

**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： U201810398**

**姓 名： 吴锦添**

**指导教师： 鲁宏伟**

**报告日期： 2021/3/19**

**网络空间安全学院**

逆向分析实验2

[1. 实验内容 1](#_Toc69775958)

[2. 实验准备 1](#_Toc69775959)

[3. 静态分析 1](#_Toc69775960)

[3.1. 逆向Java代码 1](#_Toc69775961)

[3.2. 逆向so程序 3](#_Toc69775962)

[4. 使用动态调试 9](#_Toc69775963)

[5. 实验总结 13](#_Toc69775964)

# 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。

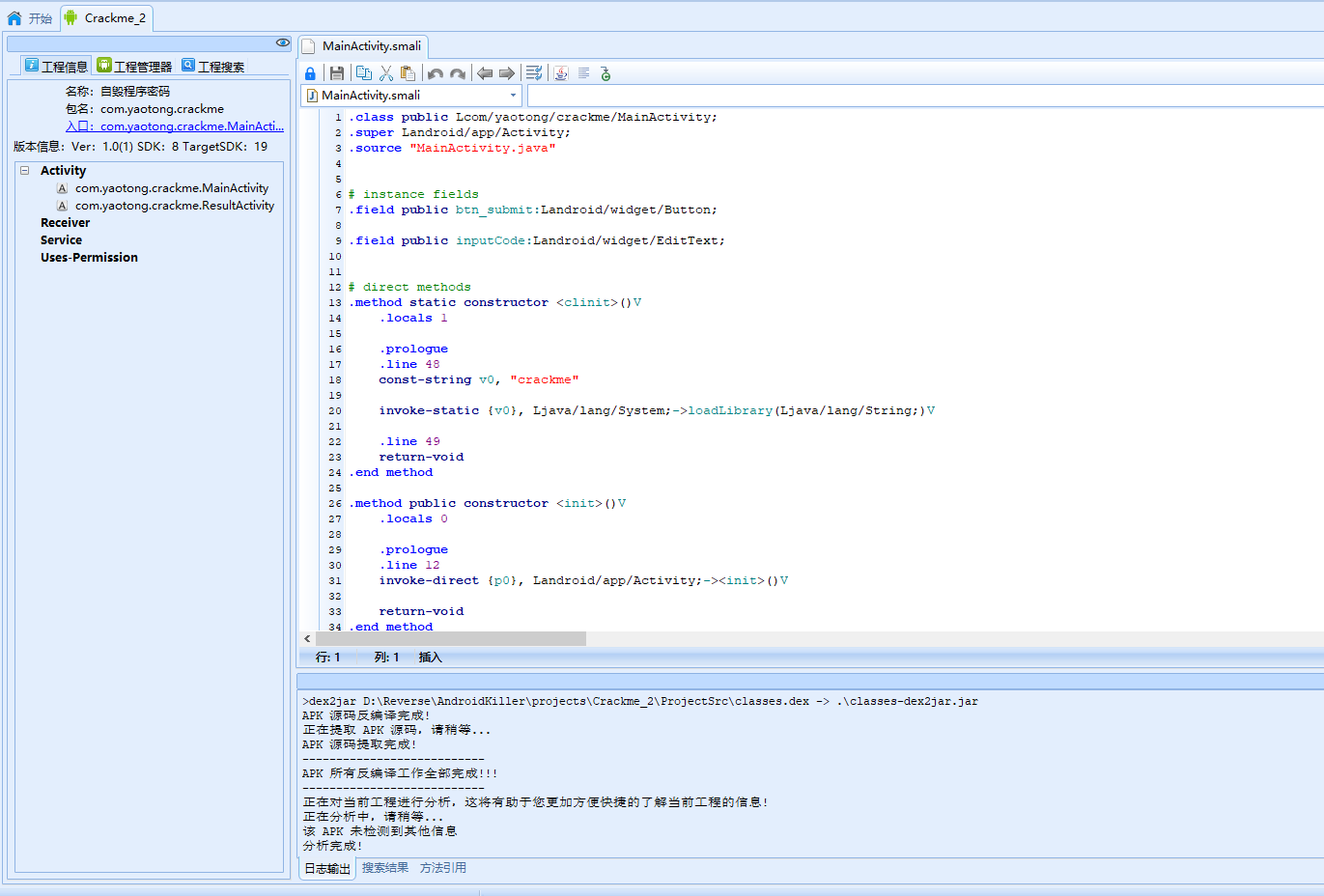
# 实验准备

1. Android手机；
2. AndroidKiller软件；
3. IDA Pro 7.0软件。

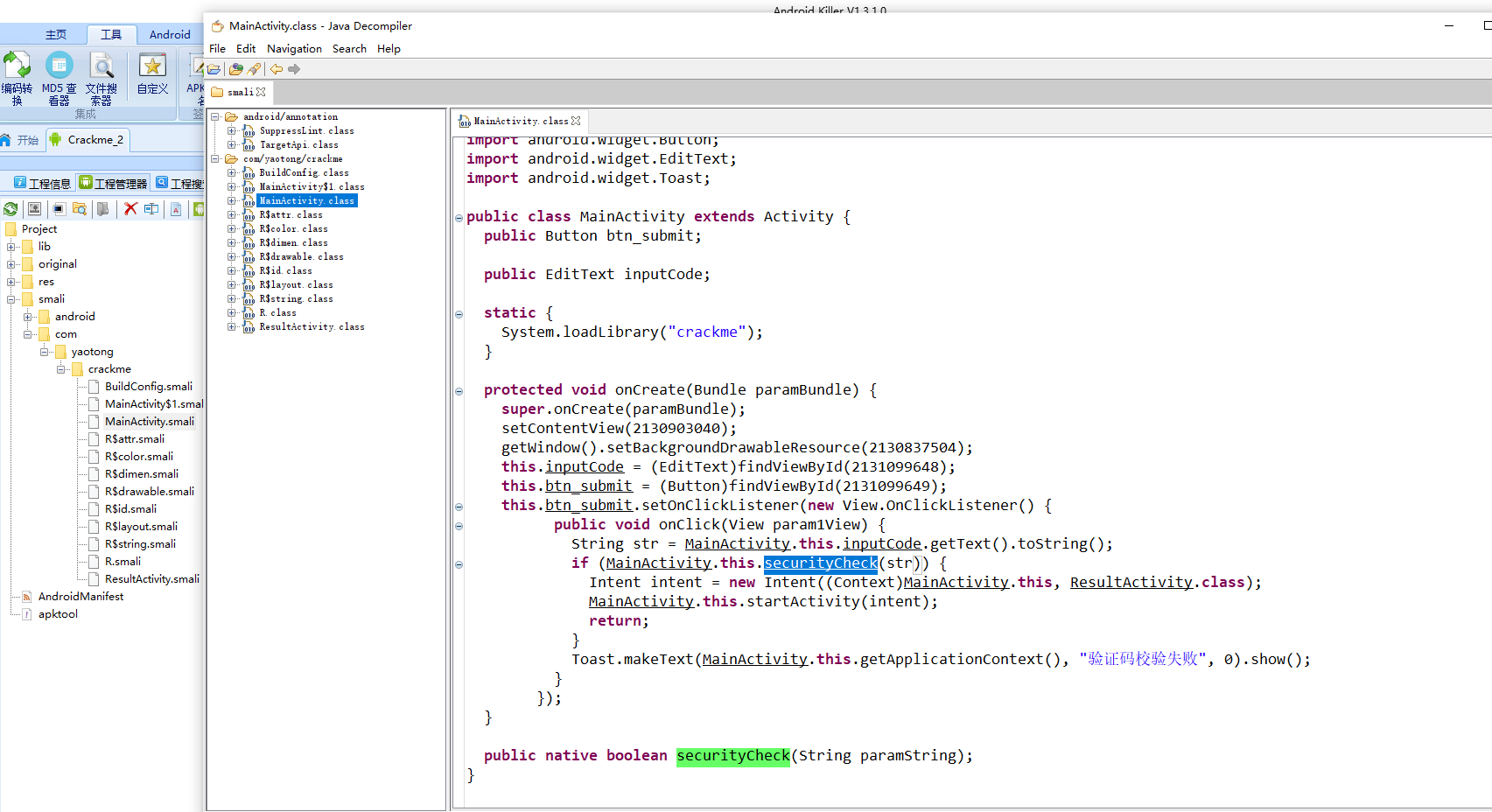
# 静态分析

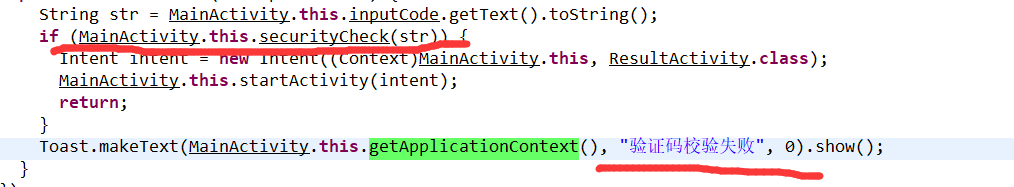
## 逆向Java代码

使用Android killer打开Crackme\_2:

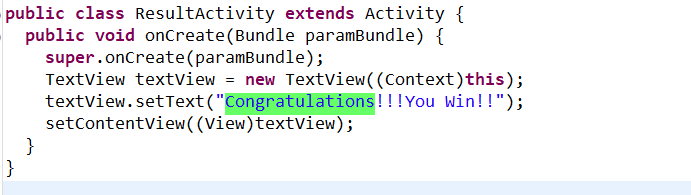


使用反编译工具Java Decompiler进行反编译：

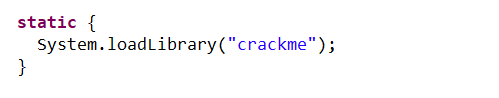




根据反编译的JAVA代码可以猜测，如果返回为0那么则输出“验证码校验失败”。那么当输入的字符串str在MainActivity.this.securityCheck(str)中返回得到1的时候能够避免手机自毁，而当返回值为1的时候执行的是ResultActivity.class文件中的函数，点开该函数查看其内容发现有字符串“Congratulations!!!You Win!!”输出：



在smali文件中找不到securitycheck()函数，而后发现在文件中加载了一个库文件：



于是在lib文件目录下查找文件，在发现了libcrackme.so文件：



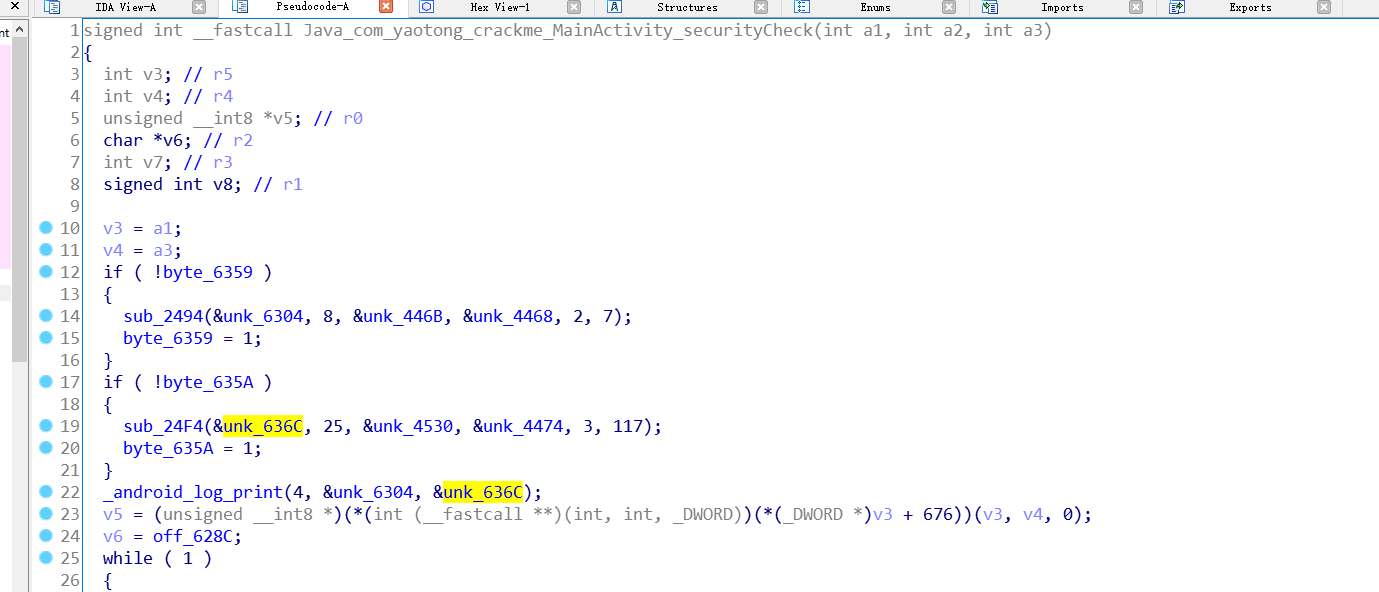
使用Android killer直接打开该文件显示的是乱码，所以使用IDA pro打开并分析该文件。对so文件的分析放在下一节中。



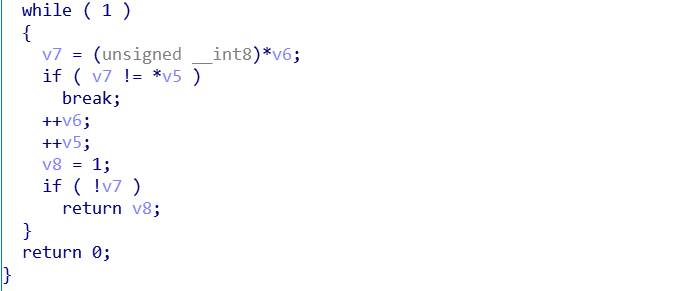
## 逆向so程序

1. **借助于IDA工具，逆向实例程序，根据Android程序执行过程，定位核心代码；**

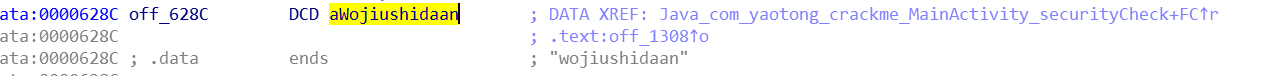
使用IDA PRO打开该文件后，查找到对应的securitycheck函数，按F5进行反汇编，得到以下：



分析该函数发现后续进行了两个字符串的对比，将v5和v6进行对比，如果两者不同那么就返回0，这也就是输入验证码错误的情况；当v5和v6一致的时候，返回v8，v8就是1 。而且发现v6是固定的一个字符串，那么猜测v6是正确的验证码，v5是输入的验证字符串。



那么点开v6查看v6的值是什么，发现v6是aWojiushidaan，而使用这个字符串进行验证的时候发现，仍然错误。





根据这个错误提示猜测程序一定是在什么地方修个了这个字符串。但代码中找不到明显的与这个变量相关的代码，如果能够进行动态调试，直接定位到比较这个变量的地方，就会知道它最终的验证码了。在Android程序运行的时候，首先是“init\_array”，Android系统在加载App时，通过系统的linker程序先加载这个函数，对App进行初始化，然后再调用“JNI\_OnLoad”。

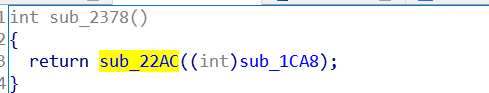
1. **分析程序在初始化过程中，如何完成对关键信息的混淆，根据分析的结果，还原这些关键信息；**

在Android程序运行的时候，首先是“init\_array”，Android系统在加载App时，通过系统的linker程序先加载这个函数，对App进行初始化。那么开始对init\_arry函数进行查看，看在其中是否进行了对字符串进行了修改。

按住shift+F7查看程序段的函数：



在program segment中查找到init\_array，并且双击查看其内容，按下F5查看伪C代码。仔细查看了这些代码，发现在init\_array中调用了一些函数对关键的代码信息进行混淆：



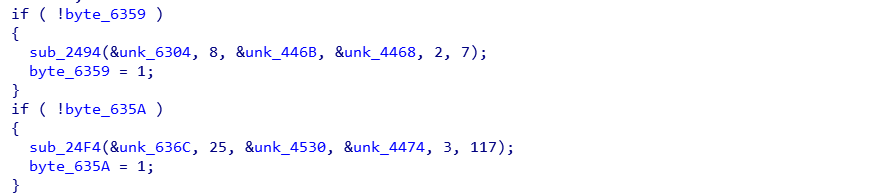


以上的一些操作就是对函数名进行加密，通过这样来混淆关键代码的信息。在JNI\_OnLoad当中还有jolin来对off\_628C字符串进行修改，同时对可执行的jolin进行加密，在执行过程中进行还原使用。这也是crackme进行混淆的方法之一。

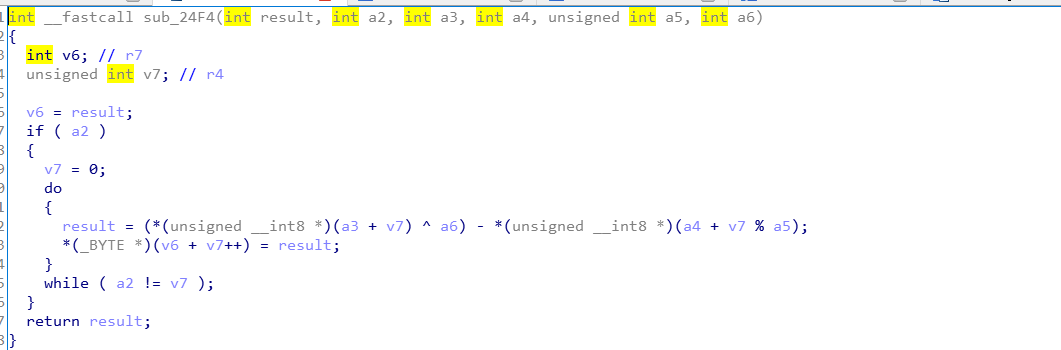
1. **分析程序中反调试的方法；**

根据后续得知的信息：在Jolin这个函数中对flag进行了修改，所以可以肯定的是反调试程序一定在Jolin函数执行之前就已经再运行了，所以需要分析的代码段就是在Jolin函数之前执行的代码段。所以可知反调试代码一定处于init\_array或JNI\_OnLoad代码段中。

再来重新看看MainActivity中的这段代码，根据if语句的条件不难看出，这段代码的效果是：如果byte\_6359和byte\_635A这两个变量为0的话则将其置为1，并且与此同时运行了sub\_2494和sub\_24F4两个函数。



进入sub\_24F4函数中，可以发现这个函数进行的是一个字符串加密操作，由模运算和异或运算组成。



使用交叉引用之后，进入其中一个函数进行查看：



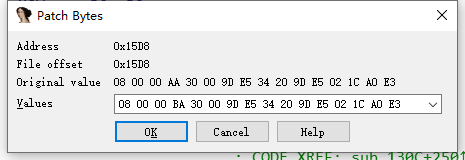
在这个函数中我们发现了反调试程序，v9是一个函数指针，并且是一个四个参数的函数，当v12>=1时就跳出这个循环，而循环节外的这个函数经过解密之后正是kill进程自身的函数。经过解密后发现v9正是pthread\_create函数，创建一个监视线程，一旦程序被调试就直接关闭自身。

然后通过以上函数检查当前进程是否被调试, 检测方法: 读取 /proc/<pid>/status 的 TracePid 字段, 如果程序没有被调试, 该字段值为0, 否则就是调试器的进程PID。如果该值不为0, 则给当前进程发送信号 SIGKILL，结束进程。

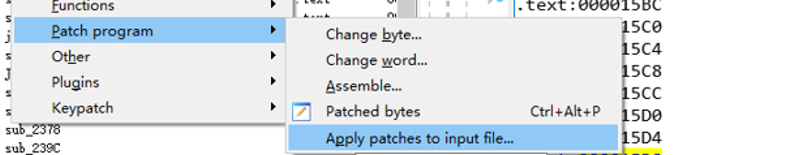
1. **借助于指令修改工具，去除反调试代码；**

从上述的反调试分析其实就可以知道如何去除反调试的代码功能了，只需要修改so文件, 将 SIGKILL 信号修改为 0 信号(0信号没有任何副作用, 只用来检测进程是否存在), 将偏移为 0x15D8 的 BGE loc\_1600 (机器码 为 08 00 00 AA) 修改为 BLT loc\_1600(机器码为 08 00 00 BA )。

使用IDA打开.so文件，而后查找到对应的地址开始修改：

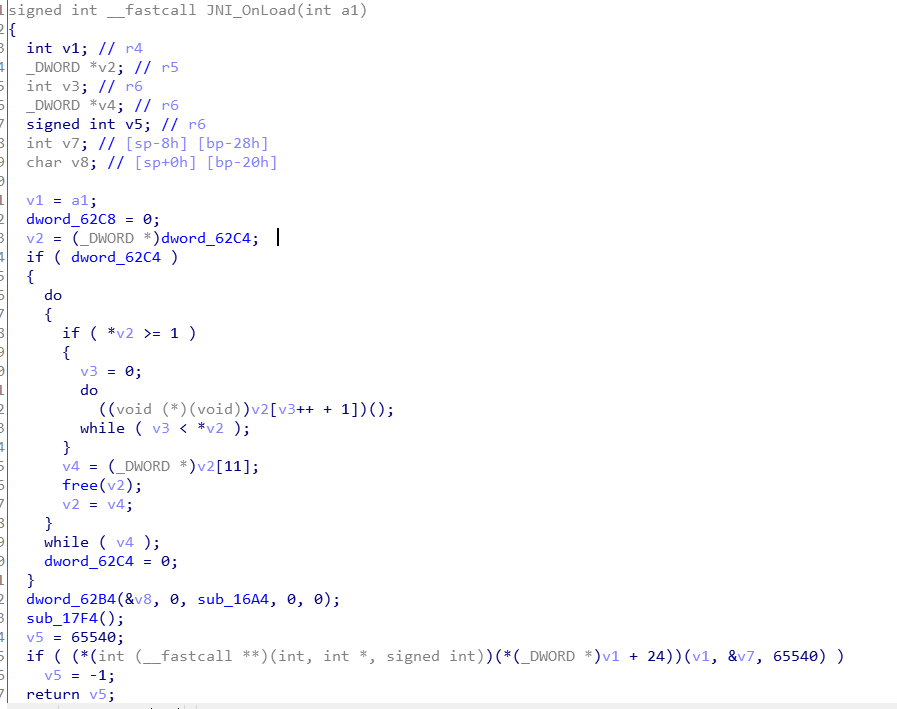


而后使用edit->Patch program->Apply patches to input file，将该文件保存：

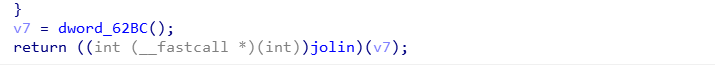


1. **通过静态分析方法，对核心代码进行还原。**

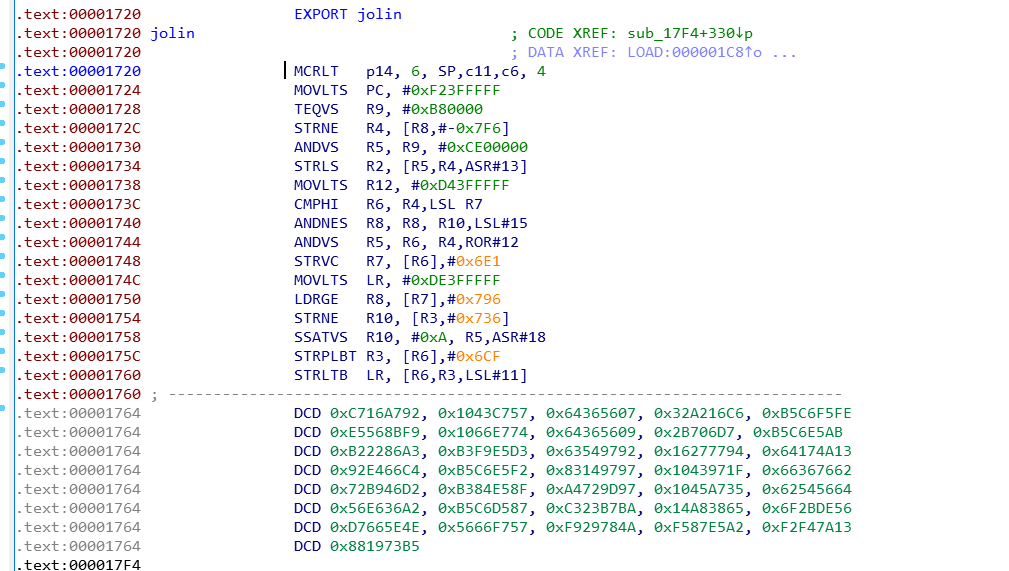
查找到JNI\_OnLoad函数开始查看相应的代码，而后不断地查找其中调用的一些函数，查看并分析其中内容。



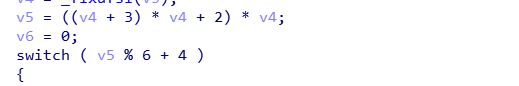
在sub\_17F4中发现了一个jolin文本段，这个文本段的内容就像是乱码，而在sub\_17F4后的代码中，却返回了jolin并且以(\_fastcall\*)的形式返回，这个jolin显然是一个函数，是可执行的。



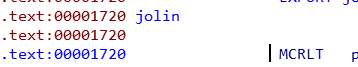
但是jolin在文本段中看起来就是一行行的乱码，无用数据段：

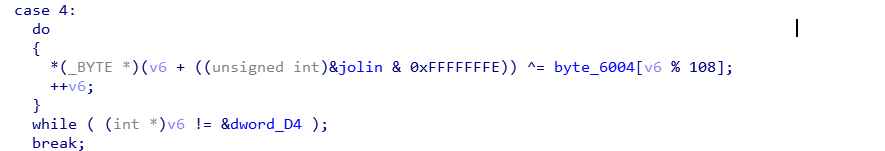


而后分析sub\_17F4中的代码，在函数中定义了v5为v4^3+3\*v4^2+2\*v4， v5=(v4+1)\*(v4+2)\*(v4)，v5就是三个连续数之积显然就是6的倍数。所以在后续的switch语句中进行的选择判断 v5%6 +4 的值一定是4。

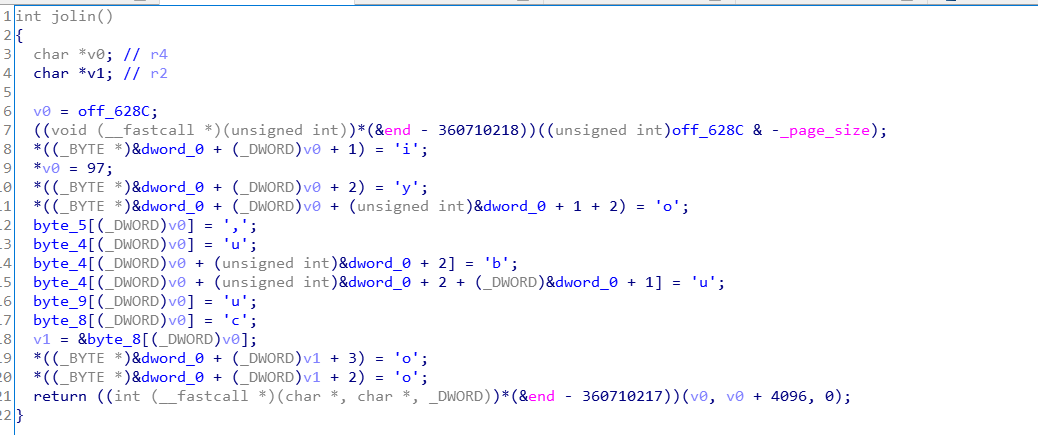


选择语句进入case 4，因为jolin的地址为1720，那么&jolin & 0xFFFFFFFFE得到的结果还是jolin的地址，而case 4的作用就是对从1720开始的D4个字节进行异或操作。

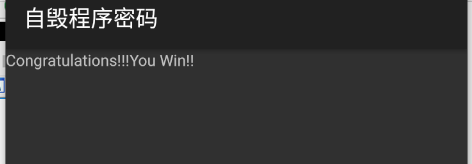




根据其中的异或规则以及byte\_6004处的字节数组信息，编写一个破译脚本，将jolin的信息进行修改，并且重新编译成新的libcrackme.so文件，再次使用IDA打开新的libcrackme.so可以查看当crackme执行完JNI\_OnLoad后的jolin代码如下：



可以看到在jolin中将off\_628C处的字符串改成了”aiyou,bucuoo”，那么猜测这个字符串就是最终的验证码。而后使用这个字符串在自爆程序中进行验证，果然得到了“congratulations“：



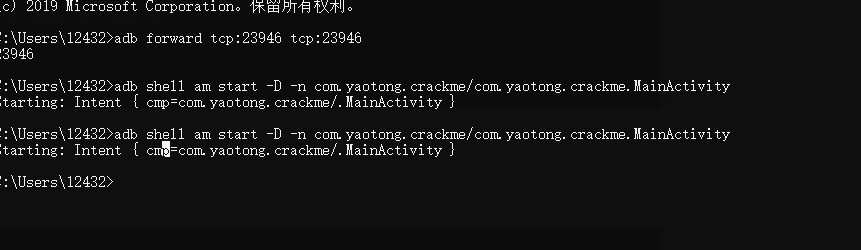
# 使用动态调试

使用模拟器模拟crackme的运行环境，连接夜神模拟器：adb connect 127.0.0.1:62001

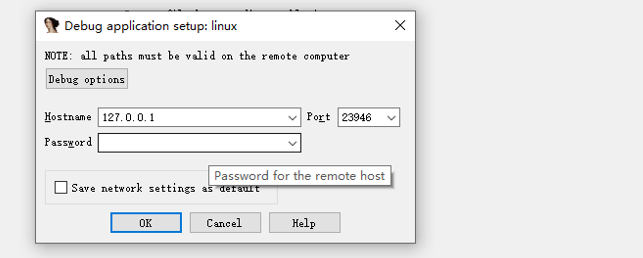
将文件push 进手机的指定目录下，进入手机端命令：adb shell，切换获取手机的root权限：su，查找push的文件是否在手机中：ls -l，拥有root权限更改文件的权限为777：chmod 777 android\_server。具体指令执行如下：



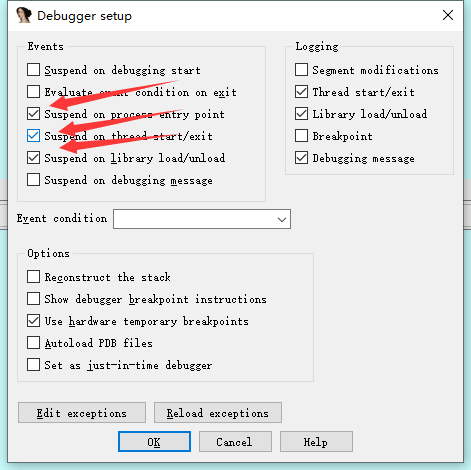
将模拟器运行端口转发到PC：adb forward tcp:23946 tcp:23946，为了反反调试：程序的so文件在加载阶段会先执行JNI\_OnLoad，之后就不再执行，在程序的so文件加载阶段才能给JNI\_OnLoad打断点调试。于是使用adb shell am start -D -n com.yaotong.crackme/.MainActivity指令，使调试APP挂在加载界面。



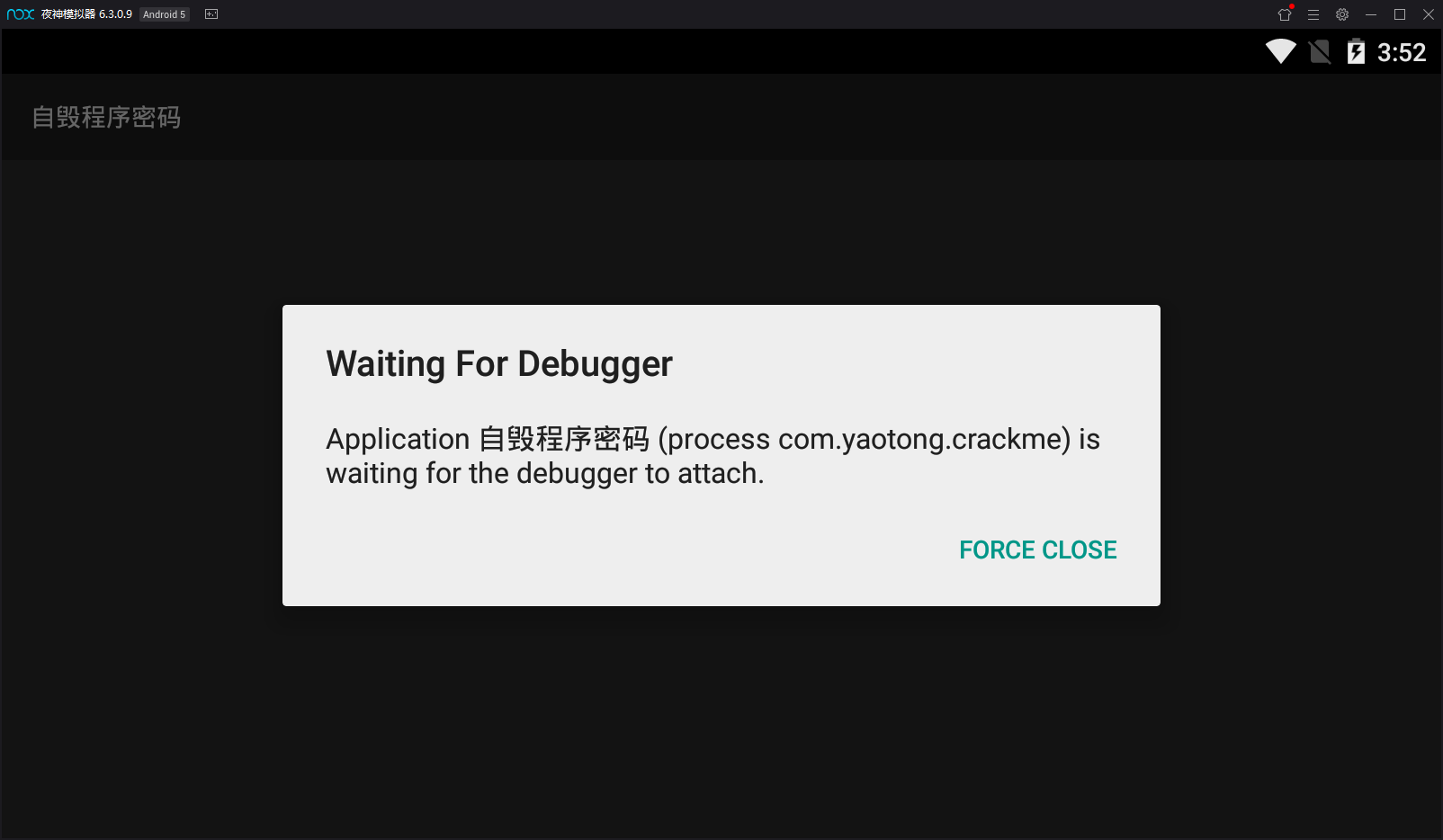
打开IDA使用动态调试功能监听23946这个端口，而后附加相应的com.yaotong.crackme进程：



在进入调试界面后设置相应的debugger options，勾选上在入口处暂停，在线程开始处、退出处暂停，在库加载、卸载时暂停这三个选项：



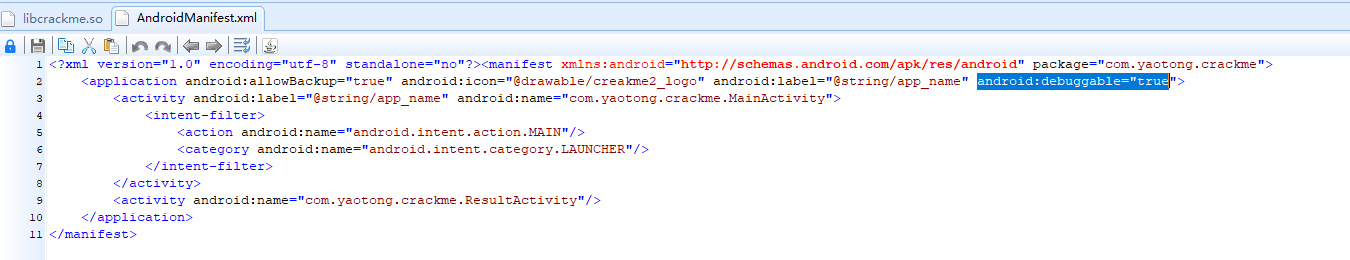
而后点击运行键(或者F9)发现只能运行以下就卡住了，这是因为在模拟器中运行的进程没有继续执行所以不能进行后续的调试，这时就需要使用ps | grep com.yaotong.crackme先获得APP的进程ID，而后再使用adb forward tcp:8700 jdwp:\*\*\*\* 监听进程的ID，最后执行db -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700 来运行进程。



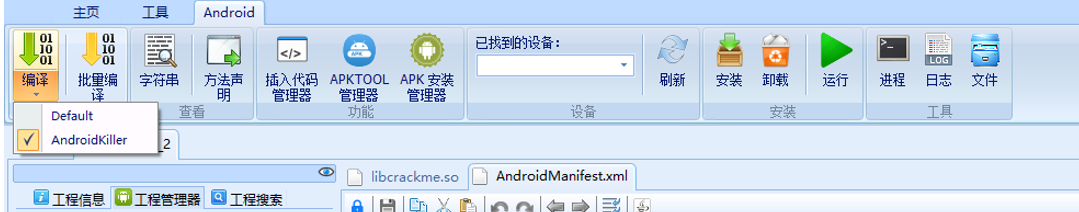
但在执行指令的过程中发现Jdb connect执行进程无法成功，甚至一开始jdb都不能使用，在修改好jdb的环境变量后，jdb终于可以使用了，但是还是不能够执行该指令让自毁程序继续执行下去。



在查阅资料后得知需要在AndroidManifest.xml文件中添加一条指令，在<application 里给APP加上可调试权限，android:debuggable="true"，重新打包APP，签名，安装。



重新进行apk编译的时候，编译功能的默认选项是Default，而需要生成整个可执行的apk需要勾选AndroidKiller项，否则会生成只有1Kb的apk文件。（在实验过程中遇到这个问题还查了好久）。

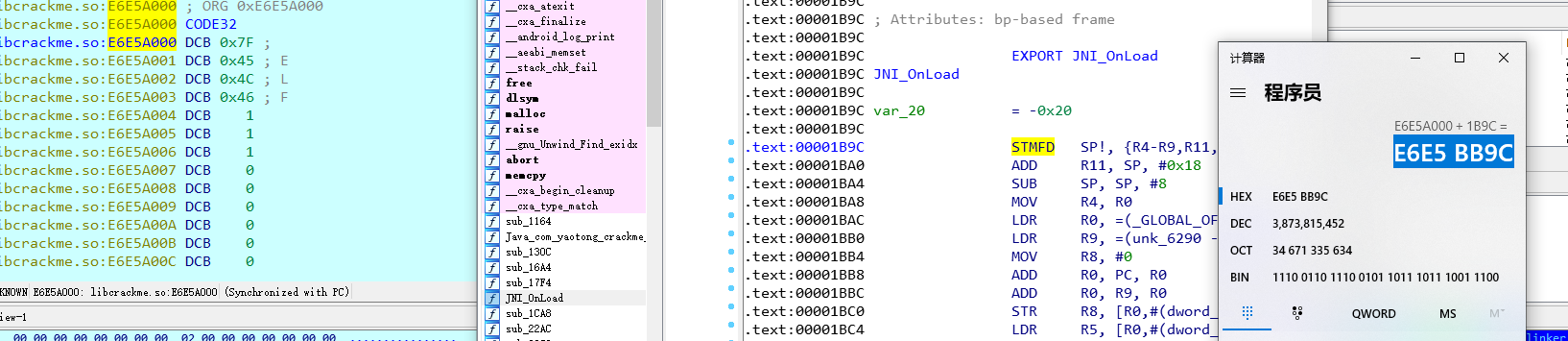




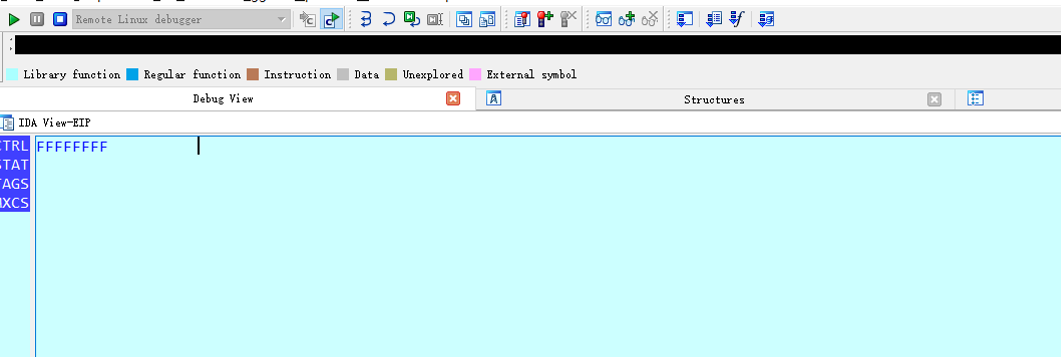
而后再次重复上述步骤即可让自毁程序继续执行下去：



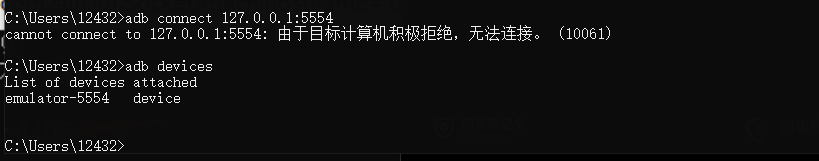
随后一步步的运行，然后查询是否载入libcrackme.so文件，载入后计算JNI\_OnLoad的绝对地址，绝对地址=执行so文件的基地址+函数的偏移量。那么计算出JNI\_OnLoad的绝对地址为：

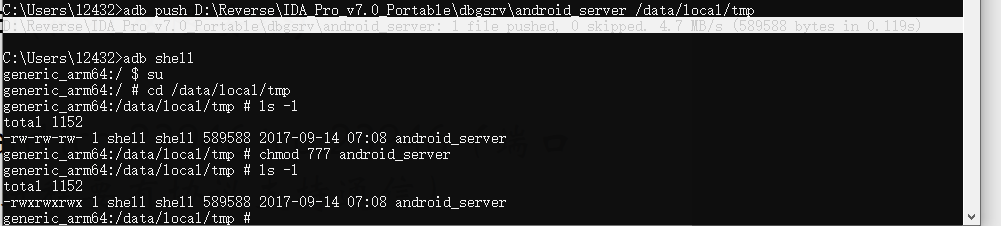


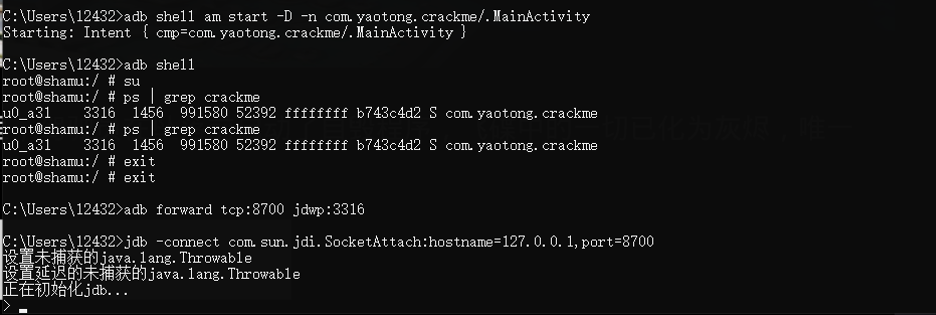
但是用虚拟机接着继续调试就会直接运行失败，结果如下：



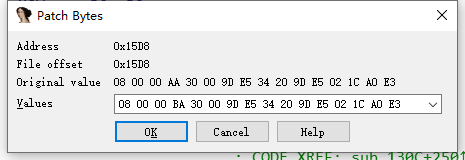
即使在尝试MuMu模拟器也不能正确执行后，选择了下载Android Emulator来模拟真机进行动态调试。动态调试的基本流程与上述一致，以下给出执行过程截图以及部分重要部分解释。

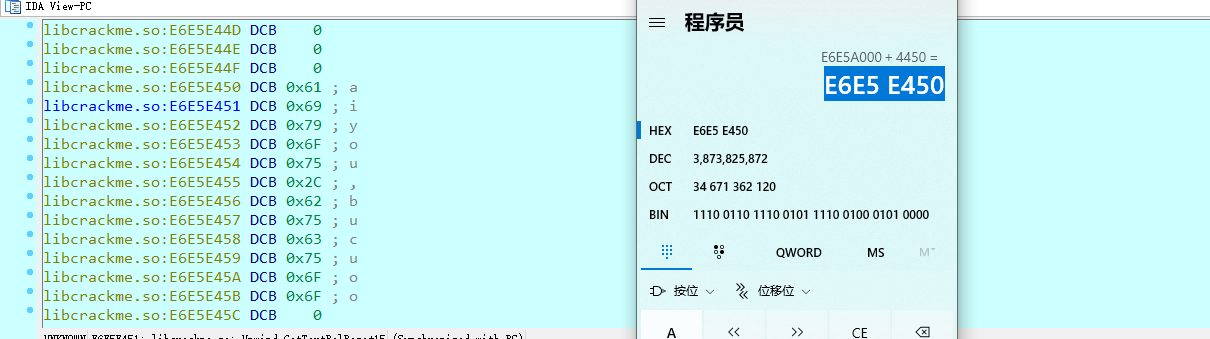




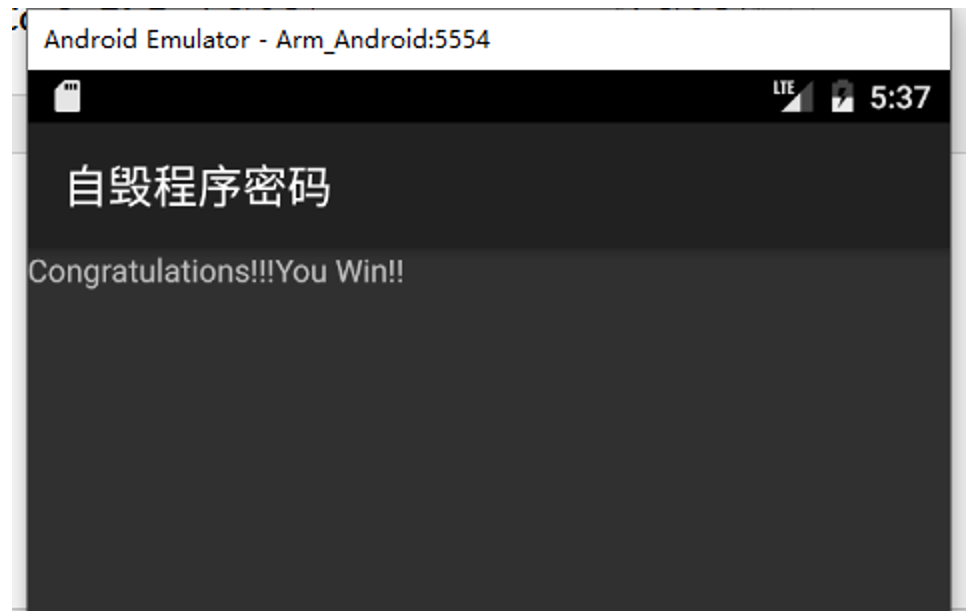


为了完成动态调试还需要反反调试，在第三节中已经详细的讲述了反反调试的原理以及操作，这里直接给出动态调试最终的结果，可以直接计算出off\_628C的绝对地址，然后使用G E6E5E450直接执行到这个绝对地址，查看他未被修改前的值：





最后在输入对应的字符串，得到了“congratulations“验证正确的提示：



# 实验总结

相对于第一次的实验，这次的实验相对来说更难，而且我也确实做了很久，一直在动态调试的地方卡住，换了几个模拟器来尝试都不太行，而且每次失败只得一遍又一遍的重新输入指令。

本次实验体会到了在很多时候静态分析是不够的或者是很复杂很繁琐的，许多程序中的数据在运行起来之后会发生各种各样的变化，而且许多部分都是被加密的，一味地静态分析费时费力，特别是对于加密函数较复杂的程序而言，最好的方法还是在程序运行中下断点调试。

本次实验对于具有反调试功能的程序有了了解，也知道了如何去解决这个问题来实现对他们的调试：

静态调试：

如果静态分析发现程序内加密较简单或者未加密的话，可以直接修改代码达到绕过反调试线程的效果，之后重新打包apk文件，然后再次调试就能比较轻松的得到flag。

动态调试：

如果发现源程序加密十分复杂，可以使用动态的方式进行调试，首先根据核心函数判断flag被修改的时间或者位置，而后在相对应的地方设置断点进行调试。

但其实发现对于一个程序的逆向，静态和动态都是不可缺少的，静态的分析有助于动态分析断点设置的位置，动态分析又能减少静态分析的工作量。

感觉本次的实验虽然比较难，但在不断地犯错查阅资料，仔细检查中也学到了挺多，感觉逆向真的超有意思！