第二次作业

使用的工具

name	version
MT manager	v2.14.0
Leidian player	v9.0.60(9)
MuMu player	v3.5.21(2169)
jadx	v1.4.4
jeb	v4.20
adb	v10.0.19045
VMOS Pro	v2.9.8

1. apk分析

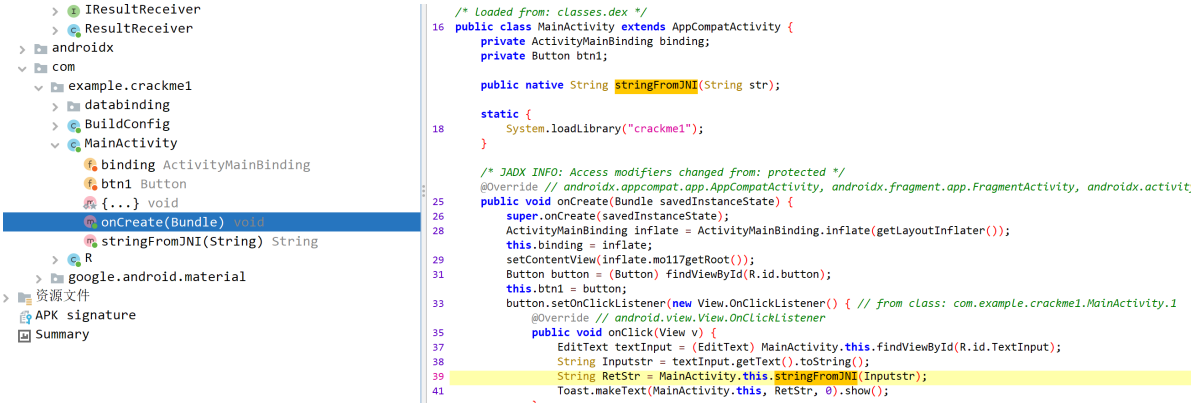
用MT管理器查看程序信息，包名为`com.example.crackme1`，未加固。

反编译AndroidManifest.xml，看到`android.intent.category.LAUNCHER`，说明该activity是主界面，即`MainActivity`为程序入口

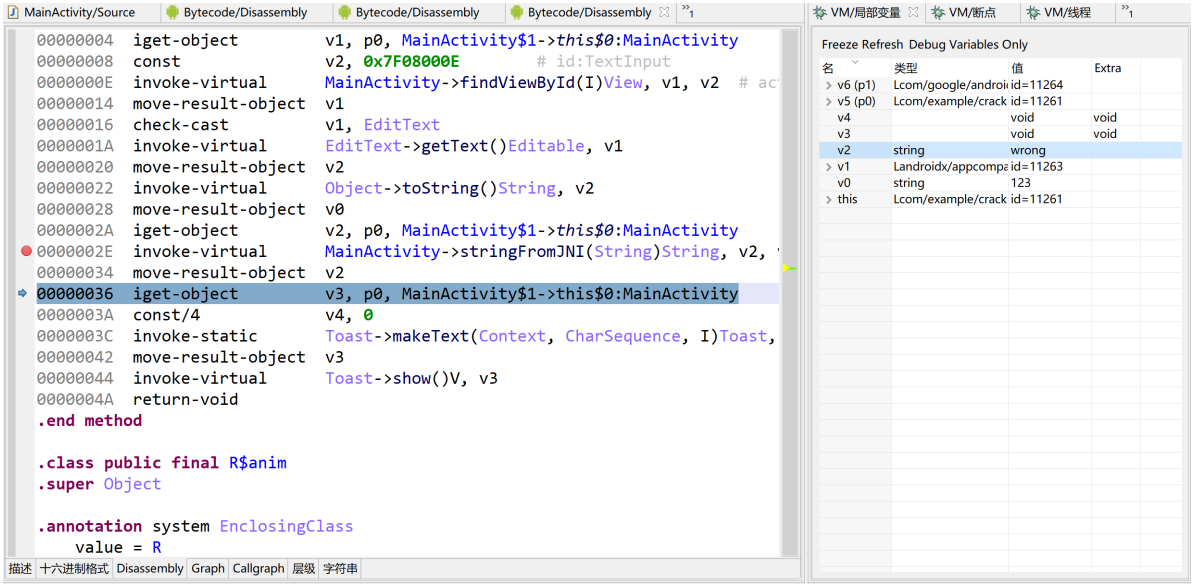
AndroidManifest.xml

```
    platformBuildVersionName="11">
    <uses-sdk
        android:minSdkVersion="21"
        android:targetSdkVersion="30" />
    <application
        android:theme="@7F0F0199"
        android:label="@7F0E001B"
        android:icon="@7F0C0000"
        android:debuggable="true"
        android:allowBackup="true"
        android:supportsRtl="true"
        android:roundIcon="@7F0C0001"
        android:appComponentFactory="androidx.core.app.CoreComponentFactory">
        <activity
            android:name="com.example.crackme1.MainActivity"
            android:exported="true">
            <intent-filter>
                <action
                    android:name="android.intent.action.MAIN" />
                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

向jadx导入crackme1.apk，找到`MainActivity`类，通过查看反编译JAVA代码，可以看到`button`类有一个`serOnClickListener`()方法。接着往下看，可以判断`btn1`是一个确认按钮，`onClick()`函数是一个槽函数，当按钮接收到点击事件，读取`EditText`中的内容并转为String类型的`Inputstr`，再由`stringFromJNI()`函数处理Inputstr，并返回`RetStr`，RetStr字符串会在`Toast.makeText()`中打印，推测这个就是窗口底下的`right`和`wrong`



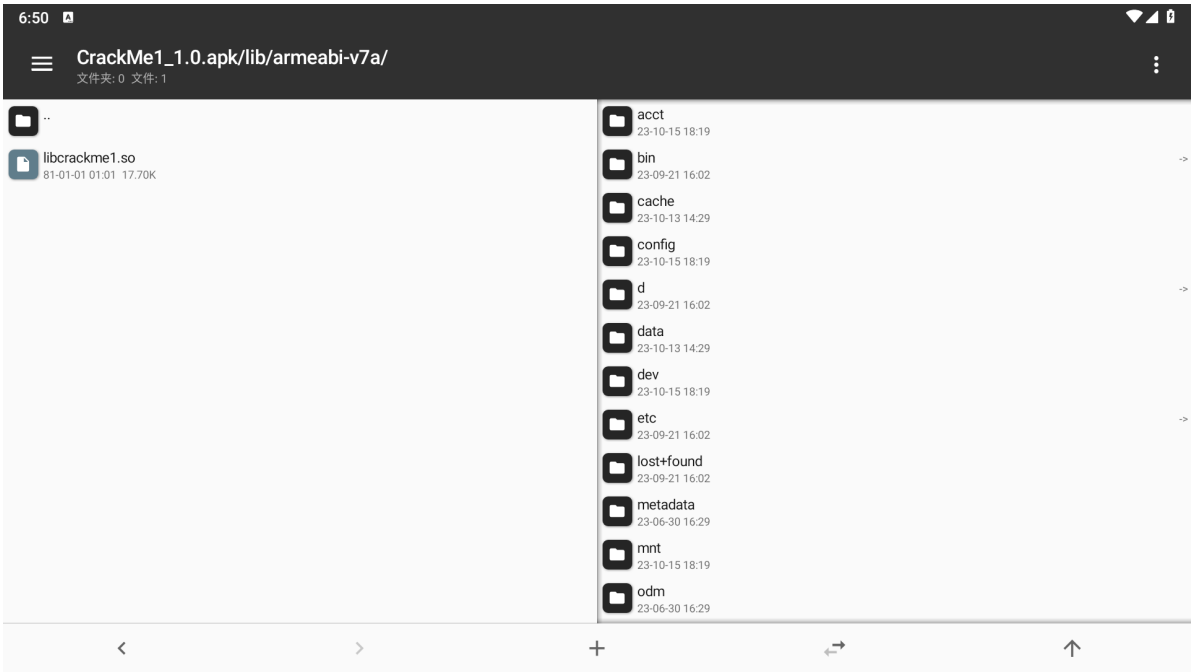
Inputstr是输入文本，RetStr是输出提示，故`stringFromJNI()`是一个验证输入文本的函数。用JEB远程附加在com.example.crackme1上进行调试，在`invoke-virtual`调用stringFromJNI处下断点，验证这里确实返回了“wrong”到v2中，其中v0是输入的文本，为123。



查看Smali代码，发现stringFromJNI只有一个public声明，说明是调用的动态链接库。再结合`System.loadLibrary("crackme1")`，推测stringFromJNI来自这里。

```
83
84 .method public native stringFromJNI(Ljava/lang/String;)Ljava/lang/String;
85 .end method
86
```

再用MT管理器打开apk，在`crackme1/lib/armeabi-v7a`找到`libcrackme1.so`，对应了前面的库加载。后面以它为主要分析目标。

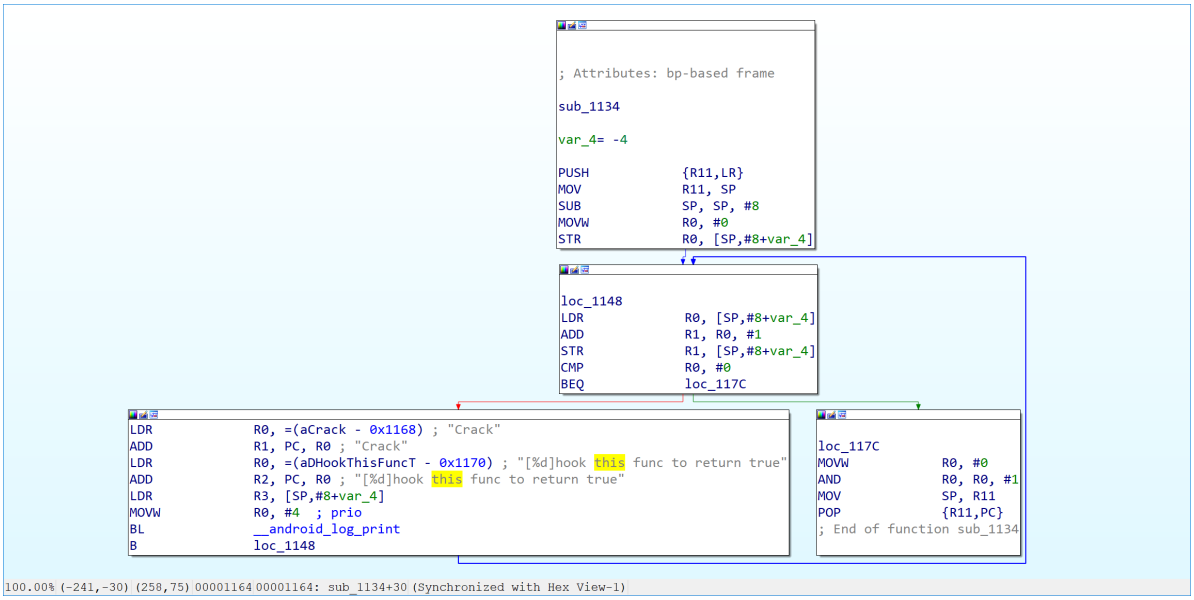


将libcrakme1.so导入IDA，反编译代码，找到判断逻辑，发现`sub_F50()`实际是一个随机函数，故`strncmp()`的结果完全是随机的，而`sub_1134()`在反编译代码中恒返回0，考虑应该通过注入inlinehook将其改为1

```
int __fastcall Java_com_example_crackme1_MainActivity_stringFromJNI(int a1, int a2, int a3)
{
  char *s2; // [sp+Ch] [bp-7Ch]
  int v6; // [sp+1Ch] [bp-6Ch]
  char s[100]; // [sp+20h] [bp-68h] BYREF

  s2 = (char *)sub_12A4(a1, a3, 0);             // 输入的字符串
  memset(s, 0, sizeof(s));
  qmemcpy(s, "123", 3);
  sub_F50(5, s);
  if ( !strncmp(s, s2, 5u) || (sub_1134() & 1) != 0 )// sub_1134()恒返回0
    v6 = sub_12EC(a1, "right");
  else
    v6 = sub_12EC(a1, "wrong");
  return v6;
}
```

查看原汇编指令，这是一个永远执行右边分支的函数，且右边分支一定返回0。左边分支调用`__android_log_print`，打印的内容是提示我们对该函数进行hook，使其永远返回true，记其为`proc_0x1134`，关于libcrackme.so基地址的地址偏移也为0x1134。



2. ptrace注入

以下是听视频课程时关于ptrace注入的笔记，后面编写注入器injector会多次调用到。

```
ptrace(PTRACE_ATTACH, pid, NULL, NULL)附加到远程进程

父进程用waitpid()判断子进程是否暂停
修改远程进程前，需要先读取和保存所有寄存器的值，detach时需要恢复到原有环境

ptrace(PTRACE_GETREGS, pid, NULL, regs)读取寄存器
ptrace(PTRACE_SETREGS, pid, NULL, regs)写入寄存器
ptrace(PTRACE_PEEKTEXT, pid, addr, pBuf)读取内存（单位为word）
ptrace(PTRACE_POKETEXT, pid, addr, data)写入内存（单位为word）
ptrace(PTRACE_DETACH, pid, NULL, 0)脱离远程进程
```

ptrace注入crackme1的思路是，①ptrace_attach附加到进程 --> ②远程调用malloc函数分配内存 --> ③ptrace_poketext将libhook.so真实路径写入进程内存 --> ④远程调用dlopen加载libhook,so文件 --> ⑤ptrace_cont恢复进程，ptrace_detach脱离进程

```
D:\Learning\2023project\TLearning\题目2\平仔骏-U202112052-第二次作业>adb shell
lmipro:/ $ su
lmipro:/ # pidof com.example.crackme1
14701 17021
lmipro:/ # pidof com.example.crackme1
14701 17021
lmipro:/ # cat proc/17021/maps | grep libc.so
eeccc000-eecf8000 r--p 00000000 fc:00 596                              /apex/com.android.runtime/lib/bionic/libc.so
eecf8000-eed8e000 r-xp 0002b000 fc:00 596                              /apex/com.android.runtime/lib/bionic/libc.so
eed8e000-eed92000 r--p 000c0000 fc:00 596                              /apex/com.android.runtime/lib/bionic/libc.so
eed92000-eed94000 rw-p 000c3000 fc:00 596                              /apex/com.android.runtime/lib/bionic/libc.so
lmipro:/ # cat proc/17021/maps | grep libdl.so
ed5c2000-ed5c3000 r--p 00000000 fc:00 597                              /apex/com.android.runtime/lib/bionic/libdl.so
ed5c3000-ed5c4000 r-xp 00000000 fc:00 597                              /apex/com.android.runtime/lib/bionic/libdl.so
ed5c4000-ed5c5000 r--p 00000000 fc:00 597                              /apex/com.android.runtime/lib/bionic/libdl.so
lmipro:/ #
```

`malloc()` 函数在 `libc.so` 文件中，`dlopen()` 函数在 `libdl.so` 文件中，查找com.example.crackme1的maps，找到了这两个库的路径，用adb pull下来，在IDA中查看两个函数的地址偏移。分别为 `0x0002D584` 和 `0x0001848`，但是有问题的是左边的指令编码有得是16 bit有得是32 bit节，这应该ARM架构的指令集模式有关。

```
0001848
0001848             ; =============== S U B R O U T I N E =======================================
0001848
0001848
0001848                             WEAK dlopen
0001848             dlopen                                  ; DATA XREF: LOAD:000002C4↑o
0001848 80 B5                       PUSH            {R7,LR}
000184A 72 46                       MOV             R2, LR
000184C 00 F0 A0 E8                 BLX             __loader_dlopen
0001850 80 BD                       POP             {R7,PC}
0001850             ; End of function dlopen
0001850
0001852
0001852             ; =============== S U B R O U T I N E =======================================
```

```
.text:0002D684              ; =============== S U B R O U T I N E =======================================
.text:0002D684
.text:0002D684
.text:0002D684                              EXPORT malloc
.text:0002D684              malloc                                  ; CODE XREF: j_malloc+8↓j
.text:0002D684                                                      ; DATA XREF: LOAD:0000236C↑o ...
.text:0002D684 B0 B5                        PUSH            {R4,R5,R7,LR}
.text:0002D686 04 46                        MOV             R4, R0
.text:0002D688 0E 48                        LDR             R0, =(__libc_globals - 0x2D68E)
.text:0002D68A 78 44                        ADD             R0, PC  ; __libc_globals
.text:0002D68C 28 30                        ADDS            R0, #0x28 ; '('
.text:0002D68E D0 E8 AF 0F                  LDA.W           R0, [R0]
.text:0002D692 30 B9                        CBNZ            R0, loc_2D6A2
.text:0002D694 20 46                        MOV             R0, R4
.text:0002D696 03 F0 9B FD                  BL              je_malloc
.text:0002D69A 05 46                        MOV             R5, R0
.text:0002D69C 38 B1                        CBZ             R0, loc_2D6AE
.text:0002D69E
.text:0002D69E              loc_2D69E                               ; CODE XREF: malloc+28↓j
.text:0002D69E 28 46                        MOV             R0, R5
.text:0002D6A0 B0 BD                        POP             {R4,R5,R7,PC}
.text:0002D6A2              ; ---------------------------------------------------------------------------
```

> ARM 指令集相对复杂，支持更多的寻址模式和指令类型，以及更多的寄存器，通常是 32 位长。
>
> Thumb 指令集是一种精简指令集，支持较少的寻址模式和指令类型，以及较少的寄存器，通常是 16 位长
>
> 当在ARM体系结构中混合使用ARM指令集和Thumb指令集时，通常函数的地址的最低位用于指示所使用的指令集，如果函数的地址的LSB为0，那么它将以ARM指令集执行。如果LSB为1，它将以Thumb指令集执行。

在实机中使用busybox的指令readelf查看偏移地址，从中可以看到dlopen()的偏移地址为 `0x1849`，malloc()的偏移地址为 `0x2d685`，这说明该Armv7a处理器以 `Thumb` 模式运行。

```
eadelf -s -W /apex/com.android.runtime/lib/bionic/libdl.so | grep dlopen
     2: 00000000     0 FUNC    WEAK   DEFAULT  UND __loader_dlopen
    10: 00000000     0 FUNC    WEAK   DEFAULT  UND __loader_android_dlopen_ext
    19: 00001849    10 FUNC    WEAK   DEFAULT   10 dlopen
    23: 0000188f    10 FUNC    WEAK   DEFAULT   10 android_dlopen_ext
```

```
eadelf -s -W /apex/com.android.runtime/lib/bionic/libc.so | grep malloc
    53: 0002d8dd   112 FUNC    GLOBAL DEFAULT   17 malloc_iterate
    87: 000d1a8c     4 OBJECT  GLOBAL DEFAULT   28 __malloc_hook
   113: 0002d94d    24 FUNC    GLOBAL DEFAULT   17 malloc_disable
   169: 000303d9     4 FUNC    GLOBAL DEFAULT   17 dlmalloc
   249: 000303eb     4 FUNC    GLOBAL DEFAULT   17 dlmalloc_trim
   536: 0002d685    76 FUNC    GLOBAL DEFAULT   17 malloc
   569: 000303e9     2 FUNC    GLOBAL DEFAULT   17 dlmalloc_inspect_all
   672: 000303d1     4 FUNC    GLOBAL DEFAULT   17 dlmalloc_usable_size
   795: 0002e611    32 FUNC    GLOBAL DEFAULT   17 free_malloc_leak_info
   806: 0002d6d1    32 FUNC    GLOBAL DEFAULT   17 malloc_usable_size
   823: 0002d601    32 FUNC    GLOBAL DEFAULT   17 malloc_info
  1150: 0002d965    24 FUNC    GLOBAL DEFAULT   17 malloc_enable
  1468: 0002e55d   180 FUNC    GLOBAL DEFAULT   17 get_malloc_leak_info
  1609: 0002e3c9    24 FUNC    GLOBAL DEFAULT   17 malloc_backtrace
```

在开始ptrace注入代码之前，还需要搞清楚远程调用函数时，对16个寄存器应该如何操作。

> regs.uregs[16]是用于存储CPU寄存器的数组
>
> R0 ~ R3用于存储函数的形参，其中R0用于存储返回值
>
> R4~R11用于存储局部变量和临时数据
>
> R12用于存储临时数据
>
> R13是SP寄存器，是堆栈的指针
>
> R14是LR寄存器，存储函数调用的返回地址
>
> R15是PC寄存器，存储指令地址
>
> R16是CPSR寄存器，第5位用于**标示Thumb模式**!

据此我可以编写远程调用`malloc`和`dlopen`的函数（以下以malloc函数为例，详情可见源代码）

```
1  void *mallOc(pid_t pid, size_t len)
2  {
3      /*
4          malloc 1个参数，压入R0中
5          返回值在R0中
6          LR寄存器存返回地址
7          手动传参后，修改PC寄存器为目标函数地址，修改LR寄存器为0
8          函数返回时触发异常，获取返回值
9      */
10     int status;
11     struct user_regs pushed_regs;
12     struct user_regs regs;
13     ul malloc_offset;
14     void *remote_lib_base;
15     void *malloc_addr;
16     malloc_offset = 0x2d685;                                    // 用IDA读取elf文件
   libc.so观察得出
17     // 计算malloc在远程进程中的地址
18     remote_lib_base = get_lib_base(pid, libc_dir);              // 获取libc.so基地
   址
```

```
    malloc_addr = remote_lib_base + malloc_offset;          // 计算malloc地址
    dlerror();
    if( ptrace(PTRACE_GETREGS, pid, NULL, (void *)&pushed_regs) < 0 )
    {
        perror("[WARNING] failed to push remote regs \n");
    }
    memcpy(&regs, &pushed_regs, sizeof(struct user_regs)); // 寄存器压栈
    regs.uregs[13] -= 0x50;                                 // 堆栈压80位，主要就是存放寄存器
    regs.uregs[14] = 0;                                     // LR寄存器
    regs.uregs[15] = ((ul)malloc_addr & 0xFFFFFFFE);        // PC寄存器，地址最低位决定处理器模式
    regs.uregs[0] = (ul)len;                                // R0
    printf("[MESSAGE] Remote call: malloc(0x%lx)\n", (ul)len);
//  如果PSR寄存器有变化，则需要修改第5位
//    if ((ul)malloc_addr & 0x1) // 设置PSR寄存器，第5位为1表Thumb模式，为0表ARM模式
//        regs.uregs[16] = regs.uregs[16] | 0x20;
//    else
//        regs.uregs[16] = regs.uregs[16] & 0xFFFFFFDF;
    if(ptrace(PTRACE_SETREGS, pid, NULL, (void *)&regs) < 0 || ptrace(PTRACE_CONT, pid, NULL, NULL) < 0 )
    {
        perror("[WARNING] call malloc() failed !\n");
        return NULL;
    }
    waitpid(pid, &status, WUNTRACED);
    while (status != 0xb7f)                 // 过滤因返回地址为0触发异常而返回的错误码: 0xb7f
    {
        ptrace(PTRACE_CONT, pid, NULL, NULL);
        waitpid(pid, &status, WUNTRACED);
    }
    ptrace(PTRACE_GETREGS, pid, NULL, (void *)&regs);
    void *ret = (void *)regs.uregs[0];
    // 寄存器出栈
    ptrace(PTRACE_SETREGS, pid, NULL, (void *)&pushed_regs);
    printf("[MESSAGE] malloc ret mem: 0x%lx\n", (ul)ret);
    return ret;
}
```

以下是ptrace注入整体框架代码，共分为4步：

1. 获取进程pid
2. 附加到进程com.example.crackme1
3. 远程调用malloc分配内存，远程调用dlopen加载libhook.so
4. 解除附加

```
 ///***** 1. 获取进程pid *****
    sprintf(process, "pidof %s", argv[1]); // 用shell指令得到pid
    if((fp = popen(process, "r")) == NULL )
    {
        printf("[WARNING] process not found !\n");
        return -1;
    }
```

```
    fread(pbuffer, 1, 256, fp);
    sscanf(pbuffer, "%d", &pid);
    printf("[MESSAGE] pid is %d\n", pid);
    pclose(fp);

///***** 2. 附加到进程 *****
    const char *lib_dir = argv[2];
    if(ptrace(PTRACE_ATTACH, pid, NULL, NULL) < 0)
    {
        perror("[WARNING] failed to attach to the process !\n");
        return -1;
    }
    int status;
    waitpid(pid, &status, 0);
    printf("[MESSAGE] STATUS : 0x%x\n", status); // 0x137f
    printf("[MESSAGE] succeed to attach to the process !\n");

///***** 3. 远程函数调用 *****
    // 远程调用malloc，为hook分配内存，写入lib文件路径
    void *buffer = mallOc(pid, 0x800);
    if( buffer == NULL )
    {
        perror("[WARNING] malloc failed !\n");
        printf("\n getchar() debugging ......\n");
        getchar();
        ptrace(PTRACE_CONT, pid, NULL, NULL);
        ptrace(PTRACE_DETACH, pid, NULL, NULL);
        return -1;
    }
    // 将hook lib写入进程内存
    ptrace_poketext(pid, strlen(lib_dir) + 1, (void *)lib_dir, buffer);
    // 将lib文件加载到进程
    if( dlopen(pid, buffer, RTLD_LAZY) == NULL )
    {
        perror("[WARNING] load lib to process failed !\n");
        return -1;
    }

///***** 4. 解除附加 *****
    printf("[MESSAGE] injection success !\n");
    ptrace(PTRACE_CONT, pid, NULL, NULL);
    ptrace(PTRACE_DETACH, pid, NULL, NULL);

```

进入实际操作环节，我尝试了LeiDian9 、MuMu12、MuMu6、VMOS Pro等多款不同操作系统上的安卓虚拟机，但是均出现一些无法解决的难题，困难之下只好借来能root的小米手机，用真实机进行测试。

> LeiDian9、MuMu12等模拟器都有一个共同问题，即ptrace_attach附加后com.example.crackme1就会自动闪退，无法进行正常注入
>
> VMOS Pro模拟器则是另外一个问题，即在com.example.crackme1的maps中没有找到libdl.so库，无法按照我的思路进行远程dlopen函数调用

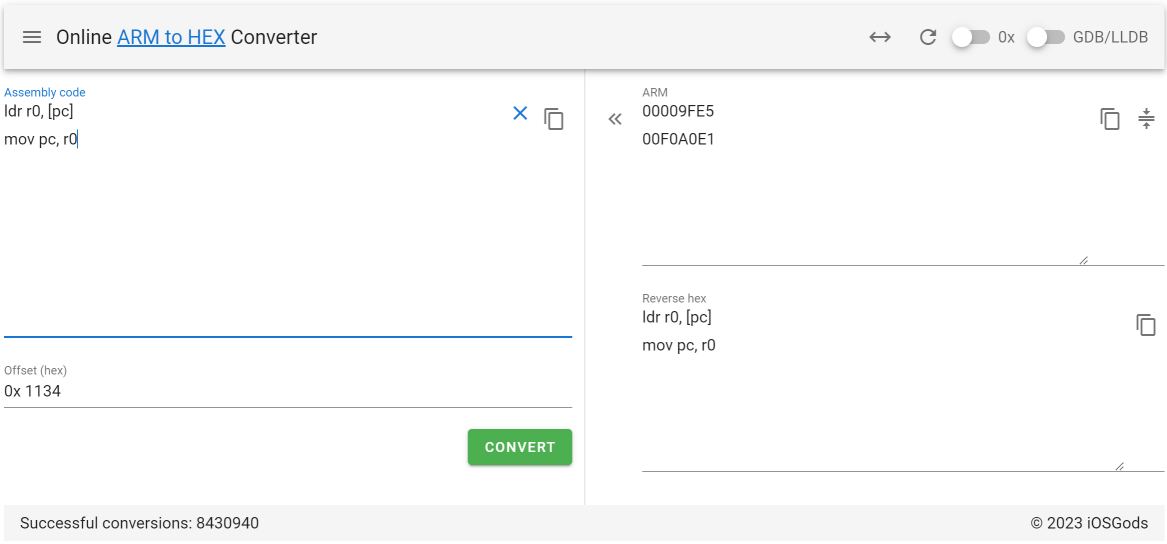
下图为真实机测试过程，运行./injector进行注入，可以在c`at /proc/21539/maps | grep libhook.so`后的列表中找到注入的库。

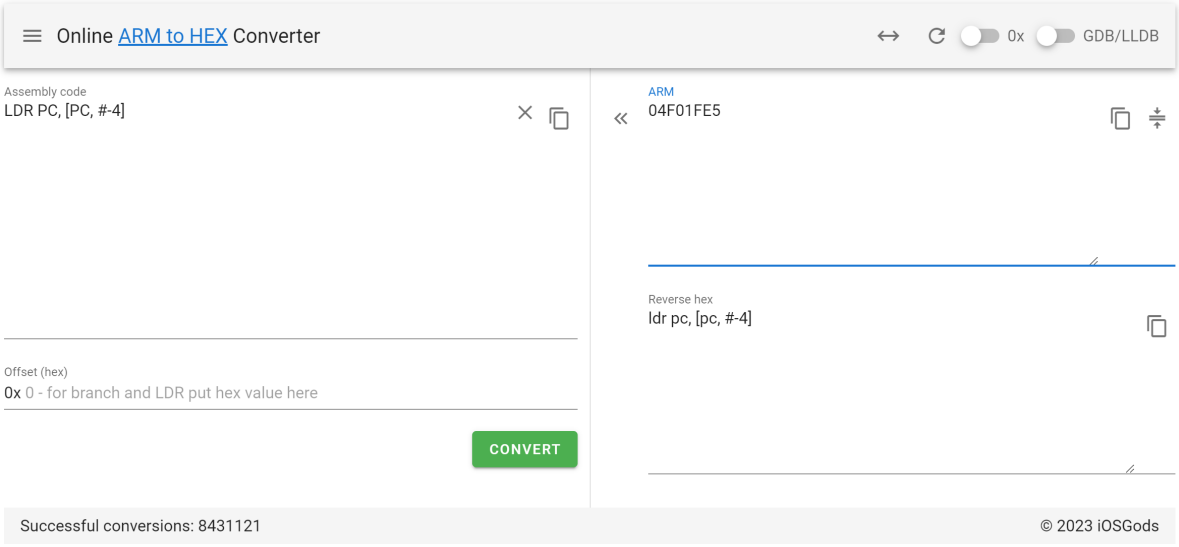
```
/injector com.example.crackme1 /data/local/tmp/ping/libhook.so
[MESSAGE] pid is 21539
[MESSAGE] STATUS : 0x137f
[MESSAGE] succeed to attach to the process !
[MESSAGE] /apex/com.android.runtime/lib/bionic/libc.so at mem 0xeeccc000 in process 21539
[MESSAGE] Remote call: malloc(0x800)
[MESSAGE] malloc ret mem: 0xef959000
[MESSAGE] /apex/com.android.runtime/lib/bionic/libdl.so at mem 0xed5c2000 in process 21539
[MESSAGE] Remote call: dlopen(0xef959000, 0x1)
[MESSGAE] STATUS: 0xb7f
[MESSAGE] dlopen ret handle: 0x49503ef7
[MESSAGE] injection success !
lmipro:/data/local/tmp/ping # cat /proc/21539/maps | grep libhook.so
e69da000-e69db000 r--p 00000000 103:12 281557                           /data/local/tmp/ping/libhook.so (deleted)
e701e000-e701f000 r--p 00000000 103:12 335457                           /data/local/tmp/ping/libhook.so (deleted)
e701f000-e7020000 r-xp 00000000 103:12 335457                           /data/local/tmp/ping/libhook.so (deleted)
e7020000-e7021000 r--p 00000000 103:12 335457                           /data/local/tmp/ping/libhook.so (deleted)
e7687000-e7688000 r--p 00000000 103:12 272916                           /data/local/tmp/ping/libhook.so (deleted)
e7688000-e7689000 r-xp 00000000 103:12 272916                           /data/local/tmp/ping/libhook.so (deleted)
e7689000-e768a000 r--p 00000000 103:12 272916                           /data/local/tmp/ping/libhook.so (deleted)
e768a000-e768b000 rw-p 00000000 103:12 272916                           /data/local/tmp/ping/libhook.so (deleted)
e7801000-e7802000 r--p 00000000 103:12 295095                           /data/local/tmp/ping/libhook.so
e7802000-e7803000 r-xp 00000000 103:12 295095                           /data/local/tmp/ping/libhook.so
e7803000-e7804000 r--p 00000000 103:12 295095                           /data/local/tmp/ping/libhook.so
e7804000-e7805000 rw-p 00000000 103:12 295095                           /data/local/tmp/ping/libhook.so
e78dc000-e78dd000 r--p 00000000 103:12 281557                           /data/local/tmp/ping/libhook.so (deleted)
e78dd000-e78de000 r-xp 00000000 103:12 281557                           /data/local/tmp/ping/libhook.so (deleted)
e78de000-e78df000 r--p 00000000 103:12 281557                           /data/local/tmp/ping/libhook.so (deleted)
e790f000-e7910000 r--p 00000000 103:12 281557                           /data/local/tmp/ping/libhook.so (deleted)
```

3. inline hook替换proc_0x1134

接上面apk分析中的内容，通过`inline hook`的方式，在`proc_0x1134`的原位置调用`proc_hook`函数，使其恒返回1，从而使得`stringFromJNI`恒返回right。

已知`proc_hook`函数的地址，需要实现插入偏移地址0x1134处的inline hook指令代码。下两张图是两种hook指令示例：





这里**直接将proc_hook的地址插入到PC寄存器**，可以实现强行跳转，注意**此时的堆栈中保存的返回地址仍然是调用proc_0x1134前保存的地址**，故可以在proc_hook执行结束后直接回到调用proc_0x1134前的位置，即完成了inline hook。

```
注意ARM状态下，PC寄存器的值为当前指令地址+8字节

0x1134  LDR R0, [PC]         00 00 9F E5
0x1138  MOV PC, R0           00 F0 A0 E1
0x1142  proc_hook

或者

0x1134  LDR PC, [PC,#-4]     04 F0 1F E5
0x1138  proc_hook
```

用Visual Studio 模版创建一个Android平台动态共享库项目，设计hook函数，**核心思想**就是向内存中写入hook指令，**让proc_hook替代proc_0x1134执行。**


```
int proc_hook()
{
    LOGI("Succeed to hook, return 1 all the time !");
    return 1;
```

```
}

void hook()
{
    unsigned char hookCommand[12] = { 0x00, 0x00, 0x9F, 0xE5, 0x00, 0xF0, 0xA0, 0xE1 };
    void* lib_base = NULL;                  // 用于存放libhook.so的基地址
    unsigned long offset_0x1134 = 0x1134;   // 目标函数偏移地址
    void* addr_0x1134;                      // 目标函数地址
    FILE* fp;
    // 遍历/proc/com.example.crackme1/maps，查找libhook.so地址
    fp = fopen("/proc/self/maps", "rt");
    do{
        fgets(line, 1024, fp);
        if (strstr(line, "libcrackme1.so") != NULL){
            sscanf(line, "%lx", (ul*)&lib_base);
            LOGI("libcrackme1.so at 0x%lx", (ul)lib_base);
            // 计算目标函数真实地址
            addr_0x1134 = (void*)((ul)lib_base + offset_0x1134);
            LOGI("proc_0x1134 at 0x%lx", (ul)addr_0x1134);
            LOGI("proc_hook at 0x%lx", (ul)hookProc);
            // 修改段保护
            mprotect(lib_base, 0x2000, PROT_READ | PROT_WRITE | PROT_EXEC);
            // 补充跳转地址为hookProc
            *(ul*)(hookCommand + 8) = (unsigned long)proc_hook;
            // 写入hook指令
            memcpy(addr_0x1134, (void*)hookCommand, 12);
            return;
        }
    } while (strlen(line));
        // 没找到 libhook.so
        LOGW("libcrackme1.so not found !");
        return;
}
```

hook函数设计好了，但是如何执行？非常感谢知乎大佬的一篇帖子: [constructor属性函数在动态库加载中的执行顺序]

> 用constructor属性指定的函数，会在目标文件加载的时候自动执行，发生在main函数执行以前，常用来隐形得做一些初始化工作。
>
> 函数声明：
>
> void **init()** attribute__((constructor));
>
> 也可以是静态函数：
>
> static void **init()** attribute__((constructor));

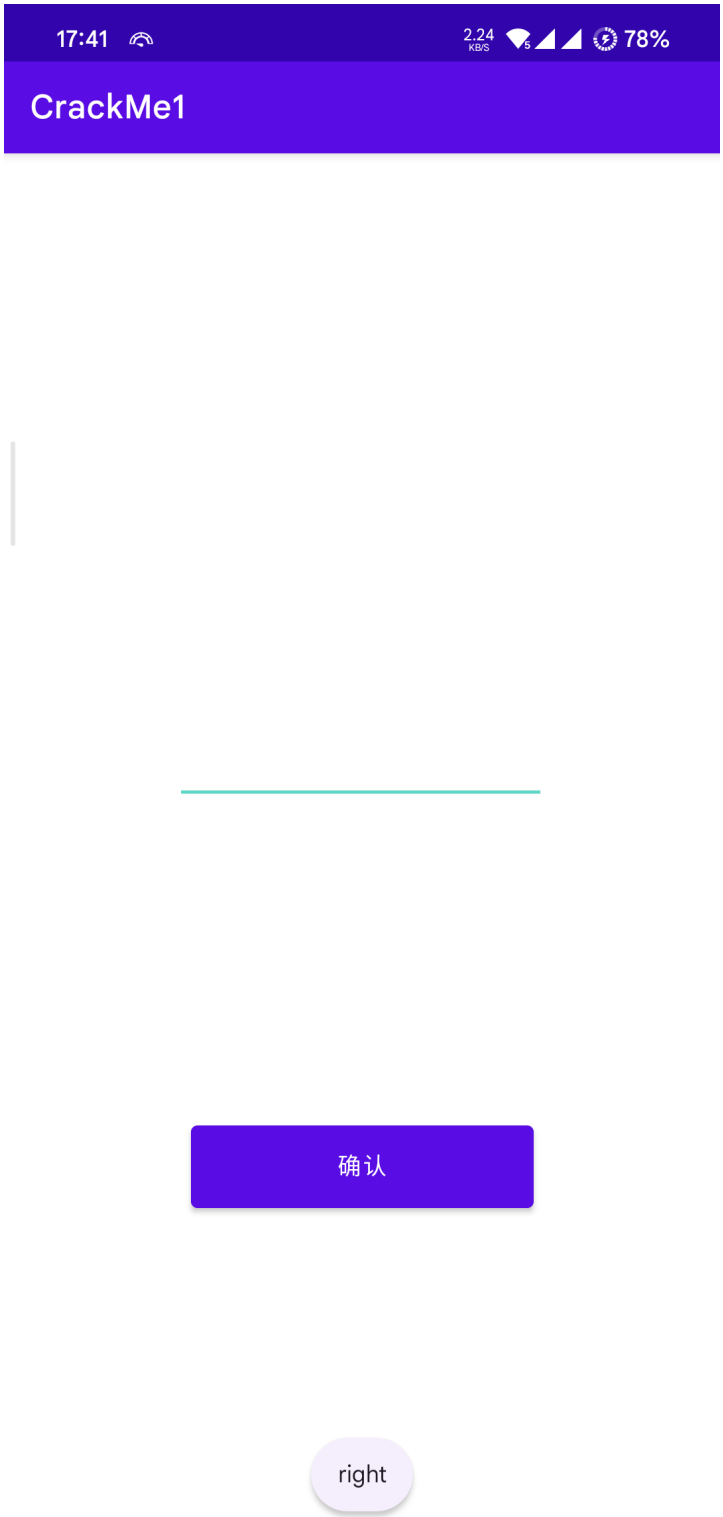
参照上述先验知识，只需要在libhooc.so中加入以下代码逻辑，就可以实现hook注入

```
void init() __attribute__((constructor));
void init()
{
    //  直接调用函数hook()
    hook();
    return;
}
```

实机hook：

```
[MESSAGE] pid is 21539
[MESSAGE] STATUS : 0x137f
[MESSAGE] succeed to attach to the process !
[MESSAGE] /apex/com.android.runtime/lib/bionic/libc.so at mem 0xeeccc000 in process 21539
[MESSAGE] Remote call: malloc(0x800)
[MESSAGE] malloc ret mem: 0xef3c1000
[MESSAGE] /apex/com.android.runtime/lib/bionic/libdl.so at mem 0xed5c2000 in process 21539
[MESSAGE] Remote call: dlopen(0xef3c1000, 0x1)
[MESSGAE] STATUS: 0xb7f
[MESSAGE] dlopen ret handle: 0x18d2e4a9
[MESSAGE] injection success !
lmipro:/data/local/tmp/ping # cat /proc/21539/maps | grep libhook.so
e69da000-e69db000 r--p 00000000 103:12 281557                           /data/local/tmp/ping/libhook.so (deleted)
e701e000-e701f000 r--p 00000000 103:12 335457                           /data/local/tmp/ping/libhook.so
e701f000-e7020000 r-xp 00000000 103:12 335457                           /data/local/tmp/ping/libhook.so
e7020000-e7021000 r--p 00000000 103:12 335457                           /data/local/tmp/ping/libhook.so
e78dc000-e78dd000 r--p 00000000 103:12 281557                           /data/local/tmp/ping/libhook.so (deleted)
e78dd000-e78de000 r-xp 00000000 103:12 281557                           /data/local/tmp/ping/libhook.so (deleted)
e78de000-e78df000 r--p 00000000 103:12 281557                           /data/local/tmp/ping/libhook.so (deleted)
e790f000-e7910000 r--p 00000000 103:12 281557                           /data/local/tmp/ping/libhook.so (deleted)
```

注入之后，点击CrackMe1窗口中的确认，即使什么都不填，也能返回`right`！


17:41
2.24
KB/S
78%
CrackMe1
确认
right