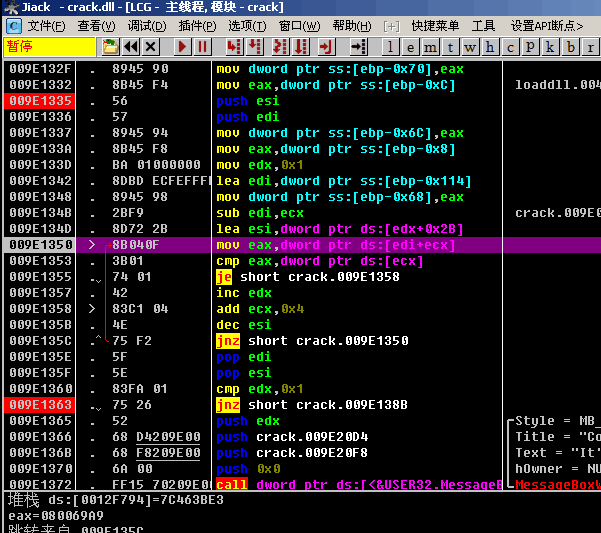
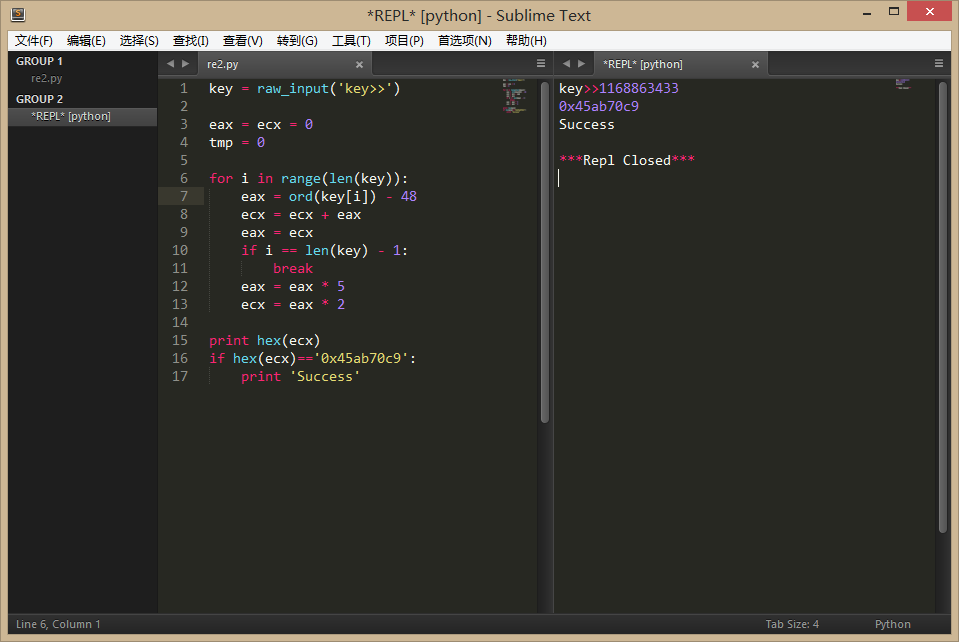
## 0x01 RE01 (200pt) heeeeeeeelp



题目给出了一个DLL，直接丢进OD，发现程序将传入参数和内存中0x44\*4字节的一段数据进行了比较，抠出数据发现有zip特征，补全文件头解压得到flag。

## 0x02 RE02 (250pt) 逆向新手

这题比较坑，有反调试，不过掉的话后面代码是有问题的，nop掉相关语句，可以得到这样的验证算法：

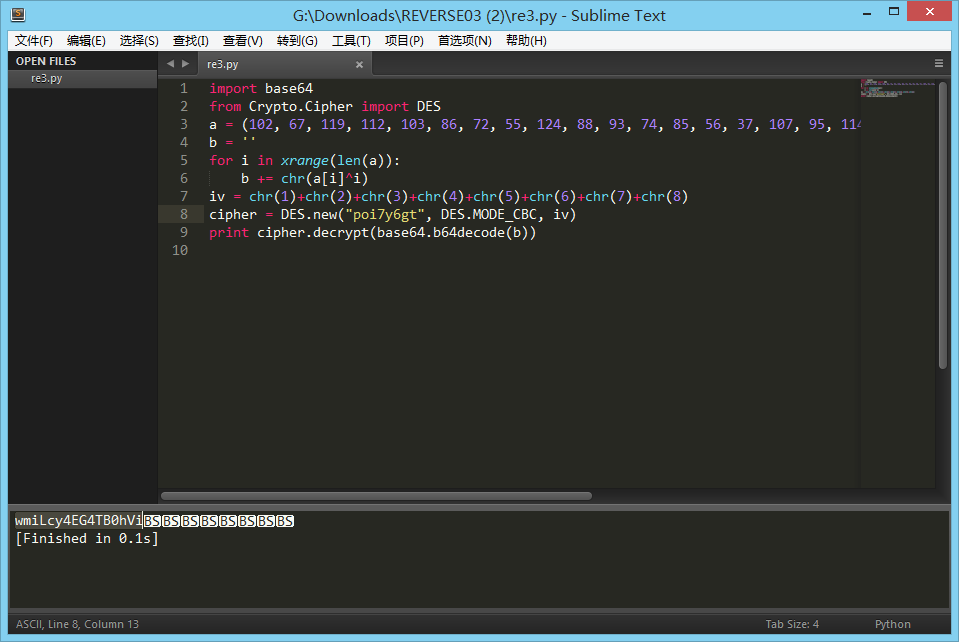


简单分析下算法，可以看出一个很明显的flag: 1168863433

## 0x03 RE03(300pt) 没有题目的题目

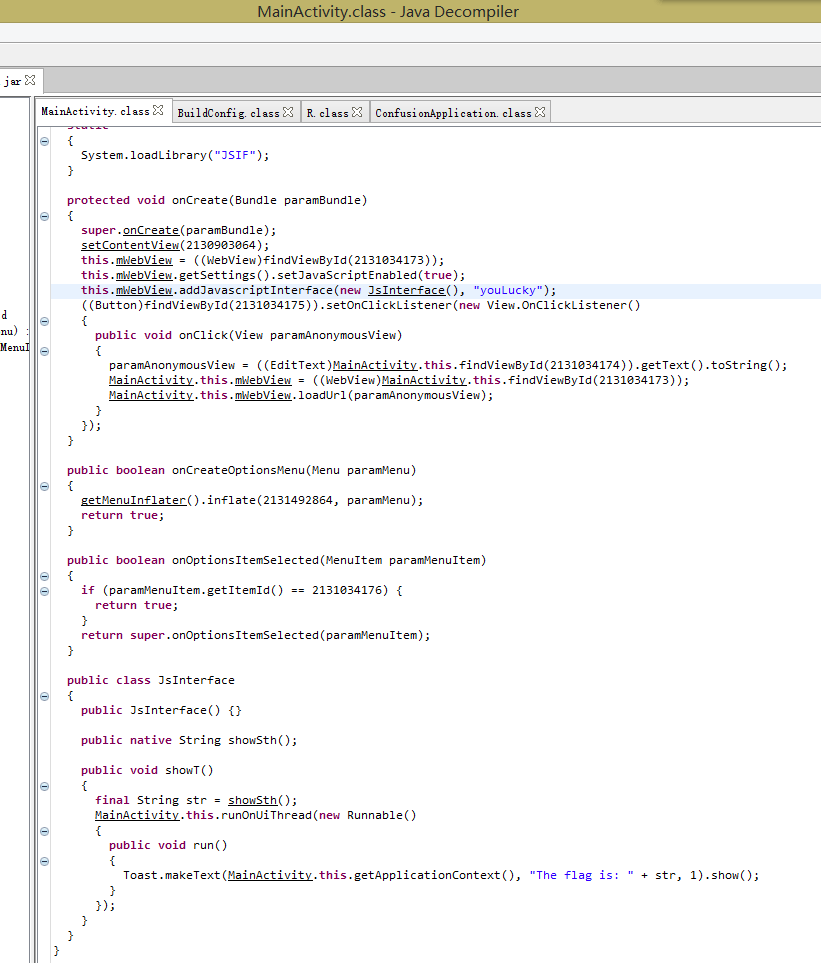
题目给出了一个损坏的APK，直接用winrar自带的修复功能修复后丢进APKIDE

算法很简单，就是一个CBC MODE的DES解密

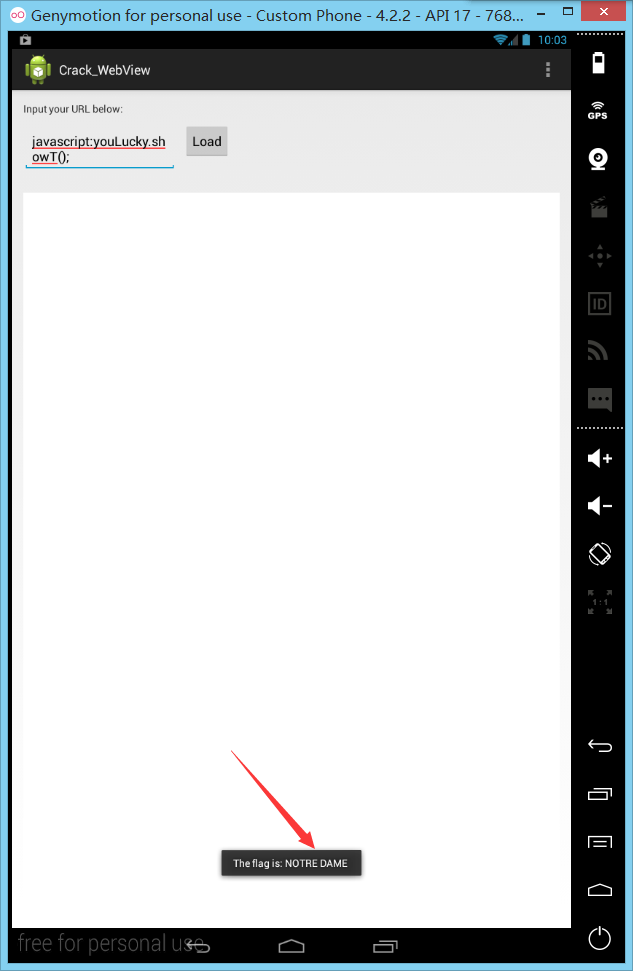


## 0x04 RE04(300pt) 猜我在哪儿？

APK加了爱加密的壳，用gdb gcore dump内存抠出两个dex，找到程序的dex。



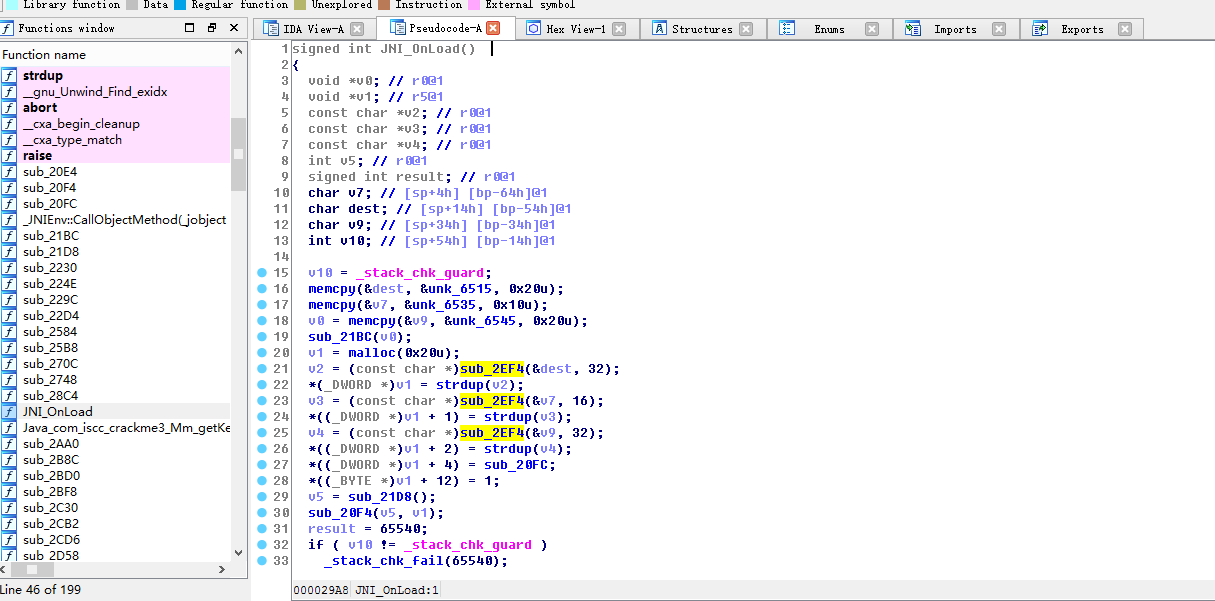
使用youLucky接口，构造js，成功弹出flag



## 0x05 RE05(350pt) 萌萌哒

这题蛮坑的，so中的java\_iscc\_com\_crackme3\_Mm\_getKey函数是假的…返回的结果是ni bei pian le……不过这个函数却可以被我们稍加修改下利用。

重新看下so文件，发现在jni\_onload中，调用了sub\_2EF4这一解密函数。



而在java\_iscc\_com\_crackme3\_Mm\_getKey中也调用了这个函数，因此我们可以用unk\_63cc的数据来替换java\_iscc\_com\_crackme3\_Mm\_getKey中的dest，也就是unk\_6595。

由于Mm文件存在检测，不能直接smali注入，于是我们新建一个类，将Mm.smali的内容全部复制过来，在调用getkey()的后面加上一句

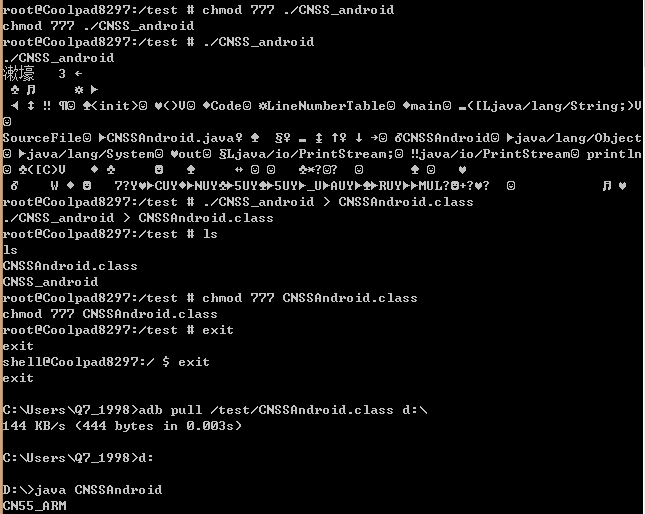
const-string v11, "info"

invoke-static {v11, v3}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I

然后将修改后的smali和so重新打包成apk，安装运行，随意输入16位字母，在logcat中便能看到flag了。

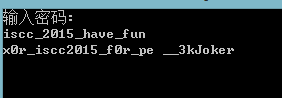
## 0x06 RE06(350pt) 不择手段

这道题目比较简单，拿到压缩包将50 4B 01 02后第五字节改为00解开伪加密压缩包后，发现是一个ELF for ARM，直接push到手机里运行下发现输出结果是个class文件，pull到pc上运行得到flag。

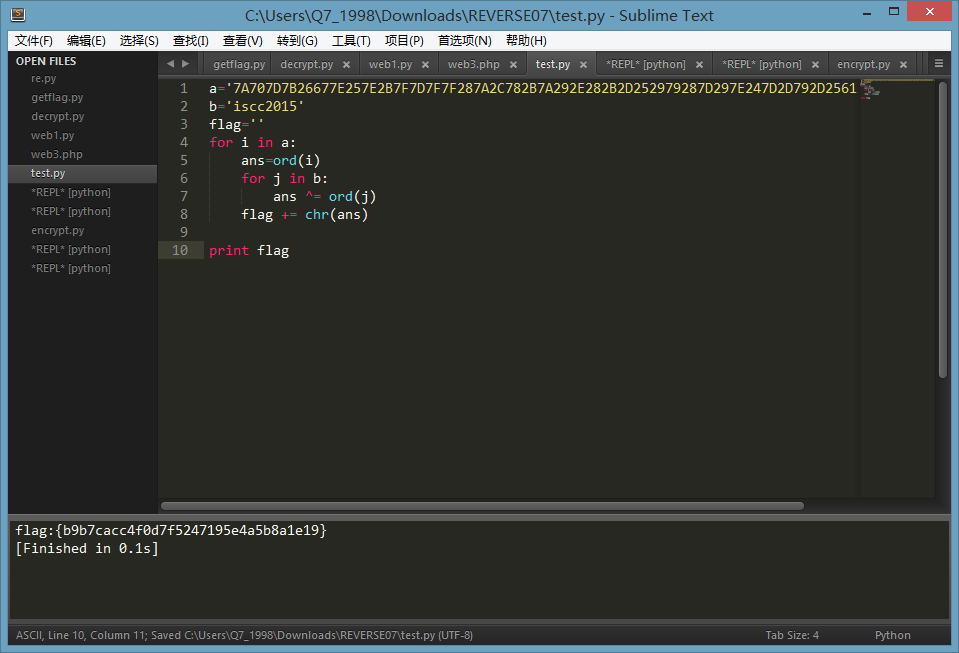


## 0x07 RE07(350pt) Find you there!

这题脑洞略大，直接丢进OD，不难分析出key:iscc\_2015\_hava\_fun



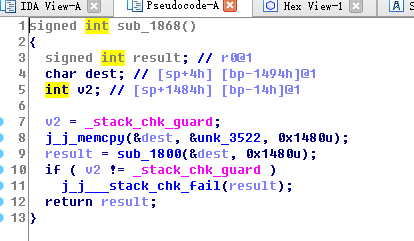
根据得到的提示将rdata异或iscc2015得到flag，不过异或方式略奇葩…



## 0x08 RE08(400pt) 忘记题目名字了orz

去除APK伪加密后直接丢进APKIDE， 发现程序利用dexclassloadr 动态加载了dex ，调用JNI\_onload 函数

再把 so文件丢进IDA 分析 发现程序从数据区取了0x1480长度的数据 进行解密 然后传给了opendexfile 的libdvm.so库函数用来动态加载，因此判断这就应该是加密的dex文件了。



IDA在jni\_onload下断，动态调试APK，跟两次后大概明白了算法。

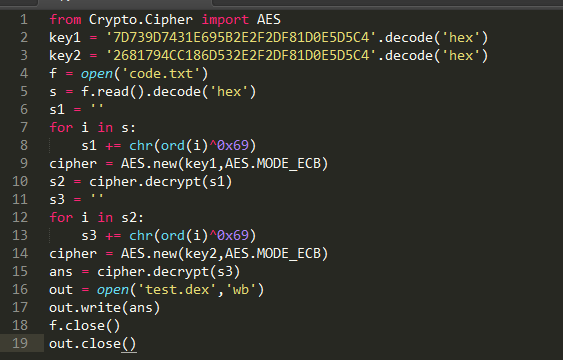
算法是AES的变形，每次AES解密前都将数据先Xor 0x69。

同时动态调试还得到了4个密钥，两个用来解密dex，两个用来解密一些其他字符串。

7D739D7431E695B2E2F2DF81D0E5D5C4

2681794CC186D532E2F2DF81D0E5D5C4

写个小脚本解密dex。



Dex转成jar后丢进jd-gui看下，算法比较简单，写个脚本算出flag

