

**课 程 实 验 报 告**

**课程名称： 逆向工程分析技术**

**专业班级： 信息安全1805班**

**学 号： U201810679**

**姓 名： 蒋树齐**

**指导教师： 鲁宏伟**

**报告日期： 2021年4月11日**

**网络空间安全学院**

1. **实验准备**
2. Android手机；（使用老师同款夜神模拟器）
3. AndroidKiller软件；
4. IDA Pro 7.0软件。
5. 安装java sdk
6. **实验内容**

结合Android程序实例“AndroidCrackme.apk”，完成以下工作：

1. 分析Android程序执行的过程，及核心代码的地址和范围；
2. 程序中关键信息混淆的方法，并还原相关内容；
3. 程序中反调试方法；
4. 还原程序中被加密的代码，结合还原后的代码重新打包生成程序，并确保程序能够正常运行；
5. 在上述工作基础上，获取实例代码中的“flag”，并进行验证。
6. **实验目标**

（1）通过上课所学定位核心代码

（2）获取实例代码中的“flag”，并进行验证。

1. **分析方法和过程**

**4.1 利用apk文件获取信息**

获取实验的apk文件，并用Android Killer打开，（需要先配置java sdk环境）即可看到该apk文件的相关信息例如名称，包名（这个非常重要后续会用到），入口和Activity。具体信息如下：



图 1apk信息

点击入口，可以直接跳转到入口的smali代码，分析代码，根据老师上课所讲，关键代码并不会放在其中，所以优先关注到了securityCheck函数，因为该文件中没有他的声明，其他函数类似于信息显示的功能，再关注到其前缀native,所以更加确定该函数位于.so文件中也就是外部链接库中，可以找到在工程管理器的lib目录下。

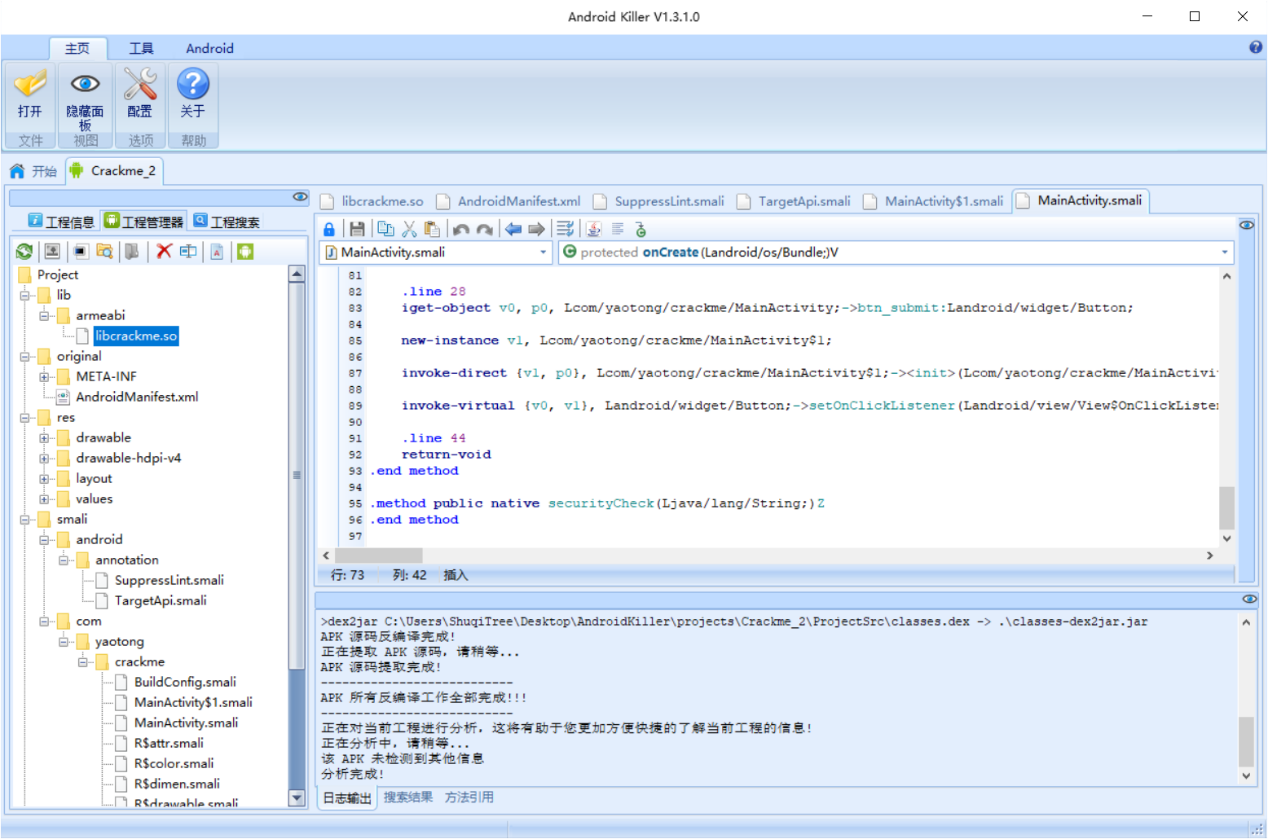


图 2定位关键代码

**4.2 获取.so文件信息**

Apk文件的本质其实就是压缩文件，可以同理与Macos下的dmg安装包文件，可以解压，为了获取.so文件，将apk后缀修改为zip后缀，使用Bandzip解压即可获得以下文件内容：



图 3修改后缀并解压

所以直接进入lib目录下找到.so文件，利用IDAPRO打开，点击Search->Text，输入securityCheck，从而直接搜索到该函数，结果如下：

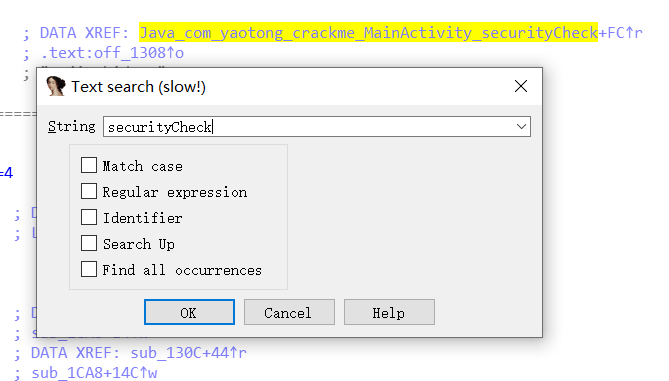


图 4搜索字符串

双击进入该函数，按下F5直接反汇编，获得代码如下：

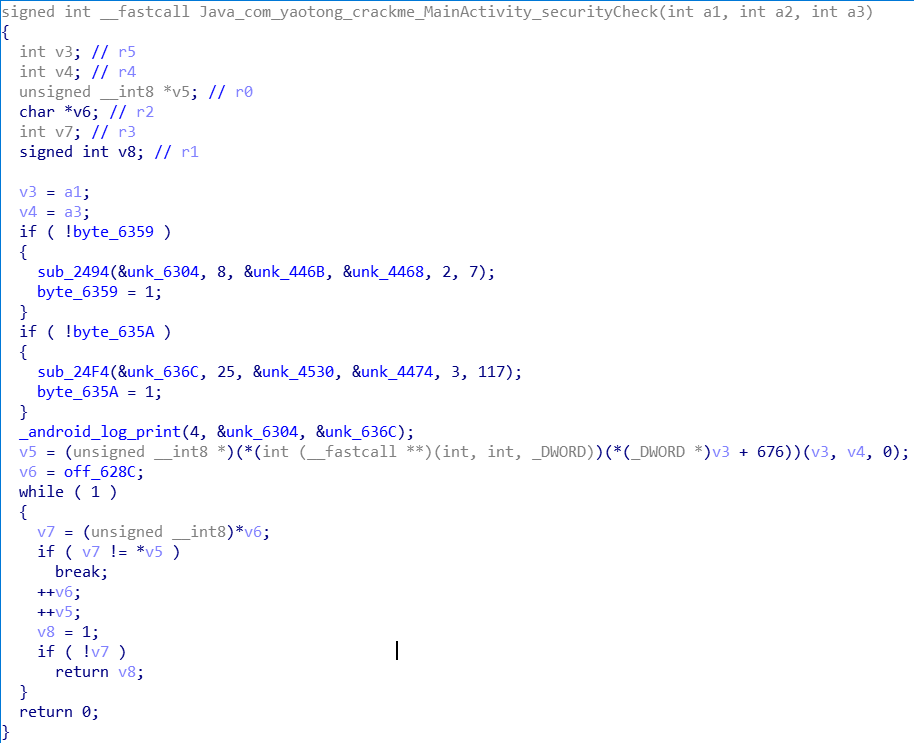


图 5反汇编代码

根据代码进行初步分析，由易到难，可以直接看懂的函数有\_android\_log\_print，应该用于输出程序界面的信息，剩下的函数都没什么头绪了。

接下来由上至下分析两个if判断中并没有获取什么信息，v5也过于复杂暂时跳过，看到一个while（1）的无线循环在死循环中把v6值赋予v7，并与v5进行比较后双方均向后移位。所以可以推测是进行字符串比对，若碰到了不同的就break然后return 0了，如果一直相同就会return v8也就是1。

由于v6是个常量，所以合理猜测v5那一行及其复杂的代码就是输入，因为我实在是没什么头绪分析那一段代码。（虽然后续找到了方法，但是此处就不写了，追求鲁老师教的一句话“不求甚解”，在后续分析中给出解释）那么地址“off\_628C”处的字符串应为正确答案即flag。

然后找到该地址的内容：



图 6off\_628C



图 7flag-1

**4.3 验证结果-1**

然后输入wojiushidaan，进行测试。当然在测试前要安装夜神模拟器，并在右侧安装apk按钮点击，安装该程序。测试结果如下：

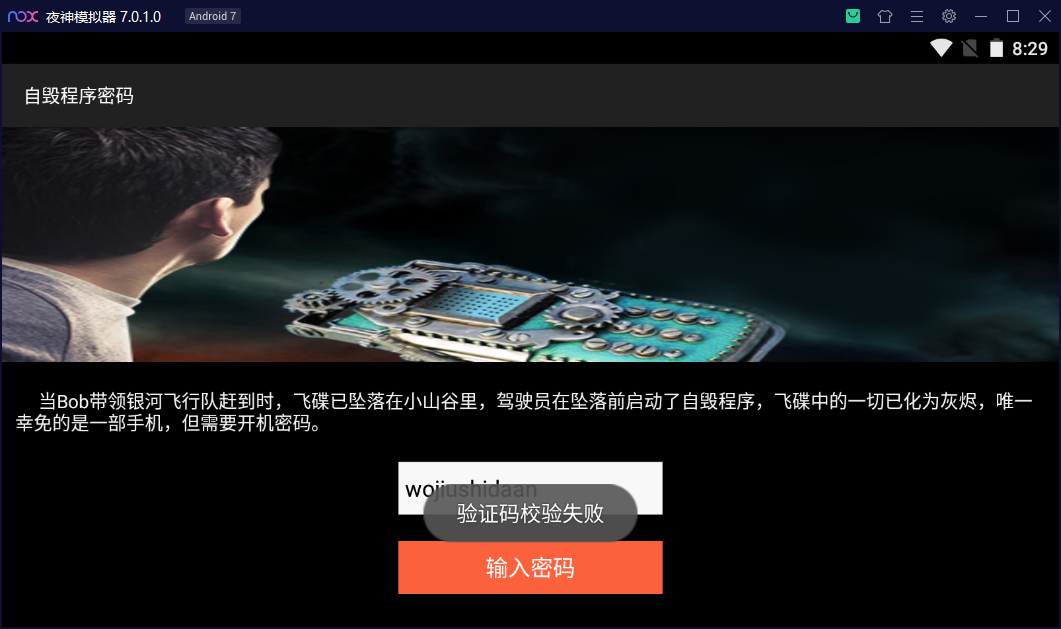


图 8验证失败

可以看到验证仍然是失败的。接下来展开动态调试。

**4.3 动态分析**

调试android程序本质上是两端之间的通信，以android为服务端，从pc客户端上设断点调试，故从IDAPRO的tools下面，将“android\_server”复制到手机上，扩大其权限之后，运行程序进行监听。操作如下图所示，可以看到android服务器端从默认端口23946开始监听。

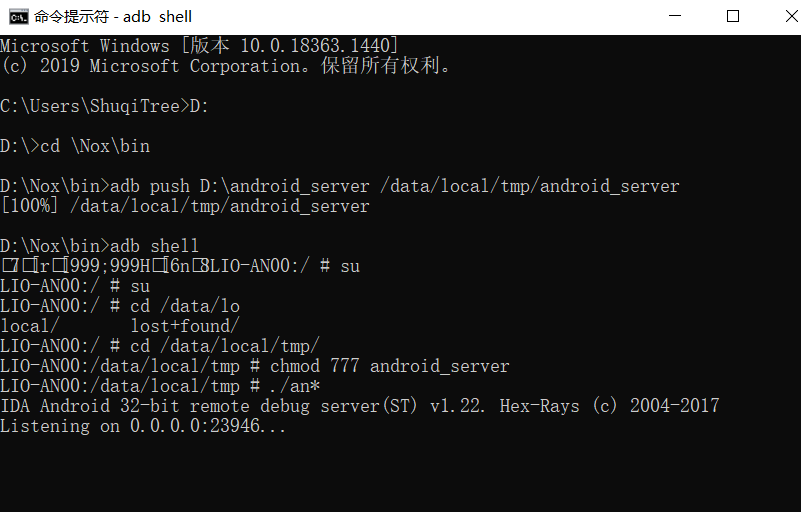


图 9开始监听

接下来进行下一步，使用“adb forward tcp:23946 tcp:23946”命令，将模拟器与pc机的23946端口相连接，然后从模拟器上启动“自毁程序密码”。

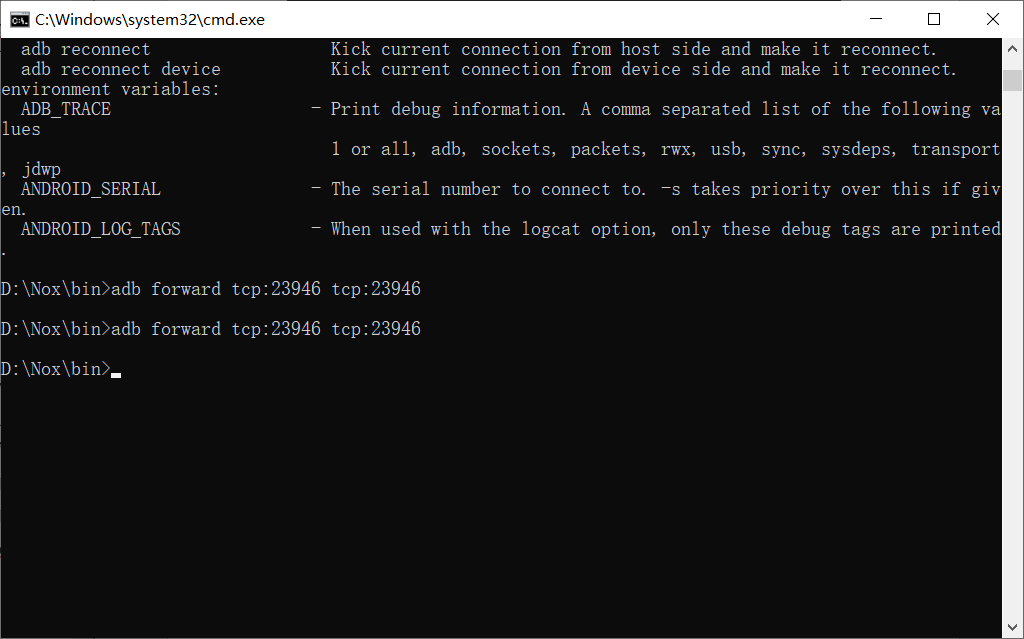


图 10adb forward

打开IDAPRO，点击GO，点击Debugger->Attach->Remote ARMLinux/Android debugger选项与夜神模拟器链接。HOSTNAME填写127.0.0.1，debug options配置如下即可

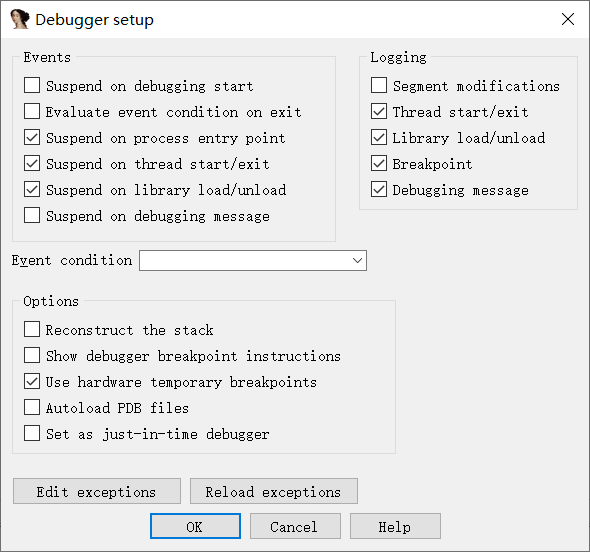


图 11debug options

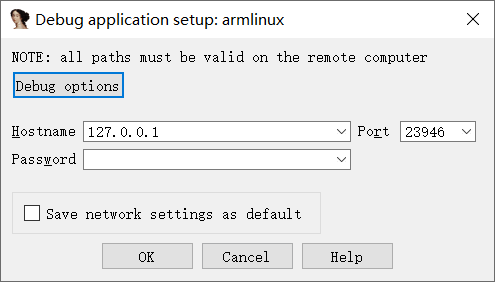


图 12设置hostname

选择模拟器上某一进程链接。注意到，其中存在包名com.yaotong.crackme，于是选择此程序链接。

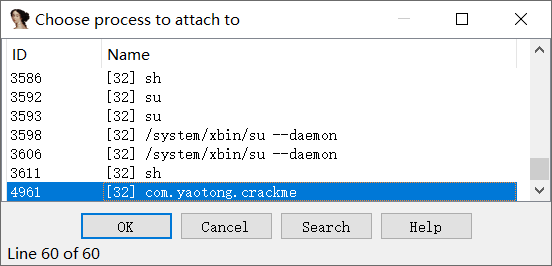


图 13选择进程

点击OK后即可进入。可以看到主界面是这样：

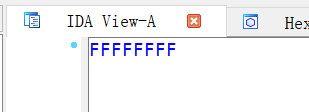


图 14FFFFFFFF

但是内存全是“？”非常的奇怪，后来我去咨询了其他的同学，使用夜神模拟器的同学无一例外不是这样，无法看到内存中内容，也就无法继续程序的分析，而且夜神模拟器在前面的步骤中就已经遇到了很多问题，比如adbshell的乱码，无论是使用chcp指令还是修改cmd的字体，都无法获得正常的编码信息，在网上查阅了很多资料都无法解决，虽然不影响使用但是看起来真的很难受，于是后续的实验，本人更换了模拟器，使用了进行安卓开发的AVD MANAGER和SDK MANAGER。

由于最近自己也在学习安卓和IOS的跨平台开发，使用Flutter，所以对模拟器的使用也相对熟悉。模拟器配置如下：

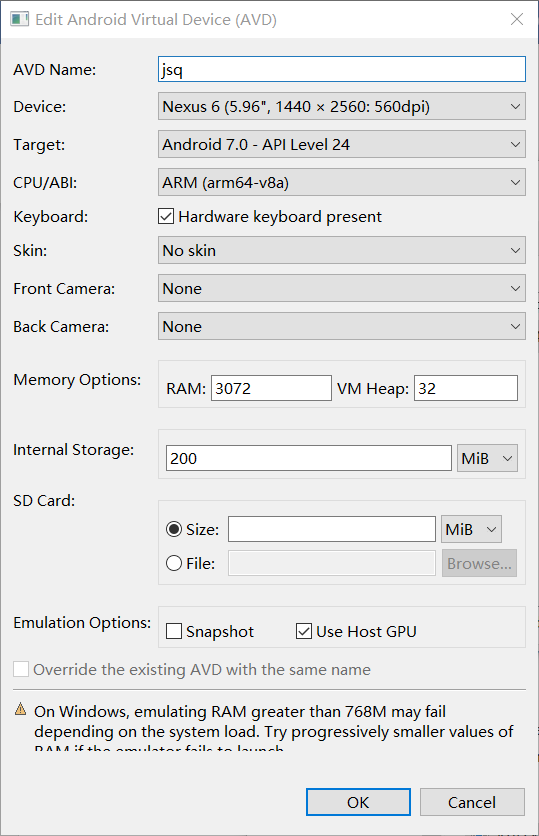


图 15模拟器配置

在更换了模拟器之后，前面的问题全部被一一解决，包括啊adb shell的乱码问题：正常的编码如下所示：

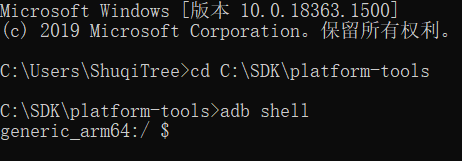


图 16乱码问题在更换模拟器后得到解决

而且关于夜神模拟器的坑还不止这些，还包括架构的问题，需要是Arm才可以正常。我在这里衷心的希望老师在下一届带这个课的时候可以不使用夜神模拟器，其他的模拟器包括android studio都很纯净，而夜神是开发给游戏玩家的，给他们用来在电脑上畅玩手机游戏的，并且内置了不少广告，上课的时候您也说了广告的事情，学生们还在一起笑，所以非常推荐SDK Manager这款模拟器！

上述的都是题外话，我们回到实验本身，在函数securityCheck起始设下断点，F9使程序运行，而程序并未在断点处停下，直接退出。

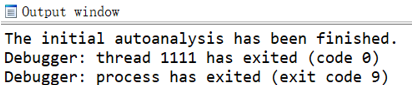


图 17意外退出

这时候主视图出现了和之前夜神模拟器一样的FFFFFFFF，这就是老师所说的，该程序设置的反调试防御，并且模拟器中的程序也自动推出了，接下来就进入实验最艰难的部分了。

**4.4 定位关键代码**

.so文件的加载过程步骤：

1. .init，此为虚拟机自带程序；
2. .init\_array，虚拟机调用此初始函数；
3. JNI\_Onload，安卓中的linker程序调用此函数；
4. Java\_com\_yaotong\_crackme\_MainActivity\_securityCheck函数，用于初始化程序，此函数运行之后才显示桌面窗口等信息。

由于.init函数是虚拟机自带的，难以被用户做手脚，故反调试代码及目标flag处理大概率在init\_array或JNI\_Onload中。所以这两个函数就成为了动态调试的关键。

**4.5 动态调试**

直接利用IDAPRO进行attach远程模拟器的时候并不能在其中打下断电，所以利用jdb工具进行接下来的调试。

启动android\_server监听，将两方23946端口链接。后使用如下图所示命令，启动crackme程序，可看到如下的等待调试界面。

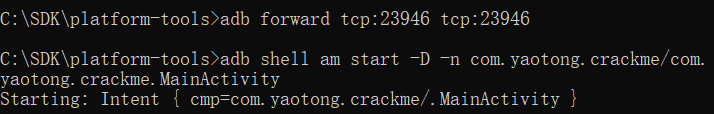


图 18调试指令

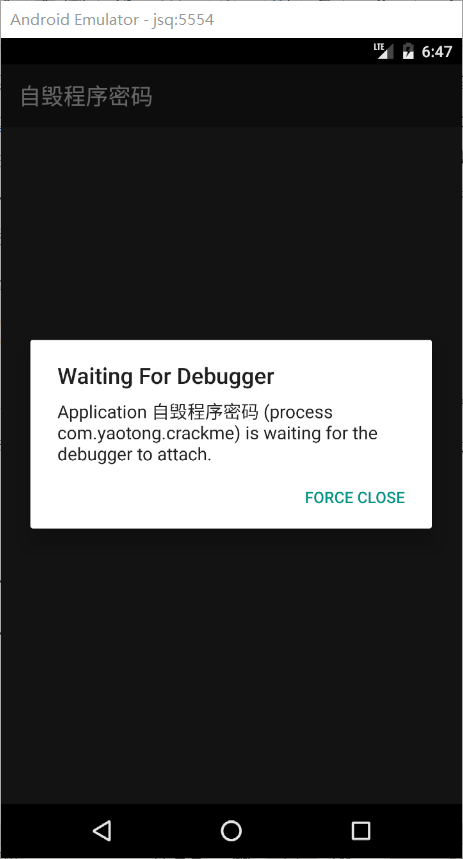


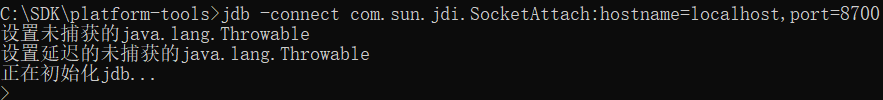
图 19等待调试界面

接下来打开IDAPRO进行之前的操作，但是要留意程序的端口：1864



图 20端口

将1057端口与模拟器8700链接，然后connect，命令如图22所示。



接下来就是正式进入动态调试的过程了。观察libcrackme.so库文件，发现一共有两个函数，一个是入口函数MainActivity，另一个叫做Jolin。

结合静态调试时做过的工作，我们可以推测，关于v6对应的内容的地址处所存的flag在调用函数之前就已经发生了修改，接下来我们直接观察被揭秘的jolin函数，便可以发现其和v6的联系！！！下面附上jolin函数的反汇编c代码：

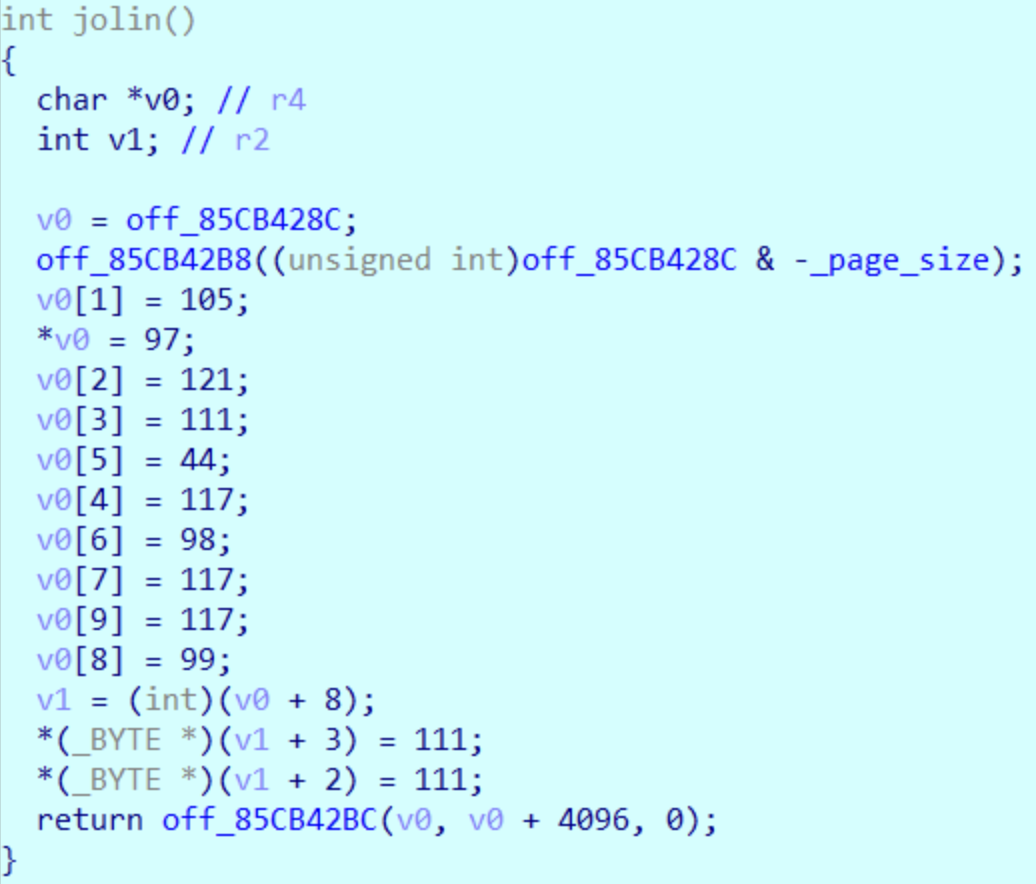


图 21jolin函数本体

该函数的功能相对于之前的静态分析需要猜测和推断而言，就显得很小儿科了，v0作为一个char形指针指向了一块地址，然后依次对这块地址上的内容进行了赋值，可以直接使用IDAPRO自带的功能，把数字选择切换到CHAR类型，并且按照0-8重新排序，可以看到v0处的字符串为“aiyou，bucuoo”

函数最后return了一个地址值 off\_85CB42BC。然后查找该地址的交叉引用，还真就和v6扯上了关系，在MainActivity函数中找到了该地址值，该地址被赋给了v6，其实就是v6的地址值在程序执行过程中被改变了。

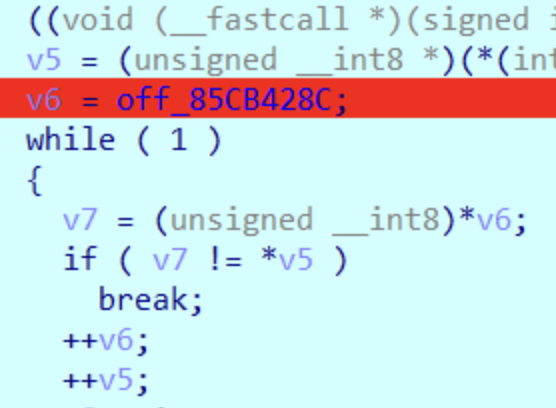


图 22v6被修改

接下来对新获得的flag进行测试，打开模拟器，输入“aiyou，bucuoo”，可以看到成功破解！动态调试就此完成！！！！！

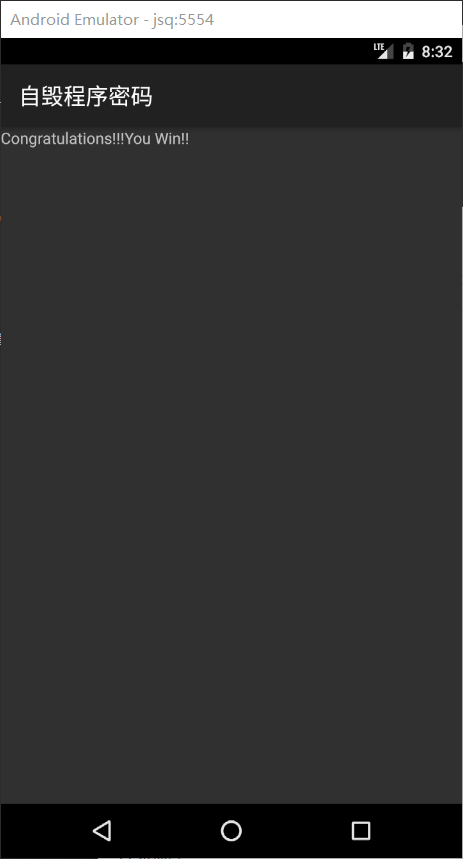
****

图 23破解成功！哎哟不错哦！

1. **实验结论**

具体如何获取该程序的flag以及分析的过程都在4.分析方法和过程中写的很详细了，遇到了什么问题和解决的方法都在里面详细阐述过，此处不再赘述。想谈一谈这个程序本身的一些内容，以及她是如何反调试的，毕竟一个实验想要有所收获，不能仅仅停留在完成任务上，要自己发掘，学到东西才算是一门值得的课程。回顾.so文件的加载过程步骤：.init，此为虚拟机自带程序；.init\_array，虚拟机调用此初始函数；JNI\_Onload，安卓中的linker程序调用此函数；Java\_com\_yaotong\_crackme\_MainActivity\_securityCheck函数，一般反调试更高概率在JNI\_Onload段，先查看init\_array函数。

该函数return了一个sub\_22AC函数，该函数里面又调用了一个参数函数sub\_1CA8。sub\_22AC函数里面主要做了一些分配空间的工作，分析的意义并不大，1CA8函数代码如下

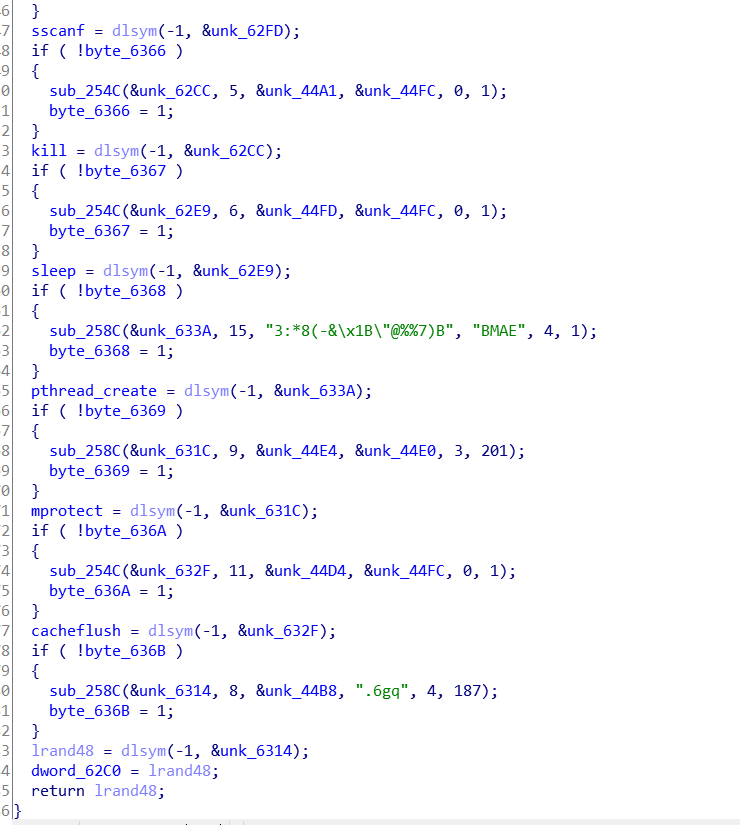
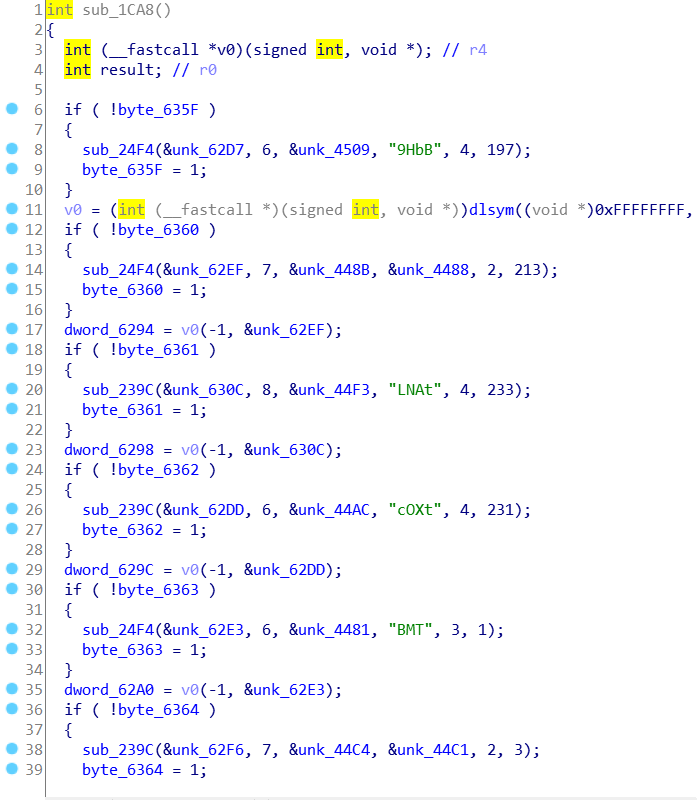


图 241CA8

整个函数运用了大量的if判断，如果符合某条件就进行某操作，然后把判断条件翻转（0变1），然后对某函数进行某操作。

再仔细分析代码。发现其中v0用到了dlsym函数，dlsym函数是根据动态链接库操作句柄与符号，返回符号对应的地址。

void\* dlsym(void\* handle,const char\* symbol)

handle是由dlopen打开动态链接库后返回的指针，symbol就是要求获取的函数的名称，函数返回值是void\*,指向函数的地址，供调用使用。

因为每次判断完之后都调用v0对某个地址进行赋值，这里推测是将所有用到的函数都用函数指针保存，隐藏函数本身，加大反调试的难度。

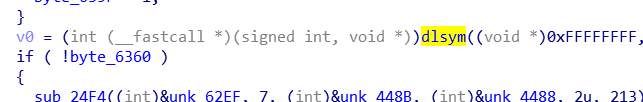


图 25dlsym函数被调用

#：关于dlsym函数，我在CSDN上查阅到了这样的内容：dlsym函数第一个参数handle可以带dlopen库的返回值，也可以带0xFFFFFFFF，带0xFFFFFFFF表示从当前内存加载的所有so中查找某个symbol。

分析到这里，现在的主要任务是破译出主要的函数调用，这样才能分析出程序逻辑。因为看到每个if块里调用的函数就是sub\_24F4、sub\_239C等几个，且重复调用。每个if块后的赋值都是用v0调用了if块里的某个地址。所以可以大胆猜测这段逻辑为if块里对函数名字进行解密，然后再将该函数用函数指针保存。

分析了程序之后我们知道整个函数内各个函数指针的获取都调用了v0这个函数,那我们隐约感觉到v0可能就是dlsym的函数指针，验证一下,直接把解密函数的伪代码粘贴到Xcode内，微调数据类型，这样一个简单的解密函数就诞生了。

输入函数unk\_62D7，即可获得解密出来的名字如下：

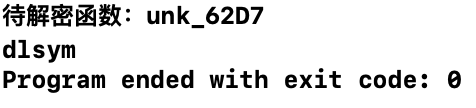


图 26解密unk\_62D7

果不其然，所以v0就是dlsym函数的指针。

利用该函数解密所有的变量名，得到一张对应的关系表，具体关系如下表所示：

|  |  |
| --- | --- |
| 变量名 | 函数名称 |
| 62D7 | dlsym |
| 62EF | getpid |
| 630C | sprintf |
| 62DD | fopen |
| 62E3 | fgets |
| 62F6 | strstr |
| 62FD | sscanf |
| 62CC | kill |
| 62E9 | sleep |
| 633A | pthread\_create |
| 631C | mprotect |
| 632F | cacheflush |
| 6314 | lrand48 |

当我们看到pthread\_create，getpid和kill这样的函数时就可以自然的把反调试与其联系起来了。

我的推测是利用getpid这样的函数来检测是否进入了调试的环境，然后就会用kill进行反调试的防御，也就是我们在模拟器中看到的闪退行为。

选中pthread\_create，然后查看这个函数引用位置。

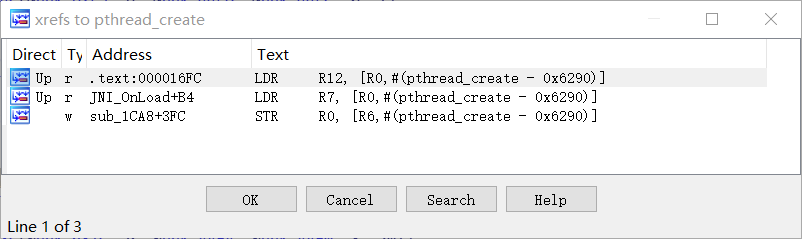


图 27交叉引用

进入查看函数。

因为pthread\_Create地址为dword\_62B4,里面调用了一个函数是sub\_16A4。然后进入sub\_16A4这个函数查看，死循环中套了一个sub\_130C。猜测反调试在sub\_130C函数里。

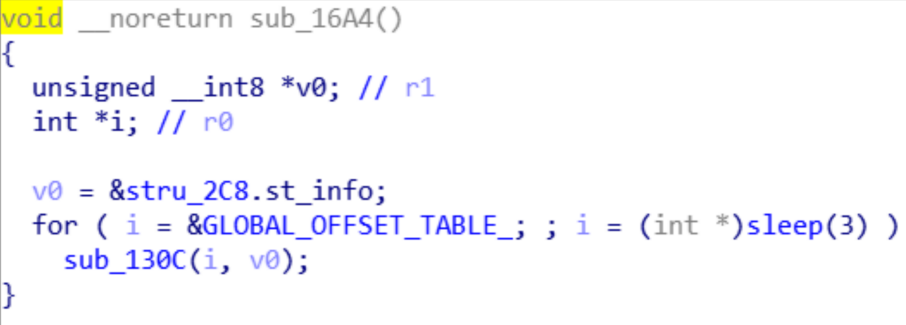


图 28sub\_16A4

进入sub\_130C查看。因为该函数中有几个比较熟悉的地址值，先进行一轮替换。该函数也是一个死循环，是否跳出以v12进行判断。

然后跳出循环后发现最后一行还出现了v3，就是kill函数。v0是getpid获得的进程码。在TracerPid>=1的时候调用的kill(发信号)函数，向对应PID发送9(SIGKILL)信号，以干掉进程，实现反调试。



图 29反调试核心代码

以上就完成了关于反调试内容的分析。

整个分析过程是实验话费时间最长的，查阅了大量的资料。也非常感谢网上的一些博客，提供了详细的解答。

下面进行去除反调试。只要程序不会break跳出循环，程序就不会被kill。只需要对v12的判断条件修改即可。

结合第一次实验进行的栈平衡修改，我们同样采用修改机器码的方式。

进行汇编代码修改，将1改为0，BGE改为BLT即可。



图 30修改汇编代码

使用UltraEdit打开.so文件，根据偏移地址查找到相应指令的机器码。将对应字节上的“01”改为“00”，“AA”改为“BA”。

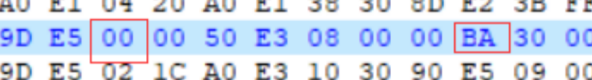


图 31机器码修改

修改完后，用android killer重新编译，形成新的apk。

然后再进行动态调试时就不用被反调试闪退了，整个实验也就算是圆满完成了。

1. **实验心得**

整个实验的过程还是非常艰辛的，花费了大量的时间和经历去查阅资料，其实实验很早就做完了，但是报告一直拖了很久，一直在构思怎么体现自己完成了更加深入的内容。

也想说点自己用心做实验的原因，一方面是对逆向工程还有些兴趣，可能是出于第一次实验被强大的IDAPRO震撼了，另一方面是因为自己最近一直在找工作，参加各种大厂的实习面试，被面试官提到最多的就是“什么都知道一点，但是知识的深度不够”。这点让我不禁开始反思是不是很多实验课自己做的不够深入，所以这次对于只获取答案而言，又做了一些其他的工作。

谈及到实验本身，获取到flag并不难，因为在动态调试的时候jolin函数被自动解密了，而且jolin函数本身是非常显而易懂的代码，并没有什么难度，跟着老师的方法走，就能获取，而且网上还有很多其他工具的教程，可以参考的内容非常多。

花费我最多的时间的不仅是关于反调试的分析，还有实验中踩到的各种坑。

一方面是夜神模拟器的问题，我在上述实验的分析中已经提到过了，夜神模拟器是老师上课用的，所以我就直接用了，但是后续发现了各种问题，比如架构不同，在attach后所有的地址都是？？？？？？的字符，根本无法看到内容，还有夜神模拟器自带的adb以及独家的nox\_adb都会出现乱码问题，并且是常规cmd中乱码问题的解决方法无法解决的，比如使用chcp65001命令，甚至在cmd的属性中修改字体都无法解决乱码的问题，后续就更换了AVD Manager以及SDK Manager，以上的问题均被解决，所以我还是觉得一个为了游戏开发的模拟器和实际为了软件开发的模拟器还是有很多不同的，在此衷心的建议鲁老师明年教这门课也能不用夜神模拟器，我觉得会更好一点。

二方面是jdb的问题，怎么去下载，配置环境，解决的方法网上有很多，我就不具体说了。

三方面是java和c之间交互的接口，这方面我查阅了很多资料，网上也都有，主要是围绕函数和指针来的，以及一些常规的调用链接库的写法，我也不多赘述。

四是关于小米手机助手的，我一开始一直链接不上去，好像是端口占用，因为我本人是个小米手机的粉丝，所以经常用这个软件用来备份恢复，甚至是刷机，但是她占用了端口会导致动态调试的问题，关闭该软件后就一切正常了。

五是关于反调试，解密函数名字是最困难的部分，但是好在平时自己是C选手，直接复制粘贴到编译器然后改一改就能破解了，如果没有这一步，估计 分析到猴年马月都分析不出来反调试的内容了。这可能也是逆向的魅力吧，在老师的“不求甚解”中又要在合适的时候进行“详细挖掘”，这样才有可能理解这个程序设计者耗费的心思。

很感谢老师提供的帮助，在实验时能快速获取flag，这是一门有趣的实验，也是我觉得审视自己学习方式的开始，不想再被面试官评价知识没有深度了，完成这次实验也给了我很大的信心，在Google上能找到我需要的所有知识，整个程序我自认为分析的比较“透彻”，透彻指的是在该透的地方透彻，在没必要的地方还是“不求甚解”。