Android逆向之旅---动态方式破解apk进阶篇(IDA调试so源码)

尼古拉斯.赵四 于 2020-04-18 21:28:30 发布 1560 收藏 13
文章标签： android
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/m0_46204016/article/details/105605620
版权

一、前言

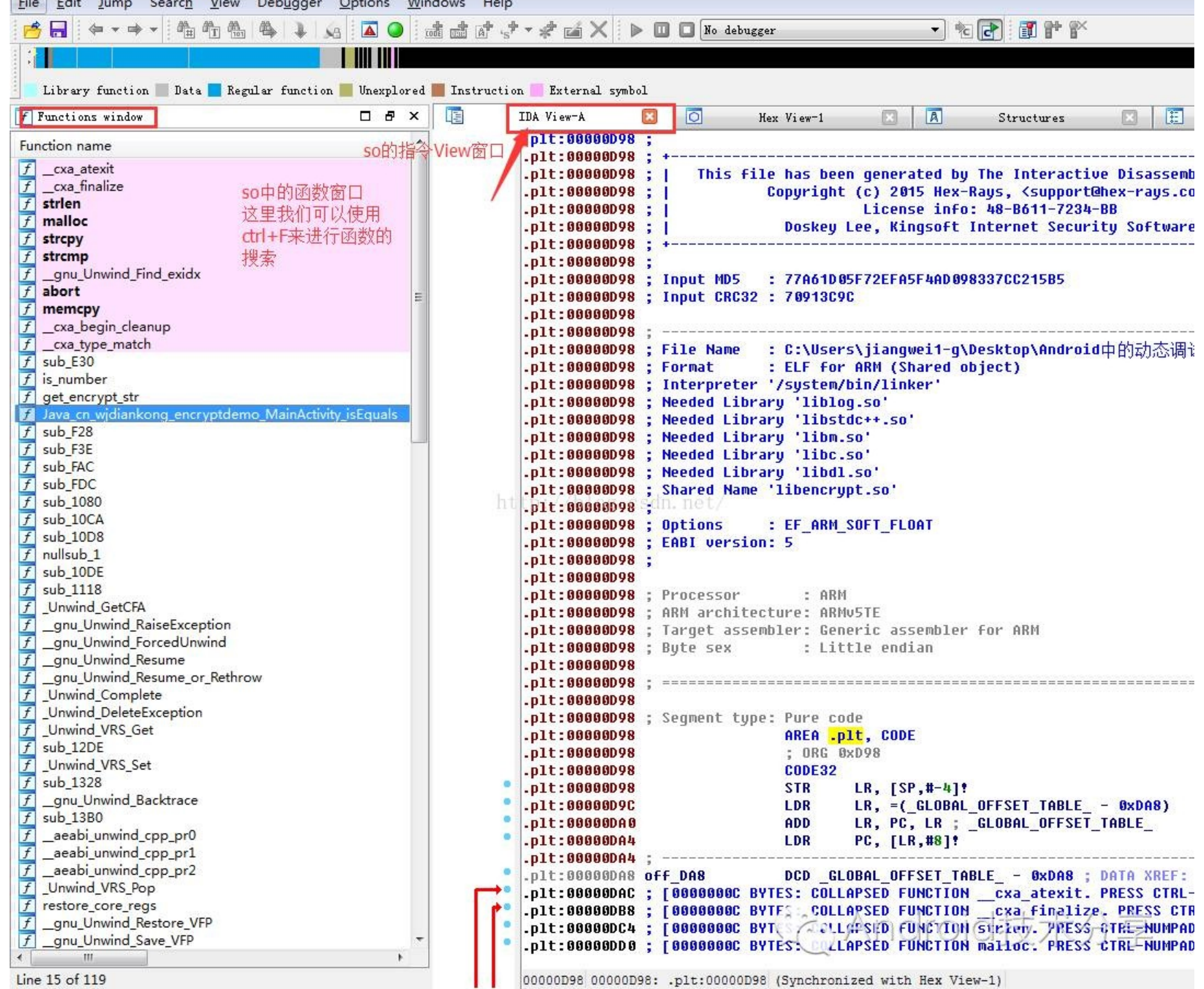
今天我们继续来看破解apk的相关知识，在前一篇：Eclipse动态调试smali源码破解apk 我们今天主要来看如何使用IDA来调试Android中的native源码，因为现在一些app，为了安全或者效率问题，会把一些重要的功能放到native层，那么这样一来，我们前篇说到的Eclipse调试smali源码就显得很无力了，因为核心的都在native层，Android中一般native层使用的是so库文件，所以我们这篇就来介绍如何调试so文件的内容，从而让我们破解成功率达到更高的一层。

二、知识准备

我们在介绍如何调试so文件的时候，先来看一下准备知识：

第一、IDA工具的使用

早在之前的使用IDA工具静态分析so文件，通过分析arm指令，来获取破解信息，比如打印的log信息，来破解apk的，在那时候我们就已经介绍了如何使用IDA工具：



这里有多个窗口，也有多个视图，用到最多的就是：

1、Function Window对应的so函数区域：这里我们可以使用ctrl+f进行函数的搜索

2、IDA View对应的so中代码指令视图：这里我们可以查看具体函数对应的arm指令代码

3、Hex View对应的so的十六进制数据视图：我们可以查看arm指令对应的数据等

当然在IDA中我们还需要知道一些常用的快捷键：

1、强大的F5快捷键可以将arm指令转化成可读的C语言，帮助分析

```
.text:00000EC8                 EXPORT Java_cn_wjdiankong_encryptdemo_MainActivity_isEquals
.text:00000EC8 Java_cn_wjdiankong_encryptdemo_MainActivity_isEquals
.text:00000EC8                 PUSH    {R3-R7,LR}
.text:00000ECA                 MOVS    R3, #0xA9
.text:00000ECC                 MOVS    R6, R2
.text:00000ECE                 LDR     R2, [R0]
.text:00000ED0                 LSLS    R3, R3, #2
.text:00000ED2                 MOVS    R1, R6
.text:00000ED4                 LDR     R3, [R2,R3]
```

http://blog.csdn.net/
Android技术分享

首先选中需要翻译成C语言的函数，然后按下F5：

```
int __fastcall Java_cn_wjdiankong_encryptdemo_MainActivity_isEquals(int a1, int a2, int a3)
{
  int v3; // r6@1
  int v4; // r5@1
  const char *v5; // r0@1
  const char *v6; // r4@1
  size_t v7; // r0@1
  char *v8; // r0@1
  int v9; // r0@1
  int v10; // r3@1
  const char *v11; // r0@2
  unsigned int v12; // r7@2

  v3 = a3;
  v4 = a1;
  v5 = (const char *)(*(int (**)(void))(*(_DWORD *)a1 + 676))();
  v6 = v5;
  v7 = j_j_strlen(v5);
  v8 = (char *)j_j_malloc(v7);
  j_j_strcpy(v8, v6);
  v9 = is_number(v6);
  v10 = 0;
  if ( v9 )
  {
    v11 = (const char *)get_encrypt_str(v6);
    v12 = j_j_strcmp("ssBCqpBssP", v11);
    (*(void (__fastcall **)(int, int, const char *))(*(_DWORD *)v4 + 680))(v4, v3, v6);
    v10 = v12 <= 0;
  }
  return v10;
}
```

看到了，立马感觉清爽多了，这些代码看起来应该会好点了。

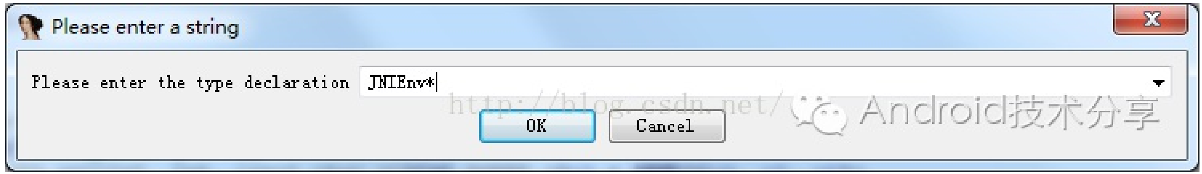
下面我们还需要做一步，就是**还原JNI函数方法名**，一般JNI函数方法名首先是一个指针加上一个数字，比如v3+676。然后将这个地址作为一个方法指针进行方法调用，并且第一个参数就是指针自己，比如(v3+676)(v3…)。这实际上就是我们在JNI里经常用到的JNIEnv方法。因为Ida并不会自动的对这些方法进行识别，所以当我们对so文件进行调试的时候经常会见到却搞不清楚这个函数究竟在干什么，因为这个函数实在是太抽象了。解决方法非常简单，只需要对JNIEnv指针做一个类型转换即可。比如说上面提到a1和v4指针：

```
v3 = a3;
v4 = a1;
v5 = (const char *)(*(int (**)(void))(*(_DWORD *)a1 + 676))();
v6 = v5;
v7 = j_j_strlen(v5);                      int a1; // r0@0 ISAR
v8 = (char *)j_j_malloc(v7);
j_j_strcpy(v8, v6);
v9 = is_number(v6);
v10 = 0;
if ( v9 )
{
  v11 = (const char *)get_encrypt_str(v6);
  v12 = j_j_strcmp("ssBCqpBssP", v11);
  (*(void (__fastcall **)(int, int, const char *))(*(_DWORD *)v4 + 680))
```

这里看到变量后面跟着一个数字
这里其实是一个函数指针，我们可以
使用y快捷键，修改成JNIEnv即可

我们可以选中a1变量，然后按一下y键：

然后将类型声明为：JNIEnv*。



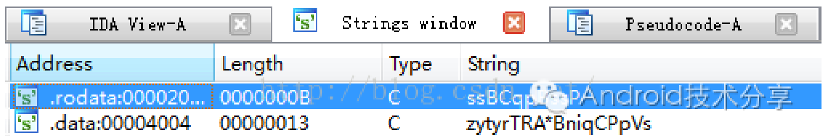
确定之后再来看：

```
v3 = a3;
v4 = a1;
v5 = (const char *)((int (*)(void))(*a1)->GetStringUTFChars)();
v6 = v5;
v7 = j_j_strlen(v5);
v8 = (char *)j_j_malloc(v7);
j_j_strcpy(v8, v6);
v9 = is_number(v6);
v10 = 0;
if ( v9 )
{
  v11 = (const char *)get_encrypt_str(v6);
  v12 = j_j_strcmp("ssBCqpBssP", v11);
  ((void (__fastcall *)(JNIEnv *, int, const char *))(*v4)->ReleaseStringUTFChars)(v4, v3, v6);
  v10 = v12 <= 0;
}
return v10;
```

修改之后，是不是瞬间清晰了很多？另外有人（貌似是看雪论坛上的）还总结了所有JNIEnv方法对应的数字，地址以及方法声明：

672	GetStringUTFLength	jsize (*)(JNIEnv*, jstring)
676	GetStringUTFChars	const char* (*)(JNIEnv*, jstring, jboolean*)
680	ReleaseStringUTFChars	void (*)(JNIEnv*, jstring, const char*)
684	GetArrayLength	jsize (*)(JNIEnv*, jarray)
688	NewObjectArray	jobjectArray (*)(JNIEnv*, jsize, jclass, jobject)

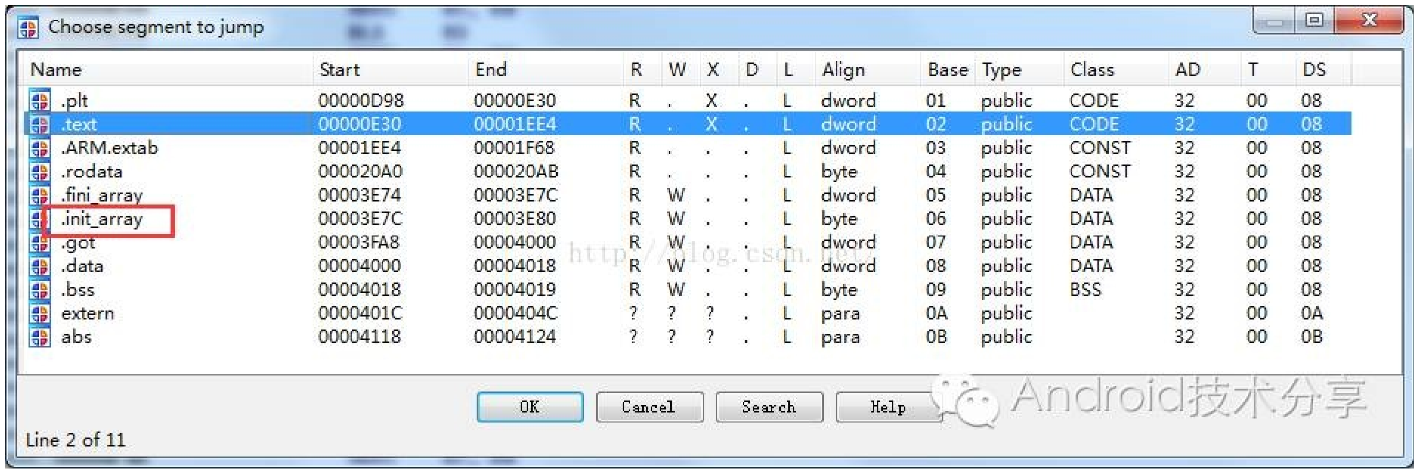
2、Shirt+F12快捷键，速度打开so中所有的字符串内容窗口



Address	Length	Type	String
.rodata:000020...	0000000B	C	ssBCq...
.data:00004004	00000013	C	zytyrTRA*BniqCPpVs

有时候，字符串是一个非常重要的信息，特别是对于破解的时候，可能就是密码，或者是密码库信息。

3、Ctrl+S快捷键，有两个用途，在正常打开so文件的IDA View视图的时候，可以查看so对应的Segement信息



Choose segment to jump

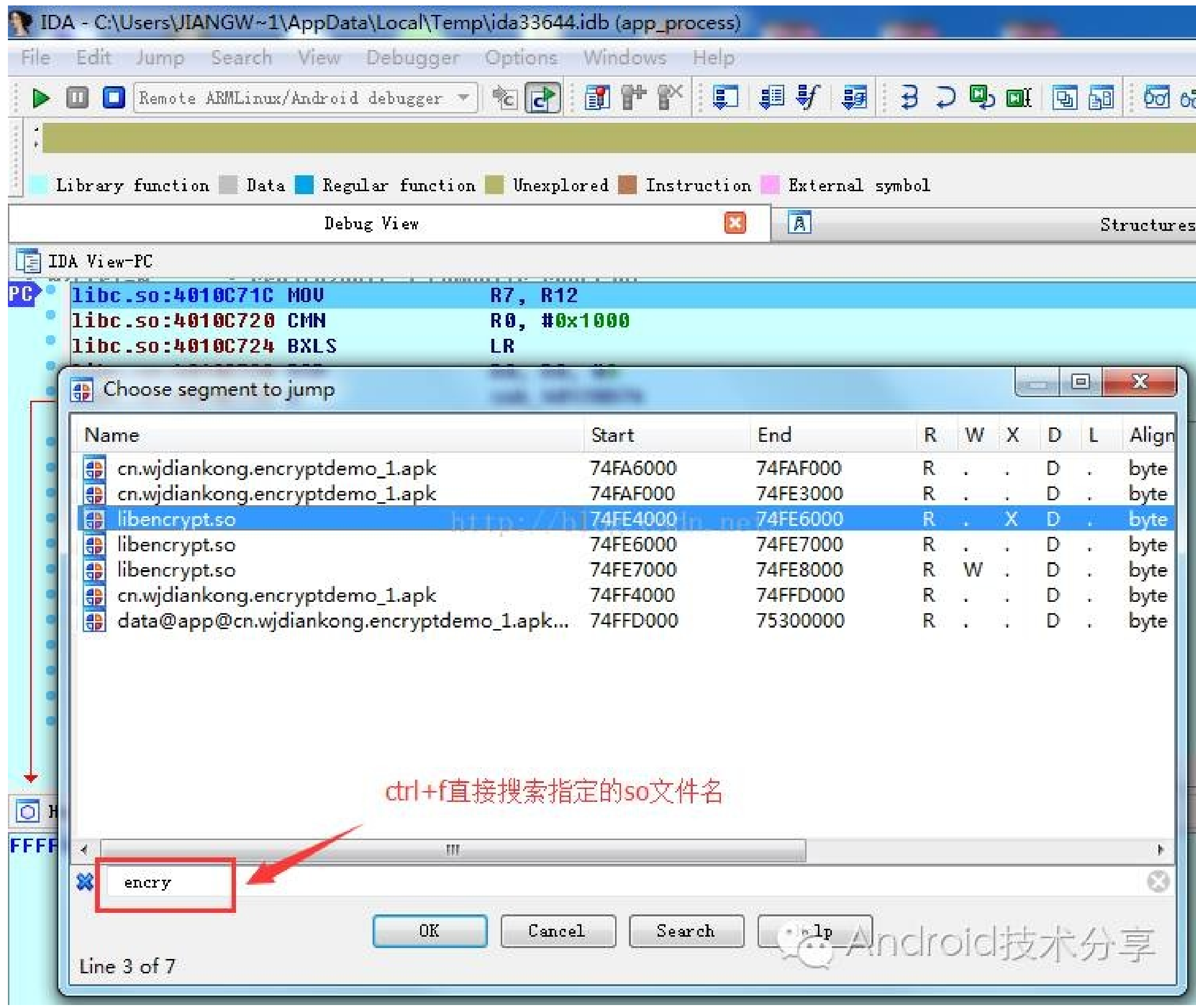
Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	T	DS
.plt	00000D98	00000E30	R	.	X	.	L	dword	01	public	CODE	32	00	08
.text	00000E30	00001EE4	R	.	X	.	L	dword	02	public	CODE	32	00	08
.ARM.extab	00001EE4	00001F68	R	.	.	.	L	dword	03	public	CONST	32	00	08
.rodata	000020A0	000020AB	R	.	.	.	L	byte	04	public	CONST	32	00	08
.fini_array	00003E74	00003E7C	R	W	.	.	L	dword	05	public	DATA	32	00	08
.init_array	00003E7C	00003E80	R	W	.	.	L	byte	06	public	DATA	32	00	08
.got	00003FA8	00004000	R	W	.	.	L	dword	07	public	DATA	32	00	08
.data	00004000	00004018	R	W	.	.	L	dword	08	public	DATA	32	00	08
.bss	00004018	00004019	R	W	.	.	L	byte	09	public	BSS	32	00	08
extern	0000401C	0000404C	?	?	?	.	L	para	0A	public		32	00	0A
abs	00004118	00004124	?	?	?	.	L	para	0B	public		32	00	0B

OK Cancel Search Help

Line 2 of 11

可以快速得到，一个段的开始位置和结束位置，不过这个位置是相对位置，不是so映射到内存之后的位置，关于so中的段信息，不了解的同学可以参看这篇文章：Android中so文件格式详解 这篇文章介绍的很很清楚了，这里就不在作介绍了。

当在调试页面的时候，ctrl+s可以快速定位到我们想要调试的so文件映射到内存的地址：



因为一般一个程序，肯定会包含多个so文件的，比如系统的so就有好多的，一般都是在/system/lib下面，当然也有我们自己的so，这里我们看到这里的开始位置和结束位置就是这个so文件映射到内存中：

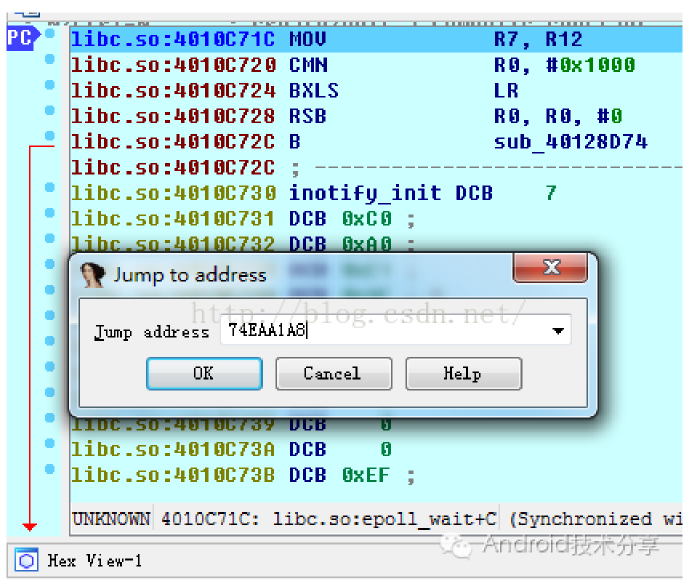
```
C:\Users\jiangwei1-g>adb shell
shell@pisces:/ $ su
root@pisces:/ # ps |grep cn.wjdiankong
u0_a145   16936 7236  880856 47852 ffffffff 4010c71c t cn.wjdiankong.encryptdemo
root@pisces:/ # cd /proc/16936
root@pisces:/proc/16936 # ll
dr-xr-xr-x u0_a145  u0_a145           2016-05-25 21:06 attr
-r-------- u0_a145  u0_a145         0 2016-05-25 21:06 auxv
-r--r--r-- u0_a145  u0_a145         0 2016-05-25 21:00 cgroup
--w------- u0_a145  u0_a145         0 2016-05-25 21:06 clear_refs
-r--r--r-- u0_a145  u0_a145         0 2016-05-25 20:56 cmdline
-rw-r--r-- u0_a145  u0_a145         0 2016-05-25 21:06 comm
lrwxrwxrwx u0_a145  u0_a145           2016-05-25 21:06 cwd -> /
-r-------- u0_a145  u0_a145         0 2016-05-25 21:06 environ
lrwxrwxrwx u0_a145  u0_a145           2016-05-25 20:57 exe -> /system/bin/app_process
dr-x------ u0_a145  u0_a145           2016-05-25 21:06 fd
dr-x------ u0_a145  u0_a145           2016-05-25 21:06 fdinfo
-r--r--r-- u0_a145  u0_a145         0 2016-05-25 21:06 limits
-rw-r--r-- u0_a145  u0_a145         0 2016-05-25 21:06 loginuid
-r--r--r-- u0_a145  u0_a145         0 2016-05-25 20:57 maps
-rw------- u0_a145  u0_a145         0 2016-05-25 21:06 mem
-r--r--r-- u0_a145  u0_a145         0 2016-05-25 21:06 mountinfo
-r--r--r-- u0_a145  u0_a145         0 2016-05-25 21:06 mounts
-r-------- u0_a145  u0_a145         0 2016-05-25 21:06 mountstats
dr-xr-xr-x u0_a145  u0_a145           2016-05-25 21:06 net
dr-x--x--x u0_a145  u0_a145           2016-05-25 21:06 ns
-rw-r--r-- u0_a145  u0_a145         0 2016-05-25 20:56 oom_adj
-r--r--r-- u0_a145  u0_a145         0 2016-05-25 21:06 oom_score
-rw-r--r-- u0_a145  u0_a145         0 2016-05-25 21:06 oom_score_adj
-r--r--r-- u0_a145  u0_a145         0 2016-05-25 21:06 pagemap
-r--r--r-- u0_a145  u0_a145         0 2016-05-25 21:06 personality
lrwxrwxrwx u0_a145  u0_a145           2016-05-25 21:06 root -> /
-r--r--r-- u0_a145  u0_a145         0 2016-05-25 21:06 sessionid
-r--r--r-- u0_a145  u0_a145         0 2016-05-25 20:57 smaps
-r--r--r-- u0_a145  u0_a145         0 2016-05-25 21:06 stack
-r--r--r-- u0_a145  u0_a145         0 2016-05-25 20:56 stat
-r--r--r-- u0_a145  u0_a145         0 2016-05-25 21:06 statm
-r--r--r-- u0_a145  u0_a145         0 2016-05-25 21:06 status
dr-xr-xr-x u0_a145  u0_a145           2016-05-25 20:56 task
-r--r--r-- u0_a145  u0_a145         0 2016-05-25 21:06 wchan
root@pisces:/proc/16936 # cat maps |grep encry
74fa6000-74faf000 r--s 00154000 b3:1b 16433      /data/app/cn.wjdiankong.encryptdemo-1.apk
74faf000-74fe3000 r--s 00039000 b3:1b 16433      /data/app/cn.wjdiankong.encryptdemo-1.apk
74fe4000-74fe6000 r-xp 00000000 b3:1b 49200      /data/app-lib/cn.wjdiankong.encryptdemo-1/libencrypt.so
74fe6000-74fe7000 r--p 00001000 b3:1b 49200      /data/app-lib/cn.wjdiankong.encryptdemo-1/libencrypt.so
74fe7000-74fe8000 rw-p 00002000 b3:1b 49200      /data/app-lib/cn.wjdiankong.encryptdemo-1/libencrypt.so
74ff4000-74ffd000 r--s 00154000 b3:1b 16433      /data/app/cn.wjdiankong.encryptdemo-1.apk
74ffd000-75300000 r--p 00000000 b3:1b 24785      /data/dalvik-cache/data@app@cn.wjdiankong.encryptdemo-1.
root@pisces:/proc/16936 #
```


http://blog.csdn.net/
Android技术分享

这里我们可以使用cat命令查看一个进程的内存映射信息：**cat /proc/[pid]/maps**

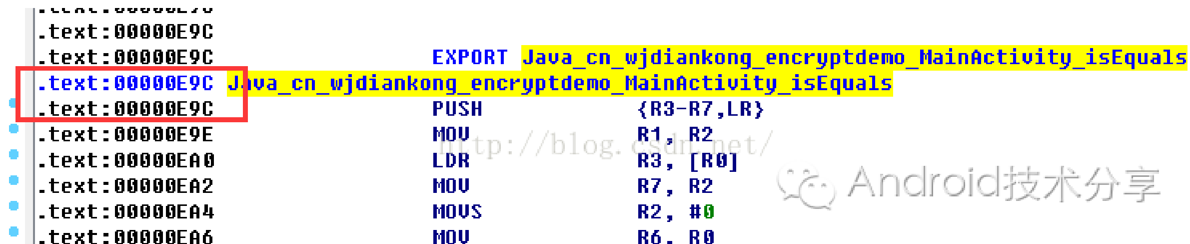
我们看到映射信息中有多so文件，其实这个不是多个so文件，而是so文件中对应的不同Segement信息被映射到内存中的，一般是代码段，数据段等，因为我们需要调试代码，所以我们只关心代码段，代码段有一个特点就是具有执行权限x，所以我们只需要找到权限中有x的那段数据即可。

4、G快捷键：在IDA调试页面的时候，我们可以使用S键快速跳转到指定的内存位置



这里的跳转地址，是可以算出来的，比如我现在想跳转到A函数，然后下断点，那么我们可以使用上面说到的ctrl+s查找到so文件的内存开始的基地址，然后再用IDA View中查看A函数对应的相对地址，相加就是绝对地址，然后跳转到即可，比如这里的：

Java_cn_wjdiankong_encryptdemo_MainActivity_isEquals 函数的IDA View中的相对地址(也就是so文件的地址)：E9C



上面看到so文件映射到内存的基地址：74FE4000

cn.wjdiankong.encryptdemo_1.apk	74FAF000	74FE3000
libencrypt.so	74FE4000	74FE6000
libencrypt.so	74FE6000	74FE7000
libencrypt.so	74FE7000	74FE8000
cn.wjdiankong.encryptdemo_1.apk	74FE4000	74FFD000
data@app@cn.wjdiankong.encryptdemo_1.apk...	74FFD000	75300000

那么跳转地址就是：74FE4000+E9C=74FE4E9C

注意：

一般这里的基地址只要程序没有退出，在运行中，那么他的值就不会变，因为程序的数据已经加载到内存中了，基地址不会变的，除非程序退出，又重新运行把数据加载内存中了，同时相对地址是永远不会变的，只有在修改so文件的时候，文件的大小改变了，可能相对地址会改变，其他情况下不会改变，相对地址就是数据在整个so文件中的位置。

```
libencrypt.so:74FE4E9C Java_cn_wjdiankong_encryptdemo_MainActivity_isEquals
libencrypt.so:74FE4E9C PUSH            {R3-R7,LR}
libencrypt.so:74FE4E9E MOV             R1, R2
libencrypt.so:74FE4EA0 LDR             R3, [R0]
libencrypt.so:74FE4EA2 MOV             R7, R2
libencrypt.so:74FE4EA4 MOVS            R2, #0
libencrypt.so:74FE4EA6 MOV             R6, R0
libencrypt.so:74FE4EA8 LDR.W           R3, [R3,#0x2A4]
libencrypt.so:74FE4EAC BLX             R3
libencrypt.so:74FE4EAE MOV             R5, R0
libencrypt.so:74FE4EB0 BLX             unk_74FE4D94
```

这里我们可以看到函数映射到内存中的绝对地址了。

注意：

有时候我们发现跳转到指定位置之后，看到的全是DCB数据，这时候我们选择函数地址，点击P键就可以看到arm指令源码了：

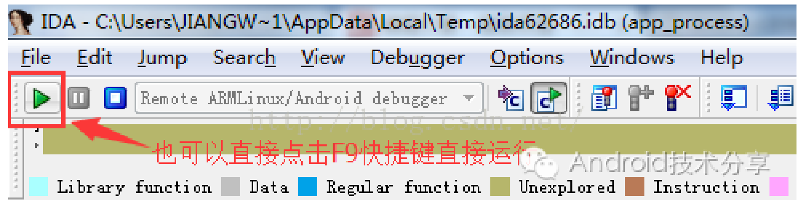
```
libencrypt.so:74FE4E9C Java_cn_wjdiankong_encryptdemo_MainActivity_isEquals DCB 0xF8 ;
libencrypt.so:74FE4E9D DCB 0xB5 ;
libencrypt.so:74FE4E9E DCB 0x11
libencrypt.so:74FE4E9F DCB 0x46 ; F
libencrypt.so:74FE4EA0 DCB    3
libencrypt.so:74FE4EA1 DCB 0x68 ; h
libencrypt.so:74FE4EA2 DCB 0x17
libencrypt.so:74FE4EA3 DCB 0x46 ; F
libencrypt.so:74FE4EA4 DCB    0
libencrypt.so:74FE4EA5 DCB 0x22 ; "
```

这里看到了全是DCB数据，我们需要按下P键进行代码的转化即可

5、调试快捷键：F8单步调试，F7单步进入调试

```
libencrypt.so:74FE4E9C Java_cn_wjdiankong_encryptdemo_MainActivity_isEquals
libencrypt.so:74FE4E9C PUSH            {R3-R7,LR}
libencrypt.so:74FE4E9E MOV             R1, R2
libencrypt.so:74FE4EA0 LDR             R3, [R0]
libencrypt.so:74FE4EA2 MOV             R7, R2
```

上面找到函数地址之后，我们可以下断点了，下断点很简单，点击签名的绿色圈点，变成红色条目即可，然后我们可以点击F9快捷键，或者是点击运行按钮，即可运行程序：



其中还有暂停和结束按钮。我们运行之后，然后在点击so的native函数，触发断点逻辑：

```
    libencrypt.so:74FE4E9C Java_cn_wjdiankong_encryptdemo_MainActivity_isEquals
R12 libencrypt.so:74FE4E9C PUSH            {R3-R7,LR}
    libencrypt.so:74FE4E9E MOV             R1, R2
PC  libencrypt.so:74FE4EA0 LDR             R3, [R0]
    libencrypt.so:74FE4EA2 MOV             R7, R2
    libencrypt.so:74FE4EA4 MOVS            R2, #0
    libencrypt.so:74FE4EA6 MOV             R6, R0
```

这时候，我们看到进入调试界面，点击F8可以单步调试，看到有一个PC指示器，其实在arm中PC是一个特殊的寄存器，用来存储当前指令的地址，这个下面会介绍到。

好了到这里，我们就大致说了一下关于IDA在调试so文件的时候，需要用到的快捷键：

1>、Shift+F12快速查看so文件中包含的字符串信息

2>、F5快捷键可以将arm指令转化成可读的C代码，这里同时可以使用Y键，修改JNIEnv的函数方法名

3>、Ctrl+S有两个用途，在IDA View页面中可以查看so文件的所有段信息，在调试页面可以查看程序所有so文件映射到内存的基地址

4>、G键可以在调试界面，快速跳转到指定的绝对地址，进行下断点调试，这里如果跳转到目的地址之后，发现是DCB数据的话，可以在使用P键，进行转化即可，关于DCB数据，下面会介绍的。

5>、F7键可以单步进入调试，F8键可以单步调试

第二、常用的ARM指令集知识

我们在上面看到IDA打开so之后，看到的是纯种的汇编指令代码，所以这就要求我们必须会看懂汇编代码，就类似于我们在调试Java层代码的时候一样，必须会smali语法，庆幸的是，这两种语法都不是很复杂，所以我们知道一些大体的语法和指令就可以了，下面我们来看看arm指令中的寻址方式，寄存器，常用指令，看完这三个知识点，我们就会对arm指令有一个大体的了解，对于看arm指令代码也是有一个大体的认知了。

1、arm指令中的寻址方式

1>. 立即数寻址
也叫立即寻址，是一种特殊的寻址方式，操作数本身包含在指令中，只要取出指令也就取到了操作数。这个操作数叫做立即数，对应的寻址方式叫做立即寻址。例如：
MOV R0,#64 ；R0 ← 64
2>. 寄存器寻址
寄存器寻址就是利用寄存器中的数值作为操作数，也称为寄存器直接寻址。例如：

ADD R0，R1， R2 ；R0 ← R1 + R2
3>. 寄存器间接寻址
寄存器间接寻址就是把寄存器中的值作为地址，再通过这个地址去取得操作数，操作数本身存放在存储器中。例如：
LDR R0，[R1]；R0 ←[R1]
4>. 寄存器偏移寻址
这是ARM指令集特有的寻址方式，它是在寄存器寻址得到操作数后再进行移位操作，得到最终的操作数。例如：
MOV R0，R2，LSL #3 ；R0 ← R2 * 8，R2的值左移3位，结果赋给R0。
5>. 寄存器基址变址寻址
寄存器基址变址寻址又称为基址变址寻址，它是在寄存器间接寻址的基础上扩展来的。它将寄存器（该寄存器一般称作基址寄存器）中的值与指令中给出的地址偏移量相加，从而得到一个地址，通过这个地址取得操作数。例如：
LDR R0，[R1，#4]；R0 ←[R1 + 4]，将R1的内容加上4形成操作数的地址，取得的操作数存入寄存器R0中。
6>. 多寄存器寻址
这种寻址方式可以一次完成多个寄存器值的传送。例如：
LDMIA R0，{R1，R2，R3，R4}；R1←[R0]，R2←[R0+4]，R3←[R0+8]，R4←[R0+12]
7>. 堆栈寻址
堆栈是一种数据结构，按先进后出（First In Last Out，FILO）的方式工作，使用堆栈指针（Stack Pointer, SP）指示当前的操作位置，堆栈指针总是指向栈顶。
堆栈寻址举例如下：
STMFD SP！，｛R1－R7, LR｝；将R1－R7, LR压入堆栈。满递减堆栈。
LDMED SP！，｛R1－R7, LR｝；将堆栈中的数据取回到R1－R7, LR寄存器。空递减堆栈。

2、ARM中的寄存器

R0-R3:用于函数参数及返回值的传递
R4-R6, R8, R10-R11:没有特殊规定，就是普通的通用寄存器
R7:栈帧指针(Frame Pointer).指向前一个保存的栈帧(stack frame)和链接寄存器(link register， lr)在栈上的地址。
R9:操作系统保留
R12:又叫IP(intra-procedure scratch)
R13:又叫SP(stack pointer)，是栈顶指针
R14:又叫LR(link register)，存放函数的返回地址。
R15:又叫PC(program counter)，指向当前指令地址。

3、ARM中的常用指令含义

ADD 加指令
SUB 减指令
STR　把寄存器内容存到栈上去
LDR　把栈上内容载入一寄存器中
.W　是一个可选的指令宽度说明符。它不会影响为此指令的行为，它只是确保生成 32 位指令。
Infocenter.arm.com的详细信息
BL　执行函数调用，并把使lr指向调用者(caller)的下一条指令，即函数的返回地址
BLX　同上，但是在ARM和thumb指令集间切换。
CMP　指令进行比较两个操作数的大小

4、ARM指令简单代码段分析

C代码：

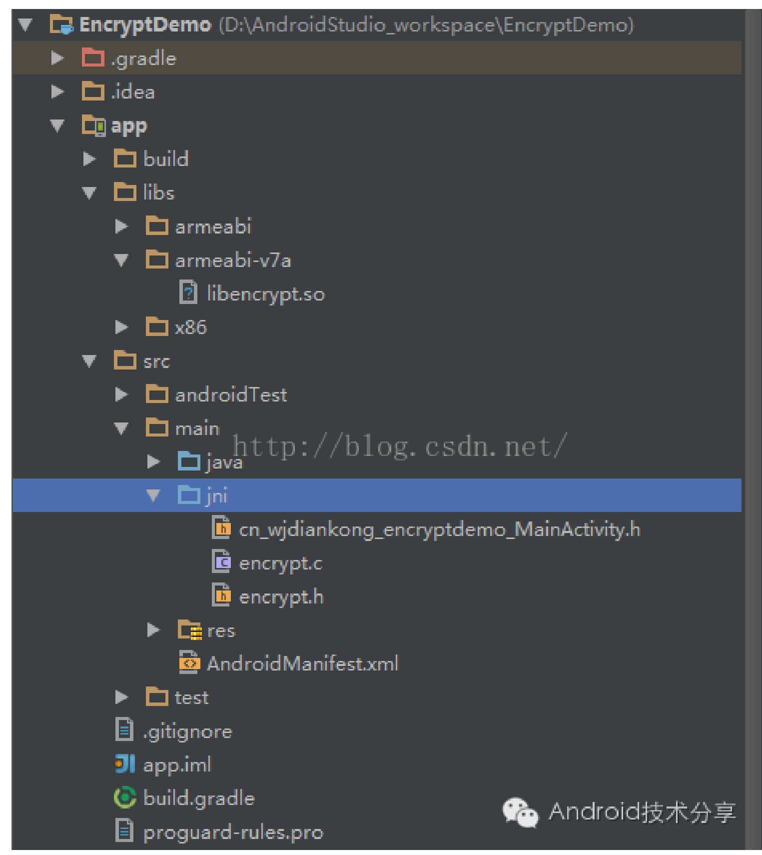
```
#include <stdio.h>
int func(int a, int b, int c, int d, int e, int f)
{
   int g = a + b + c + d + e + f;
   return g;
}
```

对应的ARM指令：

```
add r0, r1  将参数a和参数b相加再把结果赋值给r0
ldr.w r12, [sp]  把最的一个参数f从栈上装载到r12寄存器
add r0, r2  把参数c累加到r0上
ldr.w r9, [sp, #4]  把参数e从栈上装载到r9寄存器
add r0, r3  累加d累加到r0
add r0, r12  累加参数f到r0
add r0, r9  累加参数e到r0
```

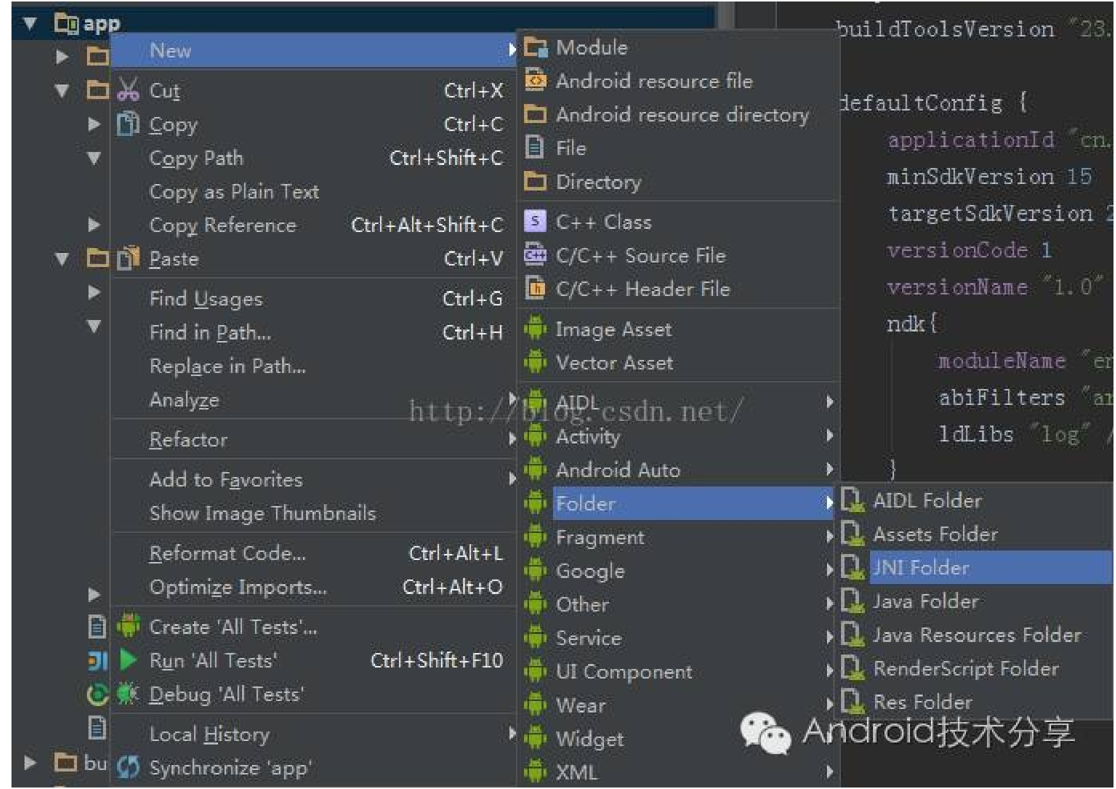
三、构造so案例

好了，关于ARM指令的相关知识，就介绍这么多了，不过我们在调试分析的时候，肯定不能做到全部的了解，因为本身ARM指令语法就比较复杂，不过幸好大学学习了汇编语言，所以稍微能看懂点，如果不懂汇编的同学那就可能需要补习一下了，因为我们在使用IDA分析so文件的时候，不会汇编的话，那是肯定行不通的，所以我们必须要看懂汇编代码的，如果遇到特殊指令不了解的同学，可以网上搜一下即可。上面我们的准备知识做完了，一个是IDA工具的时候，一个是ARM指令的了解，下面我们就来开始操刀了，为了方便开始，我们先自己写一个简单的Android native层代码，然后进行IDA进行分析即可。这里可以使用AndroidStudio中进行新建一个简单工程，然后创建JNI即可：



这里顺便简单说一下AndroidStudio中如何进行NDK的开发吧：

第一步：在工程中新建jni目录



第二步：**使用javah生成native的头文件**

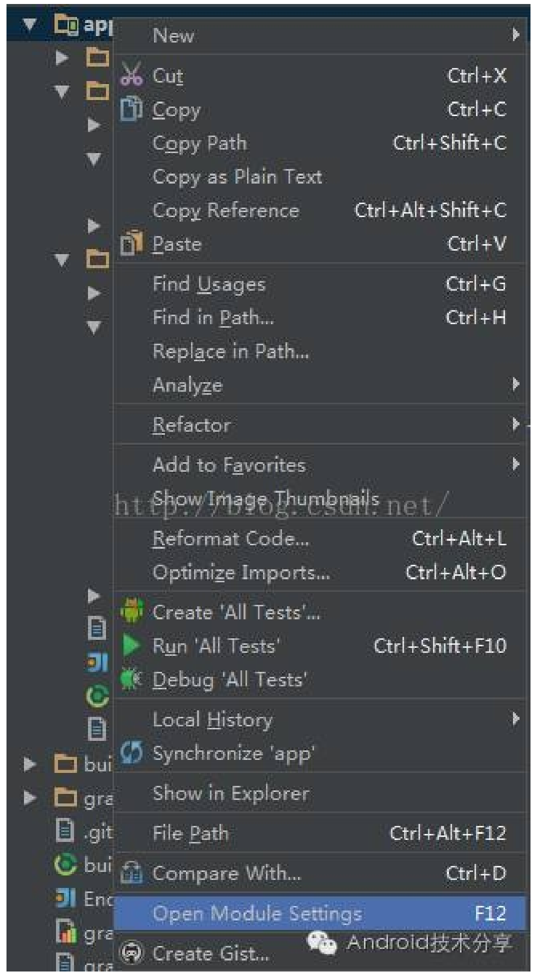
注意：

javah执行的目录，必须是类包名路径的最上层，然后执行：

javah 类全名

注意没有后缀名java哦

第三步：**配置项目的NDK目录**



选择模块的设置选线：**Open Module Settings**：

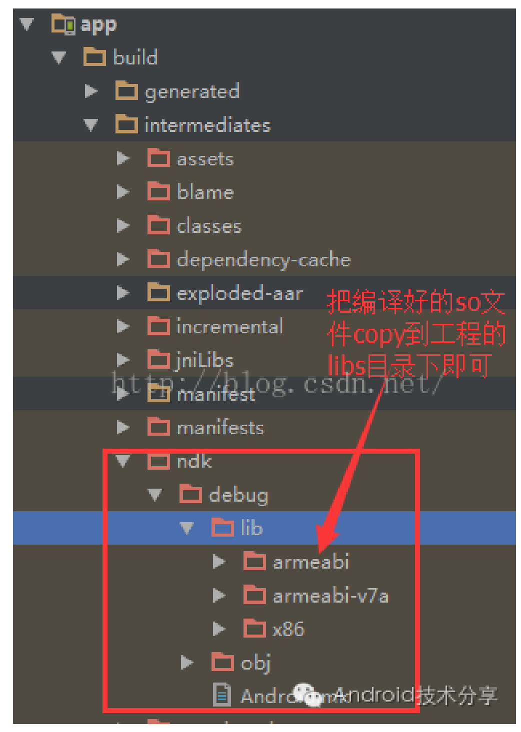
设置NDK目录即可

第四步：copy头文件到jni目录下，然后配置gradle中的ndk选项

```
defaultConfig {
    applicationId "cn.wjdiankong.encryptdemo"
    minSdkVersion 15
    targetSdkVersion 23
    versionCode 1
    versionName "1.0"
    ndk{
        moduleName "encrypt"  //设置库(so)文件名称
        abiFilters "armeabi", "armeabi-v7a", "x86"   //输出指定三种abi体系结构下的so库，如果没有这句的话，lib下面就没有so文件
        ldLibs "log" //增加log的lib库
    }
}
```

这里只需要设置编译之后的模块名，就是so文件的名称，需要产生那几个平台下的so文件，还有就是需要用到的lib库，这里我们看到我们用到了Android中打印log的库文件。

第五步：编译运行，在build目录下生成指定的so文件，copy到工程的libs目录下即可



好了，到这里我们就快速的在AndroidStudio中新建了一个Native项目，这里关于native项目的代码不想解释太多，就是Java层

传递了用户输入的密码，然后native做了校验过程，把校验结果返回到Java层即可：

```java
public class MainActivity extends Activity {

    static {
        System.loadLibrary("encrypt");
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        findViewById(R.id.btn).setOnClickListener((v) → {
                boolean res = isEquals("123456");
                Log.i("jw", "res:" + res);
        });
    }

    private native boolean isEquals(String str);
}
```

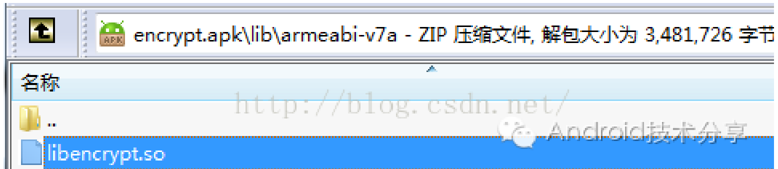
```c
JNIEXPORT jboolean JNICALL Java_cn_wjdiankong_encryptdemo_MainActivity_isEquals
        (JNIEnv * env, jobject obj, jstring str)
{
    const char *strAry = (*env)->GetStringUTFChars(env, str, 0);
    int len = strlen(strAry);
    char* dest = (char*)malloc(len);
    strcpy(dest, strAry);

    int number = is_number(strAry);
    if(number == 0){
        return 0;
    }

    char* encry_str = get_encrypt_str(strAry);
    const char* pas = "ssBCqpBssP";
    int result = strcmp(pas, encry_str);

    (*env)->ReleaseStringUTFChars(env, str, strAry);

    if(result == 0){
        return 1;
    }else{
        return 0;
    }
}
```

具体的校验过程这里不再解释了。我们运行项目之后，得到apk文件，那么下面我们就开始我们的破解旅程了

四、开始破解so文件

开始破解我们编译之后的apk文件

第一、首先我们可以使用最简单的压缩软件，打开apk文件，然后解压出他的so文件

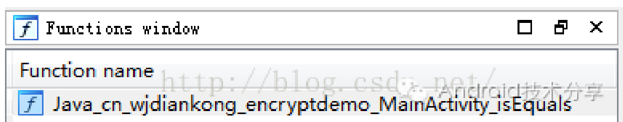


我们得到libencrypt.so文件之后，使用IDA打开它：

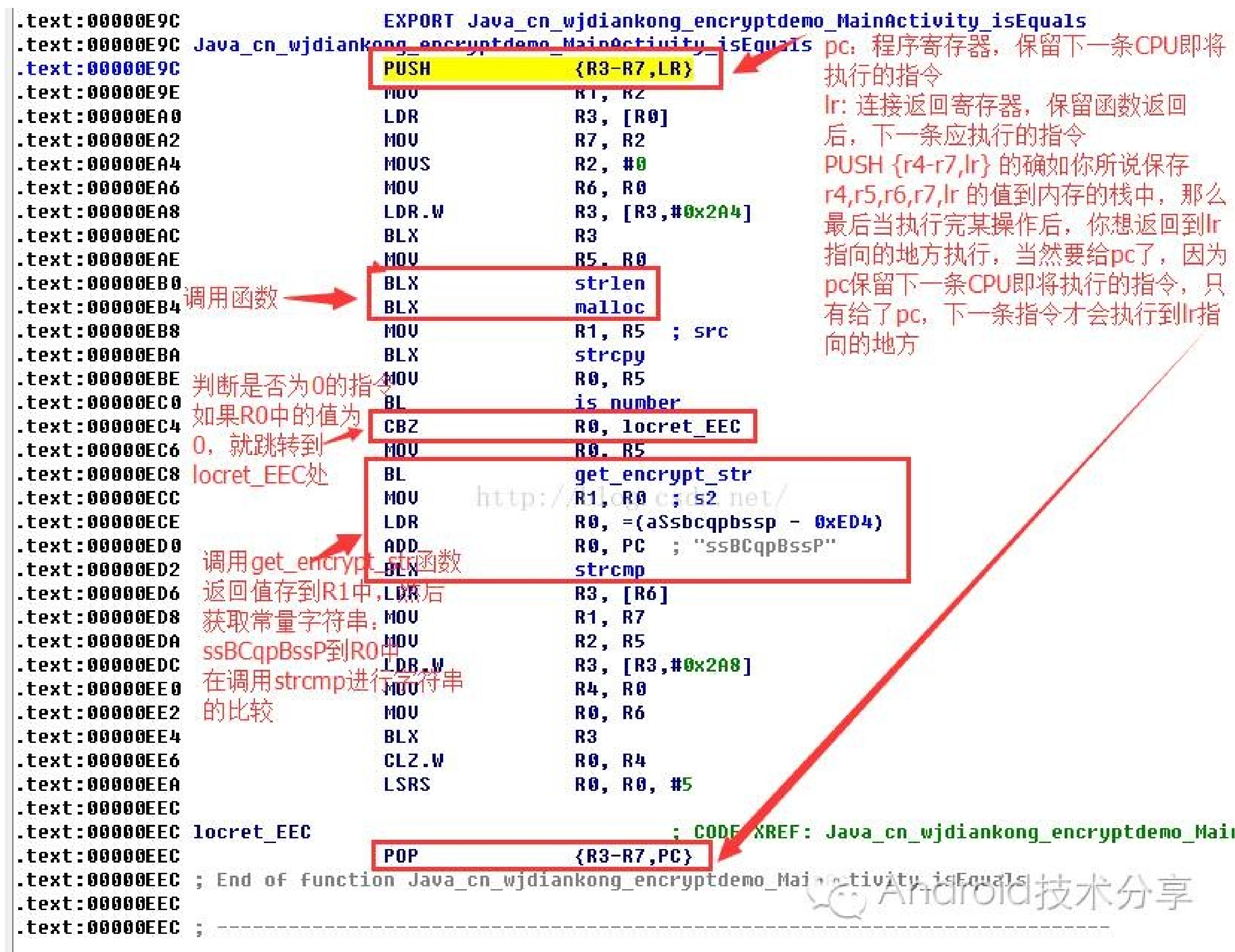
```
File  Edit  Jump  Search  View  Debugger  Options  Windows  Help
Library function  Data  Regular function  Unexplored  Instruction  External symbol

Functions window                         IDA View-A        Hex View-1        Structures
Function name                  so的指令View窗口
_cxa_atexit                    so中的函数窗口
_cxa_finalize                  这里我们可以使用
strlen                         ctrl+F来进行函数的
malloc                         搜索
strcpy
strcmp
_gnu_Unwind_Find_exidx
abort
memcpy
_cxa_begin_cleanup
_cxa_type_match
sub_E30
is_number
get_encrypt_str
Java_cn_wjdiankong_encryptdemo_MainActivity_isEquals
sub_F28
sub_F3E
sub_FAC
sub_FDC
sub_1080
sub_10CA
sub_10D8
nullsub_1
sub_10DE
sub_1118
_Unwind_GetCFA
_gnu_Unwind_RaiseException
_gnu_Unwind_ForcedUnwind
_gnu_Unwind_Resume
_gnu_Unwind_Resume_or_Rethrow
_Unwind_Complete
_Unwind_DeleteException
_Unwind_VRS_Get
sub_12DE
_Unwind_VRS_Set
sub_1328
_gnu_Unwind_Backtrace
sub_13B0
_aeabi_unwind_cpp_pr0
_aeabi_unwind_cpp_pr1
_aeabi_unwind_cpp_pr2
_Unwind_VRS_Pop
restore_core_regs
_gnu_Unwind_Restore_VFP
_gnu_Unwind_Save_VFP
Line 15 of 119

.plt:00000D98 ;
.plt:00000D98 ; +-------------------------------------------------------------
.plt:00000D98 ; |   This file has been generated by The Interactive Disassemb
.plt:00000D98 ; |          Copyright (c) 2015 Hex-Rays, <support@hex-rays.co
.plt:00000D98 ; |                License info: 48-B611-7234-BB
.plt:00000D98 ; |         Doskey Lee, Kingsoft Internet Security Software
.plt:00000D98 ; +-------------------------------------------------------------
.plt:00000D98 ;
.plt:00000D98 ; Input MD5   : 77A61D05F72EFA5F4AD098337CC215B5
.plt:00000D98 ; Input CRC32 : 70913C9C
.plt:00000D98
.plt:00000D98 ; -------------------------------------------------------------
.plt:00000D98 ; File Name   : C:\Users\jiangwei1-g\Desktop\Android中的动态调试
.plt:00000D98 ; Format      : ELF for ARM (Shared object)
.plt:00000D98 ; Interpreter '/system/bin/linker'
.plt:00000D98 ; Needed Library 'liblog.so'
.plt:00000D98 ; Needed Library 'libstdc++.so'
.plt:00000D98 ; Needed Library 'libm.so'
.plt:00000D98 ; Needed Library 'libc.so'
.plt:00000D98 ; Needed Library 'libdl.so'
.plt:00000D98 ; Shared Name 'libencrypt.so'
.plt:00000D98 ;
.plt:00000D98 ; Options     : EF_ARM_SOFT_FLOAT
.plt:00000D98 ; EABI version: 5
.plt:00000D98 ;
.plt:00000D98
.plt:00000D98 ; Processor         : ARM
.plt:00000D98 ; ARM architecture: ARMv5TE
.plt:00000D98 ; Target assembler: Generic assembler for ARM
.plt:00000D98 ; Byte sex          : Little endian
.plt:00000D98
.plt:00000D98 ; =============================================================
.plt:00000D98
.plt:00000D98 ; Segment type: Pure code
.plt:00000D98                 AREA .plt, CODE
.plt:00000D98                 ; ORG 0xD98
.plt:00000D98                 CODE32
.plt:00000D98                 STR     LR, [SP,#-4]!
.plt:00000D9C                 LDR     LR, =(_GLOBAL_OFFSET_TABLE_ - 0xDA8)
.plt:00000DA0                 ADD     LR, PC, LR ; _GLOBAL_OFFSET_TABLE_
.plt:00000DA4                 LDR     PC, [LR,#8]!
.plt:00000DA4 ; -------------------------------------------------------------
.plt:00000DA8 off_DA8         DCD _GLOBAL_OFFSET_TABLE_ - 0xDA8 ; DATA XREF:
.plt:00000DAC ; [0000000C BYTES: COLLAPSED FUNCTION __cxa_atexit. PRESS CTRL-
.plt:00000DB8 ; [0000000C BYTES: COLLAPSED FUNCTION __cxa_finalize. PRESS CTR
.plt:00000DC4 ; [0000000C BYTES: COLLAPSED FUNCTION strlen. PRESS CTRL-NUMPAD
.plt:00000DD0 ; [0000000C BYTES: COLLAPSED FUNCTION malloc. PRESS CTRL-NUMPAD
00000D98 00000D98: .plt:00000D98 (Synchronized with Hex View-1)
```

我们知道一般so中的函数方法名都是：**Java_类名_方法名**

那么这里我们直接搜：**Java关键字即可，或者使用jd-gui工具找到指定的native方法**



双击，即可在右边的IDA View页面中看到Java_cn_wjdiankong_encryptdemo_MainActivity_isEquals 函数的指令代码：



我们可以简单的分析一下这段指令代码：

1>、PUSH {r3-r7,lr} 是保存r3,r4,r5,r6,r7,lr 的值到内存的栈中，那么最后当执行完某操作后，你想返回到lr指向的地方执行，当然要给pc了，因为pc保留下一条CPU即将执行的指令，只有给了pc，下一条指令才会执行到lr指向的地方

pc：程序寄存器，保留下一条CPU即将执行的指令
lr: 连接返回寄存器，保留函数返回后，下一条应执行的指令

这个和函数最后面的POP {r3-r7,pc}是相对应的。

2>、然后是调用了strlen,malloc,strcpy等系统函数，在每次使用BLX和BL指令调用这些函数的时候，我们都发现了一个规律：就是在调用他们之前一般都是由MOV指令，用来传递参数值的，比如这里的R5里面存储的就是strlen函数的参数，R0就是is_number函数的参数，所以我们这样分析之后，在后面的动态调试的过程中可以得到函数的入口参数值，这样就能得到一些重要信息

```
MOV     R5, R0
BLX     strlen
BLX     malloc
MOV     R1, R5  ; src
BLX     strcpy
MOV     R0, R5
BL      is_number
CBZ     R0, locret_EEC
MOV     R0, R5
BL      get_encrypt_str
MOV     R1, R0  ; s2
LDR     R0, =(aSsbcqpbssp - 0xED4)
ADD     R0, PC  ; "ssBCqpBssP"
BLX     strcmp
```

这里我们可以看到，一般在使用BLX或者是BL等指令调用函数的时候，他们之前一般都是由MOV指令，用来传递参数值的，比如这里的R5里面存储的就是strlen函数的参数，R0就是is_number函数的参数，所以我们这样分析之后，在后面的动态调试的过程中可以得到函数的入口参数值，这样就能得到一些重要信息

Android技术分享

3>、在每次调用有返回值的函数之后的命令，一般都是比较指令，比如CMP，CBZ，或者是strcmp等，这里是我们破解的突破点，因为一般加密再怎么牛逼，最后比较的参数肯定是正确的密码(或者是正确的加密之后的密码)和我们输入的密码(或者是加密之后的输入密码)，我们在这里就可以得到正确密码，或者是加密之后的密码：

```
MOV             R5, R0
BLX             strlen
BLX             malloc
MOV             R1, R5  ; src
BLX             strcpy
MOV             R0, R5
BL              is_number
CBZ             R0, locret_EEC
MOV             R0, R5
BL              get_encrypt_str
MOV             R1, R0  ; s2
LDR             R0, =(aSsbcqpbssp - 0xED4)
ADD             R0, PC  ; "ssBCqpBssP"
BLX             strcmp
```

这里我们可以看到，一般在使用BLX或者是BL等指令调用函数的时候，他们之前一般都是由MOV指令，用来传递参数值的，比如这里的R5里面存储的就是strlen函数的参数，R0就是is_number函数的参数，所以我们这样分析之后，在后面的动态调试的过程中可以得到函数的入口参数值，这样就能得到一些重要信息

Android技术分享

到这里，我们就分析完了native层的密码比较函数：Java_cn_wjdiankong_encryptdemo_MainActivity_isEquals

如果觉得上面的ARM指令看的吃力，可以使用F5键，查看他的C语言代码：

```
unsigned int __fastcall Java_cn_wjdiankong_encryptdemo_MainActivity_isEquals(JNIEnv *a1, int a2, i
{
  int v3; // r7@1
  JNIEnv *v4; // r6@1
  const char *v5; // r0@1
  const char *v6; // r5@1
  size_t v7; // r0@1
  char *v8; // r0@1
  unsigned int result; // r0@1
  const char *v10; // r0@2
  int v11; // r4@2

  v3 = a3;
  v4 = a1;
  v5 = (const char *)((int (*)(void))(*a1)->GetStringUTFChars)();
  v6 = v5;
  v7 = strlen(v5);
  v8 = (char *)malloc(v7);
  strcpy(v8, v6);
  result = is_number(v6);
  if ( result )
  {
    v10 = (const char *)get_encrypt_str(v6);
    v11 = strcmp("ssBCqpBssP", v10);
    ((void (__fastcall *)(JNIEnv *, int, const char *))(*v4)->ReleaseStringUTFChars)(v4, v3, v6);
    result = __clz(v11) >> 5;
  }
  return result;
}
```

http://blog.csdn.net/
Android技术分享

我们这里看到其实有两个函数是核心点：

1>is_number函数，这个函数我们看名字应该猜到是判断是不是数字，我们可以使用F5键，查看他对应的C语言代码：

```
signed int __fastcall is_number(signed int result)
{
  int v1; // r0@2
  int v2; // r3@3
  int v3; // t1@3

  if ( result )
  {
    v1 = result - 1;
    while ( 1 )
    {
      v3 = *(_BYTE *)(v1++ + 1);
      v2 = v3;
      if ( !v3 )
        break;
      if ( (unsigned int)(v2 - 48) > 9 )
        return 0;
    }
    result = 1;
  }
  return result;
}
```

http://blog.csdn.net/
Android技术分享

这里简单一看，主要是看return语句和if判断语句，看到这里有一个循环，然后获取_BYTE*这里地址的值，并且自增加一，然后存到v2中，如果v3为'\0'的话，就结束循环，然后做一次判断，就是v2-48是否大于9，那么这里我们知道48对应的是ASCII中的数字0，所以这里可以确定的是就是：用一个循环遍历_BYTE*这里存的字符串是否为数字串。

2>get_encrypt_str函数，这个函数我们看到名字可以猜测，他是获取我们输入的密码加密之后的值，再次使用F5快捷键查看：

```
1 const char *__fastcall get_encrypt_str(const char *result)
2 {
3   const char *v1; // r4@1
4   size_t v2; // r0@2
5   int v3; // r4@2
6   int v4; // r5@2
7   const char *i; // r2@2
8   int v6; // t1@4
9   int v7; // r3@4
10
11   v1 = result;
12   if ( result )
13   {
14     v2 = strlen(result);
15     v3 = (int)(v1 - 1);
16     v4 = v2 + 1;
17     result = (const char *)malloc(v2 + 1);
18     for ( i = result; i - result < v4; ++i )
19     {
20       v6 = *(_BYTE *)(v3++ + 1);
21       v7 = v6 - 48;
22       if ( v6 == 48 )
23         v7 = 1;
24       *i = key_src[18 - v7];
25     }
26     *((_BYTE *)i + 1) = 0;
27   }
28   return result;
29 }
```

http://blog.csdn.net/
Android技术分享

这里我们看到，首先是一个if语句，用来判断传递的参数是否为NULL，如果是的话，直接返回，不是的话，使用strlen函数获取字符串的长度保存到v2中，然后使用malloc申请一块堆内存，首指针保存到result，大小是v2+1也就是传递进来的字符串长度+1，然后就开始进入循环，首指针result，赋值给i指针，开始循环，v3是通过v1-1获取到的，就是函数传递进来字符串的地址，那么v6就是获取传递进来字符串的字符值，然后减去48，赋值给v7，这里我们可以猜到了，这里想做字符转化，把char转化成int类型，继续往下看，如果v6==48的话，v7=1，也就是说这里如果遇到字符'0'，就赋值1，在往下看，看到我们上面得到的v7值，被用来取key_src数组中的值，那么这里我们双击key_src变量，就跳转到了他的值地方，果不其然，这里保存了一个字符数组，看到他的长度正好是18，那么这里我们应该明白了，这里通过传递进来的字符串，循环遍历字符串，获取字符，然后转化成数字，在倒序获取key_src中的字符，保存到result中。然后返回。

```
.data:00003004                  EXPORT key_src
.data:00003004 key_src          DCB "zytyrTRA*BniqCPpVs",0 ; DATA XREF: get_encrypt_str+26↑o
.data:00003004                                            ; .got:key_src_ptr↑o
.data:00003017                  ALIGN 4
```

好了，到这里我们就分析完了这两个重要的函数的功能，一个是判断输入的内容是否为数字字符串，一个是通过输入的内容获取密码内容，然后和正确的加密密码：ssBCqpBssP 作比较。

第二、开始使用IDA进行调试设置

那么下面我们就用动态调试来跟踪传入的字符串值，和加密之后的值，这里我们看到没有打印log的函数，所以很难知道具体的参数和寄存器的值，所以这里需要开始调试，得知每个函数执行之后的寄存器的值，我们在用IDA进行调试so的时候，需要以下准备步骤：

1、在IDA安装目录下获取android_server命令文件

本地磁盘 (C:) ▸ Program Files (x86) ▸ IDA 6.8 ▸ dbgsrv

共享 ▾ 新建文件夹

名称	修改日期
android_server	2015/4/13 18:35
android_server_nonpie	2015/4/13 18:35
armlinux_server	2015/4/13 18:35
armuclinux_server	2015/4/13 18:35
ida_kdstub.dll	2015/4/13 18:35
linux_server	2015/4/13 18:00
linux_serverx64	2015/4/13 18:00
mac_server	2015/4/13 18:35
mac_serverx64	2015/4/13 18:35
win32_remote.exe	2015/4/13 18:35
win64_remotex64.exe	2015/4/13 18:35
wince_remote_arm.dll	2015/4/13 18:35
wince_remote_tcp_arm.exe	2015/4/13 18:35

在IDA安装目录\dbgsrv\android_server

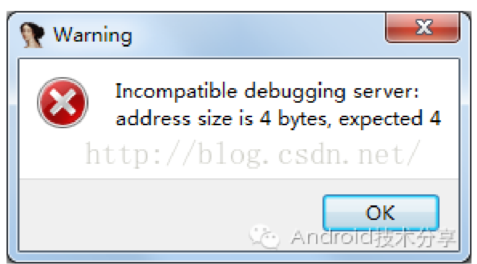
这个文件是干嘛的呢？他怎么运行呢？下面来介绍一下：Android中的调试原理，其实是使用gdb和gdbserver来做到的，gdb和gdbserver在调试的时候，必须注入到被调试的程序进程中，但是非root设备的话，注入别的进程中只能借助于run-as这个命令了，所以我们知道，如果要调试一个应用进程的话，必须要注入他内部，那么IDA调试so也是这个原理，他需要注入(Attach附加)进程，才能进行调试，但是IDA没有自己弄了一个类似于gdbserver这样的工具，那就是android_server了，所以他需要运行在设备中，保证和PC端的IDA进行通信，比如获取设备的进程信息，具体进程的so内存地址，调试信息等。

所以我们把android_server保存到设备的/data目录下，修改一下他的运行权限，然后必须在root环境下运行，因为他要做注入进程操作，必须要root。

```
C:\Users\jiangwei1-g>adb shell
shell@pisces:/ $ su
root@pisces:/ # cd /data
root@pisces:/data # ./android_server
IDA Android 32-bit remote debug server(ST) v1.19. Hex-Rays (c) 2004-2015
Listening on port #23946...
```

注意：

这里把他放在了/data目录下，然后运行./android_server，这里提示了IDA Android 32-bit，所以后面我们在打开IDA的时候一定要是32位的IDA，不是64位的，不然保存，IDA在安装之后都是有两个可执行的程序，一个是32位，一个是64位的，如果没打开正确会报这样的错误：



同样还有一类问题：

error: only position independent executables (PIE) are supported

这个主要是Android5.0以上的编译选项默认开启了pie，在5.0以下编译的原生应用不能运行，有两种解决办法，一种是用Android5.0以下的手机进行操作，还有一种就是用IDA6.6+版本即可。

然后我们再看，这里开始监听了设备的23946端口，那么如果要想让IDA和这个android_server进行通信，那么必须让PC端的IDA也连上这个端口，那么这时候就需要借助于adb的一个命令了：

adb forward tcp:远端设备端口号(进行调试程序端) tcp:本地设备端口(被调试程序端)

那么这里，我们就可以把android_server端口转发出去：

```
C:\Users\jiangwei1-g>adb forward tcp:23946 tcp:23946
```

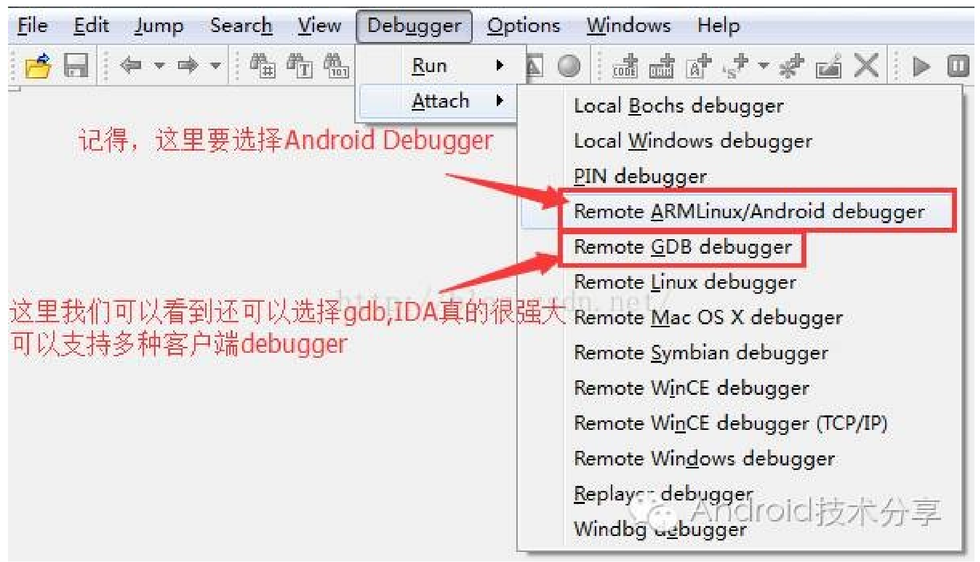
然后这时候，我们只要在PC端使用IDA连接上23946这个端口就可以了，这里面有人好奇了，为什么远程端的端口号也是23946，因为后面我们在使用IDA进行连接的时候，发现IDA他把这个端口设置死了，就是23946，所以我们没办法自定义这个端口了。

我们可以使用netstat命令查看端口23946的使用情况，看到是ida在使用这个端口

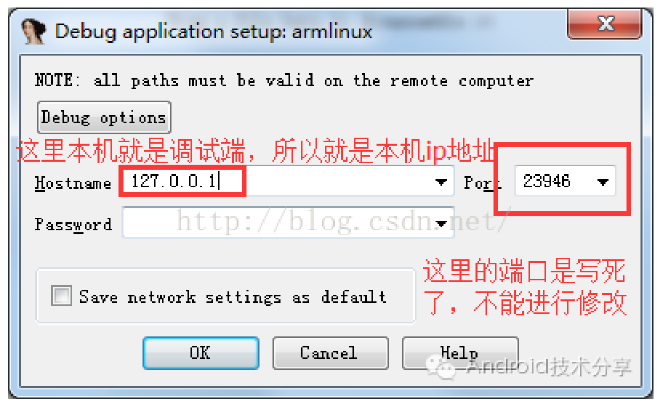
2、上面就准备好了android_server，运行成功，下面就来用IDA进行尝试连接，获取信息，进行进程附加注入

我们这时候需要在打开一个IDA，之前打开一个IDA是用来分析so文件的，一般用于静态分析，我们要调试so的话，需要在打开一个IDA来进行，所以这里一般都是需要打开两个IDA，也叫作双开IDA操作。动静结合策略。

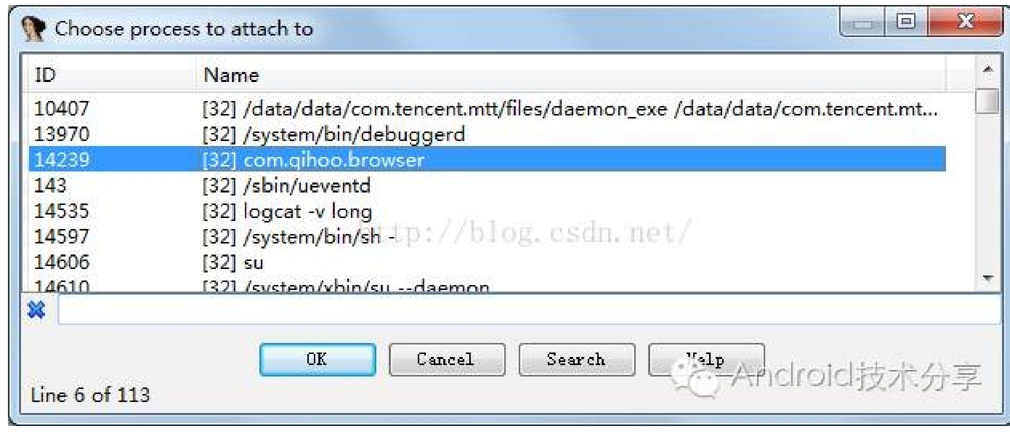
这里记得选择go这个选项，就是不需要打开so文件了，进入是一个空白页：



我们选择Debugger选项，选择Attach，看到有很多debugger，所以说IDA工具真的很强大，做到很多debugger的兼容，可以调试很多平台下的程序。这里我们选择Android debugger：



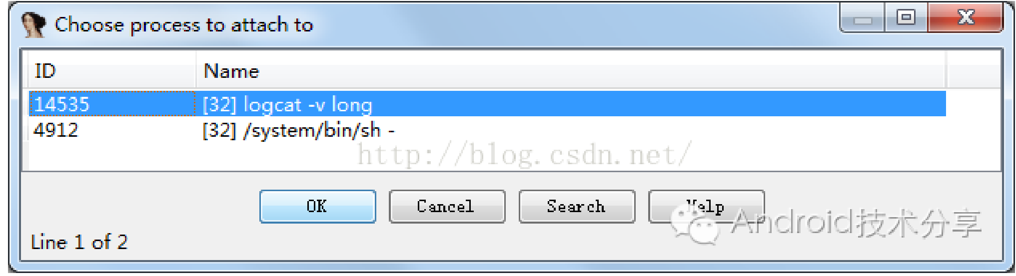
这里看到，端口是写死的：23946，不能进行修改，所以上面的adb forward进行端口转发的时候必须是23946。这里PC本地机就是调试端，所以host就是本机的ip地址：127.0.0.1，点击确定：



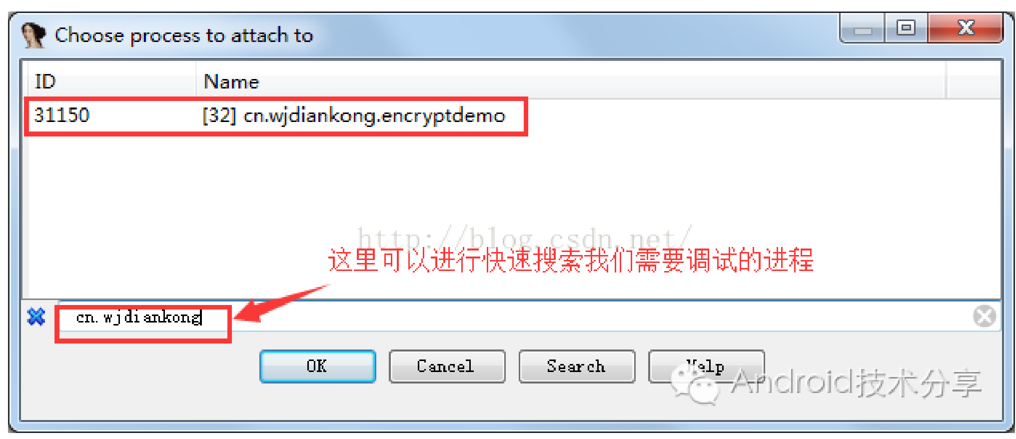
这里可以看到设备中所有的进程信息就列举出来的，其实都是android_server干的事，获取设备进程信息传递给IDA进行展示。

注意：

如果我们当初没有用root身份去运行android_server:



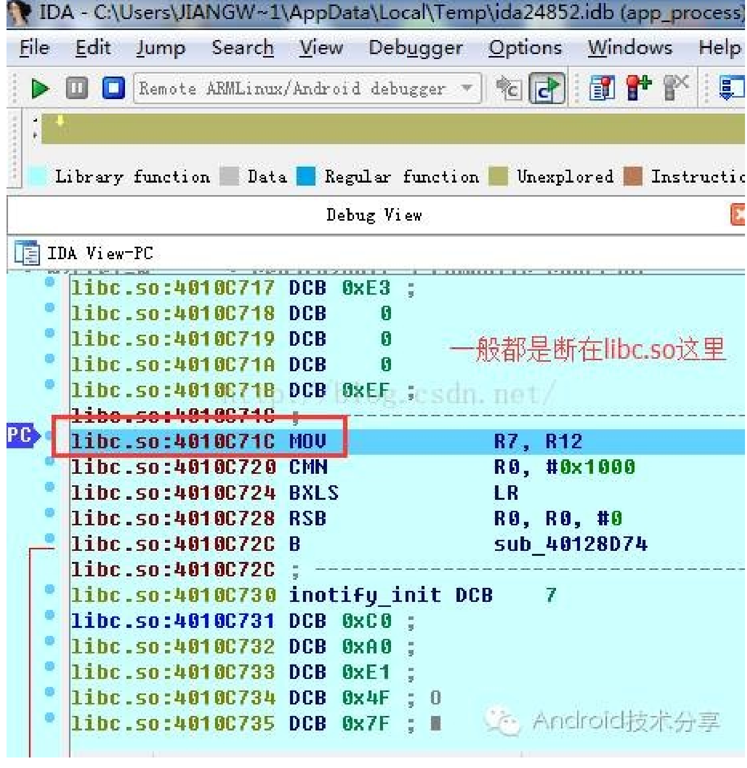
这里就会IDA是不会列举出设备的进程信息：



还有一个注意的地方，就是IDA和android_server一定要保持一致。

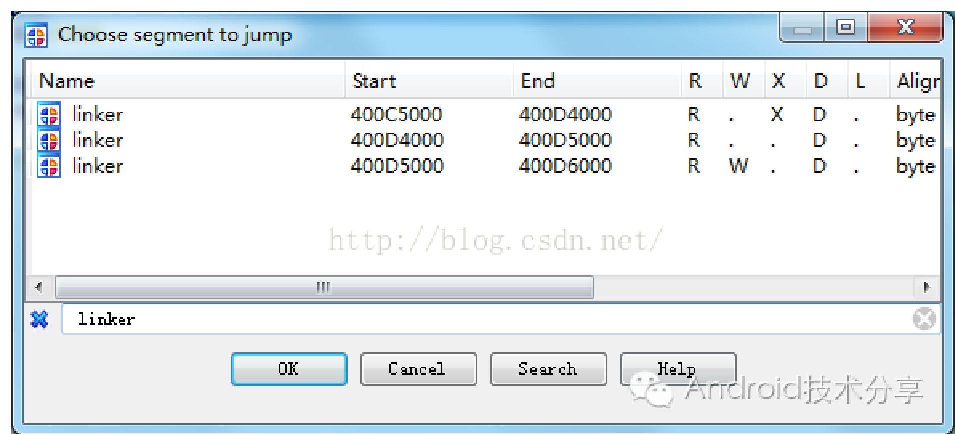
我们这里可以ctrl+F搜索我们需要调试的进程，当然这里我们必须运行起来我们需要调试的进程，不然也是找不到这个进程的

双击进程，即可进入调试页面：



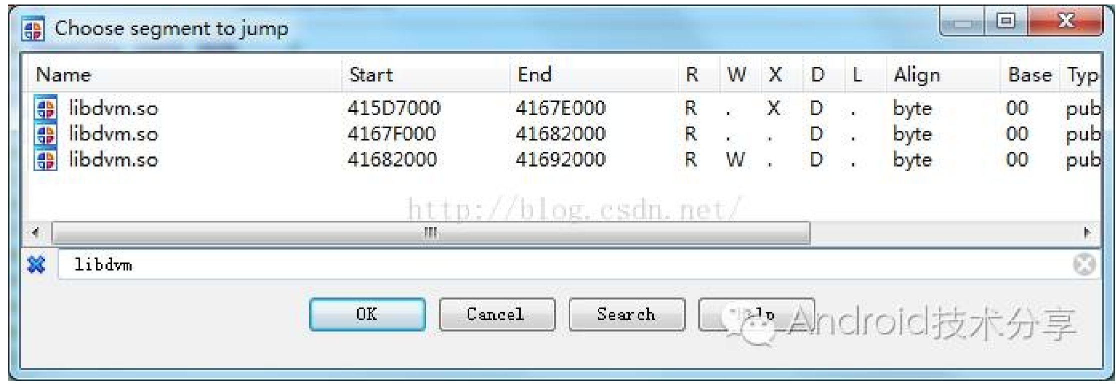
这里为什么会断在libc.so中呢？

android系统中libc是c层中最基本的函数库，libc中封装了io、文件、socket等基本系统调用。所有上层的调用都需要经过libc封装层。所以libc.so是最基本的，所以会断在这里，而且我们还需要知道一些常用的系统so,比如linker：



我们知道，这个linker是用于加载so文件的模块，所以后面我们在分析如何在.init_array处下断点

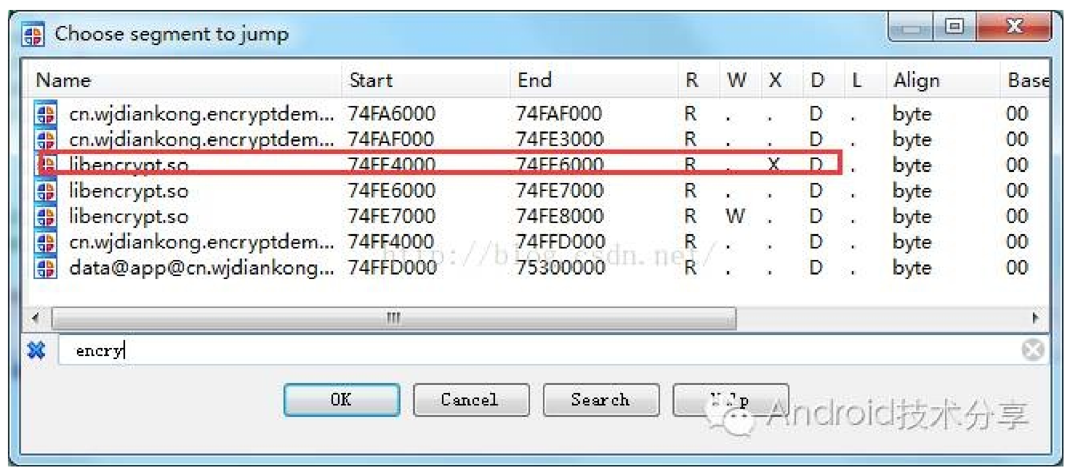
还有一个就是libdvm.so文件，他包含了DVM中所有的底层加载dex的一些方法：



我们在后面动态调试需要dump出加密之后的dex文件，就需要调试这个so文件了。

3、找到函数地址，下断点，开始调试

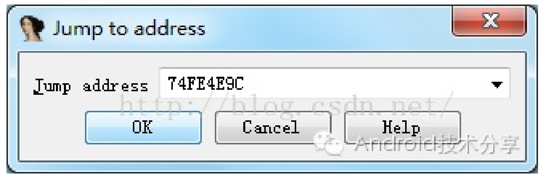
我们使用Ctrl+S找到需要调试so的基地址：74FE4000



然后通过另外一个IDA打开so文件，查看函数的相对地址：E9C

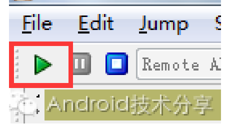
```
.text:00000E9C
.text:00000E9C                 EXPORT Java_cn_wjdiankong_encryptdemo_MainActivity_isEquals
.text:00000E9C Java_cn_wjdiankong_encryptdemo_MainActivity_isEquals
.text:00000E9C                 PUSH            {R3-R7,LR}
.text:00000E9E                 MOV             R1, R2
.text:00000EA0                 LDR             R3, [R0]
.text:00000EA2                 MOV             R7, R2
.text:00000EA4                 MOVS            R2, #0
.text:00000EA6                 MOV             R6, R0
```

那么得到了函数的绝对地址就是：74FE4E9C，使用G键快速跳转到这个绝对地址：



跳转到指定地址之后，开始下断点，点击最左边的绿色圆点即可下断点：

然后点击左上角的绿色按钮，运行，也可以使用F9键运行程序：



我们点击程序中的按钮：

触发native函数的运行：

```
libencrypt.so:74FE4E9C Java_cn_wjdiankong_encryptdemo_MainActivity_isEquals
libencrypt.so:74FE4E9C PUSH                    {R3-R7,LR}
libencrypt.so:74FE4E9E MOV                     R1, R2
libencrypt.so:74FE4EA0 LDR                     R3, [R0]
libencrypt.so:74FE4EA2 MOV                     R7, R2
```

看到了，进入调试阶段了，这时候，我们可以使用F8进行单步调试，F7进行单步进入调试：

```
libencrypt.so:74FE4EBA BLX                     unk_74FE4DAC
libencrypt.so:74FE4EBE MOV                     R0, R5
libencrypt.so:74FE4EC0 BL                      is_number
libencrypt.so:74FE4EC4 CBZ                     R0,  R0=debug127:75759FD0
libencrypt.so:74FE4EC6 MOV                     R0, DCB 0x31 ; 1
libencrypt.so:74FE4EC8 BL                      get DCB 0x32 ; 2
libencrypt.so:74FE4ECC MOV                     R1, DCB 0x33 ; 3
libencrypt.so:74FE4ECE LDR                     R0, DCB 0x34 ; 4
libencrypt.so:74FE4ED0 ADD                     R0, DCB 0x35 ; 5
libencrypt.so:74FE4ED2 BLX                     unk DCB 0x36 ; 6
libencrypt.so:74FE4ED6 LDR                     R3, DCB    0
libencrypt.so:74FE4ED8 MOV                     R1, DCB 0x40 ; @
libencrypt.so:74FE4EDA MOV                     R2, DCB 0x10
libencrypt.so:74FE4EDC LDR.W                   R3, DCB    0
```

刚刚静态分析的时候知道了R0就是函数is_number的入口参数，这里我们查看R0寄存器的值是：123456，就是Java层输入的密码

我们点击F8进行单步调试，达到is_number函数调用出，看到R0是出入的参数值，我们可以查看R0寄存器的内容，然后看到是123456，这个就是Java层传入的密码字符串，接着往下走：

```
libencrypt.so:74FE4EC4 CBZ                     R0, locret_74FE4EEC
libencrypt.so:74FE4EC6 MOV                     R0,
libencrypt.so:74FE4EC8 BL                      get_R0=00000001
```

返回值保存在R0中是1，不等于0，不进行跳转

这里把is_number函数返回值保存到R0寄存中，然后调用CBZ指令，判断是否为0，如果为0就跳转到locret_74FE4EEC处，查看R0寄存器的值不是0，继续往下走：

看到了get_encrypt_str函数的调用，函数的返回值保存在R1寄存器中，查看内容：zytyrTRA*B了，那么看到，上层传递的：123456=》zytyrTRA*B了，前面我们静态分析了get_encrypt_str函数的逻辑，继续往下看：

```
libencrypt.so:74FE4ED0 ADD                 R0, PC              ; "ssBCqpBssP"
libencrypt.so:74FE4ED2 BLX                 unk_74FE4DB8
libencrypt.so:74FE4ED6 LDR                 R3, [R6]
libencrypt.so:74FE4ED8 MOV                 R1, R7
libencrypt.so:74FE4EDA MOV                 R2, R5
libencrypt.so:74FE4EDC LDR.W               R3, [R3,#0x2A8]
libencrypt.so:74FE4EE0 MOV                 R4, R0
libencrypt.so:74FE4EE2 MOV                 R0, R6
libencrypt.so:74FE4EE4 BLX                 R3
libencrypt.so:74FE4EE6 CLZ.W               R0, R4
```

这里得到加密之后的内容和正确的密码ssBCqpBssP进行比较

看到了，这里把上面得到的字符串和ssBCqpBssP作比较，那么这里ssBCqpBssP就是正确的加密密码了，那么我们现在的资源是：

正确的加密密码：ssBCqpBssP，加密密钥库：zytyrTRA*BniqCPpVs，加密逻辑get_encrypt_str

那么我们可以写一个逆向的加密方法，去解析正确的加密密码得到值即可，这里为了给大家一个破解的机会，这里就不公布正确答案了，这个apk我随后会上传，手痒的同学可以尝试破解一下。

第三、总结IDA调试的流程

到这里，我们就分析了如何破解apk的流程，下面来总结一下：

1、我们通过解压apk文件，得到对应的so文件，然后使用IDA工具打开so,找到指定的native层函数

2、通过IDA中的一些快捷键：F5,Ctrl+S,Y等键来静态分析函数的arm指令，大致了解函数的执行流程

3、再次打开一个IDA来进行调试so

1>将IDA目录中的android_server拷贝到设备的指定目录下，修改android_server的运行权限，用Root身份运行android_server

2>使用adb forward进行端口转发，让远程调试端IDA可以连接到被调试端

3>使用IDA连接上转发的端口，查看设备的所有进程，找到我们需要调试的进程。

4>通过打开so文件，找到需要调试的函数的相对地址，然后在调试页面使用Ctrl+S找到so文件的基地址，相加之后得到绝对地址，使用G键，跳转到函数的地址处，下好断点。点击运行或者F9键。

5>触发native层的函数，使用F8和F7进行单步调试，查看关键的寄存器中的值，比如函数的参数，和函数的返回值等信息

总结就是：在调试so的时候，需要双开IDA，动静结合分析。

五、使用IDA来解决反调试问题

那么到这里我们就结束了我们这期的破解旅程了？答案是否定的，因为我们看到上面的例子其实是我自己先写了一个apk,目的就是为了给大家演示，如何使用IDA来进行动态调试so，那么下面我们还有一个操刀动手的案例，就是2014年，阿里安全挑战赛的第二题：AliCrackme_2：

阿里真会制造氛围，还记得我们破解的第一题吗，这次看到了第二题，好吧，下面来看看破解流程吧：

首先使用aapt命令查看他的AndroidManifest.xml文件，得到入口的Activity类：

```
C:\Users\jiangwei1-g\Desktop\Android中的动态调试>aapt dump xmltree AliCrackme_2.apk AndroidManifest.xml > D:\demo.txt

C:\Users\jiangwei1-g\Desktop\Android中的动态调试>start D:\demo.txt

C:\Users\jiangwei1-g\Desktop\Android中的动态调试>
```

demo.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
: android=http://schemas.android.com/apk/res/android
 E: manifest (line=2)
   A: android:versionCode(0x0101021b)=(type 0x10)0x1
   A: android:versionName(0x0101021c)="1.0" (Raw: "1.0")
   A: package="com.yaotong.crackme" (Raw: "com.yaotong.crackme")
   E: uses-sdk (line=7)
     A: android:minSdkVersion(0x0101020c)=(type 0x10)0x8
     A: android:targetSdkVersion(0x01010270)=(type 0x10)0x13
   E: application (line=11)
     A: android:label(0x01010001)=@0x7f070000
     A: android:icon(0x01010002)=@0x7f020001
     A: android:allowBackup(0x01010280)=(type 0x12)0xffffffff
     E: activity (line=15)
       A: android:label(0x01010001)=@0x7f070000
       A: android:name(0x01010003)="com.yaotong.crackme.MainActivity" (Raw: "com.yaotong.crackme.MainActivity")
       E: intent-filter (line=18)
         E: action (line=19)
           A: android:name(0x01010003)="android.intent.action.MAIN" (Raw: "android.intent.action.MAIN")
         E: category (line=21)
           A: android:name(0x01010003)="android.intent.category.LAUNCHER" (Raw: "android.intent.category.LAUNCHER")
     E: activity (line=24)
       A: android:name(0x01010003)="com.yaotong.crackme.ResultActivity" (Raw: "com.yaotong.crackme.ResultActivity")
```

然后使用dex2jar和jd-gui查看他的源码类：com.yaotong.crackme.MainActivity：

```
C:\Users\jiangwei1-g\Desktop\Android中的动态调试\dex2jar-0.0.9.15>dex2jar classes.dex
this cmd is deprecated, use the d2j-dex2jar if possible
dex2jar version: translator-0.0.9.15
dex2jar classes.dex -> classes_dex2jar.jar
Done.

C:\Users\jiangwei1-g\Desktop\Android中的动态调试\dex2jar-0.0.9.15>_
```

http://blog.csdn.net/ Android技术分享

```
com.yaotong.crackme
  BuildConfig
  MainActivity
    MainActivity
      btn_submit : Button
      inputCode : EditText
      onCreate(Bundle) : void
      securityCheck(String) : boolean
      {...}
  R
  ResultActivity
```

```java
package com.yaotong.crackme;

import android.app.Activity;

public class MainActivity extends Activity
{
  public Button btn_submit;
  public EditText inputCode;

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

  protected void onCreate(Bundle paramBundle)
  {
    super.onCreate(paramBundle);
    setContentView(2130903040);
    getWindow().setBackgroundDrawableResource(2130837504);
    this.inputCode = ((EditText)findViewById(2131099648));
    this.btn_submit = ((Button)findViewById(2131099649));
    this.btn_submit.setOnClickListener(new View.OnClickListener()
    {
      public void onClick(View paramAnonymousView)
      {
        String str = MainActivity.this.inputCode.getText().toString();
        if (MainActivity.this.securityCheck(str))
        {
          Intent localIntent = new Intent(MainActivity.this, ResultActivity.class);
          MainActivity.this.startActivity(localIntent);
          return;
        }
        Toast.makeText(MainActivity.this.getApplicationContext(), "验证码校验失败", 0).show();
      }
    });
  }

  public native boolean securityCheck(String paramString);
}
```

Android技术分享

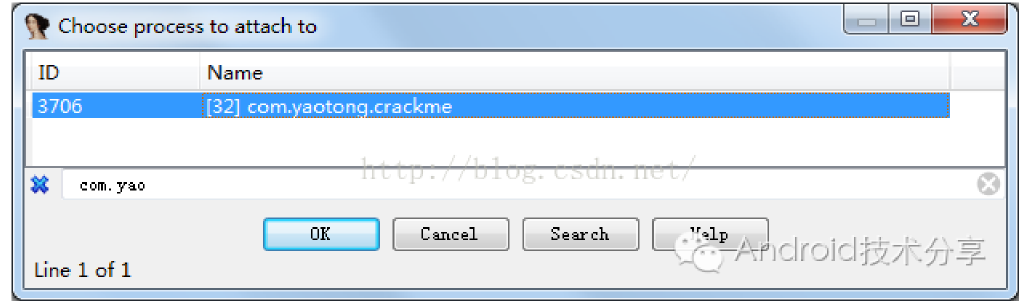
看到，他的判断，是securityCheck方法，是一个native层的，所以这时候我们去解压apk文件，获取他的so文件，使用IDA打开查看native函数的相对地址：11A8

```
Function name
__cxa_atexit
__cxa_finalize
__android_log_print
__aeabi_memset
__stack_chk_fail
free
dlsym
malloc
raise
__gnu_Unwind_Find_exidx
abort
memcpy
__cxa_begin_cleanup
__cxa_type_match
sub_1164
Java_com_yaotong_crackme_MainActivity_securityCheck
sub_130C
sub_16A4
sub_17F4
JNI_OnLoad
sub_1CA8
sub_22AC
sub_2378
sub_239C
sub_2494
sub_24F4
sub_254C
sub_258C
__umodsi3
__aeabi_drsub
__subdf3
__aeabi_dadd
__floatunsidf
__floatsidf
__extendsfdf2
__floatundidf
__floatdidf
__muldf3
__divdf3
sub_2E50
__fixdfsi
```

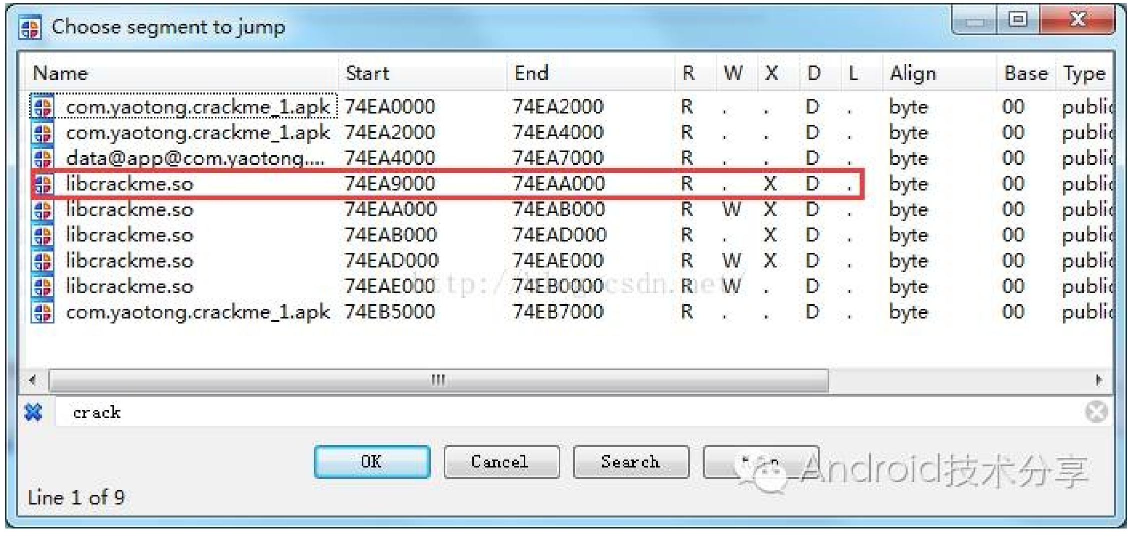
```
.text:000011A8
.text:000011A8 ; =============== S U B R O U T I N E =======================
.text:000011A8
.text:000011A8
.text:000011A8                 EXPORT Java_com_yaotong_crackme_MainActivity_se
.text:000011A8 Java_com_yaotong_crackme_MainActivity_securityCheck
.text:000011A8
.text:000011A8 var_20          = -0x20
.text:000011A8 var_1C          = -0x1C
.text:000011A8
.text:000011A8                 STMFD   SP!, {R4-R7,R11,LR}
.text:000011AC                 SUB     SP, SP, #8
.text:000011B0                 MOV     R5, R0
.text:000011B4                 LDR     R0, =(_GLOBAL_OFFSET_TABLE_ - 0x11C8)
.text:000011B8                 LDR     R6, =(unk_6290 - 0x5FBC)
.text:000011BC                 MOV     R4, R2
.text:000011C0                 ADD     R0, PC, R0 ; _GLOBAL_OFFSET_TABLE_
.text:000011C4                 ADD     R0, R6, R0 ; unk_6290
.text:000011C8                 LDRB    R0, [R0,#(byte_6359 - 0x6290)]
.text:000011CC                 CMP     R0, #0
.text:000011D0                 BNE     loc_1214
.text:000011D4                 MOV     R1, #2
.text:000011D8                 MOV     R0, #7
.text:000011DC                 STR     R1, [SP,#0x20+var_20]
.text:000011E0                 STR     R0, [SP,#0x20+var_1C]
.text:000011E4                 LDR     R0, =(_GLOBAL_OFFSET_TABLE_ - 0x11F4)
.text:000011E8                 LDR     R1, =(unk_446B - 0x5FBC)
.text:000011EC                 ADD     R0, PC, R0 ; _GLOBAL_OFFSET_TABLE_
.text:000011F0                 ADD     R2, R1, R0
.text:000011F4                 LDR     R1, =(unk_4468 - 0x5FBC)
.text:000011F8                 ADD     R7, R6, R0 ; unk_6290
.text:000011FC                 ADD     R3, R1, R0 ; unk_4468
.text:00001200                 ADD     R0, R7, #0x74
.text:00001204                 MOV     R1, #8
.text:00001208                 BL      sub_2494
.text:0000120C                 MOV     R0, #1
.text:00001210                 STRB    R0, [R7,#(byte_6359 - 0x6290)]
.text:00001214
.text:00001214 loc_1214                                ; CODE XREF: Java_com_
.text:00001214                 LDR     R0, =(_GLOBAL_OFFSET_TABLE_ - 0x1220)
.text:00001218                 ADD     R0, PC, R0 ; _GLOBAL_OFFSET_TABLE_
.text:0000121C                 ADD     R0, R6, R0 ; unk_6290
.text:00001220                 LDRB    R0, [R0,#(byte_635A - 0x6290)]
```

这里的ARM指令代码不在分析了，大家自行查看即可，我们直接进入调试即可：

在打开一个IDA进行关联调试：



选择对应的调试进程，然后确定：

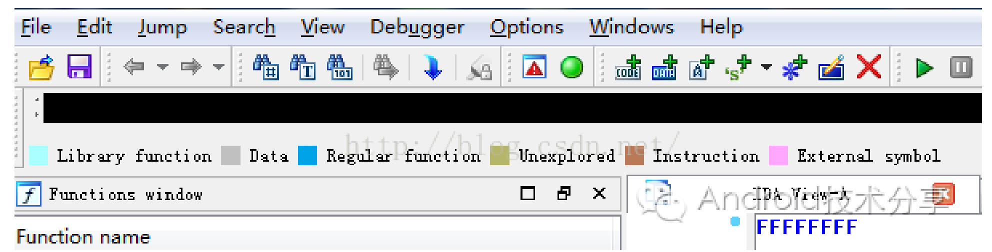


使用Ctrl+S键找到对应so文件的基地址：74EA9000

和上面得到的相对地址相加得到绝对地址：74EA9000+11A8=74EAA1A8 使用G键直接跳到这个地址：

```
libcrackme.so:74EAA1A8 Java_com_yaotong_crackme_MainActivity_securityCheck
libcrackme.so:74EAA1A8
libcrackme.so:74EAA1A8 var_20= -0x20
libcrackme.so:74EAA1A8 var_1C= -0x1C
libcrackme.so:74EAA1A8
libcrackme.so:74EAA1A8 STMFD           SP!, {R4-R7,R11,LR}
libcrackme.so:74EAA1AC SUB             SP, SP, #8
libcrackme.so:74EAA1B0 MOV             R5, R0
libcrackme.so:74EAA1B4 LDR             R0, =(unk_74EAEFBC - 0x74EAA1C8)
libcrackme.so:74EAA1B8 LDR             R6, =0x2b4
libcrackme.so:74EAA1BC MOV             R4, R2
```

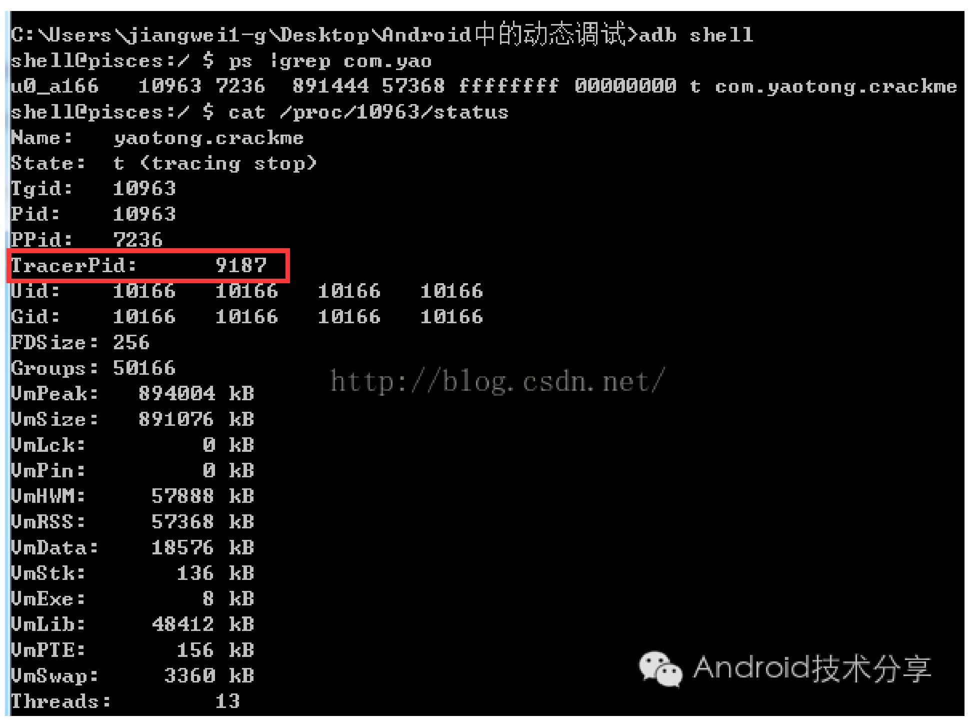
下个断点，然后点击F9运行程序：



擦，IDA退出调试页面了，我们再次进入调试页面，运行，还是退出调试页面了，好了，这下蛋疼了，没法调试了。

这里其实是阿里做了反调试侦查，如果发现自己的程序被调试了，就直接退出程序，那么这里有问题了，为什么知道是反调试呢？这个主要还是看后续自己的破解经验了，没技术可言，还有一个就是阿里如何做到的反调试策略的，这里限于篇幅，只是简单介绍一下原理：

前面说到，IDA是使用android_server在root环境下注入到被调试的进程中，那么这里用到一个技术就是Linux中的ptrace，关于这个这里也不解释了，大家可以自行的去搜一下ptrace的相关知识，那么Android中如果一个进程被另外一个进程ptrace了之后，在他的status文件中有一个字段：TracerPid 可以标识是被哪个进程trace了，我们可以使用命令查看我们的被调试的进行信息：status文件在：/proc/[pid]/status



```
C:\Users\jiangwei1-g\Desktop\Android中的动态调试>adb shell
shell@pisces:/ $ ps |grep com.yao
u0_a166   10963 7236  891444 57368 ffffffff 00000000 t com.yaotong.crackme
shell@pisces:/ $ cat /proc/10963/status
Name:   yaotong.crackme
State:  t (tracing stop)
Tgid:   10963
Pid:    10963
PPid:   7236
TracerPid:      9187
Uid:    10166   10166   10166   10166
Gid:    10166   10166   10166   10166
FDSize: 256
Groups: 50166
VmPeak:   894004 kB
VmSize:   891076 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:     57888 kB
VmRSS:     57368 kB
VmData:    18576 kB
VmStk:       136 kB
VmExe:         8 kB
VmLib:     48412 kB
VmPTE:       156 kB
VmSwap:     3360 kB
Threads:        13
```

看到了，这里的进程被9187进程trace了，我们在用ps命令看看9187是哪个进程：

```
shell@pisces:/ $ ps |grep 9187
root      9187  9138  10180  8448  ffffffff 000[illegible]er
shell@pisces:/ $
```

果不其然，是我们的android_server进程，好了，我们知道原理了，也大致猜到了阿里在底层做了一个循环检测这个字段如果不为0，那么代表自己进程在被人trace，那么就直接停止退出程序，这个反检测技术用在很多安全防护的地方，也算是一个重要的知识点了。

那么下面就来看看如何应对这个反调试？

我们刚刚看到，只要一运行程序，就退出了调试界面，说明，这个循环检测程序执行的时机非常早，那么我们现在知道的最早的两个时机是：一个是.init_array，一个是JNI_OnLoad

.init_array是一个so最先加载的一个段信息，时机最早，现在一般so解密操作都是在这里做的

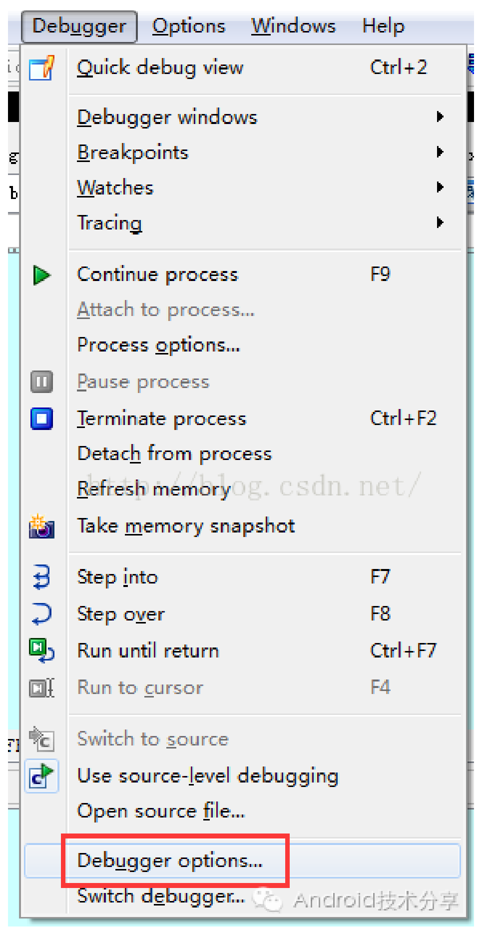
JNI_OnLoad是so被System.loadLibrary调用的时候执行，他的时机要早于哪些native方法执行，但是没有.init_array时机早

那么知道了这两个时机，下面我们先来看看是不是在JNI_OnLoad函数中做的策略，所以我们需要先动态调试JNI_OnLoad函数

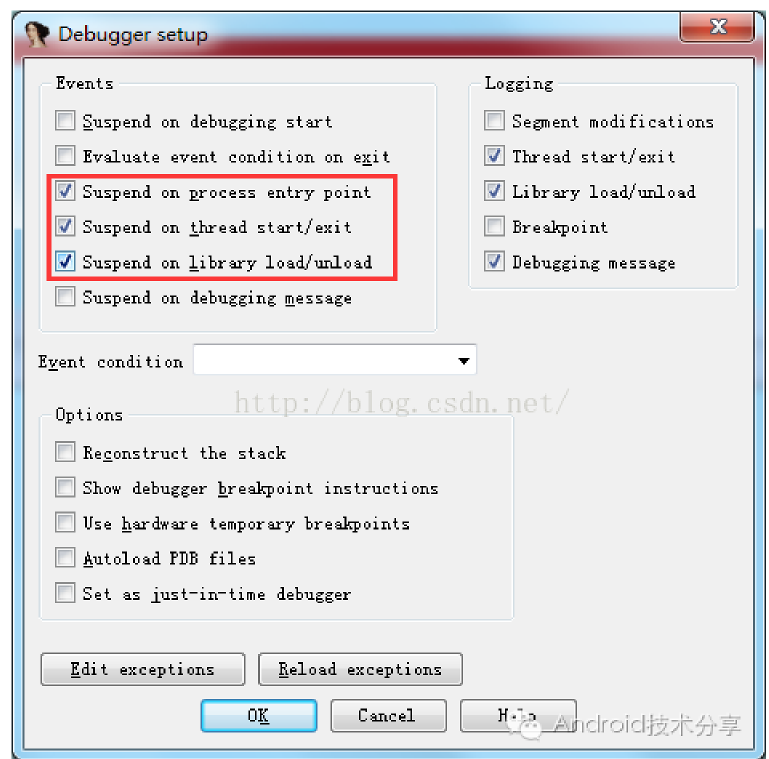
我们既然知道了JNI_OnLoad函数的时机，如果阿里把检测函数放在这里的话，我们不能用之前的方式去调试了，因为之前的那种方式时机太晚了，只要运行就已经执行了JNI_OnLoad函数，所以就会退出调试页面

幸好这里IDA提供了在so文件load的时机，我们只需要在Debug Option中设置一下就可以了：

在调试页面的Debugger 选择 Debugger Option选项：



然后勾选Suspend on library load/unload即可



这样设置之后，还是不行，因为我们程序已经开始运行，就在static代码块中加载so文件了，static的时机非常早，所以这时候，我们需要让程序停在加载so文件之前即可。

```
MainActivity.class
package com.yaotong.crackme;

import android.app.Activity;

public class MainActivity extends Activity
{
  public Button btn_submit;
  public EditText inputCode;

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

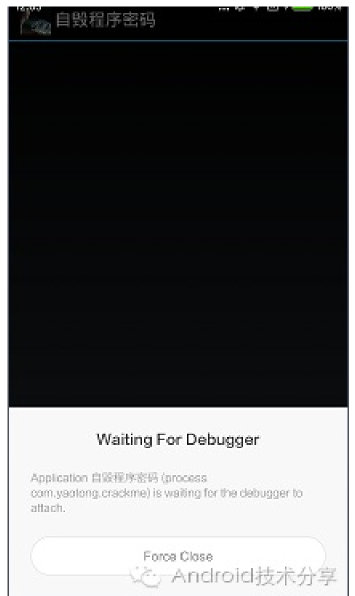
那么我想到的就是添加代码waitForDebugger代码了，这个方法就是等待debug，我们还记得在之前的调试smali代码的时候，就是用这种方式让程序停在了启动出，然后等待我们去用jdb进行attach操作。

那么这一次我们可以在System.loadLibrary方法之前加入waitForDebugger代码即可，但是这里我们不这么干了，还有一种更简单的方式就是用am命令，本身am命令可以启动一个程序，当然可以用debug方式启动：

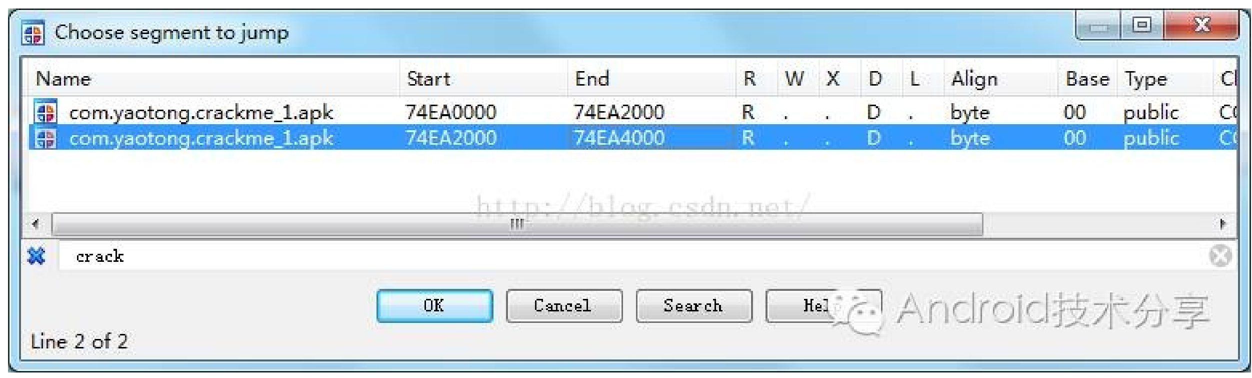
adb shell am start -D -n com.yaotong.crackme/.MainActivity

这里一个重要参数就是-D,用debug方式启动

运行完之后，设备是出于一个等待Debugger的状态：



这时候，我们再次使用IDA进行进程的附加，然后进入调试页面，同时设置一下Debugger Option选项，然后定位到JNI_OnLoad函数的绝对地址。



但是我们发现，这里没有RX权限的so文件，说明so文件没有加载到内存中，想一想还是对的，以为我们现在的程序是wait Debugger，也就是还没有走System.loadLibrary方法，so文件当然没有加载到内存中，所以我们需要让我们程序跑起来，这时候我们可以使用jdb命令去attach等待的程序，命令如下：

jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700

其实这条命令的功能类似于，我们前一篇说到用Eclipse调试smali源码的时候，在Eclipse中设置远程调试工程一样，选择Attach方式，调试机的ip地址和端口，还记得8700端口是默认的端口，但是我们运行这个命令之后，出现了一个错误：

擦，无法连接到目标的VM，那么这种问题大部分都出现在被调试程序不可调试，我们可以查看apk的android:debuggable属性：

```
C:\Windows\system32\cmd.exe

C:\Users\jiangwei1-g\Desktop\Android中的动态调试>aapt dump xmltree AliCrackme_2.apk AndroidManifest.xml > D:\demo.txt

C:\Users\jiangwei1-g\Desktop\Android中的动态调试>start D:\demo.txt

C:\Users\jiangwei1-g\Desktop\Android中的动态调试>
```

```
demo.txt - 记事本
文件(F)  编辑(E)  格式(O)  查看(V)  帮助(H)
N: android=http://schemas.android.com/apk/res/android
  E: manifest (line=2)
    A: android:versionCode(0x0101021b)=(type 0x10)0x1
    A: android:versionName(0x0101021c)="1.0" (Raw: "1.0")
    A: package="com.yaotong.crackme" (Raw: "com.yaotong.crackme")
    E: uses-sdk (line=7)
      A: android:minSdkVersion(0x0101020c)=(type 0x10)0x8
      A: android:targetSdkVersion(0x01010270)=(type 0x10)0x13
    E: application (line=11)
      A: android:label(0x01010001)=@0x7f070000
      A: android:icon(0x01010002)=@0x7f020001
      A: android:allowBackup(0x01010280)=(type 0x12)0xffffffff
      E: activity (line=15)
        A: android:label(0x01010001)=@0x7f070000
        A: android:name(0x01010003)="com.yaotong.crackme.MainActivity" (Raw: "com.yaotong.crackme.MainActivity")
        E: intent-filter (line=18)
          E: action (line=19)
            A: android:name(0x01010003)="android.intent.action.MAIN" (Raw: "android.intent.action.MAIN")
          E: category (line=21)
            A: android:name(0x01010003)="android.intent.category.LAUNCHER" (Raw: "android.intent.category.LAUNCHER")
      E: activity (line=24)
        A: android:name(0x01010003)="com.yaotong.crackme.ResultActivity" (Raw: "com.yaotong.crackme.ResultActivity")
```

这里没有android:debuggable="true"属性，所以不能调试，我们需要添加这个属性，然后回编译

果不其然，这里没有debug属性，所以这个apk是不可以调试的，所以我们需要添加这个属性，然后在回编译即可：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1" android:versionName="1.0" android:debuggable="true" package="com.yaotong.crackme"
  xmlns:android="http://schemas.android.com/apk/res/android">
    <application android:label="@string/app_name" android:icon="@drawable/creakme2_logo" android:allowBackup="true">
        <activity android:label="@string/app_name" android:name="com.yaotong.crackme.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name="com.yaotong.crackme.ResultActivity" />
    </application>
</manifest>
```

回编译：java -jar apktool.jar b -d out -o debug.apk

签名apk：java -jar .\sign\signapk.jar .\sign\testkey.x509.pem .\sign\testkey.pk8 debug.apk debug.sig.apk

然后在次安装，使用am 命令启动：

第一步：运行：adb shell am start -D -n com.yaotong.crackme/.MainActivity

出现Debugger的等待状态

第二步：启动IDA 进行目标进程的Attach操作

第三步：运行：jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700

```
C:\Users\jiangwei1-g>jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700
```

第三步：设置Debugger Option选项

第四步：点击IDA运行按钮，或者F9快捷键，运行

```
C:\Users\jiangwei1-g>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...
> _
```

看到了，这次jdb成功的attach住了，debug消失，正常运行了，

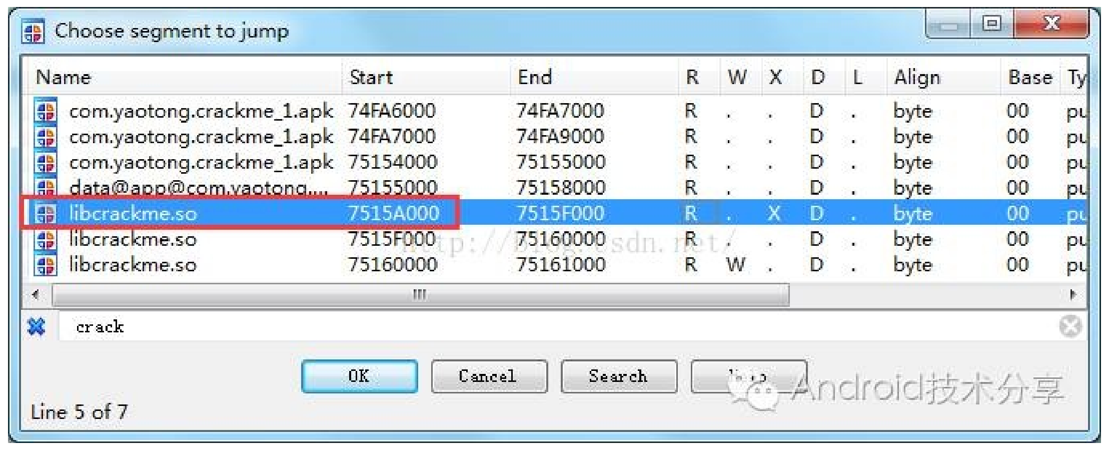
但是同时弹出了一个选择提示：

```
Add map...
IDA could not find the file "/data/app-lib/com.yaotong.crackme-1/libcrackme.so".
If that file can be accessed, please specify its directory:
Source        /data/app-lib/com.yaotong.crackme-1
Destination
              OK    Cancel
```

这时候，不用管它，全部选择取消按钮，然后就运行到了linker模块了：

```
linker:400C70D2 CODE16
linker:400C70D2 DCB 0x75 ; u
linker:400C70D3 DCB 0x4B ; K
linker:400C70D4 DCB    4
linker:400C70D5 DCB 0xF5 ;
linker:400C70D6 DCB 0x82 ;
linker:400C70D7 DCB 0x72 ; r
linker:400C70D8 DCB 0xC4 ;
linker:400C70D9 DCB 0xF8 ;
linker:400C70DA DCB    8
linker:400C70DB DCB 0x41 ; A
linker:400C70DC DCB 0xD4 ;
```

这时候，说明so已经加载进来了，我们再去获取JNI_OnLoad函数的绝对地址


Choose segment to jump

Name	Start	End	R	W	X	D	L	Align	Base	Ty
com.yaotong.crackme_1.apk	74FA6000	74FA7000	R	.	.	D	.	byte	00	pu
com.yaotong.crackme_1.apk	74FA7000	74FA9000	R	.	.	D	.	byte	00	pu
com.yaotong.crackme_1.apk	75154000	75155000	R	.	.	D	.	byte	00	pu
data@app@com.yaotong....	75155000	75158000	R	.	.	D	.	byte	00	pu
libcrackme.so	7515A000	7515F000	R	.	X	D	.	byte	00	pu
libcrackme.so	7515F000	75160000	R	.	.	D	.	byte	00	pu
libcrackme.so	75160000	75161000	R	W	.	D	.	byte	00	pu

crack

OK Cancel Search

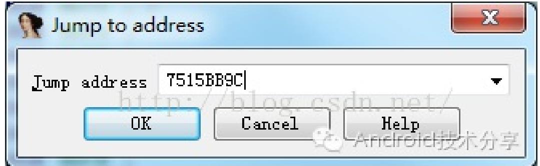
Line 5 of 7

Ctrl+S查找到了基地址：7515A000

用静态方式IDA打开so查看相对地址：1B9C

```
.text:00001B9C JNI_OnLoad
.text:00001B9C
.text:00001B9C handle          = -0x20
.text:00001B9C
.text:00001B9C                 STMFD   SP!, {R4-R9,R11,LR}
```

相加得到绝对地址：7515A000+1B9C=7515BB9C，然后点击S键，跳转：



跳转到指定的函数位置：

```
libcrackme.so:7515BB9C JNI_OnLoad
libcrackme.so:7515BB9C
libcrackme.so:7515BB9C var_20= -0x20
libcrackme.so:7515BB9C
libcrackme.so:7515BB9C STMFD             SP!, {R4-R9,R11,LR}
libcrackme.so:7515BBA0 ADD               R11, SP, #0x18
libcrackme.so:7515BBA4 SUB               SP, SP, #8
libcrackme.so:7515BBA8 MOV               R4, R0
libcrackme.so:7515BBAC LDR               R0, =(unk_7515FFBC - 0x7515BBC0)
libcrackme.so:7515BBB0 LDR               R9, =0x2D4
libcrackme.so:7515BBB4 MOV               R8, #0
libcrackme.so:7515BBB8 ADD               R0, PC, R0 ; unk_7515FFBC
libcrackme.so:7515BBBC ADD               R0, R9, R0
libcrackme.so:7515BBC0 STR               R8, [R0,#(dword_751602C8 - 0x75160290)]
libcrackme.so:7515BBC4 LDR               R5, [R0,#(dword_751602C4 - 0x75160290)]
libcrackme.so:7515BBC8 CMP               R5, #0
libcrackme.so:7515BBCC BEQ               loc_7515BC28
libcrackme.so:7515BBD0
libcrackme.so:7515BBD0 loc_7515BBD0                         ; CODE XREF: JNI_
libcrackme.so:7515BBD0 LDR               R0, [R5]
libcrackme.so:7515BBD4 CMP               R0, #1
libcrackme.so:7515BBD8 BLT               loc_7515BBFC
libcrackme.so:7515BBDC ADD               R7, R5, #4
libcrackme.so:7515BBE0 MOV               R6, #0
```

这时候再次点击运行，进入了JNI_OnLoad处的断点：

```
R8 libcrackme.so:7515BB9C STMFD             SP!, {R4-R9,R11,LR}
   libcrackme.so:7515BBA0 ADD               R11, SP, #0x18
   libcrackme.so:7515BBA4 SUB               SP, SP, #8
   libcrackme.so:7515BBA8 MOV               R4, R0
   libcrackme.so:7515BBAC LDR               R0, =(unk_7515FFBC -
PC libcrackme.so:7515BBB0 LDR               R9, =0x2D4
   libcrackme.so:7515BBB4 MOV               R8, #0
   libcrackme.so:7515BBB8 ADD               R0, PC, R0 ; unk_751
   libcrackme.so:7515BBBC ADD               R0, R9, R0
   libcrackme.so:7515BBC0 STR               R8, [R0,#(dword_7516
   libcrackme.so:7515BBC4 LDR               R5, [R0,#(dword_7516
   libcrackme.so:7515BBC8 CMP               R5, #0
   libcrackme.so:7515BBCC BEQ               loc_7515BC28
   libcrackme.so:7515BBD0
   libcrackme.so:7515BBD0 loc_7515BBD0
```

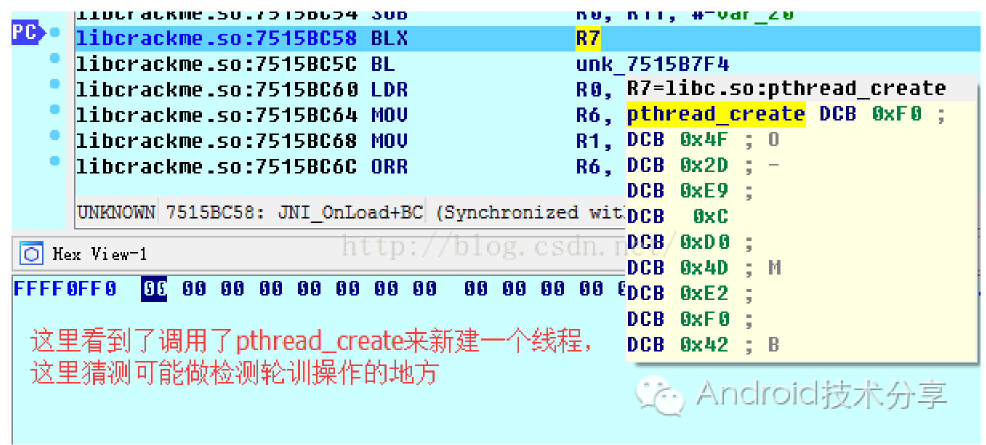
下面咱们就开始单步调试了，但是当我们每次到达BLX R7这条指令执行完之后，就JNI_OnLoad就退出了：

```
   libc.so:4010C864 ; ------------------------------
PC libc.so:4010C864 MOVS              R0, R0
   libc.so:4010C868 BEQ               loc_4010C880
   libc.so:4010C86C MOV               R7, R12
   libc.so:4010C870 CMN               R0, #0x1000
   libc.so:4010C874 BXLS              LR
   libc.so:4010C878 RSB               R0, R0, #0
   libc.so:4010C87C B                 sub_40128D74
   libc.so:4010C880 ; ------------------------------
   libc.so:4010C880
   libc.so:4010C880 loc_4010C880
   libc.so:4010C880 LDMFD             SP!, {R0,R1}
   libc.so:4010C884 MOV               R2, SP
   libc.so:4010C888 B                 __thread_entry
   libc.so:4010C888 ; ------------------------------
   libc.so:4010C88C __bionic_clone DCB   0xD
```

经过好几次尝试都是一样的结果，所以我们发现这个地方有问题，可能就是反调试的地方了

我们再次进入调试，看见BLX跳转的地方R7寄存器中是pthread_create函数，这个是Linux中新建一个线程的方法

所以阿里的反调试就在这里开启一个线程进行轮训操作，去读取/proc/[pid]/status文件中的TrackerPid字段值，如果发现不为0，就表示有人在调试本应用，在JNI_OnLoad中直接退出。其实这里可以再详细进入查看具体代码实现的，但是这里限于篇幅问题，不详细解释了，后续在写一篇文章我们自己可以实现这种反调试机制的。本文的重点是能够动态调试即可。



那么问题找到了，我们现在怎么操作呢？

其实很简单，我们只要把BLX R7这段指令干掉即可，如果是smali代码的话，我们可以直接删除这行代码即可，但是so文件不一样，他是汇编指令，如果直接删除这条指令的话，文件会发生错乱，因为本身so文件就有固定的格式，比如很多Segement的内容，每个Segement的偏移值也是有保存的，如果这样去删除会影响这些偏移值，会破坏so文件格式，导致so加载出错的，所以这里我们不能手动的去删除这条指令，我们还有另外一种方法，就是把这条指令变成空指令，在汇编语言中，nop指令就是一个空指令，他什么都不干，所以这里我们直接改一下指令即可，**arm中对应的nop指令是：00 00 00 00**

那么我们看到BLX R7对应的指令位置为：1C58

```
.text:00001C58    BLX    R7 ; __imp_dlsym
.text:00001C5C    BL     sub_17F4
.text:00001C60    LDR    R0, [R4]
.text:00001C64    MOV    R6, #4
.text:00001C68    MOV    R1, R5
.text:00001C6C    ORR    R6, R6, #0x10000
```

查看他的Hex内容是：37 FF 2F E1

```
IDA View-A        Hex View-1        Structures
7515BC18  00 10 A0 E3 00 00 8F E0  00 00 89 E0 34 10 80 E5
7515BC28  08 50 4D E2 05 D0 A0 E1  68 00 9F E5 68 10 9F E5
7515BC38  00 30 A0 E3 00 80 85 E5  00 00 8F E0 00 20 81 E0
7515BC48  00 00 89 E0 00 10 A0 E3  24 70 90 E5 20 00 4B E2
7515BC58  37 FF 2F E1 E4 FE FF EB  00 00 94 E5 04 60 A0 E3
7515BC68  05 10 A0 E1 01 68 86 E3  06 20 A0 E1 18 30 90 E5
```

我们可以使用一些二进制文件软件进行内容的修改，这里使用010Editor工具进行修改：

```
1C10h: EE FF FF 1A 80 00 9F E5 00 10 A0 E3 00 00 8F E0
1C20h: 00 00 89 E0 34 10 80 E5 08 50 4D E2 05 D0 A0 E1
1C30h: 68 00 9F E5 68 10 9F E5 00 30 A0 E3 00 80 85 E5
1C40h: 00 00 8F E0 00 20 81 E0 00 00 89 E0 00 10 A0 E3
1C50h: 24 70 90 E5 20 00 4B E2 37 FF 2F E1 E4 FE FF EB
1C60h: 00 00 94 E5 04 60 A0 E3 05 10 A0 E1 01 68 86 E3
1C70h: 06 20 A0 E1 18 30 90 E5 04 00 A0 E1 33 FF 2F E1
```

这里直接修改成00 00 00 00：

```
1C40h: 00 00 8F E0 00 20 81 E0 00 00 89 E0 00 10 A0 E3
1C50h: 24 70 90 E5 20 00 4B E2 00 00 00 00 E4 FE FF EB
1C60h: 00 00 94 E5 04 60 A0 E3 05 10 A0 E1 01 68 86 E3
```

这时候，保存修改之后的so文件，我们再次使用IDA进行打开查看：

```
.text:00001C54                 SUB     R0, R11, #-var_20
.text:00001C58                 ANDEQ   R0, R0, R0
.text:00001C5C                 BL      sub_17F4
```

哈哈，指令被修改成了：ANDEQ R0，R0，R0了

那么修改了之后，我们在替换原来的so文件，再次重新回编译，签名安装，再次按照之前的逻辑给主要的加密函数下断点，这里不需要在给JNI_OnLoad函数下断点了，因为我们已经修改了反调试功能了，所以这里我们只需要按照这么简单几步即可：

第一步：启动程序

第二步：使用IDA进行进程的attach

第三步：找到Java_com_yaotong_crackme_MainActivity_securityCheck函数的绝对地址

第四步：打上断点，点击运行，进行单步调试

```
libcrackme.so:74FAF1A8 Java_com_yaotong_crackme_MainActivity_securityCheck
libcrackme.so:74FAF1A8
libcrackme.so:74FAF1A8 var_20= -0x20
libcrackme.so:74FAF1A8 var_1C= -0x1C
libcrackme.so:74FAF1A8
libcrackme.so:74FAF1A8 STMFD           SP!, {R4-R7,R11,LR}
libcrackme.so:74FAF1AC SUB             SP, SP, #8
libcrackme.so:74FAF1B0 MOV             R5, R0
libcrackme.so:74FAF1B4 LDR             R0, =(unk_74FB3FBC - 0x74FAF1C8)
libcrackme.so:74FAF1B8 LDR             R6, =0x2D4
libcrackme.so:74FAF1BC MOV             R4, R2
libcrackme.so:74FAF1C0 ADD             R0, PC, R0 ; unk_74FB3FBC
libcrackme.so:74FAF1C4 ADD             R0, R6, R0
libcrackme.so:74FAF1C8 LDRB            R0, [R0,#(byte_74FB4359 - 0x74FB4290)]
libcrackme.so:74FAF1CC CMP             R0, #0
libcrackme.so:74FAF1D0 BNE             loc_74FAF214
libcrackme.so:74FAF1D4 MOV             R1, #2
libcrackme.so:74FAF1D8 MOV             R0, #7
libcrackme.so:74FAF1DC STR             R1, [SP,#0x20+var_20]
libcrackme.so:74FAF1E0 STR             R0, [SP,#0x20+var_1C]
libcrackme.so:74FAF1E4 LDR             R0, =(unk_74FB3FBC - 0x74FAF1F4)
```

看到了吧，这里我们可以单步调试进来了啦啦，说明我们修改反调试指令成功了。

下面就继续F8单步调试：

```
libcrackme.so:74FAF2A8 LDRB            R3, [R2]
libcrackme.so:74FAF2AC LDRB            R1, [R0]
libcrackme.so:74FAF2B0 CMP             R3, R1
libcrackme.so:74FAF2B4 BNE             loc_74FAF2D0
libcrackme.so:74FAF2B8 ADD             R2, R2, #1
libcrackme.so:74FAF2BC ADD             R0, R0, #1
libcrackme.so:74FAF2C0 MOV             R1, #1
libcrackme.so:74FAF2C4 CMP             R3, #0
libcrackme.so:74FAF2C8 BNE             loc_74FAF2A8
libcrackme.so:74FAF2CC B               loc_74FAF2D4
libcrackme.so:74FAF2D0 ; ---------------------------------------------------------
libcrackme.so:74FAF2D0
libcrackme.so:74FAF2D0 loc_74FAF2D0                            ; CODE XREF: Java_com_yao
libcrackme.so:74FAF2D0 MOV             R1, #0
```

这里我们在调试的时候，发现输入一个密码，调试到这里，就直接跳过去了，前面有一个CMP指令，说明这里很有可能是比较密码的地方，我们需要再一次进入调试，注意R3和R1寄存器的值内容

http://blog.csdn.net/
Android技术分享

调试到这里，发现一个问题，就是CMP指令之后，BNE 指令就开始跳转到loc_74FAF2D0处了，那么我们就可以猜到了，CMP指令比较的应该就是我们输入的密码和正确的密码，我们再次从新调试，看看R3和R1寄存器的值

```
libcrackme.so:74FAF2A8 LDRB                R3, [R2]
libcrackme.so:74FAF2AC LDRB                R1, [R0]
libcrackme.so:74FAF2B0 CMP                 R3, R1    [R2]=[libcrackme.so:_Unwind_GetTextRelBase+14]
libcrackme.so:74FAF2B4 BNE                 loc_74FA DCB 0x61 ; a
libcrackme.so:74FAF2B8 ADD                 R2, R2,  DCB 0x69 ; i
libcrackme.so:74FAF2BC ADD                 R0, R0,  DCB 0x79 ; y
libcrackme.so:74FAF2C0 MOV                 R1, #1   DCB 0x6F ; o
libcrackme.so:74FAF2C4 CMP                 R3, #0   DCB 0x75 ; u
libcrackme.so:74FAF2C8 BNE                 loc_74FA DCB 0x2C ; ,
libcrackme.so:74FAF2CC B                   loc_74FA DCB 0x62 ; b
libcrackme.so:74FAF2D0 ; ------------------------ DCB 0x75 ; u
libcrackme.so:74FAF2D0                              DCB 0x63 ; c
libcrackme.so:74FAF2D0 loc_74FAF2D0                 DCB 0x75 ; u
```

看到CMP指令比较的是R3寄存器，先看R3存的值是采用寄存器寻址方式，看到是一个字符串：aiyou,bucu，那么R2寄存器中的值就是这个字符串的地址，但是这里看到这个字符串应该没显示完，因为我们没看到结束字符0所以我们点击R2进入查看

看到了这里的R3寄存器的值就是用寄存器寻址方式，赋值字符串的，这里R2寄存器就是存放字符串的地址，我们看到的内容是aiyou...但是这里肯定不是全部字符串，因为我们没看到字符串的结束符：'\0'，我们点击R2寄存器，进入查看完整内容：

```
libcrackme.so:74FB2450 DCB 0x61 ; a
libcrackme.so:74FB2451 DCB 0x69 ; i
libcrackme.so:74FB2452 DCB 0x79 ; y
libcrackme.so:74FB2453 DCB 0x6F ; o
libcrackme.so:74FB2454 DCB 0x75 ; u
libcrackme.so:74FB2455 DCB 0x2C ; ,
libcrackme.so:74FB2456 DCB 0x62 ; b
libcrackme.so:74FB2457 DCB 0x75 ; u
libcrackme.so:74FB2458 DCB 0x63 ; c
libcrackme.so:74FB2459 DCB 0x75 ; u
libcrackme.so:74FB245A DCB 0x6F ; o
libcrackme.so:74FB245B DCB 0x6F ; o
libcrackme.so:74FB245C DCB    0
```

这里是全部内容：aiyou,bucuoo

我们继续查看R1寄存器的内容：

```
libcrackme.so:74FAF2A8 LDRB                R3, [R2]
libcrackme.so:74FAF2AC LDRB                R1, [R0]
libcrackme.so:74FAF2B0 CMP                 R3, R1
libcrackme.so:74FAF2B4 BNE                 loc_74FAF [R0]=[debug148:760C50B8]
libcrackme.so:74FAF2B8 ADD                 R2, R2, # DCB 0x6A ; j
libcrackme.so:74FAF2BC ADD                 R0, R0, # DCB 0x69 ; i
libcrackme.so:74FAF2C0 MOV                 R1, #1    DCB 0x61 ; a
libcrackme.so:74FAF2C4 CMP                 R3, #0    DCB 0x6E ; n
libcrackme.so:74FAF2C8 BNE                 loc_74FAF DCB 0x67 ; g
libcrackme.so:74FAF2CC B                   loc_74FAF DCB 0x77 ; w
libcrackme.so:74FAF2D0 ; ------------------------- DCB 0x65 ; e
libcrackme.so:74FAF2D0                               DCB 0x69 ; i
libcrackme.so:74FAF2D0 loc_74FAF2D0                  DCB    0
libcrackme.so:74FAF2D0 MOV                 R1, #0    DCB    0
```

这里的比较还有一个寄存器就是R1,我们查看他的内容，也是用寄存器寻址，字符串保存在寄存器R0中，我们看到这个字符串是我们输入的值

这里也是同样用寄存器寻址，R0寄存器存储的是R1中字符串的地址，我们看到这里的字符串内容是：jiangwei

这个就是我输入的内容，那么这里就可以豁然开朗了，密码是上面的：aiyou,bucuoo

我们再次输入这个密码：

哈哈哈，破解成功啦啦~~

六、技术总结

到这里我们算是讲解完了如何使用IDA来调试so代码，从而破解apk的知识了，因为这里IDA工具比较复杂，所以这篇文章篇幅有点长，所以同学们可以多看几遍，就差不多了。下面我们来整理一下这篇文章中涉及到的知识点吧：

第一、IDA中的常用快捷键使用

1、Shift+F12可以快速查看so中的常量字符串内容，有时候，字符串内容是一个很大的突破点

2、使用强大的F5键，可以查看arm汇编指令对应的C语言代码，同时可以使用Y键，进行JNIEnv*方法的还原

3、使用Ctrl+S键，可以在IDA View页面中查看so的所有段信息，在调试页面可以查找对应so文件映射到内存的基地址，这里我们还可以使用G键，进行地址的跳转

4、使用F8进行单步调试，F7进行单步跳入调试，同时可以使用F9运行程序

第二、ARM汇编指令相关知识

1、了解了几种寻址方式，有利于我们简单的读懂arm汇编指令代码

2、了解了arm中的几种寄存器的作用，特别是PC寄存器

3、了解了arm中常用的指令，比如：**MOV，ADD，SUB，LDR，STR，CMP，CBZ，BL，BLX**

第三、使用IDA进行调试so的步骤，这里分两种情况

1、IDA调试无反调试的so代码步骤：

1》把IDA安装目录中的android_server拷贝到设备的指定目录中，修改android_server的权限，并且用root方式运行起来，监听23946端口

2》使用adb forward命令进行端口的转发，将设备被调试端的端口转发到远程调试端中

3》双开IDA工具，一个是用来打开so文件，进行文件分析，比如简单分析arm指令代码，知道大体逻辑，还有就是找到具体函数的相对位置等信息，还有一个IDA是用来调试so文件的，我们在Debugger选项中设置Debugger Option，然后附加需要调试的进程

4》进入调试页面之后，通过Ctrl+S和G快捷键，定位到需要调试的关键函数，进行下断点

5》点击运行或者快捷键F9，触发程序的关键函数，然后进入断点，使用F8单步调试，F7单步跳入调试，在调试的过程中主要观察BL，BLX指令，以及CMP和CBZ等比较指令，然后在查看具体的寄存器的值。

2、IDA调试有反调试的so代码步骤：

1》查看apk是否为可调式状态，可以使用aapt命令查看他的AndroidManifest.xml文件中的android:debuggeable属性是否为true，如果不是debug状态，那么就需要手动的添加这个属性，然后回编译，在签名打包从新安装

2》使用adb shell am start -D -n com.yaotong.crackme/.MainActivity 命令启动程序，出于wait Debug状态

3》打开IDA，进行进程附加，进入到调试页面

4》使用 jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700 命令attach之前的debug状态，让程序正常运行

5》设置Debug Option选项，设置Suspend on library start/exit/Suspend on library load/unload/Suspend on process entry point选项

6》点击运行按钮或者F9键，程序运行停止在linker模块中，这时候表示so文件加载进来了，我们通过Ctrl+S和G键跳转到JNI_OnLoad函数出，进行下断点

7》然后继续运行，进入JNI_OnLoad断点处，使用F8进行单步调试，F7进行单步跳入调试，找到反调试代码处

8》然后使用二进制软件修改反调试代码为nop指令，即00值

9》修改之后，在替换原来的so文件，进行回编译，从新签名打包安装即可

10》按照上面的无反调试的so代码步骤即可

第四、学习了如何做到反调试检测

现在很多应用防止别的进程调试或者注入，通常会用自我检测装置，原理就是：

循环检测/proc/[mypid]/status文件，查看他的TracerPid字段是否为0，如果不为0，表示被其他进程trace了

那么这时候就直接退出程序。因为现在的IDA调试时需要进程的注入，进程注入现在都是使用Linux中的ptrace机制，那么这里的TracePid就可以记录trace的pid，我们可以发现我们的程序被那个进程注入了，或者是被他在调试。进而采取一些措施。

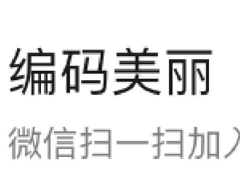
第五、IDA调试的整体原理

我们知道了上面的IDA调试步骤，其实我们可以仔细想一想，他的调试原理大致是这样的：

首先他得在被调试端安放一个程序，用于IDA端和调试设备通信，这个程序就是android_server，因为要附加进程，所以这个程序必须要用root身份运行，这个程序起来之后，就会开启一个端口23946，我们在使用adb forward进行端口转发到远程调试端，这时候IDA就可以和调试端的android_server进行通信了。后面获取设备的进程列表，附加进程，传递调试信息，都可以使用这个通信机制完成即可。IDA可以获取被调试的进程的内存数据，一般是在 **/proc/[pid]maps** 文件中，所以我们在使用Ctrl+S可以查看所有的so文件的基地址，可以遍历maps文件即可做到。

破解法则：时刻需要注意关键的BL/BLX等跳转指令，在他们执行完之后，肯定会有一些CMP/CBZ等比较指令，这时候就可以查看重要的寄存器内容来获取重要信息。

本文的目的只有一个就是学习更多的逆向技巧和思路，如果有人利用本文技术去进行非法商业获取利益带来的法律责任都是操作者自己承担，和本文以及作者没关系，本文涉及到的代码项目可以去编码美丽小密圈自取，欢迎加入小密圈一起学习探讨技术



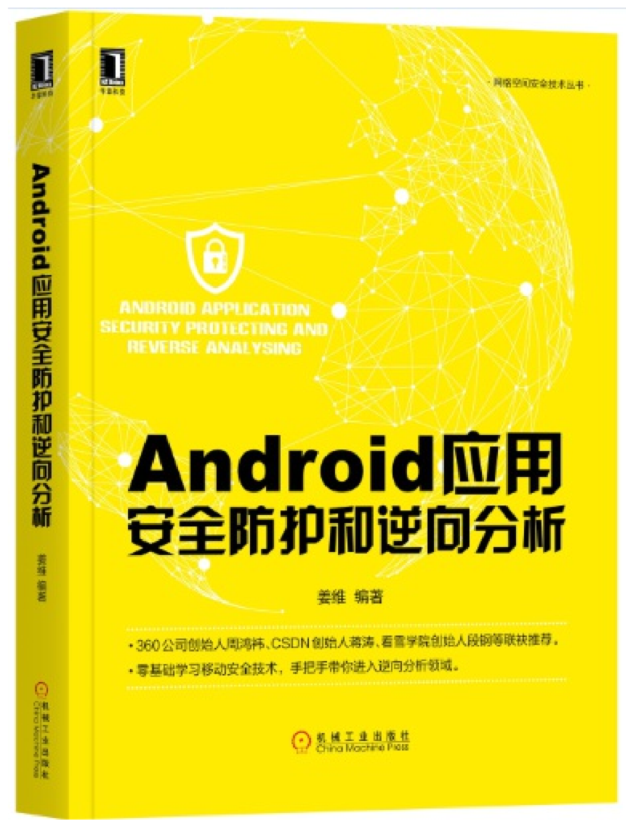
知识星球

七、总结

总算是说完了IDA调试so了这个知识点，我们也知道了一种全新的方式去破解native层的代码，现在有些程序依然把关键代码放在了Java层，那么这里我们可以使用Eclipse调试samli即可破解，如果程序为了安全，可能还会把关键代码放到native层，那么这时候，我们可以使用IDA来调试so代码来破解，当然破解和加密总是相生相克的，现在程序为了安全做了加固策略，那么这也是我们下一篇文章需要介绍的，如何去破解那些加固的apk。

《Android应用安全防护和逆向分析》

点击立即购买：京东 天猫



更多内容：点击这里

关注微信公众号，最新技术干货实时推送