，即off_20044这个数组中有十六个结构体，或者说十六组java native函数与jni实现函数的映射。

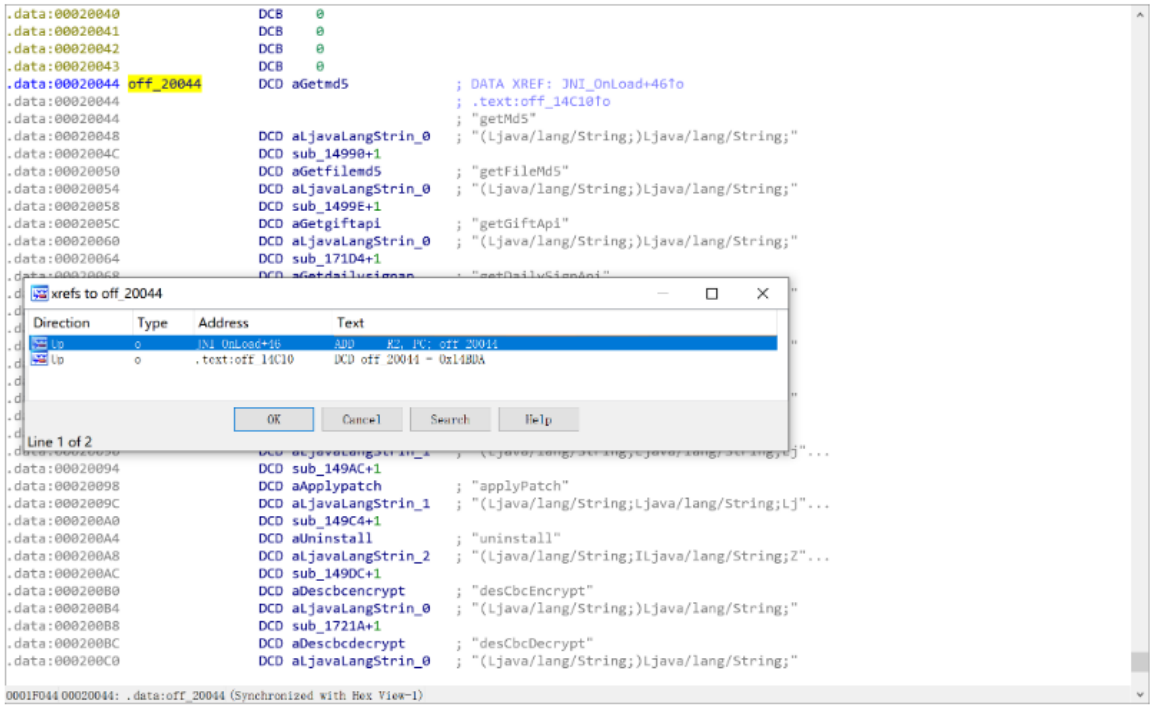
我想你应该不会对off_20044这个命名感到恐慌，这是IDA生成的假名字，详细内容见下表。off_20044即代表了这是一个数据，位于20044这个偏移位置，我们双击进去试试。



data:00020044证实了我们的想法，可以发现，IDA反汇编的效果还不错，我们从上往下划分，每三行代表一个完整的映射。只要两个地方让人不太舒服。

1.第一个结构体为什么占那么多行？

这是因为作为内容的起始部分，IDA会在右方用注释的方式展示它的交叉引用状况，交叉引用占用了正常的两行，JNI_Onload+46以及.textL0ff_14C10这两个位置引用了这份数据，正是交叉引用的注释导致第一个结构体，或者说第一行下面平白空了两行。我们可以在off_20044上按快捷键x查看其交叉引用，验证我们的观点。



2.我们之前说过，每个结构体里三块内容，Java层方法名+签名+JNI层对应的函数指针，而IDA结果正确吗？aGetmd5并不像方法名，aLjavaLangStrin_0也不像正确的签名，第三个sub_xxx，根据我们上表，它代表了一个函数的起点，这倒是和“JNI层对应的函数指针”不谋而合。可是方法名和签名是怎么回事？

这是因为IDA给方法名以及签名二次取了名字。

```
#原代码

a = 3


#IDA反编译后

a1 = 3 #a

a = a1
```

IDA用注释的形式给出了真正的值，因此我们可以直接看右边注释，这结果明显就正确了，除此之外，IDA在命名时会参考原值，因此才会有aLjavaLangStrin_0这种似是而非的名字。

3.3 应用的场景

至此，我们已经搞懂了动态注册，也称函数注册的定位，那么为什么还需要用Hook registernative函数呢？直接用IDA查看一下不就得了？

有多方面的考虑，考虑一下这两个情景

- 找不到某个Native声明的Java函数是哪个SO加载来的。
- IDA反编译时遇到了防护，JNI_Onload无法顺利反编译（常见）。

这个时候Hook动态注册函数就能一把尖刀，直刺So中函数所在的位置。为了理解上更通顺，我们不考虑一步到位，而是一步步去优化Hook代码，希望对大家有所帮助。

```
var RevealNativeMethods = function() {

    // 为了可移植性，选择使用Frida
提供的Process.pointerSize来计算指针所占用内存，也可以直接var pSize
= 4
```

```
    var pSize = Process.pointerSize;
    // 获取当前线程的JNIEnv
    var env = Java.vm.getEnv();

    //
我们所需要Hook的函数是在JNIEnv指针数组的第215位，因为我们这里只是Hook
单个函数，所以没有引入包含全体JNI函数的数组
    var RegisterNatives = 215;

    // 将通过位置计算函数地址这一步骤封装为函数
    function getNativeAddress(idx) {
        var nativrAddress = env.handle.readPointer().add(idx *
pSize).readPointer();
        console.log("nativrAddress:"+nativrAddress);
        return nativrAddress;
    }

    // 开始Hook
   Interceptor.attach(getNativeAddress(RegisterNatives), {
      onEnter: function(args) {
          console.log("Already enter getNativeAddress
Function!");
          //
遍历数组中每一个结构体，需要注意的是，参数4即代表了结构体数量，我们这里
使用了它
```

```
            for (var i = 0, nMethods = parseInt(args[3]); i < nMethods; i++) {
                var methodsPtr = ptr(args[2]);
                var structSize = pSize * 3;
                var methodName = methodsPtr.add(i * structSize).readPointer();
                var signature = methodsPtr.add(i * structSize + pSize).readPointer();
                var fnPtr = methodsPtr.add(i * structSize + (pSize * 2)).readPointer();
                /*
                    typedef struct {
                       const char* name;
                       const char* signature;
                       void* fnPtr;
                    } JNINativeMethod;
                */

                var ret = {
                    //methodName与signature都是字符串，readCString和readUtf8String是Frida提供的两个字符串解析函数，
                    //前者会先尝试用utf8的方式，不行再打印unicode编码，因此相比readUtf8String是更保险和优雅的选择
                    methodName:methodName.readCString(),
```

```
                signature:signature.readCString(),
                address:fnPtr,
            };

            // 使用JSON.stringfy()打印内容通常是好的选择
            console.log(JSON.stringify(ret))

        }
    }
  });
};

Java.perform(RevealNativeMethods);
```

由于registerNatives发生的时机往往很早，建议采用Spawn方式注入，否则可能毫无收获。

3.3.1 代码优化

似乎很不错的样子，但是自己看一下内容，却不大如人意。

Hook输出了Java方法名，但我们之前说过，Java层方法名并不携带包的路径，包的信息由第二个参数，所以方法名提供不了什么信息，第二个信息是参数签名，和我们预期一致，第三个信息是函数地址，有一个很大的问题，输出的地址是内存中的真正地址，而我们分析SO时需要用到IDA，IDA 加载模块的时候，会以基址 0 加载分析 so 模块，但是 SO 运行在 Android 上的时候，每次的加载地址不是固定的，有没有办法解决这个问题呢？

办法是很多的，我们查看Frida官方文档可以发现，Frida提供了两个根据地址得到所在SO文件等信息的函数。

我们对照一下结果，修改代码输出如下：

```
var ret = {
```

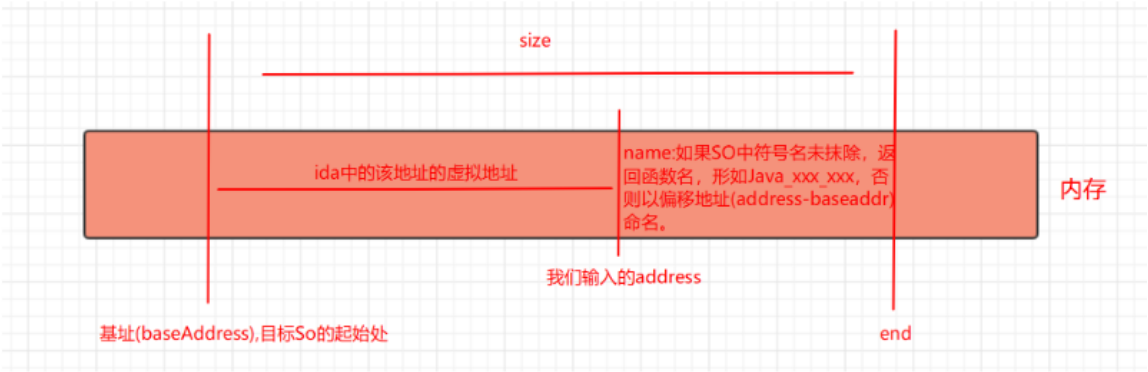
```
    //
methodName与signature都是字符串，readCString和readUtf8String是Frida提供的两个字符串解析函数，
    //
前者会先尝试用utf8的方式，不行再打印unicode编码，因此相比readUtf8String是更保险和优雅的选择
    // 只需要新增如下两行代码
    module1: DebugSymbol.fromAddress(fnPtr),
    module2: Process.findModuleByAddress(fnPtr),
    methodName:methodName.readCString(),
    signature:signature.readCString(),
    address:fnPtr,
};
```

查看任意一条输出结果，此Native方法名为tokenDecrypt


```
{"module1":{"address":"0x8a339267","name":"0x17267","moduleName":"libm4399.so","fileName":"","lineNumber":0},
"module2":{"name":"libm4399.so","base":"0x8a322000","size":135168,"path":"/data/app/com.m4399.gamecenter-1/lib/arm/libm4399.so"},
"methodName":"tokenDecrypt",
"signature":"(Ljava/lang/String;)Ljava/lang/String;",
"address":"0x8a339267"}
```


可以发现，两个API侧重点不同，地址为0x8a339267，函数1返回自身地址，符号名（0x17267），所属SO名，具体文件名和行数（这两个字段似乎无效），符号名name可能有些不理解，我们待会儿再讲。函数2返回所属SO，base字段，即为基址，表示此SO在内存中起

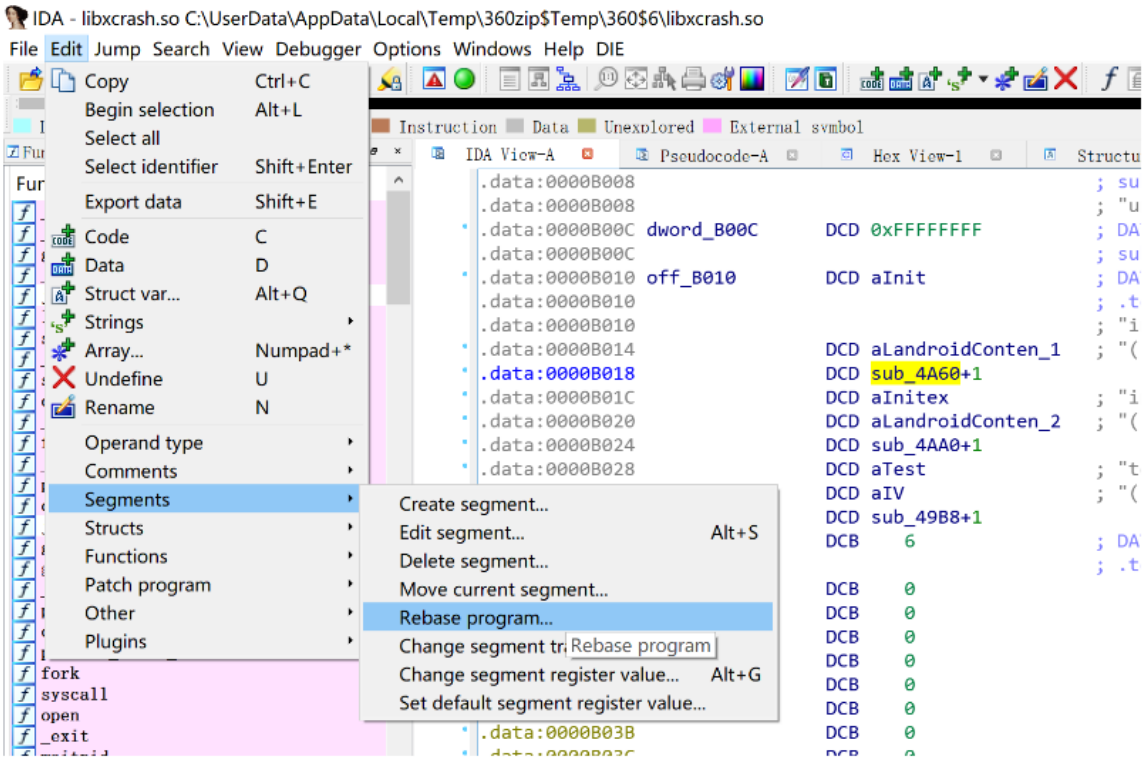
始的位置，size字段代表了SO的大小，path即为SO在手机中的真实路径。

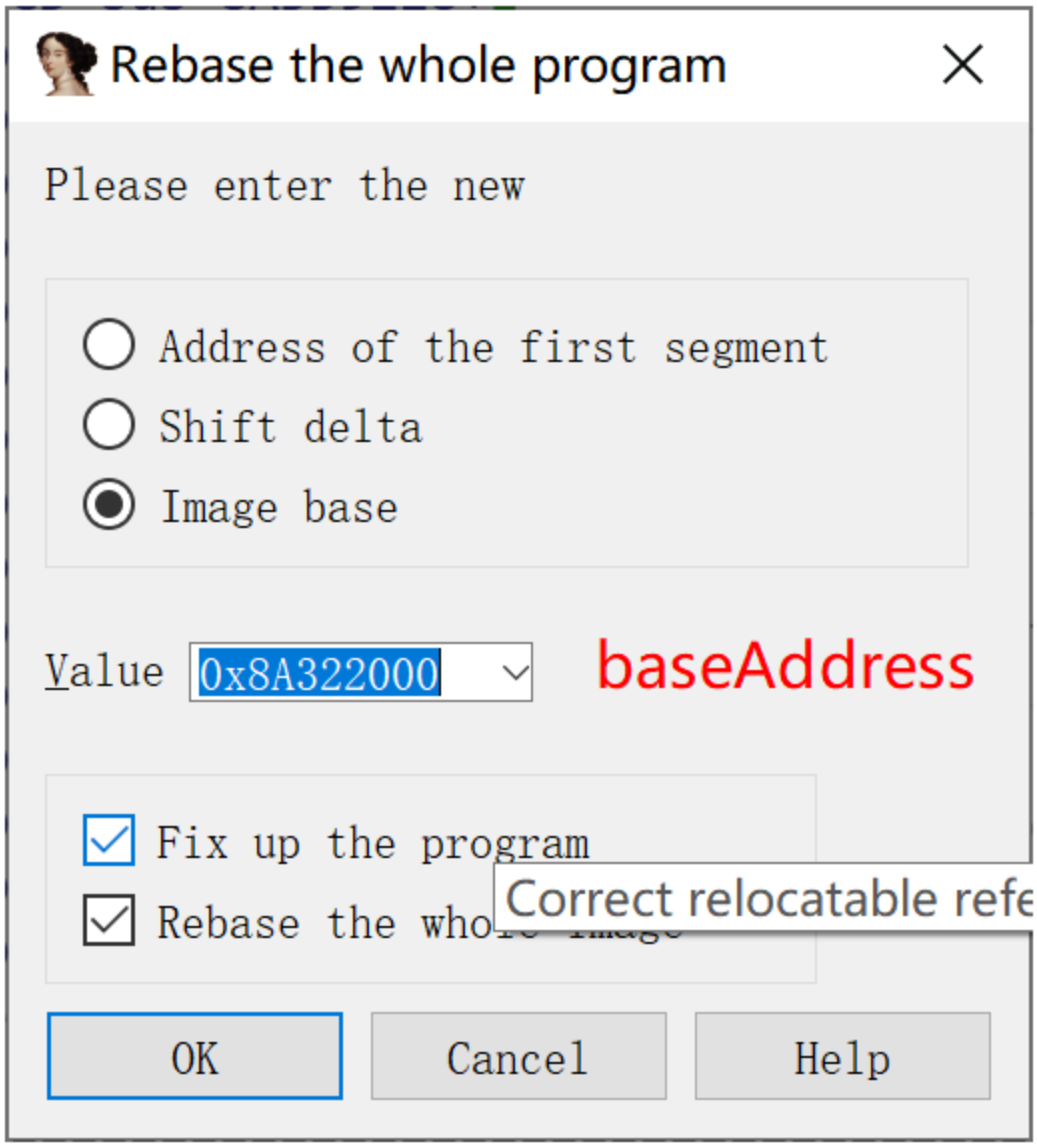


图中可以看出，如果想得到IDA中的虚拟地址，两个函数都可以做到。使用函数一的name字段，或者address减去函数二提供给我们的So基址。我们先通过IDA来验证tokenDecrypt这个函数结果是否准确。0x17266+1即0x17267,name字段被验证。0x8a339267-0x8a322000=0x17267,两种方法都OK。

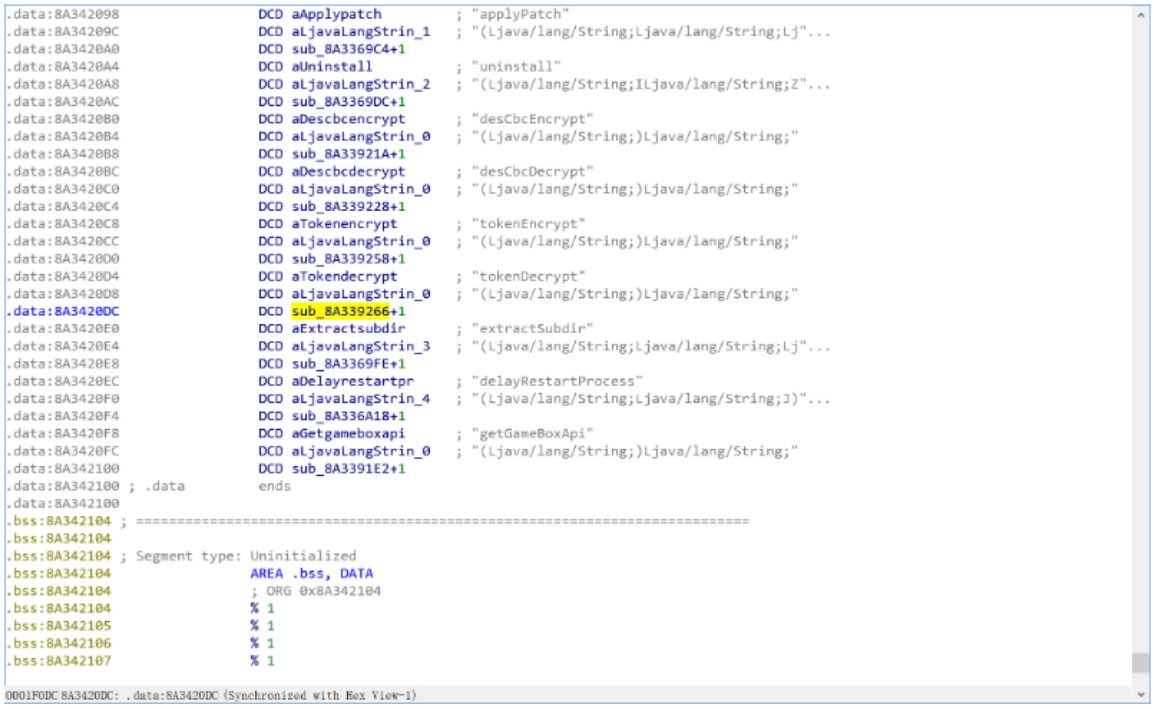
```
.data:00020080                 DCD aGetserverapi       ; "getServerApi"
.data:00020084                 DCD aLjavaLangStrin_0   ; "(Ljava/lang/String;)Ljava/lang/String;"
.data:00020088                 DCD sub_1720C+1
.data:0002008C                 DCD aMakepatch          ; "makePatch"
.data:00020090                 DCD aLjavaLangStrin_1   ; "(Ljava/lang/String;Ljava/lang/String;Lj"...
.data:00020094                 DCD sub_149AC+1
.data:00020098                 DCD aApplypatch         ; "applyPatch"
.data:0002009C                 DCD aLjavaLangStrin_1   ; "(Ljava/lang/String;Ljava/lang/String;Lj"...
.data:000200A0                 DCD sub_149C4+1
.data:000200A4                 DCD aUninstall          ; "uninstall"
.data:000200A8                 DCD aLjavaLangStrin_2   ; "(Ljava/lang/String;ILjava/lang/String;Z"...
.data:000200AC                 DCD sub_149DC+1
.data:000200B0                 DCD aDescbcencrypt      ; "desCbcEncrypt"
.data:000200B4                 DCD aLjavaLangStrin_0   ; "(Ljava/lang/String;)Ljava/lang/String;"
.data:000200B8                 DCD sub_1721A+1
.data:000200BC                 DCD aDescbcdecrypt      ; "desCbcDecrypt"
.data:000200C0                 DCD aLjavaLangStrin_0   ; "(Ljava/lang/String;)Ljava/lang/String;"
.data:000200C4                 DCD sub_17228+1
.data:000200C8                 DCD aTokenencrypt       ; "tokenEncrypt"
.data:000200CC                 DCD aLjavaLangStrin_0   ; "(Ljava/lang/String;)Ljava/lang/String;"
.data:000200D0                 DCD sub_17258+1
.data:000200D4                 DCD aTokendecrypt       ; "tokenDecrypt"
.data:000200D8                 DCD aLjavaLangStrin_0   ; "(Ljava/lang/String;)Ljava/lang/String;"
.data:000200DC                 DCD sub_17266+1
.data:000200E0                 DCD aExtractsubdir      ; "extractSubdir"
.data:000200E4                 DCD aLjavaLangStrin_3   ; "(Ljava/lang/String;Ljava/lang/String;Lj"...
.data:000200E8                 DCD sub_149FE+1
.data:000200EC                 DCD aDelayrestartpr     ; "delayRestartProcess"
.data:000200F0                 DCD aLjavaLangStrin_4   ; "(Ljava/lang/String;Ljava/lang/String;J)"...
.data:000200F4                 DCD sub_14A18+1
.data:000200F8                 DCD aGetgameboxapi      ; "getGameBoxApi"
.data:000200FC                 DCD aLjavaLangStrin_0   ; "(Ljava/lang/String;)Ljava/lang/String;"
.data:00020100                 DCD sub_171E2+1
.data:00020100 ; .data         ends
.data:00020100
.bss:00020104 ; ===========================================================================
.bss:00020104
.bss:00020104 ; Segment type: Uninitialized
0001F0EC 000200EC: .data:000200EC (Synchronized with Hex View-1)
```

通过Frida提供的Api，我们得到了地址对应的SO文件以及它在IDA中的位置，这真是可喜的事儿。除此之外，我们补充另外一种方式来定义地址，即修改IDA中SO的基址。




Rebase the whole program
Please enter the new
Address of the first segment
Shift delta
Image base
Value
0x8A322000
baseAddress
Fix up the program
Correct relocatable refe
OK
Cancel
Help

效果如下：



在我们这个场景下，这样处理并不方便，但在IDA动态调试时，通过Rebease 基址，让其与运行时 so 的基址相同，可以极大的方便静态分析。

需要注意的是，我们使用此Hook脚本时，目的不是印证IDA中反编译的地址和Frida hook得到的地址是否相同，而是为了定位。IDA中使用快捷键G可以迅速进行地址跳转。

接下来我们需要进一步优化脚本，参数2是jclass对象，可以让我们获得这个方法所在类的信息，它是JNI方法Findclass的结果，因此我们要Hook这个JNI方法。Findclass的结果需要和对应的RegisterNative函数匹配，这涉及到JNIEnv线程的问题，我们使用集合的方式处理。来看一下完整的代码吧。

```
var RevealNativeMethods = function() {
```

```
    // 为了移植性，选择使用Frida
API来计算指针所占用内存，也可以直接var pSize  = 4
    var pSize = Process.pointerSize;
    // 获取当前线程的JNIEnv
    var env = Java.vm.getEnv();
    // 我们所需要Hook的函数是在JNIEnv指针数组的第6和第215位
    var RegisterNatives = 215;
    var FindClassIndex = 6;

    // 将通过位置计算函数地址这一步骤封装为函数
    function getNativeAddress(idx) {
        var nativrAddress = env.handle.readPointer().add(idx *
pSize).readPointer();
        return nativrAddress;
    }

    // 初始化集合，用于处理两个JNI函数之间的同步关系
    var jclassAddress2NameMap = {};

    // Hook 两个JNI函数
    Interceptor.attach(getNativeAddress(FindClassIndex), {
        onEnter: function (args) {
            // 设置一个集合，不同的JNIEnv线程对应不同的class
            jclassAddress2NameMap[args[0]] =
args[1].readCString();
```

```
        }
    });


    Interceptor.attach(getNativeAddress(RegisterNatives), {
        onEnter: function(args) {
            console.log("Already enter getNativeAddress
Function!");
            //
遍历数组中每一个结构体，需要注意的是，参数4即代表了结构体数量，我们这里
使用了它
            for (var i = 0, nMethods = parseInt(args[3]); i <
nMethods; i++) {
                var methodsPtr = ptr(args[2]);
                var structSize = pSize * 3;
                var methodName = methodsPtr.add(i *
structSize).readPointer();
                var signature = methodsPtr.add(i * structSize +
pSize).readPointer();
                var fnPtr = methodsPtr.add(i * structSize +
(pSize * 2)).readPointer();
                /*
                typedef struct {
                const char* name;
                const char* signature;
                void* fnPtr;
                } JNINativeMethod;
```

```
                */
                var ret = {
                    //
methodName与signature都是字符串，readCString和readUtf8String是Frid
a提供的两个字符串解析函数，
                    //
前者会先尝试用utf8的方式，不行再打印unicode编码，因此相比readUtf8Stri
ng是更保险和优雅的选择
                    moduleName:
DebugSymbol.fromAddress(fnPtr)["moduleName"],
                    jClass:jclassAddress2NameMap[args[0]],
                    methodName:methodName.readCString(),
                    signature:signature.readCString(),
                    address:fnPtr,
                    IdaAddress:
DebugSymbol.fromAddress(fnPtr)["name"],
                };
                // 使用JSON.stringfy()打印内容通常是好的选择
                console.log(JSON.stringify(ret))
            }
        }
    });
};
Java.perform(RevealNativeMethods);
```

知其黑 守其白

分享知识盛宴，闲聊大院趣事，备好酒肉等你

长按二维码关注 酒仙桥六号部队

精选留言

用户设置不下载评论