从0开始开发一个InlineHook第一篇

前言

为什么市面上那么多成品的InlineHook,比如Dobby，SandHook，ShadowHook，还要再写一个?

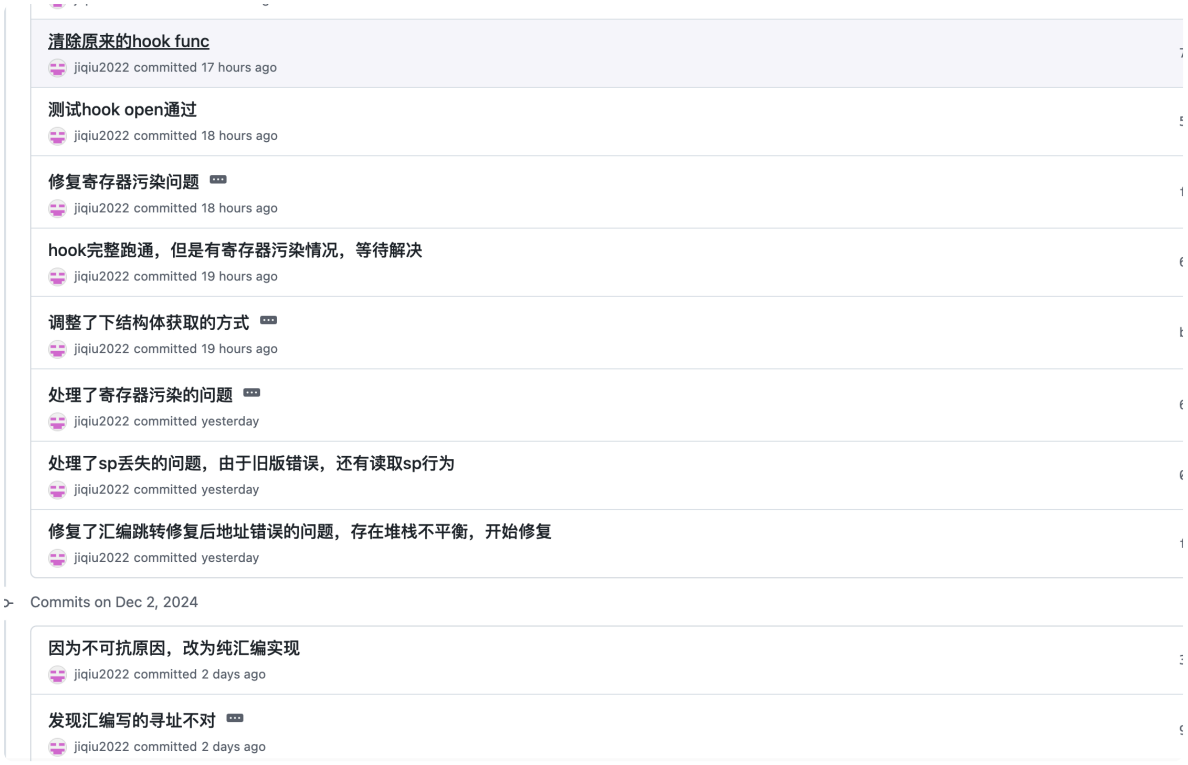
因为这些hook大多数是项目级别的，不带教程讲解，上手难度略高，以及现在对InlineHook检测比较严重，有必要了解原理，从头写一个，知道哪里能检测，哪里需要注意，需要改的点在哪

我写这篇文章主要是记录开发过程，方便后边回顾，以及给一些感兴趣的朋友作为一个入手的点，快速的上手InlineHook的原理。

先放项目地址：https://github.com/jiqiu2022/ReZeroHook

目前有两个分支，主分支只支持单个hook，newhook支持多个hook，都会开展讲解。

提交记录完全，可以看到排除各种坑的提交



> 本项目还不完善，比如完全没写测试用例，只是简单的写了两个函数测试，但是不代表我以后不写
>
> 为什么这么着急写文章，因为也搞了一星期多了，想详细记录一下

致谢

感谢https://github.com/zhuotong/Android_InlineHook提供研究的动力，尽管项目很老了，但是非常清晰，我给修复支持了最新的NDK（之前只能在NDK20）并移植到了Cmake编译（只有Arm64）

感谢https://github.com/bytedance/android-inline-hook提供了指令修复的思路

先来简单了解下什么是InlineHook

什么是InlineHook，简单来说就是给函数（指令）的几条汇编指令备份，然后跳到我们自己的函数逻辑上。

注意：不要联想整个项目，现在只是流程讲解，项目实现会在后面接着讲，结合着流程。

InlineHook的第一步，劫持原来的函数到自己的函数上

举个例子：

比如open函数，我们查看地址

```
libc.so`open:
    0x71002b4910 <+0>:  sub    sp, sp, #0x130
    0x71002b4914 <+4>:  stp    x29, x30, [sp, #0xe0]
    0x71002b4918 <+8>:  add    x29, sp, #0xe0
    0x71002b491c <+12>: str    x28, [sp, #0xf0]
    0x71002b4920 <+16>: stp    x24, x23, [sp, #0x100]
    0x71002b4924 <+20>: stp    x22, x21, [sp, #0x110]
    0x71002b4928 <+24>: stp    x20, x19, [sp, #0x120]
    0x71002b492c <+28>: stp    x5, x6, [x29, #-0x48]
(lldb)
```

前四条汇编指令是这样的，当我们hook后：

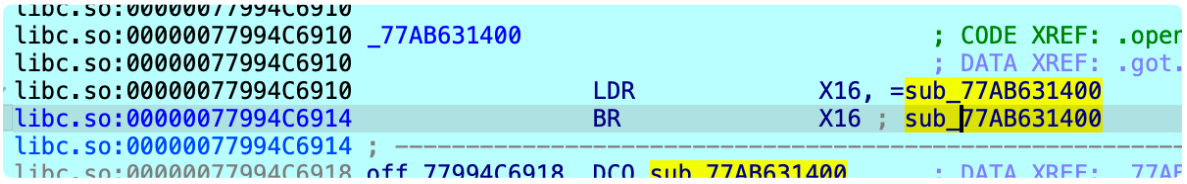
就变成了

```
libc.so`open:
    0x71002b4910 <+0>:  ldr    x17, 0x71002b4918          ; <+8>
    0x71002b4914 <+4>:  br     x17
    0x71002b4918 <+8>:  .long  0x089db010                 ; unknown opcode
    0x71002b491c <+12>: udf    #0x71
    0x71002b4920 <+16>: stp    x24, x23, [sp, #0x100]
    0x71002b4924 <+20>: stp    x22, x21, [sp, #0x110]
    0x71002b4928 <+24>: stp    x20, x19, [sp, #0x120]
    0x71002b492c <+28>: stp    x5, x6, [x29, #-0x48]
```

注意：在修改之前一定要备份这些指令，因为这些指令在后面要被执行

第一条是把地址的内容存到x17，第二条是跳转到x17

第三四条不是汇编指令，是目标要跳转到的地址，我们这个跳板用了16字节，当然你也可以继续跳转字节更少的跳板，本文用的是比较简单的跳板，让我们看看frida用的跳板是什么样的



看来我们跟frida的跳板差不多～

跳转到我们自定义的函数之后呢?

InlineHook的第二步，保存当前函数所有的寄存器状态，调用pre_hookcallback

要保存哪些寄存器呢？最简单的肯定包含X0-X30(包含了lr lr就是x30) 可以选择性保存sp(当然如果栈平衡就无所谓了) NZCV状态寄存器 以及Q系列寄存器（本项目还没有保存）

让我们看看frida是怎么做的

JS

```
ebug238:00000077AAE20004                STP             Q30,
Q31, [SP,#-32]!
debug238:00000077AAE20008                STP             Q28,
Q29, [SP,#-32]!
debug238:00000077AAE2000C                STP             Q26,
Q27, [SP,#-32]!
debug238:00000077AAE20010                STP             Q24,
Q25, [SP,#0x70+var_90]!
debug238:00000077AAE20014                STP             Q22,
```

```
Q23, [SP,#0x90+var_B0]!
debug238:00000077AAE20018                STP             Q20,
Q21, [SP,#0xB0+var_D0]!
debug238:00000077AAE2001C                STP             Q18,
Q19, [SP,#0xD0+var_F0]!
debug238:00000077AAE20020                STP             Q16,
Q17, [SP,#0xF0+var_110]!
debug238:00000077AAE20024                STP             Q14,
Q15, [SP,#0x110+var_130]!
debug238:00000077AAE20028                STP             Q12,
Q13, [SP,#0x130+var_150]!
debug238:00000077AAE2002C                STP             Q10,
Q11, [SP,#0x150+var_170]!
debug238:00000077AAE20030                STP             Q8,
Q9, [SP,#0x170+var_190]!
debug238:00000077AAE20034                STP             Q6,
Q7, [SP,#0x190+var_1B0]!
debug238:00000077AAE20038                STP             Q4,
Q5, [SP,#0x1B0+var_1D0]!
debug238:00000077AAE2003C                STP             Q2,
Q3, [SP,#0x1D0+var_1F0]!
debug238:00000077AAE20040                STP             Q0,
Q1, [SP,#0x1F0+var_210]!
debug238:00000077AAE20044                STP             X29,
X30, [SP,#0x210+var_220]!
debug238:00000077AAE20048                STP             X27,
X28, [SP,#0x220+var_230]!
debug238:00000077AAE2004C                STP             X25,
X26, [SP,#0x230+var_240]!
debug238:00000077AAE20050                STP             X23,
X24, [SP,#0x240+var_250]!
debug238:00000077AAE20054                STP             X21,
X22, [SP,#0x250+var_260]!
debug238:00000077AAE20058                STP             X19,
X20, [SP,#0x260+var_270]!
debug238:00000077AAE2005C                STP             X17,
X18, [SP,#0x270+var_280]!
debug238:00000077AAE20060                STP             X15,
```

```
X16, [SP,#0x280+var_290]!
debug238:00000077AAE20064                STP             X13,
X14, [SP,#0x290+var_2A0]!
debug238:00000077AAE20068                STP             X11,
X12, [SP,#0x2A0+var_2B0]!
debug238:00000077AAE2006C                STP             X9,
X10, [SP,#0x2B0+var_2C0]!
debug238:00000077AAE20070                STP             X7,
X8, [SP,#0x2C0+var_2D0]!
debug238:00000077AAE20074                STP             X5,
X6, [SP,#0x2D0+var_2E0]!
debug238:00000077AAE20078                STP             X3,
X4, [SP,#0x2E0+var_2F0]!
debug238:00000077AAE2007C                STP             X1,
X2, [SP,#0x2F0+var_300]!
debug238:00000077AAE20080                MRS             X1,
NZCV
debug238:00000077AAE20084                STP             X1,
X0, [SP,#0x300+var_310]!
debug238:00000077AAE20088                ADD             X0,
SP, #0x310+arg_0
debug238:00000077AAE2008C                STP             XZR,
X0, [SP,#0x310+var_320]!
debug238:00000077AAE20090                STR             X30,
[SP,#0x320+var_8]
debug238:00000077AAE20094                STR             X29,
[SP,#0x320+var_10]
debug238:00000077AAE20098                ADD             X29,
SP, #0x320+var_10
debug238:00000077AAE2009C                MOV             X1,
SP
debug238:00000077AAE200A0                ADD             X2,
SP, #0x320+var_218
debug238:00000077AAE200A4                ADD             X3,
SP, #0x320+var_10
```

为什么要保存呢？ 因为在函数调用前如果调用了pre_hookcallback 或者在函数调用后调用post_hookcallback，里面的逻辑会污染原来的寄存器，导致函数崩溃。

我们简单举个例子，现在有一个函数

```JS
uint64_t test(int a, int b, int c,int d) {
    return a+b+c+d;
}
```

这个函数主要用到了x0-x3

如果我们在pre_hookcallback调用了任意函数，会刷新x0寄存器，如果返回的是一个地址，那么x0的值在刷新后，会传入我们原来的函数中，造成寄存器污染。

我们将原寄存器保存到哪里？

frida是怎么做的？frida是直接将所有寄存器压入了栈中，然后在按顺序弹出，保证栈平衡

弹出过程（压入过程看上面）

```JS
debug238:00000077AAE200B4                 ADD             SP,
SP, #0x10
debug238:00000077AAE200B8                 LDP             X1,
X0, [SP+0x310+var_310],#0x10
debug238:00000077AAE200BC                 MSR             NZCV,
X1
debug238:00000077AAE200C0                 LDP             X1,
X2, [SP+0x300+var_300],#0x10
debug238:00000077AAE200C4                 LDP             X3,
X4, [SP+0x2F0+var_2F0],#0x10
debug238:00000077AAE200C8                 LDP             X5,
X6, [SP+0x2E0+var_2E0],#0x10
debug238:00000077AAE200CC                 LDP             X7,
X8, [SP+0x2D0+var_2D0],#0x10
debug238:00000077AAE200D0                 LDP             X9,
X10, [SP+0x2C0+var_2C0],#0x10
```

```
debug238:00000077AAE200D4                LDP             X11,
X12, [SP+0x2B0+var_2B0],#0x10
debug238:00000077AAE200D8                LDP             X13,
X14, [SP+0x2A0+var_2A0],#0x10
debug238:00000077AAE200DC                LDP             X15,
X16, [SP+0x290+var_290],#0x10
debug238:00000077AAE200E0                LDP             X17,
X18, [SP+0x280+var_280],#0x10
debug238:00000077AAE200E4                LDP             X19,
X20, [SP+0x270+var_270],#0x10
debug238:00000077AAE200E8                LDP             X21,
X22, [SP+0x260+var_260],#0x10
debug238:00000077AAE200EC                LDP             X23,
X24, [SP+0x250+var_250],#0x10
debug238:00000077AAE200F0                LDP             X25,
X26, [SP+0x240+var_240],#0x10
debug238:00000077AAE200F4                LDP             X27,
X28, [SP+0x230+var_230],#0x10
debug238:00000077AAE200F8                LDP             X29,
X30, [SP+0x220+var_220],#0x10
debug238:00000077AAE200FC                LDP             Q0,
Q1, [SP+0x210+var_210],#0x20
debug238:00000077AAE20100                LDP             Q2,
Q3, [SP+0x1F0+var_1F0],#0x20
debug238:00000077AAE20104                LDP             Q4,
Q5, [SP+0x1D0+var_1D0],#0x20
debug238:00000077AAE20108                LDP             Q6,
Q7, [SP+0x1B0+var_1B0],#0x20
debug238:00000077AAE2010C                LDP             Q8,
Q9, [SP+0x190+var_190],#0x20
debug238:00000077AAE20110                LDP             Q10,
Q11, [SP+0x170+var_170],#0x20
debug238:00000077AAE20114                LDP             Q12,
Q13, [SP+0x150+var_150],#0x20
debug238:00000077AAE20118                LDP             Q14,
Q15, [SP+0x130+var_130],#0x20
debug238:00000077AAE2011C                LDP             Q16,
Q17, [SP+0x110+var_110],#0x20
```

```
debug238:00000077AAE20120                LDP             Q18,
Q19, [SP+0xF0+var_F0],#0x20
debug238:00000077AAE20124                LDP             Q20,
Q21, [SP+0xD0+var_D0],#0x20
debug238:00000077AAE20128                LDP             Q22,
Q23, [SP+0xB0+var_B0],#0x20
debug238:00000077AAE2012C                LDP             Q24,
Q25, [SP+0x90+var_90],#0x20
debug238:00000077AAE20130                LDP             Q26,
Q27, [SP+0x70+var_70],#0x20
debug238:00000077AAE20134                LDP             Q28,
Q29, [SP+0x50+var_50],#0x20
debug238:00000077AAE20138                LDP             Q30,
Q31, [SP+0x30+var_30],#0x20
debug238:00000077AAE2013C                LDP             X16,
X17, [SP+0x10+var_10],#0x10
debug238:00000077AAE20140                RET             X16
```

这种设计需要保证栈平衡，在进入我们的跳板函数和出跳板函数 sp的值要一致，当然在不污染寄存器的情况下，你可以保存sp，在结束之前恢复（不推荐，很麻烦）

当然也可以和我一样，存在一个结构体中，我们的hook框架是怎么做的?

```
// 增加寄存器结构体定义
struct RegisterContext {
    uint64_t x[31];    // X0-X30
    uint64_t sp;       // Stack Pointer
    uint64_t pc;       // Program Counter
    uint64_t pstate;   // Processor State
};
```

然后获取结构体的地址，将数据全部压入结构体中（Q系列寄存器还没有加入）:

ARMASM

```
// 保存寄存器到 HookInfo->ctx
    ldp x0, x1, [sp], #0x10    // 恢复x16,x17
    stp x0, x1, [x16, #128]  // 保存原始x16,x17
    // 恢复x0,x1到临时寄存器
    ldp x0, x1, [sp], #0x10    // 从栈上加载原始x0,x1
    // 保存所有寄存器到ctx
    stp x0, x1, [x16, #0]
    stp x2, x3, [x16, #16]
    stp x4, x5, [x16, #32]
    stp x6, x7, [x16, #48]
    stp x8, x9, [x16, #64]
    stp x10, x11, [x16, #80]
    stp x12, x13, [x16, #96]
    stp x14, x15, [x16, #112]
    stp x18, x19, [x16, #144]
    stp x20, x21, [x16, #160]
    stp x22, x23, [x16, #176]
    stp x24, x25, [x16, #192]
    stp x26, x27, [x16, #208]
    stp x28, x29, [x16, #224]
    str x30, [x16, #240]
```

这样不用考虑栈平衡的问题，因为我们没用到栈（当然会有新的分支，使用栈来储存，因为他真的很方便）

pre_hookcallback顾名思义就是在调用hook之前的回调函数，在这时，原函数还没有开始执行，但是参数已经被压入到了寄存器中

如何调用?

ARMASM

```
// 调用pre_callback
sub x0, x16, #0x30          // HookInfo作为第一个参数
```

```
ldr x17, [x0]                // 加载pre_callback函数指针
blr x17                      // 调用pre_callback
```

我们非常简单的就实现了调用,在这里我们可以读取寄存器，修改寄存器

C++

```
// 默认的寄存器打印回调函数
void default_register_callback(HookInfo *info) {
    RegisterContext *ctx = &info->ctx;
    LOGI("Register dump:");
    for (int i = 0; i < 31; i++) {
        LOGI("X%d: 0x%llx", i, ctx->x[i]);
    }
}
```

```
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  Register dump:
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  X0: 0x1
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  X1: 0x2
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  X2: 0x3
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  X3: 0x4
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  X4: 0x7fec5c0eb0
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  X5: 0x192f5ea
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  X6: 0x10
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  X7: 0x7f7f7f7f7f7f7f7f
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  X8: 0xb40000703d566c00
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  X9: 0xf1df82edf9a42528
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  X10: 0x2
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  X11: 0xfffffffffffffffd
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  X12: 0x7fec5c0fd0
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  X13: 0x1c
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  X14: 0x7fec5c2318
2024-12-04 17:00:24.004 17504-17504 jiqiu2021          com.example.inlinehookstudy          I  X15: 0x50672671377
```

InlineHook的第三步，恢复寄存器，修复指令，调用原函数

因为前面做了太多的操作，比如

ARMASM

```
sub x0, x16, #0x30           // 这里污染了x16 x0寄存器
ldr x17, [x0]                // 这里污染了x17寄存器
blr x17                      // 这里污染了x30寄存器
```

我们需要把结构体里的寄存器复原，然后在调用原函数

ARMASM

```
ldr x16, _twojump_start //拿到结构体地址
add x16, x16, #0x30 //计算结构体ctx成员
// 恢复所有寄存器 比如在hook里修改了，那这里就要还原了
ldp x0, x1, [x16, #0]
ldp x2, x3, [x16, #16]
ldp x4, x5, [x16, #32]
ldp x6, x7, [x16, #48]
ldp x8, x9, [x16, #64]
ldp x10, x11, [x16, #80]
ldp x12, x13, [x16, #96]
ldp x14, x15, [x16, #112]
ldp x18, x19, [x16, #144]
ldp x20, x21, [x16, #160]
ldp x22, x23, [x16, #176]
ldp x24, x25, [x16, #192]
ldp x26, x27, [x16, #208]
ldp x28, x29, [x16, #224]
ldr x30, [x16, #240]
sub x16, x16, #0x30
```

做完了该做的事情以后，终于轮到被hook的函数执行了

第一种实现：

把之前备份的指令还原回被hook函数，然后跳转执行

因为函数是在我们这里被“主动调用的”

ARMASM

```
// 调用原函数
ldr x17, [x16, #16]       // 加载原函数地址
```

```
blr x17
//这里x0已经有返回值了
```

所以在执行完成后还是会返回我们的汇编指令

这种方法不提倡，因为如果被hook函数是多线程执行的（大部分都是这样）

会发生崩溃，具体大家可以思考一下

第二种实现：

将备份后的指令执行，然后跳回原来的函数(+16字节的位置)

第二种实现涉及到指令修复的问题:

为什么会出现指令修复，因为有一些特殊指令是和当前pc相关的，我们改变了指令的位置

ARMASM

```
enum class ARM64_INS_TYPE {
    UNKNOW,    //除了下面意外的所有指令
    ADR,        // 形如 ADR Xd, label
    ADRP,       // 形如 ADRP Xd, label
    B,          // 形如 B label
    BL,         // 形如 BL label
    B_COND,     // 形如 B.cond label
    CBZ_CBNZ,   // 形如 CBZ/CBNZ Rt, label
    TBZ_TBNZ,   // 形如 TBZ/TBNZ Rt, #imm, label
    LDR_LIT,    // 形如 LDR Rt, label
};
```

对应的pc值也会发生改变：

ARMASM

```
2024-12-04 17:00:24.004 17504-17504 jiqiu2021
com.example.inlinehookstudy          I  Processing
```

```
instruction[0]: 0xd100c3ff at old_addr: 0x70441c9f64, new_addr:
0x71088f815c
2024-12-04 17:00:24.004 17504-17504 jiqiu2021
com.example.inlinehookstudy          I  Processing
instruction[1]: 0xa9027bfd at old_addr: 0x70441c9f68, new_addr:
0x71088f8160
2024-12-04 17:00:24.004 17504-17504 jiqiu2021
com.example.inlinehookstudy          I  Processing
instruction[2]: 0x910083fd at old_addr: 0x70441c9f6c, new_addr:
0x71088f8164
2024-12-04 17:00:24.004 17504-17504 jiqiu2021
com.example.inlinehookstudy          I  Processing
instruction[3]: 0xb81fc3a0 at old_addr: 0x70441c9f70, new_addr:
0x71088f8168
```

我们采用shadowhook里面的代码进行修复（我给重写并加上了注释）

拿修复adrp的部分做例子：

C++

```
    static size_t fix_adrp(uint32_t *out_ptr, uint32_t ins,
void *old_addr, void *new_addr) {
        uint64_t pc = (uint64_t) old_addr;

        // 获取目标寄存器和立即数
        uint32_t rd = SH_UTIL_GET_BITS_32(ins, 4, 0);  // 目标寄
存器
        uint64_t immlo = SH_UTIL_GET_BITS_32(ins, 30, 29);  //
低2位
        uint64_t immhi = SH_UTIL_GET_BITS_32(ins, 23, 5);   //
高19位
        uint64_t offset = SH_UTIL_SIGN_EXTEND_64((immhi << 14u)
| (immlo << 12u), 33u);

        // 计算目标页地址
        uint64_t addr = (pc & 0xFFFFFFFFFFFFF000) + offset;
```

```
        // 生成新的LDR序列
        out_ptr[0] = 0x58000040u | rd;  // LDR Xd, #8
        out_ptr[1] = 0x14000003;        // B #12
        out_ptr[2] = addr & 0xFFFFFFFF;  // 低32位
        out_ptr[3] = addr >> 32u;        // 高32位

        return 16;  // 4条指令
    }
```

其实就是把adrp（8字节）等价替换成了 （16字节的指令）

C++

```
        out_ptr[0] = 0x58000040u | rd;  // LDR Xd, #8
        out_ptr[1] = 0x14000003;        // B #12
        out_ptr[2] = addr & 0xFFFFFFFF;  // 低32位
        out_ptr[3] = addr >> 32u;        // 高32位
```

InlineHook的第四步，保存寄存器，调用post_hookcallback

在调用完原函数以后，相应的寄存器的值发生了变化，我们再次保存，然后调用函数调用后的回调函数

ARMASM

```
    // 调用原函数
    ldr x17, [x16, #16]       // 加载原函数地址
    blr x17
    //这里x0已经有返回值了

    ldr x16, _twojump_start
```

```
    add x17, x16, #48          // x17指向ctx

    // 再次保存寄存器到ctx
    stp x0, x1, [x17, #0]
    stp x2, x3, [x17, #16]
    stp x4, x5, [x17, #32]
    stp x6, x7, [x17, #48]
    stp x8, x9, [x17, #64]
    stp x10, x11, [x17, #80]
    stp x12, x13, [x17, #96]
    stp x14, x15, [x17, #112]
    stp x18, x19, [x17, #144]
    stp x20, x21, [x17, #160]
    stp x22, x23, [x17, #176]
    stp x24, x25, [x17, #192]
    stp x26, x27, [x17, #208]
    stp x28, x29, [x17, #224]

    // 调用post_callback
    mov x0, x16                // HookInfo作为第一个参数
    ldr x16, [x16, #8]        // 加载post_callback
    blr x16
```

注释写的很清楚，和上面高速相似，相信大家已经看懂了

就结束了吗？还没有结束，如果你在post_callback修改了寄存器的值，还需要恢复

ARMASM

```
// 恢复所有寄存器
    ldr x16, _twojump_start
    add x16, x16, #0x30
    // 恢复所有寄存器 比如在hook里修改了，那这里就要还原了
    ldp x0, x1, [x16, #0]
    ldp x2, x3, [x17, #16]
    ldp x4, x5, [x16, #32]
```

```
    ldp x6, x7, [x16, #48]
    ldp x8, x9, [x16, #64]
    ldp x10, x11, [x16, #80]
    ldp x12, x13, [x16, #96]
    ldp x14, x15, [x16, #112]
    ldp x18, x19, [x16, #144]
    ldp x20, x21, [x16, #160]
    ldp x22, x23, [x16, #176]
    ldp x24, x25, [x16, #192]
    ldp x26, x27, [x16, #208]
    ldp x28, x29, [x16, #224]
    ldr x30, [x16, #240]
    ret
```

当然这里可以只恢复x0（返回值）

到此我们已经完整完成了一个hook的流程

项目设计思路

项目地址放在上面了。本次讲解的是newhook分支

项目的总hook管理结构体是:

C++

```
// 全局存储所有hook信息
class HookManager {
private:
    static std::map<void *, HookInfo *> hook_map; // key是目标函数地址
    static std::mutex hook_mutex;

public:
    static void registerHook(HookInfo *info) {
        if (!info) return;
```

```
        setCurrentHook(info);
        std::lock_guard<std::mutex> lock(hook_mutex);
        hook_map[info->target_func] = info;
    }

    static void setCurrentHook(HookInfo *info) {
        current_executing_hook = info;
    }

    static HookInfo *getCurrentHook() {
        return current_executing_hook;
    }

    static HookInfo *getHook(void *target_func) {
        std::lock_guard<std::mutex> lock(hook_mutex);
        auto it = hook_map.find(target_func);
        return (it != hook_map.end()) ? it->second : nullptr;
    }

    static void removeHook(void *target_func) {
        std::lock_guard<std::mutex> lock(hook_mutex);
        hook_map.erase(target_func);
    }
};
```

这里重点注意三个函数:

`registerHook` 注册函数

`removeHook` 移除注册函数

`getHook` 获取hook状态（比如传入open地址，看是不是已经被hook了） 链式hook准备做，但是感觉没啥用

C++

```
struct HookInfo {
    void (*pre_callback)(HookInfo *pHookInfo);  //储存hook前回调函数地址
    void (*post_callback)(HookInfo *ctx); //储存hook后回调函数地址
    void *backup_func; //mmap出来存汇编的地址
    void *target_func;//目标hook的地址
    void *hook_func;//无意义，历史遗留


    void *user_data;//无意义，历史遗留

    // 寄存器上下文
    RegisterContext ctx;

    // 原始代码
    uint8_t original_code[1024];//存储备份函数字节的空间
    size_t original_code_size; //备份了多少字节
};
```

hook信息储存结构体，在汇编里获取的也是这个

每一个hook，对应一个hookinfo

C++

```
void * openaddr =dlsym(RTLD_DEFAULT, "open");
    HookInfo *hookInfo = createHook((void *) openaddr,
                                    my_register_callback,
                                    post_hook_callback,
                                    (void *) hello.c_str());
```

重点讲一下createHook一些点：

C++

```
HookInfo *createHook(void *target_func,
                     void (*pre_callback)(HookInfo *) =
nullptr,
                     void (*post_callback)(HookInfo *) =
nullptr,
                     void *user_data = nullptr) {
    LOGI("Creating hook - target: %p", target_func);
    if (!target_func ) return nullptr;
    // 检查是否已经被hook
    HookInfo *existing = HookManager::getHook(target_func);
    if (existing) {
        LOGE("Function already hooked!");
        return nullptr;
    }

    // 创建HookInfo结构
    auto *hookInfo = new HookInfo();
    if (!hookInfo) return nullptr;

    // 初始化结构
    memset(hookInfo, 0, sizeof(HookInfo));
    hookInfo->target_func = target_func;
    hookInfo->pre_callback = pre_callback ? pre_callback :
default_register_callback;
    hookInfo->post_callback = post_callback;
    hookInfo->user_data = user_data;
    // 备份原始指令
    if (!backup_orig_instructions(hookInfo)) {
        delete hookInfo;
        return nullptr;
    }

    // 分配跳板内存
    size_t trampoline_size = 1024;
    void *trampoline = mmap(nullptr, trampoline_size,
                            PROT_READ | PROT_WRITE | PROT_EXEC,
                            MAP_PRIVATE | MAP_ANONYMOUS, -1,
```

```
0);

    if (trampoline == MAP_FAILED) {
        delete hookInfo;
        return nullptr;
    }
    LOGI("Trampoline allocated at %p", trampoline);

    hookInfo->backup_func = trampoline;
//print two_jump_start two_jump_end addr
    LOGI("two jump start addr = %p",two_jump_start);
    LOGI("two jump end addr = %p",two_jump_end);

    size_t two_jump_size =two_jump_end-two_jump_start;
    memcpy(hookInfo->backup_func, two_jump_start,
two_jump_size);
    LOGI("hook info addr = %p",hookInfo);
    // 在预留的NOP位置写入地址
    uint64_t info_addr = (uint64_t)hookInfo;
    uint64_t hook_addr = (uint64_t)hookInfo->hook_func;

// 填充HookInfo地址(前8字节)
    memcpy(hookInfo->backup_func, &info_addr,
sizeof(info_addr));

// 填充hook函数地址(后8字节)
    memcpy((uint8_t*)hookInfo->backup_func + 8, &hook_addr,
sizeof(hook_addr));

    // 修复指令时记录指令信息
    uint32_t *orig = (uint32_t *) hookInfo->original_code;
    for (size_t i = 0; i < hookInfo->original_code_size / 4;
i++) {
        LOGI("Original instruction[%zu]: 0x%08x", i, orig[i]);
    }

    size_t fixed_size = ARM64Fixer::fix_instructions(
            (uint32_t *) hookInfo->original_code,
```

```
            hookInfo->original_code_size,
            hookInfo->target_func,
            (uint32_t *)((uintptr_t)hookInfo->backup_func +
two_jump_size)
    );
    void *return_addr = (uint8_t *) target_func + hookInfo-
>original_code_size;
    // 添加跳回原函数的跳转
    if (!create_jump((uint8_t *) hookInfo->backup_func +
fixed_size+two_jump_size,
                     return_addr, false)) {
        munmap(trampoline, trampoline_size);
        delete hookInfo;
        return nullptr;
    }
    // 在目标函数处写入跳转到hook函数的代码
    if (!create_jump(target_func, (uint8_t*)hookInfo-
>backup_func+16, false)) {
        munmap(trampoline, trampoline_size);
        delete hookInfo;
        return nullptr;
    }
    hookInfo->backup_func=(uint8_t*)hookInfo-
>backup_func+two_jump_size;
    HookManager::registerHook(hookInfo);
    LOGI("hookinfo addr %p",hookInfo);
    LOGI("ctx addr %p",&hookInfo->ctx.x[0]);
    return hookInfo;
}
```

`backup_orig_instructions(hookInfo)` 主要是备份原来的函数,将指令拷贝到结构体里

C++

```
void *trampoline = mmap(nullptr, trampoline_size,
                            PROT_READ | PROT_WRITE | PROT_EXEC,
```

```
                                MAP_PRIVATE | MAP_ANONYMOUS, -1,
0);

    if (trampoline == MAP_FAILED) {
        delete hookInfo;
        return nullptr;
    }
    LOGI("Trampoline allocated at %p", trampoline);

    hookInfo->backup_func = trampoline;
```

每次都创建一个页大小的内存，来存储跳板函数就是.s里面的

.s里面我更喜欢叫他模板函数，因为他不是最终执行的，每创建一个函数都会创建一块内存，然后memcpy到mmap出来的这一块内存里，在填入当前hook的hookinfo

C++

```
size_t two_jump_size =two_jump_end-two_jump_start; //这里计算汇编
大小
memcpy(hookInfo->backup_func, two_jump_start, two_jump_size);
//这里开始拷贝
LOGI("hook info addr = %p",hookInfo);
// 在预留的位置填入地址，这里写hookInfo->hook_func;有点多余，其实可以直
接取
uint64_t info_addr = (uint64_t)hookInfo;
uint64_t hook_addr = (uint64_t)hookInfo->hook_func;
//填充hookinfo地址
 memcpy(hookInfo->backup_func, &info_addr, sizeof(info_addr));
// 填充hook函数地址(后8字节)
memcpy((uint8_t*)hookInfo->backup_func + 8, &hook_addr,
sizeof(hook_addr));
```

这一部分对备份的指令进行修复，然后填入mmap的那块空间里

```C++
uint32_t *orig = (uint32_t *) hookInfo->original_code;
    for (size_t i = 0; i < hookInfo->original_code_size / 4;
i++) {
        LOGI("Original instruction[%zu]: 0x%08x", i, orig[i]);
    }

    size_t fixed_size = ARM64Fixer::fix_instructions(
            (uint32_t *) hookInfo->original_code,
            hookInfo->original_code_size,
            hookInfo->target_func,
            (uint32_t *)((uintptr_t)hookInfo->backup_func +
two_jump_size)
    );
```

之后添加跳回原来函数的跳转指令:

```C++
 void *return_addr = (uint8_t *) target_func + hookInfo-
>original_code_size;
    // 添加跳回原函数的跳转
    if (!create_jump((uint8_t *) hookInfo->backup_func +
fixed_size+two_jump_size,
                     return_addr, false)) {
        munmap(trampoline, trampoline_size);
        delete hookInfo;
        return nullptr;
    }
```

mmap分配出的空间目前是

填充过的.s模板汇编地址|修复过的原函数指令|跳回原函数执行的地址

```C++
if (!create_jump(target_func, (uint8_t*)hookInfo-
>backup_func+16, false)) {
```

```
        munmap(trampoline, trampoline_size);
        delete hookInfo;
        return nullptr;
    }
```

修改目标函数头部，跳转到模板函数+16字节处(16字节存储地址)

```C++
    hookInfo->backup_func=(uint8_t*)hookInfo-
>backup_func+two_jump_size;
```

hookInfo->backup_func指针指向修复过的指令地址，并且执行完修复的指令回跳回原函数位置

由于

```C++
 // 调用原函数
    ldr x17, [x16, #16]      // 加载原函数地址
    blr x17
```

使用的是blr x30寄存器被我们修改在这里了，所以回跳回模板函数

```C++
bool inline_unhook(HookInfo *info) {
    if (!info) return false;
    HookManager::removeHook(info->target_func);

    // 修改目标函数内存权限
    size_t page_size = sysconf(_SC_PAGESIZE);
    void *page_start = (void *) ((uintptr_t) info->target_func
& ~(page_size - 1));
    if (mprotect(page_start, page_size, PROT_READ | PROT_WRITE
| PROT_EXEC) != 0) {
        return false;
```

```
    }

    // 直接恢复原始指令,而不是创建跳转
    memcpy(info->target_func, info->original_code, info->original_code_size);

    // 清理指令缓存
    __builtin___clear_cache((char *) info->target_func,
                            (char *) info->target_func + info->original_code_size);

    // 释放跳板内存
    if (info->backup_func) {
        munmap(info->backup_func, 256);
    }

    delete info;
    return true;
}
```

取消hook就很简单了，把目标函数头部还原即可

结言

目前框架还有很多bug，而且只支持arm64，我会不断发展，开出多个分支，并给出实战（配合我的注入框架，以及开发出server，像frida一样通过JIT生成代码进行hook）

有任何疑问欢迎留言，我会解答，第二篇就打算完善后继续写，或者结合我的注入框架写一篇实战。

目前TODO:

1. 支持Q系列寄存器
2. 使用栈作为参数，精简汇编

3. 解决X16 X17寄存器污染问题

最后特别感谢先辈的伟大项目，能让我学习到特别多的技巧，特别感谢卓童老师开展这样一个教学性的项目，让我受益匪浅。