

本文总结了Unidbg Hook and Call 的知识，部分Hook代码采用Frida 与 Unidbg 对照的方式，帮助熟悉Frida但不熟悉Unidbg的读者快速入门。样例前往百度云下载。链接：<https://pan.baidu.com/s/1ZRptQrx4QAPEQhrpq6gbgg> 提取码：6666

更多Unidbg使用和算法还原的教程可见星球。



一、基础知识

1. 获取SO基地址

I Frida 获取基地址

II Unidbg 获取基地址

2. 获取函数地址

I Frida 获取导出函数地址

II Unidbg 获取导出函数地址

- III Frida 获取非导出函数地址
 - IV Unidbg 获取非导出函数地址
- 3.Unidbg Hook 大盘点
 - I 以模拟执行为目的
 - II 以算法还原为目的
- 4.本篇的基础代码
- 二、Hook 函数
 - 1.Frida
 - 2.Console Debugger
 - 3.第三方Hook框架
 - I xHook
 - II HookZz
 - III Whale
 - 4.Unicorn Hook
- 三、Replace 参数和返回值
 - 1.替换参数
 - I Frida
 - II Console Debugger
 - III 第三方Hook框架
 - 2.修改返回值
 - I Frida
 - II Console Debugger
- 四、替换函数
 - 1.Frida
 - 2.第三方Hook框架
 - 3.Console Debugger
- 五、Call 函数
 - 1.Frida
 - 2.Unidbg
- 六、Patch 与内存检索
 - 1.Patch
 - I Frida
 - II Unidbg
 - 2.内存检索
 - I Frida
 - II Unidbg
- 七、Hook时机过晚问题
 - 1.提前加载libc
 - 2.固定地址下断点
 - 3.使用Unidbg提供的模块监听器
- 八、条件断点
 - 1.限于某SO
 - I Frida
 - II Unidbg
 - 2.限于某函数
 - I Frida
 - II Unidbg
 - 3.限于某处
- 九、系统调用拦截——以时间为例
 - 1.Frida
 - 2.Unidbg
- 十、Hook 检测
 - 1.检测第三方Hook框架
 - I Inline Hook
 - II Got Hook
 - 2.检测Unicorn Based Hook
- 十一、Unidbg Trace 五件套
 - 1.Instruction tracing

- 2.Function Tracing
- 3.Memory Search
- 4.Unidbg-FindKey
- 5.Unidbg-Findcrypt
- 十二、固定随机数
- 十三、杂项

一、基础知识

1.获取SO基地址

I frida 获取基地址

```
var baseAddr = Module.findBaseAddress("libnative-lib.so");
```

II Unidbg 获取基地址

```
// 加载so到虚拟内存
DalvikModule dm = vm.loadLibrary("libnative-lib.so", true);
// 加载好的so对应为一个模块
module = dm.getModule();
// 打印libnative-lib.so在Unidbg虚拟内存中的基地址
System.out.println("baseAddr:"+module.base);
```

加载了多个SO的情况

```
// 获取某个具体SO的句柄
Module yourModule = emulator.getMemory().findModule("yourModuleName");
// 打印其基地址
System.out.println("baseAddr:"+yourModule.base);
```

如果只主动加载一个SO，其基址恒为0x40000000，这是一个检测Unidbg的点，可以在 [com/github/unidbg/memory/Memory.java](https://github.com/unidbg/memory/Memory.java) 中做修改

```
public interface Memory extends IO, Loader, StackMemory {

    long STACK_BASE = 0xc0000000L;
    int STACK_SIZE_OF_PAGE = 256; // 1024k

    // 修改内存映射的起始地址
    long MMAP_BASE = 0x40000000L;

    UnidbgPointer allocateStack(int size);
    UnidbgPointer pointer(long address);
    void setStackPoint(long sp);
}
```

2. 获取函数地址

I Frida 获取导出函数地址

```
Module.findExportByName("libc.so", "strcmp")
```

II Unidbg 获取导出函数地址

```
// 加载so到虚拟内存
DalvikModule dm = vm.loadLibrary("libnative-lib.so", true);
// 加载好的 libscmain.so对应为一个模块
module = dm.getModule();
int address = (int) module.findSymbolByName("funcNmae").getAddress();
```

III Frida 获取非导出函数地址

```
var soAddr = Module.findBaseAddress("libnative-lib.so");
var FuncAddr = soAddr.add(0x1768 + 1);
```

IV Unidbg 获取非导出函数地址

```
// 加载so到虚拟内存
DalvikModule dm = vm.loadLibrary("libnative-lib.so", true);
// 加载好的so对应为一个模块
module = dm.getModule();
// offset, 在IDA中查看
int offset = 0x1768;
// 真实地址 = baseAddr + offset
int address = (int) (module.base + offset);
```

Hook 非导出函数时，不管是Frida还是Unidbg都需要考虑thumb2下地址+1的问题。

3. Unidbg Hook 大盘点

Unidbg 在Android 上支持两大类Hook方案

- Unidbg 内置的第三方Hook框架，包括xHook/Whale/HookZz
- Unicorn Hook以及基于它封装的Console Debugger，主要就指Console Debugger。

第一类是Unidbg支持并内置的第三方Hook框架，有Dobby(前身HookZz)/Whale这样的Inline Hook 框架，也有xHook这样的PLT Hook 框架，支持这些Hook框架的使用，证明了Unidbg确实相对完善。但支持Frida还有很远的路要走，Frida比Dobby或者xHook都复杂的多，使用到了多线程、信号处理等Unidbg尚不支持的机制。总体来说，Dobby + Whale + xHook 也绝对够用了，没有非Frida不可的需求。

第二类是Unicorn Hook，只有当Unidbg的底层引擎选择为Unicorn时（默认引擎），才能使用。Unicorn提供了各种级别和粒度的Hook，内存Hook/指令/基本块 Hook/异常Hook 等等，十分强大，Unidbg基于它封装了更便于使用的Console Debugger，Unidbg也支持IDA/GDB的联合调试，但仍处于实验性质，不建议尝试。

该选择哪一类Hook方案？这得看使用Unidbg的目的。如果用于**模拟执行**，那么建议使用第一类Hook，为什么？这得从Unidbg支持的汇编执行引擎说起。Unidbg支持多种底层引擎，最早也是默认的引擎是Unicorn，从名字也能看出，Unidbg和Unicorn有很大关系。但后续Unidbg又支持了数个引擎，丰富了Unidbg的底层生态。但我们知道，任何提高程序复杂度的行为，肯定都为了解决什么问题。

hypervisor 引擎用于搭载了 *Apple Silicon* 芯片的设备；

KVM 引擎用于树莓派；

Dynarmic 引擎是为了更快的模拟执行；

Unicorn 是最强大最完善的模拟执行引擎，但它相比Dynarmic太慢了，同场景下，Dynarmic比Unicorn模拟执行快数倍甚至数十倍。因此在生产环境中，采用 Dynarmic 引擎配上 [unidbg-boot-server](#) 实现高并发。

*Dynarmic*引擎使用

```
private static AndroidEmulator createARMEmulator() {  
    return AndroidEmulatorBuilder.for32Bit()  
        // 切换为Dynarmic引擎  
        .addBackendFactory(new DynarmicFactory(true))  
        .build();  
}
```

Unicorn 默认引擎

```
private static AndroidEmulator createARMEmulator() {  
    return AndroidEmulatorBuilder.for32Bit()  
        .build();  
}
```

使用Unidbg的第二个场景是**辅助算法还原**，即模拟执行只作为算法还原的前奏，在模拟执行输出结果无误后，再使用Unidbg辅助算法还原。这种情况下对执行速度要求不高，那肯定使用更强大的Unicorn引擎。这时候两大类Hook方案都可以使用，选择哪类？我倾向于自始至终使用第二类方案，即基于Unicorn Hook的方案。

我个人认为有三点优势

- HookZz或者xHook等方案，都可以基于其Hook实现原理进行检测，但Unicorn 原生Hook不容易被检测。
- Unicorn Hook 没有局限，其他方案局限性较大。比如Inline Hook方案不能Hook短函数，或者两个相邻的地址；PLT Hook 不能 Hook Sub_xxx 子函数。
- 两类方案混用时，一定几率触发bug，事实上，单使用Unicorn的某些Hook功能都有bug，因此统一用原生Hook可以少一些bug。

总结如下

I 以模拟执行为目的

使用第三方Hook方案，arm32下HookZz的支持较好，arm64下Dobby的支持较好，HookZz/Dobby Hook不成功时，如果函数是导出函数就用xHook，否则使用 Whale。

II 以算法还原为目的

使用Console Debugger 和 Unicorn Hook，不优先使用第三方Hook方案。

4.本篇的基础代码

即模拟执行demo的代码

```
package com.tutorial;

import com.github.unidbg.AndroidEmulator;
import com.github.unidbg.Emulator;
import com.github.unidbg.Module;
import com.github.unidbg.arm.HookStatus;
import com.github.unidbg.arm.backend.Backend;
import com.github.unidbg.arm.backend.CodeHook;
import com.github.unidbg.arm.context.RegisterContext;
import com.github.unidbg.debugger.BreakPointCallback;
import com.github.unidbg.hook.HookContext;
import com.github.unidbg.hook.ReplaceCallback;
import com.github.unidbg.hook.hookzz.*;
import com.github.unidbg.hook.whale.IWhale;
import com.github.unidbg.hook.whale.Whale;
import com.github.unidbg.hook.xhook.IxHook;
import com.github.unidbg.linux.android.AndroidEmulatorBuilder;
import com.github.unidbg.linux.android.AndroidResolver;
import com.github.unidbg.linux.android.XHookImpl;
import com.github.unidbg.linux.android.dvm.DalvikModule;
import com.github.unidbg.linux.android.dvm.DvmClass;
import com.github.unidbg.linux.android.dvm.DvmObject;
import com.github.unidbg.linux.android.dvm.VM;
import com.github.unidbg.memory.Memory;
import com.github.unidbg.utils.Inspector;
import com.sun.jna.Pointer;
import unicorn.ArmConst;
import unicorn.Unicorn;

import java.io.File;

public class hookInUnidbg {
    private final AndroidEmulator emulator;
    private final VM vm;
    private final Module module;

    hookInUnidbg() {

        // 创建模拟器实例
        emulator = AndroidEmulatorBuilder.for32Bit().build();

        // 模拟器的内存操作接口
        final Memory memory = emulator.getMemory();
        // 设置系统类库解析
        memory.setLibraryResolver(new AndroidResolver(23));
        // 创建Android虚拟机
        vm = emulator.createDalvikVM(new File("unidbg-android/src/test/resources/tutorial/hookinunidbg.apk"));

        //          emulator.attach().addBreakPoint(0x40000000+0xa80);

        // 加载so到虚拟内存
```

```

        DalvikModule dm = vm.loadLibrary("hookinunidbg", true);
        // 加载好的 libhookinunidbg.so对应为一个模块
        module = dm.getModule();

        // 执行JNI_OnLoad（如果有的话）
        dm.callJNI_OnLoad(emulator);
    }

    public void call(){
        DvmClass dvmClass =
vm.resolveClass("com/example/hookinunidbg/MainActivity");
        String methodSign = "call()V";
        DvmObject<?> dvmObject = dvmClass.newObject(null);

        dvmObject.callJniMethodObject(emulator, methodSign);
    }

    public static void main(String[] args) {
        hookInUnidbg mydemo = new hookInUnidbg();
        mydemo.call();
    }
}

```

运行时有一些日志输出，为正常逻辑。

二、Hook 函数

demo hookInunidbg中运行了数个函数，在本节中关注其中运行的base64_encode函数。

```

unsigned int
base64_encode(const unsigned char *in, unsigned int inlen, char *out);

```

参数解释如下

char *out: 一块buffer的首地址，用来存放转码后的内容。

char *in: 原字符串的首地址，指向原字符串内容。

int inlen: 原字符串长度。

返回值: 正常情况下返回转换后字符串的实际长度。

本节的任务就是打印base64编码前的内容，以及编码后的内容。

1.Frida

```

// Frida version
function main(){
    // get base address of target so;
    var base_addr = Module.findBaseAddress("libhookinunidbg.so");

    if (base_addr){
        var func_addr = Module.findExportByName("libhookinunidbg.so",
"base64_encode");
    }
}

```

```

console.log("hook base64_encode function")
Interceptor.attach(func_addr,{
    // 打印入参
    onEnter: function (args) {
        console.log("\n input:")
        this.buffer = args[2];
        var length = args[1];
        console.log(hexdump(args[0],{length: length.toUInt32()}))
        console.log("\n")
    },
    // 打印返回值
    onLeave: function () {
        console.log(" output:")
        console.log(this.buffer.readCString());
    }
})
}

setImmediate(main);

```

2.Console Debugger

Console Debugger 是快速打击、快速验证的交互调试器，在call JNIOnLoad之前下断点。

```

// debug
emulator.attach().addBreakPoint(module.findSymbolByName("base64_encode").getAddress());

```

需要重申和强调几个概念

- 运行到对应地址时触发断点，类似于GDB调试或者IDA调试，时机为**目标指令执行前**。
- 断点不具有函数的种种概念，需要从汇编指令的角度去理解函数。
- Console Debugger 用于辅助算法分析，快速分析、确认某个函数的功能。在Unicorn 引擎下才可以用。

针对第二条做补充

根据ARM ATPCS调用约定，当参数个数小于等于4个的时候，子程序间通过R0~R3来传递参数（即R0-R3代表参数1-参数4），如果参数个数大于4个，余下的参数通过sp所指向的数据栈进行参数传递。而函数的返回值总是通过R0传递回来。

以目标函数为例，函数调用前，调用方把三个参数依次放在R0-R2中。


```
crackDemo x
C:\Users\13352\jdk\openjdk-16.0.2\bin\java.exe ...
debugger break at: 0x40008a0
>>> r0=0x400022e0 r1=0x5 r2=0x401d2000 r3=0x0 r4=0x0 r5=0x0 r6=0x0 r7=0xbffff740 r8=0x0 sb=0x0 sl=0x0 fp=0x0 ip=0x40003fd8
>>> SP=0xbffff730 LR=RX0x40000899[libhookinunidbg.so]0x899 PC=RX0x400008a0[libhookinunidbg.so]0x8a0 cpsr: N=0, Z=0, C=0, V=0, T=1, mode=0b10000
>>> d0=0x0(0.0) d1=0x3933312032203120(3.696225012140980E-33) d2=0x3220302034203736(3.0022298612178987E-67) d3=0x3436333832203235(3.536676186840298E-57) d4=
>>> d8=0x0(0.0) d9=0x0(0.0) d10=0x0(0.0) d11=0x0(0.0) d12=0x0(0.0) d13=0x0(0.0) d14=0x0(0.0) d15=0x0(0.0)
=> *[libhookinunidbg.so]*[0x008a1]*[* 80 b5 ]*0x400008a0:*push {r7, lr}
[libhookinunidbg.so] [0x008a3] [ 6f 46 ] 0x400008a2: mov r7, sp
[libhookinunidbg.so] [0x008a5] [ 8c b0 ] 0x400008a4: sub sp, #0x30
[libhookinunidbg.so] [0x008a7] [ 5d 4b ] 0x400008a6: ldr r3, [pc, #0x174]
[libhookinunidbg.so] [0x008a9] [ 7b 44 ] 0x400008a8: add r3, pc
[libhookinunidbg.so] [0x008ab] [ 1b 68 ] 0x400008aa: ldr r3, [r3]
[libhookinunidbg.so] [0x008ad] [ 1b 68 ] 0x400008ac: ldr r3, [r3]
[libhookinunidbg.so] [0x008af] [ 0b 93 ] 0x400008ae: str r3, [sp, #0x2c]
[libhookinunidbg.so] [0x008b1] [ 09 90 ] 0x400008b0: str r0, [sp, #0x24]
[libhookinunidbg.so] [0x008b3] [ 08 91 ] 0x400008b2: str r1, [sp, #0x20]
[libhookinunidbg.so] [0x008b5] [ 07 92 ] 0x400008b4: str r2, [sp, #0x1c]
[libhookinunidbg.so] [0x008b7] [ 00 20 ] 0x400008b6: movs r0, #0
[libhookinunidbg.so] [0x008b9] [ 06 90 ] 0x400008b8: str r0, [sp, #0x18]
[libhookinunidbg.so] [0x008bb] [ 07 f8 1e 0c ] 0x400008ba: strb r0, [r7, #-0x1e]
[libhookinunidbg.so] [0x008bf] [ 0a 90 ] 0x400008be: str r0, [sp, #0x28]
[libhookinunidbg.so] [0x008c1] [ 05 90 ] 0x400008c0: str r0, [sp, #0x14]
```

立即数可以直接看，比如此处参数2是5。如果怀疑不是立即数而是指针，比如参数1和参数3，那么在交互调试中输入 `mxx` 查看其指向的内存，等价于Frida中的`hexdump(xxx)`。写法有两种，以此处`r0`为例，既可以 `mr0` 也可以 `m0x400022e0`。

Unidbg 在数据展示上，相较于Frida Hexdump，有一些不同，体现在两方面

- Frida hexdump时，左侧基地址从当前地址开始，而Unidbg从0开始。
- Unidbg 给出了所打印数据块的md5值，方便对比两块数据块内容是否一致，而且还展示数据的Hex String，方便在大量日志中搜索。

Console Debugger 支持许多调试、分析的命令，如下：

```
c: continue
n: step over
bt: back trace

st hex: search stack
shw hex: search writable heap
shr hex: search readable heap
shx hex: search executable heap

nb: break at next block
s|si: step into
s[decimal]: execute specified amount instruction
s(blx): execute until BLX mnemonic, low performance

m(op) [size]: show memory, default size is 0x70, size may hex or decimal
mr0-mr7, mfp, mip, msp [size]: show memory of specified register
m(address) [size]: show memory of specified address, address must start with 0x

wr0-wr7, wfp, wip, wsp <value>: write specified register
wb(address), ws(address), wi(address) <value>: write (byte, short, integer)
memory of specified address, address must start with 0x
wx(address) <hex>: write bytes to memory at specified address, address must
start with 0x

b(address): add temporarily breakpoint, address must start with 0x, can be
module offset
b: add breakpoint of register PC
r: remove breakpoint of register PC
blr: add temporarily breakpoint of register LR
```

```

p (assembly): patch assembly at PC address
where: show java stack trace

trace [begin end]: Set trace instructions
traceRead [begin end]: Set trace memory read
traceWrite [begin end]: Set trace memory write
vm: view loaded modules
vbs: view breakpoints
d|dis: show disassemble
d(0x): show disassemble at specify address
stop: stop emulation
run [arg]: run test
cc size: convert asm from 0x400008a0 - 0x400008a0 + size bytes to c function

```

在Frida代码中，用 `console.log(hexdump(args[0], {length: args[1].toUInt32()}))` 来打印 **参数1指向的内存块，长度为参数2的值**，Unidbg中同样可以指定长度。

```

mr0 5

>-----<
[23:41:37 891] r0=RX@0x400022e0[libhookinunidbg.so]0x22e0,
md5=f5704182e75d12316f5b729e89a499df, hex=6c696c6163
size: 5
0000: 6c 69 6c 61 63                               lilac
^-----^

```

目前Console Debugger 还不支持 `mr0 r1` 这样的语法。

至此实现了Frida OnEnter的功能，接下来要获取OnLeave即函数执行完的时机。在ARM编程中，LR寄存器存放了程序的返回地址，当函数跑到LR所指向的地址时，意味着函数结束跳转了出来。又因为断点是在目标地址执行前触发，所以在LR处的断点断下时，目标函数执行完且刚执行完，这就是Frida OnLeave 时机点的原理。在Console Debugger交互调试中，使用 `blr` 命令可以在 `lr` 处下一个临时断点，它只会触发一次。

整体逻辑如下

- 在目标函数的地址处下断点
- 运行到断点处，进入Console Debugger 交互调试
- `mxx` 系列查看参数
- `blr` 在函数返回处下断点
- `c` 使程序继续运行，到返回值处断下
- 查看此时的buffer

需要注意的是，在onLeave时机中通过`mr2`查看入参3是胡闹。R2只在程序入口处表示参数3，在函数运算的过程中，R2作为通用寄存器被用于存储、运算，它已经不是指向buffer的地址了。在Frida中也存在这个问题，所以我们在OnEnter里将`args[2]`即R2的值保存在`this.buffer`中，OnLeave中再取出来打印。而在Console Debugger交互调试中，办法更简单粗暴——鼠标往上拉一下，看看原来`r2`的值是什么，发现是`0x401d2000`，然后`m0x401d2000`即可。

这样我们就实现了Frida的等价功能。似乎有点麻烦，但熟练后你会发现Console Debugger 是最快最稳的Hook & Debug 工具。除此之外，当函数被调用了三五百次时，我们不希望它反复停下来，然后不停“`c`”来继续运行。Console Debugger 也可以做持久化的Hook，代码如下。

```

public void HookByConsoleDebugger(){

```

```

emulator.attach().addBreakPoint(module.findSymbolByName("base64_encode").getAddress(), new BreakPointCallback() {
    @Override
    public boolean onHit(Emulator<?> emulator, long address) {
        RegisterContext context = emulator.getContext();
        Pointer input = context.getPointerArg(0);
        int length = context.getIntArg(1);
        Pointer buffer = context.getPointerArg(2);

        Inspector.inspect(input.getByteArray(0, length), "base64 input");
        // OnLeave
        emulator.attach().addBreakPoint(context.getLRPointer().peer, new BreakPointCallback() {
            @Override
            public boolean onHit(Emulator<?> emulator, long address) {
                // onHit返回ture时，断点触发时不会进入交互界面；为false时会。
                String result = buffer.getString(0);
                System.out.println("base64 result:"+result);
                return true;
            }
        });
        return true;
    }
});
}

```

3.第三方Hook框架

如下目标函数均在JNIOnLoad前调用

I xHook

```

public void HookByxhook(){
    IxHook xHook = xHookImpl.getInstance(emulator);
    xHook.register("libhookinunidbg.so", "base64_encode", new ReplaceCallback()
    {
        @Override
        public HookStatus onCall(Emulator<?> emulator, HookContext context, long originFunction) {
            Pointer input = context.getPointerArg(0);
            int length = context.getIntArg(1);
            Pointer buffer = context.getPointerArg(2);
            Inspector.inspect(input.getByteArray(0, length), "base64 input");
            context.push(buffer);
            return HookStatus.RET(emulator, originFunction);
        }
        @Override
        public void postCall(Emulator<?> emulator, HookContext context) {
            Pointer buffer = context.pop();
            System.out.println("base64 result:"+buffer.getString(0));
        }
    }, true);
    // 使其生效
    xHook.refresh();
}

```

xHook是爱奇艺开源的Android PLT hook框架，优点是挺稳定好用，缺点是不能Hook Sub_xxx 子函数。这是其原理所限。

II HookZz

```
public void HookByHookZz(){
    IHookZz hookZz = HookZz.getInstance(emulator); // 加载HookZz, 支持inline hook
    hookZz.enable_arm_arm64_b_branch(); // 测试enable_arm_arm64_b_branch, 可有可无
    hookZz.wrap(module.findSymbolByName("base64_encode"), new
    WrapCallback<HookZzArm32RegisterContext>() {
        @Override
        public void preCall(Emulator<?> emulator, HookZzArm32RegisterContext
        context, HookEntryInfo info) {
            Pointer input = context.getPointerArg(0);
            int length = context.getIntArg(1);
            Pointer buffer = context.getPointerArg(2);
            Inspector.inspect(input.getByteArray(0, length), "base64 input");
            context.push(buffer);
        }
        @Override
        public void postCall(Emulator<?> emulator, HookZzArm32RegisterContext
        context, HookEntryInfo info) {
            Pointer buffer = context.pop();
            System.out.println("base64 result:"+buffer.getString(0));
        }
    });
    hookZz.disable_arm_arm64_b_branch();
}
```

HookZz 也可以实现类似于单行断点的Hook，但在Unidbg的Hook大环境下感觉用处不大，不建议使用。

```
IHookZz hookZz = HookZz.getInstance(emulator);
hookZz.instrument(module.base + 0x978 + 1, new
InstrumentCallback<RegisterContext>() {
    @Override
    public void dbiCall(Emulator<?> emulator, RegisterContext ctx, HookEntryInfo
    info) {
        System.out.println(ctx.getIntArg(0));
    }
});
```

HookZz是老名字，现在叫Dobby，Unidbg中HookZz和Dobby是两个独立的Hook库，因为Unidbg作者认为HookZz在arm32上支持较好，Dobby在arm64上支持较好。HookZz或者说Dobby采用的是inline hook方案，因此可以Hook Sub_xxx，缺点是短函数可能出bug，受限乎其 inline Hook 原理。

III Whale

```
public void HookBywhale(){
    IWhale whale = whale.getInstance(emulator);
    whale.inlineHookFunction(module.findSymbolByName("base64_encode"), new
    ReplaceCallback() {
        Pointer buffer;
        @Override
        public HookStatus onCall(Emulator<?> emulator, long originFunction) {
            RegisterContext context = emulator.getContext();
```

```

        Pointer input = context.getPointerArg(0);
        int length = context.getIntArg(1);
        buffer = context.getPointerArg(2);
        Inspector.inspect(input.getByteArray(0, length), "base64 input");
        return HookStatus.RET(emulator, originFunction);
    }

    @Override
    public void postCall(Emulator<?> emulator, HookContext context) {
        System.out.println("base64 result:" + buffer.getString(0));
    }
}, true);
}

```

Whale 是一个跨平台的Hook框架，在Andorid Native Hook 上也是inline Hook方案，具体情况我了解不多。

4.Unicorn Hook

如果想对某个函数进行集中的、高强度的、同时又灵活的调试，Unicorn CodeHook是一个好选择。比如我想查看目标函数第一条指令的r1，第二条指令的r2，第三条指令的r3，类似于这种需求。

hook_add_new 第一个参数是Hook回调，我们这里选择CodeHook，它是逐条指令Hook，参数2是起始地址，参数3是结束地址，参数4一般填null。这意味着从起始地址到终止地址这个执行范围内的每条指令，我们都可以任在其执行前处理它。

找到目标函数的代码范围

Functions window

Function name	Segment	Start	Length	Locals	Argum
j_testBase64	.plt	000007EC	0000000C		
j_base64_encode	.plt	00000804	0000000C		
testBase64	.text	00000938	00000040	00000018	FFFFFF
base64_encode	.text	0000097C	0000017A	00000038	FFFFFF
base64_decode	.text	00000B14	00000110	00000020	000000

```

public void HookByUnicorn(){
    long start = module.base+0x97C;
    long end = module.base+0x97C+0x17A;
    emulator.getBackend().hook_add_new(new CodeHook() {
        @Override
        public void hook(Backend backend, long address, int size, Object user) {
            RegisterContext registerContext = emulator.getContext();
            if(address == module.base + 0x97C){
                int r0 = registerContext.getIntByReg(ArmConst.UC_ARM_REG_R0);
                System.out.println("0x97C 处 r0:" + Integer.toHexString(r0));
            }
            if(address == module.base + 0x97C + 2){
                int r2 = registerContext.getIntByReg(ArmConst.UC_ARM_REG_R2);
                System.out.println("0x97C +2 处 r2:" + Integer.toHexString(r2));
            }
        }
    });
}

```

```

        if(address == module.base + 0x97C + 4){

            int r4 = registerContext.getIntByReg(ArmConst.UC_ARM_REG_R4);
            System.out.println("0x97C +4 处 r4:"+Integer.toHexString(r4));

        }
    }

    @Override
    public void onAttach(Unicorn.UnHook unHook) {

    }

    @Override
    public void detach() {

    }
}, start, end, null);
}

```

三、Replace 参数和返回值

1.替换参数

需求：入参改为hello world，对应的入参长度也要改，正确结果是 **aGVsbG8gd29ybGQ=**，供验证效果。

I Frida

```

// Frida Version
function main(){
    // get base address of target so;
    var base_addr = Module.findBaseAddress("libhookinunidbg.so");

    if (base_addr){
        var func_addr = Module.findExportByName("libhookinunidbg.so",
"base64_encode");
        console.log("hook base64_encode function")
        var fakeinput = "hello world"
        var fakeinputPtr = Memory.allocUtf8String(fakeinput);
        Interceptor.attach(func_addr,{
            onEnter: function (args) {
                args[0] = fakeinputPtr;
                args[1] = ptr(fakeinput.length);
                this.buffer = args[2];
            },
            // 打印返回值
            onLeave: function () {
                console.log(" output:")
                console.log(this.buffer.readCString());
            }
        })
    }
}
}

```

```
setImmediate(main);
```

II Console Debugger

Console Debugger 如何实现这一目标?

①下断点，运行代码后进入debugger

```
emulator.attach().addBreakPoint(module.findSymbolByName("base64_encode").getAddress());
```

②通过命令修改参数1和2，字符串得通过hexstring的形式传入

```
wx0x40002403 68656c6c6f20776f726c64

>-----<
[14:06:46 165]RX@0x40002403[libhookinunidbg.so]0x2403,
md5=5eb63bbbe01eed093cb22bb8f5acdc3, hex=68656c6c6f20776f726c64
size: 11
0000: 68 65 6c 6c 6f 20 77 6f 72 6c 64          hello world
^-----^
wr1 11
>>> r1=0xb
```

Console Debugger 支持下列写操作

```
wr0-wr7, wfp, wip, wsp <value>: write specified register
wb(address), ws(address), wi(address) <value>: write (byte, short, integer)
memory of specified address, address must start with 0x
wx(address) <hex>: write bytes to memory at specified address, address must
start with 0x
```

但这其实并不方便，还是做持久化比较舒服。

```
public void ReplaceArgByConsoleDebugger(){

    emulator.attach().addBreakPoint(module.findSymbolByName("base64_encode").getAddress(), new BreakPointCallback() {
        @Override
        public boolean onHit(Emulator<?> emulator, long address) {
            RegisterContext context = emulator.getContext();
            String fakeInput = "hello world";
            int length = fakeInput.length();
            // 修改r1值为新长度
            emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_R1, length);
            MemoryBlock fakeInputBlock = emulator.getMemory().malloc(length,
            true);

            fakeInputBlock.getPointer().write(fakeInput.getBytes(StandardCharsets.UTF_8));
            // 修改r0为指向新字符串的新指针
            emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_R0,
            fakeInputBlock.getPointer().peer);

            Pointer buffer = context.getPointerArg(2);
```

```

        // OnLeave
        emulator.attach().addBreakPoint(context.getLRPointer().peer, new
BreakPointCallback() {
            @Override
            public boolean onHit(Emulator<?> emulator, long address) {
                String result = buffer.getString(0);
                System.out.println("base64 result:"+result);
                return true;
            }
        });
        return true;
    }
});
}
}

```

III 第三方Hook框架

原理和Unicorn Hook完全不同，但得益于良好的封装，代码是类似的。

① xHook

```

public void ReplaceArgByXhook(){
    IxHook xHook = XHookImpl.getInstance(emulator);
    xHook.register("libhookinunidbg.so", "base64_encode", new ReplaceCallback()
    {
        @Override
        public HookStatus onCall(Emulator<?> emulator, HookContext context, long
originFunction) {
            String fakeInput = "hello world";
            int length = fakeInput.length();
            // 修改r1值为新长度
            emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_R1, length);
            MemoryBlock fakeInputBlock = emulator.getMemory().malloc(length,
true);

            fakeInputBlock.getPointer().write(fakeInput.getBytes(StandardCharsets.UTF_8));
            // 修改r0为指向新字符串的新指针
            emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_R0,
fakeInputBlock.getPointer().peer);

            Pointer buffer = context.getPointerArg(2);
            context.push(buffer);
            return HookStatus.RET(emulator, originFunction);
        }
        @Override
        public void postCall(Emulator<?> emulator, HookContext context) {
            Pointer buffer = context.pop();
            System.out.println("base64 result:"+buffer.getString(0));
        }
    }, true);
    // 使其生效
    xHook.refresh();
}

```

② HookZz

```

public void ReplaceArgByHookZz(){

```



```

IHookZz hookZz = HookZz.getInstance(emulator); // 加载HookZz, 支持inline hook
hookZz.enable_arm_arm64_b_branch(); // 测试enable_arm_arm64_b_branch, 可有可无
hookZz.wrap(module.findSymbolByName("base64_encode"), new
wrapCallback<HookZzArm32RegisterContext>() {
    @Override
    public void preCall(Emulator<?> emulator, HookZzArm32RegisterContext
context, HookEntryInfo info) {
        Pointer input = context.getPointerArg(0);
        String fakeInput = "hello world";
        input.setString(0, fakeInput);
        context.setR1(fakeInput.length());

        Pointer buffer = context.getPointerArg(2);
        context.push(buffer);
    }
    @Override
    public void postCall(Emulator<?> emulator, HookZzArm32RegisterContext
context, HookEntryInfo info) {
        Pointer buffer = context.pop();
        System.out.println("base64 result:"+buffer.getString(0));
    }
});
hookZz.disable_arm_arm64_b_branch();
}

```

因为可以用HookZzArm32RegisterContext, 相对来说代码简单一些。

2.修改返回值

修改返回值的逻辑和替换参数并没有什么区别, 但它引出第四节, 所以还是仔细讲一下。

在demo 中, 有一个verifyApkSign函数, 它总是返回1, 并导致APK校验失败, 因此目标就是让它返回0。

```

extern "C"
JNIEXPORT void JNICALL
Java_com_example_hookinunidbg_MainActivity_call(JNIEnv *env, jobject thiz) {
    int verifyret = verifyApkSign();
    if(verifyret == 1){
        LOGE("APK sign verify failed!");
    } else{
        LOGE("APK sign verify success!");
    }
    testBase64();
}

extern "C" int verifyApkSign(){
    LOGE("verify apk sign");
    return 1;
};

```

I Frida

```
// Frida version
function main(){
    // get base address of target so;
    var base_addr = Module.findBaseAddress("libhookinunidbg.so");

    if (base_addr){
        var func_addr = Module.findExportByName("libhookinunidbg.so",
"verifyApkSign");
        console.log("hook verifyApkSign function")
        Interceptor.attach(func_addr,{
            onEnter: function (args) {

            },
            onLeave: function (retval) {
                // 修改返回值为0
                retval.replace(0);
            }
        })
    }

}

setImmediate(main);
```

II Console Debugger

```
public void ReplaceRetByConsoleDebugger(){

    emulator.attach().addBreakPoint(module.findSymbolByName("verifyApkSign").getAddress(), new BreakPointCallback() {
        @Override
        public boolean onHit(Emulator<?> emulator, long address) {
            RegisterContext context = emulator.getContext();
            // OnLeave
            emulator.attach().addBreakPoint(context.getLRPointer().peer, new BreakPointCallback() {
                @Override
                public boolean onHit(Emulator<?> emulator, long address) {
                    // 修改返回值为0
                    emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_R0, 0);
                    return true;
                }
            });
            return true;
        }
    });
}
```

我们的Hook生效了，但 verifyApkSign 函数里的log 还是打印出来了。在一些情况中，我们并不希望函数执行本来的逻辑，这就引出了第四节，即需要彻底的函数替换——替换并使用自己的函数。

四、替换函数

1.Frida

```
const verifyApkSignPtr = Module.findExportByName("libhookinunidbg.so",
"verifyApkSign");
Interceptor.replace(verifyApkSignPtr, new NativeCallback(() => {
    console.log("replace verifyApkSign Function")
    return 0;
}, 'void', []));
```

Frida 这部分门道还挺多，但不是我们这里的重点。

2.第三方Hook框架

这里只演示xHook

```
public void ReplaceFuncByHookZz(){
    HookZz hook = HookZz.getInstance(emulator);
    hook.replace(module.findSymbolByName("verifyApkSign").getAddress(), new
ReplaceCallback() {
    @Override
    public HookStatus onCall(Emulator<?> emulator, HookContext context, long
originFunction) {
        emulator.getBackend().reg_write(Unicorn.UC_ARM_REG_R0,0);
        return HookStatus.RET(emulator,context.getLR());
    }
});
}
```

xHook的版本很清晰易懂，我们做了两件事

- R0 赋值为0
- LR 赋值给 PC，这意味着函数一行不执行就返回了，又因为R0赋值0所以返回值为0。

3.Console Debugger

```
public void ReplaceFuncByConsoleDebugger(){

    emulator.attach().addBreakPoint(module.findSymbolByName("verifyApkSign").getAdd
ress(), new BreakPointCallback() {
        @Override
        public boolean onHit(Emulator<?> emulator, long address) {
            System.out.println("替换函数 verifyApkSign");
            RegisterContext registerContext = emulator.getContext();
            emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_PC,
registerContext.getLRPointer().peer);
            emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_R0, 0);
            return true;
        }
    });
}
```

非常清晰易懂。

五、Call 函数

分析具体算法时，常需要对其进行主动调用，进行更灵活和细致的分析。举两个例子

1是主动调用base64_encode 函数

2是一个更复杂一些的函数。

1.Frida

2.Unidbg

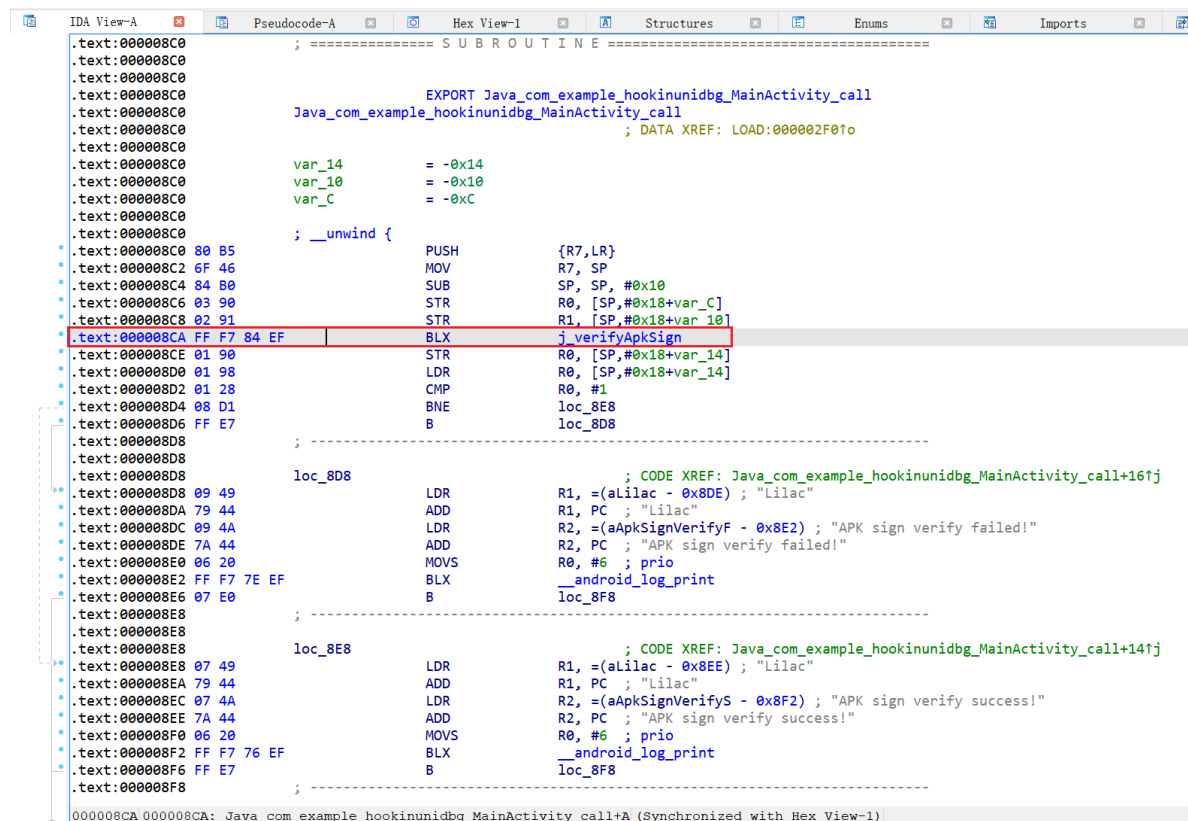
六、Patch 与内存检索

1.Patch

Patch 就是直接对二进制文件进行修改，Patch本质上只有两种形式

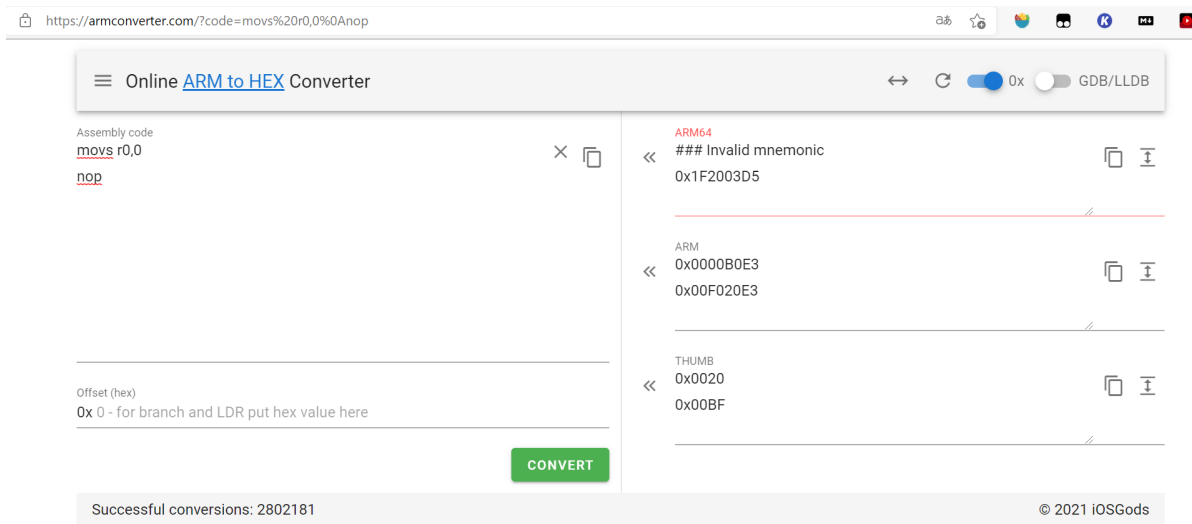
- patch 二进制文件
- 在内存里 patch

Patch的应用场景很多，有时候比Hook更简单好用，所以要介绍它。Patch 二进制文件大家都很熟悉，在IDA中使用KeyPatch即可。但这里我们关注内存Patch。



```
.text:00008C0 ; ===== S U B R O U T I N E =====
.text:00008C0
.text:00008C0 EXPORT Java_com_example_hookinunidbg_MainActivity_call
.text:00008C0 Java_com_example_hookinunidbg_MainActivity_call
.text:00008C0 ; DATA XREF: LOAD:000002F0to
.text:00008C0
.text:00008C0 var_14 = -0x14
.text:00008C0 var_10 = -0x10
.text:00008C0 var_C = -0xC
.text:00008C0
.text:00008C0 ; __unwind {
.text:00008C0 80 85 PUSH {R7,LR}
.text:00008C0 6F 46 MOV R7, SP
.text:00008C0 84 80 SUB SP, SP, #0x10
.text:00008C0 03 90 STR R0, [SP,#0x18+var_C]
.text:00008C0 02 91 STR R1, [SP,#0x18+var_10]
.text:00008CA FF F7 84 EF BLX j.verifyApkSign
.text:00008CE 01 90 STR R0, [SP,#0x18+var_14]
.text:00008D0 01 98 LDR R0, [SP,#0x18+var_14]
.text:00008D2 01 28 CMP R0, #1
.text:00008D4 08 D1 BNE loc_8D8
.text:00008D6 FF E7 B loc_8D8
.text:00008D8
.text:00008D8 ; -----
.text:00008D8 loc_8D8 LDR R1, (aLilac - 0x8DE) ; "Lilac"
.text:00008D8 09 49 ADD R1, PC ; "Lilac"
.text:00008DA 79 44 LDR R2, (aApkSignVerifyF - 0x8E2) ; "APK sign verify failed!"
.text:00008DC 09 44 ADD R2, PC ; "APK sign verify failed!"
.text:00008DE 7A 44 MOV R0, #6 ; prio
.text:00008E0 06 20 BLX __android_log_print
.text:00008E2 FF F7 7E EF B loc_8F8
.text:00008E6 07 E0
.text:00008E8
.text:00008E8 ; -----
.text:00008E8 loc_8E8 LDR R1, (aLilac - 0x8EE) ; "Lilac"
.text:00008E8 07 49 ADD R1, PC ; "Lilac"
.text:00008EA 79 44 LDR R2, (aApkSignVerifyS - 0x8F2) ; "APK sign verify success!"
.text:00008EC 07 44 ADD R2, PC ; "APK sign verify success!"
.text:00008EE 7A 44 MOV R0, #6 ; prio
.text:00008F0 06 20 BLX __android_log_print
.text:00008F2 FF F7 76 EF B loc_8F8
.text:00008F6 FF E7
.text:00008F8 ; -----
00008CA 00008CA: Java_com_example_hookinunidbg_MainActivity_call+A (Synchronized with Hex View-1)
```

0x8CA处调用了签名校验函数，第三四节中通过Replace返回值或函数的方式来处理它，但实际上，修改0x8CA处这条四字节指令也是好办法。



需要注意的是，本文只讨论了arm32，指令集只考虑最常见的thumb2，arm以及arm64可以自行测试。

I Frida

①方法一

```
var str_name_so = "libhookinunidbg.so";    //要hook的so名
var n_addr_func_offset = 0x8CA;            //要hook的函数在函数里面的偏移,thumb要+1

var n_addr_so = Module.findBaseAddress(str_name_so);
var n_addr_assemble = n_addr_so.add(n_addr_func_offset);

Memory.protect(n_addr_assemble, 4, 'rwx'); // 修改内存属性,使程序段可写
n_addr_assemble.writeByteArray([0x00, 0x20, 0x00, 0xBF]);
```

但这并不是最佳代码，Patch存在两个问题

- 是否存在多个线程同时读写这块内存？是否有冲突
- arm 的缓存刷新机制

所以Frida 提供了更安全可靠的API来修改内存中的字节

②方法二

```
var str_name_so = "libhookinunidbg.so";    //要hook的so名
var n_addr_func_offset = 0x8CA;            //要hook的函数在函数里面的偏移,thumb要+1

var n_addr_so = Module.findBaseAddress(str_name_so);
var n_addr_assemble = n_addr_so.add(n_addr_func_offset);

// safely modify bytes at address
Memory.patchCode(n_addr_assemble, 4, function () {
    // 以 thumb的方式获取一个patch对象
    var cw = new Thumbwriter(n_addr_assemble);
    // 小端序
    // 00 20
    cw.putInstruction(0x2000)
    // 00 BF
    cw.putInstruction(0xBF00);
    cw.flush(); // 内存刷新
    console.log(hexdump(n_addr_assemble))
})
```

```
});
```

II Unidbg

Unidbg在修改内存上，既可以传机器码，也可以传汇编指令，方法一和方法二其实没区别。

①方法一

```
public void Patch1(){
    // 00 20 00 bf
    int patchCode = 0xBF002000; // movs r0,0
    emulator.getMemory().pointer(module.base + 0x8CA).setInt(0,patchCode);
}
```

②方法二

```
public void Patch2(){
    byte[] patchCode = {0x00, 0x20, 0x00, (byte) 0xBF};
    emulator.getBackend().mem_write(module.base + 0x8CA, patchCode);
}
```

③方法三

```
public void Patch3(){
    try (Keystone keystone = new Keystone(KeystoneArchitecture.Arm,
        KeystoneMode.ArmThumb)) {
        KeystoneEncoded encoded = keystone.assemble("movs r0,0;nop");
        byte[] patchCode = encoded.getMachineCode();
        emulator.getMemory().pointer(module.base + 0x8CA).write(0, patchCode, 0,
            patchCode.length);
    }
}
```

2.内存检索

假设SO存在碎片化，比如要分析某个SO的多个版本，需要Patch签名校验或者某处汇编，其地址在不同版本中不一样，而且不是导出函数。内存检索+Patch就是一个好办法，可以很好适应多版本、碎片化。

搜索特征片段，可能是搜索函数开头十字节，也可能是搜索目标地址上下字节或者其他特征。

```
.text:000008C0      ; ===== SUBROUTINE =====
.text:000008C0
.text:000008C0      EXPORT Java_com_example_hookinunidbg_MainActivity_call
.text:000008C0      Java_com_example_hookinunidbg_MainActivity_call
.text:000008C0      ; DATA XREF: LOAD:000002F01o
.text:000008C0
.text:000008C0      var_14      = -0x14
.text:000008C0      var_10      = -0x10
.text:000008C0      var_C       = -0xC
.text:000008C0
.text:000008C0      ; __unwind {
.text:000008C0 50 B5      PUSH      {R7,LR}
.text:000008C2 6F 46      MOV       R7, SP
.text:000008C4 84 B0      SUB       SP, SP, #0x10
.text:000008C6 03 90      STR       R0, [SP,#0x18+var_C]
.text:000008C8 02 91      STR       R1, [SP,#0x18+var_10]
.text:000008CA FF F7 84 EF      BLX       j_verifyApkSign
.text:000008CE 01 90      STR       R0, [SP,#0x18+var_14]
.text:000008D0 01 98      LDR       R0, [SP,#0x18+var_14]
.text:000008D2 01 28      CMP       R0, #1
.text:000008D4 08 D1      BNE       loc_8E8
.text:000008D6 FF E7      B         loc_8D8
.text:000008D8      ; -----
```

I Frida

```
function searchAndPatch() {
    var module = Process.findModuleByName("libhookinunidbg.so");
    var pattern = "80 b5 6f 46 84 b0 03 90 02 91"
    var matches = Memory.scanSync(module.base, module.size, pattern);
    console.log(matches.length)
    if (matches.length !== 0)
    {
        var n_addr_assemble = matches[0].address.add(10);
        // safely modify bytes at address
        Memory.patchCode(n_addr_assemble, 4, function () {
            // 以 thumb的方式获取一个patch对象
            var cw = new ThumbWriter(n_addr_assemble);
            // 小端序
            // 00 20
            cw.putInstruction(0x2000)
            // 00 BF
            cw.putInstruction(0xBF00);
            cw.flush(); // 内存刷新
            console.log(hexdump(n_addr_assemble))
        });
    }
}

setImmediate(searchAndPatch);
```

II Unidbg

```
public void SearchAndPatch(){
    byte[] patterns = {(byte) 0x80, (byte) 0xb5,0x6f,0x46, (byte) 0x84, (byte)
    0xb0,0x03, (byte) 0x90,0x02, (byte) 0x91};
    Collection<Pointer> pointers = searchMemory(module.base,
    module.base+module.size, patterns);
    if(pointers.size() > 0){
        try (Keystone keystone = new Keystone(KeystoneArchitecture.Arm,
        KeystoneMode.ArmThumb)) {
            keystoneEncoded encoded = keystone.assemble("movs r0,0;nop");
            byte[] patchCode = encoded.getMachineCode();
            ((ArrayList<Pointer>) pointers).get(0).write(10, patchCode, 0,
            patchCode.length);
        }
    }
}

private Collection<Pointer> searchMemory(long start, long end, byte[] data) {
    List<Pointer> pointers = new ArrayList<>();
    for (long i = start, m = end - data.length; i < m; i++) {
        byte[] oneByte = emulator.getBackend().mem_read(i, 1);
        if (data[0] != oneByte[0]) {
            continue;
        }

        if (Arrays.equals(data, emulator.getBackend().mem_read(i, data.length)))
    {

```

```

        pointers.add(UnidbgPointer.pointer(emulator, i));
        i += (data.length - 1);
    }
}
return pointers;
}

```

值得一提的是，本节的内容也可用 [LIEF Patch](#) 二进制文件实现。

七、Hook时机过晚问题

上文中，Hook代码都位于 **SO加载后，执行JNI_OnLoad之前**，和如下Frida代码Spawn注入进程等价。

```

var android_dlopen_ext = Module.findExportByName(null, "android_dlopen_ext");
if (android_dlopen_ext != null) {
    Interceptor.attach(android_dlopen_ext, {
        onEnter: function (args) {
            this.hook = false;
            var soName = args[0].readCString();
            if (soName.indexOf("libhookinunidbg.so") !== -1) {
                this.hook = true;
            }
        },
        onLeave: function (retval) {
            if (this.hook) {
                this.hook = false;
                // your code
            }
        }
    });
}

```

而Unidbg Hook代码位于JNI_OnLoad后时，和如下Frida代码Spawn注入进程等价

```

var android_dlopen_ext = Module.findExportByName(null, "android_dlopen_ext");
if (android_dlopen_ext != null) {
    Interceptor.attach(android_dlopen_ext, {
        onEnter: function (args) {
            this.hook = false;
            var soName = args[0].readCString();
            if (soName.indexOf("libhookinunidbg.so") !== -1) {
                this.hook = true;
            }
        },
        onLeave: function (retval) {
            if (this.hook) {
                this.hook = false;
                var jniOnload =
Module.findExportByName("libhookinunidbg.so", "JNI_OnLoad");
                if(jniOnload != null){
                    Interceptor.attach(jniOnload, {
                        onEnter:function(args){
                            console.log("Enter libkwsgmain JNIOnLoad")
                        },
                        onLeave:function(retval){
                            console.log("After libkwsgmain JNIOnLoad");
                        }
                    });
                }
            }
        }
    });
}

```


但如果`.init`和`.init_array`段存在代码逻辑（`init`→`init_array`→`JNIOncLoad`），我们想捕获这个时机，那么上述的时机点都太晚了，这种情况下就需要将Hook时机点提前到`init`执行前。在Frida中，为了实现这一点，通常做法是Hook Linker中的`call_function`或`call_constructor`函数。

以我们的demo hookInUnidbg为例，其中init段里就有如下逻辑，比较两个字符串的大小。

```
// 编译生成后在.init段 [名字不可更改]
extern "C" void _init(void) {
    char str1[15];
    char str2[15];
    int ret;

    strcpy(str1, "abcdef");
    strcpy(str2, "ABCDEF");

    ret = strcmp(str1, str2);

    if(ret < 0)
    {
        LOGI("str1 小于 str2");
    }
    else if(ret > 0)
    {
        LOGI("str1 大于 str2");
    }
    else
    {
        LOGI("str1 等于 str2");
    }
}
```

当前显示**str1 大于 str2**，我们的Hook目标是让其显示 **str1 小于 str2**。如果还想之前那样，在JNIOncLoad之前下断，是断不下来的，因为时机太晚了，Unidbg中可以使用下面几个办法。

1.提前加载libc

提前加载libc，然后hook strcmp函数，修改其返回值为-1是一个办法。如下是完整代码，提供了 Console Debugger 以及 HookZz 两个版本。

```
package com.tutorial;
```

```

import com.github.unidbg.AndroidEmulator;
import com.github.unidbg.Emulator;
import com.github.unidbg.Module;
import com.github.unidbg.arm.context.RegisterContext;
import com.github.unidbg.debugger.BreakPointCallback;
import com.github.unidbg.hook.hookzz.*;
import com.github.unidbg.linux.android.AndroidEmulatorBuilder;
import com.github.unidbg.linux.android.AndroidResolver;
import com.github.unidbg.linux.android.dvm.*;
import com.github.unidbg.memory.Memory;
import unicorn.ArmConst;
import java.io.File;

public class hookInUnidbg {
    private final AndroidEmulator emulator;
    private final VM vm;
    private final Module module;
    private final Module moduleLibc;

    hookInUnidbg() {

        // 创建模拟器实例
        emulator = AndroidEmulatorBuilder.for32Bit().build();

        // 模拟器的内存操作接口
        final Memory memory = emulator.getMemory();
        // 设置系统类库解析
        memory.setLibraryResolver(new AndroidResolver(23));
        // 创建Android虚拟机
        vm = emulator.createDalvikVM(new File("unidbg-
android/src/test/resources/tutorial/hookinunidbg.apk"));

        // 先加载libc.so
        DalvikModule dmLibc = vm.loadLibrary(new File("unidbg-
android/src/main/resources/android/sdk23/lib/libc.so"), true);
        moduleLibc = dmLibc.getModule();

        // hook
        hookStrcmpByUnicorn();
        // 或者
        // hookStrcmpByHookZz();

        // 加载so到虚拟内存
        DalvikModule dm = vm.loadLibrary("hookinunidbg", true);
        // 加载好的 libhookinunidbg.so对应为一个模块
        module = dm.getModule();

        // 执行JNI_OnLoad (如果有的话)
        dm.callJNI_OnLoad(emulator);
    }

    public void call(){
        DvmClass dvmClass =
vm.resolveClass("com/example/hookinunidbg/MainActivity");
        String methodSign = "call()V";
        DvmObject<?> dvmObject = dvmClass.newObject(null);
        dvmObject.callJniMethodObject(emulator, methodSign);
    }
}

```

```

    }

    public static void main(String[] args) {
        hookInUnidbg mydemo = new hookInUnidbg();
        mydemo.call();
    }

    public void hookStrcmpByUnicorn(){

        emulator.attach().addBreakPoint(moduleLibc.findSymbolByName("strcmp").getAddress()
s(), new BreakPointCallback() {
            @Override
            public boolean onHit(Emulator<?> emulator, long address) {
                RegisterContext registerContext = emulator.getContext();
                String arg1 = registerContext.getPointerArg(0).getString(0);

                emulator.attach().addBreakPoint(registerContext.getLRPointer().peer, new
BreakPointCallback() {
                    @Override
                    public boolean onHit(Emulator<?> emulator, long address) {
                        if(arg1.equals("abcdef")){

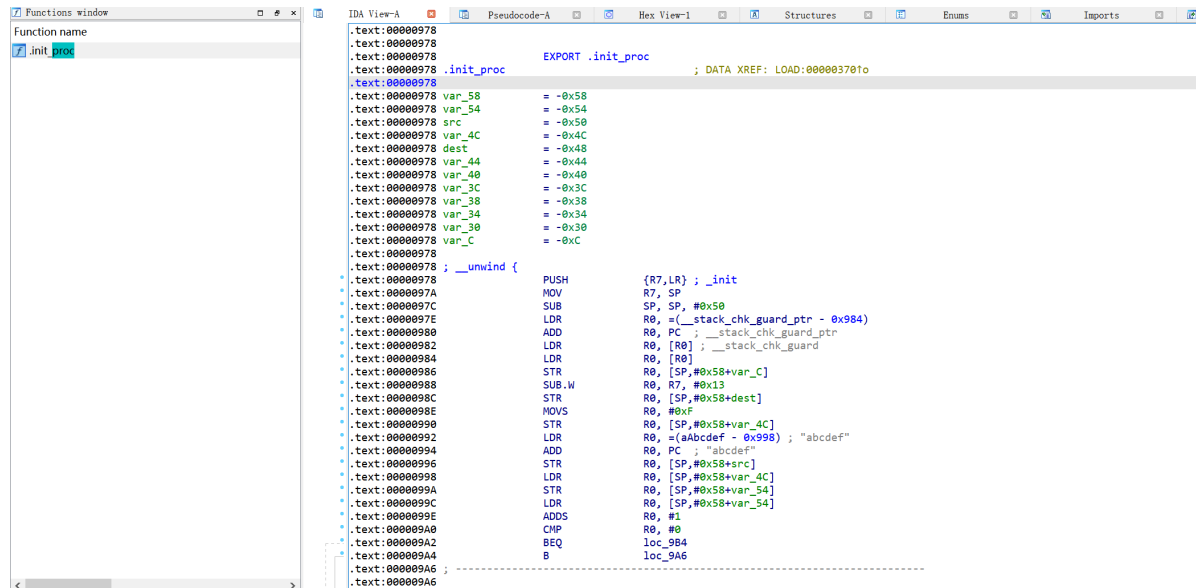
                            emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_R0, -1);
                        }
                        return true;
                    }
                });
                return true;
            }
        });
    }

    public void hookStrcmpByHookZz(){
        IHookZz hookZz = HookZz.getInstance(emulator); // 加载HookZz, 支持inline
hook
        hookZz.enable_arm_arm64_b_branch(); // 测试enable_arm_arm64_b_branch, 可有
可无
        hookZz.wrap(moduleLibc.findSymbolByName("strcmp"), new
wrapCallback<HookZzArm32RegisterContext>() {
            String arg1;
            @Override
            public void preCall(Emulator<?> emulator, HookZzArm32RegisterContext
ctx, HookEntryInfo info) {
                arg1 = ctx.getPointerArg(0).getString(0);
            }
            @Override
            public void postCall(Emulator<?> emulator,
HookZzArm32RegisterContext ctx, HookEntryInfo info) {
                if(arg1.equals("abcdef")){
                    ctx.setR0(-1);
                }
            }
        });
        hookZz.disable_arm_arm64_b_branch();
    }
}

```

```
}
```

但如果想hook的目标函数不是libc里的函数，就没效果了。比如想在0x978下个断点。



2.固定地址下断点

这是最常用也最方便的方式，但只有Unicorn引擎下可以使用。

通过 `vm.loadLibrary` 加载的第一个用户SO，其基地址是0x40000000，因此可以在IDA中看函数偏移，通过绝对地址Console Debugger Hook。

```
package com.tutorial;

import com.github.unidbg.AndroidEmulator;
import com.github.unidbg.Emulator;
import com.github.unidbg.Module;
import com.github.unidbg.arm.context.RegisterContext;
import com.github.unidbg.debugger.BreakPointCallback;
import com.github.unidbg.hook.hookzz.*;
import com.github.unidbg.linux.android.AndroidEmulatorBuilder;
import com.github.unidbg.linux.android.AndroidResolver;
import com.github.unidbg.linux.android.dvm.*;
import com.github.unidbg.memory.Memory;
import unicorn.ArmConst;
import java.io.File;

public class hookInUnidbg {
    private final AndroidEmulator emulator;
    private final VM vm;
    private final Module module;
    private Module moduleLibc;

    hookInUnidbg() {

        // 创建模拟器实例
        emulator = AndroidEmulatorBuilder.for32Bit().build();

        // 模拟器的内存操作接口
        final Memory memory = emulator.getMemory();
```

```

// 设置系统类库解析
memory.setLibraryResolver(new AndroidResolver(23));
// 创建Android虚拟机
vm = emulator.createDalvikVM(new File("unidbg-
android/src/test/resources/tutorial/hookinunidbg.apk"));

emulator.attach().addBreakPoint(0x40000000 + 0x978);

// 加载so到虚拟内存
DalvikModule dm = vm.loadLibrary("hookinunidbg", true);
// 加载好的 libhookinunidbg.so对应为一个模块
module = dm.getModule();

// 执行JNI_OnLoad (如果有的话)
dm.callJNI_OnLoad(emulator);
}

public void call(){
    DvmClass dvmClass =
vm.resolveClass("com/example/hookinunidbg/MainActivity");
    String methodSign = "call()V";
    DvmObject<?> dvmObject = dvmClass.newObject(null);
    dvmObject.callJniMethodObject(emulator, methodSign);
}

public static void main(String[] args) {
    hookInUnidbg mydemo = new hookInUnidbg();
    mydemo.call();
}
}

```

```

Run: hookinUnidbg x
C:\Users\13352\.jdk\openjdk-16.0.2\bin\java.exe ...
debugger break at: 0x40000978
>>> r0=0x0 r1=0x0 r2=0x0 r3=0x2 r4=0x0 r5=0x0 r6=0x0 r7=0x0 r8=0x0 sb=0x0 sl=0x0 fp=0x0 ip=0x4011a5e0
>>> SP=0xbffff758 LR=unidbg@0xfffff0000 PC=RX@0x40000978[libhookinunidbg.so]0x978 cpsr: N=0, Z=1, C=0, V=0, T=1, mode=0b10000
>>> d0=0x0(0.0) d1=0x3933312032203120(3.696225012140986E-33) d2=0x3220302034203736(3.0022298612178987E-67) d3=0x3436333832203235(3.53667
>>> d8=0x0(0.0) d9=0x0(0.0) d10=0x0(0.0) d11=0x0(0.0) d12=0x0(0.0) d13=0x0(0.0) d14=0x0(0.0) d15=0x0(0.0)
=> *[libhookinunidbg.so]*[0x00979]*[* 80 b5 ]*0x40000978:*push {r7, lr}
[libhookinunidbg.so] [0x0097b] [ 6f 46 ] 0x4000097a: mov r7, sp
[libhookinunidbg.so] [0x0097d] [ 94 b0 ] 0x4000097c: sub sp, #0x50
[libhookinunidbg.so] [0x0097f] [ 39 48 ] 0x4000097e: ldr r0, [pc, #0xe4]
[libhookinunidbg.so] [0x00981] [ 78 44 ] 0x40000980: add r0, pc
[libhookinunidbg.so] [0x00983] [ 00 68 ] 0x40000982: ldr r0, [r0]
[libhookinunidbg.so] [0x00985] [ 00 68 ] 0x40000984: ldr r0, [r0]
[libhookinunidbg.so] [0x00987] [ 13 90 ] 0x40000986: str r0, [sp, #0x4c]
[libhookinunidbg.so] [0x00989] [ a7 f1 13 00 ] 0x40000988: sub.w r0, r7, #0x13
[libhookinunidbg.so] [0x0098d] [ 04 90 ] 0x4000098c: str r0, [sp, #0x10]
[libhookinunidbg.so] [0x0098f] [ 0f 20 ] 0x4000098e: movs r0, #0xf
[libhookinunidbg.so] [0x00991] [ 03 90 ] 0x40000990: str r0, [sp, #0xc]
[libhookinunidbg.so] [0x00993] [ 35 48 ] 0x40000992: ldr r0, [pc, #0xd4]
[libhookinunidbg.so] [0x00995] [ 78 44 ] 0x40000994: add r0, pc
[libhookinunidbg.so] [0x00997] [ 02 90 ] 0x40000996: str r0, [sp, #8]
[libhookinunidbg.so] [0x00999] [ 03 98 ] 0x40000998: ldr r0, [sp, #0xc]

```

如果加载了多个用户SO，可以先运行一遍代码，确认目标SO的基地址（Unidbg中不存在地址随机化，目标函数每次地址都固定。）然后在loadLibrary前Hook该地址，即可保证Hook不遗漏。

3.使用Unidbg提供的模块监听器

实现自己的模块监听器

```
package com.tutorial;

import com.github.unidbg.Emulator;
import com.github.unidbg.Module;
import com.github.unidbg.ModuleListener;
import com.github.unidbg.arm.context.RegisterContext;
import com.github.unidbg.hook.hookzz.HookEntryInfo;
import com.github.unidbg.hook.hookzz.HookZz;
import com.github.unidbg.hook.hookzz.InstrumentCallback;

public class MyModuleListener implements ModuleListener {
    private HookZz hook;

    @Override
    public void onLoaded(Emulator<?> emulator, Module module) {
        // 提前加载Hook框架
        if(module.name.equals("libc.so")){
            hook = HookZz.getInstance(emulator);
        }

        // 在目标函数中Hook
        if(module.name.equals("libhookinunidbg.so")){
            hook.instrument(module.base + 0x978 + 1, new
InstrumentCallback<RegisterContext>() {
                @Override
                public void dbiCall(Emulator<?> emulator, RegisterContext ctx,
HookEntryInfo info) {
                    System.out.println(ctx.getIntArg(0));
                }
            });
        }
    }
}
```

通过 `memory.addModuleListener` 绑定。

```
package com.tutorial;

import com.github.unidbg.AndroidEmulator;
import com.github.unidbg.Module;
import com.github.unidbg.linux.android.AndroidEmulatorBuilder;
import com.github.unidbg.linux.android.AndroidResolver;
import com.github.unidbg.linux.android.dvm.*;
import com.github.unidbg.memory.Memory;
import java.io.File;

public class hookInUnidbg{
    private final AndroidEmulator emulator;
    private final VM vm;

    hookInUnidbg() {
```

```

// 创建模拟器实例
emulator = AndroidEmulatorBuilder.for32Bit().build();

// 模拟器的内存操作接口
final Memory memory = emulator.getMemory();

// 添加模块加载监听器
memory.addModuleListener(new MyModuleListener());

// 设置系统类库解析
memory.setLibraryResolver(new AndroidResolver(23));
// 创建Android虚拟机
vm = emulator.createDalvikVM(new File("unidbg-
android/src/test/resources/tutorial/hookinunidbg.apk"));

// 加载so到虚拟内存
DalvikModule dm = vm.loadLibrary("hookinunidbg", true);
// 加载好的 libhookinunidbg.so对应为一个模块
Module module = dm.getModule();

// 执行JNI_OnLoad（如果有的话）
dm.callJNI_OnLoad(emulator);
}

public void call(){
    DvmClass dvmClass =
vm.resolveClass("com/example/hookinunidbg/MainActivity");
    String methodSign = "call()v";
    DvmObject<?> dvmObject = dvmClass.newObject(null);
    dvmObject.callJniMethodObject(emulator, methodSign);
}

public static void main(String[] args) {
    hookInUnidbg mydemo = new hookInUnidbg();
    mydemo.call();
}
}

```

每种方法都有对应使用场景，按需使用。除此之外也可以修改Unidbg源码，在 `callInitFunction` 函数前添加自己的逻辑。

八、条件断点

在算法分析时，条件断点可以减少干扰信息。以 `strcmp` 为例，整个进程的所有模块都可能调用 `strcmp` 函数。

1.限于某SO

I Frida

```
Interceptor.attach(
    Module.findExportByName("libc.so", "strcmp"), {
        onEnter: function(args) {
            var moduleName =
                Process.getModuleByAddress(this.returnAddress).name;
            console.log("strcmp arg1:"+args[0].readCString())
            // 可以根据moduleName筛选打印
            console.log("call from :"+moduleName)
        },
        onLeave: function(ret) {
        }
    }
);
```

II Unidbg

```
public void hookstrcmp(){
    long address = module.findSymbolByName("strcmp").getAddress();
    emulator.attach().addBreakPoint(address, new BreakPointCallback() {
        @Override
        public boolean onHit(Emulator<?> emulator, long address) {
            RegisterContext registerContext = emulator.getContext();
            String arg1 = registerContext.getPointerArg(0).getString(0);
            String moduleName =
                emulator.getMemory().findModuleByAddress(registerContext.getLRPointer().peer).name;

            if(moduleName.equals("libhookinunidbg.so")){
                System.out.println("strcmp arg1:"+arg1);
            }
            return true;
        }
    });
}
```

在Unidbg中，“Hook限于目标SO内”的使用场景可能不那么多，因为Unidbg虚拟进程里只有我们的目标SO在活跃，但Android系统中，目标进程里可能有数十个模块。当我们想hook strlen 来窥探目标SO里的字符串操作时，如果不加筛选，会被其他模块的大量调用干扰。或者我们想替换pthread_create函数，观察或者阻止子线程创建的时候，也会发现libart等模块也在创建子线程，除此之外替换时间等操作也都要注意这个问题。

```
// 替换目标函数里对pthread_create的访问
function hookPthreadCreate(){
    var p_pthread_create = Module.findExportByName("libc.so", "pthread_create");
    var pthread_create = new NativeFunction( p_pthread_create, "int",
        ["pointer", "pointer", "pointer", "pointer"]);
    Interceptor.replace( p_pthread_create, new NativeCallback(function (ptr0,
        ptr1, ptr2, ptr3) {
            var moduleName = Process.getModuleByAddress(this.returnAddress).name;
            if (moduleName === "target.so") {
                console.log("loading fake pthread_create");
                return -1;
            }
        }
    ));
}
```



```

        } else {
            return pthread_create(ptr0, ptr1, ptr2, ptr3);
        }

    }, "int", ["pointer", "pointer", "pointer", "pointer"]));
}

```

2. 限于某函数

比如某个函数在SO中被大量使用，现在只想分析这个函数在函数A中的使用。

I Frida

```

var show = false;
Interceptor.attach(
    Module.findExportByName("libc.so", "strcmp"), {
        onEnter: function(args) {
            if(show){
                console.log("strcmp arg1:"+args[0].readCString());
            }
        },
        onLeave: function(ret) {

        }
    }
);

Interceptor.attach(
    Module.findExportByName("libhookinunidbg.so", "targetfunction"), {
        onEnter: function(args) {
            show = true;
        },
        onLeave: function(ret) {
            show = false;
        }
    }
)

```

II Unidbg

```

// 早先声明全局变量 public boolean show = false;

public void hookstrcmp(){

    emulator.attach().addBreakPoint(module.findSymbolByName("targetfunction").getAddress(), new BreakPointCallback() {
        @Override
        public boolean onHit(Emulator<?> emulator, long address) {
            RegisterContext registerContext = emulator.getContext();

            show = true;
            emulator.attach().addBreakPoint(registerContext.getLRPointer().peer,
            new BreakPointCallback() {
                @Override

```

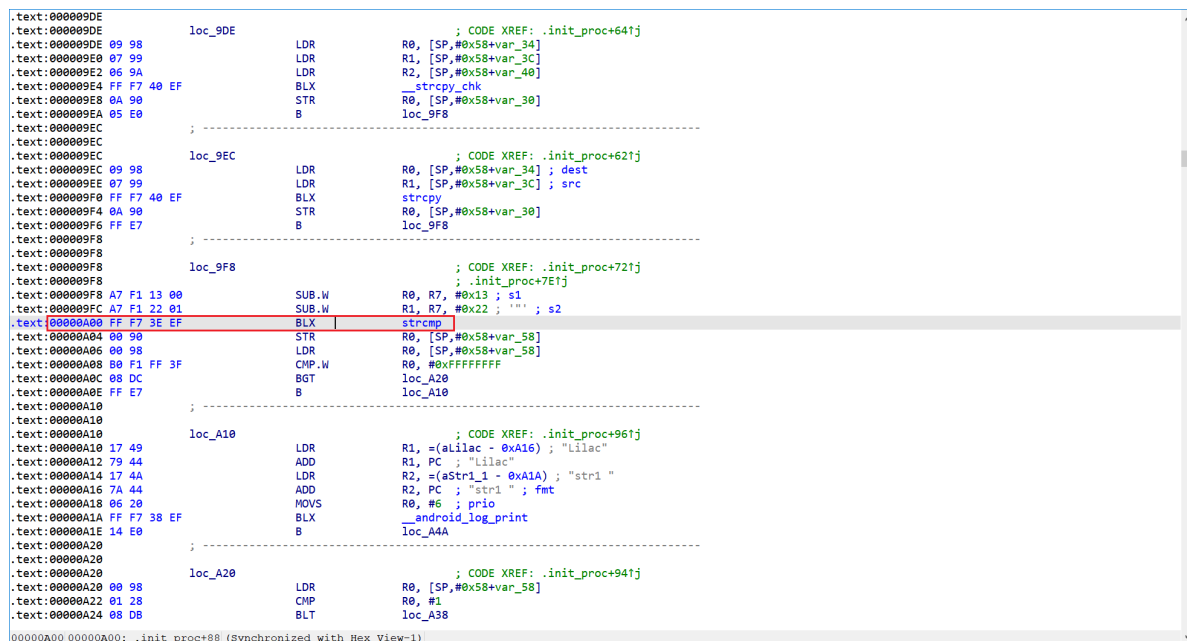
```

        public boolean onHit(Emulator<?> emulator, long address) {
            show = false;
            return true;
        }
    });
    return true;
}
});
}

emulator.attach().addBreakPoint(module.findSymbolByName("strcmp").getAddress(),
new BreakPointCallback() {
    @Override
    public boolean onHit(Emulator<?> emulator, long address) {
        RegisterContext registerContext = emulator.getContext();
        String arg1 = registerContext.getPointerArg(0).getString(0);
        if(show){
            System.out.println("strcmp arg1:"+arg1);
        }
        return true;
    }
});
}
}

```

3.限定于某处



比如上图，只关注0xA00处发生的strcmp。一个办法是hook strcmp函数，只在R寄存器=module.base + 0xA00 + 4 + 1时打印输出。

另一个办法是Console Debugger ,也很方便。

```

emulator.attach().addBreakPoint(module, 0xA00);
emulator.attach().addBreakPoint(module, 0xA04);

```

一定要掌握这些知识，并做到灵活变通。在实际使用中，诸如“A hook生效后打印B函数的输出”这样的需求是很常见的，否则每个函数都打印成百上千行会迷人眼，干扰对关键信息的寻找。

九、系统调用拦截——以时间为例

这里说的系统调用拦截，并不是要对系统调用进行Hook，比如 [frida - syscall - interceptor](#) 这样，系统调用全部是Unidbg自己实现的，日志一开就能看，显然也没有Hook的必要。Unidbg的**系统调用拦截**是为了替换系统调用，修改Unidbg中系统调用的实现。

有两个问题需要解释

- 为什么要修改系统调用？
Unidbg中部分系统调用没实现或者没实现好，以及有时候想要固定其输出，比如获取时间的系统调用，这些需求需要我们修复或修改Unidbg中系统调用的实现。
- 为什么不直接修改Unidbg源码
1是灵活性较差，2是我们的实现或修改并不是完美的，直接改Unidbg源码是对运行环境的污染，影响其他项目。

在分析算法时，输入不变的前提下，如果输出在不停变化，会干扰算法分析，这种情况的一大来源是时间戳参与了运算。在Frida中，为了控制这种干扰因素，常常会Hook libc的gettimeofday这个时间获取函数。

1.Frida

hook time

```
var time = Module.findExportByName(null, "time");
if (time != null) {
    Interceptor.attach(time, {
        onEnter: function (args) {

        },
        onLeave: function (retval) {
            // time返回秒级时间戳，修改返回值为100
            retval.replace(100);
        }
    })
}
```

hook gettimeofday

```
function hook_gettimeofday() {
    var addr_gettimeofday = Module.findExportByName(null, "gettimeofday");
    var gettimeofday = new NativeFunction(addr_gettimeofday, "int", ["pointer", "pointer"]);

    Interceptor.replace(addr_gettimeofday, new NativeCallback(function (ptr_tz, ptr_tzp) {

        var result = gettimeofday(ptr_tz, ptr_tzp);
        if (result == 0) {
            console.log("hook gettimeofday:", ptr_tz, ptr_tzp, result);
            var t = new Int32Array(ArrayBuffer.wrap(ptr_tz, 8));
            t[0] = 0xAAAA;
            t[1] = 0xBBBB;
            console.log(hexdump(ptr_tz));
        }
        return result;
    }, "int", ["pointer", "pointer"]));
}
```

但Frida做这件事并不容易做圆满，单是libc.so，就有time、gettimeofday、clock_gettime、clock 这四个库函数可以获取时间戳，而且样本可以通过内联汇编使用系统调用，获取时间戳。

2.Unidbg

Unidbg中可以更方便、更大范围的固定时间，不必像Frida那般。time和gettimeofday库函数基于gettimeofday这个系统调用，clock_gettime和clock基于clock_gettime系统调用。所以只要在Unidbg中固定gettimeofday和clock_gettime这两个系统调用获取的时间戳，就可以一劳永逸。

首先实现时间相关的系统调用处理器，其中的System.currentTimeMillis()和System.nanoTime()改成定数。

```
package com.tutorial;

import com.github.unidbg.Emulator;
import com.github.unidbg.linux.ARM32SyscallHandler;
import com.github.unidbg.memory.SvcMemory;
import com.github.unidbg.pointer.UnidbgPointer;
import com.github.unidbg.unix.struct.TimeVal32;
import com.github.unidbg.unix.struct.TimeZone;
import com.sun.jna.Pointer;
import unicorn.ArmConst;

import java.util.Calendar;

public class TimeSyscallHandler extends ARM32SyscallHandler {
    public TimeSyscallHandler(SvcMemory svcMemory) {
        super(svcMemory);
    }

    @Override
    protected boolean handleUnknownSyscall(Emulator emulator, int NR) {
        switch (NR) {
            case 78:
                // gettimeofday
                mygettimeofday(emulator);
                return true;
            case 263:
                // clock_gettime
                myclock_gettime(emulator);
                return true;
        }

        return super.handleUnknownSyscall(emulator, NR);
    }

    private void mygettimeofday(Emulator<?> emulator) {
        Pointer tv = UnidbgPointer.register(emulator, ArmConst.UC_ARM_REG_R0);
        Pointer tz = UnidbgPointer.register(emulator, ArmConst.UC_ARM_REG_R1);
        emulator.getBackend().reg_write(ArmConst.UC_ARM_REG_R0,
mygettimeofday(tv, tz));
    };

    private int mygettimeofday(Pointer tv, Pointer tz) {
        long currentTimeMillis = System.currentTimeMillis();
```

```

        long tv_sec = currentTimeMillis / 1000;
        long tv_usec = (currentTimeMillis % 1000) * 1000;

        TimeVal32 timeVal = new TimeVal32(tv);
        timeVal.tv_sec = (int) tv_sec;
        timeVal.tv_usec = (int) tv_usec;
        timeVal.pack();

        if (tz != null) {
            Calendar calendar = Calendar.getInstance();
            int tz_minuteswest = -(calendar.get(Calendar.ZONE_OFFSET) +
calendar.get(Calendar.DST_OFFSET)) / (60 * 1000);
            TimeZone timeZone = new TimeZone(tz);
            timeZone.tz_minuteswest = tz_minuteswest;
            timeZone.tz_dsttime = 0;
            timeZone.pack();
        }
        return 0;
    }

    private static final int CLOCK_REALTIME = 0;
    private static final int CLOCK_MONOTONIC = 1;
    private static final int CLOCK_THREAD_CPUTIME_ID = 3;
    private static final int CLOCK_MONOTONIC_RAW = 4;
    private static final int CLOCK_MONOTONIC_COARSE = 6;
    private static final int CLOCK_BOOTTIME = 7;
    private final long nanoTime = System.nanoTime();

    private int myclock_gettime(Emulator<?> emulator) {
        int clk_id =
emulator.getBackend().reg_read(ArmConst.UC_ARM_REG_R0).intValue();
        Pointer tp = UnidbgPointer.register(emulator, ArmConst.UC_ARM_REG_R1);
        long offset = clk_id == CLOCK_REALTIME ? System.currentTimeMillis() *
1000000L : System.nanoTime() - nanoTime;
        long tv_sec = offset / 1000000000L;
        long tv_nsec = offset % 1000000000L;

        switch (clk_id) {
            case CLOCK_REALTIME:
            case CLOCK_MONOTONIC:
            case CLOCK_MONOTONIC_RAW:
            case CLOCK_MONOTONIC_COARSE:
            case CLOCK_BOOTTIME:
                tp.setInt(0, (int) tv_sec);
                tp.setInt(4, (int) tv_nsec);
                return 0;
            case CLOCK_THREAD_CPUTIME_ID:
                tp.setInt(0, 0);
                tp.setInt(4, 1);
                return 0;
        }
        throw new UnsupportedOperationException("clk_id=" + clk_id);
    }
}

```

在自己的模拟器上使用它，原先模拟器创建是这么一句

```
// 创建模拟器实例
emulator = AndroidEmulatorBuilder.for32Bit().build();
```

修改如下

```
// 创建模拟器实例
AndroidEmulatorBuilder builder = new AndroidEmulatorBuilder(false) {
    public AndroidEmulator build() {
        return new AndroidARMEulator(processName, rootDir,
            backendFactories) {
            @Override
            protected UnixSyscallHandler<AndroidFileIO>
            createSyscallHandler(SvcMemory svcMemory) {
                return new TimesyscallHandler(svcMemory);
            }
        };
    }
};

emulator = builder.build();
```

十、Hook 检测

Anti Unidbg的方法浩如烟海，但事实上几乎没有主动Anti Unidbg的样本，有两方面原因

- Unidbg 自身的多个重大弱点没有解决，比如多线程和信号机制尚未实现。
- Unidbg 普及率和推广度还不高。

所以本节专注于Hook 检测。

1.检测第三方Hook框架

基于其Hook实现原理，可以对应检测。

I Inline Hook

以我熟悉的inline Hook 检测为例，inline Hook 需要修改Hook处的前几个字节，跳转到自己的地方实现逻辑，最后再跳转回来。那么就有两类思路实现检测，首先开辟一个检测线程，对关键函数做如下二选一循环操作

- 函数开头前几个字节是否被篡改
- 函数体是否完整未被修改，常使用crc32校验，为什么不用md5或其他哈希函数？因为crc32极快，性能影响小，碰撞率又在可接受的范围内

相关项目：[check fish inline hook](#)

II Got Hook

相关项目：[SliverBullet5563/CheckGotHook: 检测got hook \(使用xhook测试\)](#)

2.检测Unicorn Based Hook

Unicorn Hook 似乎不可检测，但Unicorn也是可检测的。在星球的Anti-Unidbg系列，就提到过一种检测方式。在Android系统中，只支持对四字节对齐的内存地址做读写操作，所以通过内联汇编尝试向SP+1的位置做读写，在真机上会导致App崩溃，而Unidbg模拟执行不会出任何问题。当然，我们并不希望App崩溃，所以需要在代码中实现自己的信号处理函数，当此处发生异常时，信号处理函数接收信号并做出某种处理，因为Unidbg中程序不会异常，所以也不会走到信号处理函数，这里面可以设计形成差异。

除此之外，Unicorn下断点调试或者做指令追踪时，必然会导致函数运行时间超出常理，基于运行时间的反调试策略也可行。

十一、Unidbg Trace 五件套

基于Frida 存在许多trace 方案，比如用于 trace JNI函数的 [JNITrace](#)，用于trace Java调用的 [ZenTrace](#)、[r0tracer](#)，又或者是官方的多功能 trace 工具 [frida-trace](#)，用于指令级 trace 的 [Frida Stalker](#)，又或者是trace SO中所有函数的 [trace natives](#)，以及Linux上著名的[strace](#) 或者 基于Frida 的 [frida-syscall-interceptor](#)，用于 trace 系统调用。

在Unidbg 上，上述的大部分trace，只需要调整日志等级就能实现。我们这里所讲的trace，聚焦于如何让使用者对代码执行流有更强的掌控。

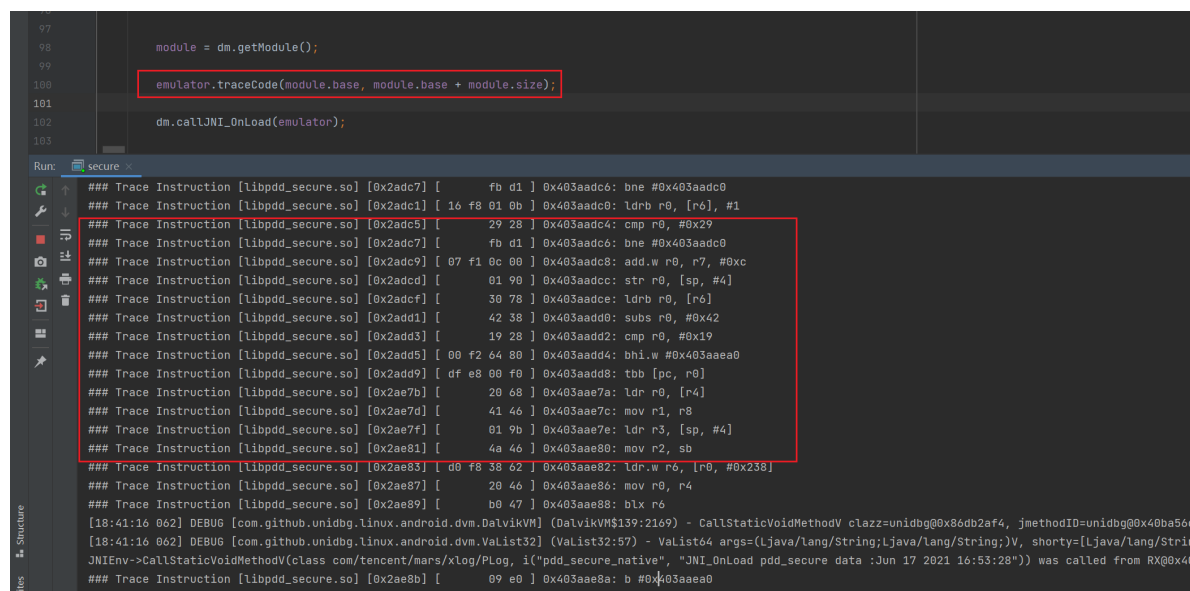
1.Instruction tracing

令追踪包括两部分

- 记录每条指令的执行，打印地址、机器码、汇编等信息
- 打印每条指令相关的寄存器值

Unidbg 基于 Unicorn CodeHook 封装了指令追踪，方法和效果如下

```
/**
 * trace instruction
 * note: low performance
 */
TraceHook traceCode();
TraceHook traceCode(long begin, long end);
TraceHook traceCode(long begin, long end, TraceCodeListener listener);
```

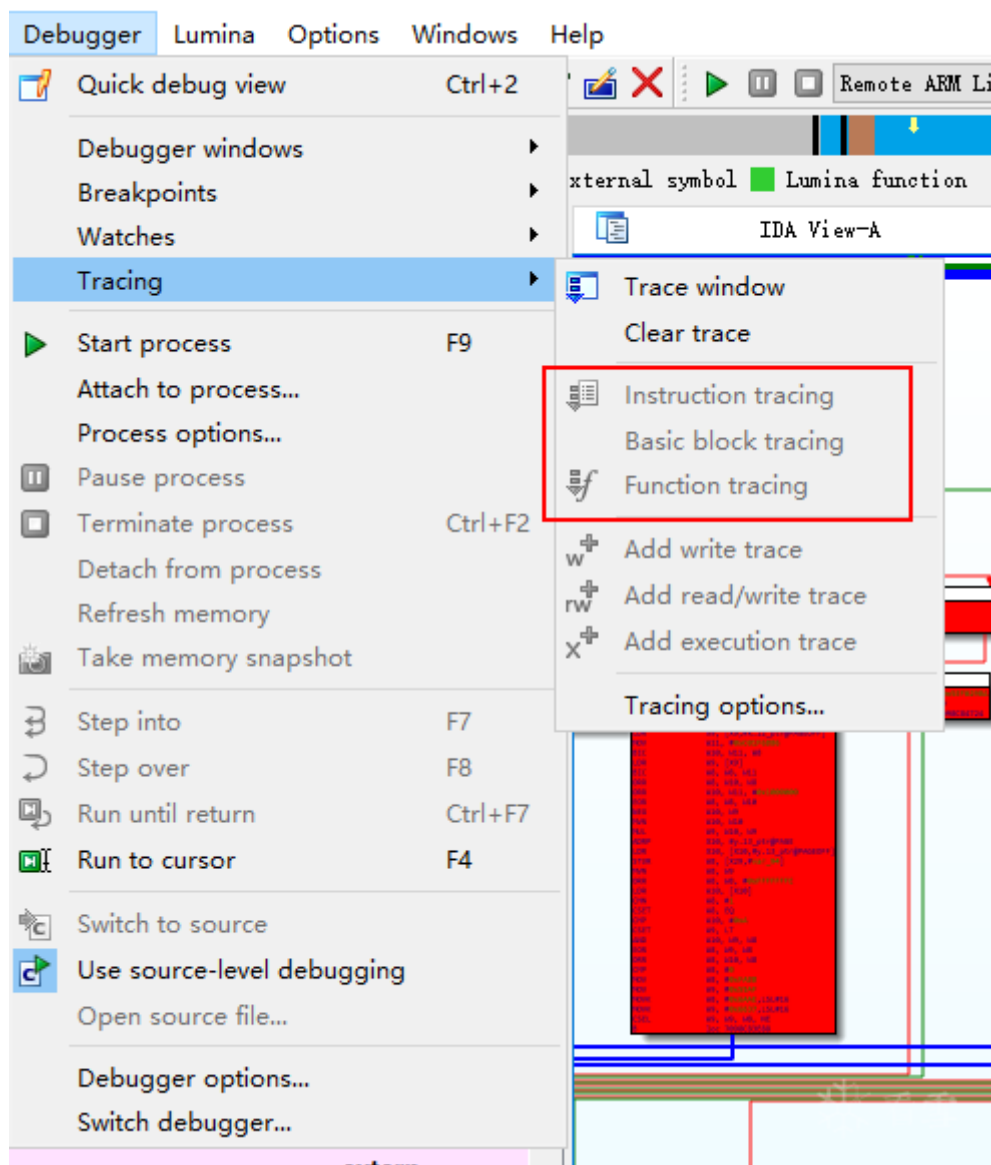


Unidbg的指令追踪，在第一部分的工作做得很好，采用 模块名+相对偏移+机器码+绝对地址+汇编 的展示形式，但美中不足的是，它并没有做第二部分的工作，可以使用如下的脚本在Unidbg中实现完善的指令追踪，其原理也是实现了一个自己的codehook。

[增加trace的部分 by dqzg12300 · Pull Request #214](#)

2.Function Tracing

指令Trace是最细粒度的Trace，优点是细，缺点是动辄数百上千万行，让人迷失其中。函数粒度的trace则不然，粗糙但容易理解全貌，在算法还原的一些场景中会起到帮助。在IDA Debug中，即可选择函数追踪来记录函数调用，包括了有符号函数以及IDA识别并命名为Sub_addr的子函数。



在Frida上，可以使用 [trace_natives](#) 对一个SO中的全部函数进行Trace，并形成如下调用图。


```

/* TID 0x5d64 */
6226 ms sub_1332d()
6227 ms | sub_13565()
6227 ms | sub_13009()
6227 ms | | sub_7fe9()
6227 ms | | | sub_8459()
6227 ms | | | sub_7ecd()
6228 ms | sub_23079()
6228 ms | | sub_a8e5()
6228 ms | | | sub_1a6c9()
6228 ms | | | sub_1a6ed()
6228 ms | sub_7ecd()
6228 ms | sub_1255d()
6228 ms | | sub_6891()
6228 ms | | | sub_7e69()
6229 ms | sub_23515()
6229 ms | | sub_60ad()
6229 ms | | sub_21b8b()
6229 ms | | | sub_6c2d()
6229 ms | | sub_60ad()
6229 ms | | sub_21b8b()
6229 ms | | | sub_6c2d()
6229 ms | | | sub_227e1()
6229 ms | | | | sub_1a6c9()
6230 ms | | | | sub_60ad()
6230 ms | | | | sub_6c2d()
6230 ms | | | | sub_1a6ed()
6230 ms | | | | sub_24439()
6230 ms | | | | sub_7fe9()
6230 ms | | | | | sub_8459()
6230 ms | | | | | sub_7ecd()
6230 ms | | | | sub_96dd()

```

Unidbg中可以做到这一点吗？不妨看一下 Frida trace_natives 脚本，其中有三个关注点。

- 如何获得一个SO的全部函数列表，就像IDA一样
- 如何Hook函数
- 如何获得调用层级关系，形成树结构

关于问题1，trace_natives怎么解决的？直接编写IDA脚本获取IDA的函数列表

```

def getFunctionList():
    functionlist = ""
    minLength = 10
    maxAddress = ida_ida.inf_get_max_ea()
    for func in idautils.Functions(0, maxAddress):
        if len(list(idautils.FuncItems(func))) > minLength:
            functionName = str(idaapi.ida_funcs.get_func_name(func))
            oneFunction = hex(func) + "!" + functionName + "\t\n"
            functionlist += oneFunction
    return functionlist

```

脚本获取了函数以及对应函数名列表，同时通过minLength过滤较短的函数，至少包含10条汇编指令的函数才会被计入。这么做有两个原因

- 过短的函数可能导致Frida Hook失败（inline hook 原理所致）
- 过短的函数可能是工具函数，调用次数多，但价值不大，让调用图变得臃肿不堪

完整的IDA插件 getFunctions 代码如下

```

import os
import time

import ida_ida
import ida_nalt
import idaapi
import idautils
from idaapi import plugin_t
from idaapi import PLUGIN_PROC
from idaapi import PLUGIN_OK

```

```

def getFunctionList():
    functionlist = ""
    minLength = 10
    maxAddress = ida_ida.inf_get_max_ea()
    for func in idautils.Functions(0, maxAddress):
        if len(list(idautils.FuncItems(func))) > minLength:
            functionName = str(idaapi.ida_funcs.get_func_name(func))
            oneFunction = hex(func) + "!" + functionName + "\t\n"
            functionlist += oneFunction
    return functionlist

# 获取SO文件名和路径
def getSoPathAndName():
    fullpath = ida_nalt.get_input_file_path()
    filepath, filename = os.path.split(fullpath)
    return filepath, filename

class getFunctions(plugin_t):
    flags = PLUGIN_PROC
    comment = "getFunctions"
    help = ""
    wanted_name = "getFunctions"
    wanted_hotkey = ""

    def init(self):
        print("getFunctions(v0.1) plugin has been loaded.")
        return PLUGIN_OK

    def run(self, arg):

        so_path, so_name = getSoPathAndName()
        functionlist = getFunctionList()
        script_name = so_name.split(".")[0] + "_functionlist_" +
str(int(time.time())) + ".txt"
        save_path = os.path.join(so_path, script_name)
        with open(save_path, "w", encoding="utf-8") as F:
            F.write(functionlist)
        F.close()
        print(f"location: {save_path}")

    def term(self):
        pass

def PLUGIN_ENTRY():
    return getFunctions()

```

关于问题2：使用Frida Native Hook

关于问题3：Frida的frida-trace自带调用层级关系，所以trace_Natives脚本依赖Frida-trace，展示出了树结构的调用图。分析源码发现，frida-trace 使用了Frida在 Interceptor.attach 环境中的depth 如下代码中的this.depth），depth表示了调用深度。那深度的值哪来的呢？其最终依赖于Frida的栈回溯。

```

Interceptor.attach(Module.getExportByName(null, 'read'), {

```

```

onEnter(args) {
    console.log('Context information:');
    console.log('Context  : ' + JSON.stringify(this.context));
    console.log('Return   : ' + this.returnAddress);
    console.log('ThreadId : ' + this.threadId);
    console.log('Depth    : ' + this.depth);
    console.log('Errornr   : ' + this.err);

    // Save arguments for processing in onLeave.
    this.fd = args[0].toInt32();
    this.buf = args[1];
    this.count = args[2].toInt32();
},
onLeave(result) {
    console.log('-----')
    // Show argument 1 (buf), saved during onEnter.
    const numBytes = result.toInt32();
    if (numBytes > 0) {
        console.log(hexdump(this.buf, { length: numBytes, ansi: true }));
    }
    console.log('Result   : ' + numBytes);
}
})

```

这三个问题能在Unidbg中解决吗？如果能解决，那就有了Unidbg 版的Function Tracing。

首先问题一，只是一个获取函数列表的插件，与使用Frida还是Unidbg无关，构不成问题。我们还可以更进一步思考，trace_Natives 依赖IDA实现对SO 函数的识别，但与此同时也增加了使用的复杂度，而且加壳的SO无法直接识别函数，必须先dump+fix SO，其实还挺折腾人，不如不依赖IDA，换个办法识别函数。ARM中，函数序言常常以 push 指令开始，这可以代表绝大多数函数。配合Unidbg的BlockHook 或者 CodeHook，就可以解析并 Hook 这些函数，问题二也顺带解决了。少部分函数会遗漏，但也无关痛痒。BlockHook 还会提供当前基本块的大小，我们设置对较小的块不予理睬。

接下来就是问题三，栈回溯这块，Unidbg也实现了arm unwind栈回溯，一些情况下有Bug，但总体应该能用。但Unidbg没有提供打印深度的函数，在Unwinder类中添加一个它。

src/main/java/com/github/unidbg/unwind/Unwinder.java

```

public final int depth(){
    int count = 0;
    Frame frame = null;
    while((frame = unw_step(emulator, frame)) != null) {
        if(frame.isFinish()){
            return count;
        }
        count++;
    }
    return count;
}

```

接下来三步骤合一，组装代码

```

PrintStream traceStream = null;
try {
    // 保存文件
    String traceFile = "unidbg-
    android/src/test/resources/app/traceFunctions.txt";
}

```

```

        traceStream = new PrintStream(new FileOutputStream(traceFile), true);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }

    final PrintStream finalTraceStream = traceStream;
    emulator.getBackend().hook_add_new(new BlockHook() {
        @Override
        public void hookBlock(Backend backend, long address, int size, Object user)
        {
            if(size>8){
                Capstone.CsInsn[] insns = emulator.disassemble(address, 4, 0);
                if(insns[0].mnemonic.equals("push")){
                    int level = emulator.getUnwinder().depth();
                    assert finalTraceStream != null;
                    for(int i = 0 ; i < level ; i++){
                        finalTraceStream.print("    |    ");
                    }
                    finalTraceStream.println("    "+"sub_"+Integer.toHexString((int)
(address-module.base))+ " ");
                }
            }

        }

        @Override
        public void onAttach(Unicorn.UnHook unHook) {

        }

        @Override
        public void detach() {

        }
    }, module.base, module.base+module.size, 0);

```

可以发现代码非常的简洁优雅，效果也不错

16					sub_3a6a4
17					sub_3a224
18					sub_3a23c
19					sub_3a294
20					sub_749b4
21					sub_74a90
22					sub_3a7c4
23					sub_3a800
24					sub_3a844
25					sub_3a85c
26					sub_3a8b4
27					sub_10124
28					sub_1018c
29					sub_72c74
30					sub_72c38
31					sub_72c9c
32					sub_72c38
33					sub_10124
34					sub_1018c
35					sub_72c74
36					sub_72c38
37					sub_72c9c
38					sub_72c38
39					sub_3ac14
40					sub_3ac2c
41					sub_3ac84
42					sub_749b4
43					sub_74a90
44					sub_3ba9c
45					sub_10124
46					sub_1018c
47					sub_72c74
48					sub_72c38
49					sub_72c9c
50					sub_72c38
51					sub_57de4
52					sub_57c70
53					sub_77b00

它还有三个明显的缺陷

1是时机过晚，init或者init_arrayz里的内容无法Hook到，可以结合第七节的方法进行完善。

2是有符号的函数也以sub_xxx 显示

3是导入表中的函数无法包括在内，因为原理上讲，只扫描so内push开头的函数。

3.Memory Search

4.Unidbg-FindKey

[Unidbg-FindKey](#)

这是我写的小工具，具体原理见星球，目前支持查找AES-128/AES-256的密钥，理论上还可以将更多的加密算法包括进去，只需要算法满足以下三点：

- 1.程序的预处理（最典型的场景即密钥编排）会产生某个结构
- 2.这个结构是可分辨的
- 3.这个结构可以解码出原始Key

AES完美符合这三点，SM4也很适用，过往的研究表明，DES、RSA、TwoFish、Separnt等加密算法都满足或者部分满足上述三条件。

5.Unidbg-Findcrypt

Findcrypt是老牌经典工具，Unidbg版的Findcrypt是要做啥？解决什么痛点？有三个主要原因

- Findcrypt 处理不了加壳SO
- Findcrypt 中说存在某种加密，但SO中并不一定用，我们的目标函数更不一定用。
- 从Findcrypt提示的常数不一定能找到对应函数，静态交叉分析有局限

// TODO

十二、固定随机数

// TODO

十三、杂项

无需Hook，Unidbg中通过其他方式实现

// TODO