

一、前言

这是《Unidbg模拟执行与算法分析实战》的样本一，资料和视频见知识星球。

- 将Unidbg 作为最主要的分析工具、IDA、Frida等作为辅助。
- 深度、全面的展示Unidbg模拟执行时补充环境的各种技巧、经验和暗坑。
- 深度、全面的展示Unidbg在算法还原上强大的能力
- 掌握使用Unidbg分析SO样本的核心思路和全套方法

这个样本，我们着重阐述Unidbg分析样本的思路。

二、准备

首先看一下我们分析的方法



这是我们的目标函数，从图上我们可以看出

- 1.System.loadLibrary("scmain") 即这个函数的实现位于libscmain.so中
- 2.simpleSign的入参有两个，一个字节数组和一个字符串，返回的结果也是字符串

除此之外，它是动态绑定还是静态绑定等等，我们一概不知。

三、Frida 预分析

目标函数的参数和返回值长什么样？

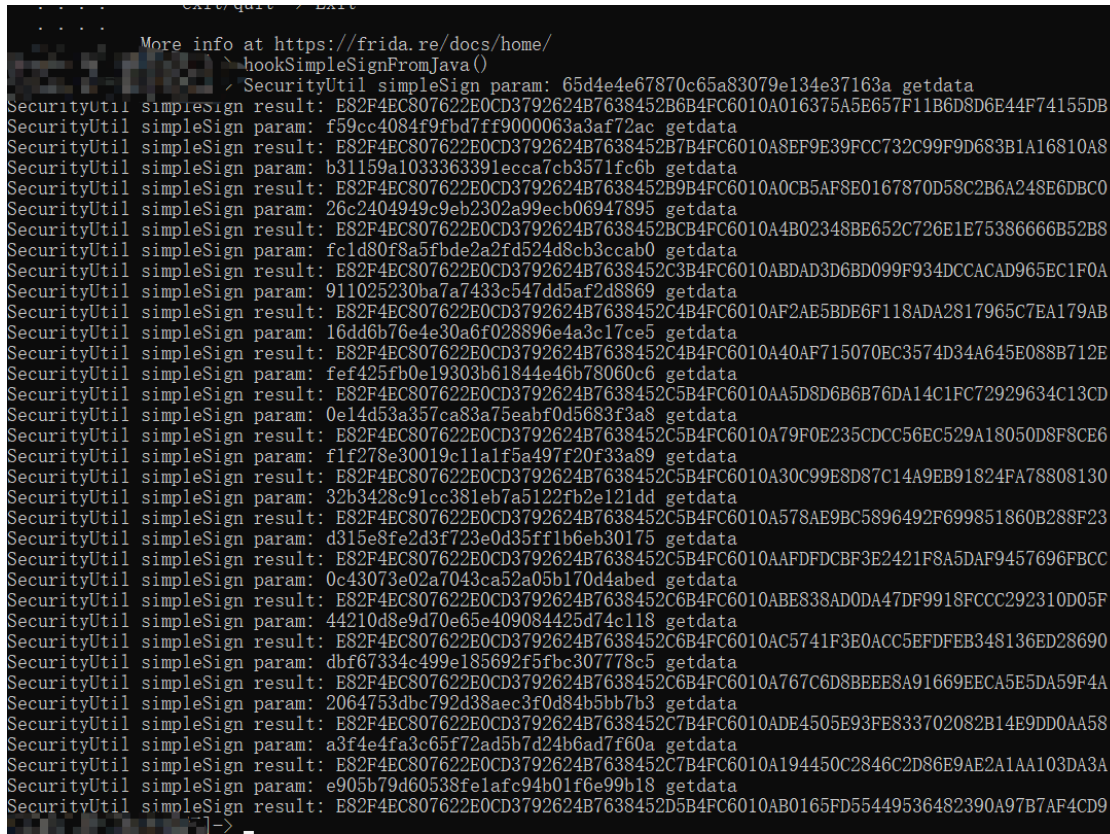
```
function hookSimpleSignFromJava(){
    Java.perform(function(){
        var securityUtil = Java.use("ctrip.android.security.SecurityUtil");
        securityUtil.simpleSign.implementation = function(byteArr, type){
            var StringCls = Java.use("java.lang.String");
```

```

        var stringVal = StringCls.$new(byteArr,"utf-8");

        console.log("SecurityUtil simpleSign param:", stringVal, type);
        var ret = this.simpleSign(byteArr, type);
        console.log("SecurityUtil simpleSign result:", ret);
        return ret;
    }
}
}
}

```



```

More info at https://frida.re/docs/home/
hookSimpleSignFromJava()
SecurityUtil simpleSign param: 65d4e4e67870c65a83079e134e37163a getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452B6B4FC6010A016375A5E657F11B6D8D6E44F74155DB
SecurityUtil simpleSign param: f59cc4084f9fbd7ff9000063a3af72ac getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452B7B4FC6010A8EF9E39FCC732C99F9D683B1A16810A8
SecurityUtil simpleSign param: b31159a1033363391ecca7cb3571fc6b getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452B9B4FC6010A0CB5AF8E0167870D58C2B6A248E6DBC0
SecurityUtil simpleSign param: 26c2404949c9eb2302a99ecb06947895 getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452BCB4FC6010A4B02348BE652C726E1E75386666B52B8
SecurityUtil simpleSign param: fcd80f8a5fbd2a2fd524d8b3ccab0 getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452C3B4FC6010ABDAD3D6BD099F934DCCACAD965EC1F0A
SecurityUtil simpleSign param: 911025230ba7a7433c547dd5af2d8869 getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452C4B4FC6010AF2AE5BDE6F118ADA2817965C7EA179AB
SecurityUtil simpleSign param: 16dd6b76e4e30a6f028896e4a3c17ce5 getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452C4B4FC6010A40AF715070EC3574D34A645E088B712E
SecurityUtil simpleSign param: fef425fb0e19303b61844e46b78060c6 getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452C5B4FC6010AA5D8D6B6B76DA14C1FC72929634C13CD
SecurityUtil simpleSign param: 0e14d53a357ca83a75eabf0d5683f3a8 getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452C5B4FC6010A79F0E235CDC56EC529A18050D8F8CE6
SecurityUtil simpleSign param: flf278e30019c11a1f5a497f20f33a89 getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452C5B4FC6010A30C99E8D87C14A9EB91824FA78808130
SecurityUtil simpleSign param: 32b3428c91cc381eb7a5122fb2e121dd getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452C5B4FC6010A578AE9BC5896492F699851860B288F23
SecurityUtil simpleSign param: d315e8fe2d3f723e0d35ff1b6eb30175 getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452C5B4FC6010A0AFDFDCBF3E2421F8A5DAF9457696FBCC
SecurityUtil simpleSign param: 0c43073e02a7043ca52a05b170d4abed getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452C6B4FC6010ABE838AD0DA47DF9918FCCC292310D05F
SecurityUtil simpleSign param: 44210d8e9d70e65e409084425d74c118 getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452C6B4FC6010AC5741F3E0ACC5EFD0FEB348136ED28690
SecurityUtil simpleSign param: dbf67334c499e185692f5fbc307778c5 getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452C6B4FC6010A767C6D8BEE8A91669EECA5E5DA59F4A
SecurityUtil simpleSign param: 2064753dbc792d38aec3f0d84b5bb7b3 getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452C7B4FC6010ADE4505E93FE833702082B1E49DD0AA58
SecurityUtil simpleSign param: a3f4e4fa3c65f72ad5b7d24b6ad7f60a getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452C7B4FC6010A194450C2846C2D86E9AE2A1AA103DA3A
SecurityUtil simpleSign param: e905b79d60538fela94b01f6e99b18 getdata
SecurityUtil simpleSign result: E82F4EC807622E0CD3792624B7638452D5B4FC6010AB0165FD55449536482390A97B7AF4CD9

```

参数1字节数组，我们转成了字符串展打印，内容似乎是32位十六进制数

参数2是固定的getdata字符串

返回值一直在变化，但始终定长，且字母是大写。

我们获得了对目标函数最基本的了解，接下来进入Unidbg的世界吧！

四、Unidbg小试

实诚的说，Unidbg的基本使用并不难，看看Unidbg的测试样例就能明白七七八八，但Unidbg的使用门槛不在此，样本和Unidbg之间摩擦出的火花才是最迷人的。或者说人话——“他妈的Unidbg怎么又报错了，我该怎么办？”

我们先加载SO，并执行其JNIOnload

```

package com.lesson1;

import com.github.unidbg.AndroidEmulator;
import com.github.unidbg.linux.android.AndroidEmulatorBuilder;
import com.github.unidbg.linux.android.AndroidResolver;

```

```

import com.github.unidbg.linux.android.dvm.AbstractJni;
import com.github.unidbg.linux.android.dvm.DalvikModule;
import com.github.unidbg.linux.android.dvm.VM;
import com.github.unidbg.memory.Memory;

import java.io.File;

public class xc extends AbstractJni {
    private final AndroidEmulator emulator;
    private final VM vm;

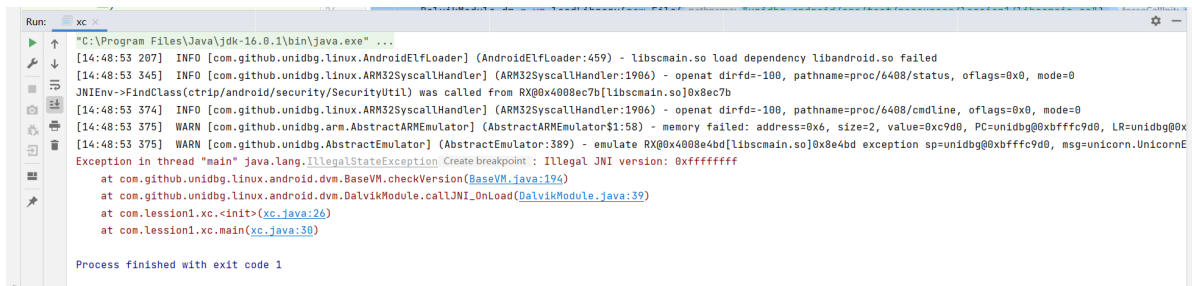
    xc() {
        emulator = AndroidEmulatorBuilder.for32Bit().build(); // 创建模拟器实例，要
        模拟32位或者64位，在这里区分
        final Memory memory = emulator.getMemory(); // 模拟器的内存操作接口
        memory.setLibraryResolver(new AndroidResolver(23)); // 设置系统类库解析

        vm = emulator.createDalvikVM(new File("unidbg-
        android/src/test/resources/lesson1/xc 8-38-2.apk"));
        vm.setVerbose(true); // 设置是否打印jni调用细节
        DalvikModule dm = vm.loadLibrary(new File("unidbg-
        android/src/test/resources/lesson1/libscmain.so"), true);
        vm.setJni(this);
        dm.callJNI_OnLoad(emulator);
    }

    public static void main(String[] args) {
        xc test = new xc();
    }
}

```

运行测试



```

Run: xc
"C:\Program Files\Java\jdk-16.0.1\bin\java.exe" ...
[14:48:53 207] INFO [com.github.unidbg.linux.AndroidElfLoader] (AndroidElfLoader:459) - libscmain.so load dependency libandroid.so failed
[14:48:53 345] INFO [com.github.unidbg.linux.ARM32SyscallHandler] (ARM32SyscallHandler:1906) - openat dirfd=-100, pathname=proc/6408/status, oflags=0x0, mode=0
JNIEnv->FindClass(ctrip/android/security/SecurityUtil) was called from RX00x4008ec7b[libscmain.so]0x8ec7b
[14:48:53 374] INFO [com.github.unidbg.linux.ARM32SyscallHandler] (ARM32SyscallHandler:1906) - openat dirfd=-100, pathname=proc/6408/cmdline, oflags=0x0, mode=0
[14:48:53 375] WARN [com.github.unidbg.arm.AbstractARMEulator] (AbstractARMEulator$1:58) - memory failed: address=0x6, size=2, value=0xc9d0, PC=unidbg@0xbfffc9d0, LR=unidbg@0x
[14:48:53 375] WARN [com.github.unidbg.AbstractEmulator] (AbstractEmulator:389) - emulate RX00x4008e4bd[libscmain.so]0x8e4bd exception sp=unidbg@0xbfffc9d0, msg=unicorn.UnicornE
Exception in thread "main" java.lang.IllegalStateException: Create breakpoint : Illegal JNI version: 0xffffffff
    at com.github.unidbg.linux.android.dvm.BaseVM.checkVersion(BaseVM.java:194)
    at com.github.unidbg.linux.android.dvm.DalvikModule.callJNI_OnLoad(DalvikModule.java:39)
    at com.lesson1.xc.<init>(xc.java:26)
    at com.lesson1.xc.main(xc.java:30)
Process finished with exit code 1

```

可以发现报错了

我们从头开始看

第一行

```

[14:48:53 207] INFO [com.github.unidbg.linux.AndroidElfLoader]
(AndroidElfLoader:459) - libscmain.so load dependency libandroid.so failed

```

我们翻译一下：libscmain.so（即目标SO）没能加载其依赖 libandroid.so

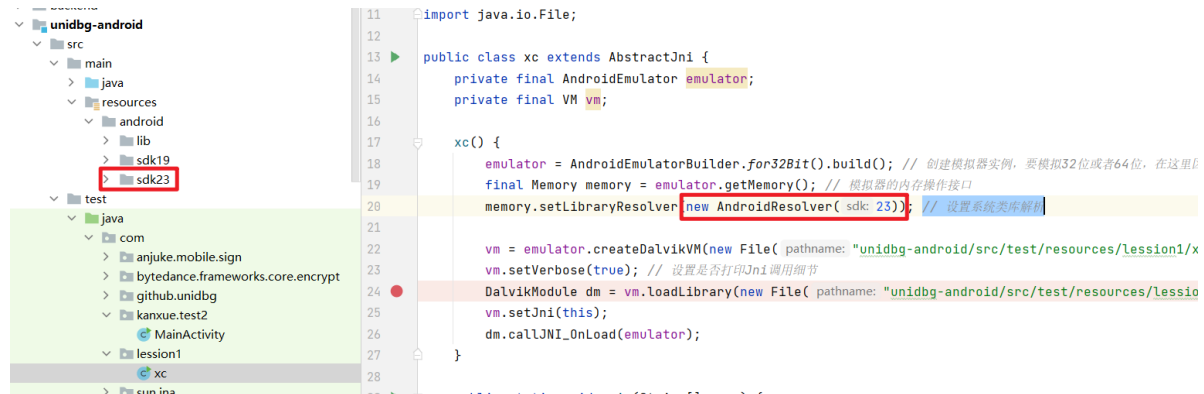
首先，每个SO文件都有一些依赖的SO


```
Run: xc
"C:\Program Files\Java\jdk-16.0.1\bin\java.exe" ...
load :libscmain.so
load :liblog.so
load :libc++.so
load :libdl.so
load :libc.so
load :libm.so
[08:25:53 966] INFO [com.github.unidbg.linux.AndroidElfLoader] (AndroidElfLoader:460) - libscmain.so load dependency libandroid.so failed
load :libstdc++.so
[08:25:54 106] INFO [com.github.unidbg.linux.ARM32SyscallHandler] (ARM32SyscallHandler:1906) - openat dirfd=-100, pathname=proc/23220/status, oflags=0x0, mode=0
```

可以发现，Unidbg加载了刚才所述的依赖SO，但libandroid.so不在其列。

这是为什么呢？

因为Unidbg暂不支持这个SO的加载



```
import java.io.File;

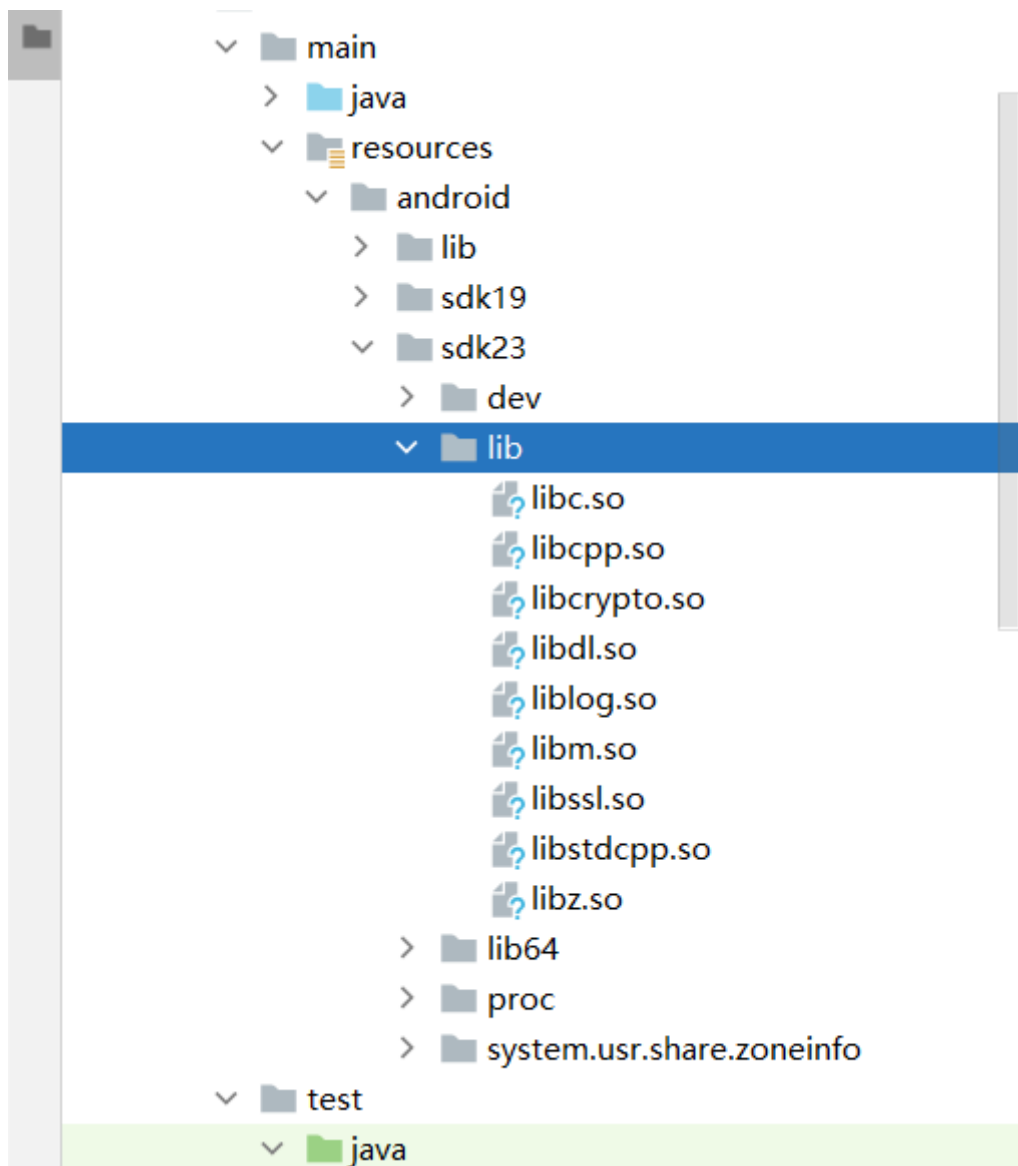
public class xc extends AbstractJni {
    private final AndroidEmulator emulator;
    private final VM vm;

    xc() {
        emulator = AndroidEmulatorBuilder.for32Bit().build(); // 创建模拟器实例，要模拟32位或者64位，在这里
        final Memory memory = emulator.getMemory(); // 模拟器的内存操作接口
        memory.setLibraryResolver(new AndroidResolver(sdk: 23)); // 设置系统类库

        vm = emulator.createDalvikVM(new File(pathname: "unidbg-android/src/test/resources/lesson1/x
        vm.setVerbose(true); // 设置是否打印Jni调用细节
        DalvikModule dm = vm.loadLibrary(new File(pathname: "unidbg-android/src/test/resources/lessio
        vm.setJni(this);
        dm.callJNI_OnLoad(emulator);
    }
}
```

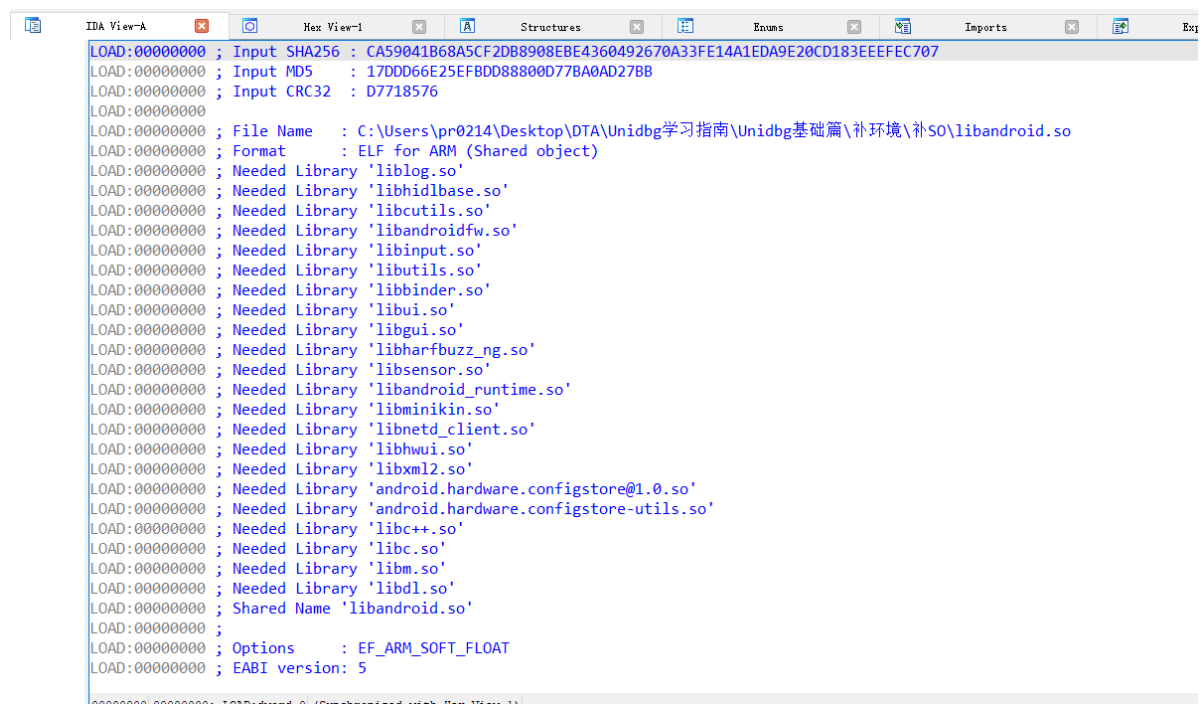
我们选择的“NDK版本”是23，即Android 6.0

看一下Unidbg为我们准备了哪些可加载的系统库



可以发现，libandroid.so不在其列，这是为什么呢？为什么Unidbg不天然支持libandroid.so这个系统SO的装载呢？

我们从手机中pull出libandroid.so，在IDA中看一眼就知道了



我们可以看到它的依赖SO极其众多，想把libandroid.so完完整整的加载进内存，恐怕绝不是一件轻松的事儿，因此libandroid.so并不是Unidbg默认支持加载的SO，这也就解释的通了。

那么该怎么解决这个问题呢？我们的libscmain.so依赖libandroid.so呀

乐观的想，SO依赖libandroid.so，但我们的目标函数里不一定用到libandroid.so里的函数，那么libandroid.so加载失败就和我们啥关系都没有。

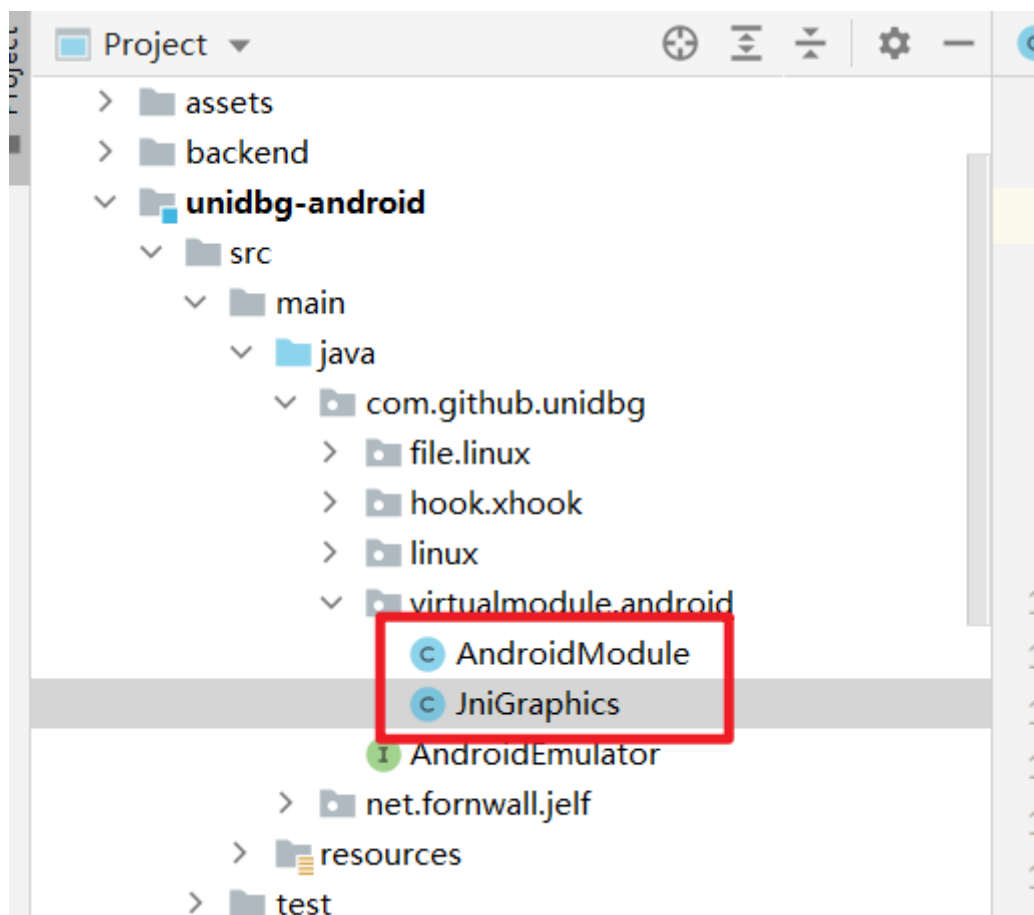
但是把主动权交给样本本身，这肯定是不太妥的做法。那如果假定目标函数中用到了libandroid.so中的函数，那怎么办呢？

Unidbg提供了两种方案

一种是hook，我们可以在libscmain中hook libandroid.so的函数，或者不管三七二十一，直接把那个SO加载进来，然后hook 其中的各种函数，反正就是Hook，然后自己实现这些个函数的逻辑，给予正确的返回值。

另一种方法是使用Unidbg提供的VirtualModule机制，创建一个虚拟SO，手动实现其中的函数，然后加载进内存。（底层应该也用到了hook 回调）

Unidbg提供了两个VirtualModule的示例



分别是libandroid.so和libJniGraphics.so，这正好解决了我们的燃眉之急，来看一下吧


```

19
20 public class AndroidModule extends VirtualModule<VM> {
21
22     private static final Log LOG = LogFactory.getLog(AndroidModule.class);
23
24     public AndroidModule(Emulator<?> emulator, VM vm) { super(emulator, vm, name: "libandroid.so"); }
25
26
27
28     @Override
29     protected void onInitialize(Emulator<?> emulator, final VM vm, Map<String, UnidbgPointer> symbols) {
30         boolean is64Bit = emulator.is64Bit();
31         SvcMemory svcMemory = emulator.getSvcMemory();
32         symbols.put("AAssetManager_fromJava", svcMemory.registerSvc(is64Bit ? (Arm64Svc) (emulator) -> {
33             return fromJava(emulator, vm);
34         }) : (ArmSvc) (emulator) -> { return fromJava(emulator, vm); }));
35         symbols.put("AAssetManager_open", svcMemory.registerSvc(is64Bit ? (Arm64Svc) (emulator) -> {
36             return open(emulator, vm);
37         }) : (ArmSvc) (emulator) -> { return open(emulator, vm); }));
38         symbols.put("AAsset_close", svcMemory.registerSvc(is64Bit ? (Arm64Svc) (emulator) -> {
39             return close(emulator, vm);
40         }) : (ArmSvc) (emulator) -> { return close(emulator, vm); }));
41         symbols.put("AAsset_getBuffer", svcMemory.registerSvc(is64Bit ? (Arm64Svc) (emulator) -> {
42             return getBuffer(emulator, vm);
43         }) : (ArmSvc) (emulator) -> { return getBuffer(emulator, vm); }));
44         symbols.put("AAsset_getLength", svcMemory.registerSvc(is64Bit ? (Arm64Svc) (emulator) -> {
45             return getLength(emulator, vm);
46         }) : (ArmSvc) (emulator) -> { return getLength(emulator, vm); }));
47         symbols.put("AAsset_read", svcMemory.registerSvc(is64Bit ? (Arm64Svc) (emulator) -> {
48             throw new BackendException();
49         }) : (ArmSvc) (emulator) -> { return read(emulator, vm); }));
50     }
51 }

```

可以发现，狸猫换太子，狸猫实现了libandroid.so中最常用的六个函数

Address	Ordinal	Name	Library
001E5764		AAsset_read	
001E5768		AAsset_close	
001E5778		AAssetManager_fromJava	
001E577C		AAssetManager_open	
001E5780		AAsset_getLength	

我们样本里需要用到的都在里面了，真是好事一件，

如果样本还依赖一些别的虚拟模块没有的函数，我们还需要照着android源码，在虚拟模块中照着模子也实现其对应逻辑。

现在我们已经知道了问题的原因以及Unidbg提供的解决方案，那怎么应用这个解决方案呢？很简单，只需要一行代码，将这个虚拟内存注册或者说加载进内存

```

14 public class xc extends AbstractJni {
15     private final AndroidEmulator emulator;
16     private final VM vm;
17
18     xc() {
19         emulator = AndroidEmulatorBuilder.for32Bit().build(); // 创建模拟器实例，要模拟32位或者64位，在这里区分
20         final Memory memory = emulator.getMemory(); // 模拟器的内存操作接口
21         memory.setLibraryResolver(new AndroidResolver( sdk: 23)); // 设置系统类库解析
22
23         vm = emulator.createDalvikVM(new File( pathname: "unidbg-android/src/test/resources/lesson1/xc 8-38-2.apk"));
24         vm.setVerbose(true); // 设置是否打印Jni调用细节
25
26         new AndroidModule(emulator, vm).register(memory);
27         DalvikModule dm = vm.loadLibrary(new File( pathname: "unidbg-android/src/test/resources/lesson1/libscmain.so"),
28
29         vm.setJni(this);
30         dm.callJNI_OnLoad(emulator);
31     }
32
33     public static void main(String[] args) {
34         xc test = new xc();
35     }
36 }
37

```


这样这个依赖找不到的报错就消失了



```
Run: xc
"C:\Program Files\Java\jdk-16.0.1\bin\java.exe" ....
[09:44:20 654] INFO [com.github.unidbg.linux.ARM32SyscallHandler] (ARM32SyscallHandler:1906) - openat dirfd=-100, pathname=proc/9720/status, oflags=0x0, mode=0
JNINet->FindClass(ctrrip/android/security/SecurityUtil) was called from RX@x4008ec7b[libscmain.so]0x8ec7b
[09:44:20 682] INFO [com.github.unidbg.linux.ARM32SyscallHandler] (ARM32SyscallHandler:1906) - openat dirfd=-100, pathname=proc/9720/cmdline, oflags=0x0, mode=0
[09:44:20 683] WARN [com.github.unidbg.arm.AbstractARMEulator] (AbstractARMEulator$1:58) - memory failed: address=0x6, size=2, value=0xc9d0, PC=unidbg@0xbfffc9d0, LR=unidbg@0x
[09:44:20 683] WARN [com.github.unidbg.AbstractEmulator] (AbstractEmulator:389) - emulate RX@x4008e4bd[libscmain.so]0x8e4bd exception sp=unidbg@0xbfffc9d0, msg=unicorn.UnicornE
Exception in thread "main" java.lang.IllegalStateException: Create breakpoint : Illegal JNI version: 0xffffffff
    at com.github.unidbg.linux.android.dvm.BaseVM.checkVersion(BaseVM.java:194)
    at com.github.unidbg.linux.android.dvm.DalvikModule.callJNI_OnLoad(DalvikModule.java:39)
    at com.lesson1.xc.<init>(xc.java:30)
    at com.lesson1.xc.main(xc.java:34)
```

这里只需要记住一件事，加载的时机必须在加载libscmain之前，这是很好理解的，libscmain.so依赖libandroid.so，那么libandroid.so肯定得更早出现在内存中，等着人家。

接下来看下一行

```
INFO [com.github.unidbg.linux.ARM32SyscallHandler] (ARM32SyscallHandler:1906) -
openat dirfd=-100, pathname=proc/9720/status, oflags=0x0, mode=0
```

这是什么呢？

首先它的前缀是“ARM32SyscallHandler”，望文生义即系统调用

openat dirfd=-100, pathname=proc/9720/status, oflags=0x0, mode=0

这儿用了一个叫openat的系统调用，打开“proc/9720/status”这个文件

那事儿就好办了，我们得搞懂两件事

- 样本想做什么
- 为什么这儿会出问题

首先是第一点，样本想做什么，从Unidbg的日志中我们了解到，这儿使用系统调用openat打开了**proc/9720/status**这个文件，但这个操作是想干啥呢？它为什么要打开或者说访问这个文件？只需要了解以下几个概念

1.proc 文件系统由内核提供，系统中正在运行的每个进程都有对应的一个目录在 proc 下，其以进程的PID 号为目录名，这个目录是读取进程信息的接口。

所以说“proc/9720/status”，就是读取9720这个进程的相关信息，status具体是啥我们还不清楚。

2.9720是什么，为什么读取9720

读者运行代码时，可能此处不是9720，事实上，每次运行时这儿的数字都会变。这个数字是什么？其实它是我们在Unidbg中的进程ID，即PID。每次运行时会自动生成一个进程PID，这显然比固定PID好，如果Unidbg固定PID，很容易依此检测和对抗Unidbg。

怎么验证这是PID呢？

我们可以打印验证一下



明白了以上两点后我们意识到，样本在读取当前进程的状态文件，除此之外我们还注意到，下面还读取了cmdline？

所以我们需要搞懂，proc/pid 目录下的status和cmdline文件分别包含进程的什么信息？

在Android系统中，cmdline里是应用的进程名，而Status，则包含的信息非常多：可执行文件名、当前状态、PID 和 PPID、实际及有效的 UID 和 GID、内存使用情况、以及其他。

我们可以在adb中验证一下

```
C:\Users\pr0214>adb shell
bullhead:/ $ su
bullhead:/ # ps -A | grep ctrip
u0_a163      26391   3000 2229116 278784 0          cb01a7ba R
ctrip.android.view
u0_a163      26658   3000 1710176 49480 sys_epoll_wait e7015a74 S
ctrip.android.view:pushsdk.v1
bullhead:/ # cd /proc/26391/
bullhead:/proc/26391 # cat cmdline
ctrip.android.viewbullhead:/proc/26391 # cat status
Name:   ip.android.view
State:  S (sleeping)
Tgid:   26391
Pid:    26391
PPid:   3000
TracerPid: 0
Uid:    10163   10163   10163   10163
Gid:    10163   10163   10163   10163
FDSize: 512
Groups: 3002 3003 9997 20163 50163
VmPeak: 2283272 kB
VmSize: 2216568 kB
VmLck:   0 kB
VmPin:   0 kB
VmHWM:   390252 kB
VmRSS:   271092 kB
VmData:  411888 kB
VmStk:    8192 kB
VmExe:    20 kB
VmLib:   200676 kB
VmPTE:    2100 kB
VmSwap:   3312 kB
Threads: 143
SigQ:    1/6517
SigPnd:  0000000000000000
```

```

ShdPnd: 0000000000000000
SigBlk: 0000000000001204
SigIgn: 0000000000000000
SigCgt: 00000006400096fc
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 0000000000000000
CapAmb: 0000000000000000
Seccomp:      2
Cpus_allowed: 0f
Cpus_allowed_list:      0-3
Mems_allowed:  1
Mems_allowed_list:      0
voluntary_ctxt_switches:      13284
nonvoluntary_ctxt_switches:   7709
bullhead:/proc/26391 #

```

那么问题来了，样本读取这两个文件做什么？

一般而言，样本检测status是为了其中的TracerPid字段，TracerPid为0说明样本没有被调试，不为0说明正在被调试，所以一般会检测TracePid，如果被调试则直接退出或者引向错误逻辑。

而cmdline返回进程的名字，它可以有效的防止样本被重打包，当样本发现进程名和自身不符时，即说明自己可能被重打包了，就会直接退出或者引向错误逻辑。

样本想做什么这一个问题我们已经搞懂了，它读取了当前进程的status以及cmdline文件，目的是反调试以及反重打包。

那么看第二个问题——Unidbg中如何处理

Unidbg不是一个完备的Android系统，可没有什么自动生成的proc文件系统，因此样本找不到这两个文件。Unidbg对文件访问的相关API全部进行文件的重定向，我们可以通过两种方式实现这种重定向。

先说一下代码的方式，即实现IOResolve接口，实现其中的Resolve方法，并绑定IO重定向，来看一下代码

```

package com.lesson1;

import com.github.unidbg.AndroidEmulator;
import com.github.unidbg.Emulator;
import com.github.unidbg.file.FileResult;
import com.github.unidbg.file.IOResolver;
import com.github.unidbg.linux.android.AndroidEmulatorBuilder;
import com.github.unidbg.linux.android.AndroidResolver;
import com.github.unidbg.linux.android.dvm.AbstractJni;
import com.github.unidbg.linux.android.dvm.DalvikModule;
import com.github.unidbg.linux.android.dvm.VM;
import com.github.unidbg.memory.Memory;
import com.github.unidbg.virtualmodule.android.AndroidModule;

import java.io.File;

// 1. 实现IOResolve
public class xc extends AbstractJni implements IOResolver {
    private final AndroidEmulator emulator;
    private final VM vm;

```

```

xc() {
    emulator = AndroidEmulatorBuilder.for32Bit().build(); // 创建模拟器实例，要
    模拟32位或者64位，在这里区分

    // 2.绑定IO重定向接口
    emulator.getSyscallHandler().addIOResolver(this);
    System.out.println("当前进程PID: "+emulator.getPid());
    final Memory memory = emulator.getMemory(); // 模拟器的内存操作接口
    memory.setLibraryResolver(new AndroidResolver(23)); // 设置系统类库解析

    vm = emulator.createDalvikVM(new File("unidbg-
    android/src/test/resources/lesson1/xc_8-38-2.apk"));
    vm.setVerbose(true); // 设置是否打印Jni调用细节

    new AndroidModule(emulator, vm).register(memory);
    DalvikModule dm = vm.loadLibrary(new File("unidbg-
    android/src/test/resources/lesson1/libscmain.so"), true);

    vm.setJni(this);
    dm.callJNI_OnLoad(emulator);

}

public static void main(String[] args) {
    xc test = new xc();
}

//3
@Override
public FileResult resolve(Emulator emulator, String pathname, int oflags) {
    System.out.println("访问: "+pathname);
    return null;
}
}

```

此处我们只是打印了文件名，尚未做任何处理，运行一下试试



```

Run: xc
"C:\Program Files\Java\jdk-16.0.1\bin\java.exe" ...
当前进程PID: 25360
访问, stdin
访问, stdout
访问, stderr
访问, /dev/_properties_
访问, /proc/stat
访问, /proc/25360/status
[11:10:39 544] INFO [com.github.unidbg.linux.ARM32SyscallHandler] (ARM32SyscallHandler:1906) - openat dirfd=-100, pathname=proc/25360/status, oflags=0x0, mode=0
JNIEnv->FindClass(ctrip/android/security/SecurityUtil) was called from RX@0x4008ec7b[libscmain.so]0x8ec7b
访问, /proc/25360/cmdline
[11:10:39 572] INFO [com.github.unidbg.linux.ARM32SyscallHandler] (ARM32SyscallHandler:1906) - openat dirfd=-100, pathname=proc/25360/cmdline, oflags=0x0, mode=0
[11:10:39 573] WARN [com.github.unidbg.arm.AbstractARMEulator] (AbstractARMEulator$1:58) - memory failed: address=0x6, size=2, value=0xc9d0, PC=unidbg@0xbfffc9d0, LR=unidbg@0x
[11:10:39 573] WARN [com.github.unidbg.AbstractEmulator] (AbstractEmulator:389) - emulate RX@0x4008e4bd[libscmain.so]0x8e4bd exception sp=unidbg@0xbfffc9d0, msg=unicorn.UnicornE
Exception in thread "main" java.lang.IllegalStateException Create breakpoint : Illegal JNI version: 0xffffffff
    at com.github.unidbg.linux.android.dvm.BaseVM.checkVersion(BaseVM.java:194)
    at com.github.unidbg.linux.android.dvm.DalvikModule.callJNI_OnLoad(DalvikModule.java:39)
    at com.lesson1.xc.<init>(xc.java:39)

```

可以发现，我们关注的两个pathName就在其中，可是还有一个问题，前面的哪些/proc/stat之类是怎么回事？

不用担心，这些都是libc里初始化过程中产生的文件访问，而不是libscmain自己的逻辑，所以不用去理睬。

接下来补/proc/pid/cmdline

```
@Override
public FileResult resolve(Emulator emulator, String pathname, int oflags) {
    if (("proc/"+emulator.getPid()+"/cmdline").equals(pathname)) {
        return FileResult.success(new ByteArrayFileIO(oflags, pathname,
"ctrip.android.view".getBytes()));
    }
    return null;
}
```

除此之外也可以新建一个文件，传入文件

```
@Override
public FileResult resolve(Emulator emulator, String pathname, int oflags) {
    if (("proc/"+emulator.getPid()+"/cmdline").equals(pathname)) {
        // return FileResult.success(new ByteArrayFileIO(oflags, pathname,
"ctrip.android.view".getBytes()));
        return FileResult.success(new SimpleFileIO(oflags, new
File("D:\\unidbg-teach\\unidbg-
android\\src\\test\\java\\com\\lesson1\\cmdline"), pathname));
    }
    return null;
}
```

接下来补status

我们看一下ABC三人的补发，以及指出其中可能的问题

先看A的做法

```
@Override
public FileResult resolve(Emulator emulator, String pathname, int oflags) {
    if (("proc/"+emulator.getPid()+"/cmdline").equals(pathname)) {
        // return FileResult.success(new ByteArrayFileIO(oflags, pathname,
"ctrip.android.view".getBytes()));
        return FileResult.success(new SimpleFileIO(oflags, new
File("D:\\unidbg-teach\\unidbg-
android\\src\\test\\java\\com\\lesson1\\cmdline"), pathname));
    }
    if (("proc/" + emulator.getPid() + "/status").equals(pathname)) {
        return FileResult.success(new ByteArrayFileIO(oflags, pathname,
"TracerPid:\t0\n".getBytes()));
    }
    return null;
}
```

先前我们说过，status主要用来检测TracerPid，所以A只返回TracerPid 为0这么一小段。

这样做是有风险的

```
Name: ip.android.view
State: S (sleeping)
Tgid: 26391
Pid: 26391
PPid: 3000
TracerPid: 0
uid: 10163 10163 10163 10163
```

```

Gid: 10163 10163 10163 10163
FDSize: 512
Groups: 3002 3003 9997 20163 50163
VmPeak: 2283272 kB
VmSize: 2216568 kB
VmLck: 0 kB
VmPin: 0 kB
VmHWM: 390252 kB
VmRSS: 271092 kB
VmData: 411888 kB
VmStk: 8192 kB
VmExe: 20 kB
VmLib: 200676 kB
VmPTE: 2100 kB
VmSwap: 3312 kB
Threads: 143
SigQ: 1/6517
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000001204
SigIgn: 0000000000000000
SigCgt: 00000006400096fc
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 0000000000000000
CapAmb: 0000000000000000
Seccomp: 2
Cpus_allowed: 0f
Cpus_allowed_list: 0-3
Mems_allowed: 1
Mems_allowed_list: 0
voluntary_ctxt_switches: 13284
nonvoluntary_ctxt_switches: 7709

```

如果样本读取了status中的Name字段，那岂不是取不到，万一出问题呢？

我们再看看B的操作，B通过adb 查看了state的内容，然后完整复制了过来

```

if (("proc/" + emulator.getPid() + "/status").equals(pathname)) {
    return FileResult.success(new ByteArrayFileIO(oflags, pathname, ("Name:
ip.android.view\n" +
    "State: R (running)\n" +
    "Tgid: 7232\n" +
    "Pid: 7232\n" +
    "PPid: 3000\n" +
    "TracerPid: 0\n" +
    "Uid: 10163 10163 10163 10163\n" +
    "Gid: 10163 10163 10163 10163\n" +
    "FDSize: 512\n" +
    "Groups: 3002 3003 9997 20163 50163\n" +
    "VmPeak: 2319784 kB\n" +
    "VmSize: 2240148 kB\n" +
    "VmLck: 0 kB\n" +
    "VmPin: 0 kB\n" +
    "VmHWM: 413060 kB\n" +
    "VmRSS: 310988 kB\n" +

```

```

        "VmData: 427160 kB\n" +
        "VmStk: 8192 kB\n" +
        "VmExe: 20 kB\n" +
        "VmLib: 200676 kB\n" +
        "VmPTE: 2100 kB\n" +
        "VmSwap: 3356 kB\n" +
        "Threads: 149\n" +
        "SigQ: 1/6517\n" +
        "SigPnd: 0000000000000000\n" +
        "ShdPnd: 0000000000000000\n" +
        "SigBlk: 000000000001204\n" +
        "SigIgn: 0000000000000000\n" +
        "SigCgt: 00000006400096fc\n" +
        "CapInh: 0000000000000000\n" +
        "CapPrm: 0000000000000000\n" +
        "CapEff: 0000000000000000\n" +
        "CapBnd: 0000000000000000\n" +
        "CapAmb: 0000000000000000\n" +
        "Seccomp: 2\n" +
        "Cpus_allowed: 0f\n" +
        "Cpus_allowed_list: 0-3\n" +
        "Mems_allowed: 1\n" +
        "Mems_allowed_list: 0\n" +
        "voluntary_ctxt_switches: 6918\n" +
        "nonvoluntary_ctxt_switches: 4988").getBytes());
    }

```

B的做法显然更稳妥一些，但是，如果样本读取status的Pid会出现什么问题呢？这份status的Pid是app在真机中的进程ID，但Unidbg每次随机生成PID，如果样本通过API读取Unidbg的这个PID，两者对比是不一致的，这显然不太好。

看看C的做法

```

if (("proc/" + emulator.getPid() + "/status").equals(pathname)) {
    return FileResult.success(new ByteArrayFileIO(oflags, pathname, ("Name:
ip.android.view\n" +
        "State: R (running)\n" +
        "Tgid: "+emulator.getPid()+"\n" +
        "Pid: "+emulator.getPid()+"\n" +
        "PPid: 3000\n" +
        "TracerPid: 0\n" +
        "Uid: 10163 10163 10163 10163\n" +
        "Gid: 10163 10163 10163 10163\n" +
        "FDSize: 512\n" +
        "Groups: 3002 3003 9997 20163 50163\n" +
        "VmPeak: 2319784 kB\n" +
        "VmSize: 2240148 kB\n" +
        "VmLck: 0 kB\n" +
        "VmPin: 0 kB\n" +
        "VmHWM: 413060 kB\n" +
        "VmRSS: 310988 kB\n" +
        "VmData: 427160 kB\n" +
        "VmStk: 8192 kB\n" +
        "VmExe: 20 kB\n" +
        "VmLib: 200676 kB\n" +
        "VmPTE: 2100 kB\n" +
        "VmSwap: 3356 kB\n" +

```

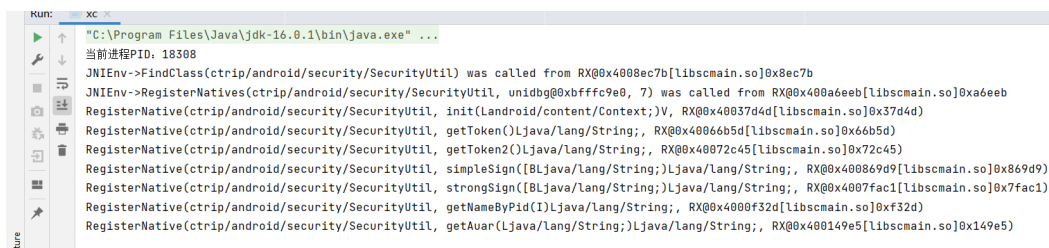


```

        "Threads:      149\n" +
        "SigQ:    1/6517\n" +
        "SigPnd: 0000000000000000\n" +
        "ShdPnd: 0000000000000000\n" +
        "SigBlk: 000000000001204\n" +
        "SigIgn: 0000000000000000\n" +
        "SigCgt: 00000006400096fc\n" +
        "CapInh: 0000000000000000\n" +
        "CapPrm: 0000000000000000\n" +
        "CapEff: 0000000000000000\n" +
        "CapBnd: 0000000000000000\n" +
        "CapAmb: 0000000000000000\n" +
        "Seccomp:      2\n" +
        "Cpus_allowed:   0f\n" +
        "Cpus_allowed_list:      0-3\n" +
        "Mems_allowed:   1\n" +
        "Mems_allowed_list:      0\n" +
        "voluntary_ctxt_switches:        6918\n" +
        "nonvoluntary_ctxt_switches:    4988").getBytes());
    }

```

C将PID做了一下统一，显然这是更好的方案，再次运行代码，已经没有报错了。

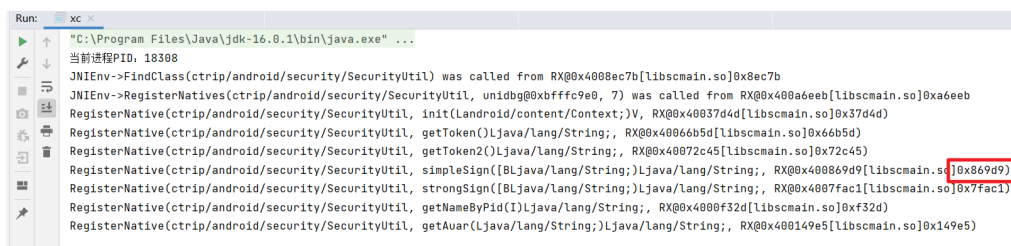


```

Run: xc
"C:\Program Files\Java\jdk-16.0.1\bin\java.exe" ...
当前进程PID: 18308
JNIEnv->FindClass(ctrip/android/security/SecurityUtil) was called from RX@0x4008ec7b[libscmain.so]0x8ec7b
JNIEnv->RegisterNatives(ctrip/android/security/SecurityUtil, unidbg@0xbfffc9e0, 7) was called from RX@0x400a6eeb[libscmain.so]0xa6eeb
RegisterNative(ctrip/android/security/SecurityUtil, init(Landroid/content/Context;)V, RX@0x40037d4d[libscmain.so]0x37d4d)
RegisterNative(ctrip/android/security/SecurityUtil, getToken()Ljava/lang/String; , RX@0x40066b5d[libscmain.so]0x66b5d)
RegisterNative(ctrip/android/security/SecurityUtil, getToken2()Ljava/lang/String; , RX@0x40072c45[libscmain.so]0x72c45)
RegisterNative(ctrip/android/security/SecurityUtil, simpleSign([BLjava/lang/String;)Ljava/lang/String; , RX@0x400869d9[libscmain.so]0x869d9)
RegisterNative(ctrip/android/security/SecurityUtil, strongSign([BLjava/lang/String;)Ljava/lang/String; , RX@0x4007fac1[libscmain.so]0x7fac1)
RegisterNative(ctrip/android/security/SecurityUtil, getNameByPid(I)Ljava/lang/String; , RX@0x400f32d[libscmain.so]0xf32d)
RegisterNative(ctrip/android/security/SecurityUtil, getAvar(Ljava/lang/String;)Ljava/lang/String; , RX@0x400149e5[libscmain.so]0x149e5)

```

接下来运行SimpleSign方法吧



```

Run: xc
"C:\Program Files\Java\jdk-16.0.1\bin\java.exe" ...
当前进程PID: 18308
JNIEnv->FindClass(ctrip/android/security/SecurityUtil) was called from RX@0x4008ec7b[libscmain.so]0x8ec7b
JNIEnv->RegisterNatives(ctrip/android/security/SecurityUtil, unidbg@0xbfffc9e0, 7) was called from RX@0x400a6eeb[libscmain.so]0xa6eeb
RegisterNative(ctrip/android/security/SecurityUtil, init(Landroid/content/Context;)V, RX@0x40037d4d[libscmain.so]0x37d4d)
RegisterNative(ctrip/android/security/SecurityUtil, getToken()Ljava/lang/String; , RX@0x40066b5d[libscmain.so]0x66b5d)
RegisterNative(ctrip/android/security/SecurityUtil, getToken2()Ljava/lang/String; , RX@0x40072c45[libscmain.so]0x72c45)
RegisterNative(ctrip/android/security/SecurityUtil, simpleSign([BLjava/lang/String;)Ljava/lang/String; , RX@0x400869d9[libscmain.so]0x8669d9)
RegisterNative(ctrip/android/security/SecurityUtil, strongSign([BLjava/lang/String;)Ljava/lang/String; , RX@0x4007fac1[libscmain.so]0x7fac1)
RegisterNative(ctrip/android/security/SecurityUtil, getNameByPid(I)Ljava/lang/String; , RX@0x400f32d[libscmain.so]0xf32d)
RegisterNative(ctrip/android/security/SecurityUtil, getAvar(Ljava/lang/String;)Ljava/lang/String; , RX@0x400149e5[libscmain.so]0x149e5)

```

我选择直接Call 这个地址，自己组装参数，而不是使用Unidbg封装的callxxx，这是我的个人习惯，同时也是因为这种方式可以应对各种复杂的场景，遇到了我们再细谈。

看一下完整的代码吧

```

package com.lesson1;

import com.github.unidbg.AndroidEmulator;
import com.github.unidbg.Emulator;
import com.github.unidbg.file.FileResult;
import com.github.unidbg.file.IOResolver;
import com.github.unidbg.linux.android.AndroidEmulatorBuilder;
import com.github.unidbg.linux.android.AndroidResolver;
import com.github.unidbg.linux.android.dvm.AbstractJni;
import com.github.unidbg.linux.android.dvm.DalvikModule;
import com.github.unidbg.linux.android.dvm.StringObject;
import com.github.unidbg.linux.android.dvm.VM;

```

```

import com.github.unidbg.linux.android.dvm.array.ByteArray;
import com.github.unidbg.linux.file.ByteArrayFileIO;
import com.github.unidbg.linux.file.SimpleFileIO;
import com.github.unidbg.memory.Memory;
import com.github.unidbg.Module;
import com.github.unidbg.virtualmodule.android.AndroidModule;

import java.io.File;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;

// 实现IOResolve
public class xc extends AbstractJni implements IOResolver {
    private final AndroidEmulator emulator;
    private final VM vm;
    private final Module module;

    xc() {
        emulator = AndroidEmulatorBuilder.for32Bit().setRootDir(new
File("target/rootfs")).build(); // 创建模拟器实例，要模拟32位或者64位，在这里区分

        // 绑定IO重定向接口
        emulator.getSyscallHandler().addIOResolver(this);
        System.out.println("当前进程PID: "+emulator.getPid());
        final Memory memory = emulator.getMemory(); // 模拟器的内存操作接口
        memory.setLibraryResolver(new AndroidResolver(23)); // 设置系统类库解析

        vm = emulator.createDalvikVM(new File("unidbg-
android/src/test/resources/lesson1/xc_8-38-2.apk"));
        vm.setVerbose(true); // 设置是否打印jni调用细节

        new AndroidModule(emulator, vm).register(memory);
        DalvikModule dm = vm.loadLibrary(new File("unidbg-
android/src/test/resources/lesson1/libscmain.so"), true);
        module = dm.getModule();
        vm.setJni(this);
        dm.callJNI_OnLoad(emulator);
    }

    public void callSimpleSign(){
        List<Object> list = new ArrayList<>(10);
        list.add(vm.getJNIEnv());
        list.add(0);
        String input = "7be9f13e7f5426d139cb4e5dbb1fdb7";
        byte[] inputByte = input.getBytes(StandardCharsets.UTF_8);
        ByteArray inputByteArray = new ByteArray(vm,inputByte);
        list.add(vm.addLocalObject(inputByteArray));
        list.add(vm.addLocalObject(new StringObject(vm, "getdata")));
        Number number = module.callFunction(emulator, 0x869d9, list.toArray()
[0]);

        System.out.println(vm.getObject(number.intValue()).getValue().toString());
    };

    public static void main(String[] args) {

```

```

        xc test = new xc();
    }

    @Override
    public FileResult resolve(Emulator emulator, String pathname, int oflags) {
        if (("proc/"+emulator.getPid()+"/cmdline").equals(pathname)) {
            // return FileResult.success(new ByteArrayFileIO(oflags, pathname,
            "ctrip.android.view".getBytes()));
            return FileResult.success(new SimpleFileIO(oflags, new
            File("D:\\unidbg-teach\\unidbg-
            android\\src\\test\\java\\com\\lesson1\\cmdline"), pathname));
        }
        if (("proc/" + emulator.getPid() + "/status").equals(pathname)) {
            return FileResult.success(new ByteArrayFileIO(oflags, pathname,
            ("Name:  ip.android.view\n" +
                "State:  R (running)\n" +
                "Tgid:   "+emulator.getPid()+"\n" +
                "Pid:    "+emulator.getPid()+"\n" +
                "PPid:   3000\n" +
                "TracerPid:  0\n" +
                "Uid:    10163  10163  10163  10163\n" +
                "Gid:    10163  10163  10163  10163\n" +
                "FDSize: 512\n" +
                "Groups: 3002 3003 9997 20163 50163\n" +
                "VmPeak: 2319784 kB\n" +
                "VmSize: 2240148 kB\n" +
                "VmLck:   0 kB\n" +
                "VmPin:   0 kB\n" +
                "VmHWM:  413060 kB\n" +
                "VmRSS:  310988 kB\n" +
                "VmData: 427160 kB\n" +
                "VmStk:   8192 kB\n" +
                "VmExe:   20 kB\n" +
                "VmLib:  200676 kB\n" +
                "VmPTE:   2100 kB\n" +
                "VmSwap:  3356 kB\n" +
                "Threads: 149\n" +
                "SigQ:    1/6517\n" +
                "SigPnd: 0000000000000000\n" +
                "ShdPnd: 0000000000000000\n" +
                "SigBlk: 000000000001204\n" +
                "SigIgn: 0000000000000000\n" +
                "SigCgt: 00000006400096fc\n" +
                "CapInh: 0000000000000000\n" +
                "CapPrm: 0000000000000000\n" +
                "CapEff: 0000000000000000\n" +
                "CapBnd: 0000000000000000\n" +
                "CapAmb: 0000000000000000\n" +
                "Seccomp: 2\n" +
                "Cpus_allowed: 0f\n" +
                "Cpus_allowed_list: 0-3\n" +
                "Mems_allowed: 1\n" +
                "Mems_allowed_list: 0\n" +
                "voluntary_ctxt_switches: 6918\n" +
                "nonvoluntary_ctxt_switches: 4988").getBytes()));
        }
        return null;
    }

```

```
}  
}
```

需要注意几个点

- 传入Native的JAVA参数，除了八个基本类型外（byte、char、short、int、long、float、double、boolean），都必须vm.addLocalObject添加到局部引用中去。
- 此处参数1是JNIEnv，参数2是JObject，但因为JObject一般不适用，所以我此处传为空，这样做有小风险。但我不想管它。

运行测试一下



我们惊喜的发现，结果出来了！

但是。。。这是正确的结果吗？

Frida Call 函数时候我们发现，结果每次都不一样，那我们怎么验证这个结果是正确的呢？

五、结果疑云

我们没法确认结果是否正确，参数不变的情况下，Frida Call的结果也一直在变。为什么参数不变，结果可以一直变化呢？可能的原因其实非常多，但最常见的是两种

- 运算过程中有时间戳
- 运算过程中有随机数

那么只能碰碰运气了

```
function callSimpleSign(){  
    var securityUtil = null;  
    Java.perform(function () {  
        Java.choose("ctrip.android.security.SecurityUtil", {  
            //枚举时调用  
            onMatch:function(instance){  
                //打印实例  
                securityUtil = instance;  
                console.log("find instance")  
            },  
            //枚举完成后调用  
            onComplete:function() {  
                console.log("end")  
            }  
        });  
        var javaString = Java.use('java.lang.String');
```

```

        var input1 =
javaString.$new("7be9f13e7f5426d139cb4e5dbb1fdb7").getBytes();
        var result = securityUtil.simpleSign(input1, "getdata");
        console.log(result);
    })
}

function main(){
    for(var i=0;i<100;i++){
        callSimpleSign()
    }
}

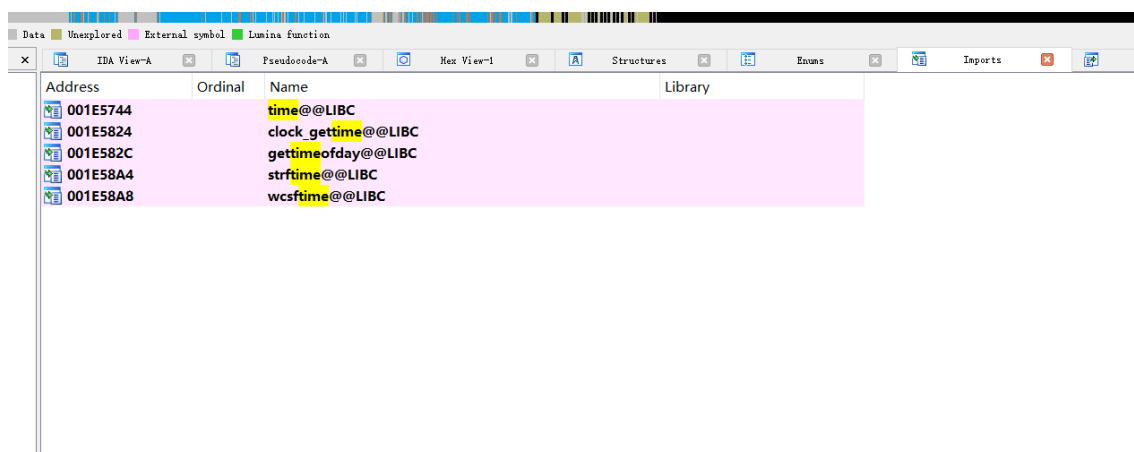
```

我们主动调用一百次SimpleSign，发现连续的十数次结果一致，这说明运算中有秒级时间戳的参与。换言之，只要我们Hook了样本中获取时间戳的API，就可以让算法结果固定下来。只要将Frida以及Unidbg中这个对应的API Hook住，对比两个结果是否一致，就可以确定Unidbg计算出了正确结果。

我们似乎定了一个妙计，但我劝大家别急着做，我们思考两个问题。

- 此样本是时间戳导致的结果不固定，如果是其他原因导致的结果不固定呢？
- 即使是时间戳导致的结果不固定，怎么确认这个时间戳是从哪来的呢？

以此样本为例



除此之外别忘了，还有诸多系统调用可以获得时间信息！那该怎么办，在这种结果不固定的情况下，尽可能保证结果的准确性呢？有人可能会说可以模拟请求，看是否能返回数据，但是，即使得到了正确返回值也并不能代表结果就是对的，它是否会在某个深夜给我们返回假数据呢？或者我们是否被打上了标签呢？我们注意到，Native函数中有一个叫init的函数，是不是执行了它再执行目标函数才是正确的道路呢？一团迷雾哟！在日后的文章里，我们会更多探讨这些问题。

总体而言，在掌握了Unidbg基础使用之后，在实战中使用Unidbg所面临的问题和难题有三类：

- 样本存在对抗或者函数初始化，导致跑不出结果
- 样本跑出了结果，但函数结果本就变化且难以固定，无法验证是否存在对抗或是否缺少初始化
- 样本使用了Unidbg尚未实现的特性（诸如部分系统调用、以及许许多多的系统依赖库）、Unidbg本身可能存在的BUG以及Unidbg较为剧烈的版本变动问题。

六、总结

在本篇中，我们似乎做了一件虎头蛇尾的事儿，并没有给出SimpleSign的解决方案，甚至感觉是在给想学Unidbg的人泼了一盆冷水。但其实我是想和大家分享这样几个感受

- Unidbg不是万能的，有其能力边界
- Unidbg不只是一个孤立的工具，我们需要使用组合拳
- Unidbg天然适合算法还原，既因为其稳定的环境、基于Unicorn的强大Hook，也因为Unidbg补环境的过程本身，就要求我们对算法从一无所知变成至少部分了解，或者专业点说，样本从黑盒变成了灰盒。

APK链接：<https://pan.baidu.com/s/1xOfx1WhSRHwbK5-xGwld0Q>

提取码：w8i5

下期再见！