目录

[概述 1](#_Toc474104505)

[确定实现反调试逻辑的关键点 2](#_Toc474104506)

## 概述

反调试一般做法主要有如下方式：

1. 程序本身通过fork子进程ptrace保护自己，使得调试器附加失败
2. 定时检测进程在调试器附加后产生的特殊标志(/proc/[pid]/status的TracerPid)，若发现则产生异常或退出
3. 通过hook: write msync mmap等函数防止拷贝进程内存(dd命令)
4. 加入ollvm混淆代码，加密dex/so等文件(破坏头部)在内存动态解密

在migu sdk中，使用了2,3,4的手段。

sdk初始化时首先加载libmg20pbase.so，该so用于校验并加载assets目录下的

libmg20p\_??.??.??\_??.so，因此无法直接通过修改该so来达到破解目的，libmg20p\_??.??.??\_??.so负责fork出3个同样的进程，并互相定时监视对方的/proc/[pid]/stat调试位标志，同时建立和其他进程的通信，若其中某一个进程调试标志出现或无法通信则立刻调用exit和kill将其杀死；而检测调试位的标志，通过逆向经过大量混淆的代码可以发现，并不是检测传统的/proc/[pid]/TracerPid而是/proc/[pid]/status，该文件其中存着进程事实状态数据，而第二个数据为是否处于被跟踪状态，在调试器附加之后该标志会从S变为t或T，如下图所示：

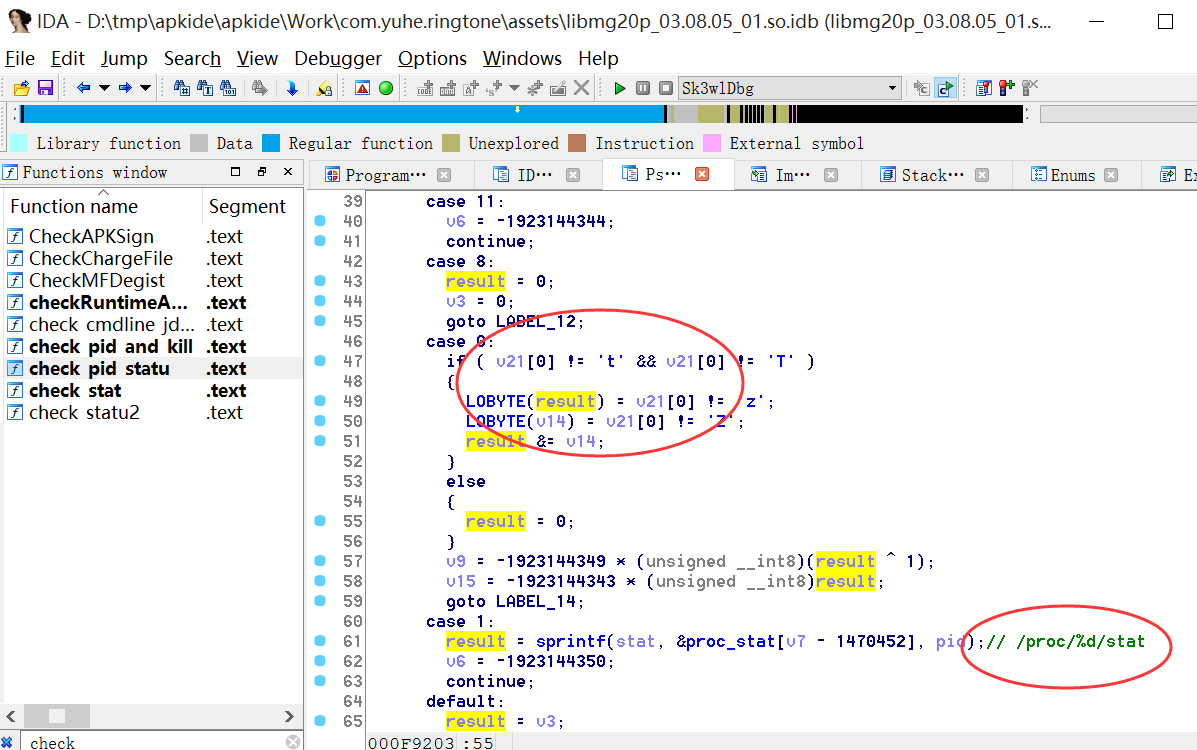


图1

针对以上限制，我们可以做的有：

1. 尝试修改read /proc/%d/stat的读取结果，将t/T改回其他字符即可。因为程序中使用read函数，因此我们需要使用hook框架substrate(支持android5.0之前的系统)进行hook，或在程序最开始运行时加载自己的hook动态链。sdk中并未做深入检查因此got表hook和内联hook均可。
2. 由于程序只对主进程的stat进行检测，因此可以通过附加task进行绕过。先通过/proc/[pid]/task找到该进程所有taskid，然后使用gdbserver命令附加taskid即可，这样可以不做任何操作绕过检测。该方式不适合android\_server。

libmg20p\_??.??.??\_??.so还会加载libmiguED.so，而libmiguED.so会用libmg20p\_??.??.??\_??.so导出的write msync mmap挂钩系统原始函数，代码逻辑是检测特殊的头部(例如odex的文件头为dey035)，然而为了防止逆向，在内存中头部是已经破坏了的，因此防止内存dump并没有意义。同样使用hook框架substrate取出加密的dex文件。

sdk中存在大量混淆，在任何一个函数中均存在，而sdk本身又存在反调试，不过由于so文件未加密因此可以进行静态分析，而使用ida虚拟机调试插件Sk3wlDbg可以进行简单的分析，以便跟踪代码流程。Arm和x86的so代码逻辑相同，而ida对于arm的switch指令分析有误，使用x86版本更容易分析，之后再同arm对照即可。这里我对lib目录下的libmg20pbase.so和asset目录下的libmg20p\_??.??.??\_??.so进行部分分析

## 建立工作环境

Android Studio

Ida6.8破解版

Root过的android手机 或 逍遥模拟器

Jeb1.5

Substrate hook框架 的安装

(由于较危险，建议在模拟器上测试)，系统要求：root过的android系统，版本<=4.4

下载安装<http://www.cydiasubstrate.com/download/com.saurik.substrate.apk>，选择“Link Substrate Files”

使用我提供的dexdump项目，在test.cpp中做修改即可，使用android studio编译安装即可

Sk3wlDbg插件的安装 (该插件用于跟踪混淆过得流程和解密字符串；我已集成在ida中)

下载Python module for windows 32 版的unicorn引擎 http://www.unicorn-engine.org/download

下载编译<https://github.com/cseagle/sk3wldbg>，将生成的plw文件和git目录下bin/windows下的文件以及unicorn的其余依赖dll拷贝到ida的plugin目录下

## 确定实现反调试逻辑的关键点

首先研究migu给的demo，使用android\_studio编译运行，看到app启动后，存在3个同名进程，而用调试器(android\_server/gdbserver)附加任何一个，很快3个进程都被杀死(常见的反调试保护方式，调试器会捕获到kill信号)；尝试strace跟踪不久进程也会退出；而使用android\_studio调试java层代码则没有任何问题，因此反调试的保护只存在于jni层。因此如果只需要调试java层的代码则无需处理反调试，只需要附加到主进程即可。通过demo的源码MiguApplication.java的onCreate函数可知首先加载的是libmg20pbase.so。由于检测反调试常见的方法要么是实现ptrace防止调试器再次ptrace，要么是创建一个线程死循环调试位标志。前者的现象是调试器在附加时失败，而后者的现象是进程退出。因此需要找到第二种情况的痕迹，重点在于对pthread\_create exit kill等函数的调用分析。通过对libmg20pbase.so的简单分析，可以发现并没有显著的反调试逻辑存在，本身为校验so完整性防篡改。通过java层调试信息可知其后加载libmg20p\_??.??.??\_??.so，我分析的版本是libmg20p\_03.08.05\_01.so，接着再分析这个文件，它存在8次线程创建操作(idb文件中标记为函数thread\_1~7)，和多次exit kill操作。最终在thread\_4 thread\_5 thread\_6 thread\_7函数中发现存在2个重要函数check\_stat和check\_stat2，用于检测/proc/[pid]/stat调试标志，如图1所示。2个函数逻辑相同，区别仅在于加密字符串的算法稍有差别，最终都在内存解密出/proc/%d/stat，加密的字符串我标记为proc\_stat。

/proc/[pid]/stat

以上仅是针对静态代码进行分析，但是由于存在反调试，这里利用hook框架验证之前的猜想，hook libc.so的exit和kill函数：

pid 10723 exit

#02 pc 000f2839 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

#03 pc 000f391d /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

#04 pc 0005e084 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

#05 pc 00061367 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

#02 pc 000f2ee6 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

#03 pc 000f3321 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

#04 pc 000f3f06 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

#05 pc 0005e084 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

#06 pc 00061367 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

pid 10715 kill pid 10680 sig 9

#02 pc 000f229a /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

pid 10714 kill pid 10680 sig 9

#02 pc 000f2ee6 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

#03 pc 000f3321 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

#04 pc 000f33f5 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

pid 10714 kill pid 10715 sig 9

#02 pc 000f2ef6 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

#03 pc 000f3321 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

#04 pc 000f33f5 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

pid 10714 exit

#02 pc 000f2f07 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

#03 pc 000f3321 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

#04 pc 000f33f5 /data/data/com.cmsc.cmmusic.common.demo/files/libmg20p\_03.08.03\_00.so

## 破解反调试机制

由于破解反调试机制，只需要破坏检测逻辑中任何一环即可，因此可在如下方面入手：

1. 重编译linux系统，使ptrace附加时不产生标志位；这种方法缺点是技术上复杂且耗时
2. 不直接附加进程id，而附加taskid，这样不会在/proc/pid/stat中留下痕迹，因此无法检测到；这种方式适用于gdbserver直接调试，而由于ida的android\_server不认taskid，因此无法用ida完美调试
3. 修改/proc/%d/stat的加密字符串为其他值；在demo中，存在2处字符串值，需要修改内存权限为读写然后修改掉该处；这种方式不通用，一旦so存在改动则需要重新分析定位
4. 尝试挂钩fopen open read fread等函数；libmg20p\_03.08.05\_01.so中使用了open read，一旦读取到/proc/[pid]/stat则修改返回结果，挂钩的方式可以有2种，第一种是利用hook框架substrate，另一种是以调试模式启动app，在libmg20p\_03.08.05\_01.so加载之前加载自己编译的用于hook的so
5. 拦截exit kill等函数；这种方式在demo中不适用，实测若检测到调试标志，则会循环exit和kill，app一直卡死

下面是我采取的过反调试方法，可直接用于安装了substrate hook框架的root系统中：

ssize\_t (\*old\_read)(**int** fd, **void** \*buf, size\_t count);  
ssize\_t new\_read(**int** fd, **void** \*buf, size\_t count)  
{  
 ssize\_t len = old\_read(fd, buf, count);  
 **char**\* ptr = (**char**\*)buf;  
 **for**(**int** i=0;i<32;i++)  
 {  
 **if**(ptr[i] == **' '** && (ptr[i+1] == **'T'** || ptr[i+1] == **'t'**))  
 {  
 LOGI(**"read hit %d"**, getpid());  
 ptr[i+1] = **'S'**;  
 }  
 }  
 **return** len;  
}

对于android 5.x以上，由于substrate框架不支持，因此需要自己实现hook

## 实现got表hook

实现在gothook/jni/hook.cpp，在jni目录下执行ndk-build编译即可生成libhook.so

在该文件中，有下面的要点：

对于无hook框架存在的情形下，由于等到libmg20p\_03.08.03\_00.so执行了fork之后，已经产生3个进程互相监视，因此已经无法获取控制权，所以要获取控制权，要在fork之前进行，这个点可以在启动app的时刻到fork之前均可。选好点之后，加载我们的libhook.so进行挂钩即可突破反调试。此时对主进程(进程id最小的)进行附加即可。这里我采用重打包方式，将加载libhook.so的逻辑嵌入到java代码中，而且刚好在加载libmg20pbase之前，具体如下一节所述，我已重编译好新的愈合之声，无需任何操作，即可附加调试。本节只关注实现去除反调试的细节。

Main函数为入口，一旦加载libhook.so就会执行，里面分别对不同版本android系统做处理(android>5.0需要hook libart.so android<=4.4需要hook libdvm)，hook该模块的dlopen的原因是，libmg20pbase会使用jni层接口反调System.load(“libmg20p\_03.08.03\_00.so”)，最终会调用libdvm和libart中的dlopen。因此我们通过dlopen捕获到libmg20p\_03.08.03\_00.so的加载完成事件。通过逆向分析可知，该模块的反调试操作在jni\_onload中完成，而这一步发生在dlopen之后。因此我们有绝对的时机进行控制。在此时，对open函数和usleep函数进行hook。

对usleep函数hook的原因是，检测调试器标志的线程是定时的，我们把usleep的时间改大一些，好不让检测线程那么频繁；而对open的hook函数new\_open正是检测是否当前线程在检测调试标志位/proc/pid/stat，如果是我们给他返回/proc/1/stat作为欺骗。此外我对exit和kill也进行了打印，目的是防止以后该sdk升级，采用了什么新的方式反调试，通过打印回溯栈就可以找到监测点。其他需要解释的，在hook.cpp中有说明。

## 愈合之声重打包

我认为最佳的方式是重打包，恰巧该app没有做重打包的防护。我的目的是把加载libhook.so的代码嵌入到app代码中。（我已编译好apk，但是如果想自己动手，可参考过程如下：

1.ndk-build编译本hook工程，生成libhook.so，将对应文件拷贝到要破 解的app files目录下

->ndk-build (cd到jni目录执行ndk-build，生成libhook.so)

->用apk改之理反编译愈合之声app，目的是将libhook.so添加到java代码进行加载

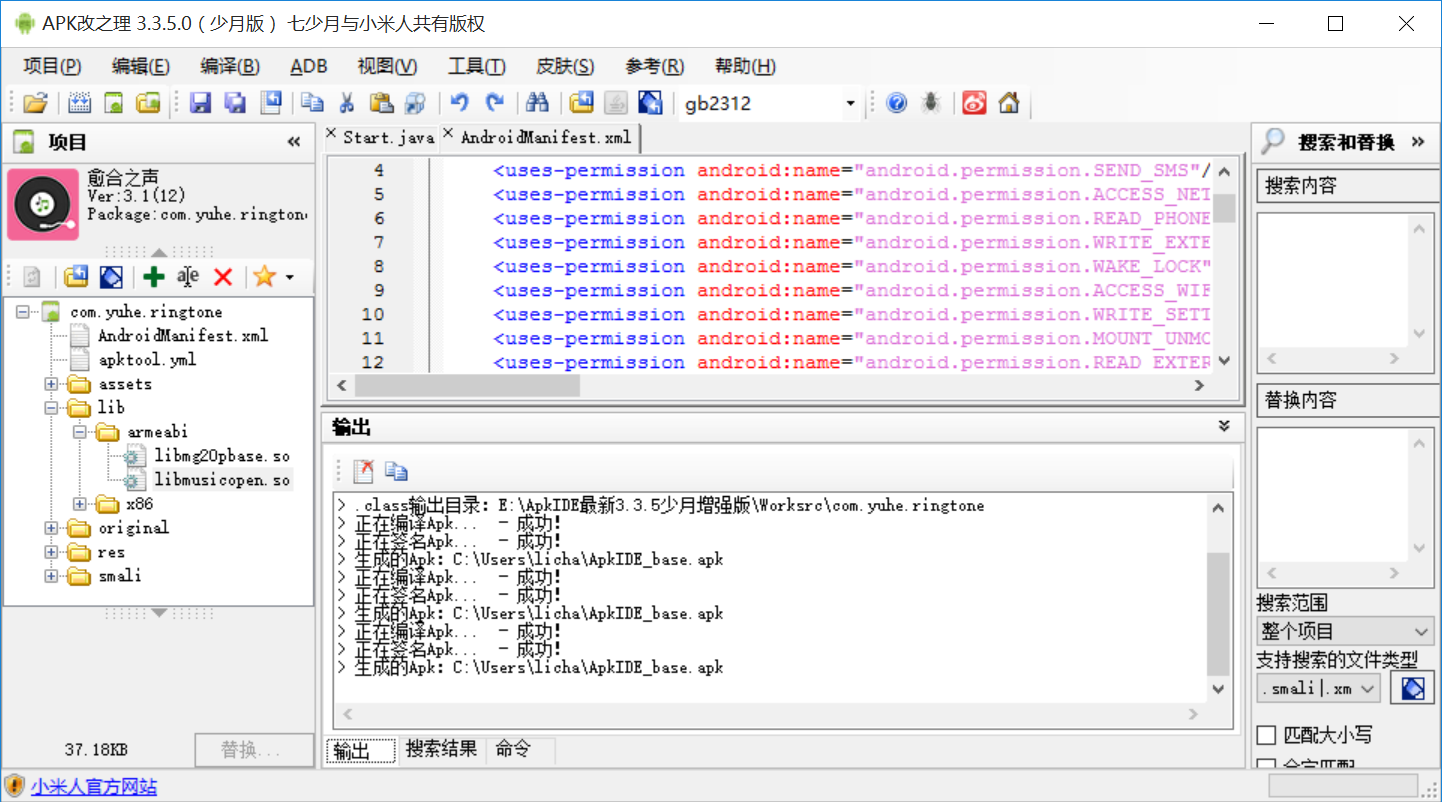
->在反编译的smali文件中(apk改之理\Work\com.yuhe.ringtone\smali\com\yuhe\ringtone\AppApplication.smali)，找到函数“.method public onCreate()V”，在“const-string v1, "mg20pbase" 前加入：

const-string v2, "hook"

invoke-static {v2}, Ljava/lang/System;->loadLibrary(Ljava/lang/String;)V

同时将libhook.so添加到 apk改之理\Work\com.yuhe.ringtone\lib\armeabi\下，使用apk改之理重编译成apk即可

在后续研究发现，apk中是存在签名校验的，具体逻辑没有深入，不过通过另类的方法不使用重打包进行libhook.so的注入，这种方式是通过adbi库对zygote进程注入完成的。



## 整合到源码

如果已经有愈合之声app的源码，那么可以更简单的如下操作：

1.将本工程添加到jni代码，在System.loadLibrary("libmg20base")之前加入System.loadLibrary("hook")

或者参考无源码方式，将libhook.so存放到合适的目录，使用System.load(绝对路径)

2.编译运行 即可使用gdbserver android\_server androidstudio的lldb 等进行调试或strace跟踪

## 调试工具

简要描述一下调试器的使用

Android arm上常用gdbserver和ida作为c层调试器。而咪咕sdk也是对c层调试器的检测，如果是调试java代码则无需该操作

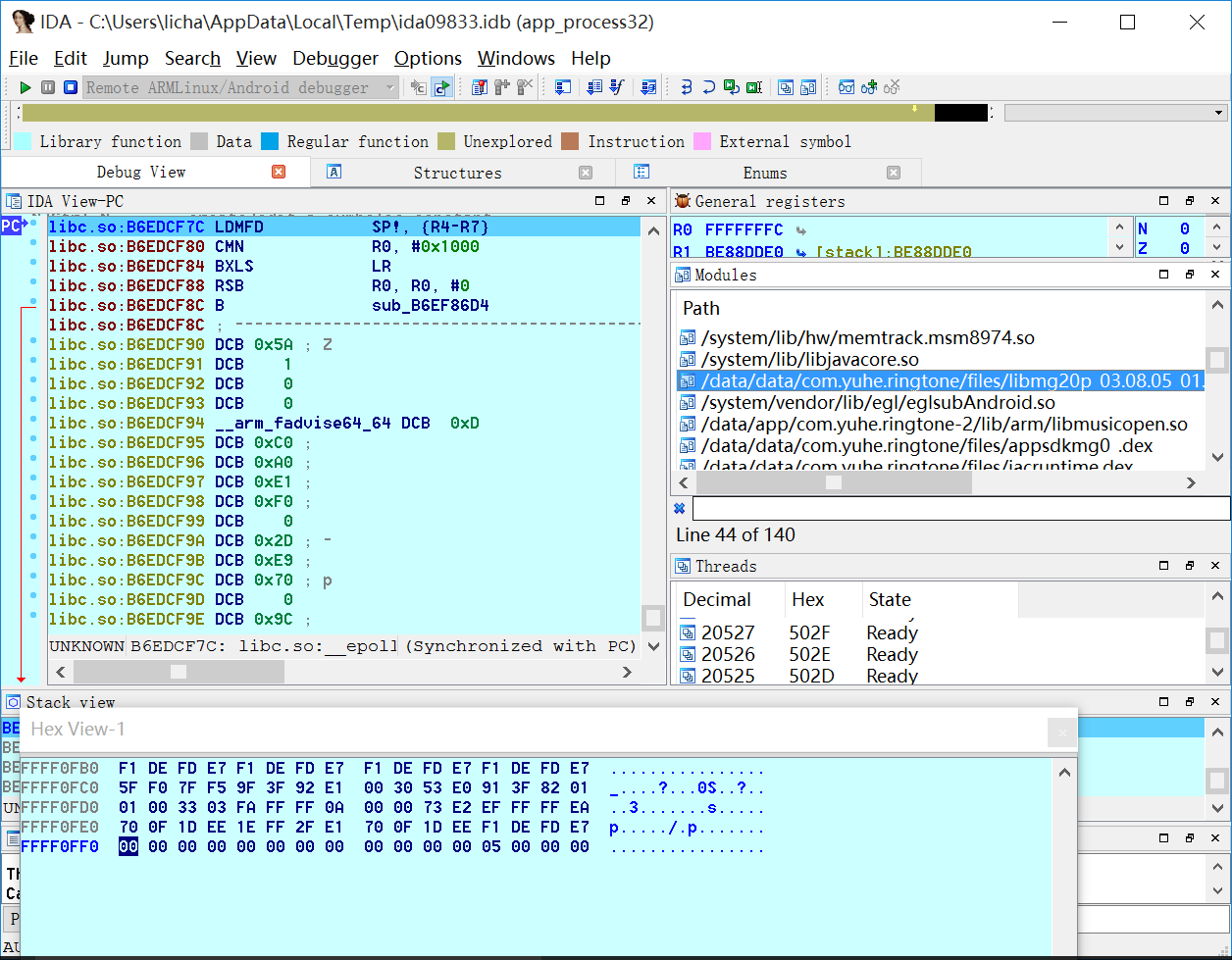
gdbserver调试：

1. Adb push gdbserver /data/local/tmp (从%NDK%\prebuilt\android-arm\gdbserver\gdbserver拷贝到手机)
2. Adb shell chmod 777 /data/local/tmp/gdbserver
3. Adb shell -> 在终端执行 su命令，以便以root权限运行gdbserver
4. /data/local/tmp/gdbserver \*:111 --attach pid (pid为用ps命令查看愈合之声pid中最小的那个，也是父进程)
5. 另开一个主机命令行，执行adb forward tcp:111 tcp:111 用于将手机的端口和主机调试端口绑定
6. 执行%NDK%下的gdb程序，使用target remote :111连接手机，即可完成调试初始化过程

还有些细节请参考我的 android逆向笔记 这篇文章的gdb调试部分章节

Ida调试：

1. 将ida目录下android\_server拷贝到手机/data/local/tmp/
2. chmod修改执行权限
3. Root权限在手机上运行android\_server
4. Adb forward tcp:23946 tcp:23946 转发ida调试端口
5. Ida调试器选择Remote Armlinux/Android debugger，填写主机：localhost和端口：23946
6. 选择app进程id，附加即可



## Dex的脱壳