華中科技大學

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

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。

2. 实验准备

（1）Android 手机；

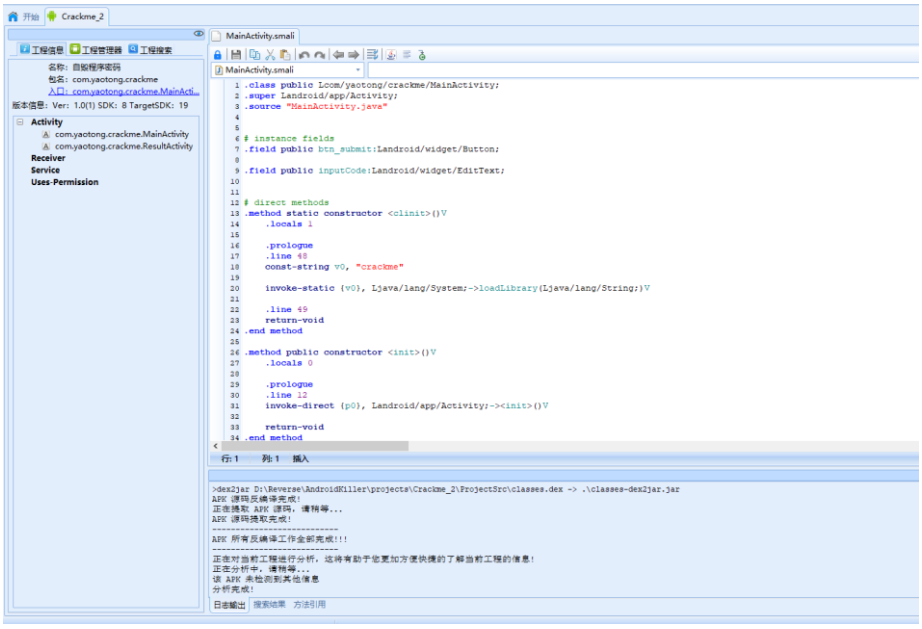
（2）AndroidKiller 软件；

（3）IDA Pro 7.0 软件。

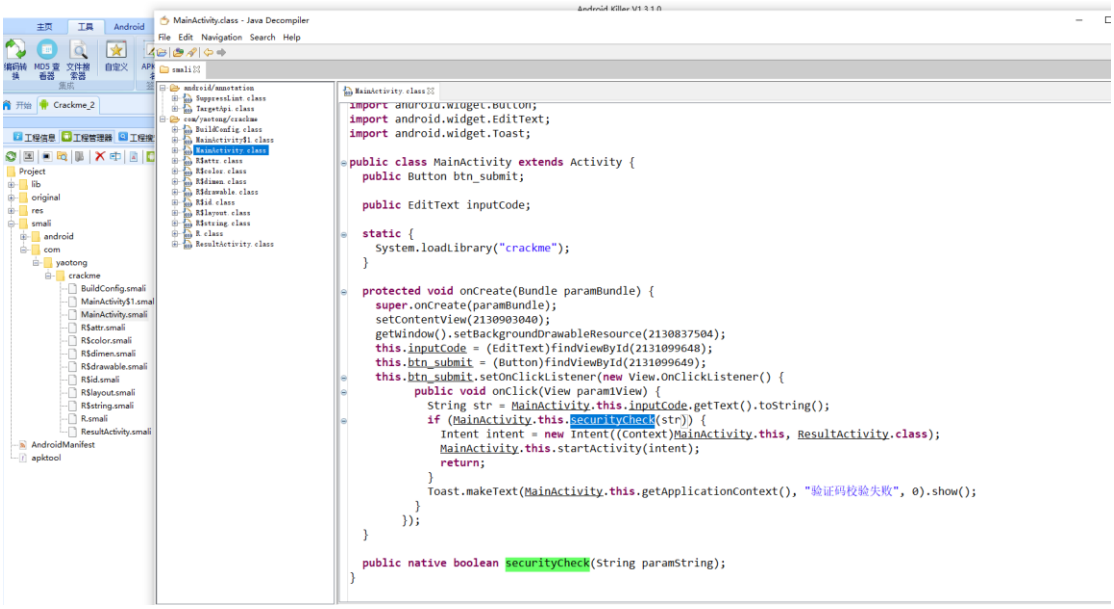
3. 静态分析

3.1. 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：




```
  String str = MainActivity.this.inputCode.getText().toString();
  if (MainActivity.this.securityCheck(str)) {
    Intent intent = new Intent((Context)MainActivity.this, ResultActivity.class);
    MainActivity.this.startActivity(intent);
    return;
  }
  Toast.makeText(MainActivity.this.getApplicationContext(), "验证码校验失败", 0).show();
}
```

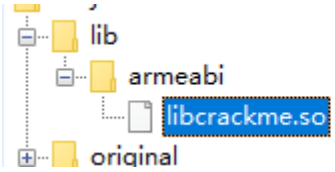
根据反编译的 JAVA 代码可以猜测，如果返回为 0 那么则输出“验证码校验失败”。那么当输入的字符串 str 在 MainActivity.this.securityCheck(str)中返回得到 1 的时候能够避免手机自毁，而当返回值为 1 的时候执行的是 ResultActivity.class 文件中的函数，点开该函数查看其内容发现有字符串“Congratulations!!!You Win!!”输出：

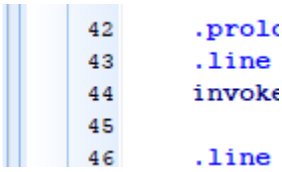
```
public class ResultActivity extends Activity {
  public void onCreate(Bundle paramBundle) {
    super.onCreate(paramBundle);
    TextView textView = new TextView((Context)this);
    textView.setText("Congratulations!!!You Win!!");
    setContentView((View)textView);
  }
}
```

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：

```
static {
  System.loadLibrary("crackme");
}
```

于是在 lib 文件目录下查找文件，在发现了 libcrackme.so 文件：



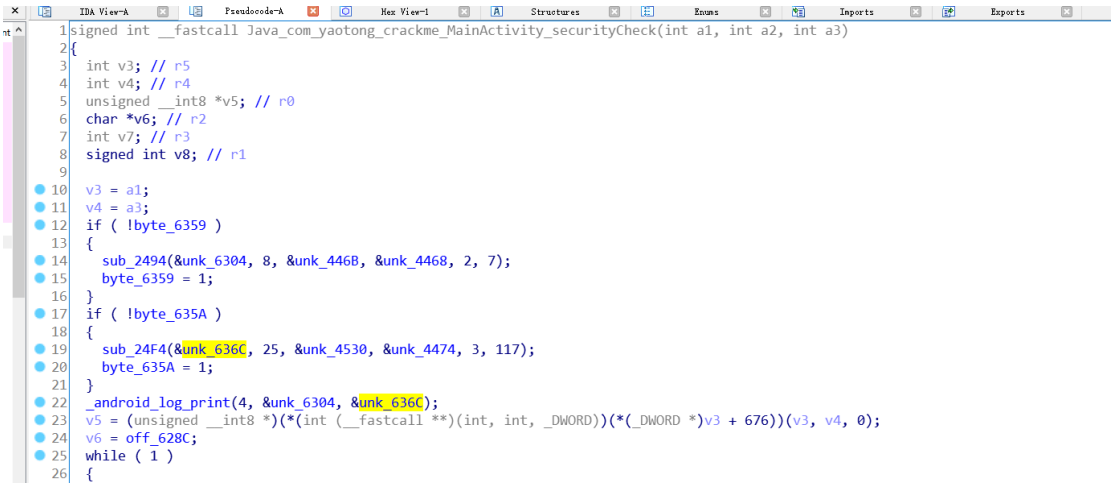


使用 Android killer 直接打开该文件显示的是乱码，所以使用 IDA pro 打开并分析该文件。对 so 文件的分析放在下一节中。

3.2. 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。

```
while ( 1 )
{
  v7 = (unsigned __int8)*v6;
  if ( v7 != *v5 )
    break;
  ++v6;
  ++v5;
  v8 = 1;
  if ( !v7 )
    return v8;
}
return 0;
}
```

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。

```
ata:0000628C off_628C       DCD aWojiushidaan       ; DATA XREF: Java_com_yaotong_crackme_MainActivity_securityCheck+FC↑r
ata:0000628C                                        ; .text:off_13088↑o
ata:0000628C ; .data        ends                    ; "wojiushidaan"
```

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”。

（2）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：

.fini_array	00005E84	00005E8C	R	W	.	.	L	dword	09	public	DATA	32	00	0C
.init_array	00005E8C	00005E94	R	W	.	.	L	dword	0A	public	DATA	32	00	0C
LOAD	00005E94	00005F94	R	W	.	.	L	mempage	02	public	DATA	32	00	0C
.got	00005F94	00006000	R	W	.	.	L	dword	0B	public	DATA	32	00	0C

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：

```
int sub_2378()
{
  return sub_22AC((int)sub_1CA8);
}
```

```
if ( !byte_635F )
{
  sub_24F4((int)&unk_62D7, 6, (int)&unk_4509, (int)"9HbB", 4u, 197);// //dlsym
  byte_635F = 1;
}
v0 = (int (__fastcall *)(signed int, void *))dlsym((void *)0xFFFFFFFF, (const char *)&unk_62D7);
if ( !byte_6360 )
{
  sub_24F4((int)&unk_62EF, 7, (int)&unk_4488, (int)&unk_4488, 2u, 213);// getpid
  byte_6360 = 1;
}
dword_6294 = (int (*)(void))v0(-1, &unk_62EF);
if ( !byte_6361 )
{
  sub_239C(&unk_630C, 8, (char *)&unk_44F3, (int)"LNAt", 4u);// sprintf
  byte_6361 = 1;
}
dword_6298 = v0(-1, &unk_630C);
if ( !byte_6362 )
{
  sub_239C(&unk_62DD, 6, (char *)&unk_44AC, (int)"cOXt", 4u);// fopen
  byte_6362 = 1;
}
dword_629C = v0(-1, &unk_62DD);
if ( !byte_6363 )
{
  sub_24F4((int)&unk_62E3, 6, (int)&unk_4481, (int)"BMT", 3u, 1);// fgets
  byte_6363 = 1;
}
dword_62A0 = (int (__fastcall *)(_DWORD, _DWORD, _DWORD))v0(-1, &unk_62E3);
if ( !byte_6364 )
{
  sub_239C(&unk_62F6, 7, (char *)&unk_44C4, (int)&unk_44C1, 2u);// strstr
  byte_6364 = 1;
}
dword_62A4 = v0(-1, &unk_62F6);
if ( !byte_6365 )
{
  sub_254C(&unk_62FD, 7, &unk_44CC, &unk_44FC, 0, 1);// sscanf
  byte_6365 = 1;
}
```

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。

（3）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。

```
if ( !byte_6359 )
{
  sub_2494(&unk_6304, 8, &unk_446B, &unk_4468, 2, 7);
  byte_6359 = 1;
}
if ( !byte_635A )
{
  sub_24F4(&unk_636C, 25, &unk_4530, &unk_4474, 3, 117);
  byte_635A = 1;
}
```

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。

```
int __fastcall sub_24F4(int result, int a2, int a3, int a4, unsigned int a5, int a6)
{
  int v6; // r7
  unsigned int v7; // r4

  v6 = result;
  if ( a2 )
  {
    v7 = 0;
    do
    {
      result = (*(unsigned __int8 *)(a3 + v7) ^ a6) - *(unsigned __int8 *)(a4 + v7 % a5);
      *(_BYTE *)(v6 + v7++) = result;
    }
    while ( a2 != v7 );
  }
  return result;
}
```

使用交叉引用之后，进入其中一个函数进行查看：

```
while ( 1 )
{
  v5 = (int (__fastcall *)(char *, void *))dword_62A4;// strstr
  if ( !byte_635D )
  {
    v6 = v4;
    v7 = v3;
    v8 = (char *)&GLOBAL_OFFSET_TABLE_ + (_DWORD)v3;// unk_6290
    sub_24F4((int)(v8 + 0x95), 10, (int)&unk_4496, (int)&unk_4493, 2u, 157);// unk_6325=v8+0x95:TracePid
    v8[205] = 1;
    v3 = v7;
    v4 = v6;
  }
  if ( v5(&v14, &unk_6325) )                // strstr(v14, "TracePid")
  {
    _aeabi_memset(v4, 128, 0);
    v12 = 0;
    v9 = (void (__fastcall *)(char *, void *, char *, int *))dword_62A8;// sscanf
    if ( !byte_635E )
    {                                       // unk_62D1 = (_BYTE *)&GLOBAL_OFFSET_TABLE_ + (_DWORD)v3 + 0x41:
                                            // %s %d
      sub_239C((_BYTE *)&GLOBAL_OFFSET_TABLE_ + (_DWORD)v3 + 0x41, 6, (char *)&unk_4461, (int)"L79", 3u);
      *((_BYTE *)&GLOBAL_OFFSET_TABLE_ + (_DWORD)v3 + 206) = 1;
    }
    v9(&v14, &unk_62D1, v4, &v12);          // sscanf(&v14, "%s %d", v4, &v12);
    if ( v12 >= 1 )
      break;
  }
  if ( !dword_62A0(&v14, 512, v11) )        // fgets(&v14, 512, v11)
    return _stack_chk_guard - v16;
}
(*(void (__fastcall **)(int, signed int))((char *)&GLOBAL_OFFSET_TABLE_ + (_DWORD)v3 + (unsigned int)&dword_1C))
  v0,
  9);                                       // dword_62AC:kill
}
return _stack_chk_guard - v16;
}
```

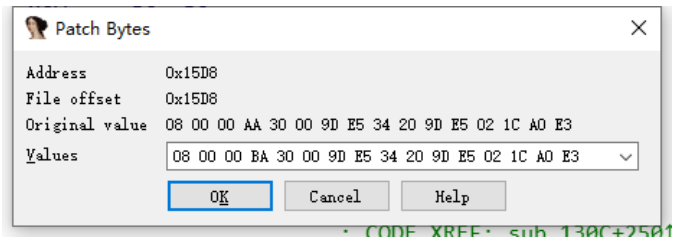
在这个函数中我们发现了反调试程序，v9 是一个函数指针，并且是一个四个参数的函数，当 v12>=1 时就跳出这个循环，而循环节外的这个函数经过解密之后正是 kill 进程自身的函数。经过解密后发现 v9 正是 pthread_create 函数，创建一个监视线程，一旦程序被调试就直接关闭自身。

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。

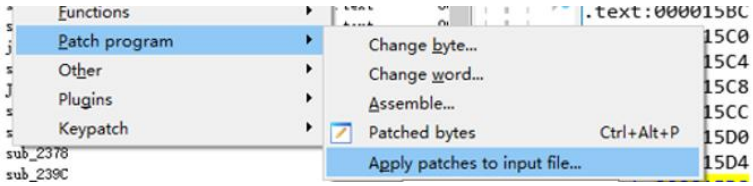
（4）借助于指令修改工具，去除反调试代码；

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：



（5）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。

```
signed int __fastcall JNI_OnLoad(int a1)
{
  int v1; // r4
  _DWORD *v2; // r5
  int v3; // r6
  _DWORD *v4; // r6
  signed int v5; // r6
  int v7; // [sp-8h] [bp-28h]
  char v8; // [sp+0h] [bp-20h]

  v1 = a1;
  dword_62C8 = 0;
  v2 = (_DWORD *)dword_62C4;
  if ( dword_62C4 )
  {
    do
    {
      if ( *v2 >= 1 )
      {
        v3 = 0;
        do
          ((void (*)(void))v2[v3++ + 1])();
        while ( v3 < *v2 );
      }
      v4 = (_DWORD *)v2[11];
      free(v2);
      v2 = v4;
    }
    while ( v4 );
    dword_62C4 = 0;
  }
  dword_62B4(&v8, 0, sub_16A4, 0, 0);
  sub_17F4();
  v5 = 65540;
  if ( (*(int (__fastcall **)(int, int *, signed int))(*(_DWORD *)v1 + 24))(v1, &v7, 65540) )
    v5 = -1;
  return v5;
```

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。

```
}
v7 = dword_62BC();
return ((int (__fastcall *)(int))jolin)(v7);
```

但是 jolin 在文本段中看起来就是一行行的乱码，无用数据段：

```
.text:00001720                 EXPORT jolin
.text:00001720 jolin                                   ; CODE XREF: sub_17F4+3304p
.text:00001720                                         ; DATA XREF: LOAD:000001C8↑o ...
.text:00001720                 MCRLT   p14, 6, SP,c11,c6, 4
.text:00001724                 MOVLTS  PC, #0xF23FFFFF
.text:00001728                 TEQVS   R9, #0xB80000
.text:0000172C                 STRNE   R4, [R8,#-0x7F6]
.text:00001730                 ANDVS   R5, R9, #0xCE00000
.text:00001734                 STRLS   R2, [R5,R4,ASR#13]
.text:00001738                 MOVLTS  R12, #0xD43FFFFF
.text:0000173C                 CMPHI   R6, R4,LSL R7
.text:00001740                 ANDNES  R8, R8, R10,LSL#15
.text:00001744                 ANDVS   R5, R6, R4,ROR#12
.text:00001748                 STRVC   R7, [R6],#0x6E1
.text:0000174C                 MOVLTS  LR, #0xDE3FFFFF
.text:00001750                 LDRGE   R8, [R7],#0x796
.text:00001754                 STRNE   R10, [R3,#0x736]
.text:00001758                 SSATVS  R10, #0xA, R5,ASR#18
.text:0000175C                 STRPLBT R3, [R6],#0x6CF
.text:00001760                 STRLTB  LR, [R6,R3,LSL#11]
.text:00001760 ; ---------------------------------------------------------------------------
.text:00001764                 DCD 0xC716A792, 0x1043C757, 0x64365607, 0x32A216C6, 0xB5C6F5FE
.text:00001764                 DCD 0xE5568BF9, 0x1066E774, 0x64365609, 0x2B706D7, 0xB5C6E5AB
.text:00001764                 DCD 0xB22286A3, 0xB3F9E5D3, 0x63549792, 0x16277794, 0x64174A13
.text:00001764                 DCD 0x92E466C4, 0xB5C6E5F2, 0x83149797, 0x1043971F, 0x66367662
.text:00001764                 DCD 0x72B946D2, 0xB384E58F, 0xA4729D97, 0x1045A735, 0x62545664
.text:00001764                 DCD 0x56E636A2, 0xB5C6D587, 0xC323B7BA, 0x14A83865, 0x6F2BDE56
.text:00001764                 DCD 0xD7665E4E, 0x5666F757, 0xF929784A, 0xF587E5A2, 0xF2F47A13
.text:00001764                 DCD 0x881973B5
.text:000017F4
```

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。

```
v5 = ((v4 + 3) * v4 + 2) * v4;
v6 = 0;
switch ( v5 % 6 + 4 )
{
```

选择语句进入 case 4，因为 jolin 的地址为 1720，那么&jolin & 0xFFFFFFFE 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。

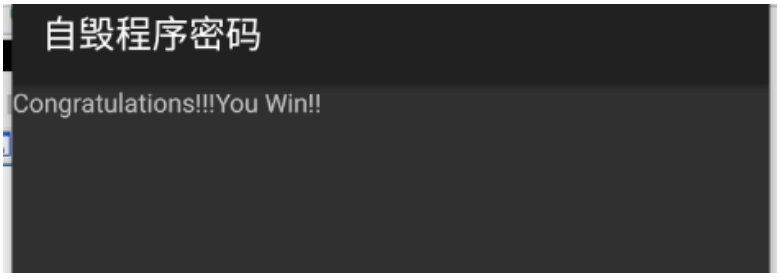
```
.text:00001720 jolin
.text:00001720
.text:00001720                 MCRLT   p
```

```
case 4:
  do
  {
    *(_BYTE *)(v6 + ((unsigned int)&jolin & 0xFFFFFFFE)) ^= byte_6004[v6 % 108];
    ++v6;
  }
  while ( (int *)v6 != &dword_D4 );
  break;
```

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：

```
int jolin()
{
  char *v0; // r4
  char *v1; // r2

  v0 = off_628C;
  ((void (__fastcall *)(unsigned int))*(&end - 360710218))((unsigned int)off_628C & -_page_size);
  *((_BYTE *)&dword_0 + (_DWORD)v0 + 1) = 'i';
  *v0 = 97;
  *((_BYTE *)&dword_0 + (_DWORD)v0 + 2) = 'y';
  *((_BYTE *)&dword_0 + (_DWORD)v0 + (unsigned int)&dword_0 + 1 + 2) = 'o';
  byte_5[(_DWORD)v0] = ',';
  byte_4[(_DWORD)v0] = 'u';
  byte_4[(_DWORD)v0 + (unsigned int)&dword_0 + 2] = 'b';
  byte_4[(_DWORD)v0 + (unsigned int)&dword_0 + 2 + (_DWORD)&dword_0 + 1] = 'u';
  byte_9[(_DWORD)v0] = 'u';
  byte_8[(_DWORD)v0] = 'c';
  v1 = &byte_8[(_DWORD)v0];
  *((_BYTE *)&dword_0 + (_DWORD)v1 + 3) = 'o';
  *((_BYTE *)&dword_0 + (_DWORD)v1 + 2) = 'o';
  return ((int (__fastcall *)(char *, char *, _DWORD))*(&end - 360710217))(v0, v0 + 4096, 0);
}
```

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“：



4. 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

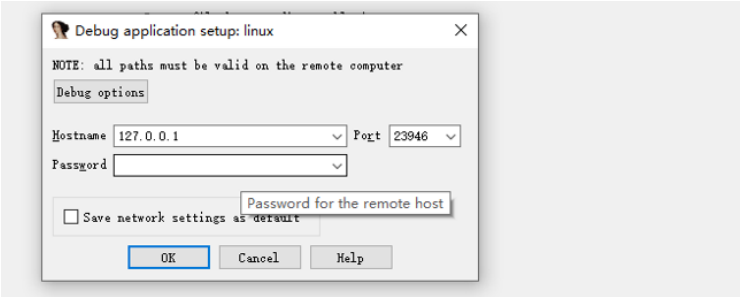
将文件 push 进手机的指定目录下，进入手机端命令：adb shell，切换获取手机的 root 权限：su，查找 push 的文件是否在手机中：ls -l，拥有 root 权限更改文件的权限为 777：chmod 777 android_server。具体指令执行如下：

```
D:\Reverse\Adroid模拟器\Nox\bin>adb connect 127.0.0.1:62001
already connected to 127.0.0.1:62001

D:\Reverse\Adroid模拟器\Nox\bin>adb push D:\Reverse\IDA_Pro_v7.0_Portable\dbgsrv\android_server /data/local/tmp/
[100%] /data/local/tmp/android_server

D:\Reverse\Adroid模拟器\Nox\bin>adb shell
root@shamu:/ # su
root@shamu:/ # cd /data/local/tmp/
root@shamu:/data/local/tmp # ls -l
-rw-rw-rw- root     root       589588 2017-09-14 15:08 android_server
root@shamu:/data/local/tmp #
```

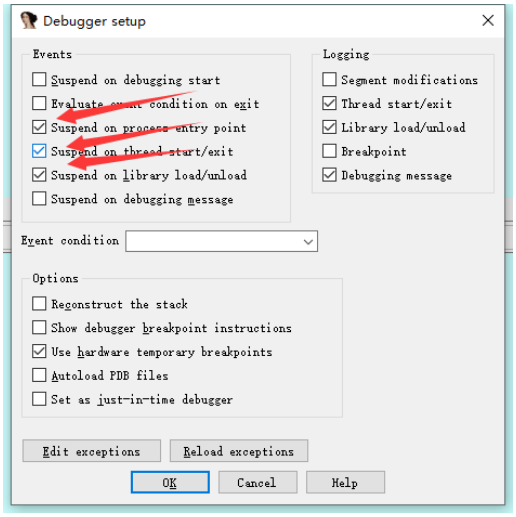
将模拟器运行端口转发到 PC：adb forward tcp:23946 tcp:23946，为了反反调试：程序的 so 文件在加载阶段会先执行 JNI_OnLoad，之后就不再执行，在程序的 so 文件加载阶段才能给 JNI_OnLoad 打断点调试。于是使用 adb shell am start -D -n com.yaotong.crackme/.MainActivity 指令，使调试 APP 挂在加载界面。

```
(c) 2019 Microsoft Corporation。保留所有权利。

C:\Users\12432>adb forward tcp:23946 tcp:23946
23946

C:\Users\12432>adb shell am start -D -n com.yaotong.crackme/com.yaotong.crackme.MainActivity
Starting: Intent { cmp=com.yaotong.crackme/.MainActivity }

C:\Users\12432>adb shell am start -D -n com.yaotong.crackme/com.yaotong.crackme.MainActivity
Starting: Intent { cmp=com.yaotong.crackme/.MainActivity }

C:\Users\12432>
```

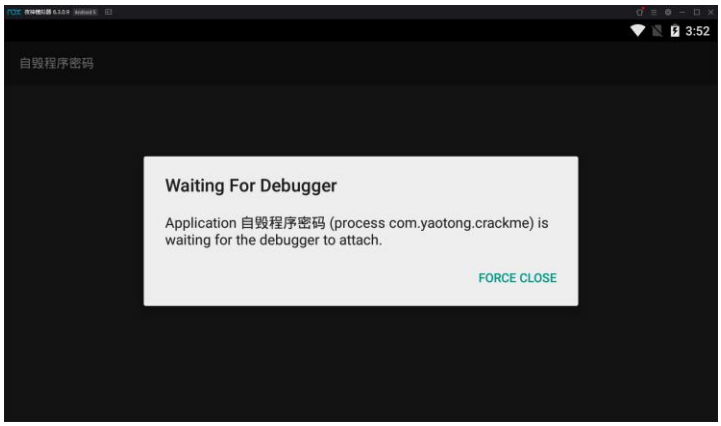
打开 IDA 使用动态调试功能监听 23946 这个端口，而后附加相应的 com.yaotong.crackme 进程：



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：



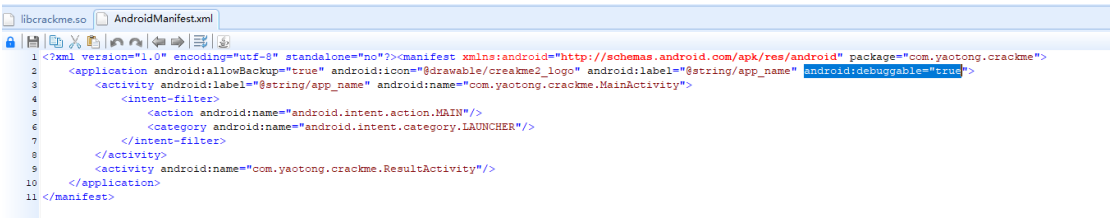
而后点击运行键(或者 F9)发现只能运行以下就卡住了，这是因为在模拟器中运行的进程没有继续执行所以不能进行后续的调试，这时就需要使用 ps | grep com.yaotong.crackme 先获得 APP 的进程 ID，而后再使用 adb forward tcp:8700 jdwp:**** 监听进程的 ID，最后执行 db -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700 来运行进程。



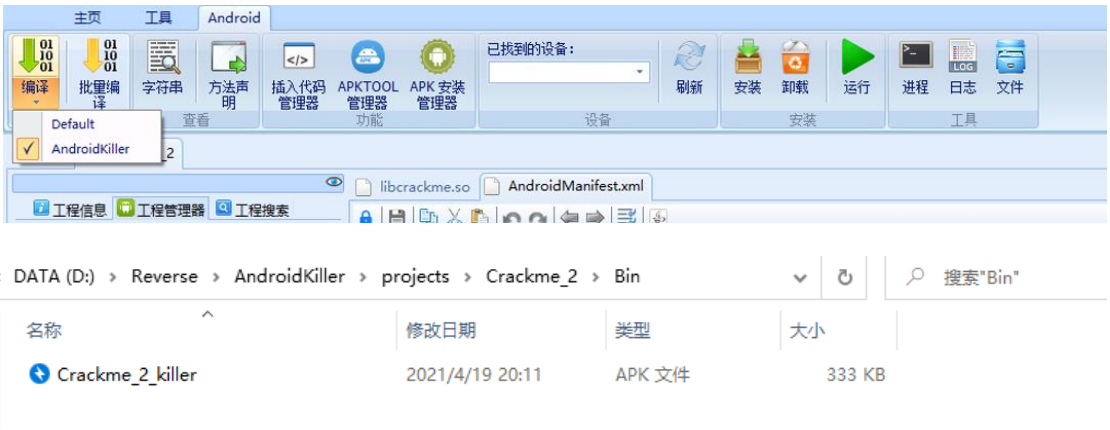
但在执行指令的过程中发现 Jdb connect 执行进程无法成功，甚至一开始 jdb 都不能使用，在修改好 jdb 的环境变量后，jdb 终于可以使用了，但是还是不能够执行该指令让自毁程序继续执行下去。

```
C:\Users\12432>jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700
java.io.IOException: handshake failed - connection prematurally closed
        at jdk.jdi/com.sun.tools.jdi.SocketTransportService.handshake(SocketTransportService.java:137)
        at jdk.jdi/com.sun.tools.jdi.SocketTransportService.attach(SocketTransportService.java:271)
        at jdk.jdi/com.sun.tools.jdi.GenericAttachingConnector.attach(GenericAttachingConnector.java:119)
        at jdk.jdi/com.sun.tools.jdi.SocketAttachingConnector.attach(SocketAttachingConnector.java:83)
        at jdk.jdi/com.sun.tools.example.debug.tty.VMConnection.attachTarget(VMConnection.java:558)
        at jdk.jdi/com.sun.tools.example.debug.tty.VMConnection.open(VMConnection.java:367)
        at jdk.jdi/com.sun.tools.example.debug.tty.Env.init(Env.java:63)
        at jdk.jdi/com.sun.tools.example.debug.tty.TTY.main(TTY.java:1113)

致命错误:
无法附加到目标 VM。
```

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。

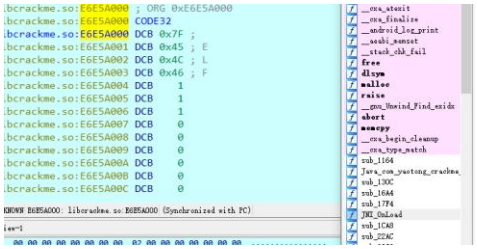


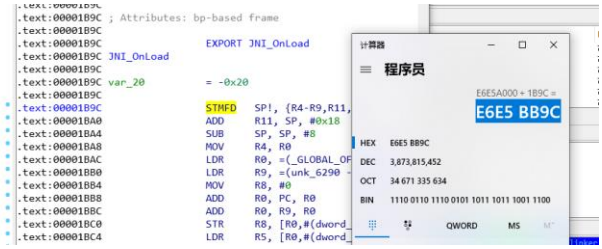
重新进行 apk 编译的时候，编译功能的默认选项是 Default，而需要生成整个可执行的 apk 需要勾选 AndroidKiller 项，否则会生成只有 1Kb 的 apk 文件。（在实验过程中遇到这个问题还查了好久）。



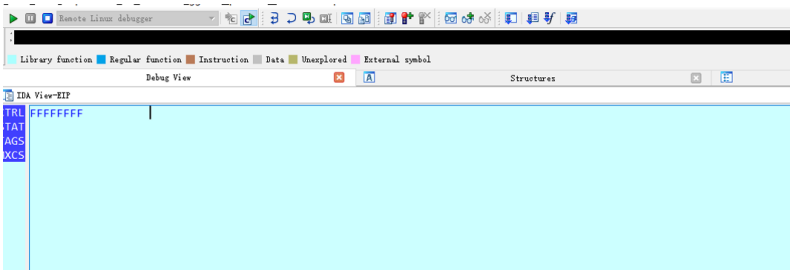
而后再次重复上述步骤即可让自毁程序继续执行下去：

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：



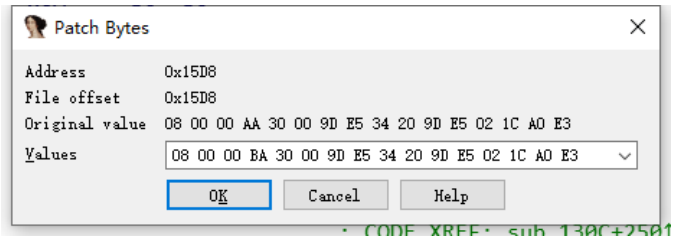
即使在尝试 MuMu 模拟器也不能正确执行后，选择了下载 Android Emulator 来模拟真机进行动态调试。动态调试的基本流程与上述一致，以下给出执行过程截图以及部分重要部分解释。

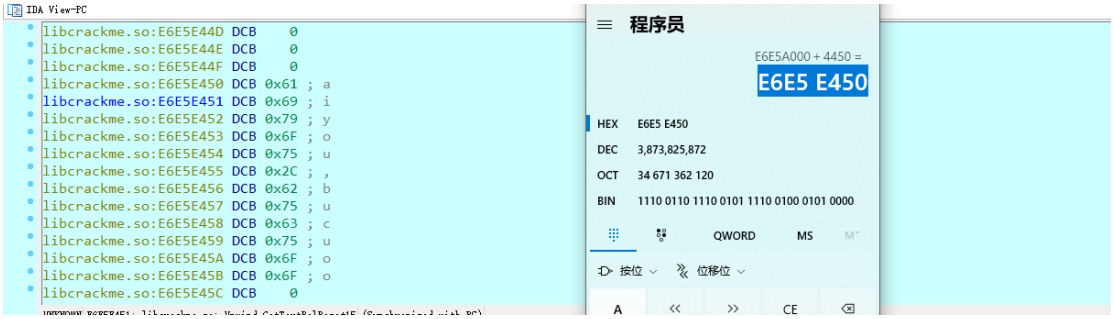
```
C:\Users\12432>adb connect 127.0.0.1:5554
cannot connect to 127.0.0.1:5554: 由于目标计算机积极拒绝，无法连接。 (10061)

C:\Users\12432>adb devices
List of devices attached
emulator-5554   device


C:\Users\12432>
```

```
C:\Users\12432>adb push D:\Reverse\IDA_Pro_v7.0_Portable\dbgsrv\android_server /data/local/tmp
D:\Reverse\IDA_Pro_v7.0_Portable\dbgsrv\android_server: 1 file pushed, 0 skipped. 4.7 MB/s (589588 bytes in 0.119s)

C:\Users\12432>adb shell
generic_arm64:/ $ su
generic_arm64:/ # cd /data/local/tmp
generic_arm64:/data/local/tmp # ls -l
total 1152
-rw-rw-rw- 1 shell shell 589588 2017-09-14 07:08 android_server
generic_arm64:/data/local/tmp # chmod 777 android_server
generic_arm64:/data/local/tmp # ls -l
total 1152
-rwxrwxrwx 1 shell shell 589588 2017-09-14 07:08 android_server
generic_arm64:/data/local/tmp #
```

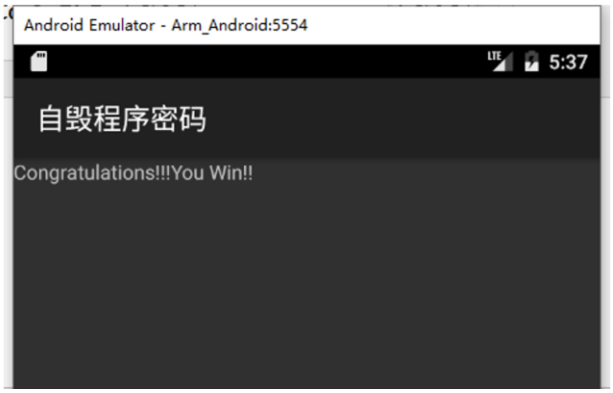
```
C:\Users\12432>adb shell am start -D -n com.yaotong.crackme/.MainActivity
Starting: Intent { cmp=com.yaotong.crackme/.MainActivity }

C:\Users\12432>adb shell
root@shamu:/ # su
root@shamu:/ # ps | grep crackme
u0_a31    3316  1456  991580 52392 ffffffff b743c4d2 S com.yaotong.crackme
root@shamu:/ # ps | grep crackme
u0_a31    3316  1456  991580 52392 ffffffff b743c4d2 S com.yaotong.crackme
root@shamu:/ # exit
root@shamu:/ # exit

C:\Users\12432>adb forward tcp:8700 jdwp:3316

C:\Users\12432>jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700
设置未捕获的java.lang.Throwable
设置延迟的未捕获的java.lang.Throwable
正在初始化jdb...
>
```

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：



5. 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！