IDA Pro 菜鸟速参手册精简版

制作者：PeterDocter
制作时间：05/09/2016

逆向准备

IDA Pro 快捷键大全

Open Subviews
Names _____ Shift+F4
Functions _____ Shift+F3
Strings _____ Shift+F12
Segments _____ Shift+F7
Segment registers _____ Shift+F8
Signatures _____ Shift+F5
Type libraries _____ Shift+F11
Structures _____ Shift+F9
Enumerations _____ Shift+F10

Data Format Options
ASCII strings style _____ Alt+A
Setup data types _____ Alt+D
File Operations
Parse C header file _____ Ctrl+F9
Create ASM file _____ Alt+F10
Save database _____ Ctrl+W

Navigation
Jump to operand _____ Enter
Jump in new window _____ Alt+Enter
Jump to previous position _____ Esc
Jump to next position _____ Ctrl+Enter
Jump to address _____ G
Jump by name _____ Ctrl+L

Jump to function _____ Ctrl+P
Jump to segment _____ Ctrl+S
Jump to segment register _____ Ctrl+G
Jump to problem _____ Ctrl+Q
Jump to cross reference _____ Ctrl+X
Jump to xref to operand _____ X
Jump to entry point _____ Ctrl+E
Mark Position _____ Alt+M
Jump to marked position _____ Ctrl+M

Debugger
Start process _____ F9
Terminate process _____ Ctrl+F2
Step into _____ F7
Step over _____ F8
Run until return _____ Ctrl+F7
Run to cursor _____ F4

Breakpoints
Breakpoint list _____ Ctrl+Alt+B

Watches
Delete watch _____ Del

Tracing
Stack trace _____ Ctrl+Alt+S

Search
Next code _____ Alt+C
Next data _____ Ctrl+D
Next explored _____Ctrl+A
Next unexplored _____ Ctrl+U
Immediate value _____ Alt+I
Next immediate value _____ Ctrl+I
Text _____ Alt+T
Next text _____ Ctrl+T
Sequence of bytes _____ Alt+B

Next sequence of bytes _____ Ctrl+B
Not function _____ Alt+U
Next void _____ Ctrl+V
Error operand _____ Ctrl+F

Graphing
Flow chart _____ F12
Function calls _____ Ctrl+F12

Miscellaneous
Calculator _____ ?
Cycle through open views _____ Ctrl+Tab
Select tab _____ Alt + [1…N]
Close current view _____ Ctrl+F4
Exit _____ Alt+X
IDC Command _____ Shift+F2

Edit (Data Types - etc)
Copy _____ Ctrl+Ins
Begin selection _____ Alt+L
Manual instruction _____ Alt+F2
Code _____ C
Data _____ D
Struct variable _____ Alt+Q
ASCII string _____ A
Array _____ Num *
Undefine _____ U
Rename _____ N

Operand Type
Offset (data segment) _____ O
Offset (current segment) _____ Ctrl+O
Offset by (any segment) _____ Alt+R
Offset (user-defined) _____ Ctrl+R
Offset (struct) _____ T
Number (default) _____ #
Hexadecimal _____ Q
Decimal _____ H
Binary _____ B

Character _____ R
Segment _____ S
Enum member _____ M
Stack variable _____ K
Change sign _____ Underscore (_)
Bitwise negate _____ ~
Manual _____ Alt+F1

Comments
Enter comment _____ :
Enter repeatable comment _____ ;
Enter anterior lines _____ Ins
Enter posterior lines _____ Shift+Ins
Insert predefined comment _____ Shift+F1

Segments
Edit segment _____ Alt+S
Change segment register value _____ Alt+G

Structs
Struct var _____ Alt+Q
Force zero offset field _____ Ctrl+Z
Select union member _____ Alt+Y

Functions
Create function _____ P
Edit function _____ Alt+P
Set function end _____ E
Stack variables _____ Ctrl+K
Change stack pointer _____ Alt+K
Rename register _____ V
Set function type _____ Y

Datarescue
Interactive Disassembler (IDA) Pro
Quick Reference Sheet
(http://www.datarescue.com)

IDA Pro plug-in defines

```
/*

   This file contains definitions used by the Hex-Rays decompiler output.
   It has type definitions and convenience macros to make the
   output more readable.

   Copyright (c) 2007-2011 Hex-Rays

*/

#if defined(__GNUC__)
  typedef          long long ll;
  typedef unsigned long long ull;
  #define __int64 long long
  #define __int32 int
  #define __int16 short
  #define __int8  char
  #define MAKELL(num) num ## LL
  #define FMT_64 "ll"
#elif defined(_MSC_VER)
  typedef          __int64 ll;
  typedef unsigned __int64 ull;
  #define MAKELL(num) num ## i64
  #define FMT_64 "I64"
#elif defined (__BORLANDC__)
  typedef          __int64 ll;
  typedef unsigned __int64 ull;
  #define MAKELL(num) num ## i64
  #define FMT_64 "L"
#else
  #error "unknown compiler"
#endif
typedef unsigned int uint;
typedef unsigned char uchar;
typedef unsigned short ushort;
typedef unsigned long ulong;
```

```
typedef          char   int8;
typedef   signed char   sint8;
typedef unsigned char   uint8;
typedef          short  int16;
typedef   signed short  sint16;
typedef unsigned short  uint16;
typedef          int    int32;
typedef   signed int    sint32;
typedef unsigned int    uint32;
typedef ll              int64;
typedef ll              sint64;
typedef ull             uint64;

// Partially defined types:
#define _BYTE  uint8
#define _WORD  uint16
#define _DWORD uint32
#define _QWORD uint64
#if !defined(_MSC_VER)
#define _LONGLONG __int128
#endif
```

```
#ifndef _WINDOWS_
typedef int8 BYTE;
typedef int16 WORD;
typedef int32 DWORD;
typedef int32 LONG;
#endif
typedef int64 QWORD;
#ifndef __cplusplus
typedef int bool;       // we want to use bool in our C programs
#endif

// Some convenience macros to make partial accesses nicer
// first unsigned macros:
#define LOBYTE(x)   (*((_BYTE*)&(x)))   // low byte
#define LOWORD(x)   (*((_WORD*)&(x)))   // low word
#define LODWORD(x)  (*((_DWORD*)&(x)))  // low dword
#define HIBYTE(x)   (*((_BYTE*)&(x)+1))
#define HIWORD(x)   (*((_WORD*)&(x)+1))
#define HIDWORD(x)  (*((_DWORD*)&(x)+1))
#define BYTEn(x, n)   (*((_BYTE*)&(x)+n))
```

```
#define WORDn(x, n)   (*((_WORD*)&(x)+n))
#define BYTE1(x)   BYTEn(x,  1)         // byte 1 (counting from 0)
#define BYTE2(x)   BYTEn(x,  2)
#define BYTE3(x)   BYTEn(x,  3)
#define BYTE4(x)   BYTEn(x,  4)
#define BYTE5(x)   BYTEn(x,  5)
#define BYTE6(x)   BYTEn(x,  6)
#define BYTE7(x)   BYTEn(x,  7)
#define BYTE8(x)   BYTEn(x,  8)
#define BYTE9(x)   BYTEn(x,  9)
#define BYTE10(x)  BYTEn(x, 10)
#define BYTE11(x)  BYTEn(x, 11)
#define BYTE12(x)  BYTEn(x, 12)
#define BYTE13(x)  BYTEn(x, 13)
#define BYTE14(x)  BYTEn(x, 14)
#define BYTE15(x)  BYTEn(x, 15)
#define WORD1(x)   WORDn(x,  1)
#define WORD2(x)   WORDn(x,  2)         // third word of the object, unsigned
#define WORD3(x)   WORDn(x,  3)
#define WORD4(x)   WORDn(x,  4)
#define WORD5(x)   WORDn(x,  5)
#define WORD6(x)   WORDn(x,  6)
#define WORD7(x)   WORDn(x,  7)

// now signed macros (the same but with sign extension)
#define SLOBYTE(x)   (*((int8*)&(x)))
#define SLOWORD(x)   (*((int16*)&(x)))
#define SLODWORD(x)  (*((int32*)&(x)))
#define SHIBYTE(x)   (*((int8*)&(x)+1))
#define SHIWORD(x)   (*((int16*)&(x)+1))
#define SHIDWORD(x)  (*((int32*)&(x)+1))
#define SBYTEn(x, n)   (*((int8*)&(x)+n))
#define SWORDn(x, n)   (*((int16*)&(x)+n))
#define SBYTE1(x)   SBYTEn(x,  1)
#define SBYTE2(x)   SBYTEn(x,  2)
#define SBYTE3(x)   SBYTEn(x,  3)
#define SBYTE4(x)   SBYTEn(x,  4)
#define SBYTE5(x)   SBYTEn(x,  5)
#define SBYTE6(x)   SBYTEn(x,  6)
#define SBYTE7(x)   SBYTEn(x,  7)
#define SBYTE8(x)   SBYTEn(x,  8)
#define SBYTE9(x)   SBYTEn(x,  9)
```

```
#define SBYTE10(x)  SBYTEn(x, 10)
#define SBYTE11(x)  SBYTEn(x, 11)
#define SBYTE12(x)  SBYTEn(x, 12)
#define SBYTE13(x)  SBYTEn(x, 13)
#define SBYTE14(x)  SBYTEn(x, 14)
#define SBYTE15(x)  SBYTEn(x, 15)
#define SWORD1(x)   SWORDn(x,  1)
#define SWORD2(x)   SWORDn(x,  2)
#define SWORD3(x)   SWORDn(x,  3)
#define SWORD4(x)   SWORDn(x,  4)
#define SWORD5(x)   SWORDn(x,  5)
#define SWORD6(x)   SWORDn(x,  6)
#define SWORD7(x)   SWORDn(x,  7)


// Helper functions to represent some assembly instructions.

#ifdef __cplusplus

// Fill memory block with an integer value
inline void memset32(void *ptr, uint32 value, int count)
{
  uint32 *p = (uint32 *)ptr;
  for ( int i=0; i < count; i++ )
    *p++ = value;
}

// Generate a reference to pair of operands
template<class T>  int16 __PAIR__( int8  high, T low) { return ((( int16)high) << sizeof(high)*8) | uint8(low); }
template<class T>  int32 __PAIR__( int16 high, T low) { return ((( int32)high) << sizeof(high)*8) | uint16(low); }
template<class T>  int64 __PAIR__( int32 high, T low) { return ((( int64)high) << sizeof(high)*8) | uint32(low); }
template<class T> uint16 __PAIR__(uint8  high, T low) { return (((uint16)high) << sizeof(high)*8) | uint8(low); }
template<class T> uint32 __PAIR__(uint16 high, T low) { return (((uint32)high) << sizeof(high)*8) | uint16(low); }
template<class T> uint64 __PAIR__(uint32 high, T low) { return (((uint64)high) << sizeof(high)*8) | uint32(low); }

// rotate left
template<class T> T __ROL__(T value, uint count)
{
  const uint nbits = sizeof(T) * 8;
  count %= nbits;

  T high = value >> (nbits - count);
```

```
  value <<= count;
  value |= high;
  return value;
}

// rotate right
template<class T> T __ROR__(T value, uint count)
{
  const uint nbits = sizeof(T) * 8;
  count %= nbits;

  T low = value << (nbits - count);
  value >>= count;
  value |= low;
  return value;
}

// carry flag of left shift
template<class T> int8 __MKCSHL__(T value, uint count)
{
  const uint nbits = sizeof(T) * 8;
  count %= nbits;

  return (value >> (nbits-count)) & 1;
}

// carry flag of right shift
template<class T> int8 __MKCSHR__(T value, uint count)
{
  return (value >> (count-1)) & 1;
}

// sign flag
template<class T> int8 __SETS__(T x)
{
  if ( sizeof(T) == 1 )
    return int8(x) < 0;
  if ( sizeof(T) == 2 )
    return int16(x) < 0;
  if ( sizeof(T) == 4 )
    return int32(x) < 0;
  return int64(x) < 0;
```

```
}

// overflow flag of subtraction (x-y)
template<class T, class U> int8 __OFSUB__(T x, U y)
{
  if ( sizeof(T) < sizeof(U) )
  {
    U x2 = x;
    int8 sx = __SETS__(x2);
    return (sx ^ __SETS__(y)) & (sx ^ __SETS__(x2-y));
  }
  else
  {
    T y2 = y;
    int8 sx = __SETS__(x);
    return (sx ^ __SETS__(y2)) & (sx ^ __SETS__(x-y2));
  }
}

// overflow flag of addition (x+y)
template<class T, class U> int8 __OFADD__(T x, U y)
{
  if ( sizeof(T) < sizeof(U) )
  {
    U x2 = x;
    int8 sx = __SETS__(x2);
    return ((1 ^ sx) ^ __SETS__(y)) & (sx ^ __SETS__(x2+y));
  }
  else
  {
    T y2 = y;
    int8 sx = __SETS__(x);
    return ((1 ^ sx) ^ __SETS__(y2)) & (sx ^ __SETS__(x+y2));
  }
}

// carry flag of subtraction (x-y)
template<class T, class U> int8 __CFSUB__(T x, U y)
{
  int size = sizeof(T) > sizeof(U) ? sizeof(T) : sizeof(U);
  if ( size == 1 )
    return uint8(x) < uint8(y);
```

```
  if ( size == 2 )
    return uint16(x) < uint16(y);
  if ( size == 4 )
    return uint32(x) < uint32(y);
  return uint64(x) < uint64(y);
}

// carry flag of addition (x+y)
template<class T, class U> int8 __CFADD__(T x, U y)
{
  int size = sizeof(T) > sizeof(U) ? sizeof(T) : sizeof(U);
  if ( size == 1 )
    return uint8(x) > uint8(x+y);
  if ( size == 2 )
    return uint16(x) > uint16(x+y);
  if ( size == 4 )
    return uint32(x) > uint32(x+y);
  return uint64(x) > uint64(x+y);
}

#else
// The following definition is not quite correct because it always returns
// uint64. The above C++ functions are good, though.
#define __PAIR__(high, low) (((uint64)(high)<<sizeof(high)*8) | low)
// For C, we just provide macros, they are not quite correct.
#define __ROL__(x, y) __rotl__(x, y)      // Rotate left
#define __ROR__(x, y) __rotr__(x, y)      // Rotate right
#define __CFSHL__(x, y) invalid_operation // Generate carry flag for (x<<y)
#define __CFSHR__(x, y) invalid_operation // Generate carry flag for (x>>y)
#define __CFADD__(x, y) invalid_operation // Generate carry flag for (x+y)
#define __CFSUB__(x, y) invalid_operation // Generate carry flag for (x-y)
#define __OFADD__(x, y) invalid_operation // Generate overflow flag for (x+y)
#define __OFSUB__(x, y) invalid_operation // Generate overflow flag for (x-y)
#endif

// No definition for rcl/rcr because the carry flag is unknown
#define __RCL__(x, y)    invalid_operation // Rotate left thru carry
#define __RCR__(x, y)    invalid_operation // Rotate right thru carry
#define __MKCRCL__(x, y) invalid_operation // Generate carry flag for a RCL
#define __MKCRCR__(x, y) invalid_operation // Generate carry flag for a RCR
#define __SETP__(x, y)   invalid_operation // Generate parity flag for (x-y)
```

```
// In the decompilation listing there are some objects declarared as _UNKNOWN
// because we could not determine their types. Since the C compiler does not
// accept void item declarations, we replace them by anything of our choice,
// for example a char:

#define _UNKNOWN char

#ifdef _MSC_VER
#define snprintf _snprintf
#define vsnprintf _vsnprintf
#endif
```

IDA load file 导入 JNI.h 解析 JNI 函数

一、需求：

使用 IDA 导入 C/C++头文件，添加头文件中的结构体，使用此结构体中的函数替换反汇编中的偏移，是文件可读性更好！！！！

二、步骤：

步骤一：点击 IDAPro 菜单项“File->Load file->Parse c header file ” 选择 jni.h 头文件

步骤二：简单修改 jni.h ,注释第 27 行的#include<stdarg.h> ,还有将 1122 行的#define JNIEXPORT_attribute_((visibility("default"))) 改成 #define JNIEXPORT 修改完后可以成功导入 {文件保存在 C:\Program Files (x86)\IDA 6.6\tools}

步骤三：导入成功后把 jni.h 修改的地方 改回来 防止编译 NDK 出错。

步骤四：点击 IDA Pro 主界面上的“Structures”选项卡 然后按下 Insert 键打开“Create structure/union”对话框，点击界面上的"Add standard structure"按钮，在打开的结构体选择对话框中选择 JNINativeInterface 并点击 OK 返回，同理 JNIInvokeInterface 结构体也导入进来；

自此已经成功的将 jni.h 中的头文件添加到了 IDA 中，但是发现 F5 反汇编得到的代码还是偏移值，没有转换成 函数调用。

下面是自己的实验结果：

5. 打开 so 的 JNi_OnLoad( ) 函数，如下； 注意此处 JNI_OnLoade ( )函数的参数是__JavaVM*，如果不是_JavaVM* 请使用图二将其转换成_JavaVM* 。

```
.text:0000114C                                          ; "UnregisterNatives()"
.text:00001150                  EXPORT JNI_OnLoad
.text:00001150 ; signed int __fastcall JNI_OnLoad(_JavaVM *a1)
.text:00001150 JNI_OnLoad       DCB 0xD8 ;
.text:00001151                  DCB 0x10
.text:00001152                  DCB 0x9F ;
.text:00001153                  DCB 0xE5 ;
.text:00001154                  DCB    6
.text:00001155                  DCB 0x20
.text:00001156                  DCB    0
.text:00001157                  DCB 0xE3 ;
.text:00001158                  DCB 0x70 ; p
.text:00001159                  DCB 0x40 ; @
.text:0000115A                  DCB 0x2D ; -
.text:0000115B                  DCB 0xE9 ;
.text:0000115C                  DCB    1
.text:0000115D                  DCB 0x50 ; P
.text:0000115E                  DCB 0x9F ;
.text:0000115F                  DCB 0xE7 ;
.text:00001160                  DCB    1
.text:00001161                  DCB 0x20
.text:00001162                  DCB 0x40 ; @
.text:00001163                  DCB 0xE3 ;
.text:00001164                  DCB    0
.text:00001165                  DCB 0x30 ; 0
.text:00001166                  DCB 0x90 ;
.text:00001167                  DCB 0xE5 ;
.text:00001168                  DCB    5
.text:00001169                  DCB 0x10
.text:0000116A                  DCB 0xA0 ;
.text:0000116B                  DCB 0xE1 ;
.text:0000116C                  DCB  0xF
```


图一

使用 “C”将其转换成 code

```
.text:00001150
.text:00001150 ; signed int __fastcall JNI_OnLoad(_JavaVM *a1)
.text:00001150                 EXPORT JNI_OnLoad
.text:00001150 JNI_OnLoad
.text:00001150                 LDR             R1, =(g_env_ptr - 0x1164)
.text:00001154                 MOV             R2, #6
.text:00001158                 STMFD           SP!, {R4-R6,LR}
.text:0000115C                 LDR             R5, [PC,R1] ; g_env
.text:00001160                 MOVT            R2, #1
.text:00001164                 LDR             R3, [R0]
.text:00001168                 MOV             R1, R5
.text:0000116C                 MOV             LR, PC
.text:00001170                 LDR             PC, [R3,#0x18]
.text:00001174                 CMP             R0, #0
.text:00001178                 MOVNE           R0, #0xFFFFFFFF
.text:0000117C                 LDMNEFD         SP!, {R4-R6,PC}
.text:00001180                 LDR             R4, =(aCom_droider_jn - 0x1194)
.text:00001184                 MOV             R0, #2
.text:00001188                 LDR             R2, =(aJni_onload - 0x1198)
.text:0000118C                 ADD             R4, PC, R4 ; "com.droider.jnimethods"
.text:00001190                 ADD             R2, PC, R2 ; "JNI_OnLoad()"
.text:00001194                 MOV             R1, R4
.text:00001198                 BL              __android_log_print
.text:0000119C                 LDR             R3, [R5]
.text:000011A0                 LDR             R1, =(aComDroiderJnim - 0x11B0)
.text:000011A4                 MOV             R0, R3
.text:000011A8                 ADD             R1, PC, R1 ; "com/droider/jnimethods/TestJniMethods"
.text:000011AC                 LDR             R3, [R3]
.text:000011B0                 MOV             LR, PC
```


图二

6. 下面是反汇编的结果发现，函数已经出现了。

```
signed int __fastcall JNI_OnLoad(_JavaVM *a1)
{
  signed int result; // r0@2

  if ( (a1->functions->GetEnv)() )
  {
    result = -1;
  }
  else
  {
    _android_log_print(2, "com.droider.jnimethods", "JNI_OnLoad()");
    native_class = (*(*g_env + offsetof(JNINativeInterface, FindClass)))(g_env, "com/droider/jnimethods/TestJniMethods");
    if ( (*(*g_env + 860))(g_env, native_class, _data_start, 4) )
    {
      _android_log_print(6, "com.droider.jnimethods", "RegisterNatives() --> nativeMethod() failed");
      result = -1;
    }
    else
    {
      _android_log_print(2, "com.droider.jnimethods", "RegisterNatives() --> nativeMethod() ok");
      result = 65542;
    }
  }
  return result;
}
```


注意：上面是操作成功后的截图，可能参数，和函数都转换成功，下面重新打开 IDA 演示

---

7. 介绍 Java_com_droider_jnimethods_TestJniMethods_test 函数，解析其中调用的 JNI 函数。

8. 源代码中发现 ，函数第一个参数是 JNIEnv*，那么在返回编代码是什么呢？ 需要转换成什么？

```
54 JNIEXPORT void Java_com_droider_jnimethods_TestJniMethods_test(JNIEnv* env, jobject object) {
55     int version = (*env)->GetVersion(env);
56     LOGV("GetVersion() --> jni version:%2x", version);
57     jclass build_class = (*env)->FindClass(env, "android/os/Build");                    //加载一个本地类
58     jfieldID brand_id = (*env)->GetStaticFieldID(env,                                   //获取类的静态字段ID
59             build_class, "MODEL", "Ljava/lang/String;");                                /*field name,  field  type
60     jstring brand_obj = (jstring)(*env)->GetStaticObjectField(env,                      //获取类的静态字段的值
61             build_class, brand_id);
62     //__asm__ ("bkpt");
63     //raise(SIGTRAP);
64
65     const char *nativeString = (*env)->GetStringUTFChars(env, brand_obj, 0);            //通过jstring生成char*
66     LOGV("GetStringUTFChars() --> MODEL:%s", nativeString);
67     LOGV("GetStaticFieldID() --> %s", nativeString);
68     LOGV("ReleaseStringUTFChars() --> %s", nativeString);
69     (*env)->ReleaseStringUTFChars(env, brand_obj, nativeString);                        //释放GetStringUTFChars()生
70     jclass test_class = (*env)->FindClass(env, "com/droider/jnimethods/TestClass");
71     if((*env)->ExceptionCheck(env)){
72         (*env)->ExceptionDescribe(env);
73         (*env)->ExceptionClear(env);
74         LOGE("ExceptionCheck()");
75         LOGE("ExceptionDescribe()");
76         LOGE("ExceptionClear()");
77     }
78     jmethodID constructor = (*env)->GetMethodID(env, test_class, "<init>", "()V");      //获取构造函数
79     jobject obj = (*env)->NewObject(env, test_class, constructor);                      //创建一个对象
80     jthrowable throwable = (*env)->ExceptionOccurred(env);
```


返汇编得到函数：

9. 下图，第一个问题解决

```
1 int __fastcall Java_com_droider_jnimethods_TestJniMethods_test(int a1)
2 {
3   int v1; // r4@1
4   int v2; // r0@1
5   int v3; // r0@1
6   int v4; // r6@1
7   int v5; // r0@1
8   int v6; // r7@1
9   int v7; // r6@1
10  int v8; // r6@1
11  int v9; // r0@2
12  int v10; // r5@2
13  int v11; // r8@5
14  int v12; // r10@5
15  int v13; // r9@5
16  void (__fastcall *v14)(_DWORD, _DWORD, _DWORD, _DWORD); // r11@5
17  int v15; // r0@5
18  int v16; // r0@5
19  int v17; // r8@5
20  int v18; // r0@5
21  int v19; // r0@5
22  int v20; // r9@5
23  int v21; // r1@7
24  int v22; // r0@10
25  int v23; // r8@12
26  int v24; // r10@12
27  int v25; // r5@14
28  int v26; // r8@14
29  int v27; // r7@14
30  int v28; // r6@14
31  int v29; // r0@14
32  __int64 v30; // r0@14
33  __int64 v31; // ST00_8@14
34  int v32; // r0@14
35  int v33; // r8@14
36  int v34; // r0@14
```

参数是int， 显然不对

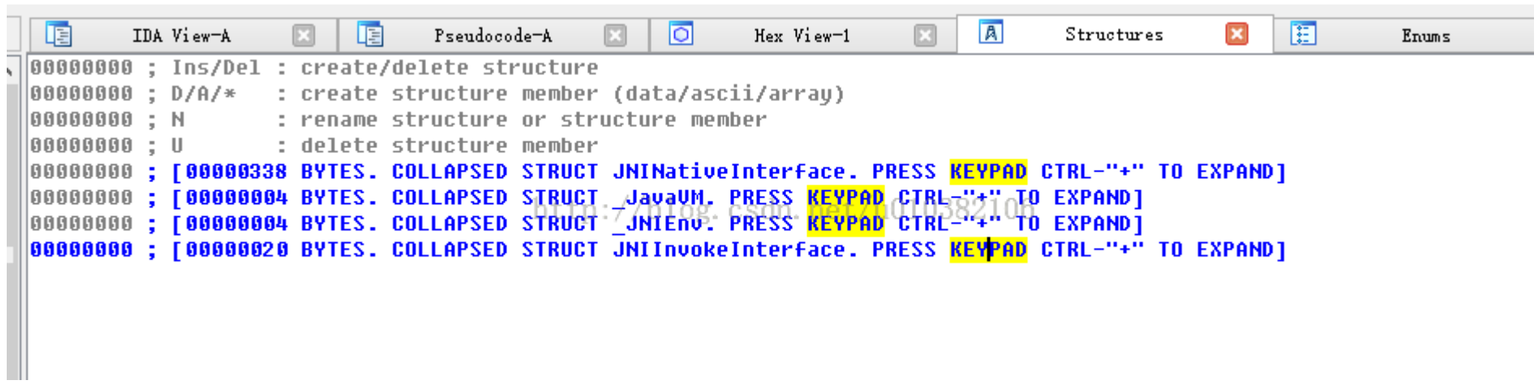
图一

```
int v65; // [sp+3Ch] [bp-2Ch]@14

v1 = a1;
v2 = (*(int (**)(void))(*(_DWORD *)a1 + 16))();
_android_log_print(2, "com.droider.jnimethods", "GetVersion() --> jni version:%2x", v2);
v3 = (*(int (__fastcall **)(int, _DWORD))(*(_DWORD *)v1 + 24))(v1, "android/os/Build");
v4 = v3;
v5 = (*(int (__fastcall **)(int, int, _DWORD, _DWORD))(*(_DWORD *)v1 + 576))(v1, v3, "MODEL", "Ljava/lang/String;");
v6 = (*(int (__fastcall **)(int, int, int))(*(_DWORD *)v1 + 580))(v1, v4, v5);
v7 = (*(int (__fastcall **)(int, int, _DWORD))(*(_DWORD *)v1 + 676))(v1, v6, 0);
_android_log_print(2, "com.droider.jnimethods", "GetStringUTFChars() --> MODEL:%s", v7);
_android_log_print(2, "com.droider.jnimethods", "GetStaticFieldID() --> %s", v7);
_android_log_print(2, "com.droider.jnimethods", "ReleaseStringUTFChars() --> %s", v7);
(*(void (__fastcall **)(int, int, int))(*(_DWORD *)v1 + 680))(v1, v6, v7);
v8 = (*(int (__fastcall **)(int, _DWORD))(*(_DWORD *)v1 + 24))(v1, "com/droider/jnimethods/TestClass");
if ( (*(int (__fastcall **)(int))(*(_DWORD *)v1 + 912))(v1) )
{
  (*(void (__fastcall **)(int))(*(_DWORD *)v1 + 64))(v1);
  (*(void (__fastcall **)(int))(*(_DWORD *)v1 + 68))(v1);
  _android_log_print(6, "com.droider.jnimethods", "ExceptionCheck()");
  _android_log_print(6, "com.droider.jnimethods", "ExceptionDescribe()");
  _android_log_print(6, "com.droider.jnimethods", "ExceptionClear()");
}
v9 = (*(int (__fastcall **)(int, int, _DWORD, char[4]))(*(_DWORD *)v1 + 132))(v1, v8, "<init>", "()V");
v10 = (*(int (__fastcall **)(int, int, int))(*(_DWORD *)v1 + 112))(v1, v8, v9);
if ( (*(int (__fastcall **)(int))(*(_DWORD *)v1 + 60))(v1) )
{
  (*(void (__fastcall **)(int))(*(_DWORD *)v1 + 64))(v1);
  (*(void (__fastcall **)(int))(*(_DWORD *)v1 + 68))(v1);
  _android_log_print(6, "com.droider.jnimethods", "ExceptionOccurred()");
}
if ( v10 )
{
  (*(void (__fastcall **)(int, int))(*(_DWORD *)v1 + 868))(v1, v10);
```

发现参数 a1（JNIEnv调用函数，目前还是通过偏移计算函数地址，这样阅读返汇编代码很不方便，所以要让他转化成C/C++ 函数调用）

图二

注意： 在这应该 将 jni.h 导入进来，并插入到 structures 窗口。 下图

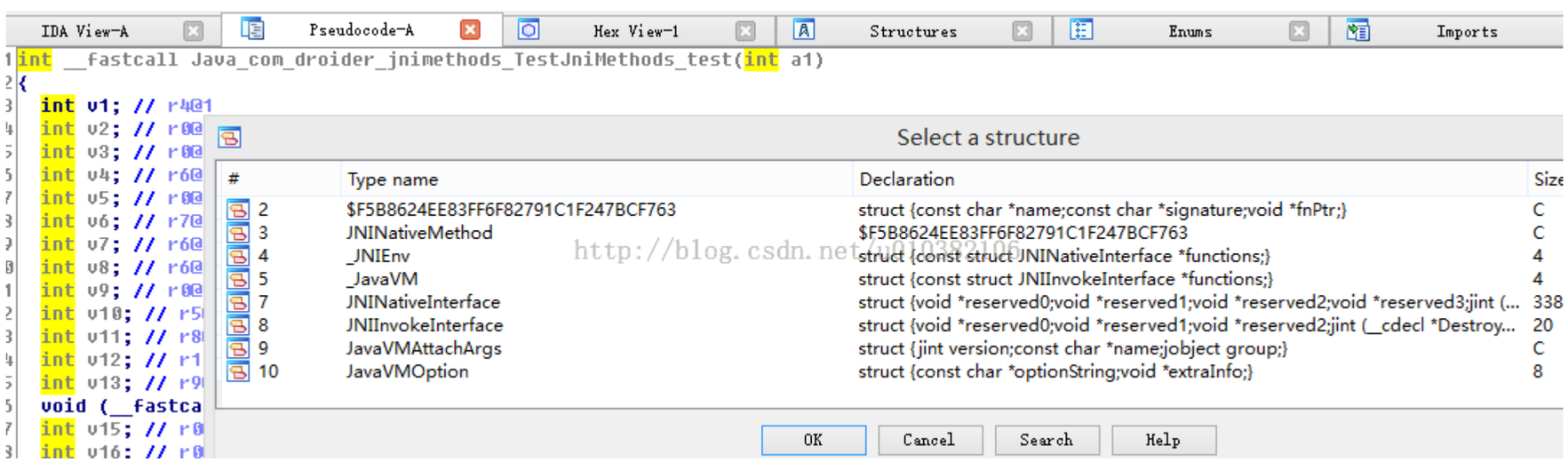


10. 第二个问题：我们应该转换 __JNIEnv，可以看到

```
4 _JNIEnv struct {const struct JNINativeInterface *functions;} 4
```

_JNIEnv 内部就是 JNINativeInterface ，但是不能转换成 JNINativeInterface ，因为在堆栈和内存中记录的是_JNIEnv 的地址，JNINativeInterface 的地址只是用来引用。

注意：转换的方法是，鼠标点击参数，然后右键选中 Convert to Struct *



结束了！

成功了！！！！！！

```
46  int v50; // r0@16
47  int v51; // r0@16
48  int v52; // r9@16
49  int v53; // r11@16
50  const struct JNINativeInterface *v54; // r12@16
51  int v55; // r0@16
52  int v56; // r7@16
53  int v57; // r0@16
54  int v58; // r0@16
55  int v59; // r0@16
56  int v60; // [sp+18h] [bp-50h]@14
57  __int64 v61; // [sp+1Ch] [bp-4Ch]@14
58  int v62; // [sp+24h] [bp-44h]@14
59  int v63; // [sp+28h] [bp-40h]@14
60  int v64; // [sp+2Ch] [bp-3Ch]@14
61  int v65; // [sp+30h] [bp-38h]@14
62  int v66; // [sp+34h] [bp-34h]@16
63  int v67; // [sp+38h] [bp-30h]@16
64  int v68; // [sp+3Ch] [bp-2Ch]@14
65
66  v1 = a1;
67  v2 = (a1->functions->GetVersion)();
68  _android_log_print(2, "com.droider.jnimethods", "GetVersion() --> jni version:%2x", v2);
69  v3 = (v1->functions->FindClass)(v1, "android/os/Build");
70  v4 = v3;
71  v5 = (v1->functions->ReleaseStringUTFChars)(v1, v3, "MODEL", "Ljava/lang/String;");
72  v6 = (v1->functions->GetArrayLength)(v1, v4, v5);
73  v7 = (v1->functions->ReleaseIntArrayElements)(v1, v6, 0);
74  _android_log_print(2, "com.droider.jnimethods", "GetStringUTFChars() --> MODEL:%s", v7);
75  _android_log_print(2, "com.droider.jnimethods", "GetStaticFieldID() --> %s", v7);
76  _android_log_print(2, "com.droider.jnimethods", "ReleaseStringUTFChars() --> %s", v7);
77  (v1->functions->ReleaseLongArrayElements)(v1, v6, v7);
78  v8 = (v1->functions->FindClass)(v1, "com/droider/jnimethods/TestClass");
79  if ( (v1->functions[1].DeleteGlobalRef)(v1) )
80  {
81    (v1->functions->ExceptionDescribe)(v1);
82    (v1->functions->ExceptionClear)(v1);
```

Select a structure

#	Type name	Declaration	Size
2	$F5B8624EE83FF6F82791C1F247BCF763	struct {const char *name;const char *signature;void *fnPtr;}	C
3	JNINativeMethod	$F5B8624EE83FF6F82791C1F247BCF763	C
4	_JNIEnv	struct {const struct JNINativeInterface *functions;}	4
5	_JavaVM	struct {const struct JNIInvokeInterface *functions;}	4
7	JNINativeInterface	struct __attribute__((aligned(2))) {_JNIEnv *reserved0;void *reserved1;void *res...	338
8	JNIInvokeInterface	struct {void *reserved0;void *reserved1;void *reserved2;jint (__cdecl *Destroy...	20
9	JavaVMAttachArgs	struct {jint version;const char *name;jobject group;}	C
10	JavaVMOption	struct {const char *optionString;void *extraInfo;}	8

OK Cancel Search Help

Line 3 of 8

http://blog.csdn.net/u010382106/article/details/44960243

IDA Pro 导入 jni.h 头文件定义

为了逆向备份一份出来导入

步骤一：

点击 IDA Pro 菜单项“File->Load file->Parse c header file ” 选择 jni.h 头文件

步骤二：

简单修改 jni.h ,注释第 27 行的#include<stdarg.h> ,还有将 1122 行的#define JNIEXPORT_attribute_((visibility("default"))) 改成 #define JNIEXPORT 修改完后可以成功导入

步骤三：

导入成功后把 jni.h 修改的地方 改回来 防止编译 NDK 出错。

步骤四：
Windows-> Structures
点击 IDA Pro 主界面上的“Structures”选项卡 然后按下 Insert 键打开“Create structure/union”对话框，点击界面上的"Add standard structure"按钮，在打开的结构体选择对话框中选择 JNINativeInterface 并点击 OK 返回，同理 JNIInvokeInterface 结构体也导入进来
http://blog.csdn.net/nightstarsoul/article/details/28093311
非虫

6.3 以上不需要导入 jni.h 头文件，直接点 a1 上按 u 这样 JNIEnv*就直接解析了

IDA 中 F5 功能反编译安卓平台的 so 文件参数问题

http://bbs.pediy.com/showthread.php?t=198729

Android 调用 JNI 本地方法跟踪目标代码(导文件头)

正如 Android 调用 JNI 本地方法经过有点改变章所说跟踪代码是可行的，但是跟踪某些代码会出现 anr，点击取消，还是不好运，有提高办法吗？回答是有（gdb 还没试过，本文只讨论 ida）。

下面是我使用 0 * Message("%s = %d\n", GetString(Dword(R2+0x10),-1, ASCSTR_C), R2+0x20)打出的记录

```
"unlockCanvasAndPost"
"native_computeBounds"
"lockCanvasNative"
"nativeDraw"
"native_getClipBounds"
"native_drawText"
"nativeDraw"
"unlockCanvasAndPost"
"native_computeBounds"
"lockCanvasNative"
"nativeDraw"
"native_getClipBounds"
```

```
"native_drawText"
"nativeDraw"

....
```

反复调用然后 anr 了。

为了改善这种情况。经过仔细查阅 IDA 文档 Edit breakpoint 一章，发现

```
  Low level condition:
        Evaluate the condition on the remote computer. Such conditions are
        faster, especially during remote debugging, because there is no
        network traffic between IDA and the remote computer on each
        breakpoint hit. More details
```

低级条件，在远程计算机计算条件。这种条件运行更快，特别是在远程调试的时候。详细内容如下：

Low level breakpoint conditions

Low level breakpoint conditions can be used to speed up the debugger. They are evaluated like this:

- for remote debugging, such a condition is evaluated on the remote computer. The following actions are bypassed:
  - copying the breakpoint event to the local computer
  - switching from debthread to the main thread
  - updating internal IDA structures and caches
  - updating the screen
- for local debugging, such a condition is evaluated at low level. The following actions are bypassed:
  - switching from debthread to the main thread
  - updating internal IDA structures and caches
  - updating the screen

In both cases, there is a significant speed up. This improvement imposes some limitations on the breakpoint condition:

- only IDC expressions can be used for low level conditions
- only functions marked as 'thread-safe' may be called
- only entire registers can be accessed (e.g. EAX is ok but AL is not)

Essentially this means that the only available functions are:

  - read/write process registers
  - read/write process memory
  - file i/o
  - auxiliary string and object functions
  - Message() function (for debugging the breakpoint conditions)

Low level breakpoint conditions are available only for Win32, WinCE, Linux, Mac, Android debuggers.

从中看到对我有影响的就是使用的函数必须带有’thread-safe’字样提示。

  诸如前文使用的

```
0 * print(GetString(DbgDword(R2+0x10),-1, ASCSTR_C))
```

```
"method" == GetString(DbgDword(R2+0x10),-1, ASCSTR_C)
```

```
0 * Message("%s = %d\n", GetString(DbgDword(R2+0x10),-1, ASCSTR_C), R2+0x20)
```

其中 DbgDword 就是线程安全的，而 Dword 就不是，如此

DbgDword

```
// Get value of program double word (4 bytes) using the debugger memory
//      ea - linear address
// returns: the value of the double word. Throws an exception on failure.
// Thread-safe function (may be called only from the main thread and debthread)

long DbgDword (long ea);
```

表达式中另一个函数也不行

GetString

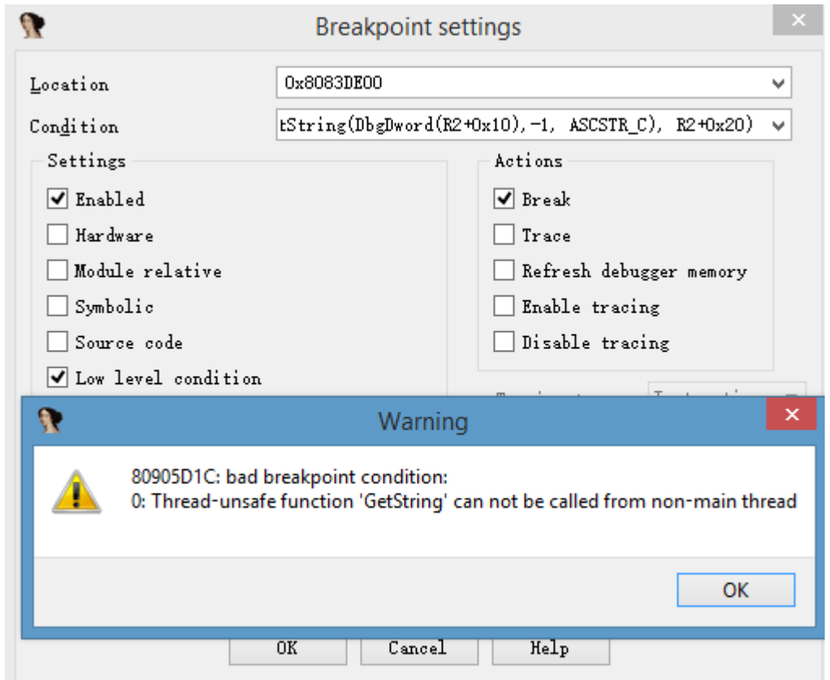
```
// Get string contents
//      ea   - linear address
//      len  - string length. -1 means to calculate the max string length
//      type - the string type (one of ASCSTR_... constants)
```

```
// Returns: string contents or empty string

string GetString(long ea, long len, long type);
```

See also GetStringType function.

就没有

所以报错，不允许。



为了找到替代，找到一大圈无果。直到一个一个比较，先比较前几个吧如案例

可以在这个论坛下载 2014 攻防对抗挑战赛

```
#include <sys/types.h>
#include <signal.h>
```

```
#include <stdio.h>
#include <unistd.h>
#include <dlfcn.h>
#include <string.h>
#include <errno.h>

/*
    package com.crackme;
    public class MainActivity{
        private native String crackme(String paramString1, String paramString2);
    }
```

Native 的对应函数名要以“Java_”开头，后面依次跟上 Java 的“package 名”、“class 名”、“函数名”，中间以下划线“_” 分割，在 package 名中的“.”也要改为“_”。

此外，关于函数的参数和返回值也有相应的规则。对于 Java 中的基本类型如 int 、double 、char 等，在 Native 端都有相对应的类型来表示，如 jint 、jdouble 、jchar 等；其他的对象类型则统统由 jobject 来表示（String 是个例外，由于其使用广泛，故在 Native 代码中有 jstring 这个类型来表示，正如在上例中返回值 String 对应到 Native 代码中的返回值 jstring ）。而对于 Java 中的数组，在 Native 中由 jarray 对应，具体到基本类型和一般对象类型的数组则有 jintArray 等和 jobjectArray 分别对应（String 数组在这里没有例外，同样用 jobjectArray 表示）。

还有一点需要注意的是，在 JNI 的 Native 函数中，其前两个参数 JNIEnv *和 jobject 是必需的——前者是一个 JNIEnv 结构体的指针是 JNI 的核心数据，这个结构体中定义了很多 JNI 的接口函数指针，使开发者可以使用 JNI 所定义的接口功能；后者指代的是调用这个 JNI 函数的 Java 对象，有点类似于 C++中的 this 指针。在上述两个参数之后，还需要根据 Java 端的函数声明依次对应添加参数。

在上例中，Java 中声明的 JNI 函数对应命名为：

```
    //Class:     com_crackme_MainActivity
    //Method:    crackme
    //Signature: (Ljava/lang/String;)Ljava/lang/String;
    jstring Java_com_crackme_MainActivity_crackme(JNIEnv *,jobject,jstring,jstring);


jstring (*crackme)(JNIEnv *,jobject,jstring, jstring) = NULL;
//事先把 libcrackme.so 放到 root/system/lib/目录下
void *filehandle = dlopen("/system/lib/libcrackme.so", RTLD_LAZY);
//(jstring (*)(JNIEnv *,jobject, jstring, jstring))
if(filehandle)
{
    crackme = (jstring (*)(JNIEnv *,jobject, jstring, jstring))dlsym(filehandle, "Java_com_crackme_MainActivity_crackme");
    if(crackme){
        jstring s = crackme(env, obj, a, b);
    }
    dlclose(filehandle);
    filehandle = NULL;
}
*/
```

```
typedef void *CRACKME;
//typedef jstring *CRACKME(JNIEnv *,jobject, jstring, jstring);



int main(int argc, char **argv)
{
    CRACKME *crackme;
    int i = 0;
    void *handle;

    handle = dlopen("/home/Sansan/a/libcrackme.so", RTLD_LAZY);
    if (!handle) {
        printf("%s, %d, NULL == handle. errno = %d, %s\n", __FUNCTION__, __LINE__, errno, strerror(errno));
        return -1;
    }
    crackme = dlsym(handle, "JNI_OnLoad");
    if (!crackme) {
        printf("%s, %d, NULL == crackme\n", __FUNCTION__, __LINE__);
        return -1;
    }
    printf("%s, %d, crackme = %p\n", __FUNCTION__, __LINE__, crackme);
    dlclose(handle);
    return 0;
}
```

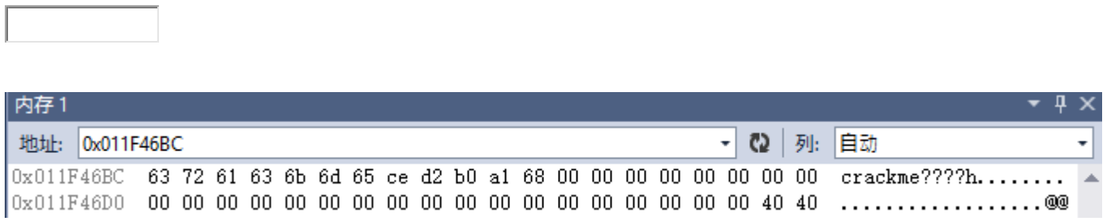
条件语句类似这样

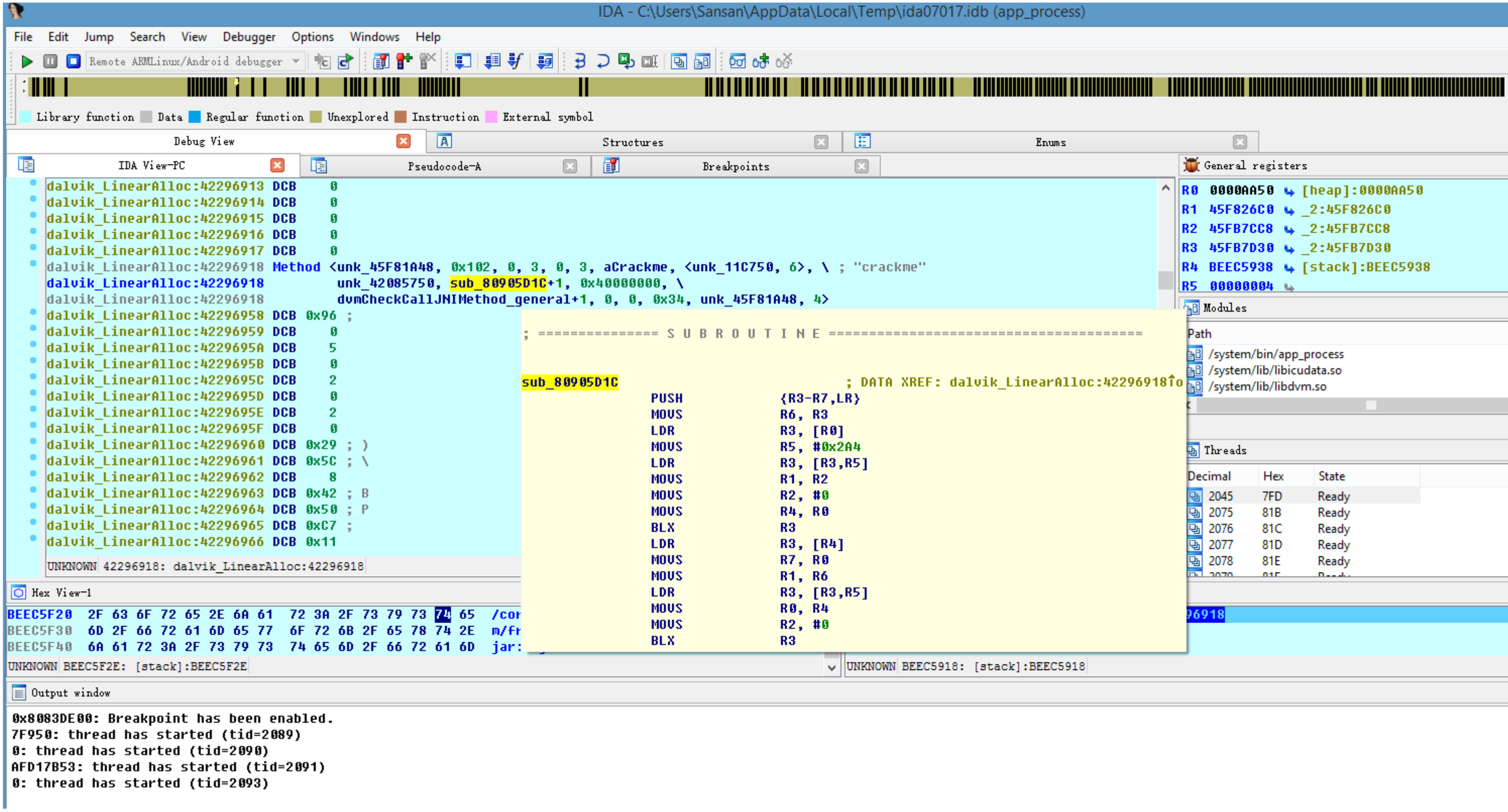
```
'c' == DbgByte(DbgDword(R2+0x10)) && 'r' == DbgByte(1+DbgDword(R2+0x10)) && 'a' == DbgByte(2+DbgDword(R2+0x10)) && 'c' == DbgByte(3+DbgDword(R2+0x10)) && 'k' == DbgByte(4+DbgDword(R2+0x10))
```

char *crac = "crac" 是 4 字节可以计算成一个 4 字节的整数，可能更好，类似这样计算出它在内存中的值 0x63617263。

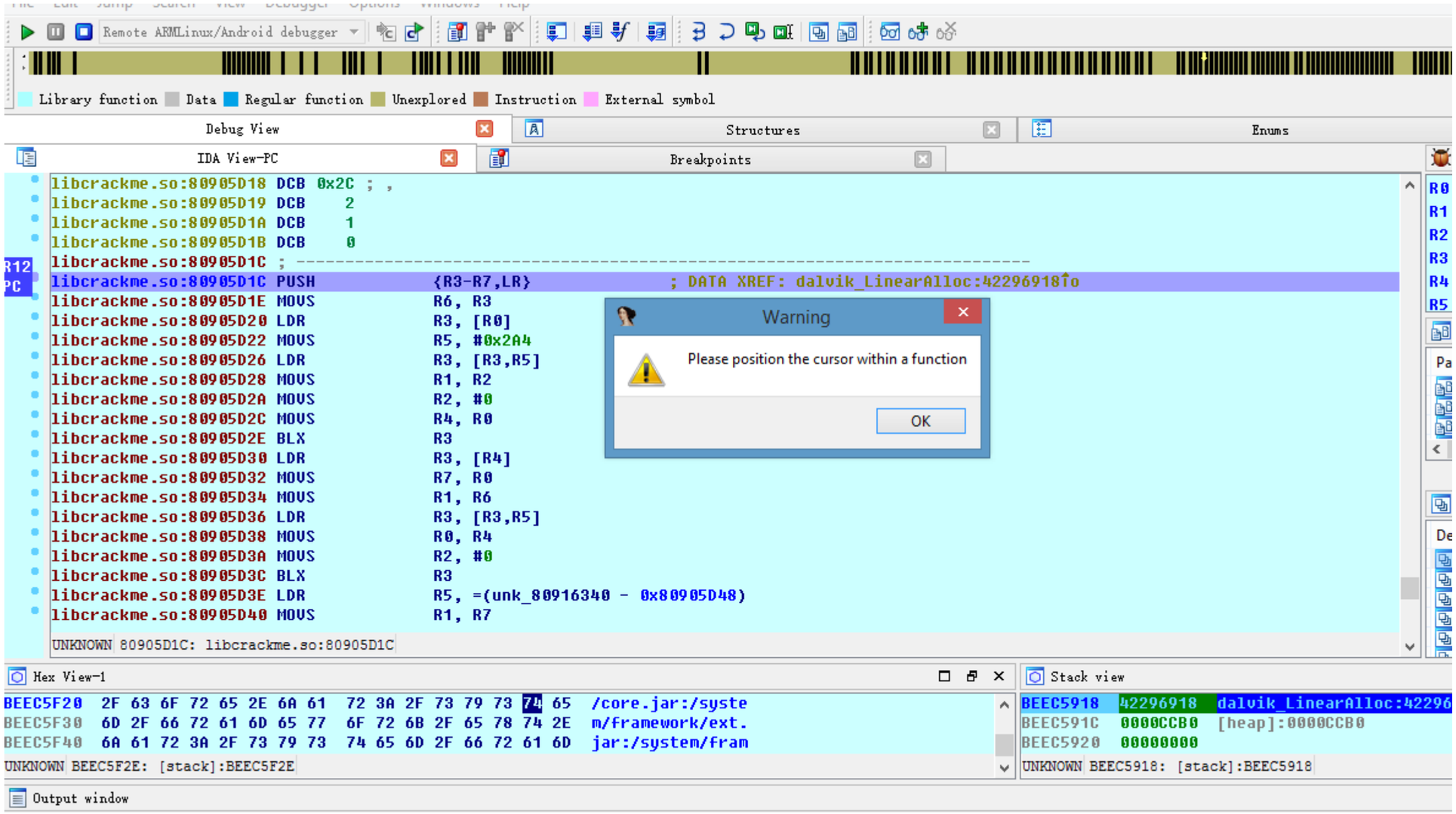
```
    ;int crac = *(int*)"crackme 我啊 h";
011EF305  mov         eax,dword ptr ds:[011F46BCh]
011EF30A  mov         dword ptr [crac],eax
```

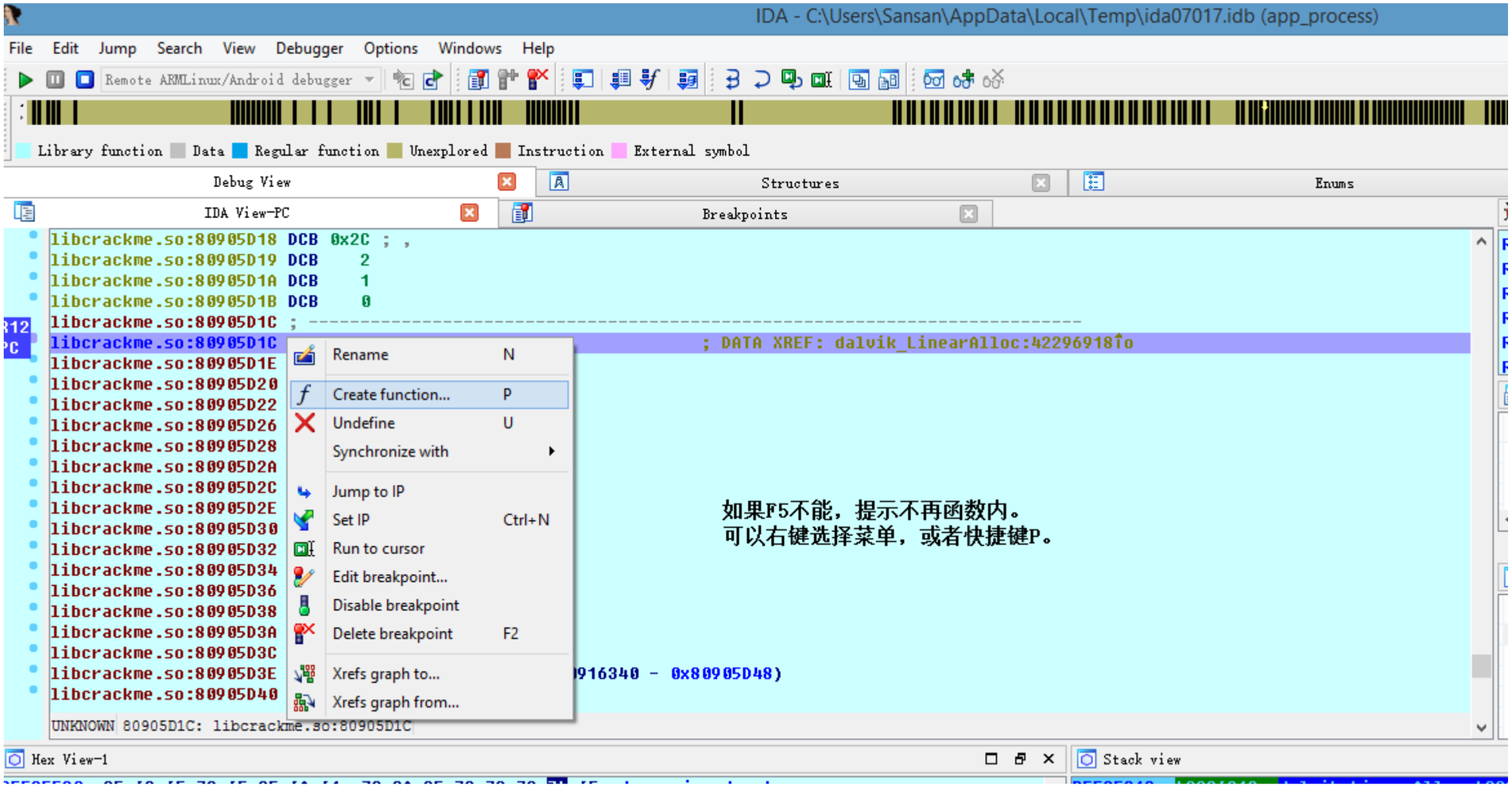
crackme


内存 1
地址: 0x011F46BC
列: 自动
0x011F46BC 63 72 61 63 6b 6d 65 ce d2 b0 a1 68 00 00 00 00 00 00 00 00 crackme????h.....
0x011F46D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 40 40 ....@@



跳到目标，如果 F5 不行需要弄一下。


Remote ARMLinux/Android debugger
Library function
Data
Regular function
Unexplored
Instruction
External symbol
Debug View
Structures
Enums
IDA View-PC
Breakpoints
libcrackme.so:80905D18 DCB 0x2C ; ,
libcrackme.so:80905D19 DCB 2
libcrackme.so:80905D1A DCB 1
libcrackme.so:80905D1B DCB 0
libcrackme.so:80905D1C ; ---
libcrackme.so:80905D1C PUSH {R3-R7,LR} ; DATA XREF: dalvik_LinearAlloc:42296918↑o
libcrackme.so:80905D1E MOVS R6, R3
libcrackme.so:80905D20 LDR R3, [R0]
libcrackme.so:80905D22 MOVS R5, #0x2A4
libcrackme.so:80905D26 LDR R3, [R3,R5]
libcrackme.so:80905D28 MOVS R1, R2
libcrackme.so:80905D2A MOVS R2, #0
libcrackme.so:80905D2C MOVS R4, R0
libcrackme.so:80905D2E BLX R3
libcrackme.so:80905D30 LDR R3, [R4]
libcrackme.so:80905D32 MOVS R7, R0
libcrackme.so:80905D34 MOVS R1, R6
libcrackme.so:80905D36 LDR R3, [R3,R5]
libcrackme.so:80905D38 MOVS R0, R4
libcrackme.so:80905D3A MOVS R2, #0
libcrackme.so:80905D3C BLX R3
libcrackme.so:80905D3E LDR R5, =(unk_80916340 - 0x80905D48)
libcrackme.so:80905D40 MOVS R1, R7
UNKNOWN 80905D1C: libcrackme.so:80905D1C
Warning
Please position the cursor within a function
OK
R0
R1
R2
R3
R4
R5
R12
PC
Hex View-1
BEEC5F20 2F 63 6F 72 65 2E 6A 61 72 3A 2F 73 79 73 74 65 /core.jar:/syste
BEEC5F30 6D 2F 66 72 61 6D 65 77 6F 72 6B 2F 65 78 74 2E m/framework/ext.
BEEC5F40 6A 61 72 3A 2F 73 79 73 74 65 6D 2F 66 72 61 6D jar:/system/fram
UNKNOWN BEEC5F2E: [stack]:BEEC5F2E
Stack view
BEEC5918 42296918 dalvik_LinearAlloc:42296
BEEC591C 0000CCB0 [heap]:0000CCB0
BEEC5920 00000000
UNKNOWN BEEC5918: [stack]:BEEC5918
Output window



最后 F5 结果，有神奇 F5 就是容易点啊，虽说君子善假于物也，依靠工具会产生惰性，分析汇编能力会下降。

```
int __fastcall sub_80905D1C(int a1, int a2, int a3, int a4)
{
  int v4; // r6@1
  int v5; // r4@1
  int v6; // r7@1
  int v7; // r0@1
```

```
  v4 = a4;
  v5 = a1;
  v6 = (*(int (**)(void))(*(_DWORD *)a1 + 676))();
  v7 = (*(int (__fastcall **)(int, int, _DWORD))(*(_DWORD *)v5 + 676))(v5, v4, 0);
  ((void (__fastcall *)(_UNKNOWN *, int, int))sub_809055F8)(&"Failure", v6, v7);
  ((void (__fastcall *)(_UNKNOWN *))sub_80905C44)(&"Failure");
  return (*(int (__fastcall **)(int, _UNKNOWN *))(*(_DWORD *)v5 + 668))(v5, &"Failure");
}
```

```
int __fastcall sub_809055F8(int a1, int a2, int a3)
{
  int v3; // r6@1
  int v4; // r4@1
  int v5; // r5@1
  int result; // r0@1
  int v7; // r0@3
  int v8; // r7@3
  int v9; // r0@3
  int v10; // r7@3
  int v11; // r3@3
  int v12; // [sp+4h] [bp-24h]@3
  int v13; // [sp+8h] [bp-20h]@3
  int v14; // [sp+Ch] [bp-1Ch]@3

  v3 = a3;
  v4 = a1;
  v5 = a2;
  result = ((int (*)(void))unk_809055B4)();
  if ( v3 )
  {
    if ( v5 )
    {
      v7 = ((int (__fastcall *)(int))strlen_0)(v5);
      v13 = v7;
      v8 = v7;
      v9 = ((int (__fastcall *)(int))strlen_0)(v3);
      v10 = v8 + 1;
```

```
      v14 = v9;
      v12 = v9 + 1;
      *(_DWORD *)(v4 + 52) = ((int (__fastcall *)(int))malloc_0)(v10);
      result = ((int (__fastcall *)(int))malloc_0)(v12);
      v11 = *(_DWORD *)(v4 + 52);
      *(_DWORD *)(v4 + 56) = result;
      if ( v11 )
      {
        if ( result )
        {
          ((void (__fastcall *)(int, _DWORD, int))memset_0)(v11, 0, v10);
          ((void (__fastcall *)(_DWORD, _DWORD, int))memset_0)(*(_DWORD *)(v4 + 56), 0, v12);
          ((void (__fastcall *)(_DWORD, int, int))memcpy_0)(*(_DWORD *)(v4 + 52), v5, v13);
          result = ((int (__fastcall *)(_DWORD, int, int))memcpy_0)(*(_DWORD *)(v4 + 56), v3, v14);
        }
      }
    }
  }
  return result;
}
```

```
int __fastcall sub_809055B4(int a1)
{
  int v1; // r4@1

  v1 = a1;
  if ( *(_DWORD *)(a1 + 52) )
  {
    ((void (*)(void))free)();
    *(_DWORD *)(v1 + 52) = 0;
  }
  if ( *(_DWORD *)(v1 + 56) )
  {
    ((void (*)(void))free)();
    *(_DWORD *)(v1 + 56) = 0;
  }
```

```
  memset_0(v1 + 60, 0, 30);
  return memset_0(v1, 0, 50);
}
```

接下来的工作分析吧，很常规了。。。

附两个头可以直接导入，使用其中的结构体，但是对于 C++方式结构体，即类的不知道怎么导入，记住先倒入依赖的头 stddarg.h（jni 依赖它）

```
/*
 * stdarg.h
 *
 * Provides facilities for stepping through a list of function arguments of
 * an unknown number and type.
 *
 * NOTE: Gcc should provide stdarg.h, and I believe their version will work
 *       with crtdll. If necessary I think you can replace this with the GCC
 *       stdarg.h.
 *
 * Note that the type used in va_arg is supposed to match the actual type
 * *after default promotions*. Thus, va_arg (..., short) is not valid.
 *
 * This file is part of the Mingw32 package.
 *
 * Contributors:
 *  Created by Colin Peters <colin@bird.fu.is.saga-u.ac.jp>
 *
 *  THIS SOFTWARE IS NOT COPYRIGHTED
 *
 *  This source code is offered for use in the public domain. You may
 *  use, modify or distribute it freely.
 *
 *  This code is distributed in the hope that it will be useful but
 *  WITHOUT ANY WARRANTY. ALL WARRANTIES, EXPRESS OR IMPLIED ARE HEREBY
 *  DISCLAMED. This includes but is not limited to warranties of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 *
 * $Revision: 1.2 $
```

```
 * $Author: noer $
 * $Date: 1998/10/10 00:51:16 $
 *
 */

#ifndef _STDARG_H_
#define _STDARG_H_

/*
 * Don't do any of this stuff for the resource compiler.
 */
#ifndef RC_INVOKED

/*
 * I was told that Win NT likes this.
 */
#ifndef _VA_LIST_DEFINED
#define _VA_LIST_DEFINED
#endif

#ifndef    _VA_LIST
#define _VA_LIST
typedef char* va_list;
#endif


/*
 * Amount of space required in an argument list (ie. the stack) for an
 * argument of type t.
 */
#define __va_argsiz(t)    \
    (((sizeof(t) + sizeof(int) - 1) / sizeof(int)) * sizeof(int))


/*
 * Start variable argument list processing by setting AP to point to the
 * argument after pN.
 */
#ifdef    __GNUC__
/*
 * In GNU the stack is not necessarily arranged very neatly in order to
 * pack shorts and such into a smaller argument list. Fortunately a
```

```
 * neatly arranged version is available through the use of __builtin_next_arg.
 */
#define va_start(ap, pN)    \
    ((ap) = ((va_list) __builtin_next_arg(pN)))
#else
/*
 * For a simple minded compiler this should work (it works in GNU too for
 * vararg lists that don't follow shorts and such).
 */
#define va_start(ap, pN)    \
    ((ap) = ((va_list) (&pN) + __va_argsiz(pN)))
#endif


/*
 * End processing of variable argument list. In this case we do nothing.
 */
#define va_end(ap)    ((void)0)


/*
 * Increment ap to the next argument in the list while returing a
 * pointer to what ap pointed to first, which is of type t.
 *
 * We cast to void* and then to t* because this avoids a warning about
 * increasing the alignment requirement.
 */

#define va_arg(ap, t)                    \
     (((ap) = (ap) + __va_argsiz(t)),        \
      *((t*) (void*) ((ap) - __va_argsiz(t))))

#endif /* Not RC_INVOKED */

#endif /* not _STDARG_H_ */
```

stdarg.h

```
//xref: /development/ndk/platforms/android-3/include/jni.h
//http://androidxref.com/2.3.6/xref/development/ndk/platforms/android-3/include/jni.h

/*
 * Copyright 2006 The Android Open Source Project
 *
 * JNI specification, as defined by Sun:
 * http://java.sun.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html
 *
 * Everything here is expected to be VM-neutral.
 */
#ifndef _JNI_H
#define _JNI_H

//#include <stdarg.h>

/*
 * Primitive types that match up with Java equivalents.
 */
#ifdef HAVE_INTTYPES_H
//@@@@@@@@@@ # include <inttypes.h>      /* C99 */
typedef uint8_t         jboolean;       /* unsigned 8 bits */
typedef int8_t          jbyte;          /* signed 8 bits */
typedef uint16_t        jchar;          /* unsigned 16 bits */
typedef int16_t         jshort;         /* signed 16 bits */
typedef int32_t         jint;           /* signed 32 bits */
typedef int64_t         jlong;          /* signed 64 bits */
typedef float           jfloat;         /* 32-bit IEEE 754 */
typedef double          jdouble;        /* 64-bit IEEE 754 */
#else
typedef unsigned char   jboolean;       /* unsigned 8 bits */
typedef signed char     jbyte;          /* signed 8 bits */
typedef unsigned short  jchar;          /* unsigned 16 bits */
typedef short           jshort;         /* signed 16 bits */
typedef int             jint;           /* signed 32 bits */
typedef long long       jlong;          /* signed 64 bits */
typedef float           jfloat;         /* 32-bit IEEE 754 */
typedef double          jdouble;        /* 64-bit IEEE 754 */
#endif

/* "cardinal indices and sizes" */
```

```
typedef jint            jsize;

#ifdef __cplusplus
/*
 * Reference types, in C++
 */
class _jobject {};
class _jclass : public _jobject {};
class _jstring : public _jobject {};
class _jarray : public _jobject {};
class _jobjectArray : public _jarray {};
class _jbooleanArray : public _jarray {};
class _jbyteArray : public _jarray {};
class _jcharArray : public _jarray {};
class _jshortArray : public _jarray {};
class _jintArray : public _jarray {};
class _jlongArray : public _jarray {};
class _jfloatArray : public _jarray {};
class _jdoubleArray : public _jarray {};
class _jthrowable : public _jobject {};

typedef _jobject*       jobject;
typedef _jclass*        jclass;
typedef _jstring*       jstring;
typedef _jarray*        jarray;
typedef _jobjectArray*  jobjectArray;
typedef _jbooleanArray* jbooleanArray;
typedef _jbyteArray*    jbyteArray;
typedef _jcharArray*    jcharArray;
typedef _jshortArray*   jshortArray;
typedef _jintArray*     jintArray;
typedef _jlongArray*    jlongArray;
typedef _jfloatArray*   jfloatArray;
typedef _jdoubleArray*  jdoubleArray;
typedef _jthrowable*    jthrowable;
typedef _jobject*       jweak;


#else /* not __cplusplus */

/*
 * Reference types, in C.
```

```
 */
typedef void*           jobject;
typedef jobject         jclass;
typedef jobject         jstring;
typedef jobject         jarray;
typedef jarray          jobjectArray;
typedef jarray          jbooleanArray;
typedef jarray          jbyteArray;
typedef jarray          jcharArray;
typedef jarray          jshortArray;
typedef jarray          jintArray;
typedef jarray          jlongArray;
typedef jarray          jfloatArray;
typedef jarray          jdoubleArray;
typedef jobject         jthrowable;
typedef jobject         jweak;

#endif /* not __cplusplus */

struct _jfieldID;                       /* opaque structure */
typedef struct _jfieldID* jfieldID;     /* field IDs */

struct _jmethodID;                      /* opaque structure */
typedef struct _jmethodID* jmethodID;   /* method IDs */

struct JNIInvokeInterface;

typedef union jvalue {
    jboolean    z;
    jbyte       b;
    jchar       c;
    jshort      s;
    jint        i;
    jlong       j;
    jfloat      f;
    jdouble     d;
    jobject     l;
} jvalue;

typedef enum jobjectRefType {
    JNIInvalidRefType = 0,
    JNILocalRefType = 1,
```

```
    JNIGlobalRefType = 2,
    JNIWeakGlobalRefType = 3
} jobjectRefType;

typedef struct {
    const char* name;
    const char* signature;
    void*       fnPtr;
} JNINativeMethod;

struct _JNIEnv;
struct _JavaVM;
typedef const struct JNINativeInterface* C_JNIEnv;

#if defined(__cplusplus)
typedef _JNIEnv JNIEnv;
typedef _JavaVM JavaVM;
#else
typedef const struct JNINativeInterface* JNIEnv;
typedef const struct JNIInvokeInterface* JavaVM;
#endif
```

```
/*
 * Table of interface function pointers.
 */
struct JNINativeInterface {
    void*       reserved0;
    void*       reserved1;
    void*       reserved2;
    void*       reserved3;

    jint        (*GetVersion)(JNIEnv *);

    jclass      (*DefineClass)(JNIEnv*, const char*, jobject, const jbyte*,
                        jsize);
    jclass      (*FindClass)(JNIEnv*, const char*);

    jmethodID   (*FromReflectedMethod)(JNIEnv*, jobject);
    jfieldID    (*FromReflectedField)(JNIEnv*, jobject);
    /* spec doesn't show jboolean parameter */
    jobject     (*ToReflectedMethod)(JNIEnv*, jclass, jmethodID, jboolean);
```

```
    jclass      (*GetSuperclass)(JNIEnv*, jclass);
    jboolean    (*IsAssignableFrom)(JNIEnv*, jclass, jclass);

    /* spec doesn't show jboolean parameter */
    jobject     (*ToReflectedField)(JNIEnv*, jclass, jfieldID, jboolean);

    jint        (*Throw)(JNIEnv*, jthrowable);
    jint        (*ThrowNew)(JNIEnv *, jclass, const char *);
    jthrowable  (*ExceptionOccurred)(JNIEnv*);
    void        (*ExceptionDescribe)(JNIEnv*);
    void        (*ExceptionClear)(JNIEnv*);
    void        (*FatalError)(JNIEnv*, const char*);

    jint        (*PushLocalFrame)(JNIEnv*, jint);
    jobject     (*PopLocalFrame)(JNIEnv*, jobject);

    jobject     (*NewGlobalRef)(JNIEnv*, jobject);
    void        (*DeleteGlobalRef)(JNIEnv*, jobject);
    void        (*DeleteLocalRef)(JNIEnv*, jobject);
    jboolean    (*IsSameObject)(JNIEnv*, jobject, jobject);

    jobject     (*NewLocalRef)(JNIEnv*, jobject);
    jint        (*EnsureLocalCapacity)(JNIEnv*, jint);

    jobject     (*AllocObject)(JNIEnv*, jclass);
    jobject     (*NewObject)(JNIEnv*, jclass, jmethodID, ...);
    jobject     (*NewObjectV)(JNIEnv*, jclass, jmethodID, va_list);
    jobject     (*NewObjectA)(JNIEnv*, jclass, jmethodID, jvalue*);

    jclass      (*GetObjectClass)(JNIEnv*, jobject);
    jboolean    (*IsInstanceOf)(JNIEnv*, jobject, jclass);
    jmethodID   (*GetMethodID)(JNIEnv*, jclass, const char*, const char*);

    jobject     (*CallObjectMethod)(JNIEnv*, jobject, jmethodID, ...);
    jobject     (*CallObjectMethodV)(JNIEnv*, jobject, jmethodID, va_list);
    jobject     (*CallObjectMethodA)(JNIEnv*, jobject, jmethodID, jvalue*);
    jboolean    (*CallBooleanMethod)(JNIEnv*, jobject, jmethodID, ...);
    jboolean    (*CallBooleanMethodV)(JNIEnv*, jobject, jmethodID, va_list);
    jboolean    (*CallBooleanMethodA)(JNIEnv*, jobject, jmethodID, jvalue*);
    jbyte       (*CallByteMethod)(JNIEnv*, jobject, jmethodID, ...);
    jbyte       (*CallByteMethodV)(JNIEnv*, jobject, jmethodID, va_list);
    jbyte       (*CallByteMethodA)(JNIEnv*, jobject, jmethodID, jvalue*);
```

```
jchar       (*CallCharMethod)(JNIEnv*, jobject, jmethodID, ...);
jchar       (*CallCharMethodV)(JNIEnv*, jobject, jmethodID, va_list);
jchar       (*CallCharMethodA)(JNIEnv*, jobject, jmethodID, jvalue*);
jshort      (*CallShortMethod)(JNIEnv*, jobject, jmethodID, ...);
jshort      (*CallShortMethodV)(JNIEnv*, jobject, jmethodID, va_list);
jshort      (*CallShortMethodA)(JNIEnv*, jobject, jmethodID, jvalue*);
jint        (*CallIntMethod)(JNIEnv*, jobject, jmethodID, ...);
jint        (*CallIntMethodV)(JNIEnv*, jobject, jmethodID, va_list);
jint        (*CallIntMethodA)(JNIEnv*, jobject, jmethodID, jvalue*);
jlong       (*CallLongMethod)(JNIEnv*, jobject, jmethodID, ...);
jlong       (*CallLongMethodV)(JNIEnv*, jobject, jmethodID, va_list);
jlong       (*CallLongMethodA)(JNIEnv*, jobject, jmethodID, jvalue*);
jfloat      (*CallFloatMethod)(JNIEnv*, jobject, jmethodID, ...);
jfloat      (*CallFloatMethodV)(JNIEnv*, jobject, jmethodID, va_list);
jfloat      (*CallFloatMethodA)(JNIEnv*, jobject, jmethodID, jvalue*);
jdouble     (*CallDoubleMethod)(JNIEnv*, jobject, jmethodID, ...);
jdouble     (*CallDoubleMethodV)(JNIEnv*, jobject, jmethodID, va_list);
jdouble     (*CallDoubleMethodA)(JNIEnv*, jobject, jmethodID, jvalue*);
void        (*CallVoidMethod)(JNIEnv*, jobject, jmethodID, ...);
void        (*CallVoidMethodV)(JNIEnv*, jobject, jmethodID, va_list);
void        (*CallVoidMethodA)(JNIEnv*, jobject, jmethodID, jvalue*);

jobject     (*CallNonvirtualObjectMethod)(JNIEnv*, jobject, jclass,
                jmethodID, ...);
jobject     (*CallNonvirtualObjectMethodV)(JNIEnv*, jobject, jclass,
                jmethodID, va_list);
jobject     (*CallNonvirtualObjectMethodA)(JNIEnv*, jobject, jclass,
                jmethodID, jvalue*);
jboolean    (*CallNonvirtualBooleanMethod)(JNIEnv*, jobject, jclass,
                jmethodID, ...);
jboolean    (*CallNonvirtualBooleanMethodV)(JNIEnv*, jobject, jclass,
                 jmethodID, va_list);
jboolean    (*CallNonvirtualBooleanMethodA)(JNIEnv*, jobject, jclass,
                 jmethodID, jvalue*);
jbyte       (*CallNonvirtualByteMethod)(JNIEnv*, jobject, jclass,
                jmethodID, ...);
jbyte       (*CallNonvirtualByteMethodV)(JNIEnv*, jobject, jclass,
                jmethodID, va_list);
jbyte       (*CallNonvirtualByteMethodA)(JNIEnv*, jobject, jclass,
                jmethodID, jvalue*);
jchar       (*CallNonvirtualCharMethod)(JNIEnv*, jobject, jclass,
                jmethodID, ...);
```

```
jchar       (*CallNonvirtualCharMethodV)(JNIEnv*, jobject, jclass,
                jmethodID, va_list);
jchar       (*CallNonvirtualCharMethodA)(JNIEnv*, jobject, jclass,
                jmethodID, jvalue*);
jshort      (*CallNonvirtualShortMethod)(JNIEnv*, jobject, jclass,
                jmethodID, ...);
jshort      (*CallNonvirtualShortMethodV)(JNIEnv*, jobject, jclass,
                jmethodID, va_list);
jshort      (*CallNonvirtualShortMethodA)(JNIEnv*, jobject, jclass,
                jmethodID, jvalue*);
jint        (*CallNonvirtualIntMethod)(JNIEnv*, jobject, jclass,
                jmethodID, ...);
jint        (*CallNonvirtualIntMethodV)(JNIEnv*, jobject, jclass,
                jmethodID, va_list);
jint        (*CallNonvirtualIntMethodA)(JNIEnv*, jobject, jclass,
                jmethodID, jvalue*);
jlong       (*CallNonvirtualLongMethod)(JNIEnv*, jobject, jclass,
                jmethodID, ...);
jlong       (*CallNonvirtualLongMethodV)(JNIEnv*, jobject, jclass,
                jmethodID, va_list);
jlong       (*CallNonvirtualLongMethodA)(JNIEnv*, jobject, jclass,
                jmethodID, jvalue*);
jfloat      (*CallNonvirtualFloatMethod)(JNIEnv*, jobject, jclass,
                jmethodID, ...);
jfloat      (*CallNonvirtualFloatMethodV)(JNIEnv*, jobject, jclass,
                jmethodID, va_list);
jfloat      (*CallNonvirtualFloatMethodA)(JNIEnv*, jobject, jclass,
                jmethodID, jvalue*);
jdouble     (*CallNonvirtualDoubleMethod)(JNIEnv*, jobject, jclass,
                jmethodID, ...);
jdouble     (*CallNonvirtualDoubleMethodV)(JNIEnv*, jobject, jclass,
                jmethodID, va_list);
jdouble     (*CallNonvirtualDoubleMethodA)(JNIEnv*, jobject, jclass,
                jmethodID, jvalue*);
void        (*CallNonvirtualVoidMethod)(JNIEnv*, jobject, jclass,
                jmethodID, ...);
void        (*CallNonvirtualVoidMethodV)(JNIEnv*, jobject, jclass,
                jmethodID, va_list);
void        (*CallNonvirtualVoidMethodA)(JNIEnv*, jobject, jclass,
                jmethodID, jvalue*);

jfieldID    (*GetFieldID)(JNIEnv*, jclass, const char*, const char*);
```

```
jobject     (*GetObjectField)(JNIEnv*, jobject, jfieldID);
jboolean    (*GetBooleanField)(JNIEnv*, jobject, jfieldID);
jbyte       (*GetByteField)(JNIEnv*, jobject, jfieldID);
jchar       (*GetCharField)(JNIEnv*, jobject, jfieldID);
jshort      (*GetShortField)(JNIEnv*, jobject, jfieldID);
jint        (*GetIntField)(JNIEnv*, jobject, jfieldID);
jlong       (*GetLongField)(JNIEnv*, jobject, jfieldID);
jfloat      (*GetFloatField)(JNIEnv*, jobject, jfieldID);
jdouble     (*GetDoubleField)(JNIEnv*, jobject, jfieldID);

void        (*SetObjectField)(JNIEnv*, jobject, jfieldID, jobject);
void        (*SetBooleanField)(JNIEnv*, jobject, jfieldID, jboolean);
void        (*SetByteField)(JNIEnv*, jobject, jfieldID, jbyte);
void        (*SetCharField)(JNIEnv*, jobject, jfieldID, jchar);
void        (*SetShortField)(JNIEnv*, jobject, jfieldID, jshort);
void        (*SetIntField)(JNIEnv*, jobject, jfieldID, jint);
void        (*SetLongField)(JNIEnv*, jobject, jfieldID, jlong);
void        (*SetFloatField)(JNIEnv*, jobject, jfieldID, jfloat);
void        (*SetDoubleField)(JNIEnv*, jobject, jfieldID, jdouble);
```

```
jmethodID   (*GetStaticMethodID)(JNIEnv*, jclass, const char*, const char*);

jobject     (*CallStaticObjectMethod)(JNIEnv*, jclass, jmethodID, ...);
jobject     (*CallStaticObjectMethodV)(JNIEnv*, jclass, jmethodID, va_list);
jobject     (*CallStaticObjectMethodA)(JNIEnv*, jclass, jmethodID, jvalue*);
jboolean    (*CallStaticBooleanMethod)(JNIEnv*, jclass, jmethodID, ...);
jboolean    (*CallStaticBooleanMethodV)(JNIEnv*, jclass, jmethodID,
                va_list);
jboolean    (*CallStaticBooleanMethodA)(JNIEnv*, jclass, jmethodID,
                jvalue*);
jbyte       (*CallStaticByteMethod)(JNIEnv*, jclass, jmethodID, ...);
jbyte       (*CallStaticByteMethodV)(JNIEnv*, jclass, jmethodID, va_list);
jbyte       (*CallStaticByteMethodA)(JNIEnv*, jclass, jmethodID, jvalue*);
jchar       (*CallStaticCharMethod)(JNIEnv*, jclass, jmethodID, ...);
jchar       (*CallStaticCharMethodV)(JNIEnv*, jclass, jmethodID, va_list);
jchar       (*CallStaticCharMethodA)(JNIEnv*, jclass, jmethodID, jvalue*);
jshort      (*CallStaticShortMethod)(JNIEnv*, jclass, jmethodID, ...);
jshort      (*CallStaticShortMethodV)(JNIEnv*, jclass, jmethodID, va_list);
jshort      (*CallStaticShortMethodA)(JNIEnv*, jclass, jmethodID, jvalue*);
jint        (*CallStaticIntMethod)(JNIEnv*, jclass, jmethodID, ...);
jint        (*CallStaticIntMethodV)(JNIEnv*, jclass, jmethodID, va_list);
```

```
jint        (*CallStaticIntMethodA)(JNIEnv*, jclass, jmethodID, jvalue*);
jlong       (*CallStaticLongMethod)(JNIEnv*, jclass, jmethodID, ...);
jlong       (*CallStaticLongMethodV)(JNIEnv*, jclass, jmethodID, va_list);
jlong       (*CallStaticLongMethodA)(JNIEnv*, jclass, jmethodID, jvalue*);
jfloat      (*CallStaticFloatMethod)(JNIEnv*, jclass, jmethodID, ...);
jfloat      (*CallStaticFloatMethodV)(JNIEnv*, jclass, jmethodID, va_list);
jfloat      (*CallStaticFloatMethodA)(JNIEnv*, jclass, jmethodID, jvalue*);
jdouble     (*CallStaticDoubleMethod)(JNIEnv*, jclass, jmethodID, ...);
jdouble     (*CallStaticDoubleMethodV)(JNIEnv*, jclass, jmethodID, va_list);
jdouble     (*CallStaticDoubleMethodA)(JNIEnv*, jclass, jmethodID, jvalue*);
void        (*CallStaticVoidMethod)(JNIEnv*, jclass, jmethodID, ...);
void        (*CallStaticVoidMethodV)(JNIEnv*, jclass, jmethodID, va_list);
void        (*CallStaticVoidMethodA)(JNIEnv*, jclass, jmethodID, jvalue*);

jfieldID    (*GetStaticFieldID)(JNIEnv*, jclass, const char*,
                const char*);

jobject     (*GetStaticObjectField)(JNIEnv*, jclass, jfieldID);
jboolean    (*GetStaticBooleanField)(JNIEnv*, jclass, jfieldID);
jbyte       (*GetStaticByteField)(JNIEnv*, jclass, jfieldID);
jchar       (*GetStaticCharField)(JNIEnv*, jclass, jfieldID);
jshort      (*GetStaticShortField)(JNIEnv*, jclass, jfieldID);
jint        (*GetStaticIntField)(JNIEnv*, jclass, jfieldID);
jlong       (*GetStaticLongField)(JNIEnv*, jclass, jfieldID);
jfloat      (*GetStaticFloatField)(JNIEnv*, jclass, jfieldID);
jdouble     (*GetStaticDoubleField)(JNIEnv*, jclass, jfieldID);

void        (*SetStaticObjectField)(JNIEnv*, jclass, jfieldID, jobject);
void        (*SetStaticBooleanField)(JNIEnv*, jclass, jfieldID, jboolean);
void        (*SetStaticByteField)(JNIEnv*, jclass, jfieldID, jbyte);
void        (*SetStaticCharField)(JNIEnv*, jclass, jfieldID, jchar);
void        (*SetStaticShortField)(JNIEnv*, jclass, jfieldID, jshort);
void        (*SetStaticIntField)(JNIEnv*, jclass, jfieldID, jint);
void        (*SetStaticLongField)(JNIEnv*, jclass, jfieldID, jlong);
void        (*SetStaticFloatField)(JNIEnv*, jclass, jfieldID, jfloat);
void        (*SetStaticDoubleField)(JNIEnv*, jclass, jfieldID, jdouble);

jstring     (*NewString)(JNIEnv*, const jchar*, jsize);
jsize       (*GetStringLength)(JNIEnv*, jstring);
const jchar* (*GetStringChars)(JNIEnv*, jstring, jboolean*);
void        (*ReleaseStringChars)(JNIEnv*, jstring, const jchar*);
jstring     (*NewStringUTF)(JNIEnv*, const char*);
```

```
    jsize       (*GetStringUTFLength)(JNIEnv*, jstring);
    /* JNI spec says this returns const jbyte*, but that's inconsistent */
    const char* (*GetStringUTFChars)(JNIEnv*, jstring, jboolean*);
    void        (*ReleaseStringUTFChars)(JNIEnv*, jstring, const char*);
    jsize       (*GetArrayLength)(JNIEnv*, jarray);
    jobjectArray (*NewObjectArray)(JNIEnv*, jsize, jclass, jobject);
    jobject     (*GetObjectArrayElement)(JNIEnv*, jobjectArray, jsize);
    void        (*SetObjectArrayElement)(JNIEnv*, jobjectArray, jsize, jobject);

    jbooleanArray (*NewBooleanArray)(JNIEnv*, jsize);
    jbyteArray    (*NewByteArray)(JNIEnv*, jsize);
    jcharArray    (*NewCharArray)(JNIEnv*, jsize);
    jshortArray   (*NewShortArray)(JNIEnv*, jsize);
    jintArray     (*NewIntArray)(JNIEnv*, jsize);
    jlongArray    (*NewLongArray)(JNIEnv*, jsize);
    jfloatArray   (*NewFloatArray)(JNIEnv*, jsize);
    jdoubleArray  (*NewDoubleArray)(JNIEnv*, jsize);

    jboolean*   (*GetBooleanArrayElements)(JNIEnv*, jbooleanArray, jboolean*);
    jbyte*      (*GetByteArrayElements)(JNIEnv*, jbyteArray, jboolean*);
    jchar*      (*GetCharArrayElements)(JNIEnv*, jcharArray, jboolean*);
    jshort*     (*GetShortArrayElements)(JNIEnv*, jshortArray, jboolean*);
    jint*       (*GetIntArrayElements)(JNIEnv*, jintArray, jboolean*);
    jlong*      (*GetLongArrayElements)(JNIEnv*, jlongArray, jboolean*);
    jfloat*     (*GetFloatArrayElements)(JNIEnv*, jfloatArray, jboolean*);
    jdouble*    (*GetDoubleArrayElements)(JNIEnv*, jdoubleArray, jboolean*);

    void        (*ReleaseBooleanArrayElements)(JNIEnv*, jbooleanArray,
                    jboolean*, jint);
    void        (*ReleaseByteArrayElements)(JNIEnv*, jbyteArray,
                    jbyte*, jint);
    void        (*ReleaseCharArrayElements)(JNIEnv*, jcharArray,
                    jchar*, jint);
    void        (*ReleaseShortArrayElements)(JNIEnv*, jshortArray,
                    jshort*, jint);
    void        (*ReleaseIntArrayElements)(JNIEnv*, jintArray,
                    jint*, jint);
    void        (*ReleaseLongArrayElements)(JNIEnv*, jlongArray,
                    jlong*, jint);
    void        (*ReleaseFloatArrayElements)(JNIEnv*, jfloatArray,
                    jfloat*, jint);
    void        (*ReleaseDoubleArrayElements)(JNIEnv*, jdoubleArray,
```

```
                jdouble*, jint);

    void        (*GetBooleanArrayRegion)(JNIEnv*, jbooleanArray,
                jsize, jsize, jboolean*);
    void        (*GetByteArrayRegion)(JNIEnv*, jbyteArray,
                jsize, jsize, jbyte*);
    void        (*GetCharArrayRegion)(JNIEnv*, jcharArray,
                jsize, jsize, jchar*);
    void        (*GetShortArrayRegion)(JNIEnv*, jshortArray,
                jsize, jsize, jshort*);
    void        (*GetIntArrayRegion)(JNIEnv*, jintArray,
                jsize, jsize, jint*);
    void        (*GetLongArrayRegion)(JNIEnv*, jlongArray,
                jsize, jsize, jlong*);
    void        (*GetFloatArrayRegion)(JNIEnv*, jfloatArray,
                jsize, jsize, jfloat*);
    void        (*GetDoubleArrayRegion)(JNIEnv*, jdoubleArray,
                jsize, jsize, jdouble*);

    /* spec shows these without const; some jni.h do, some don't */
    void        (*SetBooleanArrayRegion)(JNIEnv*, jbooleanArray,
                jsize, jsize, const jboolean*);
    void        (*SetByteArrayRegion)(JNIEnv*, jbyteArray,
                jsize, jsize, const jbyte*);
    void        (*SetCharArrayRegion)(JNIEnv*, jcharArray,
                jsize, jsize, const jchar*);
    void        (*SetShortArrayRegion)(JNIEnv*, jshortArray,
                jsize, jsize, const jshort*);
    void        (*SetIntArrayRegion)(JNIEnv*, jintArray,
                jsize, jsize, const jint*);
    void        (*SetLongArrayRegion)(JNIEnv*, jlongArray,
                jsize, jsize, const jlong*);
    void        (*SetFloatArrayRegion)(JNIEnv*, jfloatArray,
                jsize, jsize, const jfloat*);
    void        (*SetDoubleArrayRegion)(JNIEnv*, jdoubleArray,
                jsize, jsize, const jdouble*);

    jint        (*RegisterNatives)(JNIEnv*, jclass, const JNINativeMethod*,
                jint);
    jint        (*UnregisterNatives)(JNIEnv*, jclass);
    jint        (*MonitorEnter)(JNIEnv*, jobject);
    jint        (*MonitorExit)(JNIEnv*, jobject);
```

```
    jint        (*GetJavaVM)(JNIEnv*, JavaVM**);

    void        (*GetStringRegion)(JNIEnv*, jstring, jsize, jsize, jchar*);
    void        (*GetStringUTFRegion)(JNIEnv*, jstring, jsize, jsize, char*);

    void*       (*GetPrimitiveArrayCritical)(JNIEnv*, jarray, jboolean*);
    void        (*ReleasePrimitiveArrayCritical)(JNIEnv*, jarray, void*, jint);

    const jchar* (*GetStringCritical)(JNIEnv*, jstring, jboolean*);
    void        (*ReleaseStringCritical)(JNIEnv*, jstring, const jchar*);

    jweak       (*NewWeakGlobalRef)(JNIEnv*, jobject);
    void        (*DeleteWeakGlobalRef)(JNIEnv*, jweak);

    jboolean    (*ExceptionCheck)(JNIEnv*);

    jobject     (*NewDirectByteBuffer)(JNIEnv*, void*, jlong);
    void*       (*GetDirectBufferAddress)(JNIEnv*, jobject);
    jlong       (*GetDirectBufferCapacity)(JNIEnv*, jobject);

    /* added in JNI 1.6 */
    jobjectRefType (*GetObjectRefType)(JNIEnv*, jobject);
};

/*
 * C++ object wrapper.
 *
 * This is usually overlaid on a C struct whose first element is a
 * JNINativeInterface*.  We rely somewhat on compiler behavior.
 */
struct _JNIEnv {
    /* do not rename this; it does not seem to be entirely opaque */
    const struct JNINativeInterface* functions;

#if defined(__cplusplus)

    jint GetVersion()
    { return functions->GetVersion(this); }

    jclass DefineClass(const char *name, jobject loader, const jbyte* buf,
        jsize bufLen)
    { return functions->DefineClass(this, name, loader, buf, bufLen); }
```

```
jclass FindClass(const char* name)
{ return functions->FindClass(this, name); }

jmethodID FromReflectedMethod(jobject method)
{ return functions->FromReflectedMethod(this, method); }

jfieldID FromReflectedField(jobject field)
{ return functions->FromReflectedField(this, field); }

jobject ToReflectedMethod(jclass cls, jmethodID methodID, jboolean isStatic)
{ return functions->ToReflectedMethod(this, cls, methodID, isStatic); }

jclass GetSuperclass(jclass clazz)
{ return functions->GetSuperclass(this, clazz); }

jboolean IsAssignableFrom(jclass clazz1, jclass clazz2)
{ return functions->IsAssignableFrom(this, clazz1, clazz2); }

jobject ToReflectedField(jclass cls, jfieldID fieldID, jboolean isStatic)
{ return functions->ToReflectedField(this, cls, fieldID, isStatic); }

jint Throw(jthrowable obj)
{ return functions->Throw(this, obj); }

jint ThrowNew(jclass clazz, const char* message)
{ return functions->ThrowNew(this, clazz, message); }

jthrowable ExceptionOccurred()
{ return functions->ExceptionOccurred(this); }

void ExceptionDescribe()
{ functions->ExceptionDescribe(this); }

void ExceptionClear()
{ functions->ExceptionClear(this); }

void FatalError(const char* msg)
{ functions->FatalError(this, msg); }

jint PushLocalFrame(jint capacity)
{ return functions->PushLocalFrame(this, capacity); }
```

```
jobject PopLocalFrame(jobject result)
{ return functions->PopLocalFrame(this, result); }

jobject NewGlobalRef(jobject obj)
{ return functions->NewGlobalRef(this, obj); }

void DeleteGlobalRef(jobject globalRef)
{ functions->DeleteGlobalRef(this, globalRef); }

void DeleteLocalRef(jobject localRef)
{ functions->DeleteLocalRef(this, localRef); }

jboolean IsSameObject(jobject ref1, jobject ref2)
{ return functions->IsSameObject(this, ref1, ref2); }

jobject NewLocalRef(jobject ref)
{ return functions->NewLocalRef(this, ref); }

jint EnsureLocalCapacity(jint capacity)
{ return functions->EnsureLocalCapacity(this, capacity); }

jobject AllocObject(jclass clazz)
{ return functions->AllocObject(this, clazz); }

jobject NewObject(jclass clazz, jmethodID methodID, ...)
{
    va_list args;
    va_start(args, methodID);
    jobject result = functions->NewObjectV(this, clazz, methodID, args);
    va_end(args);
    return result;
}

jobject NewObjectV(jclass clazz, jmethodID methodID, va_list args)
{ return functions->NewObjectV(this, clazz, methodID, args); }

jobject NewObjectA(jclass clazz, jmethodID methodID, jvalue* args)
{ return functions->NewObjectA(this, clazz, methodID, args); }

jclass GetObjectClass(jobject obj)
{ return functions->GetObjectClass(this, obj); }
```

```
    jboolean IsInstanceOf(jobject obj, jclass clazz)
    { return functions->IsInstanceOf(this, obj, clazz); }

    jmethodID GetMethodID(jclass clazz, const char* name, const char* sig)
    { return functions->GetMethodID(this, clazz, name, sig); }

#define CALL_TYPE_METHOD(_jtype, _jname)                                \
    _jtype Call##_jname##Method(jobject obj, jmethodID methodID, ...)   \
    {                                                                   \
        _jtype result;                                                  \
        va_list args;                                                   \
        va_start(args, methodID);                                       \
        result = functions->Call##_jname##MethodV(this, obj, methodID,  \
                    args);                                              \
        va_end(args);                                                   \
        return result;                                                  \
    }
#define CALL_TYPE_METHODV(_jtype, _jname)                               \
    _jtype Call##_jname##MethodV(jobject obj, jmethodID methodID,       \
        va_list args)                                                   \
    { return functions->Call##_jname##MethodV(this, obj, methodID, args); }
#define CALL_TYPE_METHODA(_jtype, _jname)                               \
    _jtype Call##_jname##MethodA(jobject obj, jmethodID methodID,       \
        jvalue* args)                                                   \
    { return functions->Call##_jname##MethodA(this, obj, methodID, args); }

#define CALL_TYPE(_jtype, _jname)                                       \
    CALL_TYPE_METHOD(_jtype, _jname)                                    \
    CALL_TYPE_METHODV(_jtype, _jname)                                   \
    CALL_TYPE_METHODA(_jtype, _jname)

    CALL_TYPE(jobject, Object)
    CALL_TYPE(jboolean, Boolean)
    CALL_TYPE(jbyte, Byte)
    CALL_TYPE(jchar, Char)
    CALL_TYPE(jshort, Short)
    CALL_TYPE(jint, Int)
    CALL_TYPE(jlong, Long)
    CALL_TYPE(jfloat, Float)
    CALL_TYPE(jdouble, Double)
```

```
    void CallVoidMethod(jobject obj, jmethodID methodID, ...)
    {
        va_list args;
        va_start(args, methodID);
        functions->CallVoidMethodV(this, obj, methodID, args);
        va_end(args);
    }
    void CallVoidMethodV(jobject obj, jmethodID methodID, va_list args)
    { functions->CallVoidMethodV(this, obj, methodID, args); }
    void CallVoidMethodA(jobject obj, jmethodID methodID, jvalue* args)
    { functions->CallVoidMethodA(this, obj, methodID, args); }

#define CALL_NONVIRT_TYPE_METHOD(_jtype, _jname)                        \
    _jtype CallNonvirtual##_jname##Method(jobject obj, jclass clazz,    \
        jmethodID methodID, ...)                                        \
    {                                                                   \
        _jtype result;                                                  \
        va_list args;                                                   \
        va_start(args, methodID);                                       \
        result = functions->CallNonvirtual##_jname##MethodV(this, obj,  \
                    clazz, methodID, args);                             \
        va_end(args);                                                   \
        return result;                                                  \
    }
#define CALL_NONVIRT_TYPE_METHODV(_jtype, _jname)                       \
    _jtype CallNonvirtual##_jname##MethodV(jobject obj, jclass clazz,   \
        jmethodID methodID, va_list args)                               \
    { return functions->CallNonvirtual##_jname##MethodV(this, obj, clazz,   \
        methodID, args); }
#define CALL_NONVIRT_TYPE_METHODA(_jtype, _jname)                       \
    _jtype CallNonvirtual##_jname##MethodA(jobject obj, jclass clazz,   \
        jmethodID methodID, jvalue* args)                               \
    { return functions->CallNonvirtual##_jname##MethodA(this, obj, clazz,   \
        methodID, args); }

#define CALL_NONVIRT_TYPE(_jtype, _jname)                               \
    CALL_NONVIRT_TYPE_METHOD(_jtype, _jname)                            \
    CALL_NONVIRT_TYPE_METHODV(_jtype, _jname)                           \
    CALL_NONVIRT_TYPE_METHODA(_jtype, _jname)

    CALL_NONVIRT_TYPE(jobject, Object)
    CALL_NONVIRT_TYPE(jboolean, Boolean)
```

```
CALL_NONVIRT_TYPE(jbyte, Byte)
CALL_NONVIRT_TYPE(jchar, Char)
CALL_NONVIRT_TYPE(jshort, Short)
CALL_NONVIRT_TYPE(jint, Int)
CALL_NONVIRT_TYPE(jlong, Long)
CALL_NONVIRT_TYPE(jfloat, Float)
CALL_NONVIRT_TYPE(jdouble, Double)

void CallNonvirtualVoidMethod(jobject obj, jclass clazz,
    jmethodID methodID, ...)
{
    va_list args;
    va_start(args, methodID);
    functions->CallNonvirtualVoidMethodV(this, obj, clazz, methodID, args);
    va_end(args);
}
void CallNonvirtualVoidMethodV(jobject obj, jclass clazz,
    jmethodID methodID, va_list args)
{ functions->CallNonvirtualVoidMethodV(this, obj, clazz, methodID, args); }
void CallNonvirtualVoidMethodA(jobject obj, jclass clazz,
    jmethodID methodID, jvalue* args)
{ functions->CallNonvirtualVoidMethodA(this, obj, clazz, methodID, args); }

jfieldID GetFieldID(jclass clazz, const char* name, const char* sig)
{ return functions->GetFieldID(this, clazz, name, sig); }

jobject GetObjectField(jobject obj, jfieldID fieldID)
{ return functions->GetObjectField(this, obj, fieldID); }
jboolean GetBooleanField(jobject obj, jfieldID fieldID)
{ return functions->GetBooleanField(this, obj, fieldID); }
jbyte GetByteField(jobject obj, jfieldID fieldID)
{ return functions->GetByteField(this, obj, fieldID); }
jchar GetCharField(jobject obj, jfieldID fieldID)
{ return functions->GetCharField(this, obj, fieldID); }
jshort GetShortField(jobject obj, jfieldID fieldID)
{ return functions->GetShortField(this, obj, fieldID); }
jint GetIntField(jobject obj, jfieldID fieldID)
{ return functions->GetIntField(this, obj, fieldID); }
jlong GetLongField(jobject obj, jfieldID fieldID)
{ return functions->GetLongField(this, obj, fieldID); }
jfloat GetFloatField(jobject obj, jfieldID fieldID)
{ return functions->GetFloatField(this, obj, fieldID); }
```

```
jdouble GetDoubleField(jobject obj, jfieldID fieldID)
{ return functions->GetDoubleField(this, obj, fieldID); }

void SetObjectField(jobject obj, jfieldID fieldID, jobject value)
{ functions->SetObjectField(this, obj, fieldID, value); }
void SetBooleanField(jobject obj, jfieldID fieldID, jboolean value)
{ functions->SetBooleanField(this, obj, fieldID, value); }
void SetByteField(jobject obj, jfieldID fieldID, jbyte value)
{ functions->SetByteField(this, obj, fieldID, value); }
void SetCharField(jobject obj, jfieldID fieldID, jchar value)
{ functions->SetCharField(this, obj, fieldID, value); }
void SetShortField(jobject obj, jfieldID fieldID, jshort value)
{ functions->SetShortField(this, obj, fieldID, value); }
void SetIntField(jobject obj, jfieldID fieldID, jint value)
{ functions->SetIntField(this, obj, fieldID, value); }
void SetLongField(jobject obj, jfieldID fieldID, jlong value)
{ functions->SetLongField(this, obj, fieldID, value); }
void SetFloatField(jobject obj, jfieldID fieldID, jfloat value)
{ functions->SetFloatField(this, obj, fieldID, value); }
void SetDoubleField(jobject obj, jfieldID fieldID, jdouble value)
{ functions->SetDoubleField(this, obj, fieldID, value); }

    jmethodID GetStaticMethodID(jclass clazz, const char* name, const char* sig)
    { return functions->GetStaticMethodID(this, clazz, name, sig); }

#define CALL_STATIC_TYPE_METHOD(_jtype, _jname)                           \
    _jtype CallStatic##_jname##Method(jclass clazz, jmethodID methodID,   \
        ...)                                                              \
    {                                                                     \
        _jtype result;                                                    \
        va_list args;                                                     \
        va_start(args, methodID);                                         \
        result = functions->CallStatic##_jname##MethodV(this, clazz,      \
                    methodID, args);                                      \
        va_end(args);                                                     \
        return result;                                                    \
    }
#define CALL_STATIC_TYPE_METHODV(_jtype, _jname)                          \
    _jtype CallStatic##_jname##MethodV(jclass clazz, jmethodID methodID,  \
        va_list args)                                                     \
    { return functions->CallStatic##_jname##MethodV(this, clazz, methodID,  \
        args); }
```

```
#define CALL_STATIC_TYPE_METHODA(_jtype, _jname)                              \
    _jtype CallStatic##_jname##MethodA(jclass clazz, jmethodID methodID,     \
        jvalue* args)                                                         \
    { return functions->CallStatic##_jname##MethodA(this, clazz, methodID,   \
        args); }

#define CALL_STATIC_TYPE(_jtype, _jname)                                      \
    CALL_STATIC_TYPE_METHOD(_jtype, _jname)                                   \
    CALL_STATIC_TYPE_METHODV(_jtype, _jname)                                  \
    CALL_STATIC_TYPE_METHODA(_jtype, _jname)

    CALL_STATIC_TYPE(jobject, Object)
    CALL_STATIC_TYPE(jboolean, Boolean)
    CALL_STATIC_TYPE(jbyte, Byte)
    CALL_STATIC_TYPE(jchar, Char)
    CALL_STATIC_TYPE(jshort, Short)
    CALL_STATIC_TYPE(jint, Int)
    CALL_STATIC_TYPE(jlong, Long)
    CALL_STATIC_TYPE(jfloat, Float)
    CALL_STATIC_TYPE(jdouble, Double)
```

```
void CallStaticVoidMethod(jclass clazz, jmethodID methodID, ...)
{
    va_list args;
    va_start(args, methodID);
    functions->CallStaticVoidMethodV(this, clazz, methodID, args);
    va_end(args);
}
void CallStaticVoidMethodV(jclass clazz, jmethodID methodID, va_list args)
{ functions->CallStaticVoidMethodV(this, clazz, methodID, args); }
void CallStaticVoidMethodA(jclass clazz, jmethodID methodID, jvalue* args)
{ functions->CallStaticVoidMethodA(this, clazz, methodID, args); }

jfieldID GetStaticFieldID(jclass clazz, const char* name, const char* sig)
{ return functions->GetStaticFieldID(this, clazz, name, sig); }

jobject GetStaticObjectField(jclass clazz, jfieldID fieldID)
{ return functions->GetStaticObjectField(this, clazz, fieldID); }
jboolean GetStaticBooleanField(jclass clazz, jfieldID fieldID)
{ return functions->GetStaticBooleanField(this, clazz, fieldID); }
jbyte GetStaticByteField(jclass clazz, jfieldID fieldID)
{ return functions->GetStaticByteField(this, clazz, fieldID); }
```

```
jchar GetStaticCharField(jclass clazz, jfieldID fieldID)
{ return functions->GetStaticCharField(this, clazz, fieldID); }
jshort GetStaticShortField(jclass clazz, jfieldID fieldID)
{ return functions->GetStaticShortField(this, clazz, fieldID); }
jint GetStaticIntField(jclass clazz, jfieldID fieldID)
{ return functions->GetStaticIntField(this, clazz, fieldID); }
jlong GetStaticLongField(jclass clazz, jfieldID fieldID)
{ return functions->GetStaticLongField(this, clazz, fieldID); }
jfloat GetStaticFloatField(jclass clazz, jfieldID fieldID)
{ return functions->GetStaticFloatField(this, clazz, fieldID); }
jdouble GetStaticDoubleField(jclass clazz, jfieldID fieldID)
{ return functions->GetStaticDoubleField(this, clazz, fieldID); }

void SetStaticObjectField(jclass clazz, jfieldID fieldID, jobject value)
{ functions->SetStaticObjectField(this, clazz, fieldID, value); }
void SetStaticBooleanField(jclass clazz, jfieldID fieldID, jboolean value)
{ functions->SetStaticBooleanField(this, clazz, fieldID, value); }
void SetStaticByteField(jclass clazz, jfieldID fieldID, jbyte value)
{ functions->SetStaticByteField(this, clazz, fieldID, value); }
void SetStaticCharField(jclass clazz, jfieldID fieldID, jchar value)
{ functions->SetStaticCharField(this, clazz, fieldID, value); }
void SetStaticShortField(jclass clazz, jfieldID fieldID, jshort value)
{ functions->SetStaticShortField(this, clazz, fieldID, value); }
void SetStaticIntField(jclass clazz, jfieldID fieldID, jint value)
{ functions->SetStaticIntField(this, clazz, fieldID, value); }
void SetStaticLongField(jclass clazz, jfieldID fieldID, jlong value)
{ functions->SetStaticLongField(this, clazz, fieldID, value); }
void SetStaticFloatField(jclass clazz, jfieldID fieldID, jfloat value)
{ functions->SetStaticFloatField(this, clazz, fieldID, value); }
void SetStaticDoubleField(jclass clazz, jfieldID fieldID, jdouble value)
{ functions->SetStaticDoubleField(this, clazz, fieldID, value); }

jstring NewString(const jchar* unicodeChars, jsize len)
{ return functions->NewString(this, unicodeChars, len); }

jsize GetStringLength(jstring string)
{ return functions->GetStringLength(this, string); }

const jchar* GetStringChars(jstring string, jboolean* isCopy)
{ return functions->GetStringChars(this, string, isCopy); }

void ReleaseStringChars(jstring string, const jchar* chars)
```

```
{ functions->ReleaseStringChars(this, string, chars); }

jstring NewStringUTF(const char* bytes)
{ return functions->NewStringUTF(this, bytes); }

jsize GetStringUTFLength(jstring string)
{ return functions->GetStringUTFLength(this, string); }

const char* GetStringUTFChars(jstring string, jboolean* isCopy)
{ return functions->GetStringUTFChars(this, string, isCopy); }

void ReleaseStringUTFChars(jstring string, const char* utf)
{ functions->ReleaseStringUTFChars(this, string, utf); }

jsize GetArrayLength(jarray array)
{ return functions->GetArrayLength(this, array); }

jobjectArray NewObjectArray(jsize length, jclass elementClass,
    jobject initialElement)
{ return functions->NewObjectArray(this, length, elementClass,
    initialElement); }

jobject GetObjectArrayElement(jobjectArray array, jsize index)
{ return functions->GetObjectArrayElement(this, array, index); }

void SetObjectArrayElement(jobjectArray array, jsize index, jobject value)
{ functions->SetObjectArrayElement(this, array, index, value); }

jbooleanArray NewBooleanArray(jsize length)
{ return functions->NewBooleanArray(this, length); }
jbyteArray NewByteArray(jsize length)
{ return functions->NewByteArray(this, length); }
jcharArray NewCharArray(jsize length)
{ return functions->NewCharArray(this, length); }
jshortArray NewShortArray(jsize length)
{ return functions->NewShortArray(this, length); }
jintArray NewIntArray(jsize length)
{ return functions->NewIntArray(this, length); }
jlongArray NewLongArray(jsize length)
{ return functions->NewLongArray(this, length); }
jfloatArray NewFloatArray(jsize length)
{ return functions->NewFloatArray(this, length); }
```

```
jdoubleArray NewDoubleArray(jsize length)
{ return functions->NewDoubleArray(this, length); }

jboolean* GetBooleanArrayElements(jbooleanArray array, jboolean* isCopy)
{ return functions->GetBooleanArrayElements(this, array, isCopy); }
jbyte* GetByteArrayElements(jbyteArray array, jboolean* isCopy)
{ return functions->GetByteArrayElements(this, array, isCopy); }
jchar* GetCharArrayElements(jcharArray array, jboolean* isCopy)
{ return functions->GetCharArrayElements(this, array, isCopy); }
jshort* GetShortArrayElements(jshortArray array, jboolean* isCopy)
{ return functions->GetShortArrayElements(this, array, isCopy); }
jint* GetIntArrayElements(jintArray array, jboolean* isCopy)
{ return functions->GetIntArrayElements(this, array, isCopy); }
jlong* GetLongArrayElements(jlongArray array, jboolean* isCopy)
{ return functions->GetLongArrayElements(this, array, isCopy); }
jfloat* GetFloatArrayElements(jfloatArray array, jboolean* isCopy)
{ return functions->GetFloatArrayElements(this, array, isCopy); }
jdouble* GetDoubleArrayElements(jdoubleArray array, jboolean* isCopy)
{ return functions->GetDoubleArrayElements(this, array, isCopy); }

void ReleaseBooleanArrayElements(jbooleanArray array, jboolean* elems,
    jint mode)
{ functions->ReleaseBooleanArrayElements(this, array, elems, mode); }
void ReleaseByteArrayElements(jbyteArray array, jbyte* elems,
    jint mode)
{ functions->ReleaseByteArrayElements(this, array, elems, mode); }
void ReleaseCharArrayElements(jcharArray array, jchar* elems,
    jint mode)
{ functions->ReleaseCharArrayElements(this, array, elems, mode); }
void ReleaseShortArrayElements(jshortArray array, jshort* elems,
    jint mode)
{ functions->ReleaseShortArrayElements(this, array, elems, mode); }
void ReleaseIntArrayElements(jintArray array, jint* elems,
    jint mode)
{ functions->ReleaseIntArrayElements(this, array, elems, mode); }
void ReleaseLongArrayElements(jlongArray array, jlong* elems,
    jint mode)
{ functions->ReleaseLongArrayElements(this, array, elems, mode); }
void ReleaseFloatArrayElements(jfloatArray array, jfloat* elems,
    jint mode)
{ functions->ReleaseFloatArrayElements(this, array, elems, mode); }
void ReleaseDoubleArrayElements(jdoubleArray array, jdouble* elems,
```

```
    jint mode)
{ functions->ReleaseDoubleArrayElements(this, array, elems, mode); }

void GetBooleanArrayRegion(jbooleanArray array, jsize start, jsize len,
    jboolean* buf)
{ functions->GetBooleanArrayRegion(this, array, start, len, buf); }
void GetByteArrayRegion(jbyteArray array, jsize start, jsize len,
    jbyte* buf)
{ functions->GetByteArrayRegion(this, array, start, len, buf); }
void GetCharArrayRegion(jcharArray array, jsize start, jsize len,
    jchar* buf)
{ functions->GetCharArrayRegion(this, array, start, len, buf); }
void GetShortArrayRegion(jshortArray array, jsize start, jsize len,
    jshort* buf)
{ functions->GetShortArrayRegion(this, array, start, len, buf); }
void GetIntArrayRegion(jintArray array, jsize start, jsize len,
    jint* buf)
{ functions->GetIntArrayRegion(this, array, start, len, buf); }
void GetLongArrayRegion(jlongArray array, jsize start, jsize len,
    jlong* buf)
{ functions->GetLongArrayRegion(this, array, start, len, buf); }
void GetFloatArrayRegion(jfloatArray array, jsize start, jsize len,
    jfloat* buf)
{ functions->GetFloatArrayRegion(this, array, start, len, buf); }
void GetDoubleArrayRegion(jdoubleArray array, jsize start, jsize len,
    jdouble* buf)
{ functions->GetDoubleArrayRegion(this, array, start, len, buf); }

void SetBooleanArrayRegion(jbooleanArray array, jsize start, jsize len,
    const jboolean* buf)
{ functions->SetBooleanArrayRegion(this, array, start, len, buf); }
void SetByteArrayRegion(jbyteArray array, jsize start, jsize len,
    const jbyte* buf)
{ functions->SetByteArrayRegion(this, array, start, len, buf); }
void SetCharArrayRegion(jcharArray array, jsize start, jsize len,
    const jchar* buf)
{ functions->SetCharArrayRegion(this, array, start, len, buf); }
void SetShortArrayRegion(jshortArray array, jsize start, jsize len,
    const jshort* buf)
{ functions->SetShortArrayRegion(this, array, start, len, buf); }
void SetIntArrayRegion(jintArray array, jsize start, jsize len,
    const jint* buf)
```

```
{ functions->SetIntArrayRegion(this, array, start, len, buf); }
void SetLongArrayRegion(jlongArray array, jsize start, jsize len,
    const jlong* buf)
{ functions->SetLongArrayRegion(this, array, start, len, buf); }
void SetFloatArrayRegion(jfloatArray array, jsize start, jsize len,
    const jfloat* buf)
{ functions->SetFloatArrayRegion(this, array, start, len, buf); }
void SetDoubleArrayRegion(jdoubleArray array, jsize start, jsize len,
    const jdouble* buf)
{ functions->SetDoubleArrayRegion(this, array, start, len, buf); }

jint RegisterNatives(jclass clazz, const JNINativeMethod* methods,
    jint nMethods)
{ return functions->RegisterNatives(this, clazz, methods, nMethods); }

jint UnregisterNatives(jclass clazz)
{ return functions->UnregisterNatives(this, clazz); }

jint MonitorEnter(jobject obj)
{ return functions->MonitorEnter(this, obj); }

jint MonitorExit(jobject obj)
{ return functions->MonitorExit(this, obj); }

jint GetJavaVM(JavaVM** vm)
{ return functions->GetJavaVM(this, vm); }

void GetStringRegion(jstring str, jsize start, jsize len, jchar* buf)
{ functions->GetStringRegion(this, str, start, len, buf); }

void GetStringUTFRegion(jstring str, jsize start, jsize len, char* buf)
{ return functions->GetStringUTFRegion(this, str, start, len, buf); }

void* GetPrimitiveArrayCritical(jarray array, jboolean* isCopy)
{ return functions->GetPrimitiveArrayCritical(this, array, isCopy); }

void ReleasePrimitiveArrayCritical(jarray array, void* carray, jint mode)
{ functions->ReleasePrimitiveArrayCritical(this, array, carray, mode); }

const jchar* GetStringCritical(jstring string, jboolean* isCopy)
{ return functions->GetStringCritical(this, string, isCopy); }
```

```
void ReleaseStringCritical(jstring string, const jchar* carray)
{ functions->ReleaseStringCritical(this, string, carray); }

jweak NewWeakGlobalRef(jobject obj)
{ return functions->NewWeakGlobalRef(this, obj); }

void DeleteWeakGlobalRef(jweak obj)
{ functions->DeleteWeakGlobalRef(this, obj); }

jboolean ExceptionCheck()
{ return functions->ExceptionCheck(this); }

jobject NewDirectByteBuffer(void* address, jlong capacity)
{ return functions->NewDirectByteBuffer(this, address, capacity); }

void* GetDirectBufferAddress(jobject buf)
{ return functions->GetDirectBufferAddress(this, buf); }

jlong GetDirectBufferCapacity(jobject buf)
{ return functions->GetDirectBufferCapacity(this, buf); }
    /* added in JNI 1.6 */
    jobjectRefType GetObjectRefType(jobject obj)
    { return functions->GetObjectRefType(this, obj); }
#endif /*__cplusplus*/
};


/*
 * JNI invocation interface.
 */
struct JNIInvokeInterface {
    void*       reserved0;
    void*       reserved1;
    void*       reserved2;

    jint        (*DestroyJavaVM)(JavaVM*);
    jint        (*AttachCurrentThread)(JavaVM*, JNIEnv**, void*);
    jint        (*DetachCurrentThread)(JavaVM*);
    jint        (*GetEnv)(JavaVM*, void**, jint);
    jint        (*AttachCurrentThreadAsDaemon)(JavaVM*, JNIEnv**, void*);
};
```

```
/*
 * C++ version.
 */
struct _JavaVM {
    const struct JNIInvokeInterface* functions;

#if defined(__cplusplus)
    jint DestroyJavaVM()
    { return functions->DestroyJavaVM(this); }
    jint AttachCurrentThread(JNIEnv** p_env, void* thr_args)
    { return functions->AttachCurrentThread(this, p_env, thr_args); }
    jint DetachCurrentThread()
    { return functions->DetachCurrentThread(this); }
    jint GetEnv(void** env, jint version)
    { return functions->GetEnv(this, env, version); }
    jint AttachCurrentThreadAsDaemon(JNIEnv** p_env, void* thr_args)
    { return functions->AttachCurrentThreadAsDaemon(this, p_env, thr_args); }
#endif /*__cplusplus*/
};

struct JavaVMAttachArgs {
    jint        version;    /* must be >= JNI_VERSION_1_2 */
    const char* name;       /* NULL or name of thread as modified UTF-8 str */
    jobject     group;      /* global ref of a ThreadGroup object, or NULL */
};
typedef struct JavaVMAttachArgs JavaVMAttachArgs;

/*
 * JNI 1.2+ initialization.  (As of 1.6, the pre-1.2 structures are no
 * longer supported.)
 */
typedef struct JavaVMOption {
    const char* optionString;
    void*       extraInfo;
} JavaVMOption;

typedef struct JavaVMInitArgs {
    jint        version;    /* use JNI_VERSION_1_2 or later */

    jint        nOptions;
    JavaVMOption* options;
```

```
    jboolean    ignoreUnrecognized;
} JavaVMInitArgs;

#ifdef __cplusplus
extern "C" {
#endif
/*
 * VM initialization functions.
 *
 * Note these are the only symbols exported for JNI by the VM.
 */
jint JNI_GetDefaultJavaVMInitArgs(void*);
jint JNI_CreateJavaVM(JavaVM**, JNIEnv**, void*);
jint JNI_GetCreatedJavaVMs(JavaVM**, jsize, jsize*);

/*
 * Prototypes for functions exported by loadable shared libs.  These are
 * called by JNI, not provided by JNI.
 */
jint JNI_OnLoad(JavaVM* vm, void* reserved);
void JNI_OnUnload(JavaVM* vm, void* reserved);
```

```
#ifdef __cplusplus
}
#endif


/*
 * Manifest constants.
 */
#define JNI_FALSE   0
#define JNI_TRUE    1

#define JNI_VERSION_1_1 0x00010001
#define JNI_VERSION_1_2 0x00010002
#define JNI_VERSION_1_4 0x00010004
#define JNI_VERSION_1_6 0x00010006

#define JNI_OK          (0)         /* no error */
#define JNI_ERR         (-1)        /* generic error */
#define JNI_EDETACHED   (-2)        /* thread detached from the VM */
#define JNI_EVERSION    (-3)        /* JNI version error */
```

```
#define JNI_COMMIT      1           /* copy content, do not free buffer */
#define JNI_ABORT       2           /* free buffer w/o copying back */

/* need these for Windows-aware headers */
#define JNIIMPORT
#define JNIEXPORT
#define JNICALL

#endif /*_JNI_H*/
```

jni.h

然后可以使用其中的结构体了。

有人曾搞成 excel 但是不知道地址了，临时下载，当时看到下来看是不能传出去的。

IDA 自定义结构体快捷键操作方法

简单记录下，有时在分析漏洞时，以特定结构体去解析数据，更便于准确定位漏洞成因，尤其是文件格式的漏洞。

1、Shift + F1：打开本地类型；

2、insert：插入自定义的结构体；

3、Shift + F9：打开结构体；

4、Insert：添加上面创建的结构体；

5、选取相应数据，Alt + Q 将其以上面的结构体进行解析。

函数调用方式

X86 调用约定 calling convention

http://zh.wikipedia.org/wiki/X86%E8%B0%83%E7%94%A8%E7%BA%A6%E5%AE%9A

这里描述了在 x86 芯片架构上的调用约定(calling conventions)。 调用约定描述了被调用代码的接口：

- 原子(标量)参数，或复杂参数独立部分的分配顺序;
- 参数是如何被传递的(放置在栈上，或是寄存器中，亦或两者混合);
- 被调用者应保存调用者的哪个寄存器;
- 调用函数时如何为任务准备堆栈，以及任务完成如何恢复;

这与编程语言中对于大小和格式的分配紧密相关。另一个密切相关的是名称修饰，这决定了代码中的符号名称如何映射到链接器中的符号名。

调用约定，类型表示和名称修饰这三者的统称，即是总所周知的应用二进制接口(ABI)。

不同编译器在实现这些约定总是有细微的差别存在，所以在不同编译器编译出来的代码很难接合起来。

另一方面，有些约定被当作一种 API 标准(如 stdcall)，编译器实现都较为一致。

调用者清理 cdecl syscall optlink

在这些约定中，调用者自己清理栈上的变元(arguments)，这样就运行了可变参数列表的实现，如 printf()。

cdecl

cdecl(C declaration，即 C 声明)是源起 C 语言的一种调用约定，x86 架构上的许多 C 编译器都使用这个约定。
在 cdecl 中，子例程变元是在栈上传递的。EAX 寄存器返回整型值和内存地址，浮点数则是在 ST0 x87 寄存器上。
EAX, ECX 和 EDX 寄存器是由调用者保存的，其余的寄存器由被调用者保存。(EBX, EBP, ESI, EDI)
当调用一个新函数时，x87 浮点寄存器 ST0 到 ST7 都必须为空(弹出或释放掉)，而且在退出函数时 ST1 到 ST7 也必须为空。
在 C 语言中，函数参数是以相反顺序推入栈的。在 GNU/Linux GCC，把这一约定做为事实上的标准。
GCC 自 4.5 版本开始，调用函数时，堆栈上的数据必须以 16B 对齐(之前的版本只需要 4B 对齐即可)。

cdecl 调用约定通常作为 x86 C 编译器的默认调用规则，许多编译器也提供了自动切换调用约定的选项。
如果需要手动指定调用规则为 cdecl，编译器可能会支持如下语法：

```
void _cdecl funct();
```

其中_cdecl 修饰符需要在函数原型中给出，在函数声明中会覆盖掉其他的设置。

syscall

与 cdecl 类似，变元被从右到左推入栈中。EAX, ECX 和 EDX 不会保留值。参数列表的大小被放置在 AL 寄存器中(?)。

syscall 是 32 位 OS/2 API 的标准。

optlink

变元也是从右到左被推入栈。从最左边开始的三个字符变元会被放置在 EAX, EDX 和 ECX 中，最多四个浮点变元会被传入 ST(0)到 ST(3)中----

虽然这四个参数的空间也会在参数列表的栈上保留。函数的返回值在 EAX 或 ST(0)中。保留的寄存器有 EBP, EBX, ESI 和 EDI。

optlink 在 IBM VisualAge 编译器中被使用。

被调用者清理 pascal register stdcall fastcall (microsoft, borland)

如果被调用者要清理栈上的参数，需要在编译阶段知道栈上有多少字节要处理。因此，此类的调用约定并不能兼容于可变参数列表，如 printf()。
然而，这种调用约定也许会更有效率，因为需要解堆栈的代码不要在每次调用时都生成一遍。
使用此规则的函数容易在 asm 代码被认出，因为它们会在返回前解堆栈。
x86 ret 指令允许一个可选的 16 位参数说明栈字节数，用来在返回给调用者之前解堆栈。代码类似如下：

```
ret 12
```

pascal

基于 Pascal 语言的调用约定，参数从左至右入栈(与 cdecl 相反)。被调用者负责在返回前清理堆栈。 此调用约定常见在如下 16-bit API 中：OS/2 1.x，微软 Windows 3.x，以及 Borland Delphi 版本 1.x。

register

Borland fastcall 的别名而已。

stdcall

这个一个 Pascal 调用约定的变体，被调用者依旧负责清理堆栈，但是参数从右往左入栈----与 cdecl 一致。
寄存器 EAX, ECX 和 EDX 被指定在函数中使用，返回值放置在 EAX 中。

stdcall 对于微软 Win32 API 和 Open Watcom C++是标准。

fastcall

此约定还未被标准化，不同编译器的实现也不一致。 典型的 fastcall 约定会传递一个或多个变元到寄存器上，减少对内存的访问。

Microsoft fastcall

Microsoft 或 GCC 的__fastcall 约定(也即__msfastcall)传入头两个变元(从左至右)到 ECX 和 EDX 中，剩下的变元从右至左推入栈上。

Borland fastcall

从左至右，传入三个参数至 EAX, EDX 和 ECX 中。剩下的参数推入栈，也是从左至右。

在 32 位编译器 Embarcadero Delphi 中，这是缺省调用约定，在编译器中以 register 形式为人知。 在 i386 上的某些版本 Linux 也使用了此约定。

调用者或被调用者清理 thiscall

thiscall

在调用 C++非静态成员函数时使用此约定。基于所使用的编译器和函数是否使用可变参数，有两个主流版本的 thiscall。

对于 GCC 编译器，thiscall 几乎与 cdecl 等同：调用者清理堆栈，参数从右到左传递。差别在于 this 指针，thiscall 会在最后把指针推入栈中，虽然在函数原型中它是隐式的第一个参数。

在微软 Visual C++编译器中，this 指针被传到 ECX 寄存器上，被调用者负责清理堆栈，其余同此编译器的 C 版本和 Windows API 函数使用的 stdcall 约定。
当函数使用可变参数，此时调用者负责清理堆栈(参考 cdecl)。 thiscall 约定只在微软 Visual C++ 2005 及其之后的版本被显式指定。
其他编译器中，thiscall 并不是一个关键字(反汇编器如 IDA 使用__thiscall)。

x86-64 调用约定

x86-64 调用约定得益于更多的寄存器可以用来传参。而且，不兼容的调用约定也更少了，不过还是有 2 种主流的规则。

微软 x64 调用约定

微软 x64 调用约定使用 RCX, RDX, R8, R9 这四个寄存器传递头四个整型或指针变量(从左到右)，
使用 XMM0, XMM1, XMM2, XMM3 来传递浮点变量。
其他的参数直接入栈(从右至左)。

整型返回值放置在 RAX 中，浮点返回值在 XMM0 中。

少于 64 位的参数并没有做零扩展，此时高位充斥着垃圾。

在 Windows x64 环境下编译代码时，只有一种调用约定----就是上面描述的约定，也就是说，32 位下的各种约定在 64 位下统一成一种了。

在微软 x64 调用约定中，调用者的一个职责是在调用函数之前(无论实际的传参使用多大空间)，在栈上分配一个 32B 的“影子空间”；并且在调用之后用弹出此堆栈。

影子空间是用来给 RCX, RDX, R8 和 R9 提供溢出空间的(?)，即使是对于少于四个参数的函数而言。

例如， 一个函数拥有 5 个整型参数，第一个到第四个放在寄存器中，第五个就被推到影子空间栈顶上。

当函数被调用，此栈用来组成返回值----影子空间 32 位+第五个参数。

在 x86-64 体系下，Visual Studio 2008 在 XMM6 和 XMM7 中(同样的有 XMM8 到 XMM15)存储浮点数。

结果对于用户写的汇编语言例程，必须保存 XMM6 和 XMM7(x86 不用保存这两个寄存器)，

这也就是说，在 x86 和 x86-64 之间移植汇编例程时，需要注意在函数调用之前/之后，要保存/恢复 XMM6 和 XMM7。

System V AMD64 ABI

此约定主要在 Solaris，GNU/Linux，FreeBSD 和其他非微软 OS 上使用。

头六个整型参数放在寄存器 RDI, RSI, RDX, RCX, R8 和 R9 上；同时 XMM0 到 XMM7 用来放置浮点变元。

对于系统调用，R10 用来替代 RCX。同微软 x64 约定一样，其他额外的参数推入栈，返回值保存在 RAX 中。

与微软不同的是，不需要提供影子空间。在函数入口，返回值与栈上第七个整型参数相邻。

调用约定(pascal,fastcall,stdcall,thiscall,cdecl)区别等

http://blog.csdn.net/maotoula/article/details/6762062

一：函数调用约定;
　　函数调用约定是函数调用者和被调用的函数体之间关于参数传递、返回值传递、堆栈清除、寄存器使用的一种约定;
　　它是需要二进制级别兼容的强约定,函数调用者和函数体如果使用不同的调用约定,将可能造成程序执行错误,必须把它看作是函数声明的一部分;

二：常见的函数调用约定;

VC6 中的函数调用约定;

调用约定	堆栈清除	参数传递
__cdecl	调用者	从右到左,通过堆栈传递
__stdcall	函数体	从右到左,通过堆栈传递
__fastcall	函数体	从右到左,优先使用寄存器(ECX,EDX),然后使用堆栈
thiscall	函数体	this 指针默认通过 ECX 传递,其它参数从右到左入栈

__cdecl 是 C/C++的默认调用约定; VC 的调用约定中并没有 thiscall 这个关键字,它是类成员函数默认调用约定;
C/C++中的 main(或 wmain)函数的调用约定必须是__cdecl,不允许更改;
默认调用约定一般能够通过编译器设置进行更改,如果你的代码依赖于调用约定,请明确指出需要使用的调用约定;

Delphi6 中的函数调用约定;

调用约定	堆栈清除	参数传递
register	函数体	从左到右,优先使用寄存器(EAX,EDX,ECX),然后使用堆栈

pascal	函数体	从左到右,通过堆栈传递
cdecl	调用者	从右到左,通过堆栈传递(与 C/C++默认调用约定兼容)
stdcall	函数体	从右到左,通过堆栈传递(与 VC 中的__stdcall 兼容)
safecall	函数体	从右到左,通过堆栈传递(同 stdcall)

Delphi 中的默认调用约定是 register,它也是我认为最有效率的一种调用方式,而 cdecl 是我认为综合效率最差的一种调用方式;

VC 中的__fastcall 调用约定一般比 register 效率稍差一些;

C++Builder6 中的函数调用约定;

调用约定	堆栈清除	参数传递
__fastcall	函数体	从左到右,优先使用寄存器(EAX,EDX,ECX),然后使用堆栈 (兼容 Delphi 的 register)
register	函数体	从左到右,优先使用寄存器(EAX,EDX,ECX),然后使用堆栈 (兼容 Delphi 的 register)
__pascal	函数体	从左到右,通过堆栈传递
__cdecl	调用者	从右到左,通过堆栈传递(与 C/C++默认调用约定兼容)
__stdcall	函数体	从右到左,通过堆栈传递(与 VC 中的__stdcall 兼容)
__msfastcall	函数体	从右到左,优先使用寄存器(ECX,EDX),然后使用堆栈(兼容 VC 的__fastcall)

常见的函数调用约定中,只有 cdecl 约定需要调用者来清除堆栈;

C/C++中的函数支持参数数目不定的参数列表,比如 printf 函数;由于函数体不知道调用者在堆栈中压入了多少参数,
所以函数体不能方便的知道应该怎样清除堆栈,那么最好的办法就是把清除堆栈的责任交给调用者; 这应该就是 cdecl 调用约定存在的原因吧;

VB 一般使用的是 stdcall 调用约定;(ps：有更强的保证吗)
Windows 的 API 中,一般使用的是 stdcall 约定;(ps: 有更强的保证吗)
建议在不同语言间的调用中(如 DLL)最好采用 stdcall 调用约定,因为它在语言间兼容性支持最好;

三:函数返回值传递方式

其实，返回值的传递从处理上也可以想象为函数调用的一个 out 形参数； 函数返回值传递方式也是函数调用约定的一部分;
有返回值的函数返回时：一般 int、指针等 32bit 数据值(包括 32bit 结构)通过 eax 传递，(bool,char 通过 al 传递，short 通过 ax 传递),

特别的__int64 等 64bit 结构(struct) 通过 edx,eax 两个寄存器来传递(同理：32bit 整形在 16bit 环境中通过 dx,ax 传递)；
其他大小的结构(struct)返回时把其地址通过 eax 返回；(所以返回值类型不是 1,2,4,8byte 时，效率可能比较差)
参数和返回值传递中，引用方式的类型可以看作与传递指针方式相同；
float/double(包括 Delphi 中的 extended)都是通过浮点寄存器 st(0)返回；

1.__cdecl
所谓的 C 调用规则。按从右至左的顺序压参数入栈，由调用者把参数弹出栈。切记：对于传送参数的内存栈是由调用者来维护的。
返回值在 EAX 中因此，对于象 printf 这样变参数的函数必须用这种规则。编译器在编译的时候对这种调用规则的函数生成修饰名的饿时候，仅在输出函数名前加上一个下划线前缀，格式为_functionname。

2.__stdcall
按从右至左的顺序压参数入栈，由被调用者把参数弹出栈。_stdcall 是 Pascal 程序的缺省调用方式，通常用于 Win32 Api 中，切记：函数自己在退出时清空堆栈，返回值在 EAX 中。
__stdcall 调用约定在输出函数名前加上一个下划线前缀，后面加上一个“@”符号和其参数的字节数，格式为_functionname@number。如函数 int func(int a, double b)的修饰名是_func@12。

3.__fastcall
__fastcall 调用的主要特点就是快，因为它是通过寄存器来传送参数的（实际上，它用 ECX 和 EDX 传送前两个双字（DWORD）或更小的参数，剩下的参数仍旧自右向左压栈传送，被调用的函数在返回前清理传送参数的内存栈）。__fastcall 调用约定在输出函数名前加上一个“@”符号，后面也是一个“@”符号和其参数的字节数，格式为@functionname@number。
这个和__stdcall 很象，唯一差别就是头两个参数通过寄存器传送。注意通过寄存器传送的两个参数是从左向右的，即第一个参数进 ECX，第 2 个进 EDX，其他参数是从右向左的入 stack。返回仍然通过 EAX.

4.__pascal
这种规则从左向右传递参数，通过 EAX 返回，堆栈由被调用者清除

5.__thiscall

仅仅应用于"C++"成员函数。this 指针存放于 CX 寄存器，参数从右到左压。thiscall 不是关键词，因此不能被程序员指定

调用约定可以通过工程设置：Setting...\C/C++ \Code Generation 项进行选择，缺省状态为__cdecl。

函數調用方式： Stdcall Cdecl Fastcall WINAPI CALLBACK PASCAL Thiscall Fortran Syscall Declspec(Naked)

http://www.cnitblog.com/textbox/archive/2010/03/10/64575.html

现代的编程语言的函数竟然有那麼多的调用方式。这些东西要完全理解还得通过汇编代码才好理解。他们各自有自己的特点
其实这些调用方式的差别在主要在一下几个方面

1.参数处理方式（传递顺序，存取(利用盏还是寄存器)）
2.函数的结尾处理方式（善后处理 如:栈的恢复由谁恢复? 函数内恢复/还是调用后恢复）

以下是理论：

__cdecl 由调用者平栈，参数从右到左依次入栈 是 C 和 C＋＋程序的缺省调用方式。每一个调用它的函数都包含清空堆栈的代码，所以产生的可执行文件大小会比调用_stdcall 函数的大。函数采用从右到左的压栈方式。VC 将函数编译后会在函数名前面加上下划线前缀。是 MFC 缺省调用约定

__stdcall ，WINAPI，CALLBACK ，PASCAL 由被调用者平栈，参数从右到左依次入栈 ._stdcall 是 Pascal 程序的缺省调用方式，通常用于 Win32 Api 中，函数采用从右到左的压栈方式，自己在退出时清空堆栈。VC 将函数编译后会在函数名前面加上下划线前缀，在函数名后加上"@"和参数的字节数

__fastcall 由被调用者平栈，参数先赋值给寄存器，然后入栈 “人”如其名，它的主要特点就是快，因为它是通过寄存器来传送参数的（实际上，它用 ECX 和 EDX 传送前两个双字（DWORD）或更小的参数，剩下的参数仍旧自右向左压栈传送，被调用的函数在返回前清理传送参数的内存栈），在函数名修饰约定方面，它和前两者均不同.
_fastcall 方式的函数采用寄存器传递参数，VC 将函数编译后会在函数名前面加上"@"前缀，在函数名后加上"@"和参数的字节数。

__thiscall 由被调用者平栈，参数入栈，this 指针赋给 ecx 寄存器 仅仅应用于“C++”成员函数。this 指针存放于 CX 寄存器，参数从右到左压。thiscall 不是关键词，因此不能被程序员指定。

__declspec(naked) 这是一个很少见的调用约定，一般程序设计者建议不要使用。编译器不会给这种函数增加初始化和清理代码，更特殊的是，你不能用 return 返回返回值，只能用插入汇编返回结果。这一般用于实模式驱动程序设计.

以下是实践：

```
int __stdcall test_stdcall(char para1, char para2)
{
  para1 = para2;
  return 0;
}
int __cdecl test_cdecl(char para, )
{
  char p = '\n';
  va_list marker;
  va_start( marker, para );
  while( p != '\0' )
  {
    p = va_arg( marker, char);
    printf("%c\n", p);
  }
```

```
  va_end( marker );
  return 0;
}

int pascal test_pascal(char para1, char para2)
{
  return 0;
}

int __fastcall test_fastcall(char para1, char para2, char para3, char para4)
{
  para1 = (char)1;
  para2 = (char)2;
  para3 = (char)3;
  para4 = (char)4;
  return 0;
}
__declspec(naked) void __stdcall test_naked(char para1, char para2)
{
  __asm
  {
    push ebp
    mov ebp, esp
    push eax
    mov al,byte ptr [ebp + 0Ch]
    xchg byte ptr [ebp + 8],al
    pop eax
    pop ebp
    ret 8
  }
//    return ;
}

int main( int argc, char* argv[ ] )
{
  test_stdcall( 'a', 'b' );
  test_cdecl( 'c', 'd', 'e', 'f', 'g', 'h', '\0' );
  test_pascal( 'e', 'f' );
  test_fastcall( 'g', 'h', 'i', 'j' );
  test_naked( 'k', 'l' );
  return 0;
}
```

汇编代码如下

```
int main(int argc, char* argv[])
{
00411350  push        ebp
00411351  mov         ebp,esp
00411353  sub         esp,0C0h
00411359  push        ebx
0041135A  push        esi
0041135B  push        edi
0041135C  lea         edi,[ebp-0C0h]
00411362  mov         ecx,30h
00411367  mov         eax,0CCCCCCCCh
0041136C  rep stos    dword ptr es:[edi]

    test_stdcall( 'a', 'b' );
0041136E  push        62h
00411370  push        61h
00411372  call        _test_stdcall@8

    test_cdecl( 'c','d','e','f','g' ,'h' ,'\0');
00411377  push        0
00411379  push        68h
0041137B  push        67h
0041137D  push        66h
0041137F  push        65h
00411381  push        64h
00411383  push        63h
00411385  call        _test_cdecl
0041138A  add         esp,1Ch ;恢复_test_cdecl 参数压入前的堆栈指令是: add esp,n*4 n=7, 参数的数量

    test_fastcall( 'g', 'h', 'i', 'j' );
0041138D  push        6Ah
0041138F  push        69h
00411391  mov         dl,68h
00411393  mov         cl,67h
```

```
00411395  call        test_fastcall

    test_naked( 'k', 'l');
0041139A  push        6Ch
0041139C  push        6Bh
0041139E  call        _test_naked

    return 0;
004113A3  xor         eax,eax
}

int __stdcall test_stdcall(char para1, char para2)
{
004111F0  push        ebp
004111F1  mov         ebp,esp
004111F3  sub         esp,0C0h

004111F9  push        ebx
004111FA  push        esi
004111FB  push        edi
004111FC  lea         edi,[ebp-0C0h]
00411202  mov         ecx,30h
00411207  mov         eax,0CCCCCCCCh
0041120C  rep stos    dword ptr es:[edi] ;初始 edi
    para1 = para2;
0041120E  mov         al,byte ptr [para2] ;mov al,byte ptr[ebp+c]
00411211  mov         byte ptr [para1],al ;mov byte ptr[ebp+8],al
    return 0;
00411214  xor         eax,eax
00411216  pop         edi
00411217  pop         esi
00411218  pop         ebx

00411219  mov         esp,ebp
0041121B  pop         ebp
0041121C  ret          8 ;恢复到压入函数参数前堆栈,由于有两个参数所以 ret 8 相当于 pop eip 然后 esp+8
}

int __cdecl test_cdecl(char para,... )
{
00411230  push        ebp
00411231  mov         ebp,esp
```

```
00411233  sub         esp,0D8h
0041123C  lea         edi,[ebp-0D8h]
00411242  mov         ecx,36h
00411247  mov         eax,0CCCCCCCCh
0041124C  rep stos    dword ptr es:[edi]
    char    p = '\n';
0041124E  mov         byte ptr [p],0Ah
    va_list marker;
    va_start( marker, para );
00411252  lea         eax,[ebp+0Ch]
00411255  mov         dword ptr [marker],eax
    while( p != '\0' )
00411258  movsx       eax,byte ptr [p]
0041125C  test        eax,eax
0041125E  je          test_cdecl+60h (411290h)
    {
        p = va_arg( marker, char);
00411260  mov         eax,dword ptr [marker]
00411263  add         eax,4
00411266  mov         dword ptr [marker],eax
00411269  mov         ecx,dword ptr [marker]
0041126C  mov         dl,byte ptr [ecx-4]
0041126F  mov         byte ptr [p],dl
       printf("%c\n", p);
00411272  movsx       eax,byte ptr [p]
00411276  mov         esi,esp
00411278  push        eax
00411279  push        offset string "%c\n" (41401Ch)
0041127E  call        dword ptr [__imp__printf (416180h)]
00411284  add         esp,8
0041128E  jmp         test_cdecl+28h (411258h)
    }
    va_end( marker );
00411290  mov         dword ptr [marker],0
    return 0;
00411297  xor         eax,eax
004112A9  mov         esp,ebp
004112AB  pop         ebp
004112AC  ret
}

int __fastcall test_fastcall(char para1, char para2, char para3, char para4)
```

```
{
004112D0  push        ebp
004112D1  mov         ebp,esp
004112D3  sub         esp,0D8h
004112DD  lea         edi,[ebp-0D8h]
004112E3  mov         ecx,36h
004112E8  mov         eax,0CCCCCCCCh
004112ED  rep stos    dword ptr es:[edi]
004112EF  pop         ecx
004112F0  mov         byte ptr [ebp-14h],dl
004112F3  mov         byte ptr [ebp-8],cl
    para1 = (char)1;
004112F6  mov         byte ptr [para1],1
    para2 = (char)2;
004112FA  mov         byte ptr [para2],2
    para3 = (char)3;
004112FE  mov         byte ptr [para3],3
    para4 = (char)4;
00411302  mov         byte ptr [para4],4
    return 0;
00411306  xor         eax,eax
0041130B  mov         esp,ebp
0041130D  pop         ebp
0041130E  ret         8  ;由于使用了 ecx ,edx 传递参数 本来 4 个参数只使用两 push 所以这里是 ret 4*2
}


__declspec(naked) void __stdcall test_naked(char para1, char para2)
{
00411330  push        ebp       ;这里编译器没加入任何初始化和清栈的指令,你代码如何写它就复制过来
00411331  mov         ebp,esp
00411333  push        eax
00411334  mov         al,byte ptr [para2]
00411337  xchg        al,byte ptr [para1]
0041133A  pop         eax
0041133B  pop         ebp
0041133C  ret         8
}
```

http://securityetalii.es/2013/01/20/calling-conventions-hunting/

Calling Conventions Hunting

Posted on 20/01/2013 by Adrián — Leave a comment

When trying to understand a binary, it's key to be able to identify functions, and with them, their parameters and local variables. This will help the reverser figuring out APIs, data structures, etc. In short, gaining a deep understanding of the software. When dealing with functions, it's essential to be able to identify the calling convention in use, as many times that will allow the reverser to perform educated guesses on the arguments and local variables used by the function. I'll try to describe here a couple of points that may aid in identifying the calling convention of any given function and the number and ordering of its parameters.

Calling Conventions

A calling convention defines how functions are called in a program. They influence how data (arguments/variables) is laid on the stack when the function call takes place. A comprehensive definition of calling conventions is beyond the scope of this blog, nonetheless the most common ones are briefly described below.

cdecl

Description: Standard C/C++ calling convention. Allows functions to receive a dynamic number of parameters.

Cleans the stack: The caller is responsible for restoring the stack after making a function call.

Arguments passed: On the stack. Arguments are received in reverse order (i.e. from right to left). This is because the first argument is pushed onto the stack first, and the last is pushed last.

```
void _cdecl fun();
```

fastcall

Description: Slightly better performance calling convention.

Cleans the stack: The callee is responsible for restoring the stack before returning.

Arguments passed: First two arguments are passed in registers (ECX and EDX). The rest are passed through the stack.

```
void __fastcall func();
```

stdcall

Description: Very common in Windows (used by most APIs).

Cleans the stack: The callee is responsible for cleaning up the stack before returning. Usually by means of a RETN #N instruction.

Arguments passed: On the stack. Arguments received from left to right (opposite to cdecl). First argument is pushed last.

```
void __stdcall fun();
```

thiscall

Description: Used when C++ method with a static number of parameters is called. Specially thought to improve performance of OO languages (saves EDX for the *this* pointer with VC++. GCC pushes the *this* pointer onto the stack last). When a dynamic number of parameters is required, compilers usually fall back to cdecl and pass the *this* pointer as the first parameter on the stack.

Cleans the stack: In GCC, caller cleans the stack. In Microsoft VC++ the callee is responsible for cleaning up.

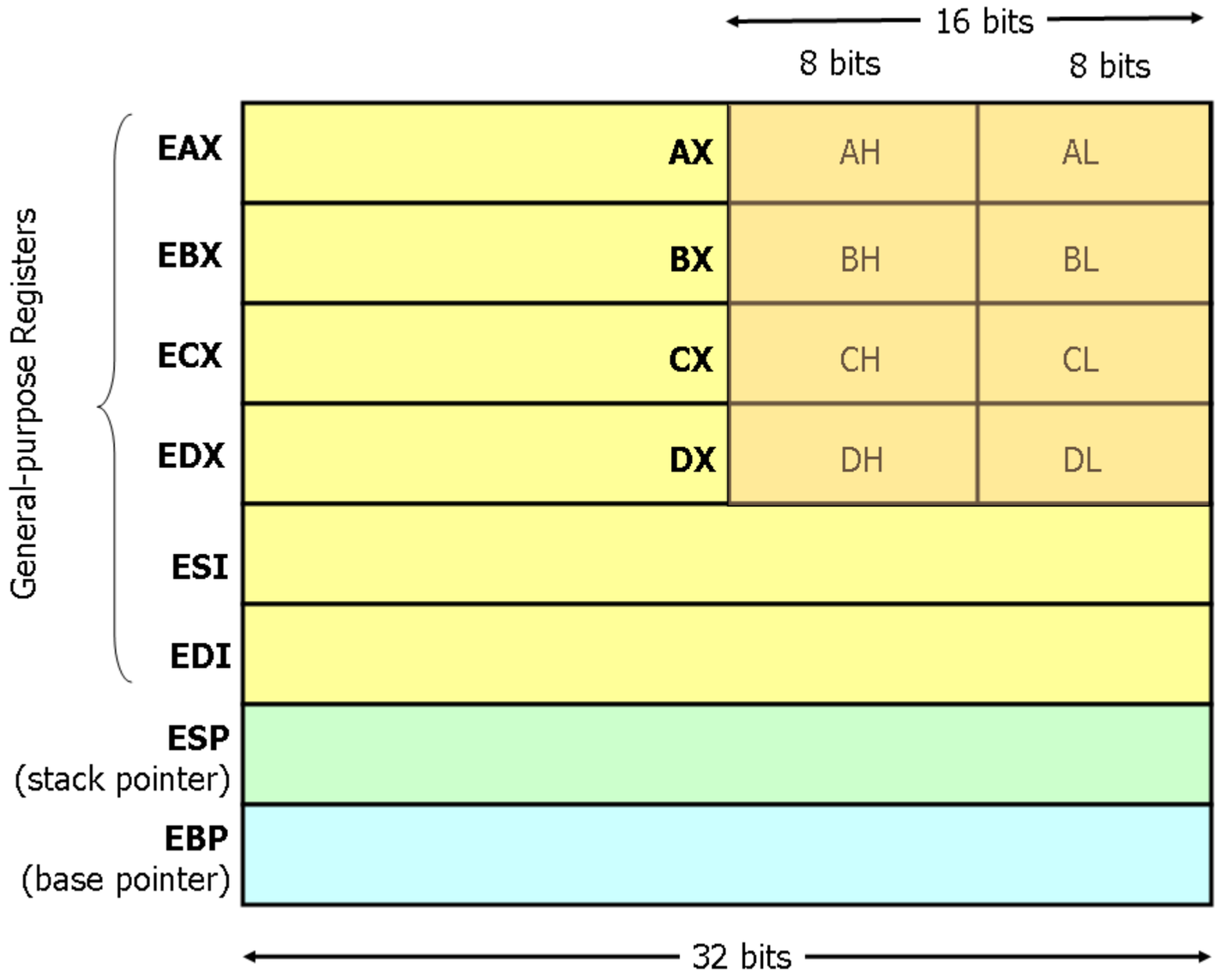
Arguments passed: From right to left (as cdecl). First argument is pushed first, and last argument is pushed last.

```
void __thiscall func();
```

Let the small table below serve as a quick reminder.

	Cleans Stack	Arguments	Arg Ordering
cdecl	Caller	On the Stack	Right-to-left
fastcall	Callee	ECX,EDX, then stack	Left-to-Right
stdcall	Callee	On the Stack	Left-to-Right
VC++ thiscall	Callee	EDX (this), then stack	Right-to-left
GCC thiscall	Caller	On the Stack (this pointer first)	Right-to-left

http://www.cs.virginia.edu/~evans/cs216/guides/x86.html



Calling Convention

To allow separate programmers to share code and develop libraries for use by many programs,

and to simplify the use of subroutines in general, programmers typically adopt a common *calling convention*.

The calling convention is a protocol about how to call and return from routines.

For example, given a set of calling convention rules, a programmer need not examine the definition of a subroutine to determine

how parameters should be passed to that subroutine.

Furthermore, given a set of calling convention rules, high-level language compilers can be made to follow the rules,

thus allowing hand-coded assembly language routines and high-level language routines to call one another.

In practice, many calling conventions are possible.

We will use the widely used C language calling convention.

Following this convention will allow you to write assembly language subroutines that are safely callable from C (and C++) code,

and will also enable you to call C library functions from your assembly language code.

The C calling convention is based heavily on the use of the hardware-supported stack.

It is based on the push, pop, call, and ret instructions.

Subroutine parameters are passed on the stack.

Registers are saved on the stack, and local variables used by subroutines are placed in memory on the stack.

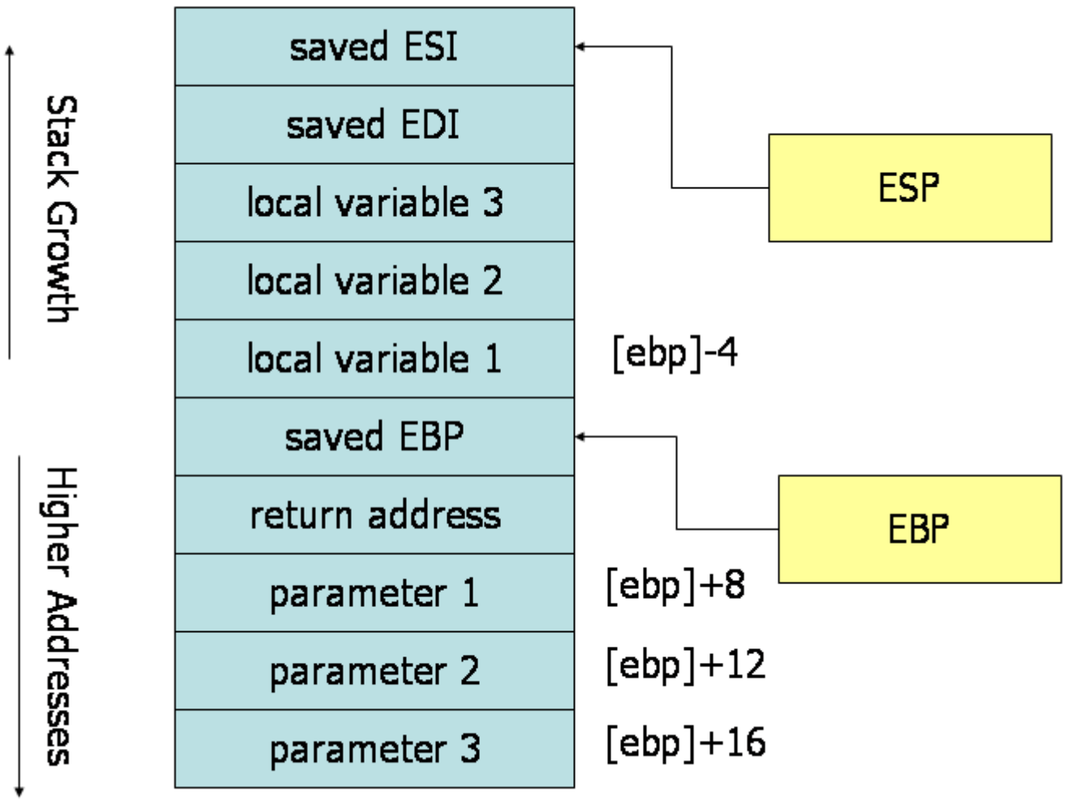
The vast majority of high-level procedural languages implemented on most processors have used similar calling conventions.

The calling convention is broken into two sets of rules.

The first set of rules is employed by the caller of the subroutine, and the second set of rules is observed by the writer of the subroutine (the callee).

It should be emphasized that mistakes in the observance of these rules quickly result in fatal program errors

since the stack will be left in an inconsistent state; thus meticulous care should be used when implementing the call convention in your own subroutines.



A good way to visualize the operation of the calling convention is to draw the contents of the nearby region of the stack during subroutine execution. The image above depicts the contents of the stack during the execution of a subroutine with three parameters and three local variables. The cells depicted in the stack are 32-bit wide memory locations, thus the memory addresses of the cells are 4 bytes apart. The first parameter resides at an offset of 8 bytes from the base pointer. Above the parameters on the stack (and below the base pointer), the `call` instruction placed the return address, thus leading to an extra 4 bytes of offset from the base pointer to the first parameter. When the `ret` instruction is used to return from the subroutine, it will jump to the return address stored on the stack.

Caller Rules

To make a subrouting call, the caller should:

1. Before calling a subroutine, the caller should save the contents of certain registers that are designated *caller-saved*. The caller-saved registers are EAX, ECX, EDX. Since the called subroutine is allowed to modify these registers, if the caller relies on their values after the subroutine returns, the caller must push the values in these registers onto the stack (so they can be restore after the subroutine returns.
2. To pass parameters to the subroutine, push them onto the stack before the call. The parameters should be pushed in inverted order (i.e. last parameter first). Since the stack grows down, the first parameter will be stored at the lowest address (this inversion of parameters was historically used to allow functions to be passed a variable number of parameters).
3. To call the subroutine, use the call instruction. This instruction places the return address on top of the parameters on the stack, and branches to the subroutine code. This invokes the subroutine, which should follow the callee rules below.

After the subroutine returns (immediately following the `call` instruction), the caller can expect to find the return value of the subroutine in the register EAX. To restore the machine state, the caller should:

1. Remove the parameters from stack. This restores the stack to its state before the call was performed.
2. Restore the contents of caller-saved registers (EAX, ECX, EDX) by popping them off of the stack. The caller can assume that no other registers were modified by the subroutine.

Example

The code below shows a function call that follows the caller rules. The caller is calling a function *_myFunc* that takes three integer parameters. First parameter is in EAX, the second parameter is the constant 216; the third parameter is in memory location *var*.

```
push [var] ; Push last parameter first
push 216   ; Push the second parameter
push eax   ; Push first parameter last

call _myFunc ; Call the function (assume C naming)
add esp, 12
```

Note that after the call returns, the caller cleans up the stack using the `add` instruction.

We have 12 bytes (3 parameters * 4 bytes each) on the stack, and the stack grows down.

Thus, to get rid of the parameters, we can simply add 12 to the stack pointer.

The result produced by *_myFunc* is now available for use in the register EAX.

The values of the caller-saved registers (ECX and EDX), may have been changed.

If the caller uses them after the call, it would have needed to save them on the stack before the call and restore them after it.

Callee Rules

The definition of the subroutine should adhere to the following rules at the beginning of the subroutine:

1. Push the value of EBP onto the stack, and then copy the value of ESP into EBP using the following instructions:

```
push ebp
mov  ebp, esp
```

This initial action maintains the *base pointer*, EBP. The base pointer is used by convention as a point of reference for finding parameters and local variables on the stack.
When a subroutine is executing, the base pointer holds a copy of the stack pointer value from when the subroutine started executing.
Parameters and local variables will always be located at known, constant offsets away from the base pointer value.

We push the old base pointer value at the beginning of the subroutine so that we can later restore the appropriate base pointer value for the caller when the subroutine returns.
Remember, the caller is not expecting the subroutine to change the value of the base pointer.
We then move the stack pointer into EBP to obtain our point of reference for accessing parameters and local variables.

3. Next, allocate local variables by making space on the stack. Recall, the stack grows down, so to make space on the top of the stack, the stack pointer should be decremented.
   The amount by which the stack pointer is decremented depends on the number and size of local variables needed.
   For example, if 3 local integers (4 bytes each) were required, the stack pointer would need to be decremented by 12 to make space for these local variables
   (i.e., sub esp, 12). As with parameters, local variables will be located at known offsets from the base pointer.

4. Next, save the values of the *callee-saved* registers that will be used by the function must be saved.
   To save registers, push them onto the stack. The callee-saved registers are EBX, EDI, and ESI
   (ESP and EBP will also be preserved by the calling convention, but need not be pushed on the stack during this step).

After these three actions are performed, the body of the subroutine may proceed. When the subroutine is returns, it must follow these steps:

1. Leave the return value in EAX.

2. Restore the old values of any callee-saved registers (EDI and ESI) that were modified.
   The register contents are restored by popping them from the stack. The registers should be popped in the inverse order that they were pushed.

3. Deallocate local variables.
   The obvious way to do this might be to add the appropriate value to the stack pointer (since the space was allocated by subtracting the needed amount from the stack pointer).
   In practice, a less error-prone way to deallocate the variables is to move the value in the base pointer into the stack pointer:
   mov esp, ebp.
   This works because the base pointer always contains the value that the stack pointer contained immediately prior to the allocation of the local variables.

4. Immediately before returning, restore the caller's base pointer value by popping EBP off the stack.
   Recall that the first thing we did on entry to the subroutine was to push the base pointer to save its old value.

5. Finally, return to the caller by executing a ret instruction. This instruction will find and remove the appropriate return address from the stack.

Note that the callee's rules fall cleanly into two halves that are basically mirror images of one another.

The first half of the rules apply to the beginning of the function, and are commonly said to define the *prologue* to the function.

The latter half of the rules apply to the end of the function, and are thus commonly said to define the *epilogue* of the function.

Example
Here is an example function definition that follows the callee rules:

```
.486
.MODEL FLAT
.CODE
PUBLIC _myFunc
_myFunc PROC
  ; Subroutine Prologue
  push ebp     ; Save the old base pointer value.
  mov ebp, esp ; Set the new base pointer value.
  sub esp, 4   ; Make room for one 4-byte local variable.
  push edi     ; Save the values of registers that the function
  push esi     ; will modify. This function uses EDI and ESI.
  ; (no need to save EBX, EBP, or ESP)

  ; Subroutine Body
  mov eax, [ebp+8]   ; Move value of parameter 1 into EAX
  mov esi, [ebp+12]  ; Move value of parameter 2 into ESI
  mov edi, [ebp+16]  ; Move value of parameter 3 into EDI

  mov [ebp-4], edi   ; Move EDI into the local variable
  add [ebp-4], esi   ; Add ESI into the local variable
  add eax, [ebp-4]   ; Add the contents of the local variable
                     ; into EAX (final result)

  ; Subroutine Epilogue
  pop esi      ; Recover register values
  pop  edi
  mov esp, ebp ; Deallocate local variables
  pop ebp ; Restore the caller's base pointer value
  ret
_myFunc ENDP
END
```

The subroutine prologue performs the standard actions of saving a snapshot of the stack pointer in EBP (the base pointer),
allocating local variables by decrementing the stack pointer, and saving register values on the stack.

In the body of the subroutine we can see the use of the base pointer.
Both parameters and local variables are located at constant offsets from the base pointer for the duration of the subroutines execution.
In particular, we notice that since parameters were placed onto the stack before the subroutine was called, they are always located below the base pointer (i.e. at higher addresses) on the stack.
The first parameter to the subroutine can always be found at memory location [EBP+8], the second at [EBP+12], the third at [EBP+16].

Similarly, since local variables are allocated after the base pointer is set, they always reside above the base pointer (i.e. at lower addresses) on the stack.

In particular, the first local variable is always located at [EBP-4], the second at [EBP-8], and so on.

This conventional use of the base pointer allows us to quickly identify the use of local variables and parameters within a function body.

The function epilogue is basically a mirror image of the function prologue.
The caller's register values are recovered from the stack, the local variables are deallocated by resetting the stack pointer,
the caller's base pointer value is recovered, and the ret instruction is used to return to the appropriate code location in the caller.

函数调用约定（Calling Convention）

函数调用约定不仅决定了发生函数调用时函数参数的入栈顺序，还决定了是由调用者函数还是被调用函数负责清除栈中的参数，还原堆栈。函数调用约定有很多方 式，除了常见的__cdecl，__fastcall 和__stdcall 之外，C++的编译器还支持 thiscall 方式，不少 C/C++编译器还支持 naked call 方式。这么多函数调用约定常常令许多程序员很迷惑，到底它们是怎么回事，都是在什么情况下使用呢？下面就分别介绍这几种函数调用约定。

1.__cdecl

编译器的命令行参数是/Gd。__cdecl 方式是 C/C++编译器默认的函数调用约定，所有非 C++成员函数和那些没有用__stdcall 或 __fastcall 声明的函数都默认是__cdecl 方式，它使用 C 函数调用方式，函数参数按照从右向左的顺序入栈，函数调用者负责清除栈中的参数， 由于每次函数调用都要由编译器产生清除（还原）堆栈的代码，所以使用__cdecl 方式编译的程序比使用__stdcall 方式编译的程序要大很多，但是 __cdecl 调用方式是由函数调用者负责清除栈中的函数参数，所以这种方式支持可变参数，比如 printf 和 windows 的 API wsprintf 就是__cdecl 调用方式。对于 C 函数，__cdecl 方式的名字修饰约定是在函数名称前添加一个下划线；对于 C++函数，除非特别使 用 extern "C"，C++函数使用不同的名字修饰方式。

2.__fastcall

编译器的命令行参数是/Gr。__fastcall 函数调用约定在可能的情况下使用寄存器传递参数，通常是前两个 DWORD 类型的参数或较小的参数使用 ECX 和 EDX 寄存器传递，其余参数按照从右向左的顺序入栈，被调用函数在返回之前负责清除栈中的参数。编译器使用 两个@修饰函数名字，后跟十进制数表示的函数参数列表大小，例如：@function_name@number。需要注意的是__fastcall 函数调 用约定在不同的编译器上可能有不同的实现，比如 16 位的编译器和 32 位的编译器，另外，在使用内嵌汇编代码时，还要注意不能和编译器使用的寄存器有冲突。

3.__stdcall

编译器的命令行参数是/Gz，__stdcall 是 Pascal 程序的缺省调用方式，大多数 Windows 的 API 也是__stdcall 调用约定。 __stdcall 函数调用约定将函数参数从右向左入栈，除非使用指针或引用类型的参数，所有参数采用传值方式传递，由被调用函数负责清除栈中的参数。对 于 C 函数，__stdcall 的名称修饰方式是在函数名字前添加下划线，在函数名字后添加@和函数参数的大小，例 如：_functionname@number

4.thiscall

thiscall 只用在 C++成员函数的调用，函数参数按照从右向左的顺序入栈，类实例的 this 指针通过 ECX 寄存器传递。需要注意的是 thiscall 不是 C++的关键字，不能使用 thiscall 声明函数，它只能由编译器使用。

5.naked call

采用前面几种函数调用约定的函数，编译器会在必要的时候自动在函数开始添加保存 ESI，EDI，EBX，EBP 寄存器的代码，在退出函数时恢复这些寄存器 的内容，使用 naked call 方式声明的函数不会添加这样的代码，这也就是为什么称其为 naked 的原因吧。naked call 不是类型修饰符，故必须和_declspec 共同使用。

VC 的编译环境默认是使用__cdecl 调用约定，也可以在编译环境的 Project Setting...菜单－》C/C++ ＝》Code Generation 项选择设置函数调用约定。也可以直接在函数声明前添加关键字__stdcall、__cdecl 或__fastcall 等单独确定函 数的调用方式。在 Windows 系统上开发软件常用到 WINAPI 宏，它可以根据编译设置翻译成适当的函数调用约定，在 WIN32 中，它被定义为 __stdcall。

关键字 __stdcall、__cdecl 和__fastcall 可以直接加在要输出的函数前，也可以在编译环境的 Setting...\C/C++ \Code Generation 项选择。当加在输出函数前的关键字与编译环境中的选择不同时，直接加在输出函数前的关键字有效。它们对应的命令行参数分别为/Gz、 /Gd 和/Gr。缺省状态为/Gd，即__cdecl。

要完全模仿 PASCAL 调 用约定首先必须使用__stdcall 调用约定，至于函数名修饰约定，可以通过其它方法模仿。还有一个值得一提的是 WINAPI 宏，Windows.h 支 持该宏，它可以将出函数翻译成适当的调用约定，在 WIN32 中，它被定义为__stdcall。使用 WINAPI 宏可以创建自己的 APIs。

名字修饰约定

1、修饰名(Decoration name)

“C”或者“C++”函数在内部（编译和链接）通过修饰名识别。修饰名是编译器在编译函数定义或者原型时生成的字符串。有些情况下使用函数的修饰名是必要 的，如在模块定义文件里头指定输出“C++”重载函数、构造函数、析构函数，又如在汇编代码里调用“C””或“C++”函数等。

修饰名由函数名、类名、调用约定、返回类型、参数等共同决定。

2、名字修饰约定随调用约定和编译种类(C 或 C++)的不同而变化。函数名修饰约定随编译种类和调用约定的不同而不同，下面分别说明。

a、C 编译时函数名修饰约定规则：

__stdcall 调用约定在输出函数名前加上一个下划线前缀，后面加上一个“@”符号和其参数的字节数，格式为_functionname@number。

__cdecl 调用约定仅在输出函数名前加上一个下划线前缀，格式为_functionname。

__fastcall 调用约定在输出函数名前加上一个“@”符号，后面也是一个“@”符号和其参数的字节数，格式为@functionname@number。

它们均不改变输出函数名中的字符大小写，这和 PASCAL 调用约定不同，PASCAL 约定输出的函数名无任何修饰且全部大写。

b、C++编译时函数名修饰约定规则：

__stdcall 调用约定：
1、以“?”标识函数名的开始，后跟函数名；
2、函数名后面以“@@YG”标识参数表的开始，后跟参数表；
3、参数表以代号表示：
X--void ，
D--char，
E--unsigned char，
F--short，
H--int，
I--unsigned int，
J--long，
K--unsigned long，
M--float，
N--double，
_N--bool，
....
PA--表示指针，后面的代号表明指针类型，如果相同类型的指针连续出现，以“0”代替，一个“0”代表一次重复；
4、参数表的第一项为该函数的返回值类型，其后依次为参数的数据类型,指针标识在其所指数据类型前；
5、参数表后以“@Z”标识整个名字的结束，如果该函数无参数，则以“Z”标识结束。

其格式为“?functionname@@YG*****@Z”或“?functionname@@YG*XZ”，例如
int Test1（char *var1,unsigned long）-----“?Test1@@YGHPADK@Z”
void Test2（） -----“?Test2@@YGXXZ”

__cdecl 调用约定：
规则同上面的_stdcall 调用约定，只是参数表的开始标识由上面的“@@YG”变为“@@YA”。

__fastcall 调用约定：
规则同上面的_stdcall 调用约定，只是参数表的开始标识由上面的“@@YG”变为“@@YI”。

VC++对函数的省缺声明是"__cedcl",将只能被 C/C++调用.

CB 在输出函数声明时使用 4 种修饰符号
//__cdecl
cb 的默认值，它会在输出函数名前加_，并保留此函数名不变，参数按照从右到左的顺序依次传递给栈，也可以写成_cdecl 和 cdecl 形式。

//__fastcall
她修饰的函数的参数将尽肯呢感地使用寄存器来处理，其函数名前加@，参数按照从左到右的顺序压栈；
//__pascal
它说明的函数名使用 Pascal 格式的命名约定。这时函数名全部大写。参数按照从左到右的顺序压栈；
//__stdcall
使用标准约定的函数名。函数名不会改变。使用__stdcall 修饰时。参数按照由右到左的顺序压栈，也可以是_stdcall；
http://blog.csdn.net/eqera/article/details/7713781

__cdecl 调用约定

规则同上面的_stdcall 调用约定，只是参数表的开始标识由上面的"@@YG"变为"@@YA"。
__fastcall 调用约定：
规则同上面的_stdcall 调用约定，只是参数表的开始标识由上面的"@@YG"变为"@@YI"。
VC++对函数的省缺声明是"__cedcl",将只能被 C/C++调用.

注意：
1、_beginthread 需要__cdecl 的线程函数地址，_beginthreadex 和 CreateThread 需要__stdcall 的线程函数地址。
2、一般 WIN32 的函数都是__stdcall。而且在 Windef.h 中有如下的定义：
#define CALLBACK __stdcall
#define WINAPI __stdcall
3、extern "C" _declspec(dllexport) int __cdecl Add(int a, int b);
typedef int (__cdecl*FunPointer)(int a, int b);
修饰符的书写顺序如上。
4、extern "C"的作用：如果 Add(int a, int b)是在 c 语言编译器编译，而在 c++文件使用，则需要在 c++文件中声明：extern "C" Add(int a, int b)，因为 c 编译器和 c++编译器对函数名的解释不一样（c++编译器解释函数名的时候要考虑函数参数，这样是了方便函数重载，而在 c 语言中不存在函数重 载的问题），使用 extern "C"，实质就是告诉 c++编译器，该函数是 c 库里面的函数。如果不使用 extern "C"则会出现链接错误。
一般象如下使用：

```
#ifdef _cplusplus
#define ETERN_C extern "C"
#else
#define EXTERN_C extern
#endif
#ifdef _cplusplus
extern "C"{
#endif
EXTERN_C int func(int a, int b);
#ifdef _cplusplus
}
#endif
```

5、MFC 提供了一些宏，可以使用 AFX_EXT_CLASS 来代替__declspec(DLLexport)，并修饰类名，从而导出类，AFX_API_EXPORT 来修饰函数，AFX_DATA_EXPORT 来修饰变量
AFX_CLASS_IMPORT：__declspec(DLLexport)
AFX_API_IMPORT：__declspec(DLLexport)

AFX_DATA_IMPORT：__declspec(DLLexport)
AFX_CLASS_EXPORT：__declspec(DLLexport)
AFX_API_EXPORT：__declspec(DLLexport)
AFX_DATA_EXPORT：__declspec(DLLexport)
AFX_EXT_CLASS：#ifdef _AFXEXT
AFX_CLASS_EXPORT
#else
AFX_CLASS_IMPORT
6、DLLMain 负责初始化(Initialization) 和结束(Termination)工作，每当一个新的进程或者该进程的新的线程访问 DLL 时，或者访问 DLL 的每一个进程或者线程不再使用 DLL 或者结束 时，都会调用 DLLMain。但是，使用 TerminateProcess 或 TerminateThread 结束进程或者线程，不会调用 DLLMain。
7、一个 DLL 在内存中只有一个实例
DLL 程序和调用其输出函数的程序的关系：
1)、DLL 与进程、线程之间的关系
DLL 模块被映射到调用它的进程的虚拟地址空间。
DLL 使用的内存从调用进程的虚拟地址空间分配，只能被该进程的线程所访问。
DLL 的句柄可以被调用进程使用；调用进程的句柄可以被 DLL 使用。
DLLDLL 可以有自己的数据段，但没有自己的堆栈，使用调用进程的栈，与调用它的应用程序相同的堆栈模式。
2)、关于共享数据段
DLL 定义的全局变量可以被调用进 程访问；DLL 可以访问调用进程的全局数据。使用同一 DLL 的每一个进程都有自己的 DLL 全局变量实例。如果多个线程并发访问同一变量，则需要使用同步机 制；对一个 DLL 的变量，如果希望每个使用 DLL 的线程都有自己的值，则应该使用线程局部存储(TLS，Thread Local Strorage)。
http://blog.csdn.net/jia_xiaoxin/article/details/2868216

Linux 下如何指定调用约定(calling convention)

Windows 下的调用约定可以是 stdcall/cdecl/fastcall，这些标识加在函数名前面，如：
int __stdcall funca()
但在 Linux 下，如按照上面写法后，编译程序将导致编译错误，Linux 下正确的语法如下：
int __attribute__((__stdcall__)) funca()
int __attribute__((__cdecl__)) funca()
Linux 下如果函数不指定调用约定，默认的情况应该是__attribute__((__cdecl__))
int __attribute__((__cdecl__)) myfunc(int i, int j, int k)
{
return i+j+k;
}
080483d0 <myfunc>:
80483d0: 55 push %ebp
80483d1: 89 e5 mov %esp,%ebp
80483d3: 8b 45 0c mov 0xc(%ebp),%eax
80483d6: 03 45 08 add 0x8(%ebp),%eax
80483d9: 03 45 10 add 0x10(%ebp),%eax
80483dc: 5d pop %ebp

```
 80483dd:    c3                   ret
 80483de:    66 90                xchg   %ax,%ax
int __attribute__((__stdcall__)) myfunc(int i, int j, int k)
{
    return i+j+k;
}
 080483d0 <myfunc>:
 80483d0:    55                   push   %ebp
 80483d1:    89 e5                mov    %esp,%ebp
 80483d3:    8b 45 0c             mov    0xc(%ebp),%eax
 80483d6:    03 45 08             add    0x8(%ebp),%eax
 80483d9:    03 45 10             add    0x10(%ebp),%eax
 80483dc:    5d                   pop    %ebp
 80483dd:    c2 0c 00             ret    $0xc
```

http://www.linuxidc.com/Linux/2011-04/35032.htm

ARM 调用约定 calling convention

```
int bar( int a, int b, int c, int d, int e, int f, int g )
{
  int array2[ 7 ];
  array2[ 0 ] = a + b;
  array2[ 1 ] = b + c;
  array2[ 2 ] = c + d;
  array2[ 3 ] = d + e;
  array2[ 4 ] = e + f;
  array2[ 5 ] = f + g;
  array2[ 6 ] = g + a;
  return array2[ 0 ] + array2[ 1 ] + array2[ 2 ] + array2[ 3 ] + array2[ 4 ]
    + array2[ 5 ] + array2[ 6 ];
}

int foo( int a, int b, int c, int d, int e, int f, int g )
{
  int array1[ 7 ];
  array1[ 0 ] = a + b;
  array1[ 1 ] = b + c;
  array1[ 2 ] = c + d;
  array1[ 3 ] = d + e;
  array1[ 4 ] = e + f;
  array1[ 5 ] = f + g;
```

```
  array1[ 6 ] = g + a;


  return bar( array1[ 0 ], array1[ 1 ], array1[ 2 ], array1[ 3 ], array1[ 4 ],
    array1[ 5 ], array1[ 6 ] );
}

int func( int * buffer )
{
  int a, b, c, d, e, f, g, h;
  a = 1 + buffer[ 0 ];
  b = 2 + buffer[ 1 ];
  c = 3 + buffer[ 2 ];
  d = 4 + buffer[ 3 ];
  e = 5 + buffer[ 4 ];
  f = 6 + buffer[ 5 ];
  g = 7 + buffer[ 6 ];
  h = foo( a, b, c, d, e, f, g );

  return h;
}

int main( )
{
  int array0[ 7 ];
  array0[ 0 ] = 0;
  array0[ 1 ] = 1;
  array0[ 2 ] = 2;
  array0[ 3 ] = 3;
  array0[ 4 ] = 4;
  array0[ 5 ] = 5;
  array0[ 6 ] = 6;
  while ( func( array0 ) )

  {
  }
}
```

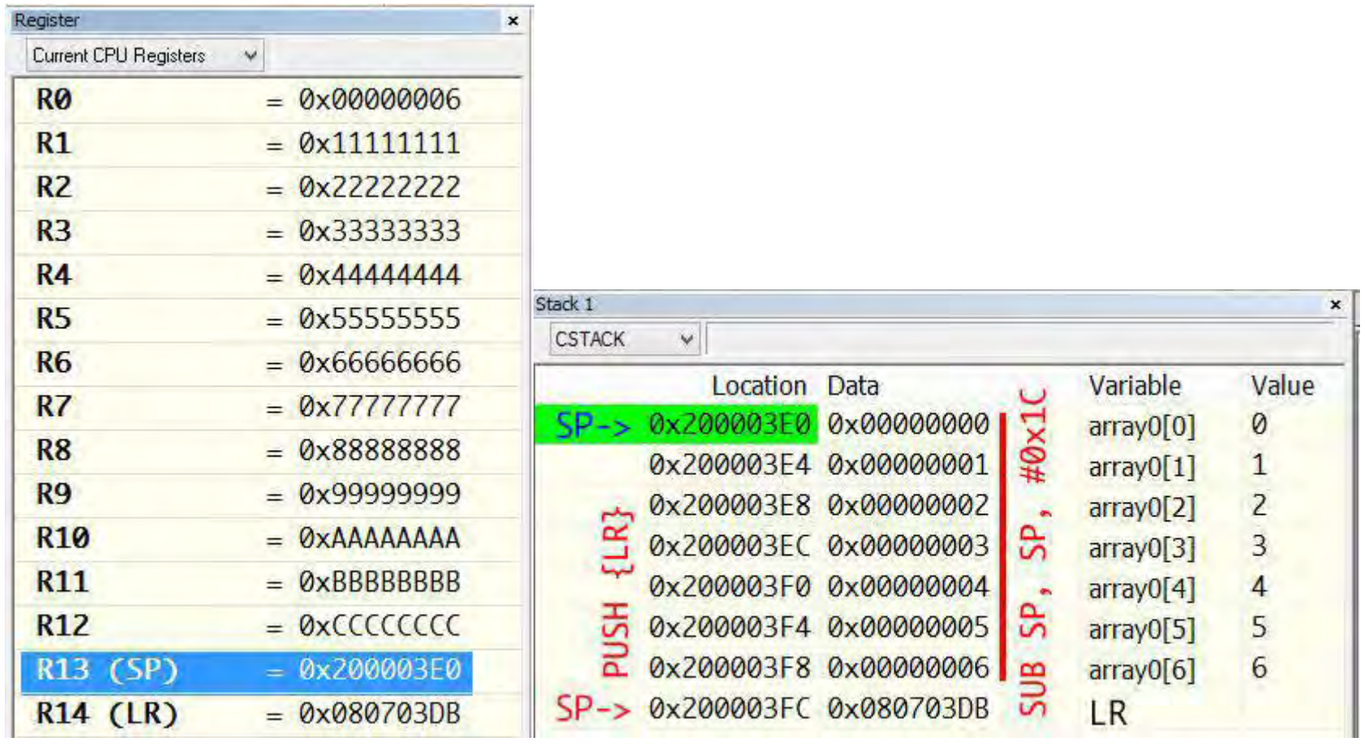
Register

Current CPU Registers

R0	= 0x00000000
R1	= 0x11111111
R2	= 0x22222222
R3	= 0x33333333
R4	= 0x44444444
R5	= 0x55555555
R6	= 0x66666666
R7	= 0x77777777
R8	= 0x88888888
R9	= 0x99999999
R10	= 0xAAAAAAAA
R11	= 0xBBBBBBBB
R12	= 0xCCCCCCCC
R13 (SP)	= 0x20000400
R14 (LR)	= 0x080703DB
⊞APSR	= 0x60000000
⊞IPSR	= 0x00000000
⊞EPSR	= 0x01000000
PC	= 0x08070252

```
int main( )
{
main:
    0x8070252: 0xb500          PUSH      {LR}
    0x8070254: 0xb087          SUB       SP, SP, #0x1c
  array0[0] = 0;
    0x8070256: 0x2000          MOVS      R0, #0
    0x8070258: 0x9000          STR       R0, [SP]
  array0[1] = 1;

    0x807025a: 0x2001          MOVS      R0, #1
    0x807025c: 0x9001          STR       R0, [SP, #0x4]
  array0[2] = 2;
```

```
    0x807025e: 0x2002          MOVS      R0, #2
    0x8070260: 0x9002          STR       R0, [SP, #0x8]
  array0[3] = 3;
    0x8070262: 0x2003          MOVS      R0, #3
    0x8070264: 0x9003          STR       R0, [SP, #0xc]
  array0[4] = 4;
    0x8070266: 0x2004          MOVS      R0, #4
    0x8070268: 0x9004          STR       R0, [SP, #0x10]
  array0[5] = 5;
    0x807026a: 0x2005          MOVS      R0, #5
    0x807026c: 0x9005          STR       R0, [SP, #0x14]
  array0[6] = 6;
    0x807026e: 0x2006          MOVS      R0, #6
    0x8070270: 0x9006          STR       R0, [SP, #0x18]
  while ( func(array0) )
??main_0:
    0x8070272: 0xa800          ADD       R0, SP, #0x0
    0x8070274: 0xf7ff 0xffc3   BL        func                    ; 0x80701fe
    0x8070278: 0x2800          CMP       R0, #0
    0x807027a: 0xd1fa          BNE.N     ??main_0                ; 0x8070272
}
```




```
int main( )
{
main:
    0x8070252: 0xb500          PUSH      {LR}
    0x8070254: 0xb087          SUB       SP, SP, #0x1c
```

执行 PUSH {LR} 之前，SP = 0x20000400， 执行 PUSH {LR} 之后，LR 入栈，SP = 0x200003FC

接着执行 SUB SP, SP, #0x1c，为局部变量分配空间 28 bytes，SP = 0x200003E0

```
    0x8070272: 0xa800 ADD R0, SP, #0x0
    0x8070274: 0xf7ff 0xffc3 BL func ; 0x80701fe
```

调用 func()

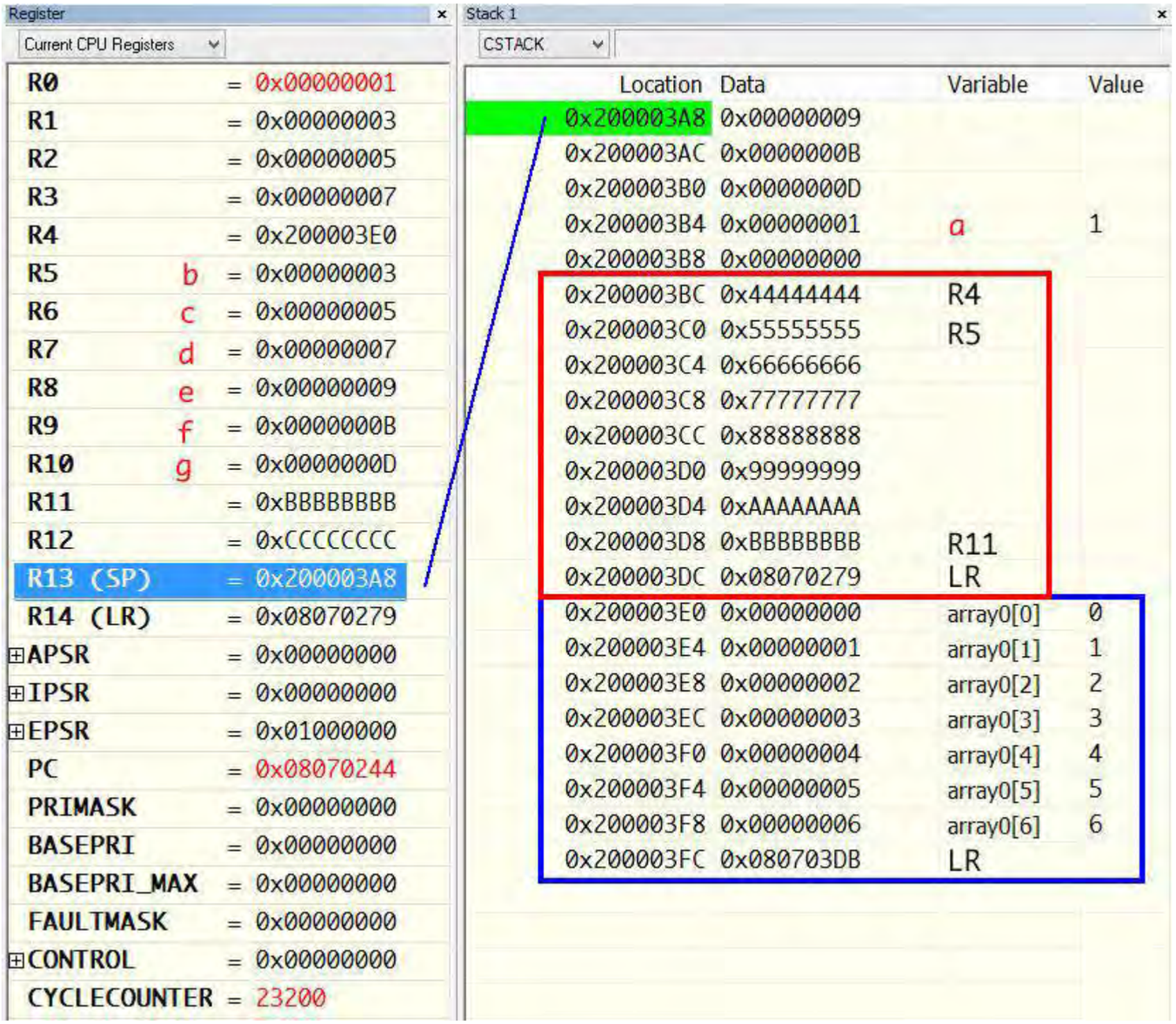
```
int func(int * buffer)
{
```

```
func:
    0x80701fe: 0xe92d 0x4ff0   PUSH.W     {R4-R11, LR}
    0x8070202: 0xb085          SUB         SP, SP, #0x14
    0x8070204: 0x0004          MOVS        R4, R0
    ... ... ... ...
  h = foo(a, b, c, d, e, f, g);
```

PUSH .W {R4-R11, LR}, SUB SP, SP, #0x14, 保存寄存器, 为局部变量分配空间 a, 其他变量 b,c,d,e,f,g 使用寄存器 R5-R10

h 使用寄存器 R11〈图中没有显示〉

同时分配 3 个变量的空间, 用于通过堆栈传递参数, 调用 foo(a, b, c, d, e, f, g)




```
h = foo(a, b, c, d, e, f, g);
  0x8070230: 0xf8cd 0xa008  STR.W     R10, [SP, #0x8]     g
  0x8070234: 0xf8cd 0x9004  STR.W     R9, [SP, #0x4]      f
  0x8070238: 0xf8cd 0x8000  STR.W     R8, [SP]            e
  0x807023c: 0x003b         MOVS      R3, R7              d
```

```
  0x807023e: 0x0032          MOVS      R2, R6              c
  0x8070240: 0x0029          MOVS      R1, R5              b
  0x8070242: 0x9803          LDR       R0, [SP, #0xc]      a
  0x8070244: 0xf7ff 0xffad   BL        foo                     ; 0x80701a2
  0x8070248: 0x4683          MOV       R11, R0
return h;
  0x807024a: 0x4658          MOV       R0, R11
  0x807024c: 0xb005          ADD       SP, SP, #0x14
  0x807024e: 0xe8bd 0x8ff0   POP.W     {R4-R11, PC}
```

```
foo(a, b, c, d,  e, f, g) : a -> R0, b -> R1, c  -> R2, d -> R3, e -> [SP, #0x00], f  -> [SP, #0x04],  e -> [SP, #0x08]
```

Register

Current CPU Registers

R0	a	=	0x00000001
R1	b	=	0x00000003
R2	c	=	0x00000005
R3	d	=	0x00000007
R4		=	0x200003E0
R5	b	=	0x00000003
R6	c	=	0x00000005
R7	d	=	0x00000007
R8	e	=	0x00000009
R9	f	=	0x0000000B
R10	g	=	0x0000000D
R11		=	0xBBBBBBBB
R12		=	0xCCCCCCCC
R13 (SP)		=	0x200003A8
R14 (LR)		=	0x08070279
⊞APSR		=	0x00000000
⊞IPSR		=	0x00000000
⊞EPSR		=	0x01000000
PC		=	0x08070244
PRIMASK		=	0x00000000
BASEPRI		=	0x00000000
BASEPRI_MAX		=	0x00000000
FAULTMASK		=	0x00000000
⊞CONTROL		=	0x00000000
CYCLECOUNTER		=	23200

Stack 1

CSTACK

Location	Data	Variable	Value
0x200003A8	0x00000009	e	
0x200003AC	0x0000000B	f	
0x200003B0	0x0000000D	g	
0x200003B4	0x00000001	a	1
0x200003B8	0x00000000		
0x200003BC	0x44444444	R4	
0x200003C0	0x55555555	R5	
0x200003C4	0x66666666		
0x200003C8	0x77777777		
0x200003CC	0x88888888		
0x200003D0	0x99999999		
0x200003D4	0xAAAAAAAA		
0x200003D8	0xBBBBBBBB	R11	
0x200003DC	0x08070279	LR	
0x200003E0	0x00000000	array0[0]	0
0x200003E4	0x00000001	array0[1]	1
0x200003E8	0x00000002	array0[2]	2
0x200003EC	0x00000003	array0[3]	3
0x200003F0	0x00000004	array0[4]	4
0x200003F4	0x00000005	array0[5]	5
0x200003F8	0x00000006	array0[6]	6
0x200003FC	0x080703DB	LR	

```
foo:
    0x80701a2: 0xe92d 0x47f0  PUSH.W    {R4-R10, LR}
    0x80701a6: 0xb08a         SUB       SP, SP, #0x28
```

```
    0x80701a8: 0x0004          MOVS      R4, R0                a 通过寄存器传递
    0x80701aa: 0x000d          MOVS      R5, R1                b 通过寄存器传递
    0x80701ac: 0x0016          MOVS      R6, R2                c 通过寄存器传递
    0x80701ae: 0x001f          MOVS      R7, R3                d 通过寄存器传递
    0x80701b0: 0xf8dd 0x8048   LDR.W     R8, [SP, #0x48]       e 通过堆栈传递
    0x80701b4: 0xf8dd 0x904c   LDR.W     R9, [SP, #0x4c]       f 通过堆栈传递
    0x80701b8: 0xf8dd 0xa050   LDR.W     R10, [SP, #0x50]      g 通过堆栈传递
```

PUSH .W {R4-R10, LR}, SUB SP, SP, #0x28, 保存寄存器, 为局部变量分配空间 array1[7], 同时分配 3 个变量的空间, 用于通过堆栈传递参数, 调用

bar(array1[0], array1[1], array1[2], array1[3], array1[4], array1[5], array1[6]);

foo (a, b, c, d, e, f, g) : a, b, c, d 通过寄存器传递, e, f, g 通过堆栈传递

获取参数, 运算之后, 存入局部变量 array1[7], 然后准备调用 bar()

array1[0], array1[1], array1[2], array1[3]   通过寄存器传递

array1[4], array1[5], array1[6] 通过堆栈传递

```
return bar( array1[0], array1[1], array1[2], array1[3], array1[4], array1[5], array1[6] );
    0x80701e0: 0x9809          LDR       R0, [SP, #0x24]          array1[6]
    0x80701e2: 0x9002          STR       R0, [SP, #0x8]

    0x80701e4: 0x9808          LDR       R0, [SP, #0x20]            array1[5]
    0x80701e6: 0x9001          STR       R0, [SP, #0x4]

    0x80701e8: 0x9807          LDR       R0, [SP, #0x1c]          array1[4]
    0x80701ea: 0x9000          STR       R0, [SP]

    0x80701ec: 0x9b06          LDR       R3, [SP, #0x18]          array1[3]
    0x80701ee: 0x9a05          LDR       R2, [SP, #0x14]          array1[2]
    0x80701f0: 0x9904          LDR       R1, [SP, #0x10]          array1[1]
    0x80701f2: 0x9803          LDR       R0, [SP, #0xc]           array1[0]

    0x80701f4: 0xf7ff 0xffa5   BL        bar                     ; 0x8070142
    0x80701f8: 0xb00a          ADD       SP, SP, #0x28
    0x80701fa: 0xe8bd 0x87f0   POP.W     {R4-R10, PC}
```

Register

Current CPU Registers

R0 array1[0]	=	0x00000004
R1 array1[1]	=	0x00000008
R2 array1[2]	=	0x0000000C
R3 array1[3]	=	0x00000010
R4	=	0x00000001
R5	=	0x00000003
R6	=	0x00000005
R7	=	0x00000007
R8	=	0x00000009
R9	=	0x0000000B
R10	=	0x0000000D
R11	=	0xBBBBBBBB
R12	=	0xCCCCCCCC
R13 (SP)	=	0x20000360
R14 (LR)	=	0x08070249
⊞APSR	=	0x00000000
⊞IPSR	=	0x00000000
⊞EPSR	=	0x01000000
PC	=	0x080701F4
PRIMASK	=	0x00000000
BASEPRI	=	0x00000000
BASEPRI_MAX	=	0x00000000
FAULTMASK	=	0x00000000
⊞CONTROL	=	0x00000000
CYCLECOUNTER	=	23254

Stack 1

CSTACK

Location	Data	Variable	Value
0x20000360	0x00000014	array1[4]	
0x20000364	0x00000018	array1[5]	
0x20000368	0x0000000E	array1[6]	
0x2000036C	0x00000004	array1[0]	4
0x20000370	0x00000008	array1[1]	8
0x20000374	0x0000000C	array1[2]	12
0x20000378	0x00000010	array1[3]	16
0x2000037C	0x00000014	array1[4]	20
0x20000380	0x00000018	array1[5]	24
0x20000384	0x0000000E	array1[6]	14
0x20000388	0x200003E0	R4	
0x2000038C	0x00000003	R5	
0x20000390	0x00000005		
0x20000394	0x00000007		
0x20000398	0x00000009		
0x2000039C	0x0000000B		
0x200003A0	0x0000000D	R10	
0x200003A4	0x08070249	LR	
0x200003A8	0x00000009	e	
0x200003AC	0x0000000B	f	
0x200003B0	0x0000000D	g	
0x200003B4	0x00000001	a	1
0x200003B8	0x00000000		
0x200003BC	0x44444444		
0x200003C0	0x55555555		
0x200003C4	0x66666666		
0x200003C8	0x77777777		
0x200003CC	0x88888888		
0x200003D0	0x99999999		
0x200003D4	0xAAAAAAAA		
0x200003D8	0xBBBBBBBB		
0x200003DC	0x08070279		
0x200003E0	0x00000000	array0[0]	0
0x200003E4	0x00000001	array0[1]	1
0x200003E8	0x00000002	array0[2]	2
0x200003EC	0x00000003	array0[3]	3
0x200003F0	0x00000004	array0[4]	4
0x200003F4	0x00000005	array0[5]	5
0x200003F8	0x00000006	array0[6]	6
0x200003FC	0x080703DB		

```
bar:
    0x8070142: 0xb4f0          PUSH      {R4-R7}
    0x8070144: 0xb087          SUB       SP, SP, #0x1c
    0x8070146: 0x0004          MOVS      R4, R0                  a <- array1[0]
    0x8070148: 0x9d0b          LDR       R5, [SP, #0x2c]         e <- array1[4] -- 通过堆栈传递的参数
    0x807014a: 0x9e0c          LDR       R6, [SP, #0x30]         f <- array1[5] -- 通过堆栈传递的参数
    0x807014c: 0x9f0d          LDR       R7, [SP, #0x34]         g <- array1[6] -- 通过堆栈传递的参数
```

PUSH.W {R4-R7, LR}, SUB SP, SP, #0x1C, 保存寄存器, 为局部变量分配空间 array2[7]

Register

Current CPU Registers

R0	=	0x00000004
R1	=	0x00000008
R2	=	0x0000000C
R3	=	0x00000010
R4	=	0x00000004
R5	=	0x00000014
R6	=	0x00000018
R7	=	0x0000000E
R8	=	0x00000009
R9	=	0x0000000B
R10	=	0x0000000D
R11	=	0xBBBBBBBB
R12	=	0xCCCCCCCC
R13 (SP)	=	0x20000334

Stack 1

CSTACK

	Location	Data	Variable	Value
local vars	0x20000334	0x00000000	array2[0]	0
	0x20000338	0x00000000	array2[1]	0
	0x2000033C	0x00000000	array2[2]	0
	0x20000340	0x00000000	array2[3]	0
	0x20000344	0x00000000	array2[4]	0
	0x20000348	0x00000000	array2[5]	0
	0x2000034C	0x00000000	array2[6]	0
bar() regs	0x20000350	0x00000001	R4	
	0x20000354	0x00000003	R5	
	0x20000358	0x00000005	R6	
	0x2000035C	0x00000007	R7	

Register

Current CPU Registers

R0	array1[0]	= 0x00000004	args
R1	array1[1]	= 0x00000008	
R2	array1[2]	= 0x0000000C	
R3	array1[3]	= 0x00000010	
R4		= 0x00000001	
R5		= 0x00000003	
R6		= 0x00000005	
R7		= 0x00000007	
R8		= 0x00000009	
R9		= 0x0000000B	
R10		= 0x0000000D	
R11		= 0xBBBBBBBB	
R12		= 0xCCCCCCCC	
R13 (SP)		= 0x20000360	
R14 (LR)		= 0x08070249	
APSR		= 0x00000000	
IPSR		= 0x00000000	
EPSR		= 0x01000000	
PC		= 0x080701F4	
PRIMASK		= 0x00000000	
BASEPRI		= 0x00000000	
BASEPRI_MAX		= 0x00000000	
FAULTMASK		= 0x00000000	
CONTROL		= 0x00000000	
CYCLECOUNTER		= 23254	

Stack 1

CSTACK

	Location	Data	Variable	Value
args	0x20000360	0x00000014	array1[4]	
	0x20000364	0x00000018	array1[5]	
	0x20000368	0x0000000E	array1[6]	
local vars	0x2000036C	0x00000004	array1[0]	4
	0x20000370	0x00000008	array1[1]	8
	0x20000374	0x0000000C	array1[2]	12
	0x20000378	0x00000010	array1[3]	16
	0x2000037C	0x00000014	array1[4]	20
	0x20000380	0x00000018	array1[5]	24
	0x20000384	0x0000000E	array1[6]	14
foo() saved regs	0x20000388	0x200003E0	R4	
	0x2000038C	0x00000003	R5	
	0x20000390	0x00000005		
	0x20000394	0x00000007		
	0x20000398	0x00000009		
	0x2000039C	0x0000000B		
	0x200003A0	0x0000000D	R10	
	0x200003A4	0x08070249	LR	
args	0x200003A8	0x00000009	e	
	0x200003AC	0x0000000B	f	
	0x200003B0	0x0000000D	g	
vars	0x200003B4	0x00000001	a	1
	0x200003B8	0x00000000		
regs	0x200003BC	0x44444444		
	0x200003C0	0x55555555		
	0x200003C4	0x66666666		
	0x200003C8	0x77777777		
	0x200003CC	0x88888888		
	0x200003D0	0x99999999		
	0x200003D4	0xAAAAAAAA		
	0x200003D8	0xBBBBBBBB		
func() saved	0x200003DC	0x08070279		
vars	0x200003E0	0x00000000	array0[0]	0
	0x200003E4	0x00000001	array0[1]	1
	0x200003E8	0x00000002	array0[2]	2
	0x200003EC	0x00000003	array0[3]	3
	0x200003F0	0x00000004	array0[4]	4
	0x200003F4	0x00000005	array0[5]	5
	0x200003F8	0x00000006	array0[6]	6
main() regs	0x200003FC	0x080703DB		

http://www.cnblogs.com/shangdawei/p/3324101.html

常用功能

IDA 的导航条

IDA 是一个功能非常强大的反汇编工具，工具提供的工具也很多。只有在多多使用之后才会发现它的精细之处...

导航条：

也许你之前一直忽视它的存在，有一天你会发现称它的“导航”并非浪得虚名~

-蓝色：.text section

深蓝：用户自己写的函数编译后的代码区

浅蓝：编译器自己添加的函数，像启动函数，异常函数等（我自己猜的，不一定百分百正确）

-粉红色：.idata section

有关输入表的一些数据信息

-军绿色：.rdata section

纯数据，只读

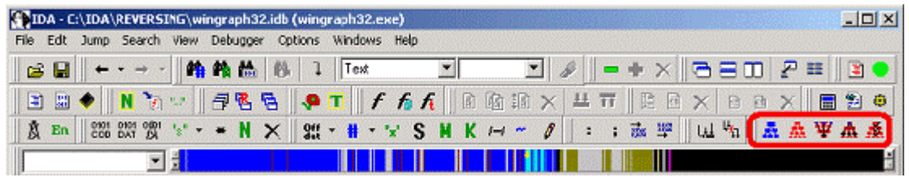
-灰色：为了段对齐而留下的空隙

-黑色：禁区，不存在任何数据

http://blog.csdn.net/chence19871/article/details/7716527

IDA PRO 的流程图功能

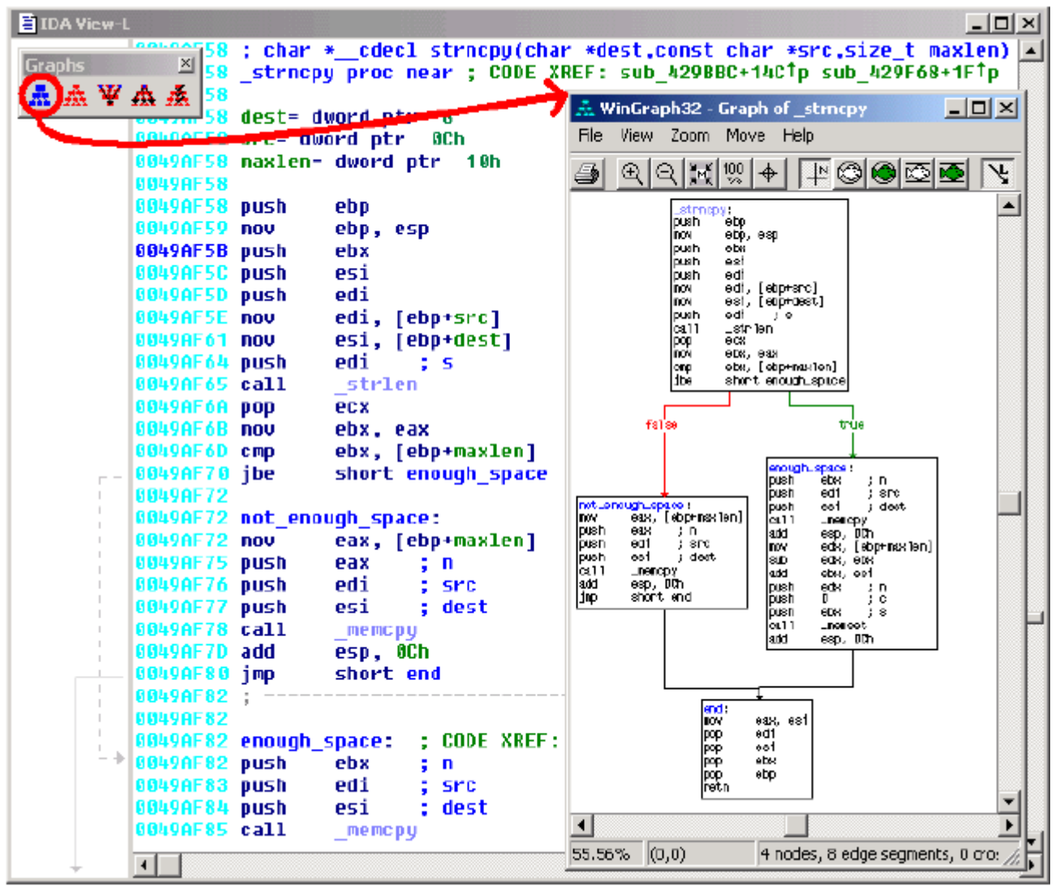
从 4.17 版开始 IDA PRO 就提供了流程图功能，它使用了 VCG 图形库里面的 WinGrah32 工具。IDA PRO 可以提供标准的 GDL 图形给 WinGrah32 来绘制流程图。流程图工具条如下所示。



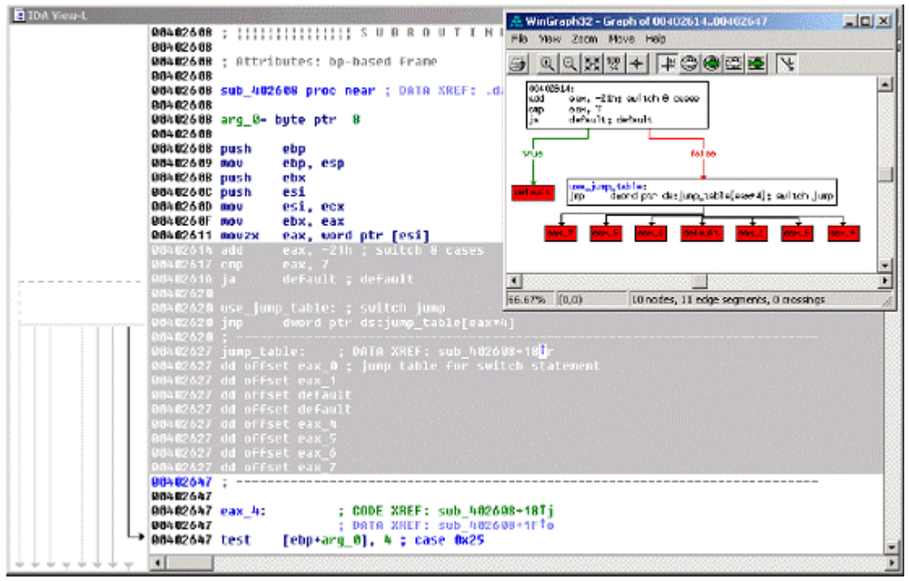
下面我们来看一些流程图工具的一些具体使用方法。

流程图

通过流程图来分析一个功能函数，可以清晰的看出程序的流程结构，使得分析更清楚，更容易。使用 *FlowChart* 按钮，可以绘制流程图。

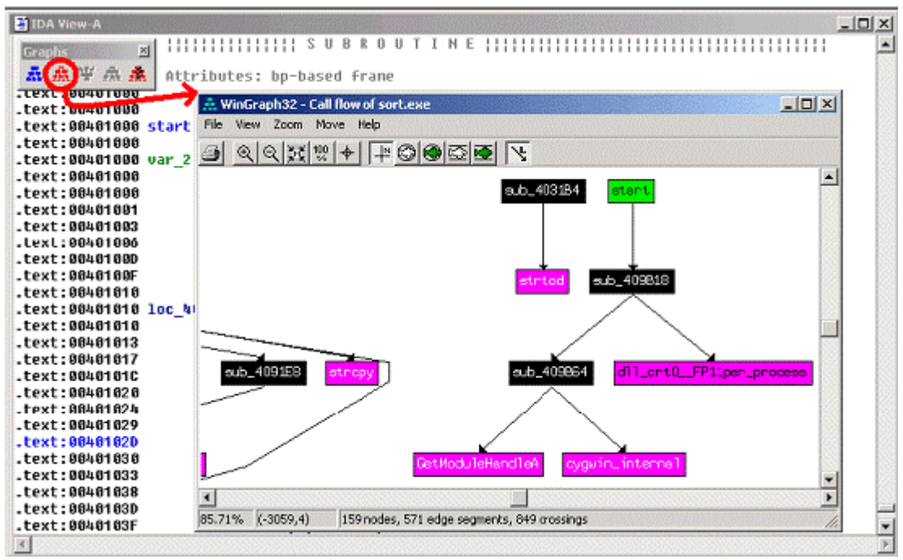


如果你希望能在一张流程图中同时显示几个函数，或只显示一个大的函数中的一部分流程，我们可以选择相应代码区域来进行流程图显示。



函数调用

很多时候，我们需要分析函数之间的依存关系，IDA 提供了一项分析函数间依存关系（交叉参考）的功能。如图：

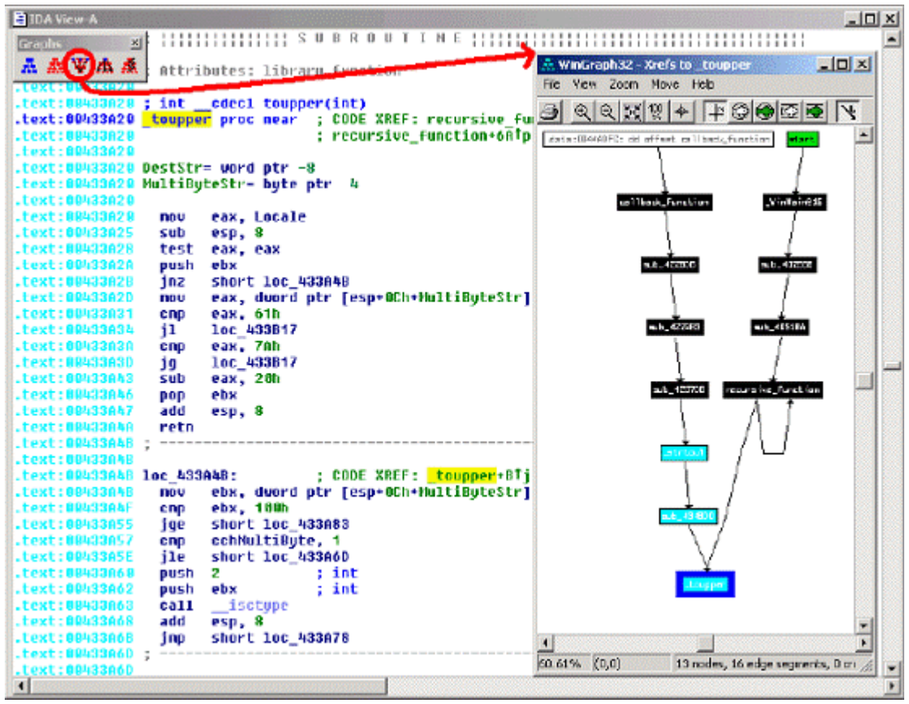


这条命令对一些小程序比较有效，这是因为当程序比较庞大，包含大量函数时，其依存关系变得异常复杂。可以看到上面图中，根据函数地址的属性，被区分为不同的颜色（如程序入口点，外部函数，库函数……），颜色规则与反汇编窗口中的分析结果有关。

函数交叉参考

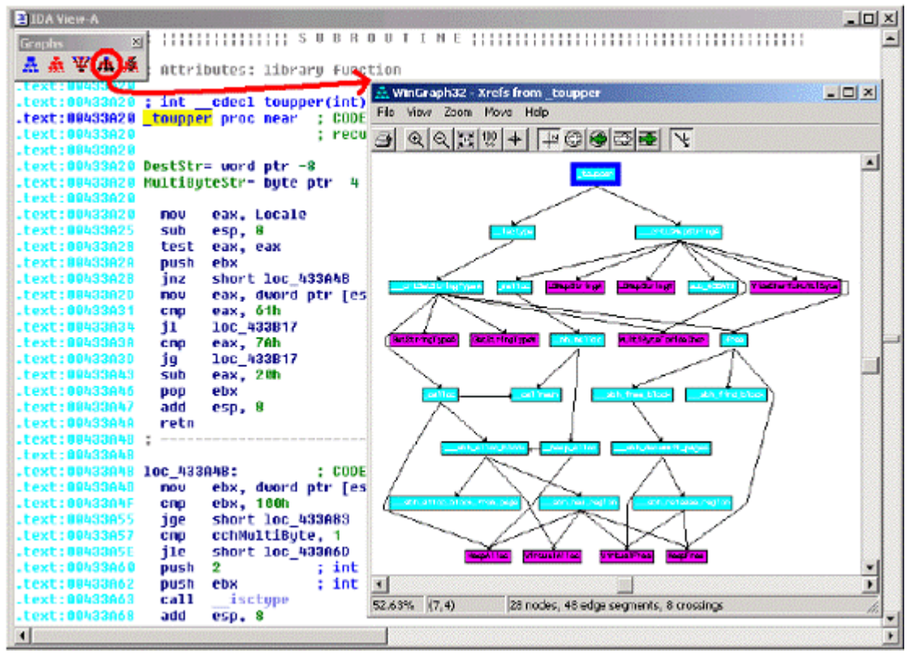
Xrefs to 和 *Xrefs from* 可以看出一个指定函数的引用关系和被引用关系。

Xrefs to 命令对指定函数的被其它代码或数据的引用情况给出了图形参考，让我们看一下 C 标准库中的 *toupper()*函数，它的功能是将小写字母转换为大写字母。



在图形的顶部，我们注意到 *start()* 函数，它调用了 *WinMain()*函数，在左侧，我们注意到 *callback_function()*的函数指针，在右侧底部我们注意到 *recursive_function()*，调用自己。我们也看到 *strtoul()*函数（C 标准库里的一个函数，它转换一个字符串为一个无符号长整形），需要调用 *toupper()*函数。最后，注意被选择的函数，总是使用蓝色粗线框。

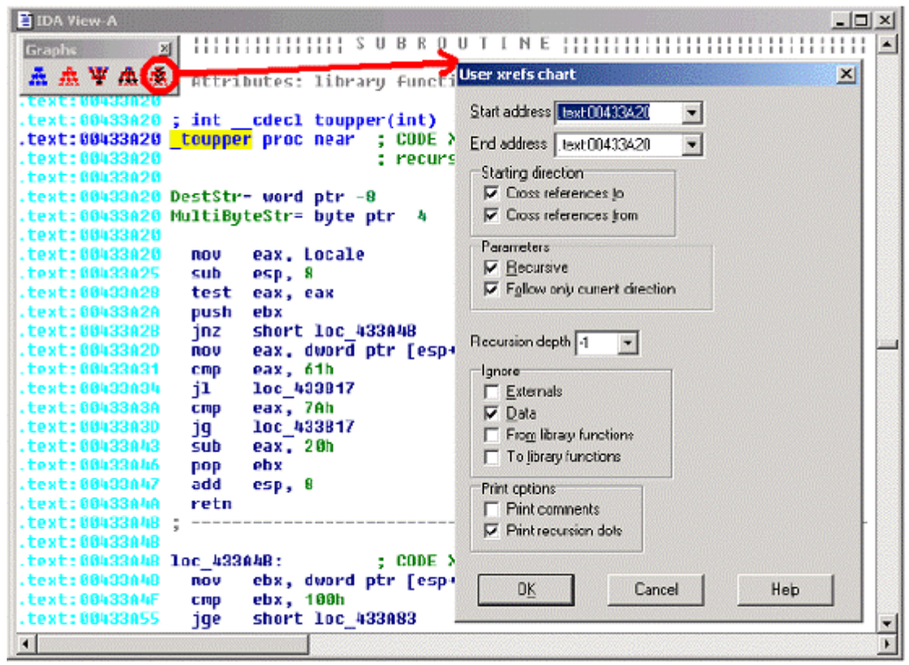
Xrefs from 命令给出了指定函数调用其它函数的图形参考。让我们再看一下 *toupper()*函数的引用关系图。



通过这个引用关系图，我们注意到，*toupper()*函数调用了大量其它 C 标准库函数，它们直接或间接的引用了一些 *WINDOWSAPI* 函数，如 *VitrualAlloc(),WideCharToMultiByte()*。和函数被引用图一样，它也可以指定选择区域，来显示多个函数的引用关系图，或一个函数的部分引用关系图。

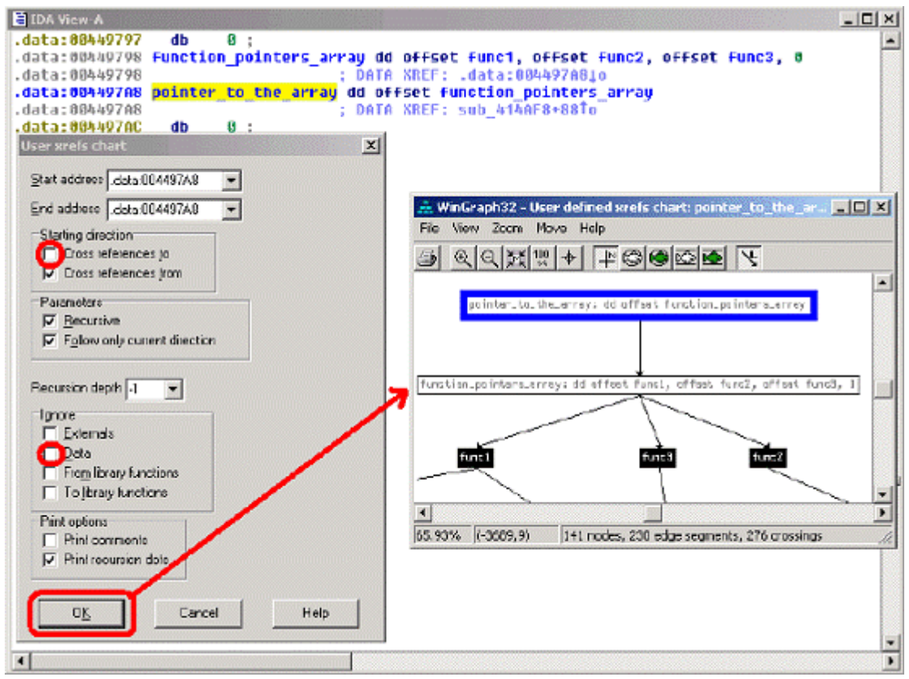
用户自定义的交叉参考图

IDA PRO 还为一些专业使用者提供一些高级图形功能，让我们实践一下如何使用这些功能。

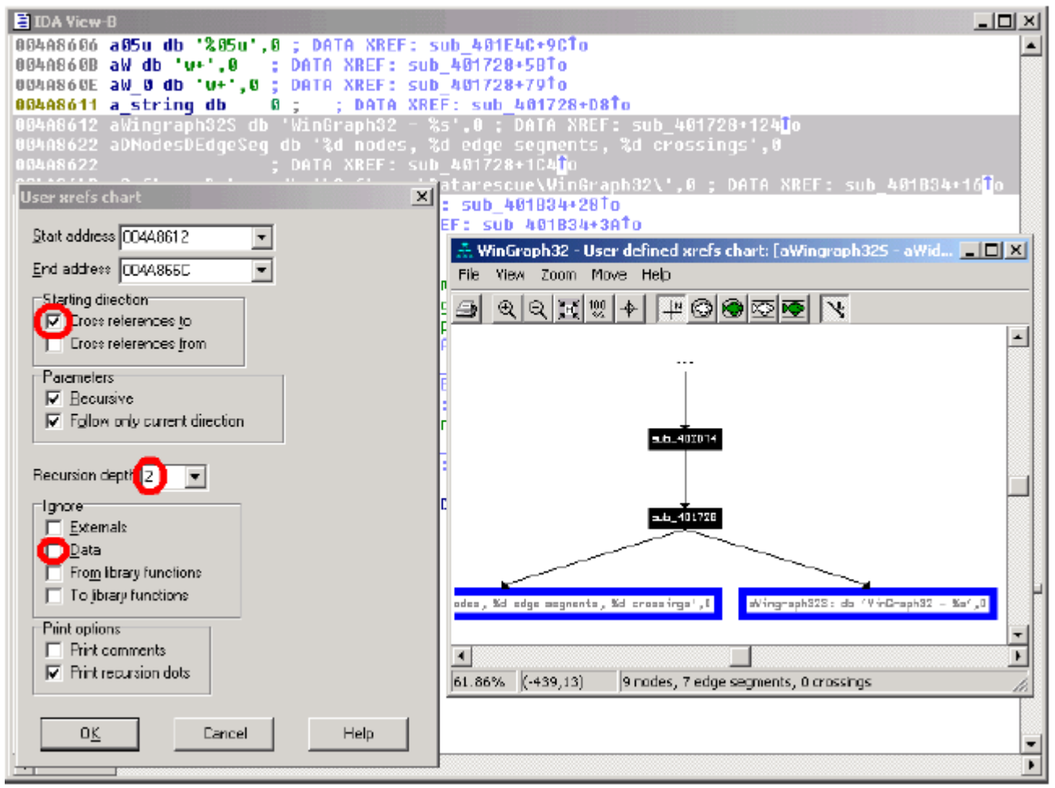


下面我们具体看一下这些选项的用法。

函数被 *Xrefsfrom* 命令只绘出**代码的交叉参考，**但有时，数据的交叉参考也非常有用。让我们看一个一组函数指针的指针，我们取消 *Crossreferences to* 和 *IgnoreData* 的选项。



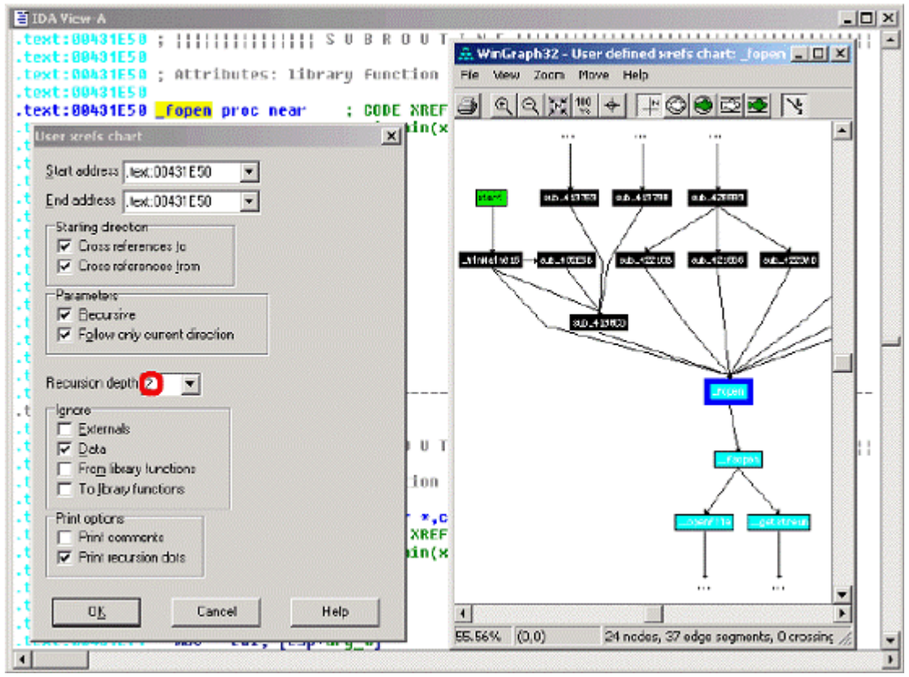
IDA 也可以显示一个全局数据的参考关系图，我们可以显示一个全局数据的或一组数据的函数调用交叉参考图。最好指定最大的递归深度，以避免图形无法显示。



由于我们保留了 *Printrecursion dots*,一些省略点显示在顶部，表示至少有一个交叉参考在指定的递归深度之外。

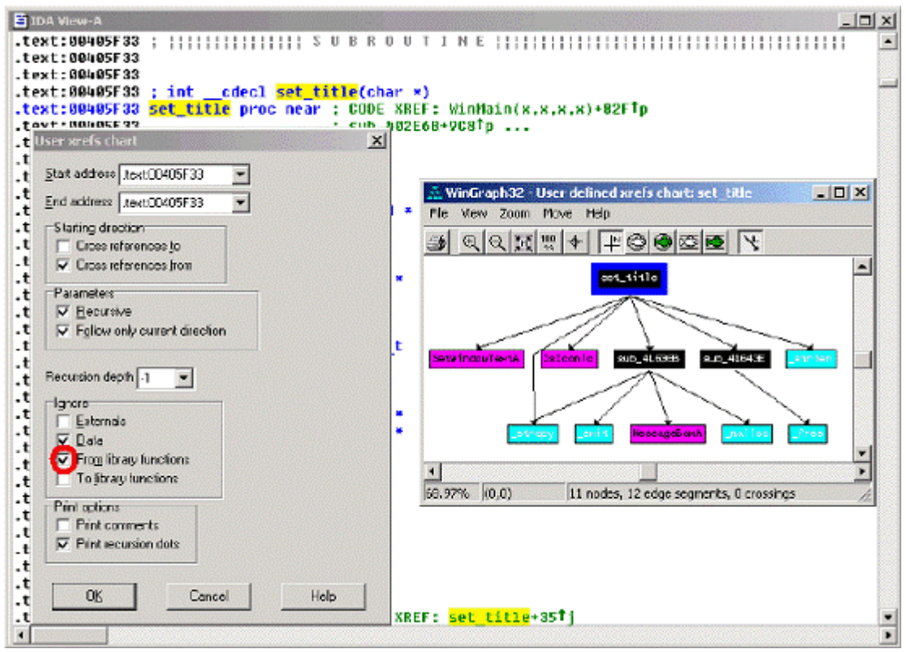
递推深度

正如我们前面提到的，函数调用 *FunctionsCall* 功能无法显示一个包含大量函数的正常程序的关系图。但如果我们设定递推深度 *RecursionDepth*，我们就能得到同样类型的图形。我们来看一下 *fopen()*函数。

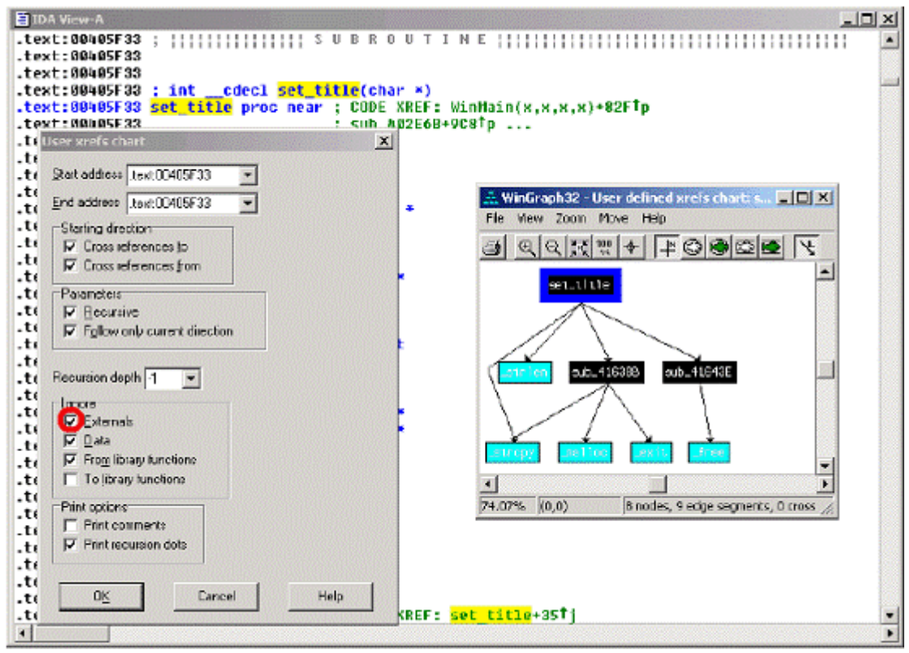


忽略特定的函数

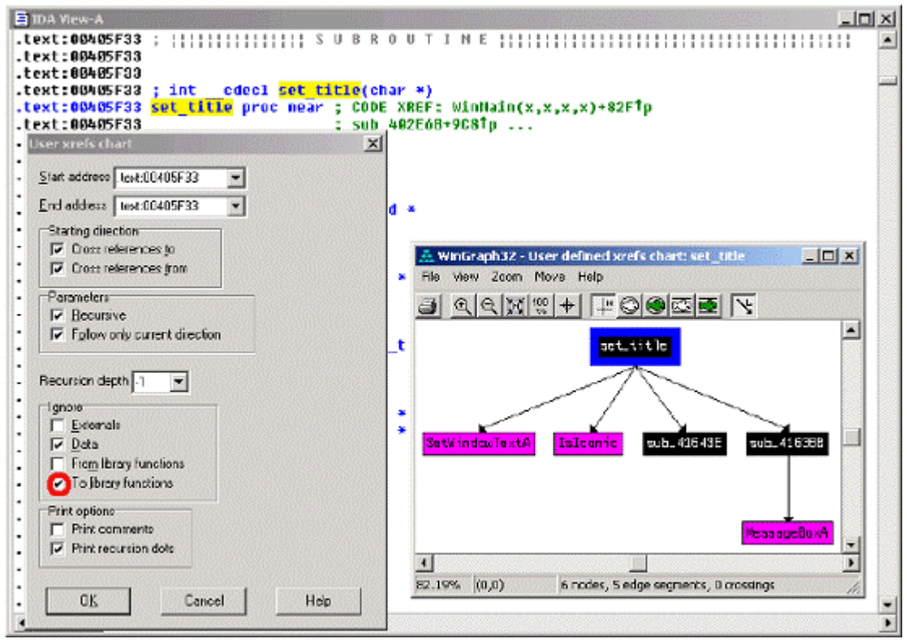
IDA 的 FLIRT(库快速识别技术)可以从许多库中识别出标准函数。由于这些函数的资料很多，我们可以不用关心这些函数的内部实现。我们可以通过使用 *Ignore From LibraryFunctions* 的选项，来忽略这些库函数内部的交叉参考。



然而，一些外部 Windows API 函数如 *MessageBox()*总是显示在交叉参考的图形中。如果我们也想屏蔽这些函数的显示，只关注对库函数的调用，我们可以选择 *IgnoreExternals* 这个选项。

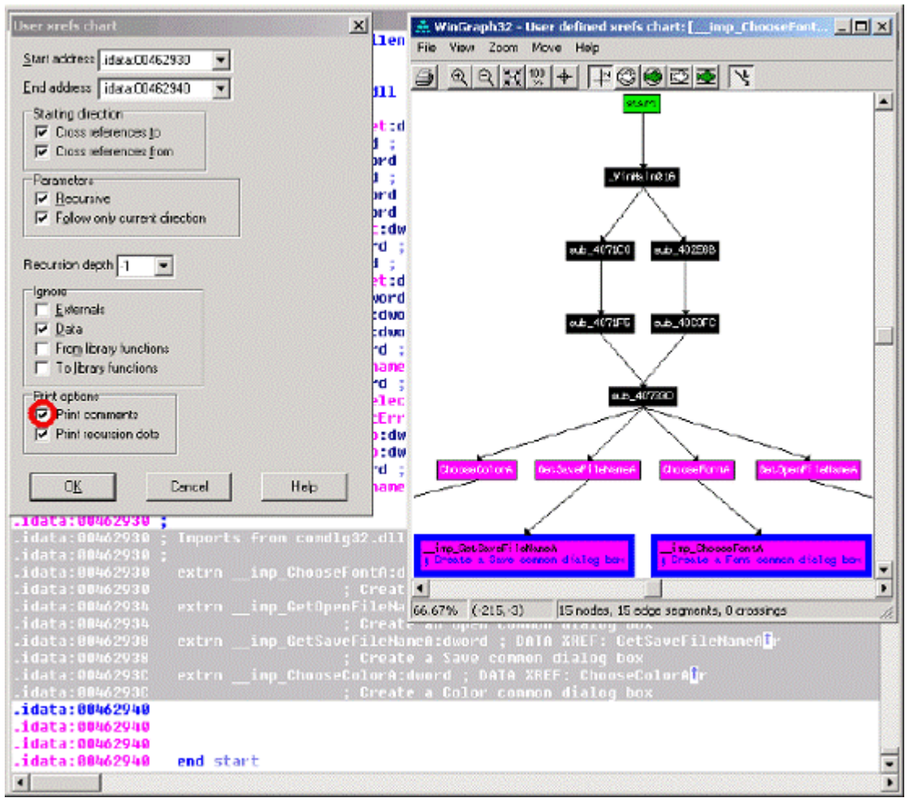


反之，我们也可以只显示对外部函数的引用，而屏蔽对库函数的调用。这可以看出一个函数对 DLL 的依赖程度。



显示注释

最后，在结束这个小教程之前，让我们来看一下，如何绘制一个 Windows 应用程序对一个特定 DLL 文件的依赖关系图。这可以通过创建一个基于特定 DLL 相关的所有外部声明的参考图来实现。我们可以选中 *PrintComment* 选项，它将会对这些外部函数打印注释。下面的例子给出了一个程序与 *comdlg32.dll* 的依存关系。



IDA 和其集成的 WinGraph32 可以帮助我们快速分析复杂的程序，它将有用信息从复杂的程序中筛检出来，并使用流程图的方式清晰的展现给我们。

http://blog.csdn.net/eqera/article/details/8237949

使用 IDA 分析高级数据结构

使用 IDA 反汇编时，一些数据和操作数的处理方式可能并不准确。IDA 允许我们手动去修改这些数据的类型和定义，我们甚至可以使用类似高级语言的数据结构。

一个 C 语言程序

为了介绍 IDA 的数据分析处理功能，我们先看一个 C 语言编写的使用特殊数据类型和结构

的小程序：

```
#include <stdio.h>
#include <alloc.h>
// our structures
// ==============
// information about our customers
struct customer_t { // a typical structure
long id;
char name[32];
char sex; // 'm'ale - 'f'emale
};
// we sell books
struct book_t {
char title[128]; // an ASCII string
};
// and we sell computer softwares
struct software_info_t { // a structure containing various bitfields
unsigned int plateform : 2; // 2 bits reserved for the plateform -
// plateforms can be combined (0x03)
#define PC 0x1 // 0x01
```

```
#define MAC 0x2 // 0x02
unsigned int os : 3; // 3 bits reserved for the OS -
// OS can be combined (0x1C)
#define WINDOWS 0x1 // 0x04
#define DOS 0x2 // 0x08
#define OS_X 0x4 // 0x10
unsigned int category : 2; // 2 bits reserved for the category -
// categories can't be combined (0x60)
#define DISASSEMBLY 0x1 // 0x20
#define RECOVERY 0x2 // 0x40
#define CRYPTOGRAPHY 0x3 // 0x60
};
struct software_t {
software_info_t info;
char name[32];
};
// generic products we're selling
enum product_category_t { // an enumerated type
BOOK,
SOFTWARE,
HARDWARE // we actually don't sell hardware
};
```

```
union product_u { // an union to contain product information
// depending on its category
book_t book;
software_t software;
// struct hardware_t hardware; // we actually don't sell hardware
};
struct product_t { // a structure containing another structure
long id;
product_category_t category;
product_u p;
};
// our data
// ========
// our customers
customer_t customers[] = { // an initialized array to memorize our customers
{ 1, "Peter", 'm' },
{ 2, "John", 'm' },
{ 3, "Mary", 'f' },
{ 0 }
};
// our products
book_t ida_book = { "IDA QuickStart Guide" };
```

```
softwares_t softwares = // an initialized variable length structure
{
3,
{
{ { PC, WINDOWS|DOS, DISASSEMBLY }, "IDA Pro" },
{ { PC|MAC, WINDOWS|OS_X, RECOVERY }, "PhotoRescue" },
{ { PC, WINDOWS, CRYPTOGRAPHY }, "aCrypt" }
}
};
#define PRODUCTS_COUNT 4
// our functions
// =============
// check software information
int check_software(software_info_t software_info)
{
bool valid = true;
if (software_info.plateform & PC)
{
if (! (software_info.plateform & MAC) && (software_info.os & OS_X))
valid = false; // OS-X isn't yet available on PC ;)
}
else if (software_info.plateform & MAC)
```

```
{
if (! (software_info.plateform & PC) && ((software_info.os & WINDOWS) ||
(software_info.os & DOS)))
valid = false; // Windows & DOS aren't available on Mac...
}
else
valid = false;
return valid;
}
// check product category
int check_product(product_category_t product_category)
{
bool valid = true;
if (product_category == HARDWARE)
{
valid = false;
printf("We don't sell hardware for the moment...\n");
}
return valid;
}
// print customer information
void print_customer(customer_t *customer)
```

```
{
printf("CUSTOMER %04X: %s (%c)\n", customer->id,
customer->name,
customer->sex);
}
// print book information
void print_book(book_t *book)
{
printf("BOOK: %s\n", book->title);
}
// print software information
void print_software(software_t *software)
{
printf("SOFTWARE: %s:", software->name);
// plateform
// we use 'if', as plateforms can be combined
if (software->info.plateform & PC)
printf(" PC");
if (software->info.plateform & MAC)
printf(" MAC");
printf(";");
// OS
```

```
// we use 'if', as os can be combined
if (software->info.os & WINDOWS)
printf(" WINDOWS");
if (software->info.os & DOS)
printf(" DOS");
if (software->info.os & OS_X)
printf(" OS-X");
printf(";");
// category
// we use 'switch', as categories can't be combined
switch(software->info.category)
{
case DISASSEMBLY:
printf(" DISASSEMBLY");
break;
case RECOVERY:
printf(" RECOVERY");
break;
case CRYPTOGRAPHY:
printf(" CRYPTOGRAPHY");
break;
}
```

```
printf("\n");
}
// print product information
bool print_product(product_t *product)
{
if (! check_product(product->category))
return false;
printf("PRODUCT %04X: ", product->id);
switch(product->category) {
case BOOK:
print_book(&product->p.book);
break;
case SOFTWARE:
print_software(&product->p.software);
break;
}
return true;
}
// our main program
// ================
void main()
{
```

```
// print customers listing

printf("CUSTOMERS:\n");

customer_t *customer = customers;

while (customer->id != 0)

{

print_customer(customer);

customer++;

}

// allocate a small array to store our products in memory

product_t *products = (product_t*) malloc(PRODUCTS_COUNT * sizeof(product_t));

// insert our products

products[0].id = 1;

products[0].category = BOOK;

products[0].p.book = ida_book;

products[1].id = 2;

products[1].category = SOFTWARE;

products[1].p.software = softwares.softs[0]; // we insert softwares from our

// variable length structure

products[2].id = 3;

products[2].category = SOFTWARE;

products[2].p.software = softwares.softs[1];

products[3].id = 4;
```

```
products[3].category = SOFTWARE;

products[3].p.software = softwares.softs[2];

// verify and print each product

printf("\nPRODUCTS:\n");

for (int i = 0; i < PRODUCTS_COUNT; i++)

{

// check validity of the product category

if (! check_product(products[i].category))

{

printf("Invalid product !!!\n");

break;

}

// check validity of softwares

if (products[i].category == SOFTWARE)

{

if (! check_software(products[i].p.software.info))

{

printf("Invalid software !!!\n");

break;

}

}

// and print the product
```

```
    print_product(&products[i]);
    }
    free(products);
    }
```

运行这个可执行程序可得到如下结果：

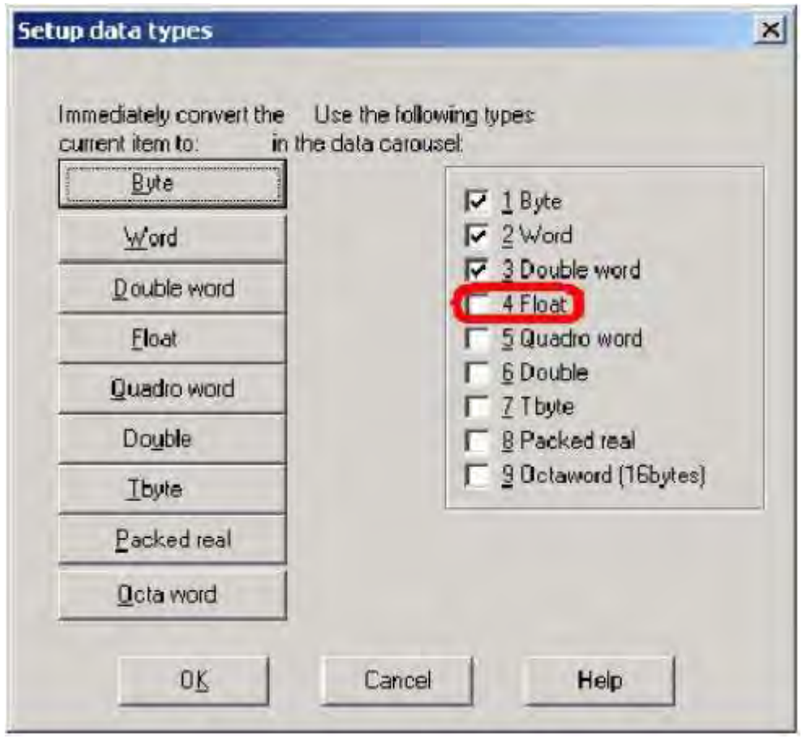
```
CUSTOMERS:
CUSTOMER 0001: Peter (m)
CUSTOMER 0002: John (m)
CUSTOMER 0003: Mary (f)
PRODUCTS:
PRODUCT 0001: BOOK: IDA QuickStart Guide
PRODUCT 0002: SOFTWARE: IDA Pro: PC; WINDOWS DOS; DISASSEMBLY
PRODUCT 0003: SOFTWARE: PhotoRescue: PC MAC; WINDOWS OS-X; RECOVERY
PRODUCT 0004: SOFTWARE: aCrypt: PC; WINDOWS; CRYPTOGRAPHY
```

下面让我们打开 IDA 装载这个程序来分析它。

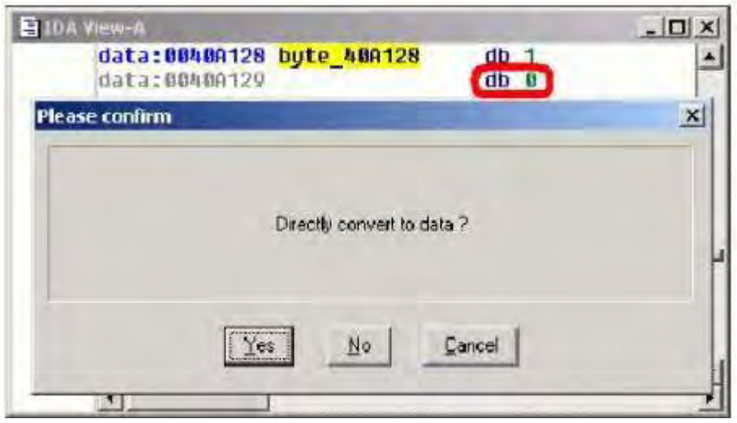
基础类型

我们可以很容易的将数据和一些数据基础类型联系起来，我们可以简单的使用快捷键“D”在字节 db、字 dw、双字 dd 间进行循环切换。

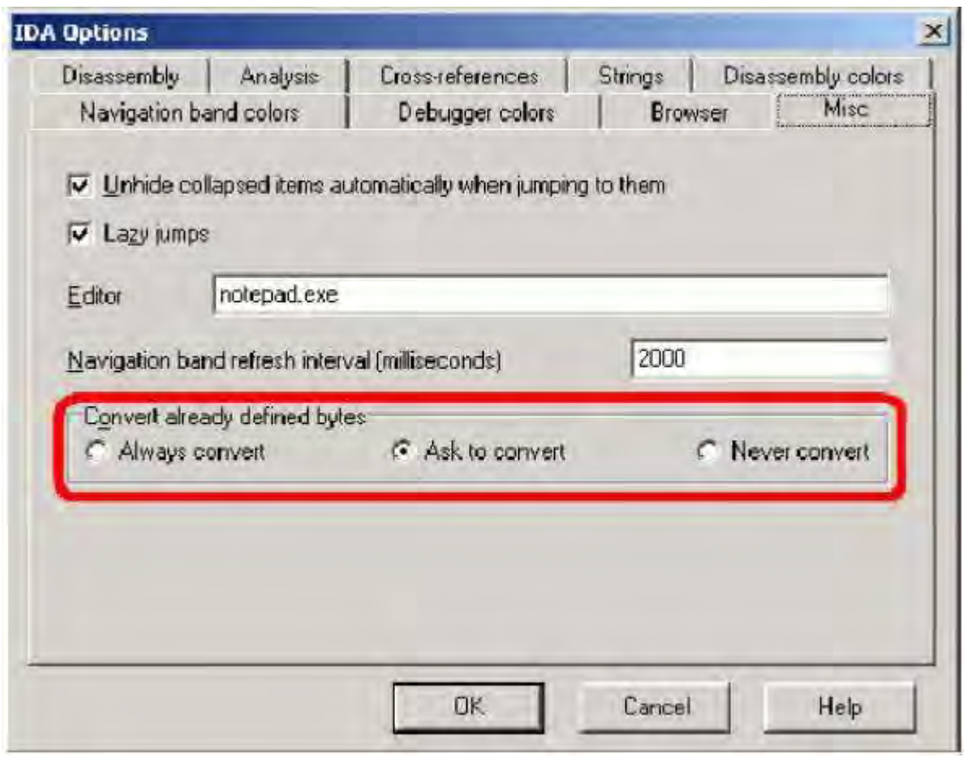
你可以定义循环切换包含的数据类型。我们试着在这个循环切换队列种加入浮点数 float：
使用“D”可以将一个双字类型 dd 的数据转换为浮点数 float。使用 options 菜单中 Setup data types 命令即可打开下面的窗口。



注意，数据类型改变时，相应数据的大小也随之变化。如下图，当我们在一个定义过的字节 db 上按“D”时，它将转换为一个字 dw，但是由于下一个字已经定义，因此 IDA 给出一个提示以让我们确认。



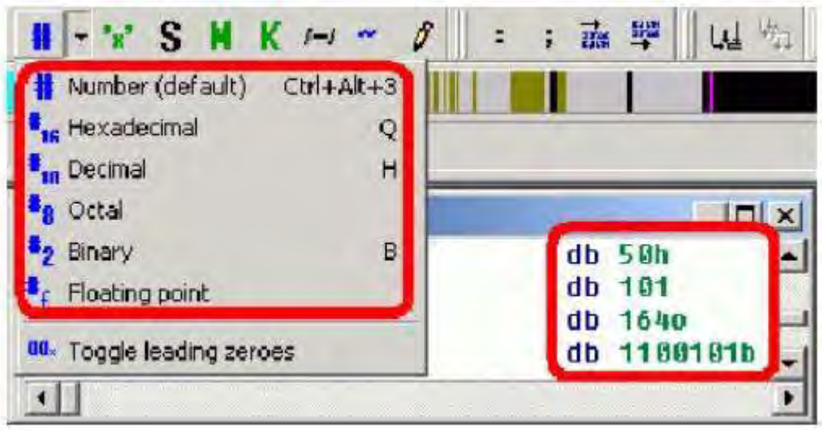
这是数据类型切换时的默认操作，该操作可以通过修改 options 对话框种的 Convert already defined bytes 来修改这个设置。



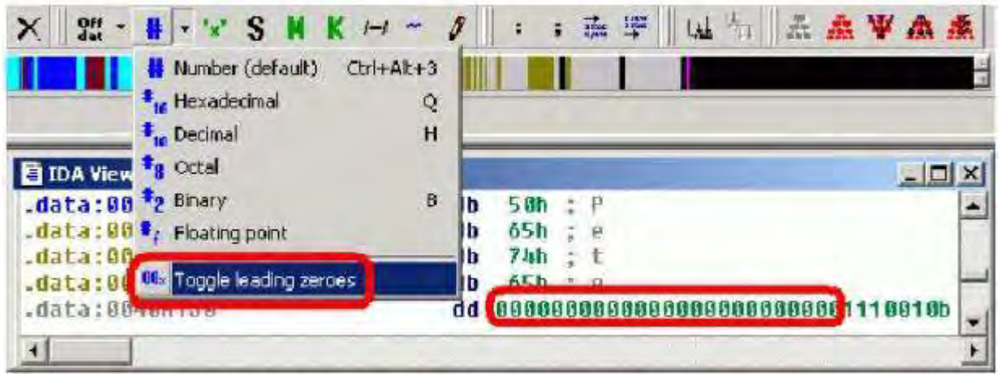
你也可以使用快捷键 U 来取消对数据的定义。

操作数格式

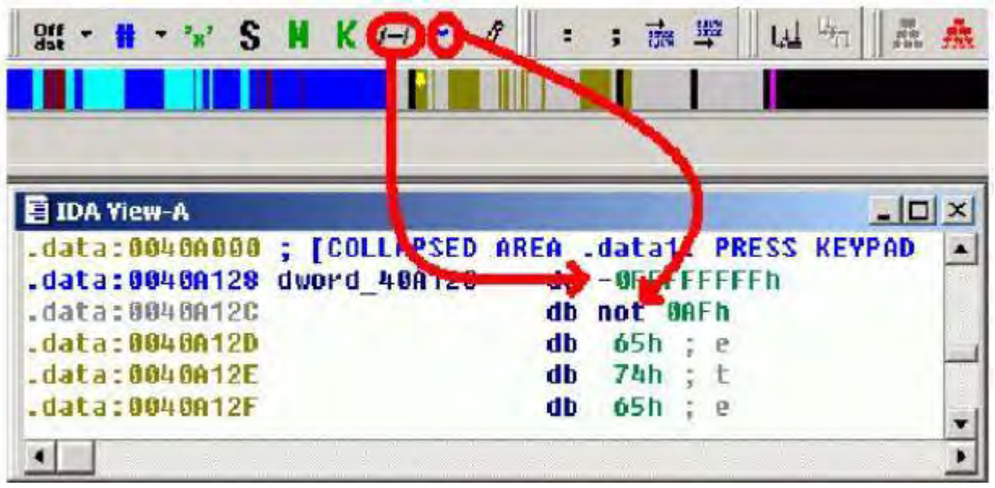
一旦数据类型被确定，我们就希望能用特定的格式来显示他们。IDA 提供了很多不同命令来支持格式转换。我们来看一些有趣的。请注意所有这些命令也可以用在指令的操作数上。通过在 Operands 工具条上的 Number 命令，我们可以随意切换数据格式。



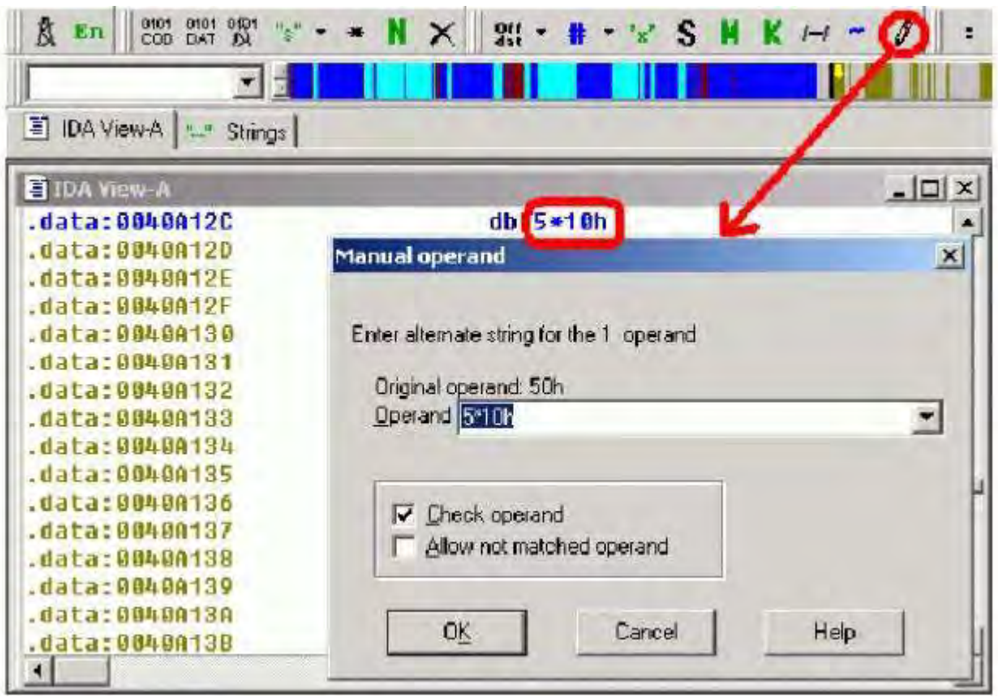
我们可以在数字数据的前面补零显示。如下图：



也可以改变操作数的符号，或对操作数进行逐位取反。

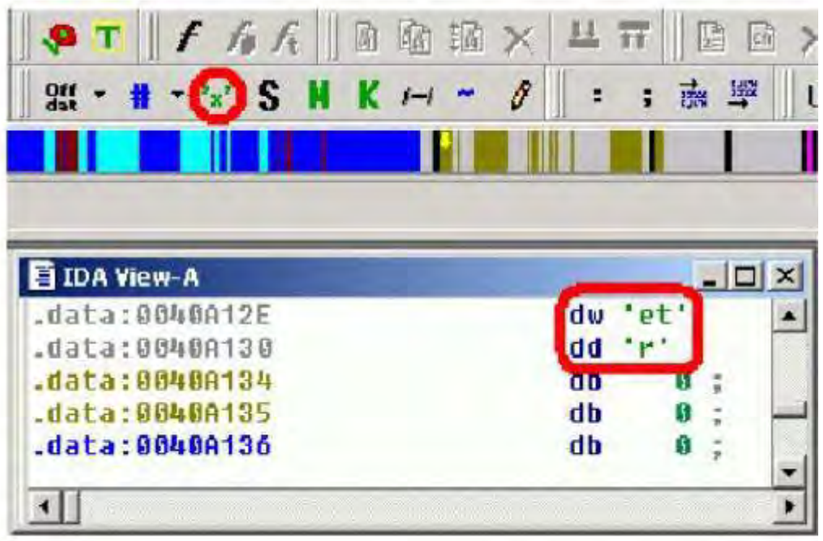


最后，如果你需要的格式不在其中，你可以自己手动定义。

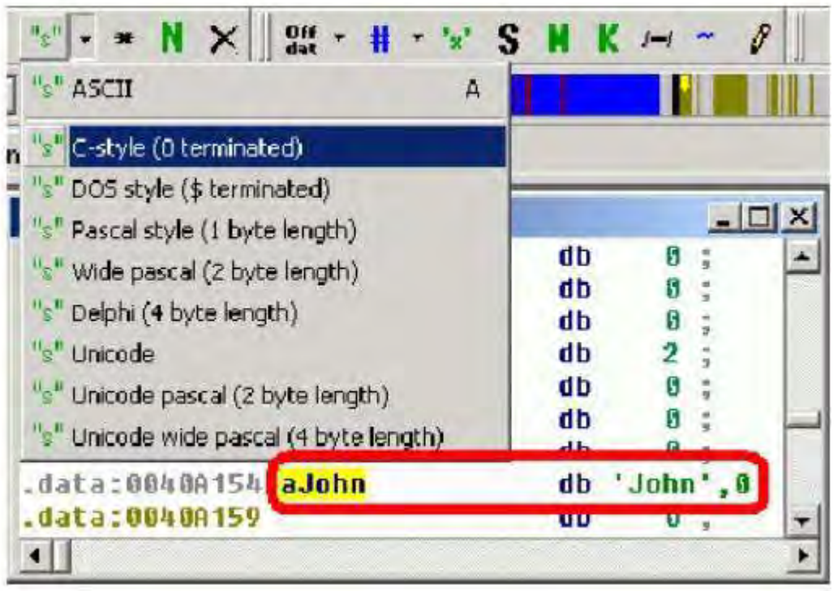


字符和字串

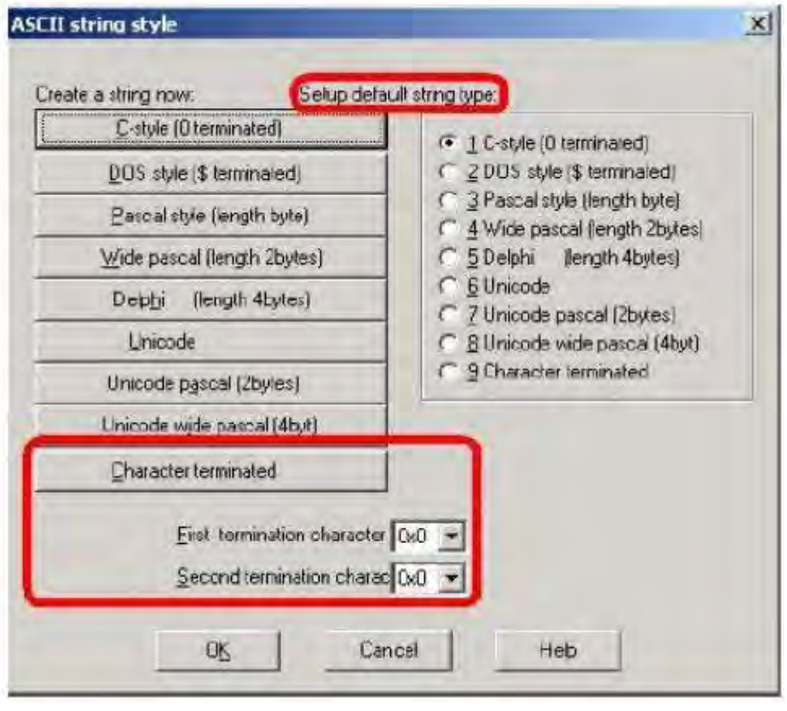
很多程序包含字符串。如果需要将定义的数据显示为字符，我们需要使用操作数工具条上的字串命令。



当然，有很多不同的字符串类型。IDA 通过 Strings 命令，支持多种字串类型。一旦你创建一个字串，IDA 自动给出一个字串名字。下面我们用这种方法来处理我们 C 程序中的一些字串。



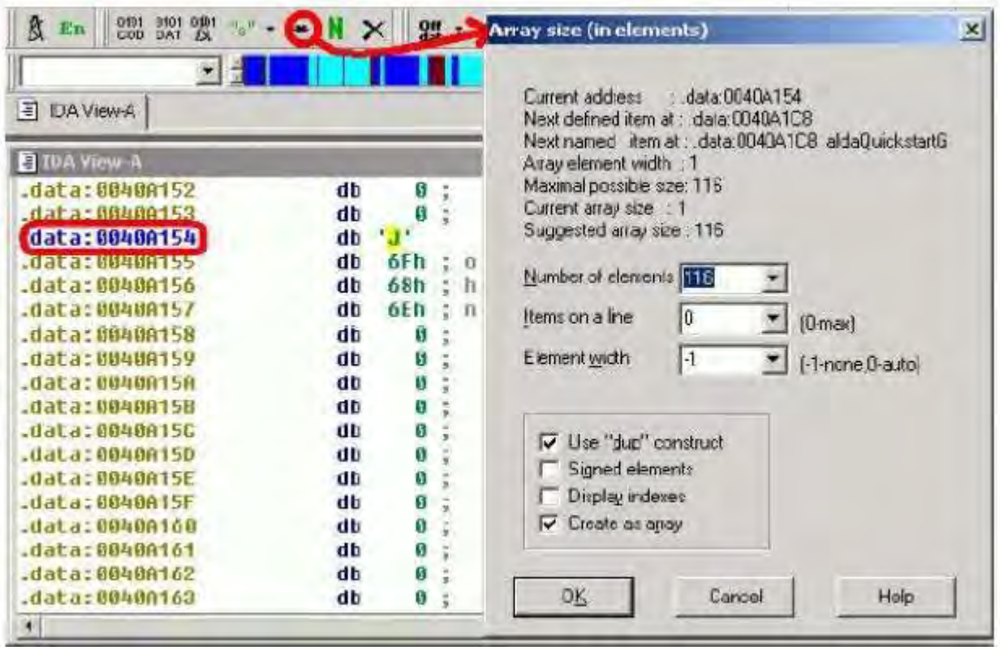
如果我们不是处理一个 C 程序该如何做呢？在 Options 菜单中提供了一些 ASCII 字串风格条目选项，允许我们来改变默认的字串类型。也可以手动定义一些带有不常用的结束字符的特殊的字串。



数组

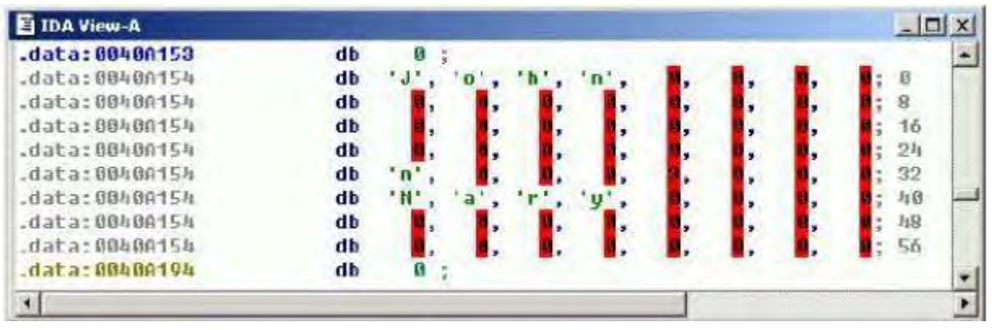
在 C 中，ASCII 字串通常使用一个字符数组来表示。IDA 是如何处理数组的呢？

首先，我们要从定义数组的第一个元素来开始。在本例中，我们设定第一元素类型为字节并设定它的格式为字符，接着我们按”*”键或使用 Edition 工具条的数组命令来产生实际的数组，这时，出现一个对话框，通过它我们可以生成不同的数组设定。

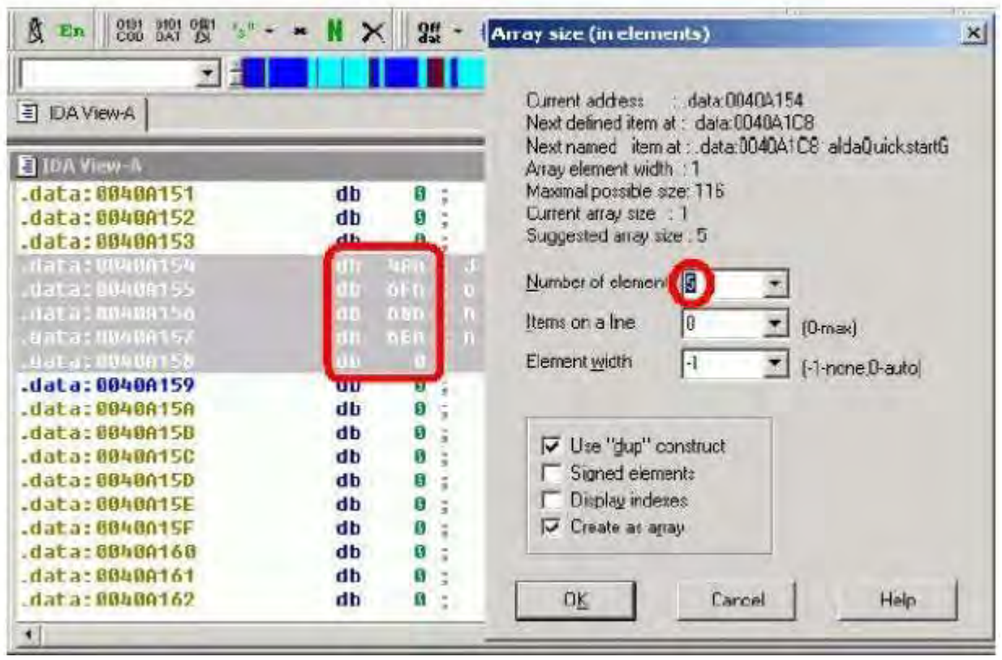


IDA 给出了建议的数组大小，它是根据在没有取消已有数据定义的前提下，可以使用的最大的范围来给出的。你可以指定在一行中显示的元素的数目，元素的宽度。DUP 结构选项允许你将相同的连续字节组合起来，Display Index 选项用于显示数组的索引。

例如，如果我们创建一个 64 个元素的数组，每行有 8 个元素，每个元素宽度为 4，不需要使用 dup 结构，带有索引注释，我们将得到下面这个数组形式

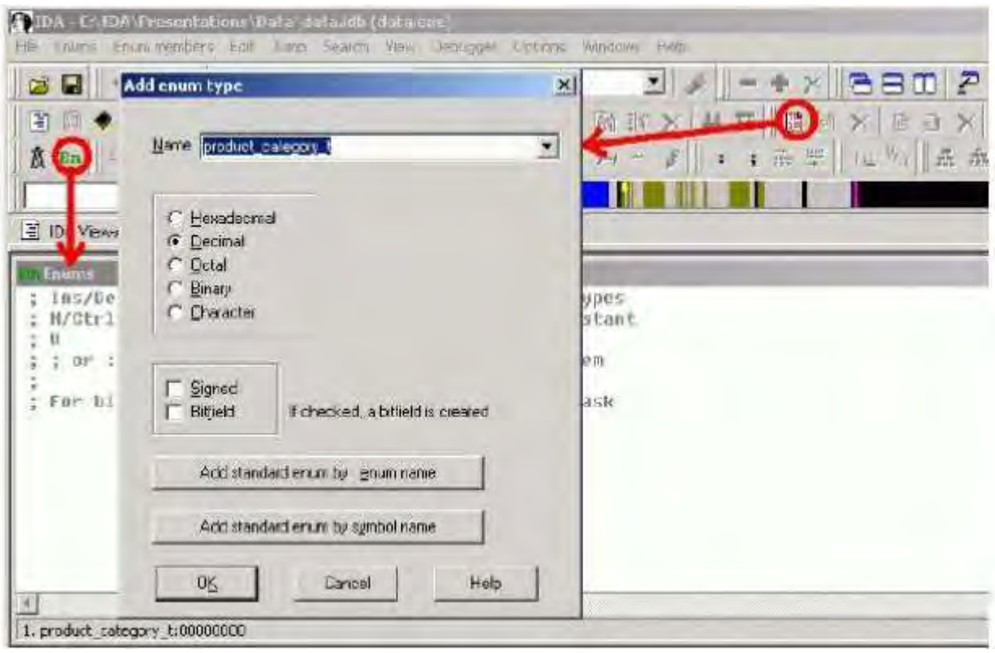


当 IDA 无法根据选择的类型（本例中是字符类型）来显示时，它会使用红色来高亮这些字节。也可以通过选择一个范围的方式，来生成一个合适大小的数组。

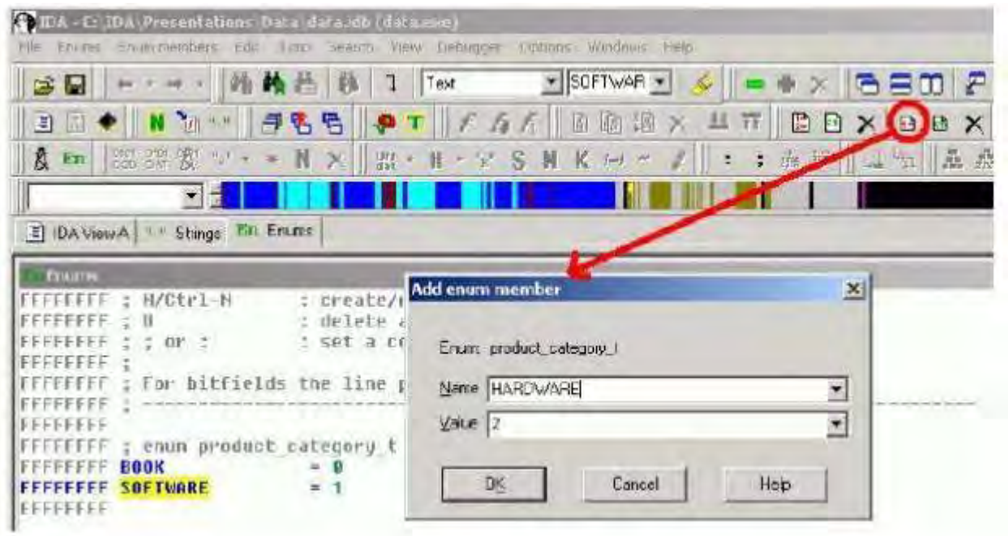


枚举类型

还记得在前面的 C 程序里定义的 product_category_t 类型码？我们可以在 IDA 中使用 Enumerations 来定义它。首先，我们打开 Enumerations 窗口，创建一个枚举类型。



在这个枚举类型中，我们添加枚举数据。

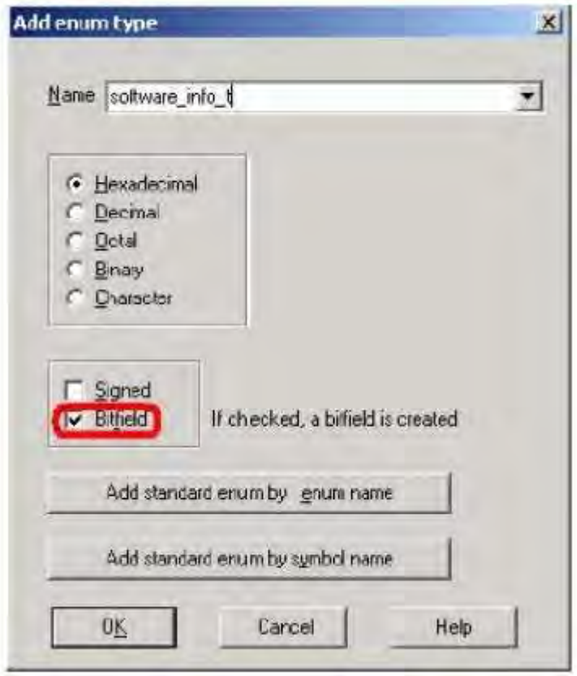


在 check_product()函数中，我们可以使用枚举类型来替换其中的部分操作数。在数值上使用鼠标右键菜单，选择 Symbolic constant：IDA 会列出与当前数值匹配的所有的枚举数值。

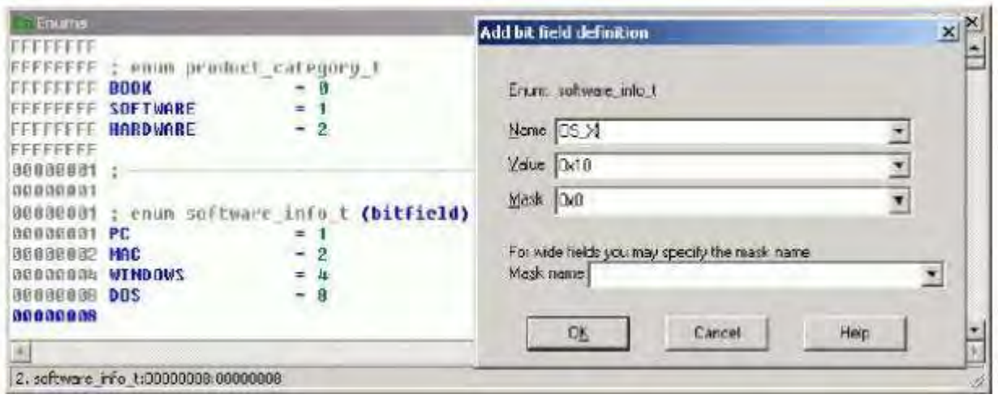
应用后，我们获得以下结果。

位域类型

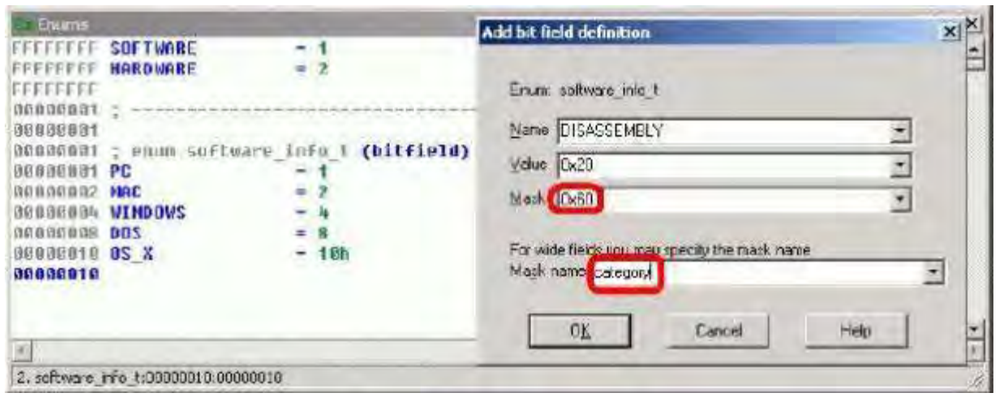
现在我们来尝试定义一些在 software_info_t 结构中定义的位域。从 IDA 的角度，位域只是一种特殊的枚举类型。我们在枚举类型创建对话框中选择位域的选项。



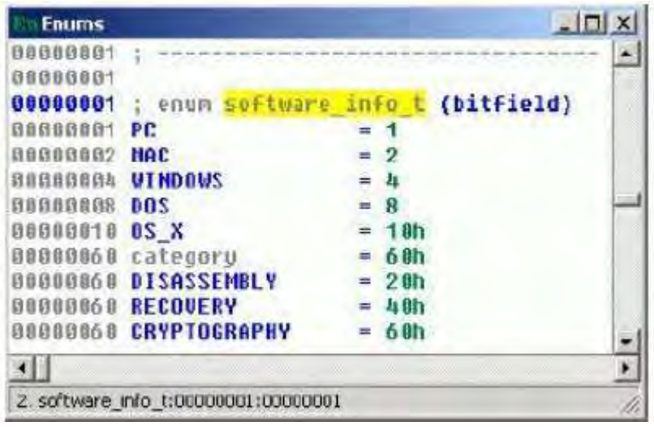
注意，在我们的 C 程序中，有两种不同的位域类型。Platform 和 os 位域包含在一个合并数值的掩码中（通过使用或逻辑运算符号）：一个产品能够应用在几个平台或操作系统中。另一方面，category 位域包含一个代表一个类别的数字：同时，一个产品只能属于一个类别！对 IDA,一个位域可以在一个特定的掩码中只包含一个数值。因此，为了表示 platform 和 category 位域，我们不得不为每个数值的每一位创建小位域，以允许他们组合。



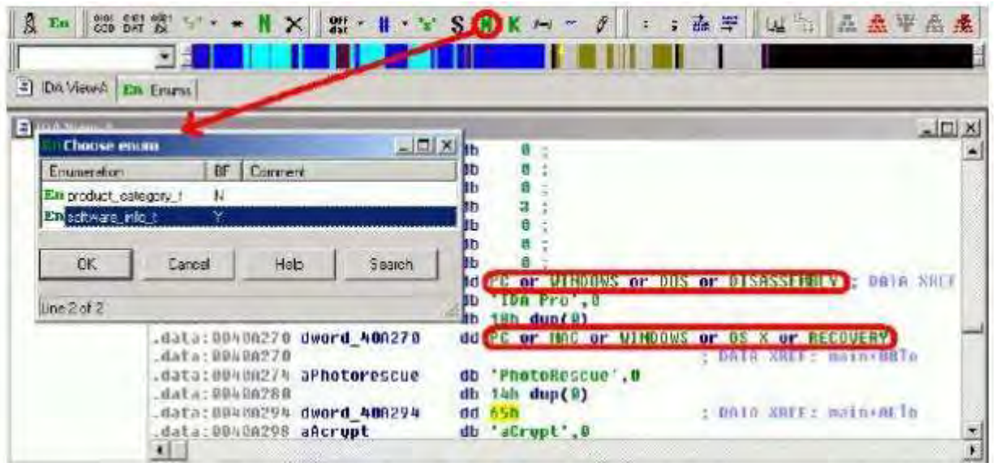
现在，我们可以创建类别位域，使用掩码数值 0x3(2bits)。我们指定一个成员名字，一个成员数值和一个位域掩码。我们也可以指定一个掩码名字：它不会被 IDA 使用，只是一个帮助记忆的注释。



当所有的位域都定义好后，我们可以获得如下定义

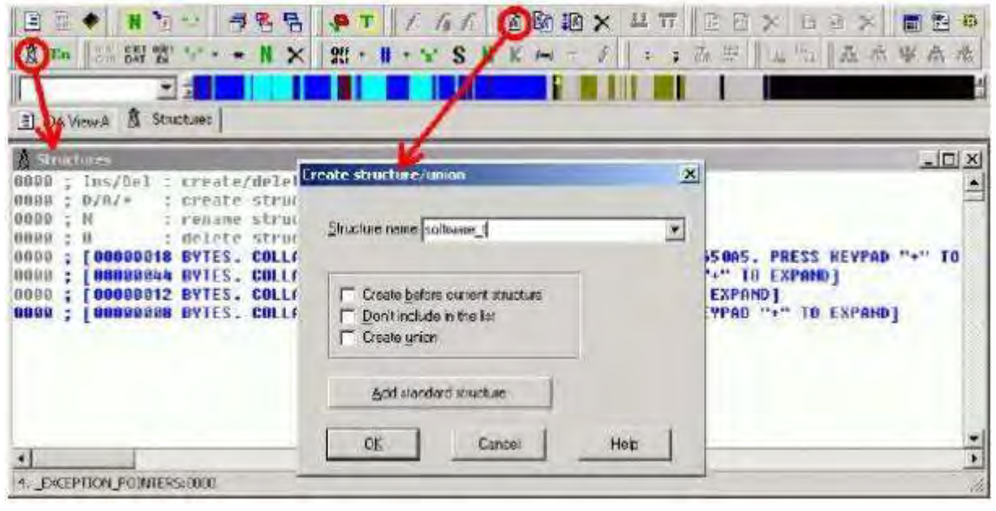


在 operands 工具条中使用枚举成员命令，来应用这些位域定义到我们的软件数据中。

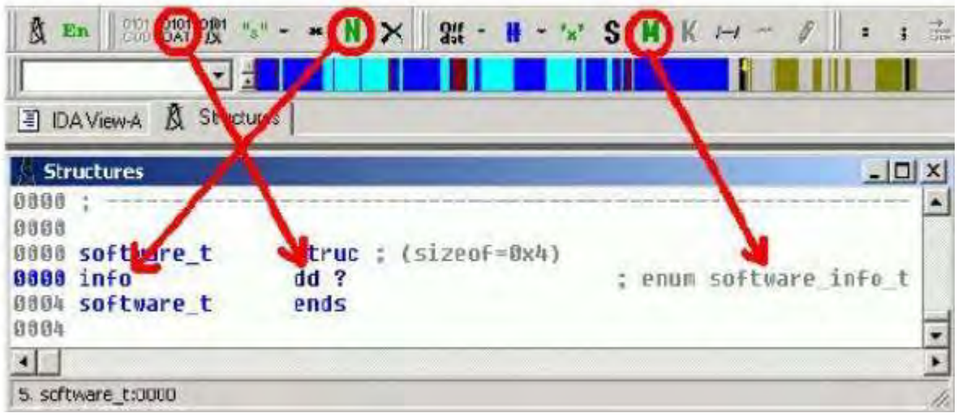


结构类型

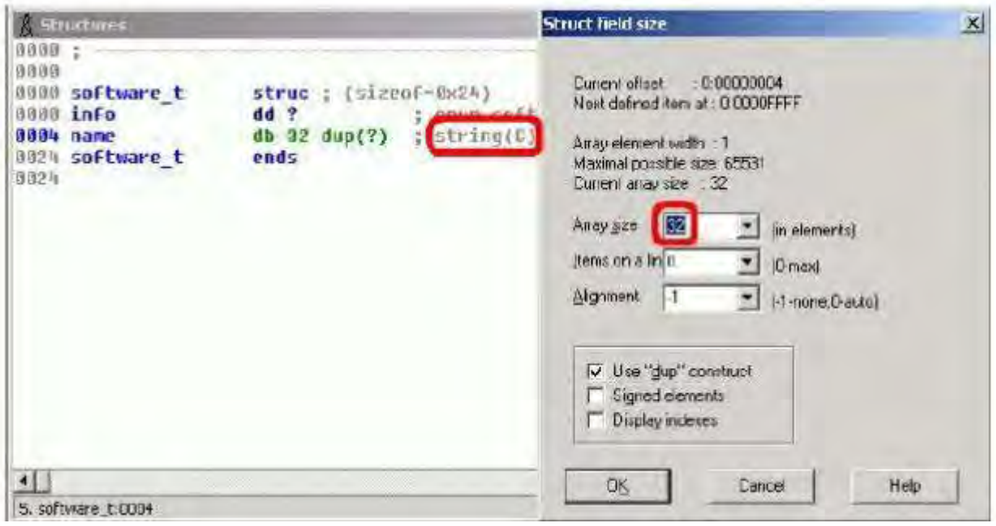
我们的程序中包含许多结构。我们可以在 IDA 中应用结构定义，看看它是如何提高反汇编结果的可读性。首先，我们打开结构窗口并创建一个新的结构类型。



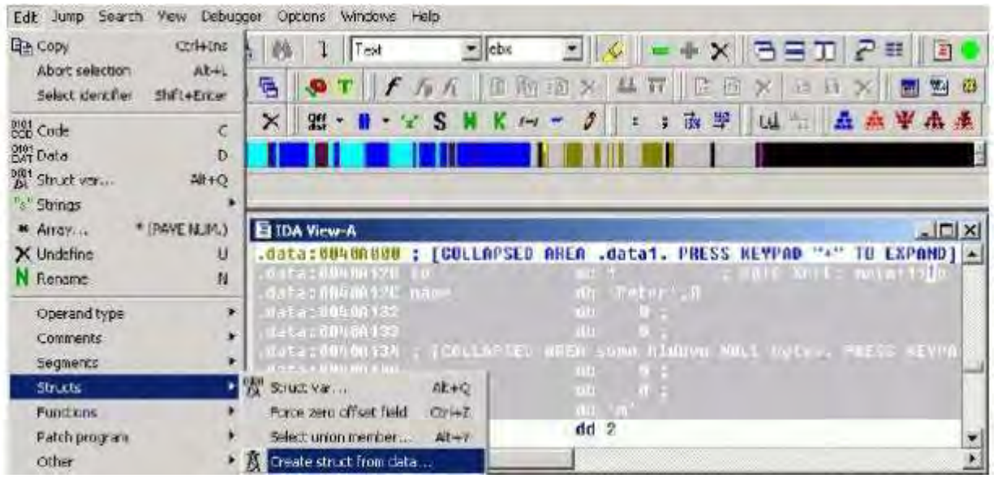
我们定义结构成员与在反汇编的视图窗口中定义数据的方法是类似的。首先我们定义 software_t 结构体的第一个成员。按“D”直到我们获得一个 dd 类型，它表示该成员的数值保存在一个双字里。我们指定它的格式为我们之前定义的 software_info_t 枚举结构，并通过 Rename 命令。给它一个合适的名字,Info。



接着，我们使用 ASCII 命令定义该结构体的第二个成员。在本例中，IDA 打开了一个对话框询问我们字串的大小。



我们也可以从已经定义的数据中创建一个结构类型。例如，假设我们已经在一个代表 customer_t 的结构的范围的数据做好了定义。我们可以使用 IDA，通过选择合适的范围，并使用“Create struct from data”命令来创建一个结构，。



一旦我们运行这个命令，IDA 创建了一个相应的结构，并打开结构窗口。为获得一个完美的结构类型，我们按”A”键来修正 name 成员的长度为 32 字节（正如我们源码中定义的），同时给这个结构一个更准确的名字。

对这些结构类型，我们该如何使用呢？IDA 提供给我们两种方式：

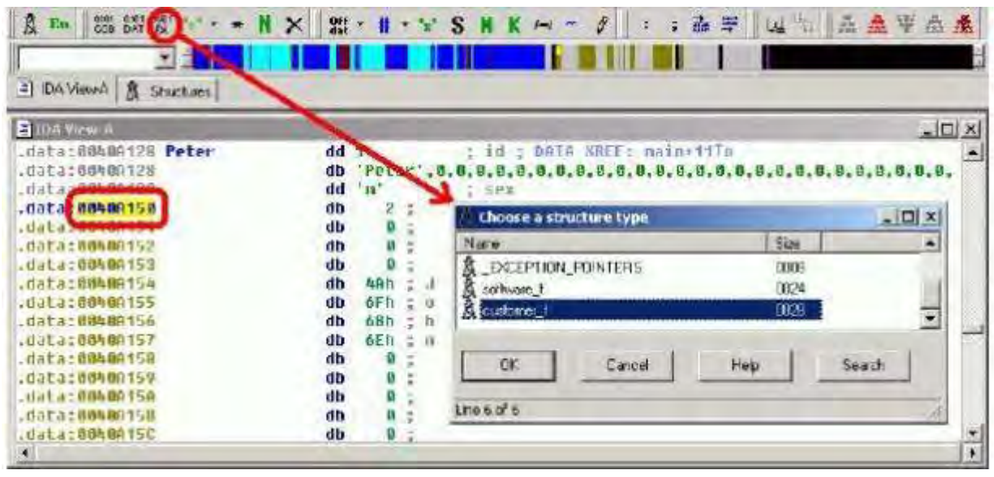
u 应用结构类型来初始化数据

u 转换操作数为结构中的偏移量

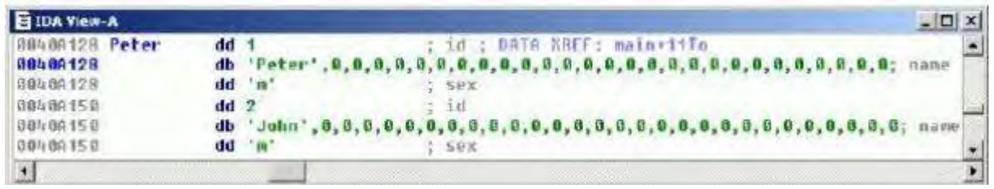
我们将会在本教程的后面演示这两种方式

结构变量和结构数组

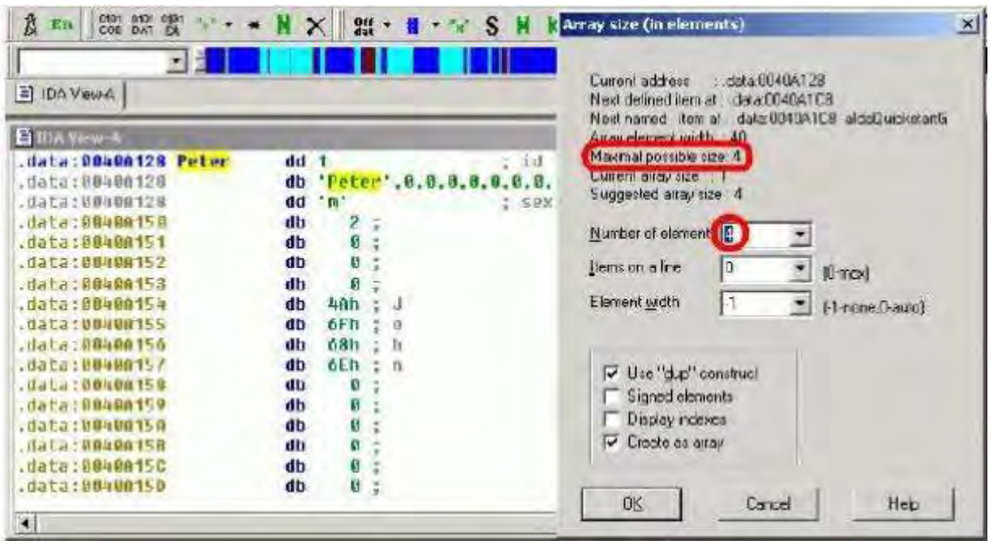
现在让我们定义一个包含 John 的客户信息的数据为一个 customer_t 结构。我们将鼠标指针放在该数据结构体的第一个字节上，然后使用 Struct var 命令



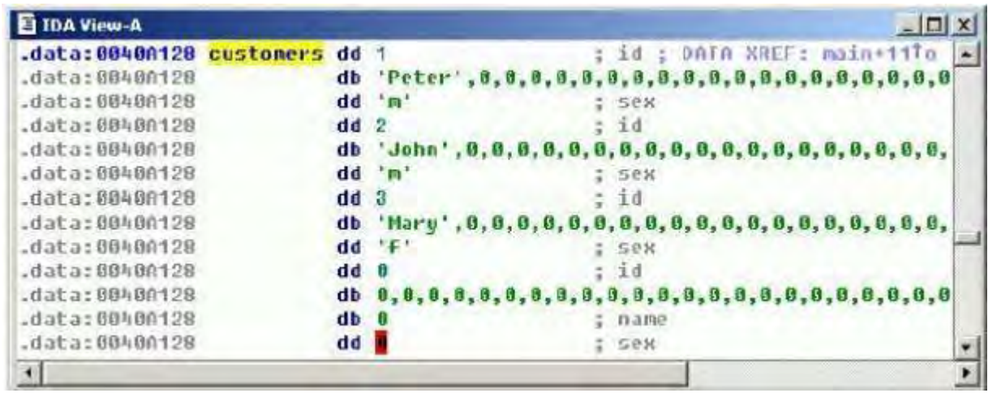
这样我们得到了一个新的结构体变量。注意 IDA 是如何使用注释显示结构体成员的名字的。



通过查看我们的源码，我们知道 customers 是一个包含四个成员的数组。我们前面已定义 Peter 和 John 为 customer_t 结构体。现在取消对 John 结构的定义，然后在 Peter 结构上按“*”键创建我们的 customer 数组，IDA 会弹出一个数组设置对话框，它会自动检测出我们能创建的最大的数组。



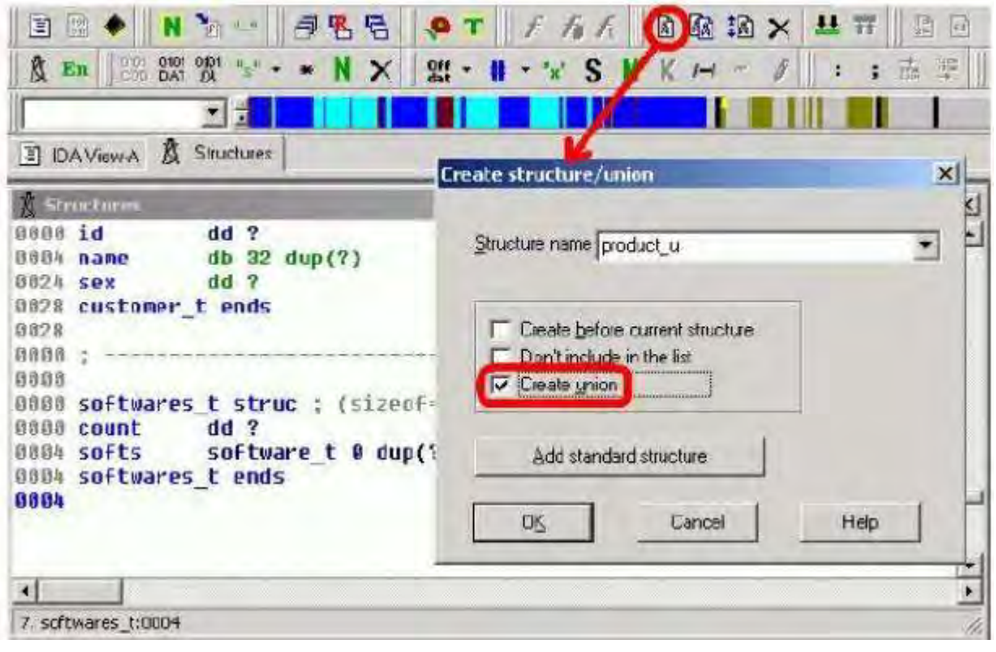
我们创建完这个数组，并给它一个合适的名字。



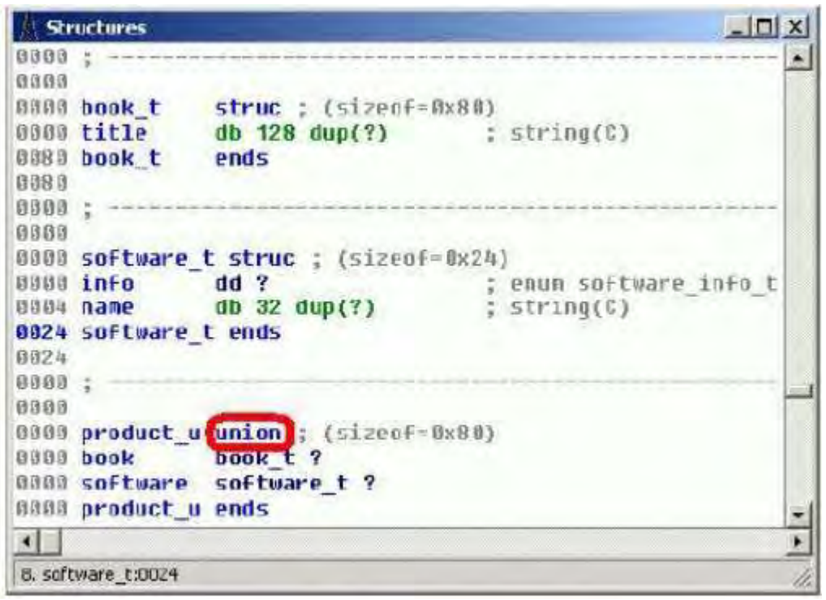
联合类型和结构中的结构

IDA 中定义联合体可以象定义标准结构体那样容易。

让我们来试着定义 product_u 这个联合体吧。我们假定 book_t 和 software_t 这两个结构体已经定义完成。对 IDA 来说，联合体相当于是一种特殊的结构体。我们打开 Structures 窗口，运行 Add struct type 命令，在对话框中我们选择创建 Create union 选项。

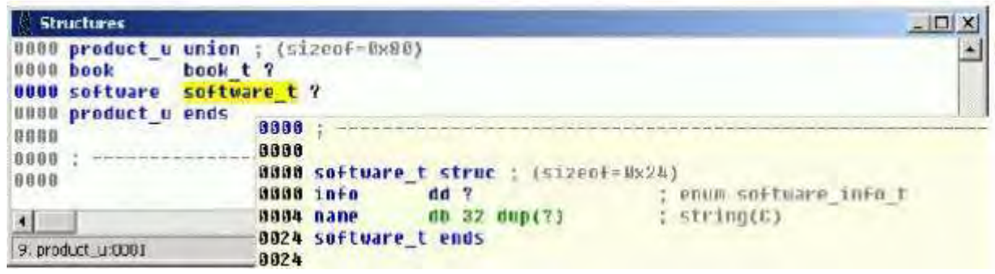


我们可以使用 IDA 所有常规数据定义命令来创建联合体成员，我们定义一个 book_t 类型的 book 成员，和一个 software_t 类型的 software 成员。



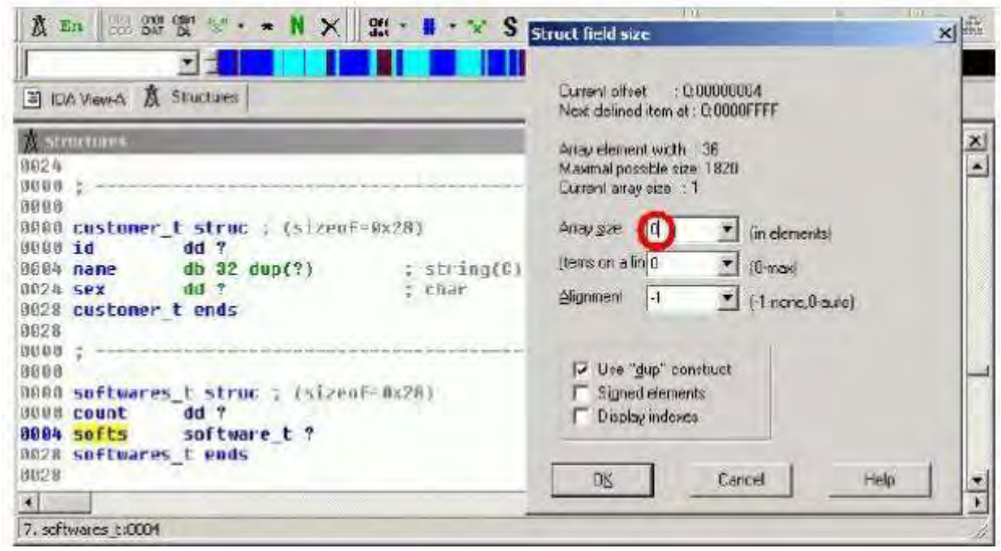
结构体中也也可以放置另一个结构体。事实上，我们在之前的创建联合体成员例子中就是这样做的。记住：IDA 认为联合体只是一种特殊的结构体。

将你的鼠标放在一个成员的结构名字上，可以看到它相关的结构类型的定义。



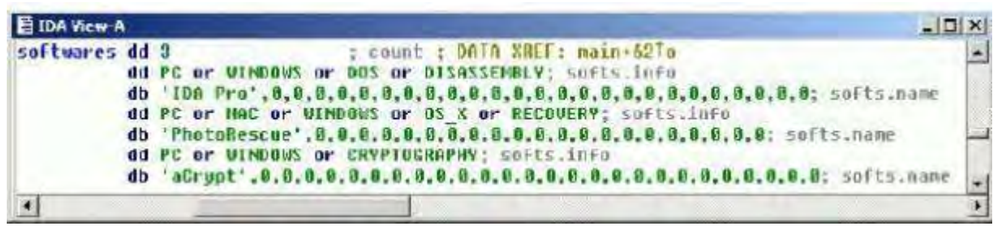
可变大小的结构体类型

现在我们看一下 softwares_t 这个结构体，结构体中 softs 域长度没有被指定。为了在反汇编中创建这样的结构，我们必须创建一种特殊类型的结构，我们称其为可变大小的结构。这种结构的创建域普通结构基本一样：唯一的不同是结构体的最后一个成员应被定义为一个有 0 个元素的数组。



由于 IDA 无法计算出这种结构体的大小，我们需要通过选择一个区域，来指定我们期望的结构体大小。

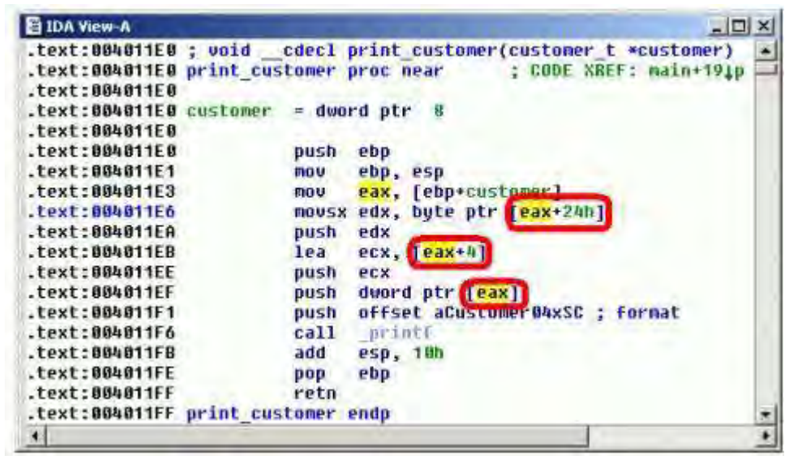
注意，IDA 是如何应用这些类型信息和如何以注释的方式来添加成员名称。



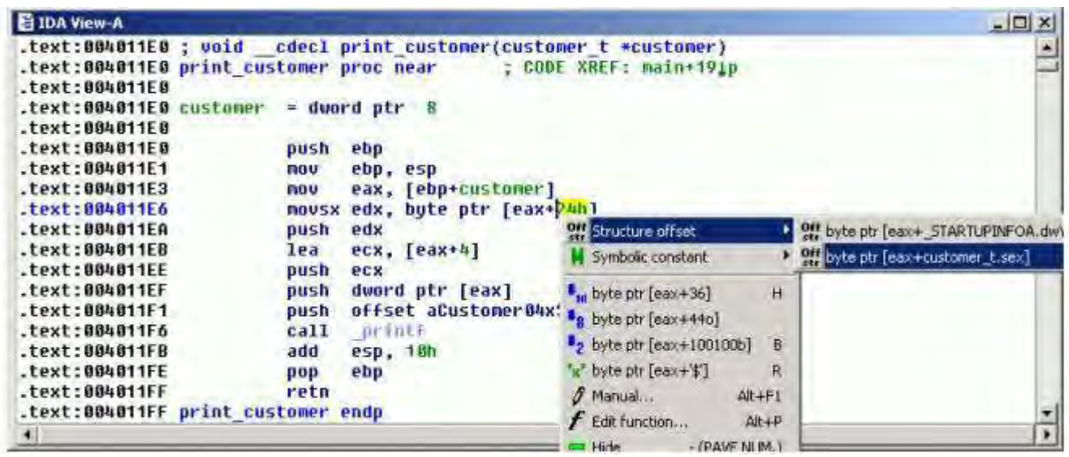
结构体偏移

现在我们已经知道如何定义我们需要的结构体和联合体。下面我们来看一下如何转换数据操作数为结构体的偏移。

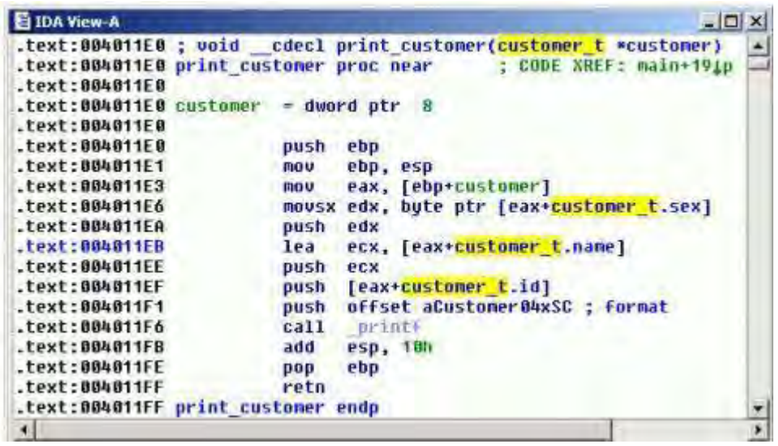
在 print_customer()函数中，我们知道唯一的参数是一个指向 cuttomer_t 的指针。EAX 寄存器被初始化为这个指针：它指向一个 customer_t 的结构。因此，我们推断出所有形式为[EAX+…]的操作数实际均是在 cutstomer_t 结构中的偏移。



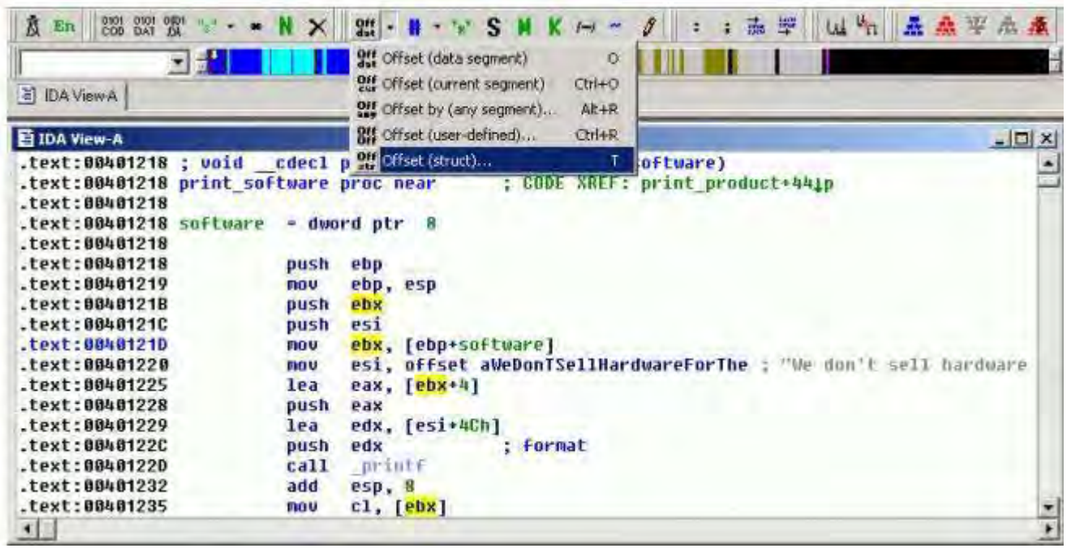
为了定义每个操作数为一个结构体的偏移，我们使用鼠标右击这个操作数：IDA 会列出所有可能的结构体偏移。



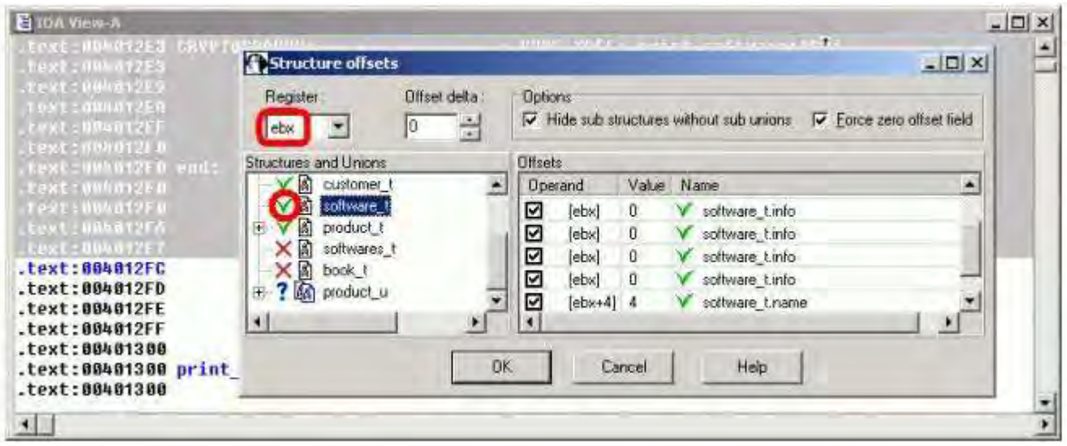
当我们把所有的数字偏移替换为我们结构体偏移后，汇编代码变得清晰多了。



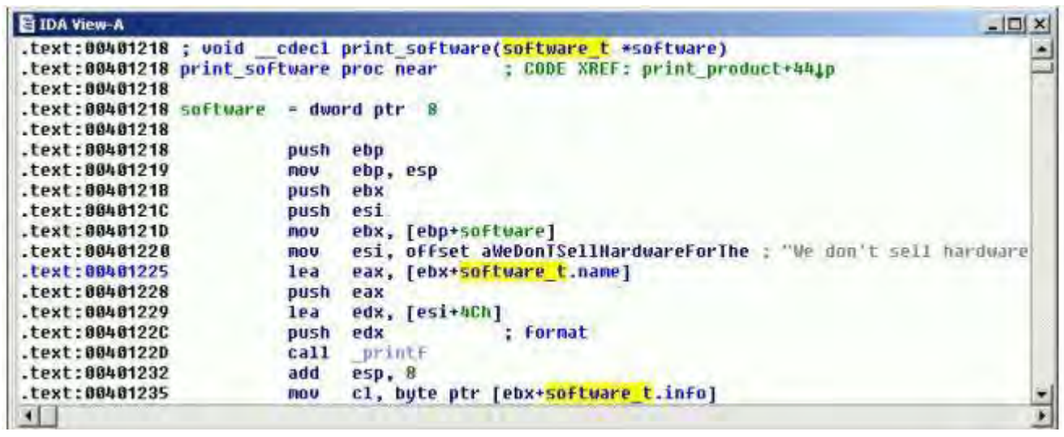
另外一个例子是 print_software()函数：EBX 寄存器被初始化为一个指向 software_t 结构体的指针。注意在整个函数中，EBX 寄存器都代表这个结构体。不用担心，IDA 可以使用一个简单操作就能替换所有偏移：点击 Operands 工具跳上的 Offset (struct) 命令。



此时 Structure offsets 窗口打开。我们在所有可能的寄存器列表中选择 EBX 寄存器。对话框左边树形视图显示了已在 IDA 中定义的所有结构。右边的列表显示了 EBX 相关的所有操作数。如果我们在左边树形视图中选择了一个结构，IDA 将会格式化选择的操作数为结构中的偏移量。名称前面的符号帮助我们确定是否所有的选择的操作数与选择的结构中的偏移匹配。在本例中，software_t 结构似乎与所有的操作数匹配。



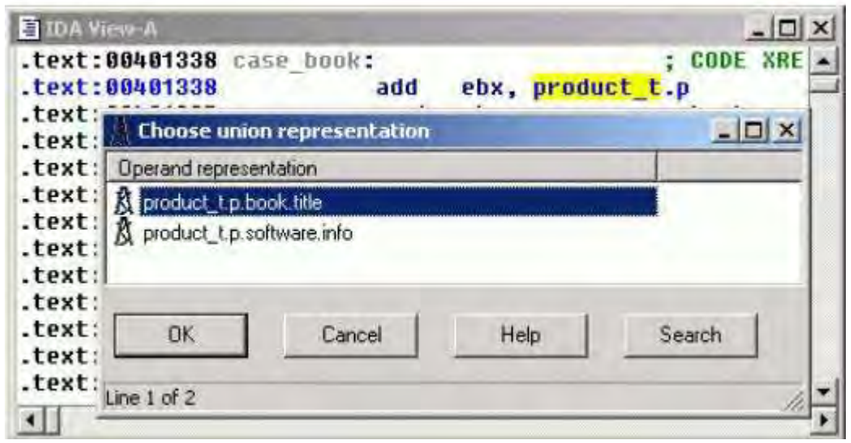
应用完成后，我们得到下面的结果：



联合体偏移

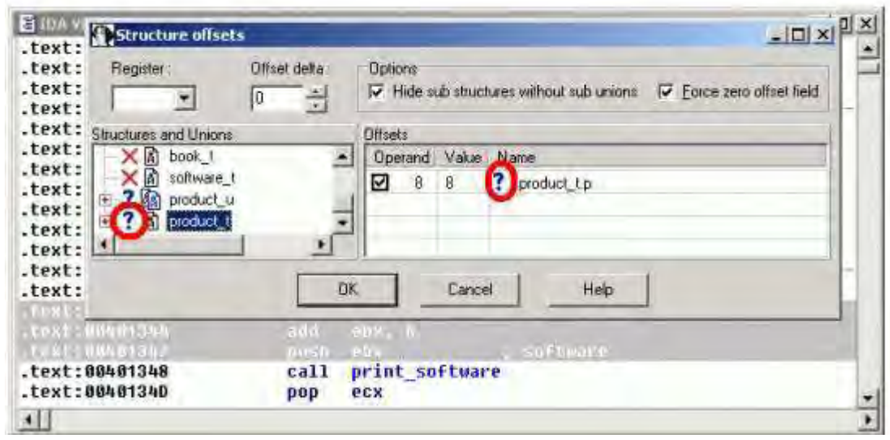
在 Print_func()中使用 EBX 寄存器指向了一个 product_u 的联合体。在这个函数的结尾，根据产品的种类，我们调用相应的函数来打印产品信息。

但此时结构偏移将会有一个不同的表示方法，如果它代表了 Product_u 联合体中第一个成员的偏移，或是这个联合体中的第二个成员的偏移。为了选择合适的成员，从 Edit struct 菜单中使用 select union member 命令。在这个对话框中，我们可以选择期望的联合体成员。

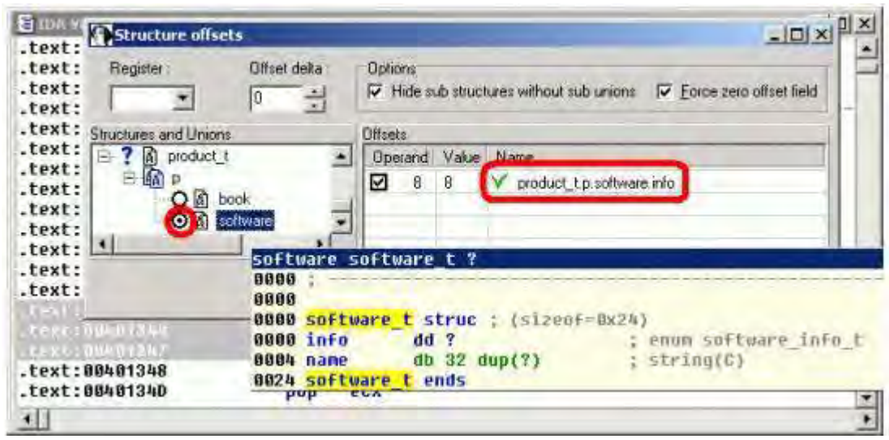


结果如下：

结构体偏移对话框显示了如何选择一个联合体成员影响了偏移的表示方法。如果我们选择一个区域并打开这个对话框，我们注意到结构体类型前面有个?号标记。



如果我们展开合适的树形分支，我们能够选择代表操作数偏移的联合体成员。一旦一个联合体成员被中(本例中是 software),IDA 使用一个绿色符号表示偏移与一个联合体成员记录匹配。

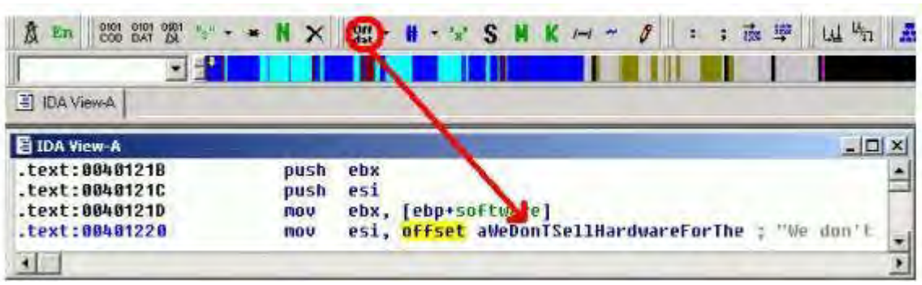


最后，如上图所示，你可以使用在树形视图中的提示信息来查看结构类型的声明，。

地址偏移

IDA 可以在反汇编程序中将操作数定义为偏移。在下面的例子中，橘黄色表示存在一个可能的有效参考。

使用 Operands 工具栏上的 Offset 按钮可以将这个操作数转换为一个偏移地址。



最终的反汇编结果

下面我们给出最终的完整反汇编结果，来结束这个教程：

```
; ---------------------------------------------------------------

customer_t struc ; (sizeof=0x28)

id dd ?

name db 32 dup(?) ; string(C)

sex dd ? ; char

customer_t ends

; ---------------------------------------------------------------

softwares_t struc ; (sizeof=0x4, variable size)

count dd ?

softs software_t 0 dup(?)

softwares_t ends

; ---------------------------------------------------------------

book_t struc ; (sizeof=0x80)

title db 128 dup(?) ; string(C)

book_t ends

; ---------------------------------------------------------------

software_t struc ; (sizeof=0x24)

info dd ? ; enum software_info_t
```

```
name db 32 dup(?) ; string(C)
software_t ends
; ---------------------------------------------------------------
product_u union ; (sizeof=0x80)
book book_t ?
software software_t ?
product_u ends
; ---------------------------------------------------------------
product_t struc ; (sizeof=0x88)
id dd ?
category dd ? ; enum product_category_t
p product_u ?
product_t ends
; ---------------------------------------------------------------
; enum product_category_t
BOOK = 0
SOFTWARE = 1
HARDWARE = 2
; ---------------------------------------------------------------
; enum software_info_t (bitfield)
PC = 1
MAC = 2
```

```
WINDOWS = 4

DOS = 8

OS_X = 10h

category = 60h

DISASSEMBLY = 20h

RECOVERY = 40h

CRYPTOGRAPHY = 60h

;

; +-------------------------------------------------------------------------+

; | This file is generated by The Interactive Disassembler (IDA) |

; | Copyright (c) 2005 by DataRescue sa/nv, <ida@datarescue.com> |

; | Licensed to: Eric <eric@datarescue.be> |

; +-------------------------------------------------------------------------+

;

; File Name : C:\IDA\Presentations\Data\data.exe

; Format : Portable executable for IBM PC (PE)

; Section 1. (virtual address 00001000)

; Virtual size : 00009000 ( 36864.)

; Section size in file : 00008E00 ( 36352.)

; Offset to raw data for section: 00000600

; Flags 60000020: Text Executable Readable

; Alignment : 16 bytes ?
```

```
unicode macro page,string,zero

irpc c,<string>

db '&c', page

endm

ifnb <zero>

dw zero

endif

endm

.686p

.mmx

.model flat

; ---------------------------------------------------------------------------

; Segment type: Pure code

; Segment permissions: Read/Execute

_text segment para public 'CODE' use32

assume cs:_text

;org 401000h

; [COLLAPSED AREA .text1. PRESS KEYPAD "+" TO EXPAND]

; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||

; Attributes: bp-based frame

; int __cdecl check_software(software_info_t software_info)

check_software proc near ; CODE XREF: main+108p
```

```
push ebp

mov ebp, esp

mov al, 1

mov dl, [ebp+software_info]

and edx, PC or MAC

test dl, PC

jz short not_PC

mov cl, [ebp+software_info]

and ecx, PC or MAC

test cl, MAC

jnz short end

mov dl, [ebp+software_info]

shr edx, 2

and edx, (WINDOWS or DOS or OS_X) >> 2

test dl, OS_X >> 2

jz short end

xor eax, eax

jmp short end

; ---------------------------------------------------------------------------

not_PC: ; CODE XREF: check_software+Ej

mov cl, [ebp+software_info]

and ecx, PC or MAC
```

```
test cl, MAC

jz short not_MAC

mov dl, [ebp+software_info]

and edx, PC or MAC

test dl, PC

jnz short end

mov cl, [ebp+software_info]

shr ecx, 2

and ecx, (WINDOWS or DOS or OS_X) >> 2

test cl, WINDOWS >> 2

jnz short not_windows

mov dl, [ebp+software_info]

shr edx, 2

and edx, (WINDOWS or DOS or OS_X) >> 2

test dl, DOS >> 2

jz short end

not_windows: ; CODE XREF: check_software+4Fj

xor eax, eax

jmp short end

; ---------------------------------------------------------------------------

not_MAC: ; CODE XREF: check_software+36j

xor eax, eax
```

```
end: ; CODE XREF: check_software+19j check_software+27j ...
xor edx, edx
mov dl, al
mov eax, edx
pop ebp
retn
check_software endp
; ---------------------------------------------------------------------------
align 4
; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||
; Attributes: bp-based frame
; int __cdecl check_product(product_category_t product_category)
check_product proc near ; CODE XREF: print_product+Ap main+D8p
product_category= dword ptr 8
push ebp
mov ebp, esp
push ebx
mov bl, 1
cmp [ebp+product_category], HARDWARE
jnz short not_hardware
xor ebx, ebx
push offset aWeDonTSellHardwareForThe ; format
```

```
call _printf

pop ecx

not_hardware: ; CODE XREF: check_product+Aj

xor eax, eax

mov al, bl

pop ebx

pop ebp

retn

check_product endp

; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||

; Attributes: bp-based frame

; void __cdecl print_customer(customer_t *customer)

print_customer proc near ; CODE XREF: main+19p

customer= dword ptr 8

push ebp

mov ebp, esp

mov eax, [ebp+customer]

movsx edx, byte ptr [eax+customer_t.sex]

push edx

lea ecx, [eax+customer_t.name]

push ecx

push [eax+customer_t.id]
```

```
push offset aCustomer04xSC ; format

call _printf

add esp, 10h

pop ebp

retn

print_customer endp

; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||

; Attributes: bp-based frame

; void __cdecl print_book(book_t *book)

print_book proc near ; CODE XREF: print_product+38p

book= dword ptr 8

push ebp

mov ebp, esp

push [ebp+book]

push offset aBookS ; format

call _printf

add esp, 8

pop ebp

retn

print_book endp

; ---------------------------------------------------------------------------

align 4
```

```
; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||
; Attributes: bp-based frame
; void __cdecl print_software(software_t *software)
print_software proc near ; CODE XREF: print_product+44p
software= dword ptr 8
push ebp
mov ebp, esp
push ebx
push esi
mov ebx, [ebp+software]
mov esi, offset aWeDonTSellHardwareForThe ; "We don't sell hardware for the
moment.."...
lea eax, [ebx+software_t.name]
push eax
lea edx, (aSoftwareS - aWeDonTSellHardwareForThe)[esi] ; "We don't sell
hardware for the moment.."...
push edx ; format
call _printf add esp, 8
mov cl, byte ptr [ebx+software_t.info]
and ecx, PC or MAC
test cl, PC
jz short not_pc
```

```
lea eax, (aPc - aWeDonTSellHardwareForThe)[esi] ; "We don't sell hardware
for the moment.."...
push eax ; format
call _printf
pop ecx
not_pc:
mov dl, byte ptr [ebx+software_t.info]
and edx, PC or MAC
test dl, MAC
jz short not_mac
lea ecx, (aMac - aWeDonTSellHardwareForThe)[esi] ; "We don't sell hardware
for the moment.."...
push ecx ; format
call _printf
pop ecx
not_mac: ; "We don't sell hardware for the moment.."...
lea eax, (asc_40A31B - aWeDonTSellHardwareForThe)[esi]
push eax ; format
call _printf
pop ecx
mov dl, byte ptr [ebx+software_t.info]
shr edx, 2
```

```
and edx, (WINDOWS or DOS or OS_X) >> 2

test dl, WINDOWS >> 2

jz short not_windows

lea ecx, (aWindows - aWeDonTSellHardwareForThe)[esi] ; "We don't sell

hardware for the moment.."...

push ecx ; format

call _printf

pop ecx

not_windows:

mov al, byte ptr [ebx+software_t.info]

shr eax, 2

and eax, (WINDOWS or DOS or OS_X) >> 2

test al, DOS >> 2

jz short not_dos

lea edx, (aDos - aWeDonTSellHardwareForThe)[esi] ; "We don't sell hardware

for the moment.."...

push edx ; format

call _printf

pop ecx

not_dos:

mov cl, byte ptr [ebx+software_t.info]

shr ecx, 2
```

```
and ecx, (WINDOWS or DOS or OS_X) >> 2
test cl, OS_X >> 2
jz short not_os_x
lea eax, (aOsX - aWeDonTSellHardwareForThe)[esi] ; "We don't sell hardware
for the moment..."...
push eax ; format
call _printf
pop ecx
not_os_x: ; "We don't sell hardware for the moment..."...
lea edx, (asc_40A331 - aWeDonTSellHardwareForThe)[esi]
push edx ; format
call _printf
pop ecx
mov cl, byte ptr [ebx+software_t.info]
shr ecx, 5
and ecx, category >> 5
dec ecx
jz short DISASSEMBLY
dec ecx
jz short RECOVERY
dec ecx
jz short CRYPTOGRAPHY
```

```
jmp short end
; ---------------------------------------------------------------------------
DISASSEMBLY: ; "We don't sell hardware for the moment.."...
lea eax, (aDisassembly - aWeDonTSellHardwareForThe)[esi]
push eax ; format
call _printf
pop ecx
jmp short end
; ---------------------------------------------------------------------------
RECOVERY: ; "We don't sell hardware for the moment.."...
lea edx, (aRecovery - aWeDonTSellHardwareForThe)[esi]
push edx ; format
call _printf
pop ecx
jmp short end
; ---------------------------------------------------------------------------
CRYPTOGRAPHY: ; "We don't sell hardware for the moment.."...
lea ecx, (aCryptography - aWeDonTSellHardwareForThe)[esi]
push ecx ; format
call _printf
pop ecx
end: ; "We don't sell hardware for the moment.."...
```

```
lea eax, (asc_40A358 - aWeDonTSellHardwareForThe)[esi]
push eax ; format
call _printf
pop ecx
pop esi
pop ebx
pop ebp
retn
print_software endp
; ---------------------------------------------------------------------------
align 4
; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||
; Attributes: bp-based frame
; int __cdecl print_product(product_t *product)
print_product proc near ; CODE XREF: main+128p
product= dword ptr 8
push ebp
mov ebp, esp
push ebx
mov ebx, [ebp+product]
push [ebx+product_t.category] ; product_category
call check_product
```

```
pop ecx

test eax, eax

jnz short check_product_ok

xor eax, eax

pop ebx

pop ebp

retn

; ---------------------------------------------------------------------------

check_product_ok: ; CODE XREF: print_product+12j

push [ebx+product_t.id]

push offset aProduct04x ; format

call _printf

add esp, 8

mov edx, [ebx+product_t.category]

sub edx, 1

jb short case_book

jz short case_software

jmp short default

; ---------------------------------------------------------------------------

case_book: ; CODE XREF: print_product+2Ej

add ebx, product_t.p.book.title

push ebx ; book
```

```
call print_book

pop ecx

jmp short default

; ---------------------------------------------------------------------------

case_software: ; CODE XREF: print_product+30j

add ebx, product_t.p.software.info

push ebx ; software

call print_software

pop ecx

default: ; CODE XREF: print_product+32j print_product+3Ej

mov al, 1

pop ebx

pop ebp

retn

print_product endp

; ---------------------------------------------------------------------------

align 4

; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||

; Attributes: bp-based frame

; void __cdecl main()

main proc near ; DATA XREF: .data:0040A0D0o

push ebp
```

```
mov ebp, esp

push ebx

push esi

push edi

push offset aCustomers ; format

call _printf

pop ecx

mov ebx, offset customers

jmp short loc_401376

; ---------------------------------------------------------------------------

loop_print_customer: ; CODE XREF: main+25j

push ebx ; customer

call print_customer

pop ecx

add ebx, 40

loc_401376: ; CODE XREF: main+16j

cmp [ebx+customer_t.id], 0

jnz short loop_print_customer

push 544 ; size

call _malloc

pop ecx

mov ebx, eax
```

```
mov [ebx+product_t.id], 1

xor eax, eax ; BOOK

mov [ebx+product_t.category], eax

mov esi, offset aIdaQuickstartG ; "IDA QuickStart Guide"

lea edi, [ebx+product_t.p.book.title]

mov ecx, 32

rep movsd

mov dword ptr [ebx+product_t[1].id], 2

mov dword ptr [ebx+product_t[1].category], SOFTWARE

mov esi, offset softwares.softs

lea edi, [ebx+product_t[1].p.software]

mov ecx, 9

rep movsd

mov dword ptr [ebx+product_t[2].id], 3

mov dword ptr [ebx+product_t[2].category], SOFTWARE

mov esi, (offset softwares.softs.info+24h)

lea edi, [ebx+product_t[2].p.software]

mov ecx, 9

rep movsd

mov dword ptr [ebx+product_t[3].id], 4

mov dword ptr [ebx+product_t[3].category], SOFTWARE

mov esi, (offset softwares.softs.info+48h)
```

```
lea edi, [ebx+product_t[3].p.software]
mov ecx, 9
rep movsd
push offset aProducts ; format
call _printf
pop ecx
xor esi, esi
loop_verify_print_product: ; CODE XREF: main+132j
mov eax, esi
shl eax, 4
add eax, esi
push [ebx+eax*8+product_t.category] ; product_category
call check_product
pop ecx
test eax, eax
jnz short product_is_valid
push offset aInvalidProduct ; format
call _printf
pop ecx
jmp short exit
; ---------------------------------------------------------------------------
product_is_valid: ; CODE XREF: main+E0j
```

```
mov edx, esi

shl edx, 4

add edx, esi

cmp [ebx+edx*8+product_t.category], SOFTWARE

jnz short print_product

mov ecx, esi

shl ecx, 4

add ecx, esi

push [ebx+ecx*8+product_t.p.software.info] ; software_info

call check_software

pop ecx

test eax, eax

jnz short print_product

push offset aInvalidSoftwar ; format

call _printf

pop ecx

jmp short exit

; ---------------------------------------------------------------------------

print_product: ; CODE XREF: main+FBj main+110j

imul eax, esi, 88h

add eax, ebx

push eax ; product
```

```
call print_product

pop ecx

inc esi

cmp esi, 4

jl short loop_verify_print_product

exit: ; CODE XREF: main+EDj main+11Dj

push ebx ; block

call _free

pop ecx

pop edi

pop esi

pop ebx

pop ebp

retn

main endp

; [COLLAPSED AREA .text2. PRESS KEYPAD "+" TO EXPAND]

; Section 2. (virtual address 0000A000)

; Virtual size : 00003000 ( 12288.)

; Section size in file : 00002800 ( 10240.)

; Offset to raw data for section: 00009400

; Flags C0000040: Data Readable Writable

; Alignment : 16 bytes ?
```

```
; ---------------------------------------------------------------------------

; Segment type: Pure data

; Segment permissions: Read/Write

_data segment para public 'DATA' use32

assume cs:_data

;org 40A000h

; [COLLAPSED AREA .data1. PRESS KEYPAD "+" TO EXPAND]

customers customer_t <1, 'Peter', 'm'> ; DATA XREF: main+11o

customer_t <2, 'John', 'm'>

customer_t <3, 'Mary', 'f'>

customer_t <0>

aIdaQuickstartG db 'IDA QuickStart Guide',0 ; DATA XREF: main+3Fo

db 6Bh dup(0)

softwares dd 3 ; count ; DATA XREF: main+62o

dd PC or WINDOWS or DOS or DISASSEMBLY; softs.info

db 'IDA Pro',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; softs.name

dd PC or MAC or WINDOWS or OS_X or RECOVERY; softs.info

db 'PhotoRescue',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; softs.name

dd PC or WINDOWS or CRYPTOGRAPHY; softs.info

db 'aCrypt',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; softs.name

aWeDonTSellHardwareForThe db 'We don',27h,'t sell hardware for the

moment...',0Ah,0      ; DATA XREF: check_product+Eo print_software+8o ...
```

```
aCustomer04xSC db 'CUSTOMER %04X: %s (%c)',0Ah,0 ; DATA XREF: print_customer+11o
aBookS db 'BOOK: %s',0Ah,0 ; DATA XREF: print_book+6o
aSoftwareS db 'SOFTWARE: %s:',0 ; DATA XREF: print_software+11r
aPc db ' PC',0       ; DATA XREF: print_software+27r
aMac db ' MAC',0      ; DATA XREF: print_software+3Br
asc_40A31B db ';',0    ; DATA XREF: print_software:not_macr
aWindows db ' WINDOWS',0   ; DATA XREF: print_software+5Cr
aDos db ' DOS',0      ; DATA XREF: print_software+72r
aOsX db ' OS-X',0     ; DATA XREF: print_software+89r
asc_40A331 db ';',0    ; DATA XREF: print_software:not_os_xr
aDisassembly db ' DISASSEMBLY',0 ; DATA XREF: print_software:DISASSEMBLYr
aRecovery db ' RECOVERY',0 ; DATA XREF: print_software:RECOVERYr
aCryptography db ' CRYPTOGRAPHY',0 ; DATA XREF: print_software:CRYPTOGRAPHYr
asc_40A358 db 0Ah,0    ; DATA XREF: print_software:endr
aProduct04x db 'PRODUCT %04X: ',0 ; DATA XREF: print_product+1Bo
aCustomers db 'CUSTOMERS:',0Ah,0 ; DATA XREF: main+6o
aProducts db 0Ah      ; DATA XREF: main+C0o
db 'PRODUCTS:',0Ah,0
aInvalidProduct db 'Invalid product !!!',0Ah,0 ; DATA XREF: main+E2o
aInvalidSoftwar db 'Invalid software !!!',0Ah,0 ; DATA XREF: main+112o
; [COLLAPSED AREA .data2. PRESS KEYPAD "+" TO EXPAND]
; ---------------------------------------------------------------------------
```

```
; [00001000 BYTES: COLLAPSED SEGMENT _tls. PRESS KEYPAD "+" TO EXPAND]
; ---------------------------------------------------------------------------
; [00001000 BYTES: COLLAPSED SEGMENT _rdata. PRESS KEYPAD "+" TO EXPAND]
; ---------------------------------------------------------------------------
; [000000C4 BYTES: COLLAPSED SEGMENT _idata. PRESS KEYPAD "+" TO EXPAND]
```

http://blog.csdn.net/eqera/article/details/8239994

IDA 使用小结

这边文章是对于 IDA Pro 的使用小结。想要学习 IDA 使用的请看这里。 我这里只是小小的个人的总结。不是作为教程来写的。

1. Begin

这里有这样一段代码：

```
#include "stdafx.h"
#include <stdlib.h>

int _tmain(int argc, _TCHAR* argv[])
{
    int i = 0;
    ++i;
    double b = 0.0;
    char c = 'a';

    printf("%d, %f, %c", i, b, c);

    return 0;
}
```

我们用 IDA 来进行反汇编的练习。

```
_main proc near

var_C= qword ptr -0Ch
argc= dword ptr  4
argv= dword ptr  8
envp= dword ptr  0Ch

fldz
push    61h
sub     esp, 8
fstp    [esp+0Ch+var_C]
push    1
push    offset Format   ; "%d, %f, %c"
call    ds:__imp__printf
add     esp, 14h
xor     eax, eax
retn
_main endp
```

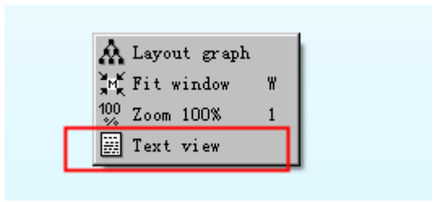
发现 IDA 的反汇编还是比较强大的而且结构也比较清晰。

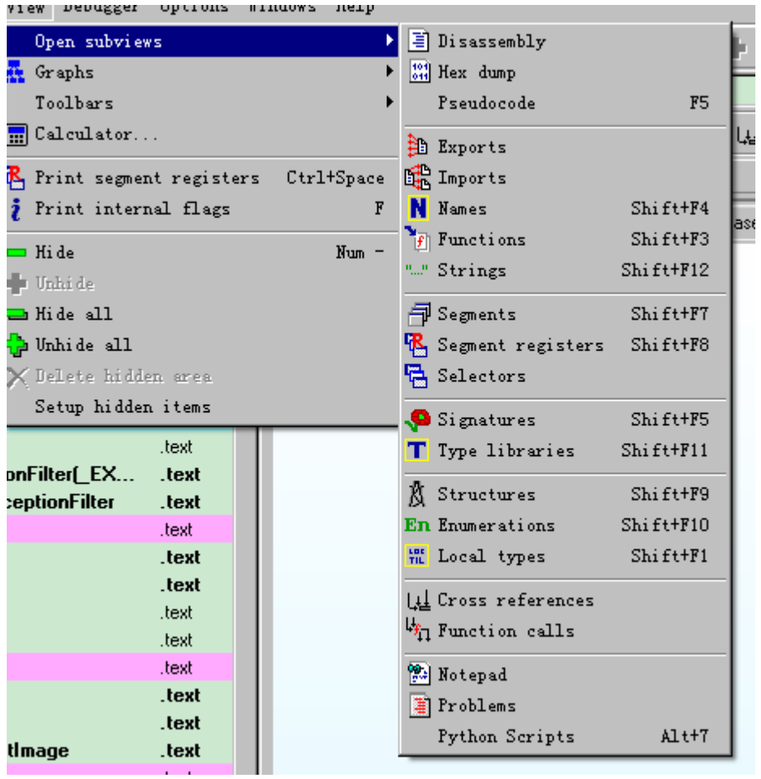
Var_C 表明声明了一块内存地址。

argc, argv, envp 是对函数参数的赋值。

fldz 清除状态寄存器
push 61h 第三个参数
sub esp8, fstp [esp+0ch+var_c], 第二个参数;
调用 printf 函数
堆栈平衡。

我们可以按改变不同类型的视图。也可以在这里打开几个相同的 view





2. 基本类型的识别

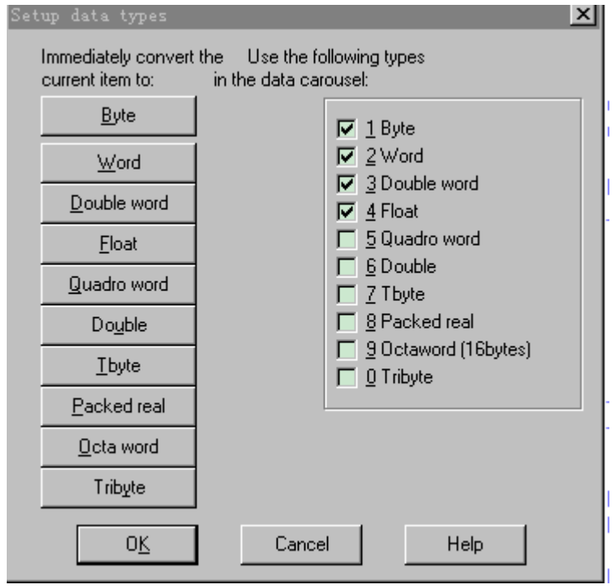
我们按下 Ctrl+S 可以转到不同的段中。

这里选择数据段。

只要按"D"我们就可以任意转换这些不确定的类型.可以变成 byte,word,dword(db,dw,dd)。

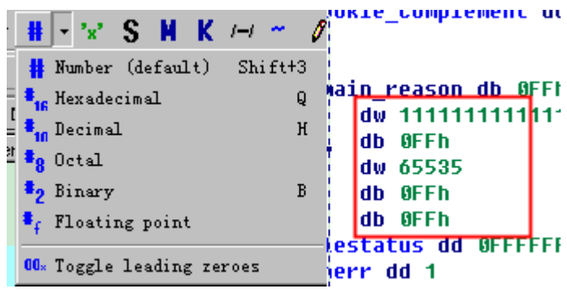
```
data:00403009                   dw 0FFFFh
data:0040300B                   db 0FFh
data:0040300C                   dw 0FFFFh
data:0040300E                   db 0FFh
data:0040300F                   db 0FFh
```

选择“*Options*”菜单的“*Setup data types*”命令就可以设置了



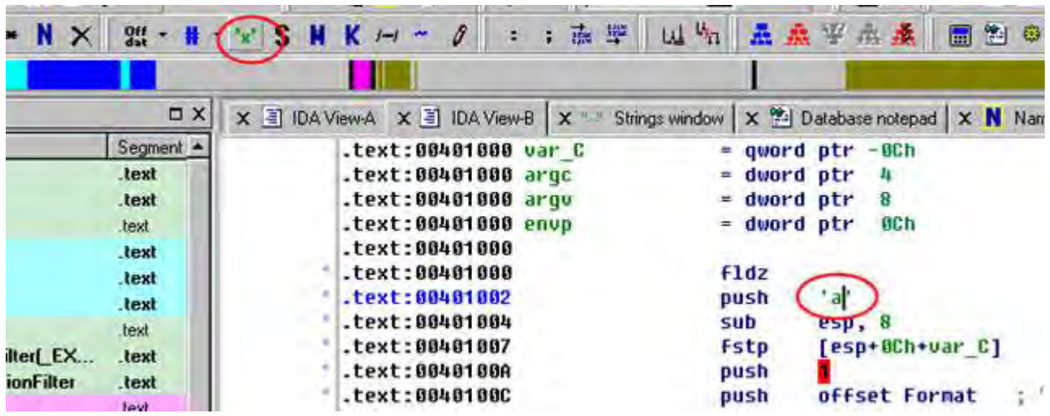
撤销所有的转换按”U”键。

可以按下这里来选择显示数据的进制格式：



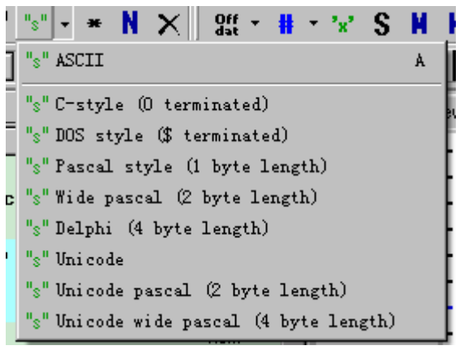
3. 字符串的操作

字符串是一个比较重要的内容， 所以这里单独列出来。



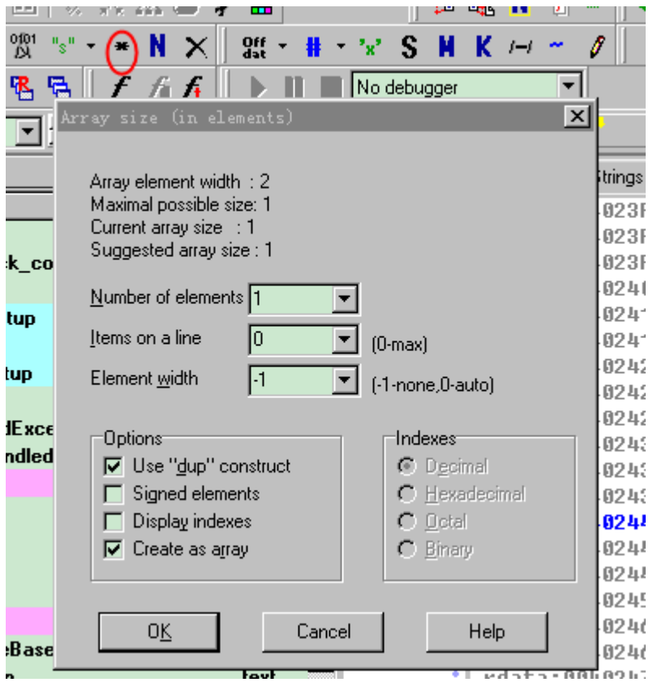
这样就能显示字符串。

不同的编程语言值不同的格式，我们在这里选择我们的字符串格式。



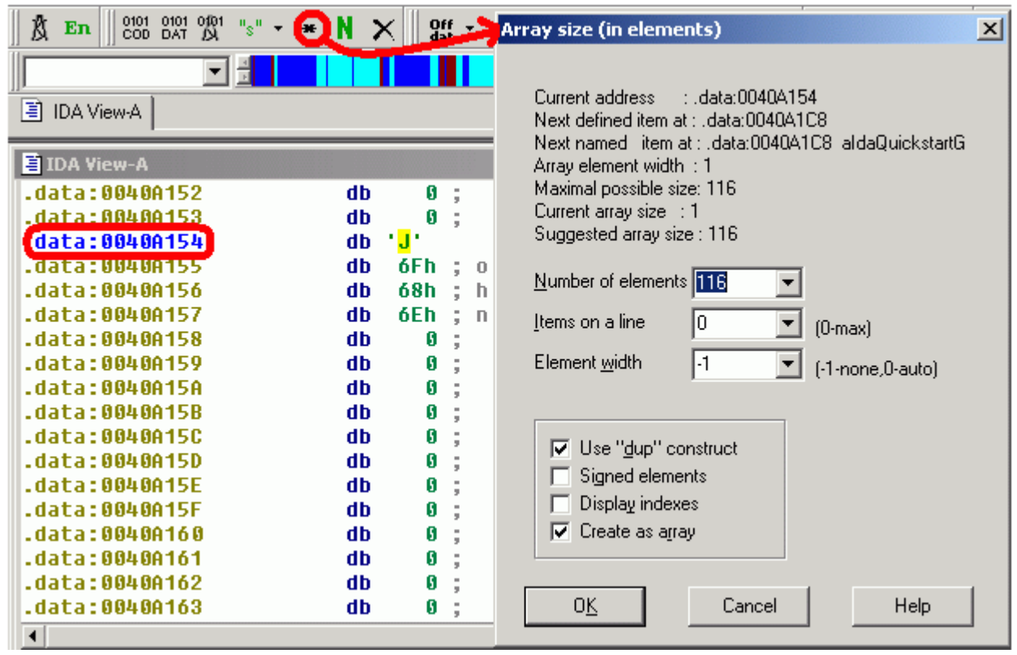
4. 数组

用：



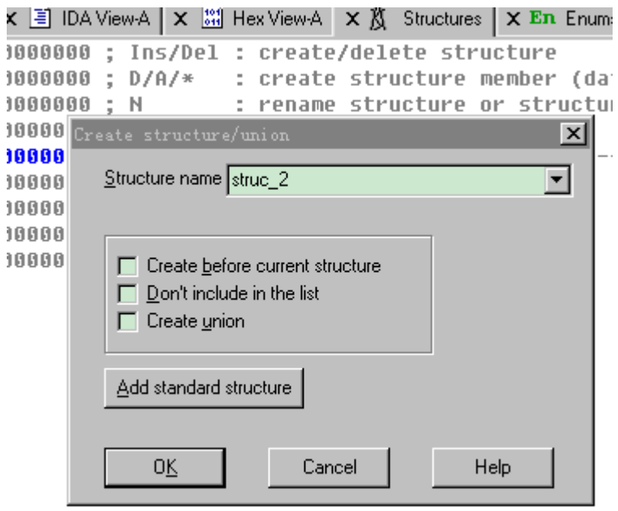
来 make array.

然后：



5. 结构体

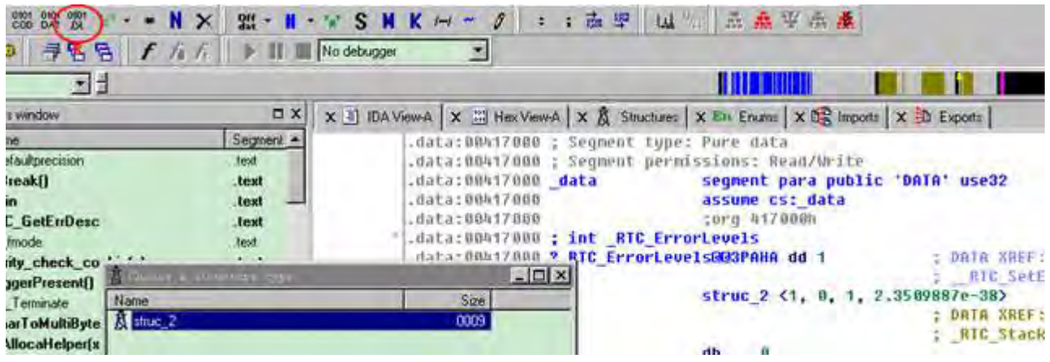
选择创建一个结构体：



按下 D 键增加一个成员变量， N 重命名，这样我们就建立完成了一个结构体：

```
00000000 struc_2         struc ; (sizeof=0x9)
00000000 stc_a           dw ?
00000002 stc_b           dw ?
00000004 stc_c           db ?
00000005 stc_d           dd ?                    ; seg
00000009 struc_2         ends
00000009
```

定义了自己的结构体之后可以这么使用：



http://www.cnblogs.com/sld666666/archive/2011/04/08/2009964.html

IDA 配置显示机器码

定制 ida pro
ida.cfg
OPCODE_BYTES = 6 // don't display instruction/data bytes
反汇编文本模式显示

GRAPH_OPCODE_BYTES = 6 // don't display instruction/data bytes
反汇编流程模式显示
如果不想显示就赋值为 0

撤销 IDA 中的字符串地址转换

```
/*
*    作者：0x 明天去要饭
*    微博：http://t.qq.com/kgdiwss
*    原文链接：http://www.sanwho.com/?p=82
*    转载请注明出处
*/
```

IDA 中，当在字符串上右键再点击左边的“=”或“（”时，IDA 会自动把字符串的地址转换成计算后的，如图：

```
.text:00000CE0
.text:00000CE0                  EXPORT Java_com_yaofans_javacallso_MainActivity_Hello
.text:00000CE0 Java_com_yaofans_javacallso_MainActivity_Hello
.text:00000CE0                  STMFD   SP!, {R4-R8,LR}
.text:00000CE4                  LDR     R6, =(__stack_chk_guard_ptr - 0xCF8)
.text:00000CE8                  LDR     R5, =(aLibplasma - 0xD04)
.text:00000CEC                  MOV     R8, R2
.text:00000CF0                  LDR     R6, [PC,R6]
.text:00000CF4                  LDR     R2, =(aYaofansJava_co - 0xD14)
.text:00000CF8                  LDR     R3, [R6]
.text:00000CFC                  ADD     R5, PC, R5      ; "libplasma"
.text:00000D00                  SUB     SP, SP, #0x20
.text:00000D04                  MOV     R4, R0
.text:00000D08                  MOV     R1, R5
.text:00000D0C                  ADD     R2, PC, R2      ; "[yaofans] Java_com_yaofans_javacallso_M"...
.text:00000D10                  MOV     R0, #4
.text:00000D14                  STR     R3, [SP,#0x1C]
.text:00000D18                  BL      __android_log_print
```

转换前

```
.text:00000CE0                 EXPORT Java_com_yaofans_javacallso_MainActivity_Hello
.text:00000CE0 Java_com_yaofans_javacallso_MainActivity_Hello
.text:00000CE0                 STMFD   SP!, {R4-R8,LR}
.text:00000CE4                 LDR     R6, =(__stack_chk_guard_ptr - 0xCF8)
.text:00000CE8                 LDR     R5, =(aLibplasma - 0xD04)
.text:00000CEC                 MOV     R8, R2
.text:00000CF0                 LDR     R6, [PC,R6]
.text:00000CF4                 LDR     R2, =0x168C
.text:00000CF8                 LDR     R3, [R6]
.text:00000CFC                 ADD     R5, PC, R5      ; "libplasma"
.text:00000D00                 SUB     SP, SP, #0x20
.text:00000D04                 MOV     R4, R0
.text:00000D08                 MOV     R1, R5
.text:00000D0C                 ADD     R2, PC, R2      ; "[yaoFans] Java_com_yaofans_javacallso_M"...
```

转换后

撤销转换的方法为：

点击转换后的数字，比如这里是 0x168C，右键选择“撤销转换”或按快捷键“U”，此时反汇编代码会变成混乱格式，如下：

```
.text:00000CE0                 EXPORT Java_com_yaofans_javacallso_MainActivity_Hello
.text:00000CE0 Java_com_yaofans_javacallso_MainActivity_Hello
.text:00000CE0                 STMFD   SP!, {R4-R8,LR}
.text:00000CE4                 LDR     R6, =(__stack_chk_guard_ptr - 0xCF8)
.text:00000CE8                 LDR     R5, =(aLibplasma - 0xD04)
.text:00000CEC                 MOV     R8, R2
.text:00000CF0                 LDR     R6, [PC,R6]
.text:00000CF0 ; ---------------------------------------------------------------------------
.text:00000CF4                 DCB 0xF4 ; ô
.text:00000CF5                 DCB 0x20
.text:00000CF6                 DCB 0x9F ; Ÿ
.text:00000CF7                 DCB 0xE5 ; å
.text:00000CF8                 DCB    0
.text:00000CF9                 DCB 0x30 ; 0
.text:00000CFA                 DCB 0x96 ; –
.text:00000CFB                 DCB 0xE5 ; å
.text:00000CFC                 DCB    5
.text:00000CFD                 DCB 0x50 ; P
.text:00000CFE                 DCB 0x8F ; Ï
.text:00000CFF                 DCB 0xE0 ; à
.text:00000D00                 DCB 0x20
```

然后选中乱码的最上面一行（这里是**.text:00000CF4**），右键选择“转换为代码”或按快捷键“C”即可恢复正常。但是这样恢复后，原来自定义的注释就会丢失。

IDA 的导航条

IDA 是一个功能非常强大的反汇编工具，工具提供的工具也很多。只有在多多使用之后才会发现它的精细之处...

导航条：

也许你之前一直忽视它的存在，有一天你会发现称它的“导航”并非浪得虚名~

-蓝色：.text section

深蓝：用户自己写的函数编译后的代码区

浅蓝：编译器自己添加的函数，像启动函数，异常函数等（我自己猜的，不一定百分百正确）

-粉红色：.idata section

有关输入表的一些数据信息

-军绿色：.rdata section

纯数据，只读

-灰色：为了段对齐而留下的空隙

-黑色：禁区，不存在任何数据

http://blog.csdn.net/chence19871/article/details/7716527

一 将 ARM 指令转换成 thumb 指令

在使用 IDA 静态分析的时候，我们可能会发现明明是 thumb 指令，IDA 却当作了 ARM 指令来处理，或者相反情况。那么这就需要我们手动进行修复。修复步骤如下：

①鼠标定位到错误指令处

②快捷键 alt + g，在打开的 segment register value 窗口中选择 T，然后将值改为 0x1（ARM 变 thumb）或者 0x0（thumb 变 ARM）。

注意：目前值知道单条指令的修复，不知道多条指令如何同时修复，待以后补充~

http://www.cnblogs.com/wanyuanchun/articles/3937213.html

IDA 修改程序反汇编代码基址

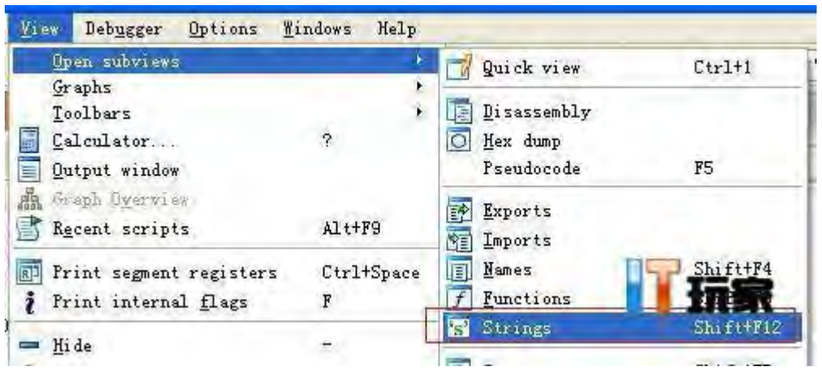
在使用 IDA 对二进制文件进行逆向分析的时候，有时候如果能把特定的代码段的基址修改为指定值，将使得代码更好的理解。
在 IDA 中进行这样的操作的方法有如下几种：

- Manual load 并指定基址
- Edit->Segements->Rebase program

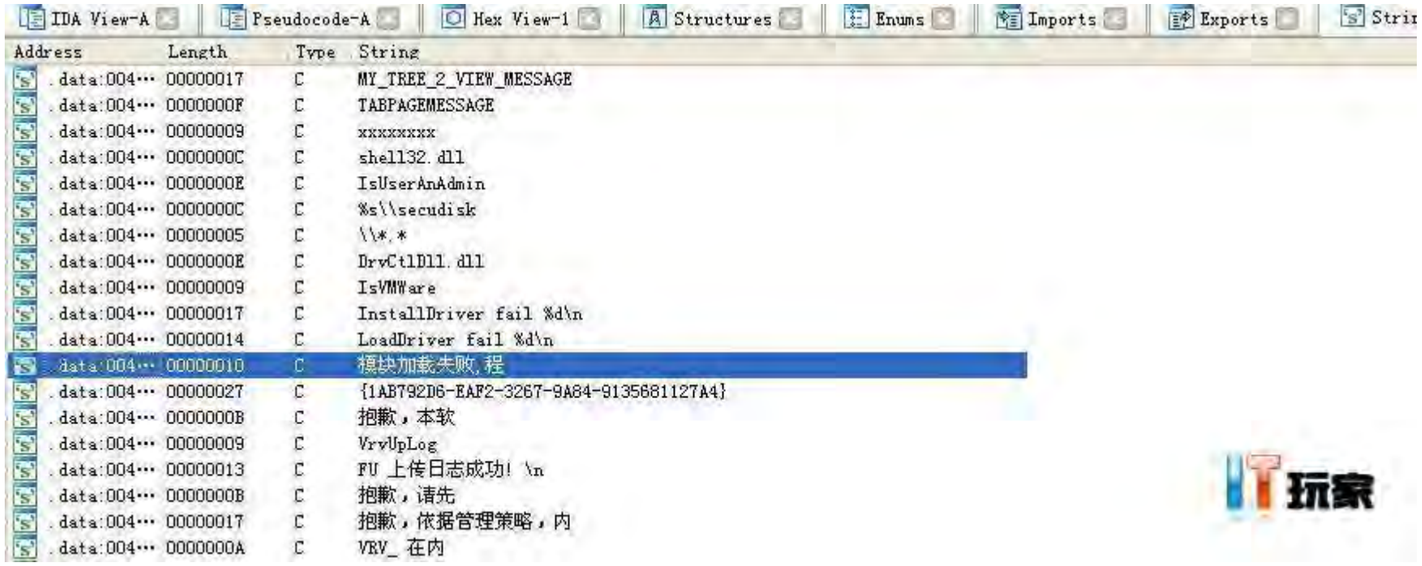
参考资料：http://www.debugman.com/thread/5204/1/1

IDA 初学者笔记之字符串分析

程序中往往包含很多字符串资源，这些资源存在于 PE 文件的 rdata 段，使用 IDA 反编译后，可以查找到这些字符串，
逆向破解程序通常需要一个突破点，而这个突破点，往往就是一个 Messagebox，因为这个 Messagebox 可以很直观的
让我们知道当前位置的代码负责哪些功能，而同时 Messagebox 可以提供一个字符串让我们来查找定位。
首先，打开 IDA,主菜单 View-Open subviews-strings:



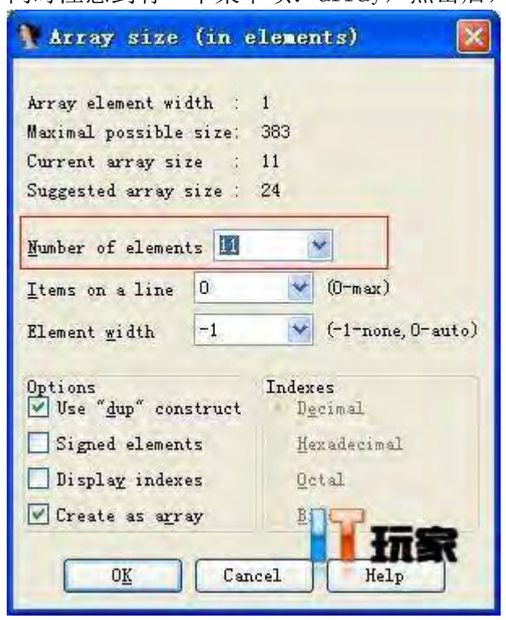
这时出来一个 strings 窗口，如下图所示：



这时候窗口里显示的是程序中的字符串，但很遗憾，从图中可以看到大部分字符串只有一部分，难道 IDA 没法识别完整的字符串吗？
当然不是，双击图中的一项字符串，会自动调整到 IDE View 窗口，部分内容如下图所示：

助记符是 IDA 自动生成的，可以修改，右键单击该助记符，选择菜单 rename 即可。

同时注意到有一个菜单项：array，点击后，出现亮点了：



不管中文该怎么翻译，元素数量、数组长度都行，对于我们选择的字符串来说，这里的作用就是控制字符串长度，且 11 与上面的注释中 aZGmI[11]相呼应，直接修改，对照前面图中 IDA View 的字符串数据，直接修改到 00 的位置，因为 00 是字符串结束的标志。

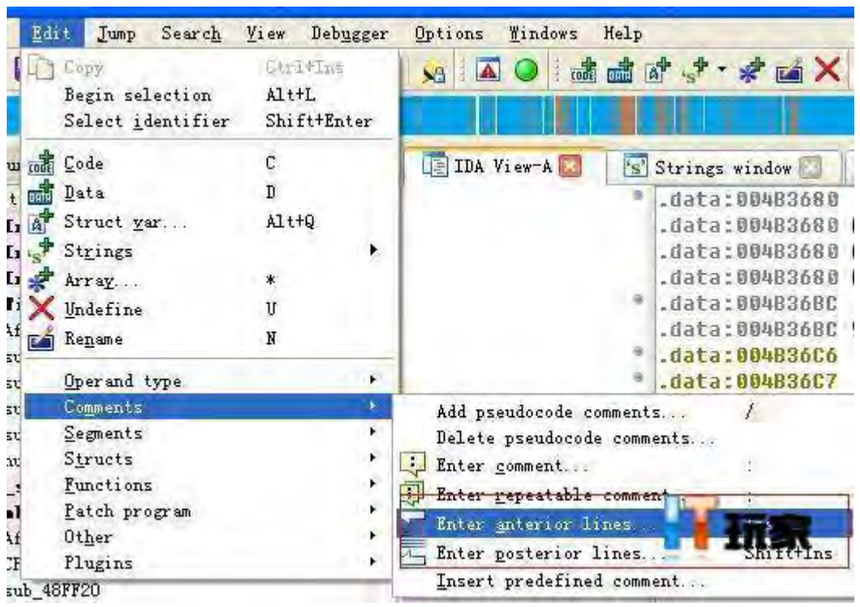
这时可以看到完整通顺的字符串句子出来了，这里再在 aZGmI 上单击鼠标左键选中，再打开右键菜单-rename，修改一个比较有意义的助记符，也就是字符串名称：

关于那句默认注释，若要修改，目前知道 2 个方法，

1）跳转到代码段中的引用处，按 F5，在 C 代码中右键单击该助记符名称，选择，Set Item Type，在新窗口直接修改即可，

2）直接点击在 IDA View 的那句默认注释，快捷键 Y，即可。

若实在想注释，可以添加新的注释，用 Insert 快捷键，或者在主菜单 edit-comments 中开启，删除的时候没有指定的按钮，而是需要再开启注释窗口，清空原来的内容，再确定保存即可：



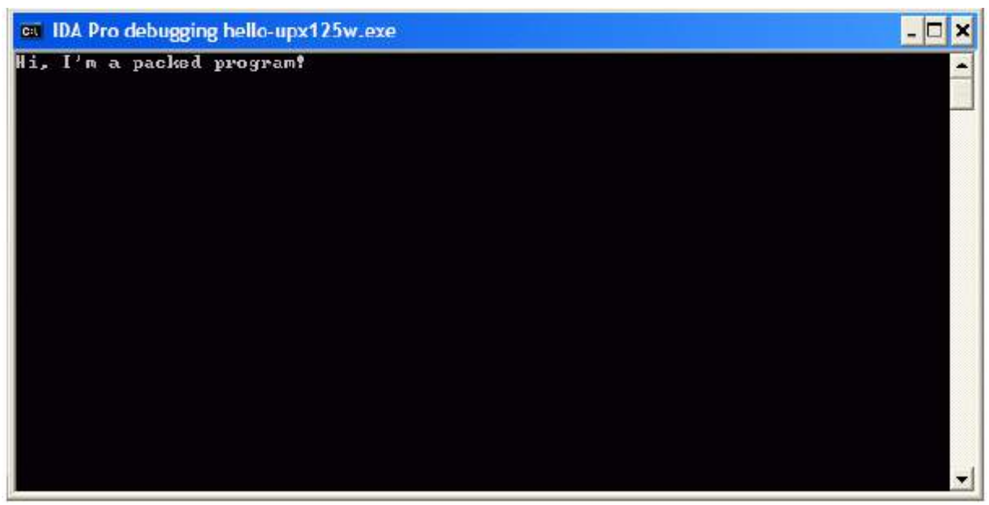
http://www.v5pc.com/2474.html

使用 IDA 的通用解压插件

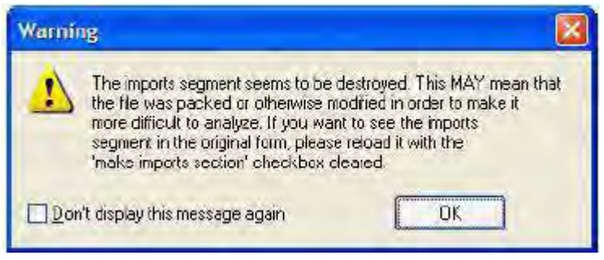
从 4.9 版开始，IDA 就集成了 Universal PE Unpacker 通用解压插件，其源代码可以在 IDA Pro SDK 中找到。这个小教程将会演示如何使用这个插件，并简单介绍其内部工作原理

一个压缩的应用程序

下面是当我们执行这个可执行程序的运行结果：



非常简单的程序，但是，如果我们使用 IDA 来打开它，会出现下面的警告提示：



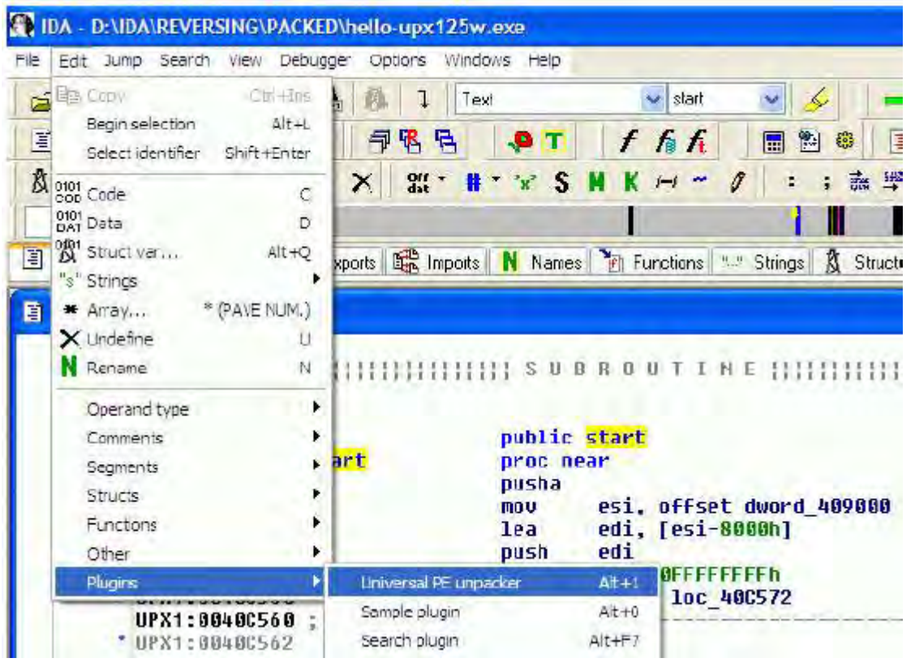
IDA 检测到异常的输入段，并提示我们文件可能被压缩了…

如果我们看一下它的输入表窗口，我们就会发现：

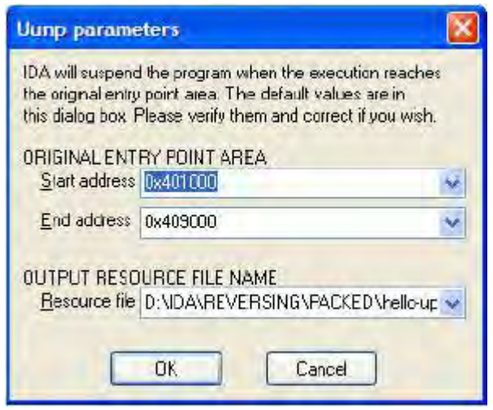
我们的程序只导入了 *kerne132.dll* 中的 3 个函数。我们可以看到在加压缩壳的程序种常用到的 2 个动态链接库函数 *LoadLibraryA* 和 *GetProcAddress*，它们通常用于恢复程序的输入表。

使用通用 PE Universal Unpacker

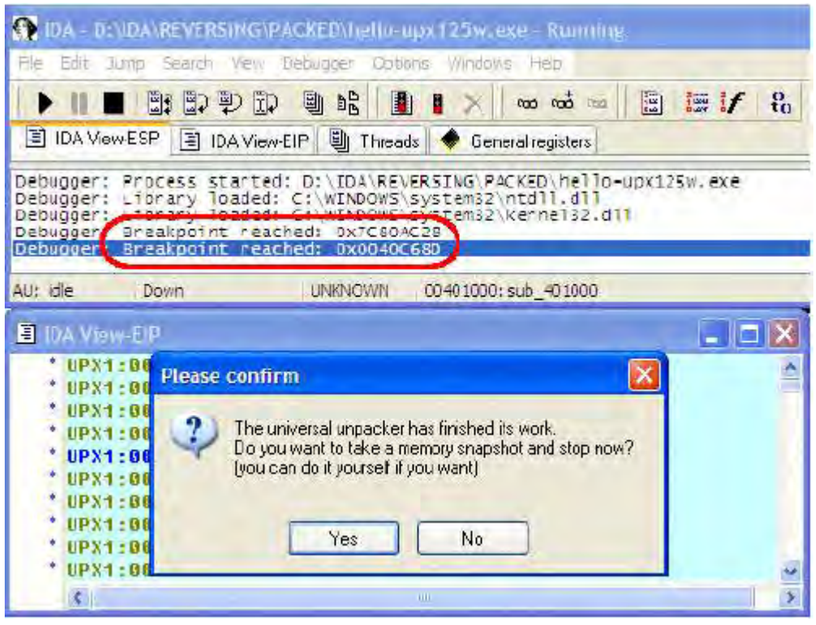
在插件的子菜单种选择 *UniversalPE unpacker*，开始解压：



你会看到插件的选项对话框：

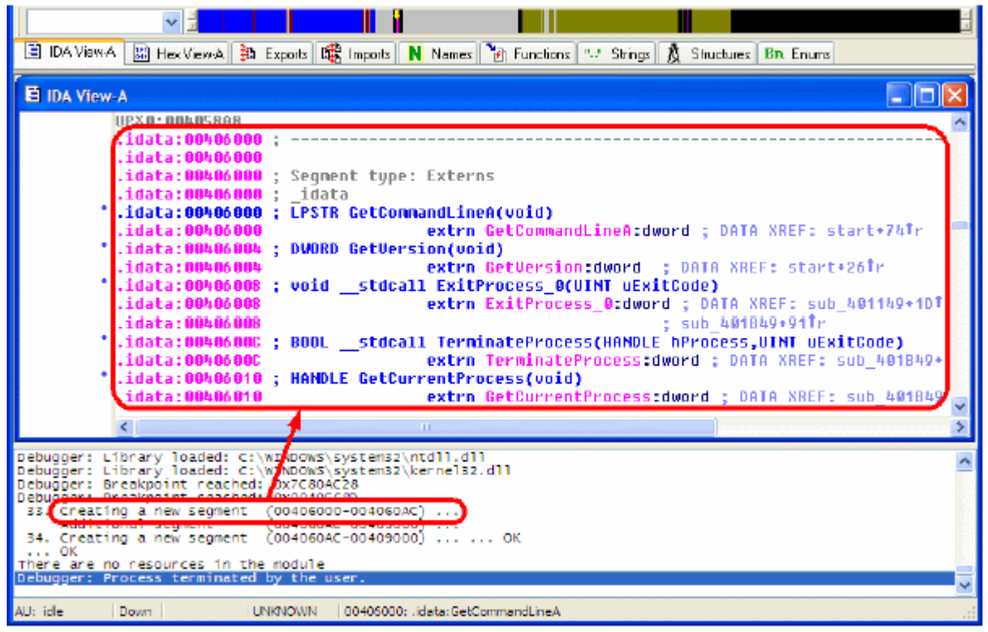


在这个对话框中，我们可以配置一个的地址范围，当程序运行到这个区域（原始的程序入口 点区域），它会挂起程序的执行。你也可以指定一个文件用于保存解压的资源。按过确定按钮后，插件开始运行，它启动了我们的程序，并自己解压，直到它运行到 我们设定的地址范围内。这时解压已经结束，依照对话框中的提示，我们保存了当前的内存快照。

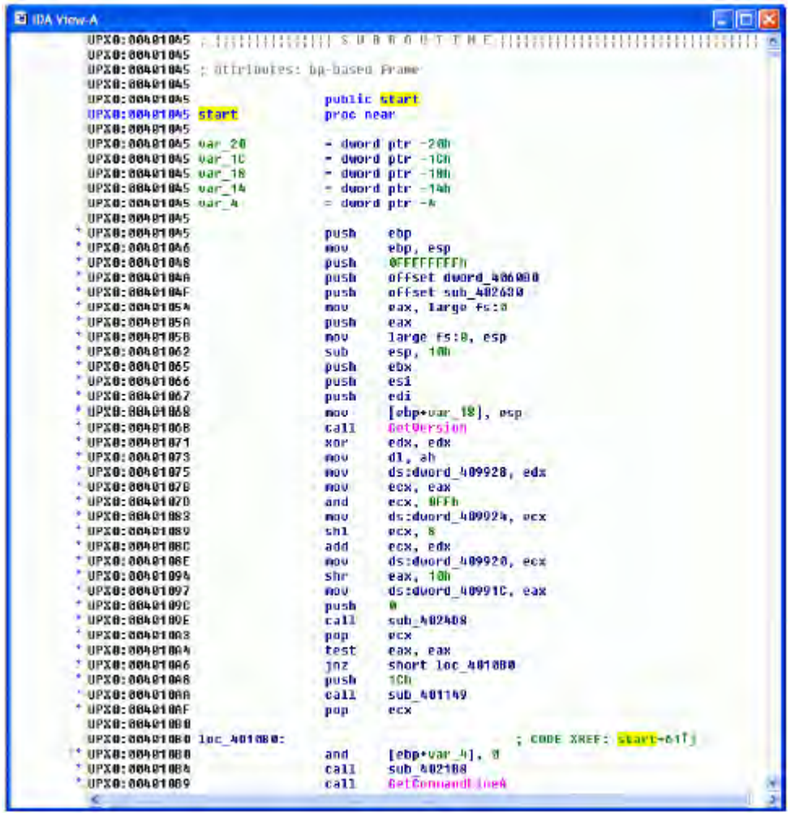


你会注意到，我们遇到了两个断点，这个我们最后再去讨论它。

为了重新构造原始的程序输入表，插件创建了一个新段：



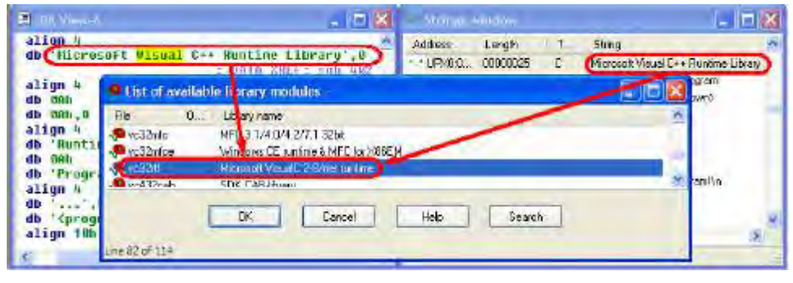
解压完成后，我们在 *start()*函数处可以看到更多我们熟悉的代码结构



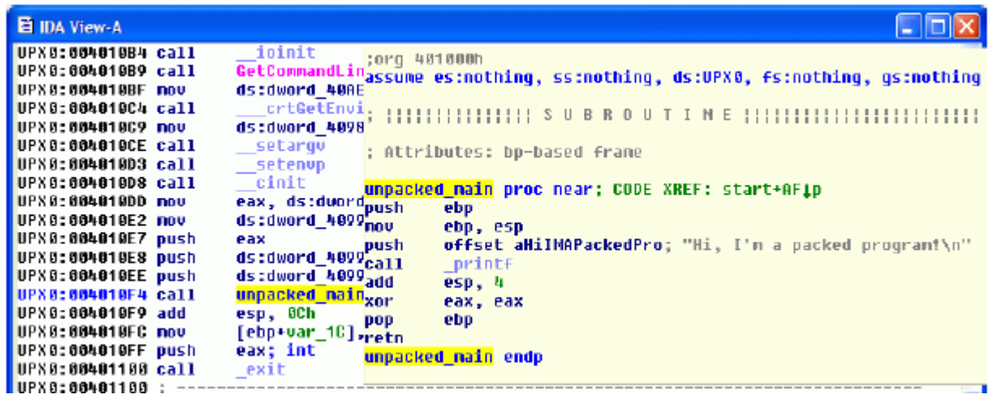
我们还可以进一步改进，让这个反汇编结果尽可能的更完美一些！

使用签名

如果我们查看解压后的程序的字符串窗口，我们可以看出我们的程序是使用 VirsualC++编译的，我们可以应用相应的 FLIRT(快速库识别恢复技术)的库签名。



应用签名库后，最终的反汇编结果如下：



插件分析

下面我们仔细研究一下这个插件，看它是如何使用 SDK 的调试 API 来完成这些工作的。

主要的操作就是启动这个进程，然后根据调试器捕捉到的一些事件，进行相应的处理，直到我们确认程序已经完全被解压。我们先设定一个句柄用于接收调试器的事件，并启动这个程序直到它到达入口点。

```
if (!hook_to_notification_point(HT_DBG, callback, NULL) )
{
warning("Could not hook tonotification point\n");
return;
}
// Let's start the debugger
if ( !run_to(inf.beginEA) )
{
warning("Sorry, could not startthe process");
unhook_from_notification_point(HT_DBG,callback, NULL);
}
```

事件将会被送到我们声明的句柄，定义如下：

```
static int idaapi callback(void * /*user_data*/,

int notification_code,

va_list va)

{

switch ( notification_code )

{

case dbg_process_start:

...

case dbg_library_load:

...

case dbg_run_to:

...

case dbg_bpt:

...

case dbg_trace:

...

case dbg_process_exit:

...

...

}

return 0;
```

```
}
```

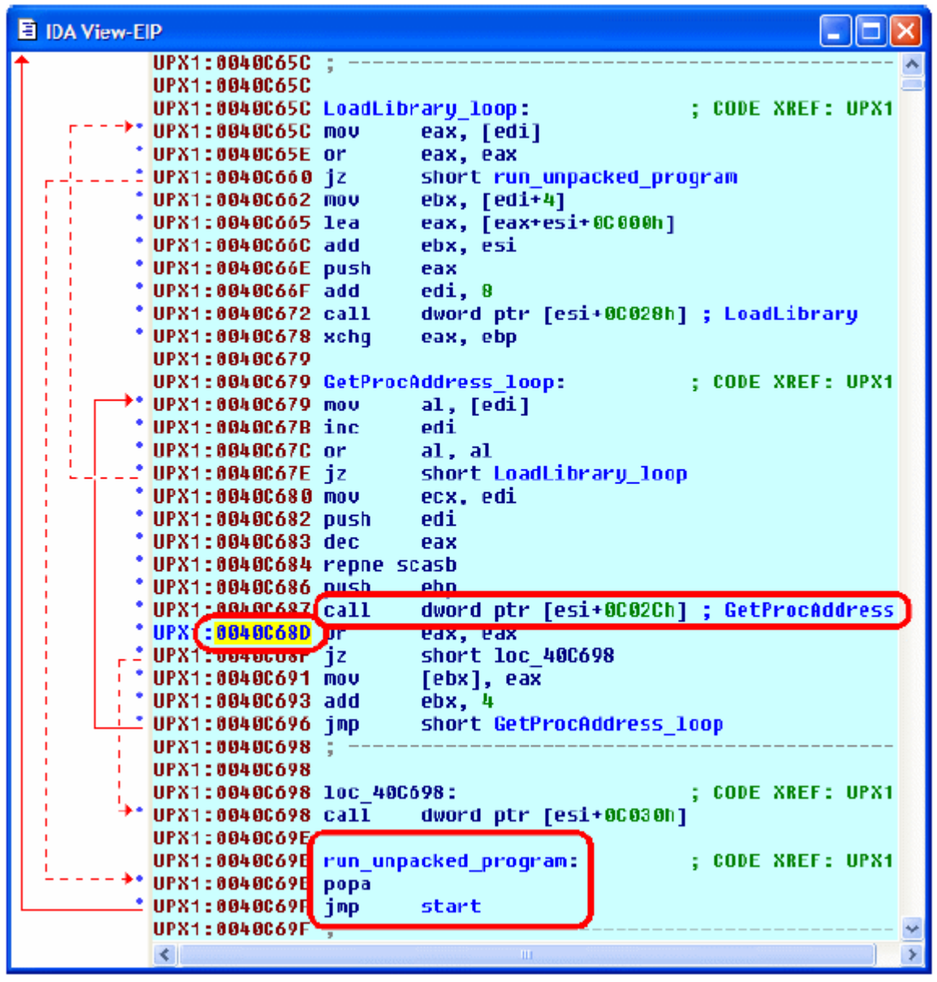
当我们通过一个函数 *run_to()*来开始我们的进程时，我们将会收到一个相应的事件 *dbg_run_to*，它表示 *run_to()*命令被正确的执行。现在我们来到压缩文件的入口点，在 *GetProcAddress()*这个函数上设定一个断点，（假定这个解压代码在重构原始输入表之前结束）：

```
case dbg_run_to: // Parameters: thread_id_t tid

dbg->stopped_at_debug_event(true);

gpa = get_name_ea(BADADDR, "kernel32_GetProcAddress");

...

else if( !add_bpt(gpa) )

{

bring_debugger_to_front();

warning("Sorry, can not set bptto kernel32.GetProcAddress");

goto FORCE_STOP;

}

else

{

++stage;

set_wait_box("Waiting for a call toGetProcAddress()");

}

continue_process();

break;
```

当程序运行到 *GetProcAddress()*断点，我们收到一个 *dbg_bpt* 事件，我们可以从堆栈中提取出这个地址，然后删除这个断点，并在返回地址设定第二个断点，以便能在 *GetProcAddress()*函数返回时立刻中止。

```
case dbg_bpt: // A user defined breakpointwas reached.
// Parameters: thread_id_t tid
// ea_t breakpoint_ea
{
/*tid_t tid =*/ va_arg(va, tid_t);
ea_t ea = va_arg(va, ea_t);
...
if ( ea == gpa )
{
regval_t rv;
if ( get_reg_val("esp", &rv) )
{
ea_t esp = rv.ival;
invalidate_dbgmem_contents(esp,1024);
ea_t ret = get_long(esp);
...
if ( !del_bpt(gpa) ||!add_bpt(ret) )
error("Can not modifybreakpoint");
```

还记得我们之前在运行解压插件时碰到的两个断点吗？中断在 *0x7C80AC28* 和 *0x00040C68D*，第一个是我们的 *GetProcAddress()*断点,而第而个是在返回地址上的断点。在下面的反汇编窗口中，你可以看到进入 *GetProcAddress()* 断点的那个调用指令，现在我们要继续运行直到解压程序恢复了程序寄存器的原始内容，然后跳到解压后的程序的真实入口点：



我们在第二个断点之后开始单步跟踪，直到指令执行到我们之前设定的地址范围。

```
del_bpt(ea);
if ( !is_library_entry(ea) )
{
deb(IDA_DEBUG_PLUGIN, "%a: reached unpackercode, switching to trace mode\n",
ea);
```

```
enable_step_trace(true);

...

set_wait_box("Waiting for theunpacker to finish");

}

else

{

warning("%a: bpt in librarycode", ea);// howcan it be?

add_bpt(gpa);

}
```

在每一步指令执行时，我们判断其地址是否在我们之前设定的范围之内。如果我们运行到了那个区域，重新分析这些解压后的代码，调整入口点，重建输入表，保存资源然后……最后来一张内存快照！

```
case dbg_trace: // A step occured (oneinstruction was executed). This event

// notification is onlygenerated if step tracing is enabled.

// Parameter: none

...

/*tid_t tid =*/ va_arg(va, tid_t);

ea_t ip = va_arg(va, ea_t);

if ( oep_area.contains(ip) )

{

// stop the trace mode

enable_step_trace(false);
```

```
// reanalyze the unpacked code
set_wait_box("Reanalyzing theunpacked code");
do_unknown_range(oep_area.startEA,oep_area.endEA, false);
auto_make_code(ip);
noUsed(oep_area.startEA,oep_area.endEA);
auto_mark_range(oep_area.startEA,oep_area.endEA, AU_FINAL);
// mark the program's entrypoint
move_entry(ip);
set_wait_box();
...
set_wait_box("Recreating the importtable");
invalidate_dbgmem_config();
...
create_impdir();
set_wait_box("Storing resources to'resource.res'");
if ( resfile[0] != '\0' )
extract_resource(resfile);
set_wait_box();
if ( take_memory_snapshot(true) )
goto FORCE_STOP;
```

这样，就在 IDA 数据库中得到了进程的内存映像，我们可以象通常那样来分析这些解压后的代码了。

现在就去看看那些在 SDK 中的源代码吧！去了解一下这个插件的所有实现细节。

http://blog.csdn.net/eqera/article/details/8237717

使用 IDC 分析加密代码

第一步：

病毒的二进制映像被装载到 IDA 中，从程序的入口点开始

```
loc_0_40:                                       ; CO
                cli
                xor     ax, ax
                mov     ss, ax
                assume ss:nothing
                mov     sp, 7C00h
                sti
                mov     si, 7C50h
                push    cs
                call    near ptr sub_0_E2
;
unk_0_50        db  21h ; !
                db  5Eh ; ^
unk_0_52        db  0Bh ;
                db 0B9h ; ╣
                db 0AEh ; «
```

很明显，在调用函数 call 命令里的地址，没什么意义，但它给了我们一个提示，它是加密代码的所在。

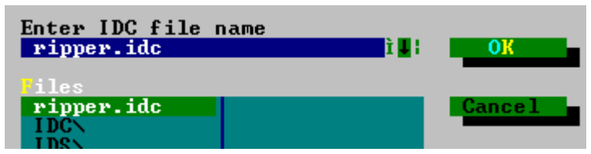
```
sub_0_E2        proc far                ; CODE XREF: seg000:004D↑p
                mov     di, si
                push    cs
                pop     ds
                push    cs
                pop     es
                assume es:seg000

loc_0_E8:                               ; CODE XREF: sub_0_E2+14↓j
                lodsb
                xor     al, 0AAh
                stosb
                push    di
                and     di, 0FFh
                cmp     di, 0DFh ; '▀'
                pop     di
                jnz     loc_0_E8
                xor     ax, ax
                mov     ds, ax
```

第二步：

我们生成一个 IDC 程序来模仿解密代码：

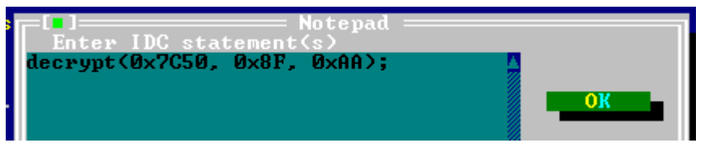
```
static decrypt(from, size, key ) {
 auto i, x;      // we define the variables//定义变量
  for ( i=0; i < size; i=i+1 ) {
   x = Byte(from);   // fetch the byte//取原始代码
    x = (x^key);    // decrypt it//解密
    PatchByte(from,x); // put it back//放回解密代码
    from = from + 1;  // next byte//下一字节
  }
}
```

我们保存它,并使用快捷键 F2 来装载它。



第三步

我们使用 shift+F2 来执行这段 IDC 程序，注意输入相应参数，注意其中的起始点地址。



现在代码已经解密了。

```
loc_0_40:                                       ; CODE X
                cli
                xor     ax, ax
                mov     ss, ax
                assume ss:nothing
                mov     sp, 7C00h
                sti
                mov     si, 7C50h
                push    cs
                call    near ptr sub_0_E2
;
unk_0_50        db  8Bh ; ï
                db 0F4h ; ¶
unk_0_52        db 0A1h ; í
                db  13h ;
                db    4 ;
                db  48h ; H
                db  48h ; H
                db  50h ; P
                db 0B1h ; ▒
                db    6 ;
                db 0D3h ; Ë
                db 0E0h ; ó
                db  8Eh ; Ä
                db 0C0h ; └
                db  33h ; 3
                db 0FFh ;
                db 0B9h ; ╣
                db    0 ;
                db    1 ;
                db 0F3h ; ¾
                db 0A5h ; Ñ
```

第四步

在解密代码的起始未知，我们使用快捷键 C，将其翻译为汇编代码。

```
loc_0_40:                                       ; COD
                cli
                xor     ax, ax
                mov     ss, ax
                assume ss:nothing
                mov     sp, 7C00h
                sti
                mov     si, 7C50h
                push    cs
                call    near ptr sub_0_E2

loc_0_50:
                mov     si, sp

unk_0_52:
                mov     ax, ds:413h
                dec     ax
                dec     ax
                push    ax
                mov     cl, 6
                shl     ax, cl
                mov     es, ax
                xor     di, di
                mov     cx, 100h
                repe movsw
                mov     ax, 79h
                push    ds
                push    es
                push    ax
                retf
; ---------------------------------------------------------------------------
aFuckEmUp       db 'FUCK ',27h,'EM UP !'
```

http://blog.csdn.net/eqera/article/details/8237790

IDA6.1 按照说明打不开旧的 IDB 文件

找到解决方法 目前还没发现有什么 BUG
IDA.WLL

```
.text:100043AE              test    al, al
.text:100043B0              jnz     short loc_100043BC
.text:100043B2              push    offset aDatabaseIsCo_0 ; "Database is corrupt"
.text:100043B7              call    sub_100010F0
```

修改为

```
.text:100043AE              xor     eax, eax
.text:100043B0              jmp     short loc_100043BC
.text:100043B2 ; ---------------------------------------------------------------------------
.text:100043B2              push    offset aDatabaseIsCo_0 ; "Database is corrupt"
.text:100043B7              call    sub_100010F0
```

http://bbs.pediy.com/showthread.php?p=981400#poststop

IDA 插件 idbtopat.plw 的用途

当用 IDA 分析某个 exe 文件，有很多函数都相同，在前版本已经分析过并加有很多注释，exe 版本更新后又要重新分析。请问各位是怎么解决这个问题的？

1.从上一个版本程序的 idb 制作出 pat 文件。
2.做 sig 文件。

如何从 dll 得到 IDA 的 sig 文件?

如果只有 dll,则反汇编该 dll,然后用 IDB2PAT 插件生成.PAT 文件然后 sigmake 生成.sig 文件.

http://blog.csdn.net/zhangmiaoping23/article/details/16941603

IDAPython

IDAPython 的妙用

看过 Python 灰帽子的密友都知道 IDAPython 在逆向的场景中常常能够发挥出巨大的威力，笔者偶然一次在逆向的过程中使用了它，下面就来总结一下使用的方法和一些注意事项：

环境：IDA 6.1 ，android 2.3 AVD。

我这个 IDAPython 的版本是：

Python 2.6.6 (r266:84297, Aug 24 2010, 18:46:32) [MSC v.1500 32 bit (Intel)]

IDAPython v1.5.2 final (serial 0) (c) The IDAPython Team idapython@googlegroups.com

---

下载这个包

安装方法：

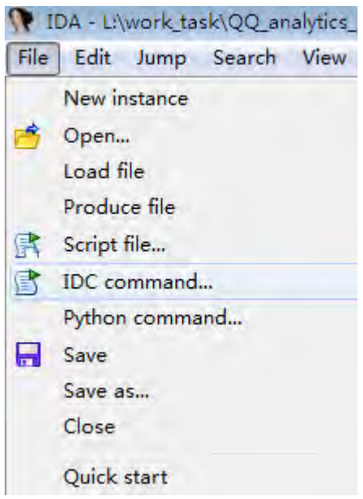
1）解压压缩包

2）将 python.plw，python.p64 复制到 IDA 的插件目录 plugins 下。

3）将 python 的目录拷贝到 IDA 的主目录中。

如果安装成功，在启动 IDA 后，输出面板会有如上的输出。

安装好了以后，我的只有一个 Python command... 命令项，并没有 Python file...项



应用：

```
from idaapi import *
from idc import *

count = 0

class DumpHook(DBG_Hooks):
    def dbg_bpt (self,tid,ea):
        global count
        count += 1;
        print "[*] Hit: 0x%08x the %d time\n" % (ea, count)
        data = "\xBE\x91\x0A\xF3\x9A\x26\xA4\xA9\x92\xC6\xFD\x01\xA1\x43\xED\x19"
        dbg_write_memory(GetRegValue("r7"), data)
        return 1

try:
    if debugger:
        print("Removing previous hook ...")
        debugger.unhook()
except:
```

```
    pass

AddBpt (0x8050a42e)
SetBptAttr(0x8050a42e, BPTATTR_FLAGS, BPT_ENABLED|BPT_TRACE)
print "[*] set hook OK...\n"

debugger = DumpHook()
debugger.hook()
```

关于里面的函数，可以去查看 IDA 主目录下 python 文件夹里的 idc.py 和 idaapi.py，里面有对各个函数的详细说明。

注意：python 脚本从 python command...里输入，而不是作为一个文件从 script file...里输入

http://blog.csdn.net/chence19871/article/details/20446119

浅谈 IDA 脚本在漏洞挖掘中的应用

博文作者：dragonltx[TSRC]

发布日期：2012-08-30

阅读次数：9928

博文内容：

[目录]

[1] - 前言

IDA 毫无疑问是逆向领域里的一大神器，无所不能。有人的地方就有江湖，有江湖的地方就有武器。那么，在逆向这个江湖中，IDC 和 IDAPython 就好比倚天剑和屠龙刀，威力无比。

在漏洞挖掘领域，IDA 同样能够大展身手。OpenRCE 上提供的 BugScam 脚本正是 IDC 应用的最好诠释，著名的 Paimei 也是应用了 IDApython。

接下来，笔者将以自己的经验来分享下两把利器“倚天剑”和“屠龙刀”的应用。

[2] - IDC 和 IDAPython 简介

事实上，没有哪一个应用程序能够满足每名用户的一切需求。应用开发者面临两种选择：要么满足用户提出的无止境的功能要求，要么提供一种方法，供用户解决问题。IDA 采用了后一种方法，它集成了一个脚本引擎，让用户从编程角度对 IDA 的操作进行全面控制。

IDA 脚本语言可看成是一种查询语言，它能够以编程方式访问 IDA 数据库的内容。IDA 的脚本语言叫做 IDC，之所以取这个名称，可能是因为它的语法与 C 语言的语法非常相似。

得 益于 IDA Pro 极为开放的构架，Gergely Erdelyi 和 Ero Carrera 在 2004 年发布了 IDAPython--一款 IDA Pro 的插件。通过这款插件，逆向工程师能够以 Python 脚本的形式访问 IDC 脚本引擎核心、完整的 IDA 插件 API，以及所有与 Python 捆绑在一 起的常见模块。IDAPython 无论是在商业产品中（例如 Zynamics 的 BinNavi），还是在一些开源项目中（例如 Paimei 和 PyEmu） 均有所应用。

[3]- 倚天剑：IDC 应用

如果各位读者对这篇文章感兴趣，应该都对 IDC 有了解。不过不了解也没关系，那就请先参考下相关资料[1]，里面有详细的 IDC 语言介绍，这里就不再进行介绍。

当我们想通过自动化运行 IDA 获取一些对漏洞挖掘有用的信息，而不是手工运行 IDA，该怎么做？

IDA 提供了如下两个函数，可以帮助我们实现自动化。

Wait

```
// Wait for the end of autoanalysis
// This function will suspend execution of IDC program
// till the autoanalysis queue is empty.

void    Wait            ();                     // Process all entries in the
                                                // autoanalysis queue
```

Exit

```
// Stop execution of IDC program, close the database and exit to OS
//      code - code to exit with.

void    Exit            (long code);            // Exit to OS
```


在 IDA 启动后，IDA 会执行一些自动分析操作。Wait 函数会等待，直到这些自动分析结束。该函数会挂起我们的 IDC 脚本，直到自动分析队列为空。当自动 分析队列为空时，就开始执行我们的 IDC 脚本。Exit 函数会结束 IDC 函数的执行，并将 idb 关闭，然后结束 IDA 主进程，相当 nice 的功能。

有了这两个函数后，还不够，革命尚未成功。IDA 还提供了丰富的命令行参数，帮助我们实现自动化。

```
-A      autonomous mode. IDA will not display dialog boxes.
        Designed to be used together with -S switch.

-S###   Execute a script file when the database is opened.
        The script file extension is used to determine which extlang
        will run the script.

        It is possible to pass command line arguments after the script name.

        For example: -S"myscript.idc argument1 \"argument 2\" argument3"

        The passed parameters are stored in the "ARGV" global IDC variable.
        Use "ARGV.count" to determine the number of arguments.
        The first argument "ARGV[0]" contains the script name.
```

“-A” 参数是自动模式，IDA 将不会显示对话框，是和“-S”参数一起使用。“-S”参数指定执行那个 IDC 脚本，后面可以跟 IDC 脚本的参数。参数会放在 ARGV 这个全局变量里，其中 ARGV[0]，存放的是 IDC 脚本名。IDA 还提供了-c 参数，用来反汇编一个文件[3]。

现在实现自动化的各个因素都凑齐了，“万事具备，只欠东风”，接下来是一个自动导出一个文件中的所有函数名、起始地址、结束地址的 IDC 脚本。

```
#include <idc.idc>

static main()
{

  auto addr, end, args, locals, frame, firstArg, name, ret ,handle, path, index, filename, outputfilename ,segaddr;

  addr = 0;

  Wait();    //等待直到 IDA 自动分析完成

  segaddr = MinEA();

  Message("Base:%x\n",segaddr);

  handle = fopen(ARGV[1],"w");

  for( addr = NextFunction(addr); addr != BADADDR; addr = NextFunction(addr))

  {
    name = Name(addr);

    end  = GetFunctionAttr(addr, FUNCATTR_END);
    if(substr(name,0,4) == "sub_")
    continue;

    Message("Function:%s, starts at %x,ends at %x\n", name, addr-segaddr, end-segaddr);

    fprintf(handle,"Function:%s, starts at %x,ends at %x\n", name, addr-segaddr, end-segaddr);
```

```
  }

  fclose(handle);

  Exit(0);
}
```

当我们以这样的命令行 idaq -c -A -S"dumpfunc.idc E:\func.txt" E:\test.dll 运行 IDA，结果就会自动保存在 E 盘的 func.txt 中，相当惬意吧！心动了吧，心动了就赶快行动吧！你可以尽情发挥自己的才 华，向 IDA 获取你想要的东西。

[4] - 屠龙刀：IDAPython 应用

在 《Python 灰帽子》[2]第十章中，Justin 提供了一种自动化获取驱动程序 IO 控制码的方法，不过该脚本是基于 Immunity Debugger 的库。笔者用 Immunity Debugger 加载驱动文件，发现加载失败。后来一想，既然是基于静态分析的方法，何必用 Immunity Debugger，IDA 才是静态分析领域的王者。下面探讨用 IDAPython 来实现自动获取驱动程序 IO 控制码的初级版程序。

[4.1] - 获取驱动程序设备名

通过 FindText 这个函数来查找包含“\\Device\\”这个函数的偏移地址，然后通过 GetString 来获取字符串，如果获取的字符串为空，继续查找。

```
def getDeviceName():
    """
    Get Device Name from a driver.
    @rtype:   void
    @returns: void
    """
```

```
ea = 0
while True:
 ea =  FindText(ea, SEARCH_NEXT | SEARCH_REGEX, 0, 0, "\\\\Device\\\\")
 string = GetString(ea, -1, ASCSTR_UNICODE)
 if string is None:
    continue
 else:
    #Message("Find in %x\n" % ea)
    Message("device is %s\n" % string)
   Break
```

[4.2] - 获取驱动分发函数地址

首先用 FindText 查找 mov dword ptr [edx+70h], offset sub_11010 类似这种形式的指令，通过正则匹配查找。找到后，用 GetOperandValue 函数获取第二个操作数的值，即是分发函数的地址。

```
def getDispatchAddress():
    """
    Get Device Dispatch Address from a driver.
    @rtype:   int
    @returns: Dispatch Address
    """
    ea = 0
```

```
    ea =  FindText(ea,  SEARCH_DOWN |SEARCH_NEXT | SEARCH_REGEX, 0, 0, "mov *dword *ptr *\\[[a-zA-Z]* *\\+ *70h\\],[a-zA-Z0-9_ ]*")
    #ea =  FindText(ea, SEARCH_NEXT | SEARCH_REGEX, 0, 0, "test *[a-zA-Z]*, +[a-zA-Z]*")
    #Message("Find in %x\n" % ea)
    if ea == BADADDR:
        Message("Cann't find the Dispatch address")
        address = BADADDR
    else:
        address = GetOperandValue(ea,1)
        Message("Dispatch address is %x\n" % address)
return address
```

[4.3] - 获取函数内所有指令或指令偏移

通过 GetFunctionAttr 获取函数的结束地址，再通过 ItemSize 来获取每条指令的大小，然后循环遍历即可获得这个函数的所有指令的偏移地址。这边先获取所有指令的偏移地址，而不是指令，下面获取 io 控制码会用到。

```
def getFunctionInstructions():
    """
    Get All Instructions from a function.
    Here,Just Get All Instructions Offset,and store them in list
    @rtype:   List
```

```
@returns: List of All Instructions

"""

Instructions = []

DispatchBeginAddress = getDispatchAddress()

if DispatchBeginAddress == BADADDR:

   Message("Cann't find the Function Instructions List")

   return None

DispatchEndAddress = GetFunctionAttr(DispatchBeginAddress,FUNCATTR_END)

i =  DispatchBeginAddress

while True:

  #Instructions.append(GetDisasm(i))

  Instructions.append(i)

  tmp = i + ItemSize(i)

  if tmp < DispatchEndAddress:

     i = i + ItemSize(i)

  else:

     break

address = i

return Instructions
```

[4.4] - 获取驱动程序的所有 IO 控制码

获 取分发函数的所有指令偏移后，倒序查找。如果碰到是 jz 或者是 je 的，且接下来是 cmp 的指令，并且比较操作的寄存器是否一样，一样的话，则把 io 控制码 存储。（这样还是不够准确的，如果碰到其他的 jz 且连着 jmp 的指令，但不是 io 控制码。纯自动分析有时候不能识别）。

```
def getIoctlCode():
    """
    Get All IoctlCodes from a driver.
    @rtype:   List
    @returns: List of All IoctlCodes
    """
    isConditionalJmp             = False
    isFirst                      = True
    BaseRegister                 = None
    OperRegister                 = None
    IoctlCode                    = []
    DispatchFunctionInstructions = []
    DispatchFunctionInstructions = getFunctionInstructions()[::-1]
    if DispatchFunctionInstructions == None:
       Message("Cann't get the IoctlCodes")
       return
    for i in DispatchFunctionInstructions:
```

```
#Message("The instrucion of this function is %x\n" % i)

mnem = GetMnem(i)

if "jz" in mnem or "je" in mnem:

   isConditionalJmp = True

   continue


if "cmp" in mnem and isConditionalJmp and isFirst:

   sisConditionalJmp = False

   BaseRegister  = GetOpnd(i,0)

   IoctlCode.append(GetOperandValue(i,1))

   isFirst = False

   continue


  if "cmp" in mnem and isConditionalJmp and not isFirst:

   isConditionalJmp = False

   OperRegister  =  GetOpnd(i,0)

   if OperRegister == BaseRegister:

      IoctlCode.append(GetOperandValue(i,1))



for i in IoctlCode:
```

```
        Message("The ioctlcode of this driver is %x\n" % i)
```

[4.5]不足与缺陷

上面实现的自动获取 io 控制码的比较简单，有些情况没有考虑到，算是初级版。Switch 反汇编的形式有很多种，上面只是考虑了 cmp 的形式。有兴趣的读者可以继续深入挖掘。上面的 IDAPython 脚本可以在这里（http://bbs.pediy.com/showthread.php?t=153965）获取到，里面还有对函数的解释。

[5] - 结语

本文主要对 IDA 脚本在漏洞挖掘领域应用进行简单的探讨，主要起到抛砖引玉的效果。希望对给位读者有所帮助。如果你有更好的思路，可以跟我探讨。

“思想有多远，就能走多远”。尽情发挥你的奇思妙想，在漏洞挖掘的海洋里尽情畅游吧！

References

[1]IDA 权威指南

[2]Python 灰帽子--黑客与逆向工程师的 Python 编程之道

[3]IDA Pro Documentation

http://security.tencent.com/index.php/blog/msg/4

IDApython & IDAperl

两个 IDA PRO 的插件：IDApython & IDAperl

wooshi@gmail.com

IDA PRO 是目前应用最广的静态反汇编分析工具，功能十分强大，但其自带的 IDC 脚本语言却十分丑陋，写起脚本来非常不方便。有两位大 牛就写了两个工具来方便大家编写 IDC 的脚本，一个是 Gergely Erdelyi 写的 IDApython, 另一个是 RedPlait 写的 IDAperl。总的说来，IDApython 写出来的脚本非常漂亮，也非常简单，但 IDAperl 有一项功能是 IDApython 不具有的，那就是 IDAperl 的脚本支持 debug 功能。下面我们分别简单的介绍一下这两个强大的插件。

IDApython

安装：非常简单，先安装 python2.3,然后在这里下载 IDApython：http://www.d-dome.net /idapython/，把 IDApython 安装包的里的 python 目录拷到 IDA 的目录下，把 plugins/python.plw 拷到 IDA 的 plugins 目录下，再在 IDA 的 plugins/plugins.cfg 文件中添加如下几行（添不添好像问题不大）

Execute_Python_File python Alt-9 0

Execute_Python_Statement python Alt-8 1

Execute_Python_ScriptBox python Alt-7 2

这时候 IDA 的 edit 菜单的 plugins 的子菜单下会出现一个 IDApython 的菜单，你可以通过这里运行你的 python for IDA 的脚本了。

遗憾的是，IDApython 的文档严重缺乏，好在大部分的函数名字跟 IDC 脚本的函数相类似，新加的函数在源程序里有比较详细的注释，不 过就算是这样，也比较让人头疼，没办法，只好自己慢慢摸索了。IDApython 有三个 py 文件，idc.py 中定义的函数与 idc 脚本的函数是对应 的，idaapi.py 中定义了很多 class,使脚本具有了面向对象的特征，idautils.py 中定义了一些高等函数，使相当大部分的 IDC 功能可 以简单的表示出来。

下面以一个例子来对比一下 IDApython 的好处。这个例子搜索一个函数内部的索引，并打印出来，我们先用 idc.py 中定义的函数来一下：

```
from idc import *
```

```
ea = 0x4011f6 （一个函数的开始地址，这里 idapython 好像有点问题，应该用 ChooseFunction，但老是说不对）
funcend = FindFuncEnd(ea)
for ea in range(ea,funcend):
  x = Rfirst0(ea)
  while x <> BADADDR :
    Message( str(hex(x)) + " refers to " + Name(x) + " : " + str(hex(x)) + " ");
    x = Rnext0(ea,x);
Message("End of output. ");
```

跟标准的 IDC 脚本比较一下：

```
#include
```

```
static main(){

auto ea,x,f_end;

ea = ChooseFunction("Select a function to parse:");

f_end = FindFuncEnd(ea);

Message(" *** Code References from " + GetFunctionName(ea) + " : "  + atoa(ea) + " ");

for ( ea ; ea <= f_end; ea = NextAddr(ea) ) {

    x = Rfirst0(ea);
```

```
        while ( x != BADADDR) {
            Message(atoa(ea) + " refers to " + Name(x) + " : " + atoa(x) + " ");
            x = Rnext0(ea,x);
        }
    }
    Message("End of output. ");
}
```

可以看出，idapython 写出来的代码漂亮了不少，而且有个更重要的好处，我们可以使用 python 庞大的函数库，使我们的开发省力不少。

下面我们再用 idaapi.py 中定义的函数来一遍：

```
from idaapi import *


func = choose_func("test",1)
print "begin print refs"
for funcea in range(func.startEA,func.endEA):
    ref = get_first_fcref_from(funcea)
    while ref != BADADDR:
        print "  called from 0x%x(%s)" % (funcea,get_name(BADADDR,ref))

        ref = get_next_fcref_from(funcea, ref)
```

最后用 idautils.py 函数来一次：

```
from idautils import *
func = choose_func("test",1)
print "begin print refs"
for funcea in range(func.startEA,func.endEA):
   reflist = CodeRefsFrom(funcea,0)
   for ref in reflist :

      print "  called from 0x%x(%s)" % (funcea,get_name(BADADDR,ref))
```

恩，好像说服力不够，不过可以看到这个例子还是有些面向对象的特征了。总之，有了 idapython，特别是 idaapi 的面向对象的包装与 idautils 中的常用函数的使用简化，使我们的工作简单化了。好，下面我们介绍一下 idaperl,这个东西难看一些，不过功能也十分强大。

IDAPerl

安装：不是很简单喔，先安装 perl for windows, 再到一个俄文的网站上去下载 IDAperl: http://www.wasm.ru/pub/23/files/perl_src.zip, 这些东西需要有相应的 IDAsdk 才能 编译，我编译好了，放到了 http://www.team509.com/download/tools/security/idaperl.rar，把 pm/下的内容放到 perl 的 lib 里去，把 perl_dbg.plw 和 rp_vc.plw 拷到 IDA4.7 的 plugins 目录下，这样就可以了。这时 候打开 ida4.7 的时候，选择一个 pe 的 windows 程序，就会弹出一个”perl script for

debugger”的对话框，就可以加载你的 perl for debugger 的脚本了，同时 edit 菜单的 plugins 子菜单会有 embedded perl 菜单，这里你可以加载一般的非 debugger 的分析脚本。

使用：由于 IDAPerl 的主要特长在于 debugger 脚本的编写，我们也主要介绍这部分功能，其他一般的功能函数与 IDC 脚本的函数的 名称基本上是一样的，就不多做介绍了。IDAperl 定义了很多程序运行状态改变时的回调(callback)函数,比如说进程启动的时候它定义了 CbProcessStart,你可以写一个函数叫这个名字，同时把你要在进程启动时你要做的工作写到这个函数里。IDAperl 一共定义了如下的回调函 数：

CbProcessStart (unsigned long p_id)--- 进程启动时

CbProcessExit (unsigned long p_id, int exit_code)--- 进程退出时

CbThreadStart (unsigned long t_id)--- 线程启动时，在主线程启动时，不调用

CbThreadExit(unsigned long t_id, int exit_code) ----线程退出时

CbLibraryLoad(struct deb_module *dm) --- modules 加载时

CbLibraryUnload (char *name)--- modules 卸载时

CbBpt(unsigned long t_id, unsigned long addr) --- 遇到断点时

CbException(unsigned long t_id, int code, unsigned long ea, char *info) --- 异常抛出时

IDA 还定义了很多支撑函数来支撑这些功能：

ProcessQty ? 进程的数量

GetProcessInfo (n, out_hash) ? 进程的一些信息

ThreadQty ? 线程的数量

GetThreadN (n) -返回线程（句柄？）

ProcessState ? 返回进程状态

StartProcess (path, args, dir) ? 等价 winexec

SuspendProcess ? 暂停目前调试的进程

ContinueProcess ?继续运行目前调试的进程

Exit_Process ? 退出进程

AttachProcess (p_id) ? 加载调试进程

DetachProcess ?卸载调试进程

SelectThread (t_id) ? 激活选择的线程

StepInto ? 步入

StepOver ? 步过

RunTo (addr) ? 运行到

StepUntilRet ? 运行到返回

BptQty ? 断点的数量

GetBptN (n, out_hash) ? 得到断点的一些信息，out_hash 的关键字有：

Ea ? 断点地址

Size - 断点大小

Type - 断点类型

Pass_count - 断点 pass 多少才激活

Flags - 断点的标志

GetBpt (ea, out_hash) ? 跟上面函数基本一样，不过第一个参数是地址。

AddBpt (ea, size, type) ? 加个断点

DelBpt (ea) ? 删掉断点

EnableBpt (ea) ? 激活断点

DisableBpt (ea) ? 禁止断点

Dbg_module (out_hash) ? 调试模块的信息

Modules_list (out_array) ? 返回调试进程的 module 信息

ThreadContext (t_id, kind_of_context, out_hash) ? 返回线程的上下文信息。

Set_DrX (t_id, reg_value) ?为调试寄存器赋值

Set_Gs (t_id, reg_value) ? 为 Gs 寄存器赋值

Set_Fs (t_id, reg_value) -为 Fs 寄存器赋值

Set_Es (t_id, reg_value) -为 Es 寄存器赋值

Set_Ds (t_id, reg_value) -为 Ds 寄存器赋值

Set_Cs (t_id, reg_value) -为 Cs 寄存器赋值

Set_Ss (t_id, reg_value) -为 Ss 寄存器赋值.

Set_Efl (t_id, reg_value) ? 为标志寄存器赋值

Set_REG (t_id, reg_value) 为通用寄存器(Edi, Esi, Ebx, Edx, Ecx, Eax, Ebp, Esp or Eip)赋值。

怎么样，功能还是很强大吧？下面我们给出个例子来说明这个工具的妙用。

这个例子在程序运行时，在各种条件触发下，往 test.log 文件里写记录。

```
use IDA;
use IDADbg;
sub cbBpt
{
    my( $tid,$addr) = @_;

    printf(FILE "cpBpt instruction:%s ",GetMnem($addr));   #这里可以看出可以调用 Idc 的处理函数，非常好

    my $context;
  if ( ThreadContext($t_id, CONTEXT_ALL, $context) ) #得到当时的 context 信息
  {
    my $reg;
    # dump all registers values
    foreach $reg ( keys %$context )
    {
      printf(FILE "%s .eq. %X ", uc($reg), $context->{$reg} ); #把寄存器信息存入文件
    }

  }

}



sub cbThreadExit
```

```
{
    printf(FILE "cbThreadExit ");
}
sub cbThreadStart
{
    printf(FILE "cbThreadStart ");
}
sub cbProcessExit
{
    printf(FILE "cbProcessExit ");
    close(FILE);
}
sub cbProcessStart
{
    open(FILE, ">d:test.log") or die("Cannot open test.log");
    printf(FILE "cbProcessStart ");


}
```

从上面的例子可以看到, 在 debug 脚本里可以调用强大的 perl 包装的 IDC 脚本函数，可以对程序进行比较深入的研究了。

参考：

http://www.wasm.ru/article.php?article=ida_perl
http://bbs.pediy.com/showthread.php?t=35375

IDA pdb 自动下载

找到 IDA/CFG/PDB.CFG

```
// PDB plugin

#ifdef __PC__                                                   // INTEL 80x86 PROCESSORS
//
// The downloaded symbols are stored in the specified directory.
// Microsoft's public symbol store is used for downloading the symbols.
//
// If this option is omitted or empty   - use _NT_SYMBOL_PATH if set, otherwise use %TEMP%\ida directory
// If the value is not empty                    - use it

PDBSYM_DOWNLOAD_PATH       = "c:\\symbols\\";

// Full symbol path (in _NT_SYMBOL_PATH format)
// If set, PDBSYM_DOWNLOAD_PATH and _NT_SYMBOL_PATH are ignored
PDBSYM_SYMPATH = "SRV*c:\\symbols*http://symbols.mozilla.org/firefox;SRV*c:\\symbols*http://msdl.microsoft.com/download/symbols";

// remote server where win32_remote.exe is running
// used when loading PDB symbols on non-Windows platforms
// NB: it will be used only if there is not already an existing debugging session started
PDB_REMOTE_SERVER = "localhost";
PDB_REMOTE_PORT     = 23946
// password for the remote server
PDB_REMOTE_PASSWD = "";

#endif
```

http://blog.csdn.net/zhangmiaoping23/article/details/15336381

让 WinDbg、IDA、VC 自动下载符号表

- 标 题：让 WinDbg、IDA、VC 自动下载符号表
- 作 者：dnapcex
- 时 间：2010-02-27 13:11:12
- 链 接：http://bbs.pediy.com/showthread.php?t=107893

很多人跟踪到 Windows 自带的 DLL 里的时候有时会很茫然……
突然找到一种方法能让它们都有符号表……
然后里面的各种变量名、函数名等全部显现出来……

设置方法：
“我的电脑”属性中，高级设置里，
有个环境变量，
变量名：_NT_SYMBOL_PATH
变量值：SRV*{$Path}*http://msdl.microsoft.com/download/symbols/
将“{$Path}”替换为要存储 pdb 符号表文件的路径，比如：C:\PDB

于是在 VC 里写程序调试时，或者用 IDA 时，都会从网上自动下载符号表……
msvcrt.pdb，ole32.pdb，System.Data.pdb，ws2_32.pdb，WebDev.WebServer.pdb……

但是 VC 运行程序时会自动下载完所有要用到的 pdb，所以第一次启动会很慢，若想立即看结果，只要把网络断开就行了……
仅此分享……

IDA 修改程序反汇编代码基址

在使用 IDA 对二进制文件进行逆向分析的时候，有时候如果能把特定的代码段的基址修改为指定值，将使得代码更好的理解。
在 IDA 中进行这样的操作的方法有如下几种：

- Manual load 并指定基址
- Edit->Segements->Rebase program

参考资料：http://www.debugman.com/thread/5204/1/1

http://www.programlife.net/ida-rebase.html.

IDA sp-analysis failed 不能 F5 的 解决方案之(一)

转：http://bbs.pediy.com/showthread.php?t=140002

第一种情况：

有时候用 IDA F5 的时候遇到带 sp-analysis failed 的函数会失败.
这只是一个极其简单的一个例子.希望大家能依此类推.也更希望大家把类似的代码放出来一起解决。原本是一朋友问我的,我只是顺便发出来.以后遇到类似情况百度就可以找到了。

代码:

```
.text:004015ED  sub_4015ED          proc  near                          ;  CODE  XREF:  sub_40116A+1A6p
.text:004015ED                                                                ;  sub_40116A+1C7p
.text:004015ED
.text:004015ED  var_C                   =  byte  ptr  -0Ch
.text:004015ED  var_8                   =  dword  ptr  -8
```

```
.text:004015ED  var_4                   =  dword  ptr  -4
.text:004015ED  arg_0                   =  dword  ptr    10h
.text:004015ED
.text:004015ED                              push        ecx
.text:004015EE                              push        esi
.text:004015EF                              push        ebp
.text:004015F0                              mov          ebp,  esp
.text:004015F2                              add          esp,  0FFFFFFF8h
.text:004015F5                              mov          ebx,  offset  String
.text:004015FA                              cmp          dword  ptr  [ebx],  0FFFFFFFFh
.text:004015FD                              jz            short  loc_401606
.text:004015FF                              mov          eax,  offset  loc_401607
.text:00401604                              jmp          eax
.text:00401606  ;  ---------------------------------------------------------------------------
.text:00401606
.text:00401606  loc_401606:                                                 ;  CODE  XREF:  sub_4015ED+10j
.text:00401606                              nop
.text:00401607
.text:00401607  loc_401607:                                                 ;  DATA  XREF:  sub_4015ED+12o
.text:00401607                              xor          eax,  eax
.text:00401609                              mov          esi,  [ebp+arg_0]
.text:0040160C                              mov          al,  [esi+ebx]
.text:0040160F                              xor          ebx,  ebx
.text:00401611                              push        eax
.text:00401612                              call        sub_4014BF
.text:00401617                              mov          [ebp+var_4],  ebx
.text:0040161A                              call        sub_4014D5
.text:0040161F                              mov          [ebp+var_8],  ebx
.text:00401622                              push        0
.text:00401624                              push        offset  dword_403425
.text:00401629                              push        [ebp+var_4]
.text:0040162C                              push        [ebp+var_8]
.text:0040162F                              call        sub_401561
.text:00401634                              mov          eax,  dword_403425
.text:00401639                              ror          eax,  8
.text:0040163C                              cmp          eax,  5
.text:0040163F                              jz            short  loc_401649
.text:00401641                              push        eax
.text:00401642                              mov          eax,  offset  loc_40164A
.text:00401647                              jmp          eax
.text:00401649  ;  ---------------------------------------------------------------------------
.text:00401649
```

```
.text:00401649  loc_401649:                                             ;  CODE  XREF:  sub_4015ED+52j
.text:00401649                              nop
.text:0040164A
.text:0040164A  loc_40164A:                                             ;  DATA  XREF:  sub_4015ED+55o
.text:0040164A                              pop         eax
.text:0040164B                              xor         ah,  [ebp+var_C]
.text:0040164E                              xor         al,  [ebp+var_C]
.text:00401651                              xor         edx,  edx
.text:00401653                              mov         bx,  ax
.text:00401656                              mov         ecx,  4
.text:0040165B
.text:0040165B  loc_40165B:                                             ;  CODE  XREF:  sub_4015ED+83j
.text:0040165B                              and         al,  0Fh
.text:0040165D                              cmp         al,  9
.text:0040165F                              jle         short  loc_401663
.text:00401661                              add         al,  7
.text:00401663
.text:00401663  loc_401663:                                             ;  CODE  XREF:  sub_4015ED+72j
.text:00401663                              add         al,  30h
.text:00401665                              mov         dl,  al
.text:00401667                              ror         edx,  8
.text:0040166A                              shr         bx,  4
.text:0040166E                              mov         al,  bl
.text:00401670                              loop        loc_40165B
.text:00401672                              add         esp,  1Ch
.text:00401675                              pop         ebp
.text:00401676                              pop         esi
.text:00401677                              pop         ecx
.text:00401678                              retn
.text:00401678  sub_4015ED          endp  ;  sp-analysis  failed
```

废话我就不多说了,造成不能 F5 的原因在于:

```
.text:00401642                              mov         eax,  offset  loc_40164A
.text:00401647                              jmp         eax
```

用 OD 打开之后 改掉代码在保存,然后重新用 IDA 打开 F5 就可以了,sp-analysis failed 也就消失了.

在 OD 里直接改成 JMP 40164A

分析之后不难发现不能 F5 的原因,就在于跳转.
一个函数里面的一个跳转在未知的情况下会出现不能 F5.

结果:

代码:

```
char  __cdecl  sub_4015ED(int  a1)
{
    unsigned  __int16  v1;  //  ax@1
    int  v2;  //  edx@1
    signed  int  v3;  //  ecx@1
    unsigned  __int16  v4;  //  bx@1
    char  v5;  //  ST10_1@1
    int  v6;  //  eax@1
    char  v7;  //  ST10_1@1
    char  v8;  //  al@2

    sub_4014BF(String[a1]);
    sub_4014D5(v5);
    sub_401561(0,  0,  &dword_403425,  0);
    v6  =  __ROR__(dword_403425,  8);
    HIBYTE(v1)  =  v7  ^  BYTE1(v6);
    LOBYTE(v1)  =  v7  ^  v1;
    v2  =  0;
    v4  =  v1;
    v3  =  4;
    do
    {
        v8  =  v1  &  0xF;
        if  (  v8  >  9  )
            v8  +=  7;
        LOBYTE(v2)  =  v1  +  48;
        v2  =  __ROR__(v2,  8);
        v4  >>=  4;
        LOBYTE(v1)  =  v4;
        --v3;
    }
    while  (  v3  );
    return  v1;
}
```

第二种情况:

还有一个问题

看看 OPENFILENAMEA 的声明:

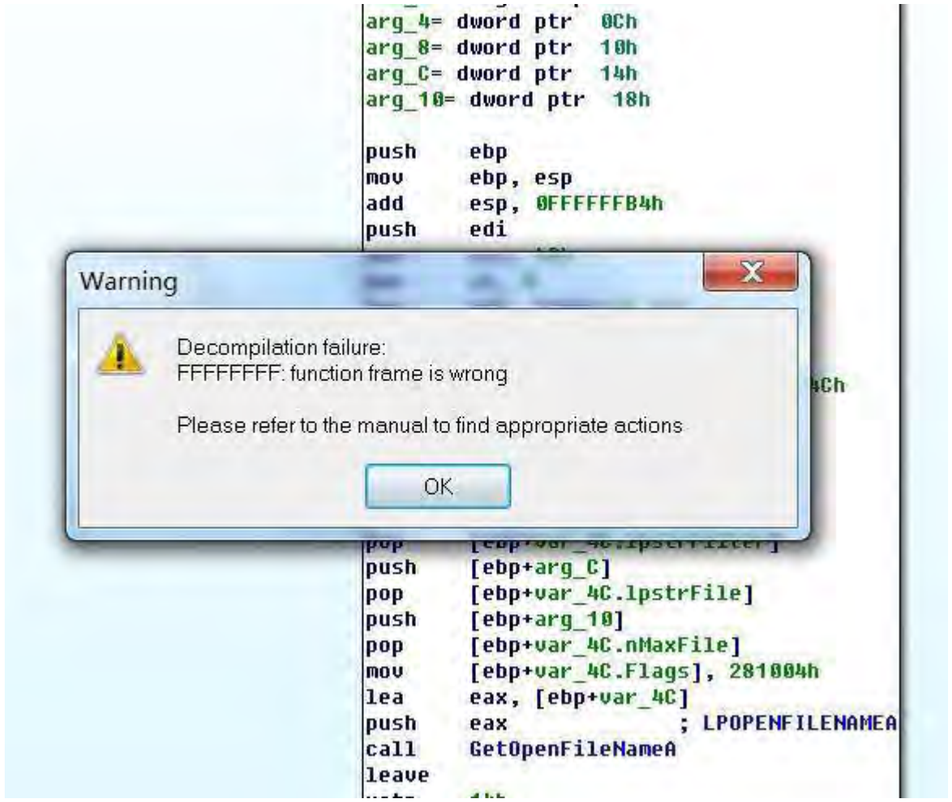
```
typedef  struct  tagOFN  {      DWORD  lStructSize;      HWND  hwndOwner;      HINSTANCE  hInstance;      LPCTSTR  lpstrFilter;      LPTSTR  lpstrCustomFilter;      DWOR
D  nMaxCustFilter;      DWORD  nFilterIndex;      LPTSTR  lpstrFile;      DWORD  nMaxFile;      LPTSTR  lpstrFileTitle;      DWORD  nMaxFileTitle;      LPCTSTR  lpstrIniti
alDir;      LPCTSTR  lpstrTitle;      DWORD  Flags;      WORD  nFileOffset;      WORD  nFileExtension;      LPCTSTR  lpstrDefExt;      LPARAM  lCustData;      LPOFNHOOKPRO
```

C lpfnHook; LPCTSTR lpTemplateName; #if (_WIN32_WINNT >= 0x0500) void * pvReserved; DWORD dwReserved; DWORD FlagsEx; #endif // (_WIN32_WINNT >= 0x0500) } OPENFILENAME, *LPOPENFILENAME;

有个条件就是#if (_WIN32_WINNT >= 0x0500) 的时候，结构多出三个 DWORD，结果大小为 0X58H，在 MASM32 中，没有这三个 DWORD，结果为 0X4CH，看看反汇编代码：

```
OpenFileDlg proc near

var_4C= tagOFNA ptr -4Ch
arg_4= dword ptr  0Ch
arg_8= dword ptr  10h
arg_C= dword ptr  14h
arg_10= dword ptr  18h

push    ebp
mov     ebp, esp
add     esp, 0FFFFFFB4h
push    edi
mov     ecx, 4Ch
mov     al, 0
lea     edi, [ebp+var_4C]
rep stosb
pop     edi
mov     [ebp+var_4C.lStructSize], 4Ch
push    [ebp+var_4C.FlagsEx]
pop     [ebp+var_4C.hwndOwner]
push    [ebp+arg_4]
pop     [ebp+var_4C.hInstance]
push    [ebp+arg_8]
pop     [ebp+var_4C.lpstrFilter]
push    [ebp+arg_C]
pop     [ebp+var_4C.lpstrFile]
push    [ebp+arg_10]
pop     [ebp+var_4C.nMaxFile]
mov     [ebp+var_4C.Flags], 281004h
lea     eax, [ebp+var_4C]
push    eax             ; LPOPENFILENAMEA
call    GetOpenFileNameA
leave
retn    14h
OpenFileDlg endp
```

如果按下 F5 就会酱紫：



这个时候的解决方法：

SHIFT+F9 打开结构 subview,双击 tagOFNA,在最后三个成员上按 U 键，取消掉它们，结果如下：

```
00000000
00000000 tagOFNA struc ; (sizeof=0x4C)
00000000 lStructSize dd ?
00000004 hwndOwner dd ?                  ; offset
00000008 hInstance dd ?                  ; offset
0000000C lpstrFilter dd ?                ; offset
00000010 lpstrCustomFilter dd ?          ; offset
00000014 nMaxCustFilter dd ?
00000018 nFilterIndex dd ?
0000001C lpstrFile dd ?                  ; offset
00000020 nMaxFile dd ?
00000024 lpstrFileTitle dd ?             ; offset
00000028 nMaxFileTitle dd ?
0000002C lpstrInitialDir dd ?            ; offset
00000030 lpstrTitle dd ?                 ; offset
00000034 Flags dd ?
00000038 nFileOffset dw ?
0000003A nFileExtension dw ?
0000003C lpstrDefExt dd ?                ; offset
00000040 lCustData dd ?
00000044 lpfnHook dd ?                   ; offset
00000048 lpTemplateName dd ?             ; offset
0000004C tagOFNA ends
0000004C
```

然后返回反汇编窗口，在函数名上按 U，然后按 P 键，现在按 F5 就可以啦！！

第三种情况：

继续。。

0xXXXXXX: call analysis failed 的解决方案

原因是:反编译插件无法确定其中函数的参数个数.

代码:

```
push        4D8h
call        xxxx
push        eax
call        xxxPrint
add          esp,  8
```

在 IDA 里双击这个函数,进去之后

```
                        ; int (__cdecl *SomePrint)(_DWORD, _DWORD, DWORD, DWORD)
D0 38 E4 00             SomePrint       dd offset sub_E438D0    ; DATA XREF: sub_401C
```

按 Y, 修改函数约定和参数个数....(Y 是 IDA 中的一个快捷键)

把函数　修改成　如图

```
                        ; int (*SomePrint)(_DWORD, _DWORD, ...)
D0 38 E4 00             SomePrint       dd offset sub_E438D0    ; DATA
```

然后就可以　F5　继续分析了.

这个函数本身是别人写的,格式化字符串的函数,像 format　参数本就是无法确定的.所以 IDA 没能识别出来.
我知道　这个函数有两个参数是固定的　后面是不确定的,所以　　三个点　...　就 ok 啦
希望对某些人能有所帮助.

http://www.hex-rays.com/products/decompiler/manual/failures.shtml

英语好的人,可以去官方主页去找解决方案.

IDA sp-analysis failed 不能 F5 的 解决方案之(二)

转：http://bbs.pediy.com/showthread.php?t=158896

高手飘过~,本文章只想让搜索引擎收录,以便那些新手查找的。

如图,

```
:01251EC0                   mov     al, 1
:01251EC2
:01251EC2 loc_1251EC2:                                  ; CODE XRE
:01251EC2                   mov     ecx, [ebp-0Ch]
:01251EC5                   mov     large fs:0, ecx
:01251ECC                   pop     ecx
:01251ECD                   pop     edi
:01251ECE                   pop     esi
:01251ECF                   pop     ebx
:01251ED0                   mov     ecx, [ebp-10h]
:01251ED3                   xor     ecx, ebp
:01251ED5                   call    sub_1382EF3
:01251EDA                   mov     esp, ebp
:01251EDC                   pop     ebp
:01251EDD                   retn
:01251EDE ; ---------------------------------------------------------------------------
:01251EDE
:01251EDE loc_1251EDE:                                  ; CODE XRE
:01251EDE                   lea     ecx, [ebp-890h]
:01251EE4                   call    sub_455120
:01251EE9
```

代码是红色的,就是说 IDA 没有分析出来,为了方便分析,我们会用快捷键 P(Create Function) 来创建函数.只是我们在函数头部 按 P 的时候 总会提示

The function has undefined instruction/data at the specified address.
Your request has been put in the autoanalysis queue.

大家注意 retn 后面两行多余的代码,这是罪魁祸首.如果我们不能创建函数,就不能使用 F5 来分析代码了,想让这些代码变绿色的方法(创建函数)很简单.
用鼠标从函数头部开始到 retn 结尾 拖一下,变成灰色.

```
__:01430FFA locret_1430FFA:                             ; CODE XRE
__:01430FFA                 retn
__:01430FFB ; ---------------------------------------------------------------------------
__:01430FFB                 mov     eax, [ebp-10h]
__:01430FFE                 and     eax, 8
__:01431001                 jz      locret_1431013
__:01431007                 and     dword ptr [ebp-10h], 0FFFF
__:01431008                 lea     ecx, [ebp-24h]
__:0143100E                 jmp     sub_591D80
__:01431013 ; ---------------------------------------------------------------------------
__:01431013
__:01431013 locret_1431013:                             ; CODE XRE
__:01431013                 retn
__:01431014 ; ---------------------------------------------------------------------------
__:01431014
__:01431014 loc_1431014:                                ; DATA XRE
__:01431014                 mov     edx, [esp+8]
__:01431018                 lea     eax, [edx+0Ch]
__:0143101B                 mov     ecx, [edx-28h]
```

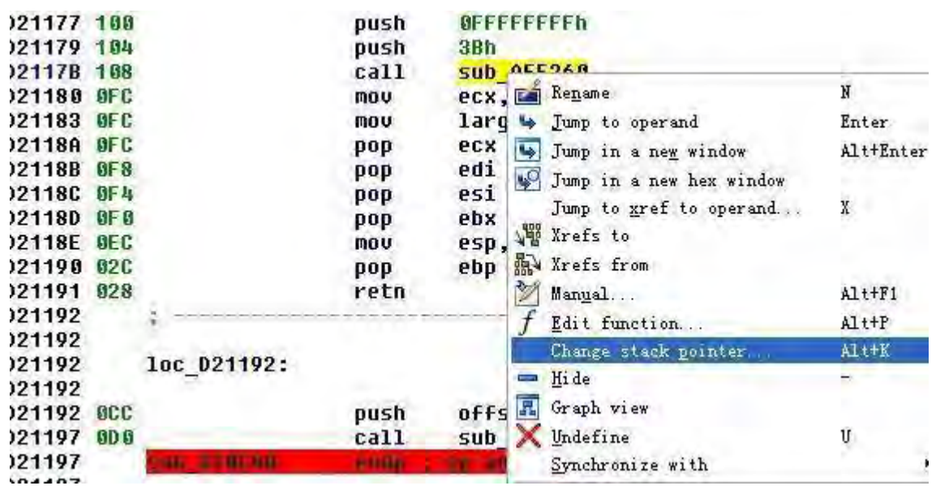
然后用快捷键　P　或者右键菜单,CreateFunction　来创建函数就可以了。

用拖的方法,去创建一个函数的好处是,我们可以选择任意的一部分 IDA 分析不出来的代码(就是红色代码),成为一个绿色的代码(IDA 可识别的代码).

当然了,到这里也不算完.下图是代码变绿色的,我用上面的方法创建函数了,可还是不能 F5,

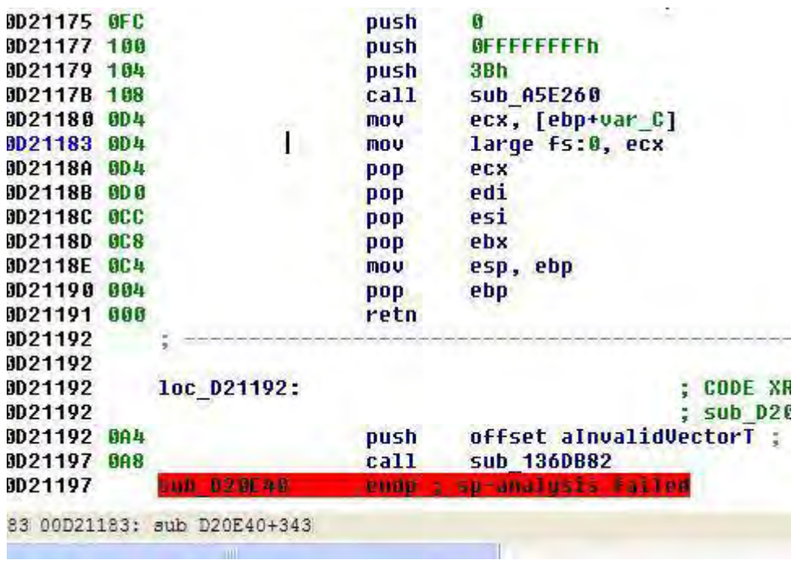
```
00D21177 100                    push    0FFFFFFFFh
00D21179 104                    push    3Bh
00D2117B 108                    call    sub_A5E260
00D21180 0FC                    mov     ecx, [ebp+var_C]
00D21183 0FC                    mov     large fs:0, ecx
00D2118A 0FC                    pop     ecx
00D2118B 0F8                    pop     edi
00D2118C 0F4                    pop     esi
00D2118D 0F0                    pop     ebx
00D2118E 0EC                    mov     esp, ebp
00D21190 02C                    pop     ebp
00D21191 028                    retn
00D21192     ; ---------------------------------------------------------
00D21192
00D21192     loc_D21192:                                ; CODE XR
00D21192                                                ; sub_D20
00D21192 0CC                    push    offset aInvalidVectorT ;
00D21197 0D0                    call    sub_136DB82
00D21197     sub_D20E50         endp ; sp-analysis failed
00D21197
```

我们通过 Stack　Pointer　发现这个函数的堆栈是不平衡的.此时我们需要找接近 retn　的　最后一个 Call　然后去修正堆栈.



我们看到　retn　左边绿色的数字是　28,我们通过 ALT　+　k　修正堆栈,把　28　retn　改成　0　就可以了。

修正堆栈的时候一定要选择 Call 然后修正堆栈,下图是修正后的 代码



很简单的一个东西,篇幅有点大了,可能存在一些问题,不过我的目的是 F5 就可以了 嘿嘿.

> 引用:
> *最初由 **曾半仙**发布*
> 从上往下逐渐的对每个 call, 通过编辑功能的堆栈弹出数和保护寄存器数量这样修, 这样可以扩散到以后每个调用的地方. 修好重新点 Anaylist, 自动调整某些单独没有接受的地方.
> 只在最后强行改的话, hexrays 为了配平堆栈, 会给一些函数增删错误的参数数量, 导致其他函数分析也会受阻.

多谢兄台,不过 F5 本身在参数个数的分析上会有问题,不是很精确,F5 之后 点函数用快捷键 Y 修正一下函数原型就行了。

关于 IDA 显示中文字符串的问题

DA 显示中文字符串总是乱码，很不给力有木有！思索良久，忽然想起《加密与解密》第三版上曾经对此有所提及，所以赶紧翻书查看解决方案，说是要修改 ida.cfg 这个配置文件。
所以，来到 IDA 的安装目录，找到 cfg\ida.cfg，然后用 Notepad++(最好不要用微软自带的记事本)打开，搜索 AsciiStringChars，将其中 cp866 version 的几行注释掉，然后去掉 full version 段的几行的注释，保存文件并重启 IDA。

啊！还是乱码？坑爹啊有木有！最后无赖的到处找解决方案，终于找到了一个插件，可以显示中文字符（不过还是感觉不太方便）。
点击下载插件 里面有源码，有兴趣的同学可以改进一下。把压缩包解压下，把 **unispector.plw** 复制到 IDA 的插件目录下重启 IDA，然后来到要显示为中文的字符串的定义的地方（**注意是定义字符串的地方，也就是 db**），把鼠标在变量名上点一下，然后按下 Ctrl + Alt + Z 快捷键（这时候可能和 QQ 的冲突，所以你可以选择退出 QQ 或者改掉 QQ 的快捷键），然后就可以显示出中文了。

总之觉得这样虽然可以显示中文，但还是有点麻烦。如果要是去字符串窗口一个一个按快捷键，太费事了。

更正：DBank 现在服务确实很差，看雪原帖地址：http://bbs.pediy.com/showthread.php?t=106225

http://www.programlife.net/ida-chinese-string-plugin.html

改善 IDA6.8 对中文等非英语国家的 ANSI 字符串显示支持不佳的问题

```
int _tmain(int argc, _TCHAR* argv[])
{
  printf("%s","我是中国人");
  return 0;
}
```

这个几句代码编译的程序，在 IDA6.4 Demo 的字符串窗口还能识别出来

可是到了 IDA6.6,就显示乱码了

BUG 呀，弄起我 IDA6.4 一直不敢删除，算了忍忍，以为到了后面的版本出来了，会修复，天天看看雪，等啊等，终于等到 IDA6.8 出来了，一开程序，发现 MD 还是这样，情何以堪，

不晓得官方，是不是因为上次是中国区泄露了他们的软件，特别不照顾中国客户，还是怎么的，没办法了，还是自己动手吧，自己动手丰衣足食。
逆向 IDA，寻寻觅觅，倒腾到字符串列表管理的几个函数
get_strlist_item refresh_strlist get_strlist_qty
get_ascii_contents2 ea2str str2user print_ascii_string_type areacb_t_get_area，
调试跟踪还是无果。
第二天休息好，继续调试，猜测，调试，
想是不是 QT 版本的问题，替换成 IDA6.4 用的 4.8.3，还是不行，差个函数没有，试了几次，不好替换，会报错，只好放弃这个方案。继续打开 IDA 逆向 IDA，漫无目的的想从代码中，寻找其他线索，但是心里还是没底的，几次想放弃，又不舍
搜索字符串嘛 Strings，Strings window，String 还是无果，
试了试 Ansi,Asc
真的是灵感只青睐于思考着的人们啊。老天把我引导了这个地方

感觉好像发现了契机，另外开个 IDA，依然逆向这个示例程序，按 Alt+A，这个窗口出现了

encording，对这就是 encording，编码问题，F1 结合 IDA 的帮助文档,知道了 IDA 关于 ANSI 编码问题的操作使用，如何添加一个编码 名，更改默认编码， 首先测试默认已有的编码名<default>，<no conversion>， UTF- 16LE，UTF-8，都是无法使 ANSI 中文识别出来的，接着尝试添加简体中文编码 GBK，成功了，可以添加的（IDA 还支持添加‘CP+代码页整 数’，‘代码页整数’的形式的编码名），然后应用到"8 位和多字节字符串"。

再次打开 Strings Windows，OH， YES！终于正确了，熟悉的中文出现了

最后思考，为什么 IDA6.4 和 IDA6.8 都是 Default(<no conversion>)

却显示结果不同呢，但是他们的帮助文档中关于<default> 和<no conversion>的描述都是相同 的，Default(<no conversion>)这句的意思是 Default(默认)就 是<no conversion>，
,如上图，编码可选列表中也没有<default>这项，足以见得<default>就是<no conversion>
而帮助文档中显示

<default> - the default encoding for this string type (8-bit or 16-bit)

<no conversion> - the string bytes are printed using the current system encoding (after translating with XlatAsciiOutput array in the configuration file).

<no conversion>是 the current system encoding（当前系统编码），那么理论上应该简体中文的系统 就是简体中文为<no conversion>的代表了。可是 IDA6.8 中却默认显示乱码，无法正确显示。
足以见得这是一个 BUG。
修复方式是：
1.每次新逆向一个程序时，去'Options'->'ASCII string style'中将 ANSI 字符串设置成简体中文。
2.上面的方案，太麻烦了，要用户留心一个事情，不符合程序设计的初衷，写程序的初衷就是让机器自动去完成那些我们不想记住，不想完成，繁琐机械的 任务，提高生产效率。所以接着我写了一个 IDA 插件，会在 IDB 初始化时，自动帮你添加一个当前代码页的编码名，并将当前"8 位和多字节字符串"设置到这 个编码名上。

下面是本插件的程序及源码，请解压后把 bin 目录下的两个文件放到 IDA6.8 的插件目录(plugins)中。其他 IDA 版本的用户可能需要自己编译源码。
AutoSetToLocalAnsiCodePage.7z.
最后说一句，还是请大家支持正版，如果大家都等着使用盗版，hex-rays 就没资金继续开发完善这么优秀的软件，给我们继续使用，这样就是一个恶性循环了，不好。有钱的就买正版，在公司单位上班的需要用到 IDA 作为工作工具就要求公司买正版。
http://bbs.pediy.com/showthread.php?t=206381

跟踪到当前

Set current ip
Action name: ThreadSetCurrentIp

This command sets the instruction pointer of the current suspended thread to the current cursor location.

It is accessible only when the debugger is active and the process is suspended.

See also Debugger submenu.
某某
快捷键 ctrl+n

无名诸葛
直接修改 ip 寄存器的值

Eric
ida 右键
set ip 就可以，nop 掉 会被内存效验发现的
其实就是该 PC 寄存器的值
0x86 的 eip

Ida 修改程序

Edit (patch) a binary file in IDA Pro
IDA 6.2 has the "Edit" -> "Patch program" menu enabled out of the box. No need to modify idagui.cfg. There is also an "Apply patches to input file" option that that will modify the target file.

Binary Cracking & Byte Patching with IDA Pro

Introduction

This rare dissertation is committed to demonstrate cracking and byte patching of a binary executable using IDA Pro with the intention of subverting various security constraints, as well as generating or producing the latest modified version (patched) of that particular binary. IDA Pro is typically utilized to analyze the disassembled code of a binary so that internal mechanism could be comprehended and identify the inherent vulnerability in the source code.

This article comprises the following contents:

- Binary Sample
- Patching Configuration in IDA Pro

- Binary Analysis
- Binary Cracking & Patching with IDA Pro
- Script Patching Substitute
- Final Note

IDA Pro appears to have managed mystical potentials in the reverse engineer minds by having the impression that merely opening a binary with IDA will reveal all the secrets of a target file. IDA Pro is intended to assist you in considering the behavior of a binary by offering us disassembled code. IDA Pro is in fact, not designed to modify or patch the binary code to suit your needs like other tools such as OllyDbg and CFF Explorer. It is really only a static-analysis disassembler tool. It can only facilitate your attempts to locate software vulnerabilities, bugs and loopholes which are typically, utilized by both white hat and black hat professionals. Ultimately, it is up to your skills and how you apply them as to whether IDA makes your search for vulnerabilities easier.

Essential

This tutorial requires thorough knowledge of Assembly Programming and Hex Code manipulation because patching binary with IDA Pro especially deals with assembly opcode instructions. Besides that, the reverse engineer is supposed to operate the IDA Pro Software IDE features perfectly. This operation lists the following tools of trades as:

- The Target Binary (C/C++ code)
- IDA Pro Interactive Dissembler
- IDA-Script File (*.idc files)
- Assembly Language skills
- ASCII Converter

Binary Sample

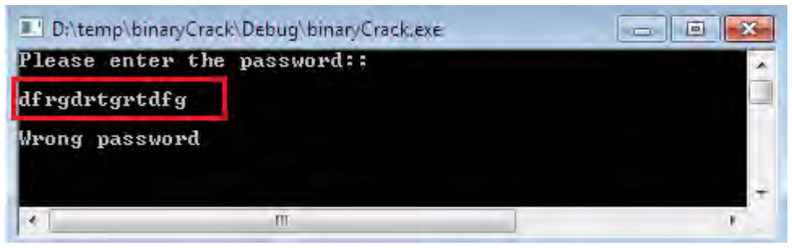
This article exposes the demonstration of byte patching over a typical C++ binary which essentially required user password to validate his identity and let him log into the system and such confidential information is only provided to the registered person indeed. There is, of course, no direct method to breach into this application without being authenticated except reverse engineer or patch the critical bytes which are responsible for performing validation. The following code will make the binary executable live as *binaryCrack.exe* as:

Hide Copy Code

```
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>

#define password "ajay"

int _tmain(int argc, _TCHAR* argv[])
{
  char pwd[100];
```

```
  printf("Please enter the password::\n\n");
  scanf("%s", pwd);
  if ( strcmp(pwd, password) == 0 )
  {
      printf("Congratulation!!\n\n");
      printf("Ready to login with: %s",password);
  }
  else
  {
      printf("Wrong password");
  }
  getch();
  return 0;
}
```

We can use any compiler to execute the aforementioned binary source code which makes an executable as *binaryCrack.exe* and when we run that file, it will prompt to enter the password. If we enter the correct password as “ajay”, then it shows the congratulations message, otherwise issue Wrong Password message as follows:



It is probable that we might not be aware of the real password and in such circumstances; we can’t proceed without this information. So the only option left is Reverse Engineering this binary and manipulates the sensitive bytes to suit your needs as we shall see in the next sections.

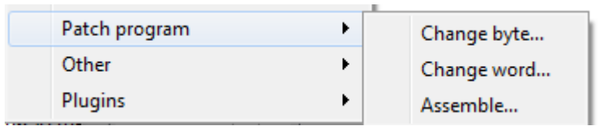
Patching Configuration in IDA Pro

The patching or editing assembly code features are normally invisible in the IDA Pro. You can double check it from the Edit menu that none of the Patch program options appeared. Thus, in order to make this option visible, open the *idagui.cfg* configuration file of IDA Pro which is located at Drive: *\Program Files\IDA PRO Advanced Edition\cfg* folder and scroll down to find the DISPLAY_PATCH_SUBMENU option which is typically set to NO. So, make the following changes and save this file.

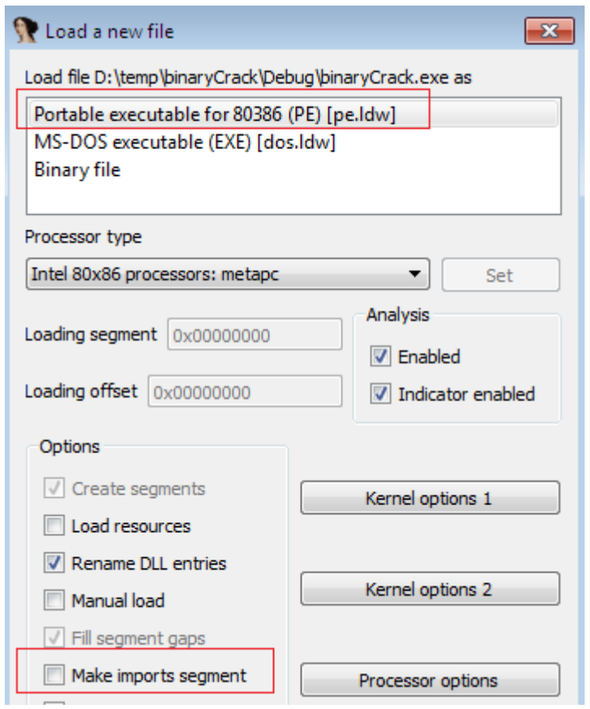
```
DISPLAY_PATCH_SUBMENU     = YES
DISPLAY_COMMAND_LINE      = NO
```

```
  // Display the Edit,Patch submenu
// Display the expressions/IDC command line
 // To turn on/off the command line,
 // right click on the main toolbar after
 // setting this parameter to YES
```

After saving this file, re-launch the IDA PRO and the movement you change the submenu option in the configuration file, you can notice that Patch program option becomes visible in the Edit menu as:

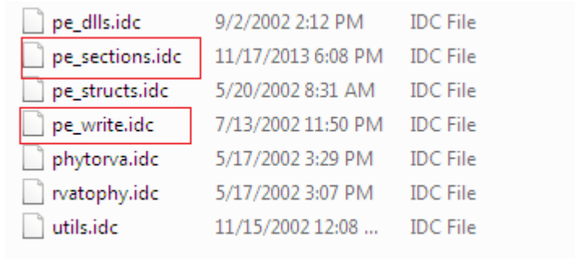


Now, load the target binary in to IDA Pro, it will ask to create a new database as usual, then select the PE file option as given in the following figure and the important point to remember is don't forget to uncheck the Make imports segment option because some useful date can be lost from the database if this option is enabled as follows:

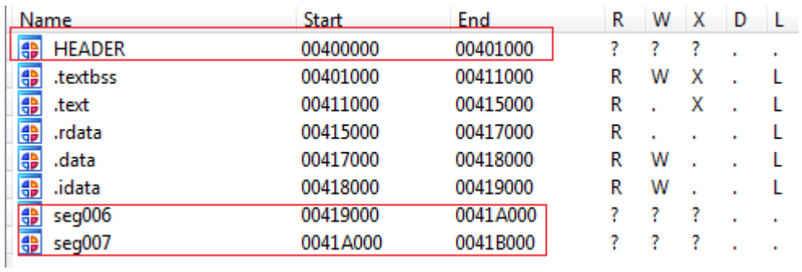


The target file will be loaded into the IDA Pro but we could still not modify the byte sequence of the binary file even if enabling the Patch program option in the Edit menu earlier. So here, the role of special IDA script files came into light as they are able to modify the byte like OllyDbg as well as writing the changes into the executable to make the effect permanent.

The IDC script files can be downloaded from this URL as http://www.openrce.org/downloads/details/57/PE_Scripts. When downloaded, extract the files in a separate folder on the file system of your machine. There are couple of scripts files provided but mainly; two script files are significant as follows:



After loading the target binary into the IDA Pro, open the folder where the aforesaid IDA script files are located and execute the *pe_sections.idc* file in order to extending new functionality into IDA Pro such binary patching and writing. You can ensure about new specification from the Segments (shift + F7) windows that certain new segments are automatically added.



Name	Start	End	R	W	X	D	L
HEADER	00400000	00401000	?	?	?	.	.
.textbss	00401000	00411000	R	W	X	.	L
.text	00411000	00415000	R	.	X	.	L
.rdata	00415000	00417000	R	.	.	.	L
.data	00417000	00418000	R	W	.	.	L
.idata	00418000	00419000	R	W	.	.	L
seg006	00419000	0041A000	?	?	?	.	.
seg007	0041A000	0041B000	?	?	?	.	.

After successful completion of such aforesaid operations, we can modify as well as write the byte sequence into binary file.

Binary Analysis

We have only the binary executable and so it is almost impossible to know about the logic implementation without the source code. But we can disassemble the source code of any binary by employing IDA Pro because unless we are aware with logic flow, how can we subvert any security mechanism. Hence, let’s analyze the proper logic flow path of the binary file.

As we can consider the following image, IDA Pro disassembles the binary into raw assembly instruction sets. This program first, prompts the user to enter the password by displaying a string message, then compares this value to a predefined value “ajay” which might be the real password. The comparison happens via string class strcmp method and if the value is 0, then the entered value is corrected otherwise incorrect.

```
.text:004113C8                mov     esi, esp
.text:004113CA                push    offset aPleaseEnterThe ; "Please enter the p
.text:004113CF                call    ds:printf
.text:004113D5                add     esp, 4
.text:004113D8                cmp     esi, esp
.text:004113DA                call    sub_411145
.text:004113DF                mov     esi, esp
.text:004113E1                lea     eax, [ebp+var_6C]
.text:004113E4                push    eax
.text:004113E5                push    offset aS       ; "%s"
.text:004113EA                call    ds:scanf
.text:004113F0                add     esp, 8
.text:004113F3                cmp     esi, esp
.text:004113F5                call    sub_411145
.text:004113FA                push    offset aAjay    ; "ajay"
.text:004113FF                lea     eax, [ebp+var_6C]
.text:00411402                push    eax
.text:00411403                call    j_j_strcmp
.text:00411408                add     esp, 8
.text:0041140B                test    eax, eax
.text:0041140D                jnz     short loc_411444
.text:0041140F                mov     esi, esp
```

If the value of eax register is 1, then the execution is shifted towards the loc_411444 block by jnz statement, where the “wrong password” message would be echoed in the screen as follows:

```
text:00411444 loc_411444:                             ; CODE XREF: sub_4113A0+6D↑j
text:00411444                mov     esi, esp
text:00411446                push    offset aWrongPassword ; "Wrong password"
text:0041144B                call    ds:printf
text:00411451                add     esp, 4
text:00411454                cmp     esi, esp
text:00411456                call    sub_411145
```

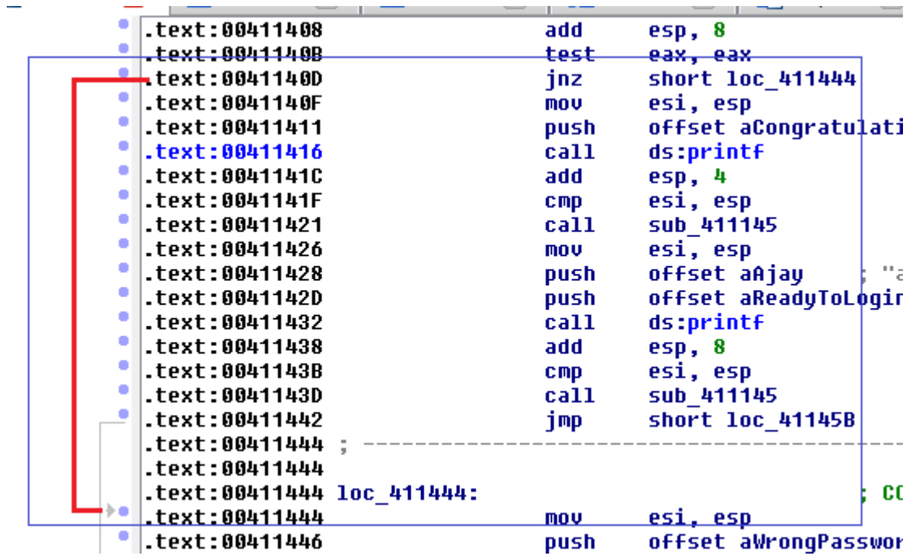
And if the value of eax is 0, then jump won’t throw the execution anywhere. The program intimates us that we have entered the correct password as follows:

```
mov     esi, esp
push    offset aCongratulation ; "Congratulation!!\n\n"
call    ds:printf
add     esp, 4
cmp     esi, esp
call    sub_411145
mov     esi, esp
push    offset aAjay    ; "ajay"
push    offset aReadyToLoginWi ; "Ready to login with: %s"
call    ds:printf
add     esp, 8
cmp     esi, esp
call    sub_411145
jmp     short loc_41145B
```

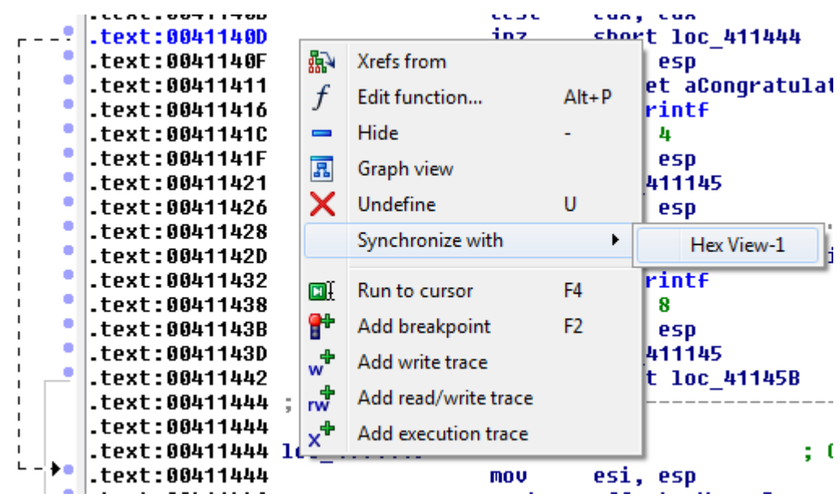
We can easily anticipate that Jump instruction determines either the password is correct or incorrect. If the eax register has value 1, then execution is diverted towards the “wrong password” box flow otherwise the “congratulation” message would be displayed. As we are implying from execution flow, everything depends on jnz instruction. So, the reverse engineer would surely be interested in this instruction because manipulation of this instruction can produce different results.

Binary Cracking and Patching

As we have concluded from the analysis, the outcome of this binary application is regulated by the jnz short loc_411444 statements. The loc_411444 instruction belongs to the unmatched password displaying block and execution is directly jump to this block as you can notice in the RED arrow.



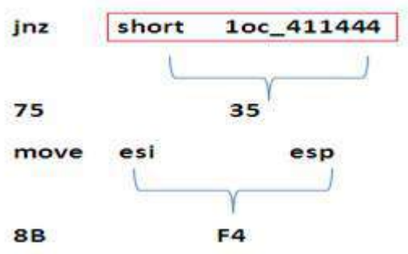
However, all we have to do is to change the corresponding jnz statement related bytes. So, right click on the current location of this statement and synchronize it with Hex Editor View in order to examine this statement byte sequence as:



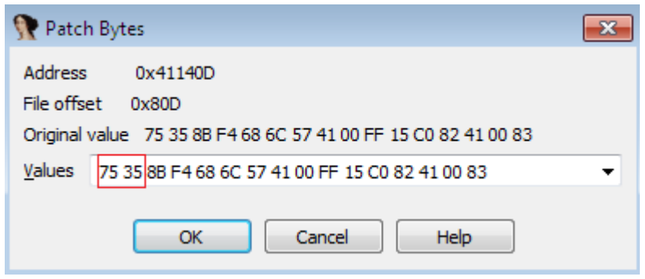
The hex view offers a 16 bytes of sequence in one line and each two bytes represent only one assembly instruction set such that the hex value 75 35 belongs to .text: 0041140D address location where assembly code jnz short 10c_411444 is implemented as follows:

```
004113CD  41 00 FF 15 C0 82 41 00  83 C4 04 3B F4 E8 66 FD  A. .+éA.â-.;(Fƒ²
004113DD  FF FF 8B F4 8D 45 94 50  68 8C 57 41 00 FF 15 C4    ÿ(ìEöPhîWA. .-
004113ED  82 41 00 83 C4 08 3B F4  E8 4B FD FF FF 68 84 57  éA.â-.;(FK²  häW
004113FD  41 00 8D 45 94 50 E8 9D  FC FF FF 83 C4 08 85 C0  A.ìEöPF¥n  â-.à+
0041140D  75 35 8B F4 68 6C 57 41  00 FF 15 C0 82 41 00 83  u5ï(hlWA. .+éA.â
0041141D  C4 04 3B F4 E8 1F FD FF  FF 8B F4 68 84 57 41 00  -.;(F.²  ï(häWA.
0041142D  68 50 57 41 00 FF 15 C0  82 41 00 83 C4 08 3B F4  hPWA. .+éA.â-.;(
0041143D  E8 03 FD FF FF EB 17 8B  F4 68 3C 57 41 00 FF 15  F.²  d.ï(h<WA. .
0041144D  C0 82 41 00 83 C4 04 3B  F4 E8 EA FC FF FF 8B F4  +éA.â-.;(FOn  ï(
```

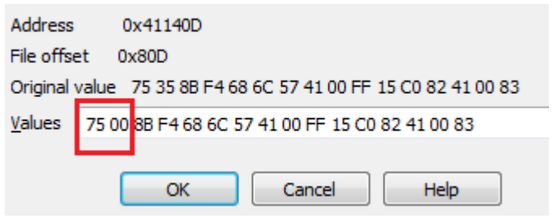
So we have to identify the correct bytes to specific instruction so that we can modify to suit our needs. The following figure showcasing the hex code refers to which instruction as follows:



Finally, we have concluded that hex code 35 is the key value that is directing the execution flow of the program. However, select the address location 0041140D into text view and go to Edit menu, choose Patch program and select Patch Bytes over there. It will show up the entire 16 hex bytes sequence alike to hex view as follows:



In order to diffuse the effect of jnz statement where from the execution is shifting towards the 1oc_411444 block, we have to change its corresponding hex value 35 to 00 which typically fill nothing action in the memory as follows:



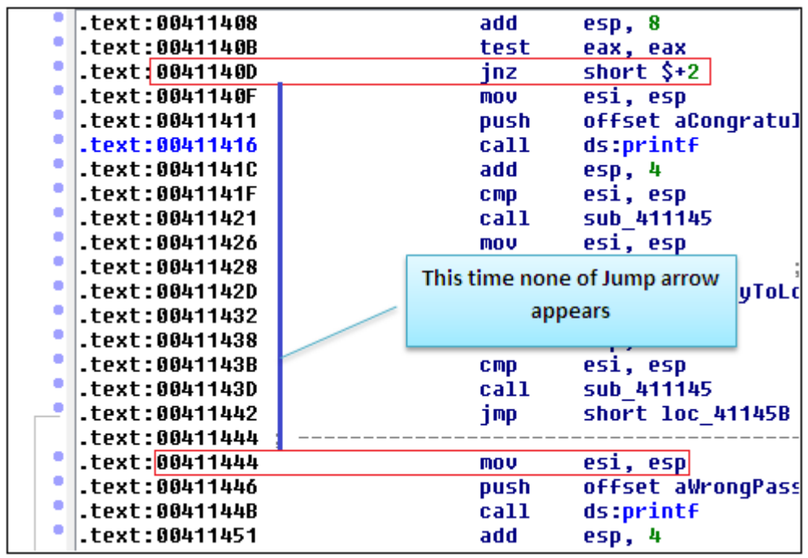
The moment you change the loc_411444 instruction code to 00, it will also reflect in the assembly as follows:

```
.text:0041140B                  test    eax, eax
.text:0041140D                  jnz     short $+2
.text:0041140F                  mov     esi, esp
.text:00411411                  push    offset aCongratulation ; "Congratulation!
```

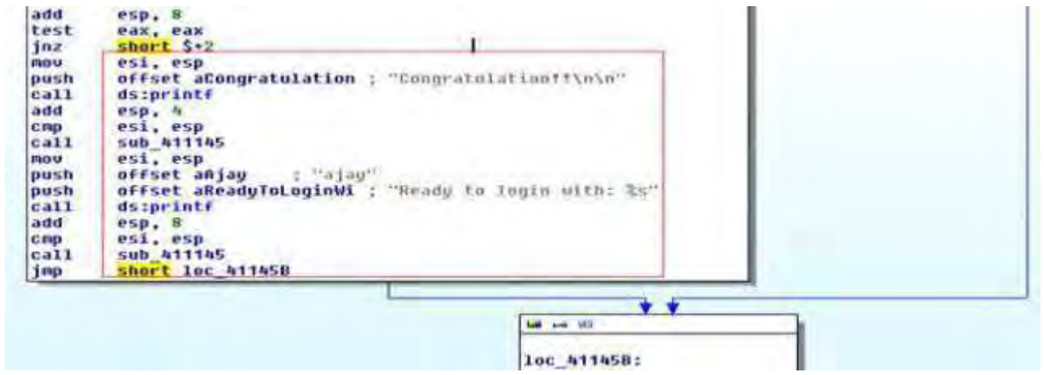
You can also notice the modification in the hex view as follows:

```
004113FD  41 00 8D 45 94 50 E8 9D
0041140D  75 00 8B F4 68 6C 57 41
0041141D  C4 04 3B F4 E8 1F FD FF
```

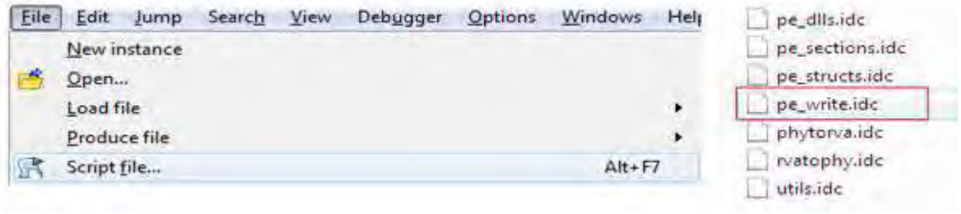
One of the important change to notice, is that after making the value 00, this time no jump arrow is showing which was pointing the execution divert to loc_411444 like earlier.



Now open the graph view, and notice that the “congratulation” block is merged into the main code rather than being separated like earlier before the editing the hex value. The BLUE arrow finally goes to the end of statement block and “wrong password” block is isolated or disconnected from the main operation. So even if we enter the wrong password value, always the correct password block would be executed because jnz statement code is diffuse to 00 and loc_411444 block is disconnected as follows:



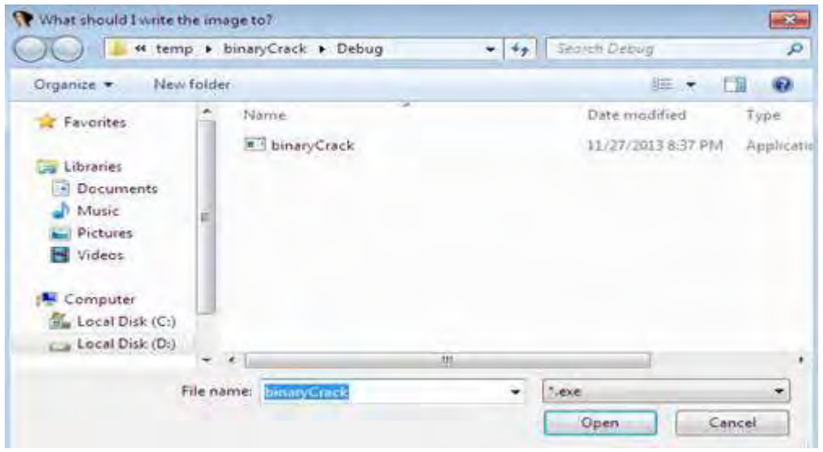
Ok, we have done the byte editing, it is time to save the effect permanently into memory, but IDA Pro is not capable to write bytes of the binary file into memory, instead it can write altered bytes into database. So for this purpose, IDA special script files are used here. Go to File and select Script file and choose *pe_write.idc* which make the perpetual effects in the memory as:



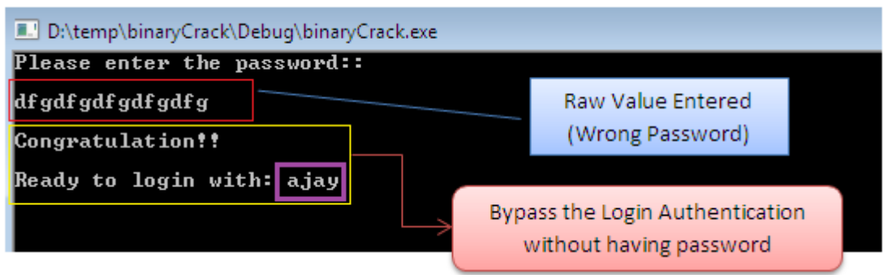
The moment you run the *pe_write.idc* file, you will notice that bytes has been written to segments successfully and lastly, IDA will prompt to re-save the binary file as follows:

```
Sections written out:

   HEADER : [00400000] --> [00000000, 00000400]
  .textbss : [00401000] --> [00000000, 00000000]
    .text : [00411000] --> [00000400, 00003600]
   .rdata : [00415000] --> [00003A00, 00001E00]
    .data : [00417000] --> [00005800, 00000200]
   .idata : [00418000] --> [00005A00, 00000A00]
   seg006 : [00419000] --> [00006400, 00000600]
   seg007 : [0041A000] --> [00006A00, 00000600]

The file has been written out
------------------------------------------------------
```

Save the modified binary with the same name as *binaryCrack.exe* as:

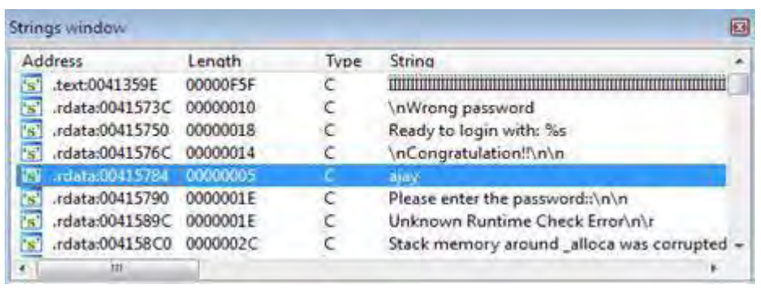


Now, load the *binaryCrack.exe* and it will prompt to enter the password, merely enter any value and BINGO@!!!!!!! Congratulations message along with original password appearing. We have successfully bypassed or subverted the password mechanism by patching some related critical bytes using IDA Pro.



Patching String Bytes

As we can observe in the *binaryCrack.exe*, couple of strings messages are showing. We can access all these strings into place via String window (shift + F12) and can directly reach its assembly code merely by clicking the string.



It is not a good programming practice to show the sensitive strings likes serial keys or passwords directly. Such information should be hidden because after patching (reverse engineering) the passwords assembly code, the hacker can easily aware that the password is “ajay” as it is showing after “congratulations” message. So we can hide or rename the showing password on screen by patching its corresponding bytes.

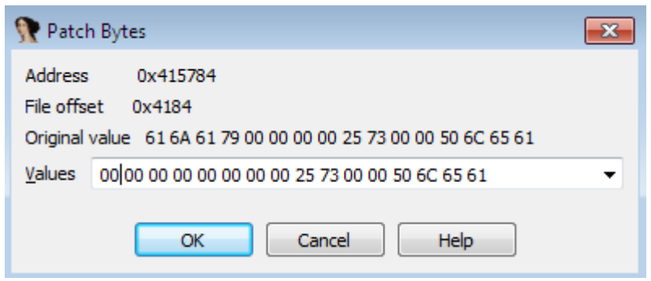
So, double click the “ajay” in the String windows, and IDA would let us reach to its assembly code where from we can modify its visibility as:

```
.rdata:00415781                 db    0
.rdata:00415782                 db    0
.rdata:00415783                 db    0
.rdata:00415784 aAjay           db 'ajay',0
.rdata:00415784
.rdata:00415789                 align 4
```

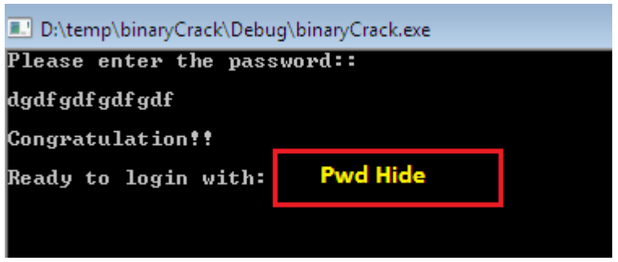
Thereafter open the Hex view and we can observe the “ajay” byte code sequence as 61 6A 61 79 00 as follows:

```
61 73 73 77 6F 72 64 00  00 00 00 00 52 65 61 64
79 20 74 6F 20 6C 6F 67  69 6E 20 77 69 74 68 3A
20 25 73 00 00 00 00 00  0A 43 6F 6E 67 72 61 74
75 6C 61 74 69 6F 6E 21  21 0A 0A 00 00 00 00 00
61 6A 61 79 00 00 00 00  25 73 00 00 50 6C 65 61
73 65 20 65 6E 74 65 72  20 74 68 65 20 70 61 73
73 77 6F 72 64 3A 3A 0A  0A 00 00 00 00 00 00 00
```

Again open the, the Patch Bytes from the Patch program resides in the Edit menu and replace all these bytes with 00 as follows:



Now once again, run the *pe_write.idc* to make changes perpetual in the binary file and the *binaryCrack.exe*, enter any value as password and observe that this time real password is not showing.

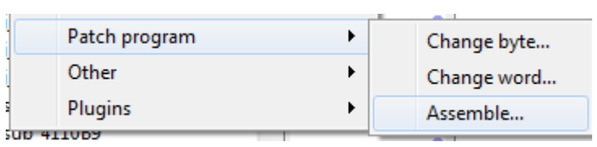


Removing Segments

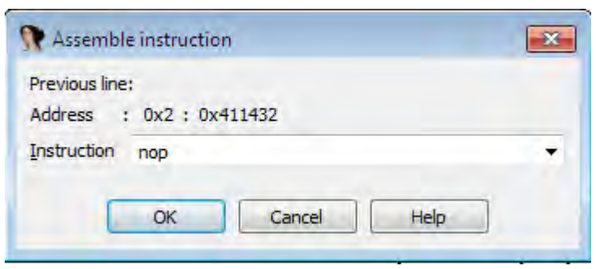
A question comes into mind, why should we display the string message after the "Congratulations" message when we subverted the password mechanism. We cannot allow to execute a particular assembly code by using IDA Pro. Here, we can easily figure out that the code instruction after address location 00411411 is senseless to be display as:

```
.text:0041140F          mov     esi, esp
.text:00411411          push    offset aCongratulation ; "\nCongratulation!!
.text:00411416          call    ds:printf
.text:0041141C          add     esp, 4
```

This time, we are not patching the bytes, instead we are integrating new assembly code so that any string message won't display after "Congratulations" message. Hence, go Patch program and choose Assemble as follows:



Now place nop assembly instruction which stands for no operation and it decides that none of the code would be executed as follows:



After this, you can notice that nop is placed after 0041132 locations as follows:

```
.text:0041142D          push    offset aReadyToLoginW:
.text:00411432          nop
.text:00411433          adc     eax, offset printf
```

Now, run *pe_write.idc* again and notice that none of the code is executing after the message as follows:

```
Please enter the password::
gdfgdg
Congratulation!!
```

Script Patching Substitute

It is not necessary that we can only patch bytes by using IDA Script file which exports the current Ida's database into the EXE binary executable. Rather, we can opt with another approach. First, make changes in the byte or hex code just by editing them and produce a new DIF file as *test.dif*. Now open the *test.dif* file and it show the original hex code and patched code as follows:

```
This difference file has been created by IDA Pro
binaryCrack.exe
0000080E: 35      00
          Patched Byte
```

Later, compile this following C program by using any editor (I suggest use GCC in the linux platform) as:

Hide Shrink Copy Code

```
#include <stdio.h>

int main(int argc, char **argv) {
   char line[256];
   FILE *patch = stdin;
   FILE *input = NULL;
   unsigned int offset;
   int orig;
   int newval;

   int i;

   for (i = 1; i < argc; i += 2) {
      if (!strcmp(argv[i], "-p")) {
         if ((i + 1) < argc) {
```

```
      FILE *f = fopen(argv[i+1], "r");
      if (f) {
         patch = f;
      }
      else {
         fprintf(stderr, "Failed to open patch file %s\n", argv[i+1]);
         exit(0);
      }
   }
}
else if (!strcmp(argv[i], "-i")) {
   if ((i + 1) < argc) {
         fprintf(stderr, "Opening %s\n", argv[i+1]);
      input = fopen(argv[i+1], "rb+");
      if (input == NULL) {
         fprintf(stderr, "Failed to open input file %s\n", argv[i+1]);
         exit(0);
      }
   }
}
else {
      fprintf(stderr, "usage:\n\t%s [-i <binary>] [-p <dif file>]\n", argv[0]);
      fprintf(stderr, "\t%s [-p <dif file>]\n", argv[0]);
      fprintf(stderr, "\t%s [-i <binary>] < <dif file>\n", argv[0]);
      fprintf(stderr, "\t%s < <dif file>\n", argv[0]);
      exit(0);
   }
}

if (patch == stdin) {
   fprintf(stderr, "Reading patch data from stdin.\n");
}
fgets(line, sizeof(line), patch); /* eat dif file intro line */
fgets(line, sizeof(line), patch); /* eat blank line */

if (input == NULL) {
   fprintf(stderr, "Inferring input file name from patch file data.\n");
        fscanf(patch, "%256s", line);
   input = fopen(line, "rb+");
   if (input == NULL) {
      fprintf(stderr, "Failed to open input file %s\n", line);
      exit(0);
```

```
      }
   }
   else { /* don't need input file name, but need to skip it in dif file */
           fgets(line, sizeof(line), patch);
       }

   while (fscanf(patch, "%x: %x %x", &offset, &orig, &newval) == 3) {
      fseek(input, offset, SEEK_SET);
      if (fgetc(input) == orig) {
              fseek(input, offset, SEEK_SET);
              fputc(newval, input);
      }
      else {
         //original bytes don't match expected?
      }
   }
   fclose(input);
   if (patch != stdin) {
      fclose(patch);
   }
}
```

After compiling this code successfully, execute the following command on DOS which requires the DIF file and new patched file name as:

Hide Copy Code

```
ida_patcher.exe -i PatchedApp.exe -p binaryCrackPatched.dif
```

This approach also fulfills the same objective and produces patched binary as defined earlier in the papers.

Final Notes

So, we have learnt one of the amazing tactics of patching the binary and producing new executable file using IDA Pro, for which crackers used to struggle because as of the IDA limitation, it can only disassemble the binary code to analyze the vulnerabilities or bugs in the code. We can temporarily divert the instruction code by modifying the ZF register during debugging just by placing the breakpoint but can't patch or modify the bytes sequence into the memory directly, to produce new binary executable because IDA Pro makes change in the database, not in the binary executable. This paper taught us how to make visible the hidden feature of Patch program in the IDA Pro IDE. We have also come across string patching and a new way to modify byte by producing DIF file which is later passed as an argument to a custom C patch code.

http://www.codeproject.com/Articles/833955/Binary-Cracking-Byte-Patching-with-IDA-Pro

加载两个 so 文件

Reload the input file
Action　　name: ReloadFile

This command reloads the same input file into the database. IDA tries to retain as much information as possible in the database. All the names, comments, segmentation information and similar will be retained.
Only the values of individual bytes will be changed.
This command works for some input file types only: if the file was loaded into the database with special settings, this command may fail. In this case, use Dump database to IDC file command and reload the file manually.
See also Load... submenu commands.

IDA 一日速成记

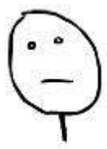
/*安全经典 如厕必备 此广告位长期招租*/

/*大家好我是@无所不能的魂大人，求关注求微博粉丝过百*/

0x0000 背景

事情大概是这个样子的，在那啥那啥那啥的冬季，一坨 CTF 联赛 正在如火如荼地进行中，上个月刚刚送走了溅得我一脸热翔的 hctf，这两天又迎来了卖萌又坑爹的 SCTF，连智商还有 IQ 还有智商直接被虐到了地里，算 了，不吐槽了，说正题，首先背景就是有个 SCTF，里面第二道题是个 100 分的 MISC 题，为啥是 MISC 而不是 RE 我也不太清楚，反正就是大概有那么一 个 ELF 文件，也就是 linux 下的可执行文件，是个贪吃蛇，大概题目的内容是这样（尼玛刚说去上面截个图结果发现网站关了）。就是有这么个贪吃蛇程序， 提示说吃到 30 的样子就可以知道答案了，但是，怎么控制它呢？

这种问题其实对于我这种 机智的少年而言太简单了，上面不是有提示么，UP-’8’、DOWN-’2’啥啥的，像我这种 IQ 过百的人玩儿这种游戏不是分分钟搞定的事儿么。于是我按 照提示，DOWN 是 2 对吧，2，2，2，22222222222222222222krelkur984375493irdjlsjiwury 32-03alshg49540 啪！啪！啪！



（此处省略换键盘买键盘等一系列过程）

然 后聪明的我很快就发现了原来这个程序的提示是骗人的，可是那该怎么办呢。虽然我的技术很渣，但也不至于啥都不懂，逆向，反编译这俩词儿我还是知道的，不过 以前都是粗略地看过一些用 W32ASM 暴力破解软件和用 OD 搞个啥的一些文章，可这是 ELF 文件不是 PE 文件啊，用这些东西到底能不能行该咋整呢。经常听 痛经大牛说什么 IDA，于是默默打开某厂主页，发现果然它可以用。

好，工欲善其事，必先利其器，工具找到了，这道题不就做出一半了么！

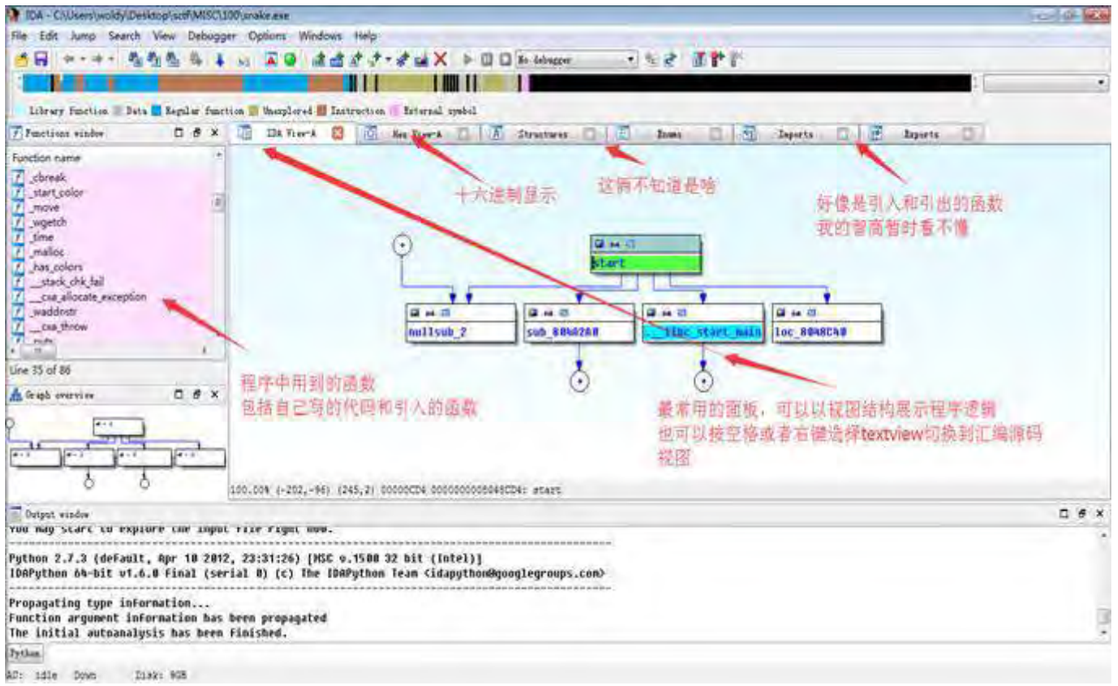
0x0001 初识

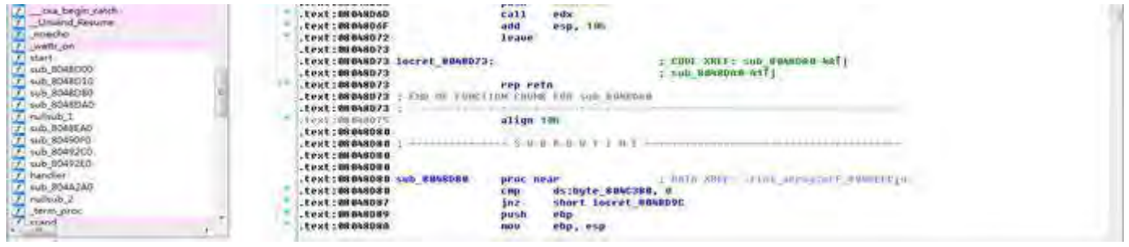
好的我们经过千辛万苦总算找到了一款相对比较新的 IDA，分享地址给出。http://pan.baidu.com/s/1mgza77i，友情提示安装完后请务必把里面那个小 rar 里的东西解压覆盖到程序目录，目测那个好像就是 F5 神器。

打开程序目录，很明显主程序就是这俩。

我是 64 位的 win7，当然要用 64 位的那个了！哎，没办法，我就是如此的机智……

什 么打开文件啥啥的我就不截图了，但凡智商比我差不了多少的人都会用，直接打开 snake.exe（题目中的是 snake-final.exe，痛经大神说 那个有壳，然后现在的 snake 就是被脱壳的 exe，为什么明明是个 elf 文件却要叫 exe 呢，我也不明白，我甚至不明白它为啥在 xp 下竟然可以运行）。

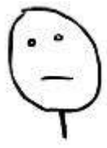




最后打开就是这个样子的，研究了好久才勉强研究明白主界面的这些东西大概都有啥用。然后接下来当然就是看代码了！

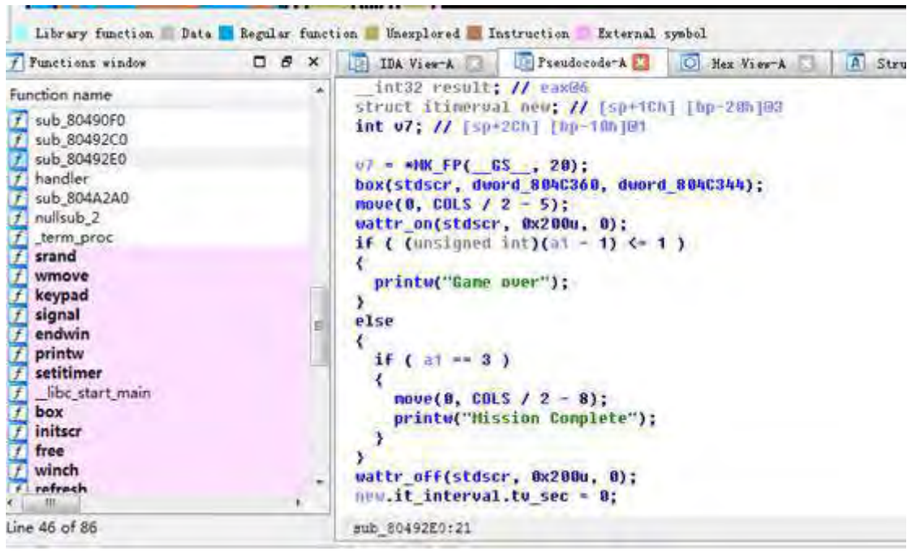
日啊！这尼玛都啥！再介绍下背景，其实我是一个 WEB 程序员，虽然我也会写代码，但是尼玛这都啥跟啥跟啥啊！智商明显跟不上了。

这个时候痛经牛又来拯救世界了，他说按 F5 可以把汇编代码还原成 C 语言，我一听卧槽牛逼啊，F5 是吧？F5，F5，F5，F5F6F7F8riejrkioekjr45094roedowsjf3=3ir3e2 啪！啪！啪！

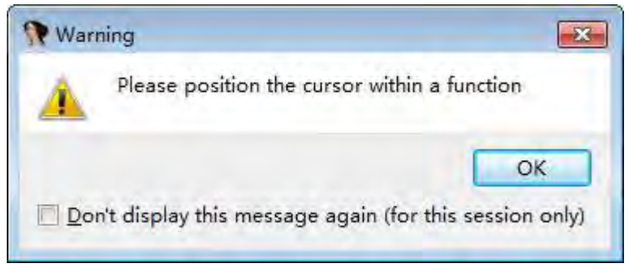


（此处省略换键盘买键盘等一系列过程）

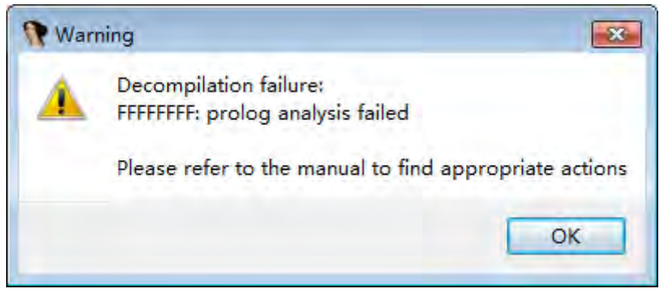
后来，发现貌似 IDA64 不支持 F5 欢迎成 C 这一功能。于是我默默打开了 idaq.exe，随便找了个函数，然后 F5。我靠果然好使！

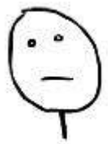


但是后来发现貌似不是所有的地方都能 F5 成 C 代码，比如会提示这个。



然后聪明的我马上就找到了其中的规律，嗯，只有函数是可以 F5 成 C 代码的。但是函数都有哪些呢？左边有。比如那个 printw，我双击它，然后 F5，他就会被反编译成……





额……后来，机智的我发现右边那个 functions window 可以拉伸！！

Function name	Segment	Start	Length	Locals
__cxa_end_catch	.plt	08048BB0	00000006	
_wattr_off	**.plt**	**08048BC0**	**00000006**	
_rand	**.plt**	**08048BD0**	**00000006**	
_curs_set	**.plt**	**08048BE0**	**00000006**	
__cxa_begin_catch	.plt	08048BF0	00000006	
__Unwind_Resume	.plt	08048C10	00000006	
_noecho	**.plt**	**08048C20**	**00000006**	
_wattr_on	**.plt**	**08048C30**	**00000006**	
start	.text	08048CD4	00000022	
sub_8048D00	.text	08048D00	00000004	00000000
sub_8048D10	.text	08048D10	0000002B	
sub_8048D80	.text	08048D80	0000001E	
sub_8048DA0	.text	08048DA0	0000002B	
nullsub_1	.text	08048DD0	00000002	
sub_8048EA0	.text	08048EA0	000001CF	0000003C
sub_80490F0	.text	080490F0	000001CC	0000001C
sub_80492C0	.text	080492C0	00000020	0000000C
sub_80492E0	.text	080492E0	00000165	0000003C
handler	.text	08049AE0	000004F3	000000A4
sub_804A2A0	.text	0804A2A0	00000061	0000002C
nullsub_2	.text	0804A310	00000002	
_term_proc	.fini	0804A314	00000014	0000000C
srand	**extern**	**0804C438**	**00000004**	
wmove	**extern**	**0804C43C**	**00000004**	
keypad	**extern**	**0804C440**	**00000004**	
signal	**extern**	**0804C444**	**00000004**	
endwin	**extern**	**0804C448**	**00000004**	
printw	**extern**	**0804C44C**	**00000004**	

它的 segment 属性可以分为 plt，text 和 extern 几个值，其中.plt 和.extern 我也忘了都是啥了之前百度过来着反正大概就是外部库引入的一些函数吧，只有.text 是真正的程序代码，所以 F5 只针对.text 的属性有效。

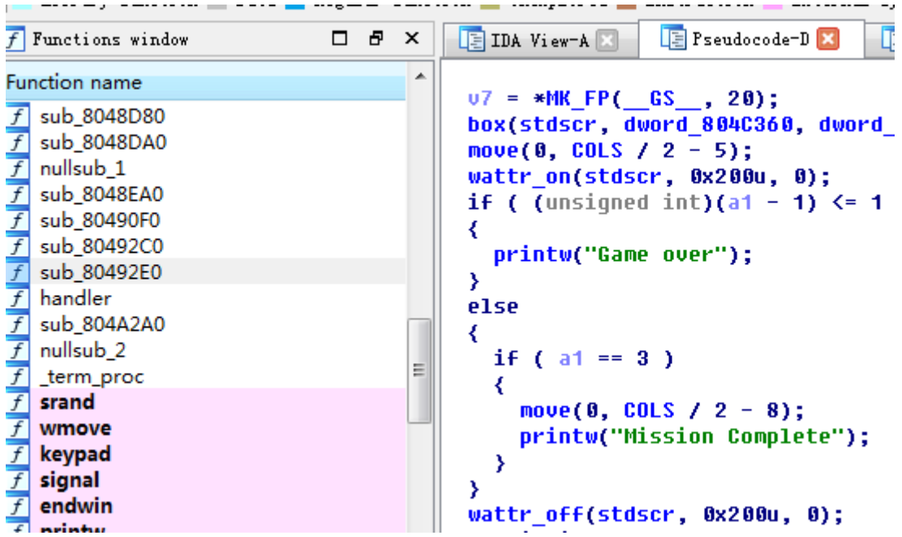
所以，理论上双击这里的任何 text 函数再 F5 都可以逆向成 C 代码。

嗯，有了这等神器，这题不就做出一半了么！

0x0002 爆破

//前面是 7 号写的。现在是 12 月 9 号 18:46，最近时间不大够……

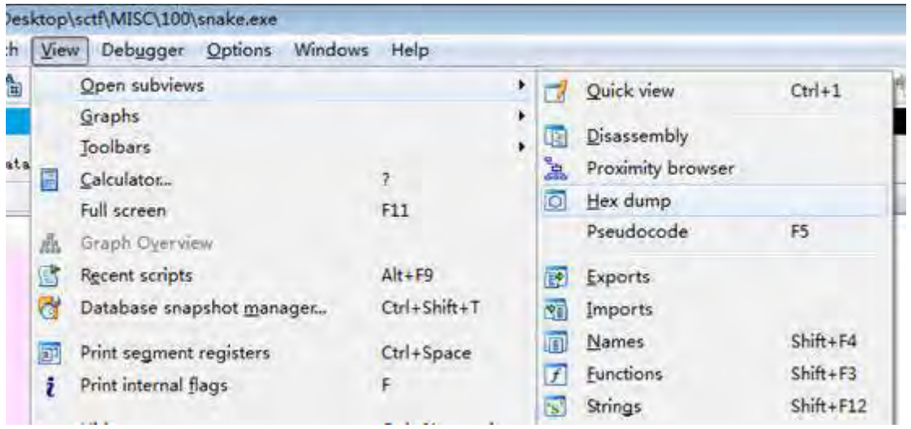
嗯，前面说到 F5 了对吧，确实挺牛逼的，于是我就在各个 function 上面 F5，然后就定位到了这里。



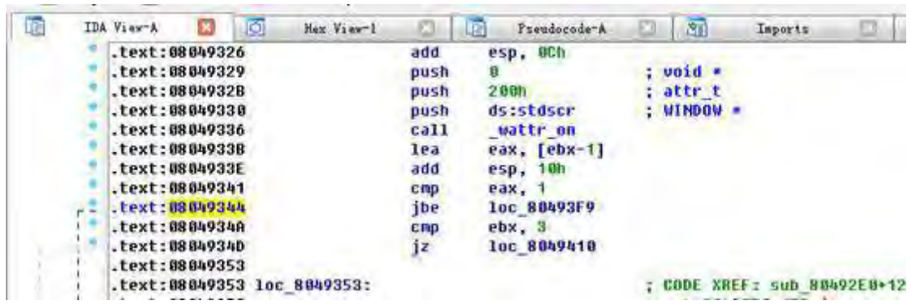
很 明显，这里是判断游戏最终是成功还是失败的地方，主要的判断点有两部分，首先在 if((unsigned int)(a1-1)<1)那里，把光标放在上面，按 tab（我也不知道为啥是按 tab 但是反正我就是莫名其妙知道按 tab 可以在 C 和 asm 之间切 换定位了），然后定位到了这里：

Cmp 肯定是 compare 的意思，就是个判断语句，下面的 jbe 虽然我不知道是啥意思，但是以我多年遭到的潜移默化的知识经验告诉我，凡是带 j 开头的都是和 jump 相关的，这句代码大部分都是代表 if(条件) jump 到某个程序段，不同的 jxx 代表不同的条件，所以我们可以查一下 jbe 是啥意思，http://ctf.idf.cn/index.php?g=portal&m=article&a=index&id=31 然后找到了这么一篇文章，搜 JBE 的时候可能会搜到两条，第一条是 8 位的，虽然不知道是干啥的但是貌似我们的是 32 位的，找到下一个搜索结果，16 位和 32 位的那里，可以看到说明是什么小于等于则跳，对应的机器码是 0F86，跟它相反的指令肯定是 JNBE ，不大于等于则跳，虽然我屡不清这其中的逻辑，但是大概应该是没问题的，机器码是 0F87。

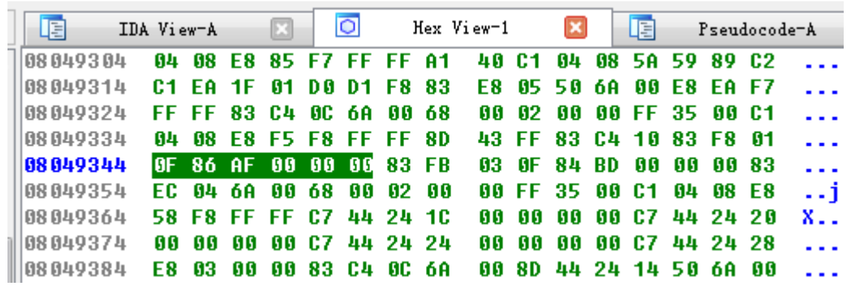
根据我零散的知识，所有的文件本质都是以 2 进制方式存储的，都有自己的格式。可执行文件的执行指令代码也一样，存在文件的某一块区域中，这些代码就是传说中的机器码，而机器码和 ASM 代码又基本是一一对应的。所以，下一步就是，从 ASM 定位到机器码，然后把文件对应的机器码部分修改掉。点开 view 菜单，open subviews，打开 Hexdump：



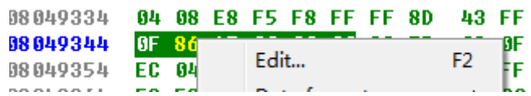
这样我们就会得到反汇编和机器码相对应的视图：



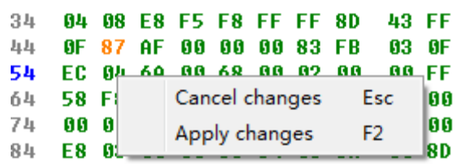
可以看到刚刚反汇编的跳转地址是 0809344，点到 Hex view 的视图，就会自动跳转到这里。



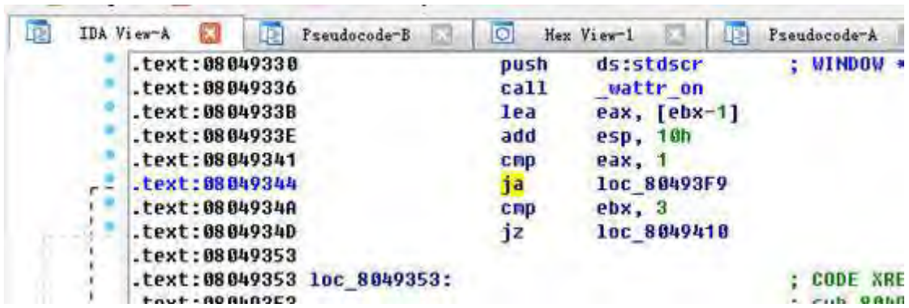
0F86AF000000 就是 jbe loc_80493F9 所对应的十六进制机器码，（PS：我也不知道为什么有时候直接切换视图就会自动定位并选中代码，有时候就不行，反正大概就是切换视图的时候还是要看看偏移是否对应的），刚才说了 0F86 就是 jbe，我们要把它改成 0f87 就会反向原来程序中的条件。也就是刚才反出来的那段 if((unsigned int)(a1-1)<1) 会变成 if((unsigned int)(a1-1)>=1)，在上面右键，点 EDIT：

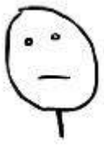


然后把 86 改成 87，然后继续右键，apply changes，当然也可以 F2。



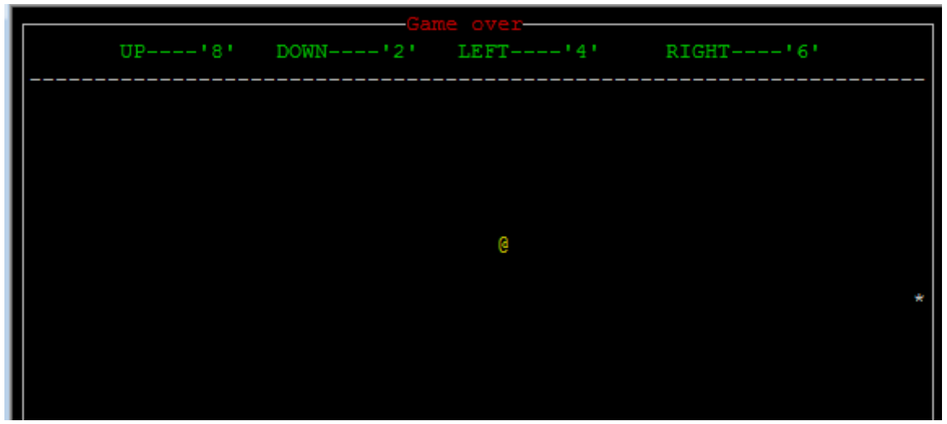
然后我们再回到 ida view，就是主视图去看我们改过的汇编代码，不出意外就会从 jbe 变成 jnbe 了，看！

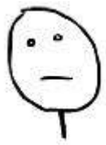




卧槽？ja 是个啥？后来又回到那篇文章看了看，jnbe 是不小于等于则跳，ja 是大于则跳，机器码都是 87，我读书少，但是发明汇编的人你这么逗我真的好吗……

因为刚才那段 C 代码明显看出是有两段对比，所以同理，在下面那个 if 也要改一下，就是 0804934D 那个 jz，等于则跳，改成 jnz，不等于则跳。机器码从 84 改成 85，F2 生效。好，保存我们的工程。理论上来讲只要运行我们的程序，当我们游戏结束的时候，就会出现 mission complate 字样，我简直太机智了，看！

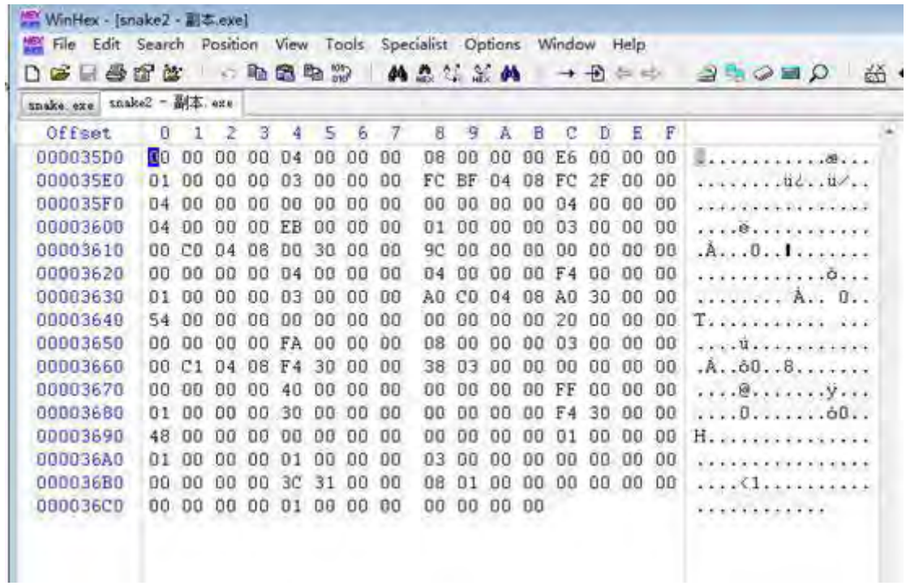


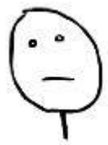


一定是我打开的方式不对……

咦？为啥修改日期还是没变？卧槽 IDA 你这不坑爹么我都把十六进制改了你居然没保存到文件！！

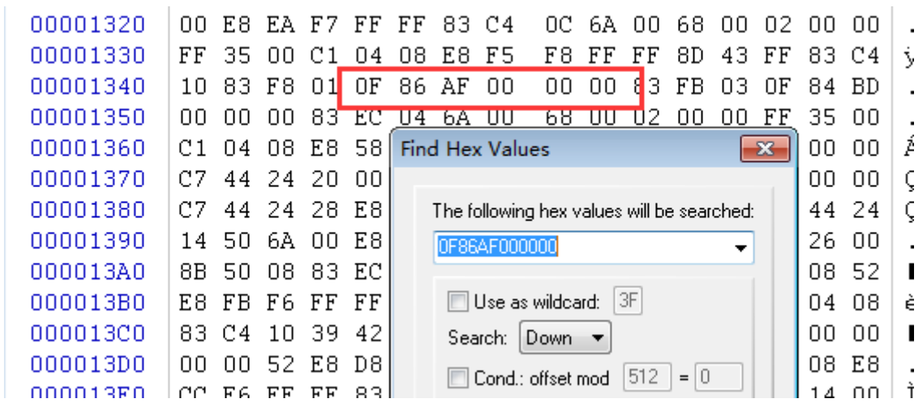
好吧，拖出一个副本，直接扔到 WinHex 中，08049344 是吧，我在 winhex 里面改一下不完了么。



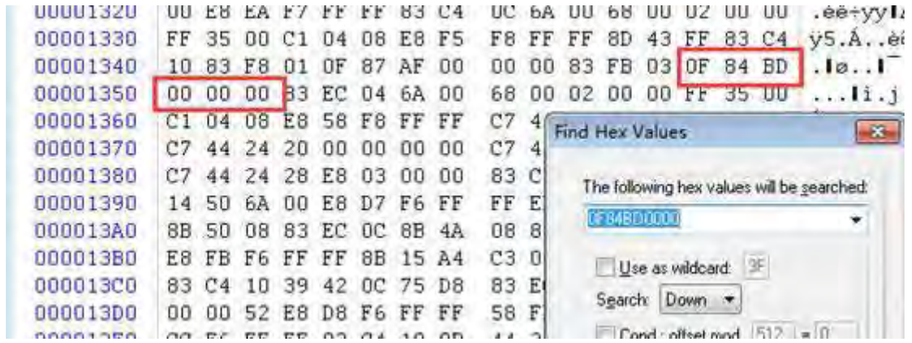


怎么感觉少了点什么，一定是我复制的不完整……

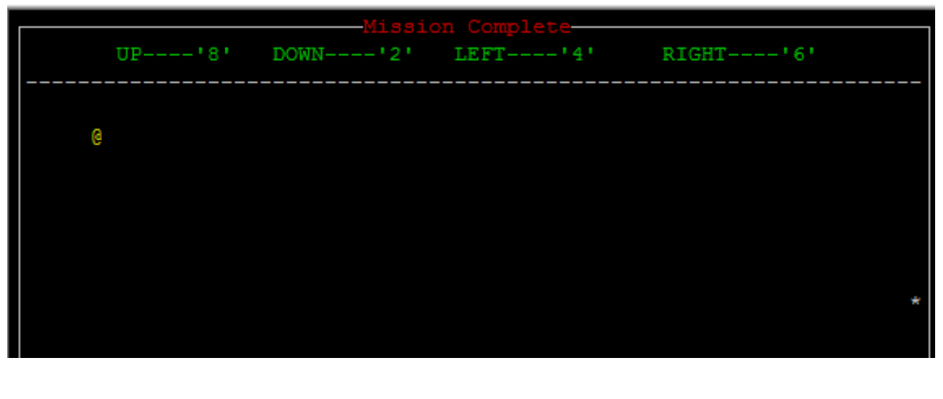
不过这难不倒机智的我，那段机器码是啥来着？0f86af000000 是吧。



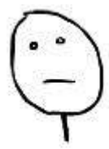
0f84bd000000 是吧？



全部改完，走你！



mission complate！肿么样，我是不是很厉害~这题太弱了！有能耐你们整四岁的！



说好的 flag 呢……

不过没关系，都已经搞出 mission complate 了，这题不就做出一半了么！

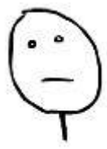
0x0003 篡改

（此处省略无数探索过程）

一般遇到这种情况，我都会想静静。

嗯，在 11 上被虐了无数盘之后，我又回到了贪吃虫上来，然后机智的我很快就想到，这个牛逼的贪吃蛇居然可以根据窗口的大小自动调整，我要把它高度改成 1 那岂不就是直接过关？走你~

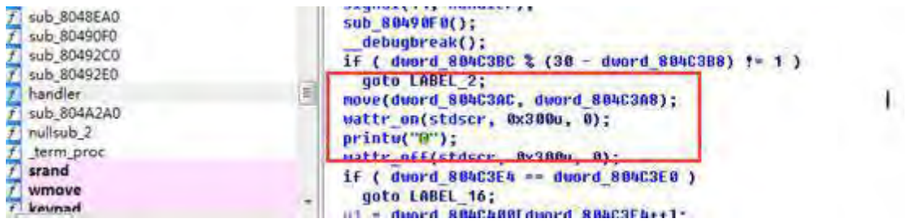
它咋不动呢……咋不动就 gameover 了呢。后来机智的我又发现一个秘密，就是每次的*号出现的位置都是固定的，都是大概第六七八行左右我数学不好反正就是那块儿吧，如果窗口过矮就会不动。



嗯，咦？那@的位置是咋写的呢？

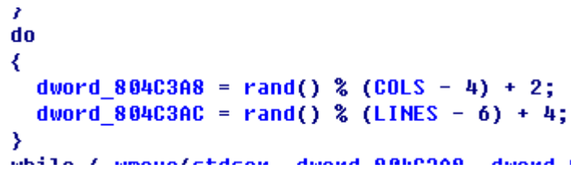
继续一点一点读代码。

然后我在 handler 的函数中看到了这一段代码。



聪明的我马上就猜到@的位置是根据那个啥啥 804C3AC 和 804C3A8 决定的，一个是 x 一个是 y。那么这俩变量是哪里的呢。

于是又在 hanlder 里找到了这个。



在 sub_804AEA0 里找到了这个，这个函数应该是初始化用的，即首次位置。

于是我突然想，去尼玛我想办法每次把@的位置放到*号前面不就完了么我真是太机智了我靠我自己都佩服自己！

804C3AC 肯定是 y，来我先把它固定。dword_804C3AC=rand()%(LINES-6)+4，

```
.text:08048FBE                call    _rand
.text:08048FC3                mov     esi, ds:LINES
.text:08048FC9                cdq
.text:08048FCA                mov     [esp+3Ch+timer], 10h ; size
.text:08048FD1                lea     ecx, [esi-6]
.text:08048FD4                idiv    ecx
.text:08048FD6                add     edx, 4
```

对应的汇编代码是这里，虽然我不懂汇编，但是 add eax,4 就是最后的那一步加 4 我还是能看出来的，然后我就机智地想到，把加 4 变成等于 4 不就可以了么，改成 mov edx,4，嗯，汇编代码编出来了，再来改对应的机器码。

```
08049F11  FC F7 F9 83 C2 02 89 15  A8 C3
08049F21  FF 8B 35 04 C1 04 08 99  83 EC
08049F31  83 C2 04 52 FF 35 A8 C3  04 08
08049F41  89 15 AC C3 04 08 E8 C4  EA FF
08049F51  FF 0F 84 60 FE FF FF 83  EC 0C
08049F61  E8 5A EB FF FF 83 C4 10  83 F8
```

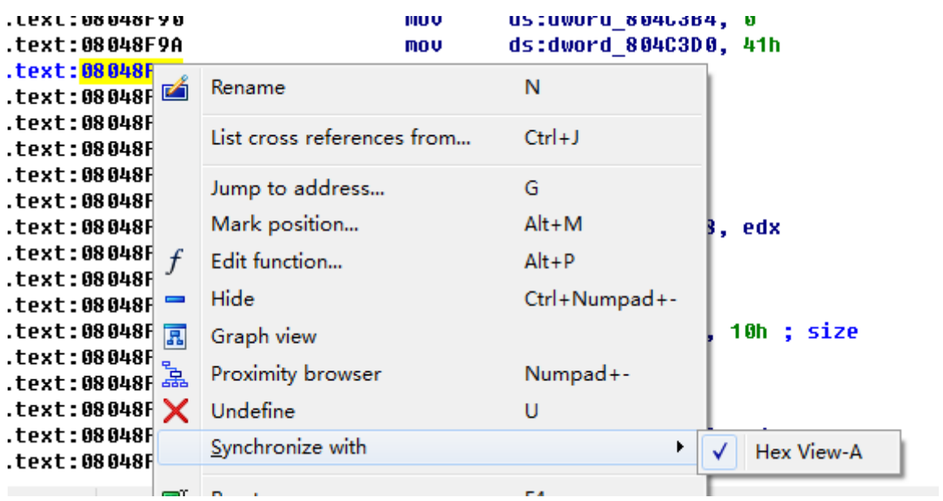
这个时候我突然想起来，小时候我好像是看过汇编的，好像什么从哪 mov 到哪事儿挺多的，刚才是 add 我改成 mov 会不会有些问题，虽然我还不知道 mov 的机器码是啥，于是找啊找啊找的终于找到了这个。

卧槽神器啊，but，为啥这么长……

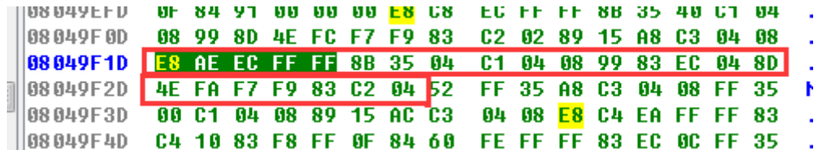
好吧我们再来看看源码。

```
.text:08048FBE                call    _rand
.text:08048FC3                mov     esi, ds:LINES
.text:08048FC9                cdq
.text:08048FCA                mov     [esp+3Ch+timer], 10h ; size
.text:08048FD1                lea     ecx, [esi-6]
.text:08048FD4                idiv    ecx
.text:08048FD6                add     edx, 4
```

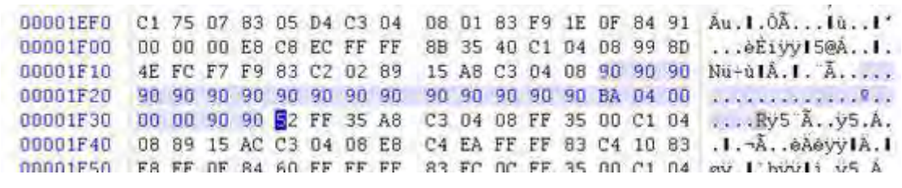
整个的计算流程大概是从 08048FBE 一直到 08048FD6，好，反正他们是连着的，只要我把任何一个地方改成 B804000000 然后两边 nop 掉不就可以了，咦？卧槽我在写这篇文章的时候突然学会怎么同步反汇编窗口和 hex 窗口了，这样。



嗯，然后继续说。我要改代码了，首先用 winhex 先打开 snake.exe，搜一下这两段之间的代码，大概就是下面这一段。



然后到 winhex 中，全他妈的改成 90，然后再加一个 BA04000000。

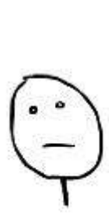


刚才说过这里有两段，一个是每次吃完@后重新生成的，这个在 hanlder 中，另一个是初始化的，在 sub_804AEA0 里面，修改同理。

```
08048F7E  C0 C3 04 08 01 00 00 00  C7 05 B0 C3 04 08 01 00  ..
08048F8E  00 00 C7 05 B4 C3 04 08  00 00 00 00 C7 05 D0 C3  ..
08048F9E  04 08 41 00 00 00 E8 27  FC FF FF 8B 0D 40 C1 04  ..
08048FAE  08 99 83 F9 04 F7 F9 83  C2 02 89 15 A8 C3 04 08  ..
08048FBE  E8 0D FC FF FF 8B 35 04  C1 04 08 99 C7 04 24 10  ..
08048FCE  00 00 00 8D 4E FA F7 F9  83 C2 04 89 15 AC C3 04  ..
08048FDE  08 E8 5C FB FF FF C7 04  24 10 00 00 00 89 C3 E8  ..
```

```
- [snake - 副本.exe]
dit  Search  Position  View  Tools  Specialist  Options  Window  Help
本. exe
t    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
B0   83 E9 04 F7 F9 83 C2 02  89 15 A8 C3 04 08 90 90  ▌é.÷ù▌
C0   90 90 90 90 90 BA 04 00  00 00 90 90 90 90 90 90  .....º
D0   90 90 90 90 90 90 90 90  90 89 15 AC C3 04 08 E8  ......
E0   5C FB FF FF C7 04 24 10  00 00 00 89 C3 E8 4E FB  \ûÿÿÇ.
F0   FF FF C7 04 24 10 00 00  00 89 C6 A3 A4 C3 04 08  ÿÿÇ.$.
```

```
*** glibc detected *** ./snake_-_副本.exe: free(): invalid next size (fast): 0x09e78e20 ***
                                                                                Aborted
[root@10-9-9-106 wolay]#
```



卧槽哪里不对么……

对比了好几遍，还是不明白为啥会这样，不管那么多了，回去再看一看代码：

```
.text:08048FB3                 idiv    ecx
.text:08048FB5                 add     edx, 2
.text:08048FB8                 mov     ds:dword_804C3A8, edx
```

从 idiv 到 add 这两行的代码正好对应五个字节：

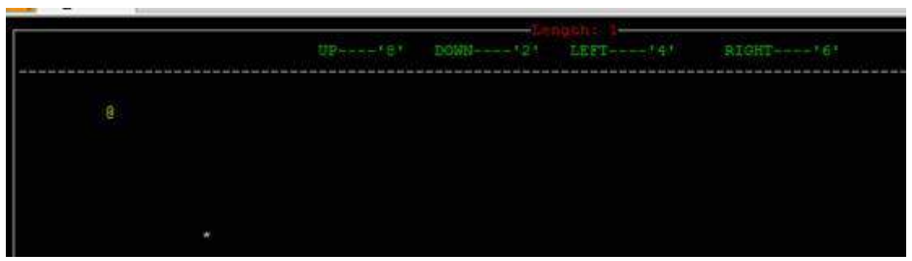
```
08048FA3  00 E8 27 FC FF FF 8B 0D
08048FB3  F7 F9 83 C2 02 89 15 A8
08048FC3  8B 35 04 C1 04 08 99 C7
```

直接把 F7F983C204 改成 BA0400000。

切回代码，可以看到，dword_804C3AC 变成 4 了，卧槽我好厉害！

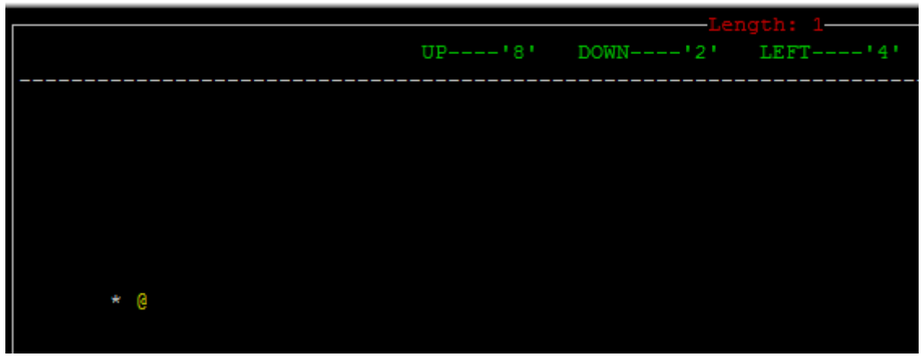
```
dword_804C3C0 = 1;
dword_804C3B0 = 1;
dword_804C3B4 = 0;
dword_804C3D0 = 65;
dword_804C3A8 = rand() % (COLS - 4) + 2;
rand();
dword_804C3AC = 4;
v2 = malloc(0x10u);
```

跑一下看看，看到那个@了没，果然是在第四行！！



现在我把 04 改成 0A,也就是每次都出现在第 10 行，当然前面那个 BA0400000 也要改。

OK，没问题了，这样每次的高度就固定住了，我们再来研究研究 x 轴坐标，首先初始化的坐标我们可以改一下，嗯，就改成 10 好了。方法很简单，和上面同理，在 sub_804AEA0 里面找到 dword_804c3a8 对应的汇编代码，也是 idiv 啥的，对应的机器码是 F7F983C202，改成 ba0a000000，不过直接在 winhex 里搜的话肯定会搜到两个结果，因为两个地方代码是相同的，我们只要改初始化的那里就行了，区分方法也很简单，搜索前面的机器码就可以了，04089983E904，然后把后面改成 BA0A000000。



成功！，但是吃完了这个@之后，后面的@的 x 位置就不受控制了，有时候甚至会跑到*的前面，咋整呢？想了想，让 dword_804c3a8 每次的值都自动+1 不就完了！

但是，卧槽汇编代码咋写……

不过 ida 帮我逆出来了这么多的汇编代码，难道我抄还不会么……咦？这里有一段。

```
if ( v5 > 60 )
{
  ++dword_804C3CC;
  dword_804C3C8 = 0;
}
```

看看汇编代码咋写的：

```
.text:08049187                  jle     short loc_804919A
.text:08049189                  add     ds:dword_804C3CC, 1
.text:08049190                  mov     ds:dword_804C3C8, 0
```

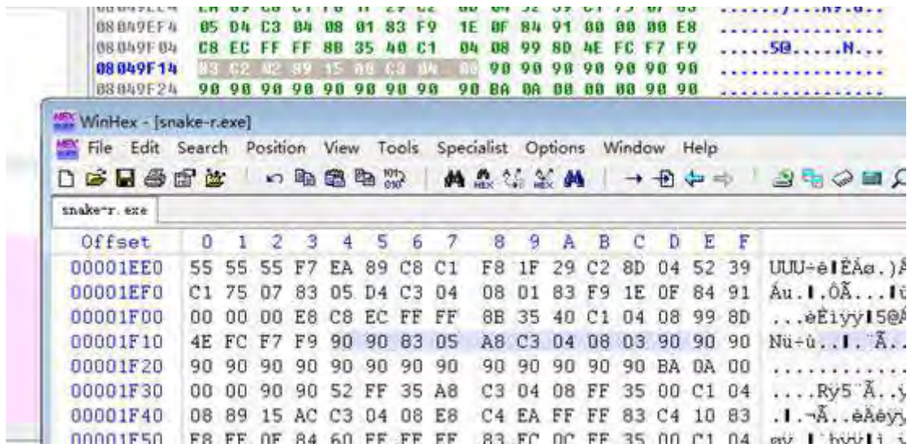
再看看机器码是啥。

```
08049179  0F 8F 21 01 00 00 A1 C8
08049189  83 05 CC C3 04 08 01 C7
08049199  00 A1 40 C1 04 08 83 EC
```

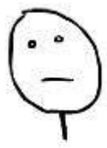
嗯，大概七个字节，8305 肯定是 add 啥啥的了，后面是地址，最后一位是 1。

找到 hanlder 的 function，找到这个 add 和 mov 啥啥的，直接跳转到机器码，把它改成 8305 啥啥啥的就 ok 了。

```
.text:08049F0F                  lea     ecx, [esi-4]
.text:08049F12                  idiv    ecx
.text:08049F14                  add     edx, 2
.text:08049F17                  mov     ds:dword_804C3A8, edx
.text:08049F1D                  nop
```



最终看看效果：

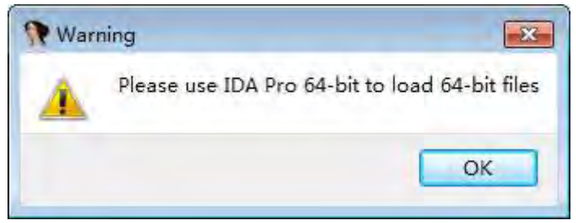


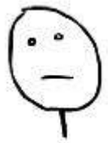
成功！不过，说好的 flag 呢……

0x0004 RE50

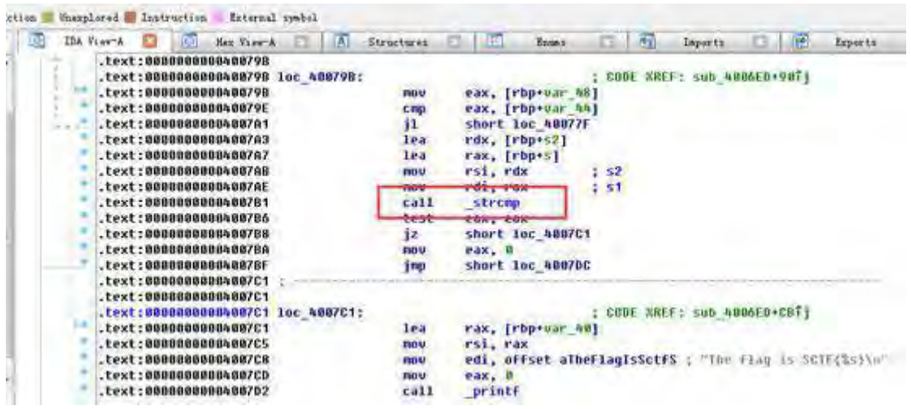
话说这个贪吃蛇严重打击了我的自信心，让我对 CTF 这个比赛彻底失去了信心，就在这个时候，一道 RE50 的题横空出世，就像那啥中的那啥啥给找指明了方向，100 分的题我不会，50 分的题还能难道我吗！

直接下载文件，打开。



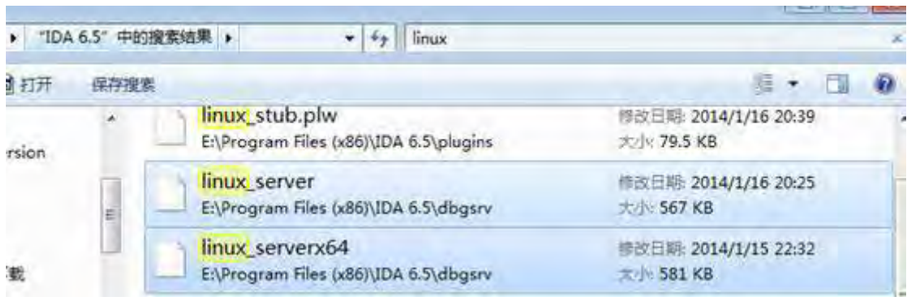


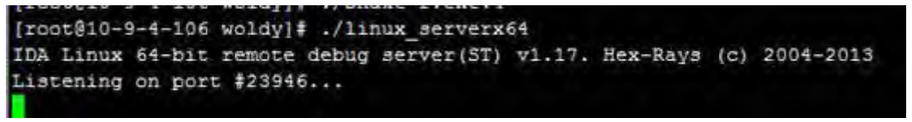
一定是我打开的方式不对，换 64 位打开。



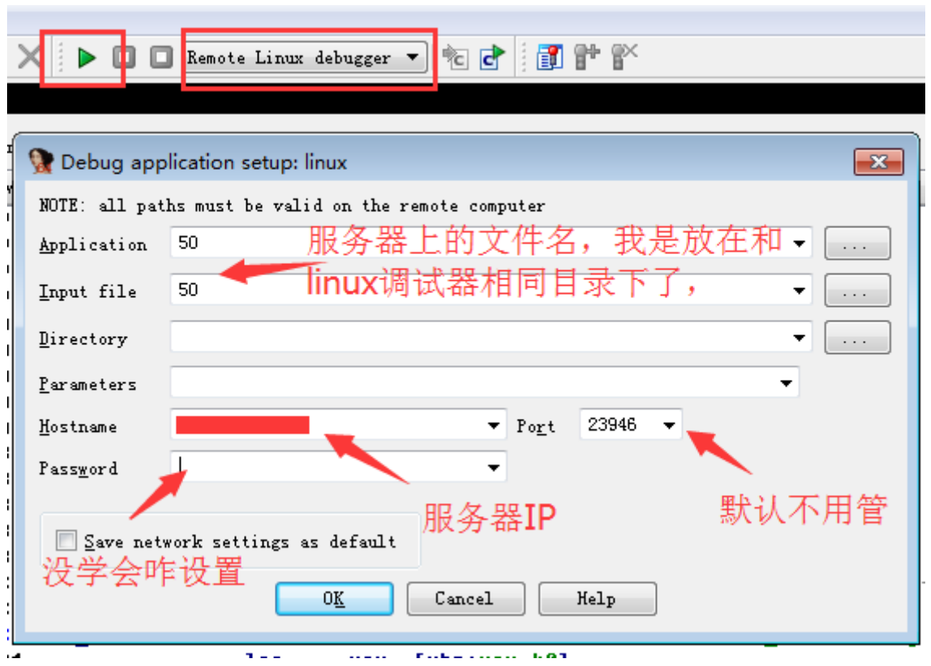
首先映入眼帘的就是这一坨东西，虽然在 64 位下没办法 F5，不过 strcmp 我还是认识的，字符串比较么！这个时候要是能动态调试的话直接看看他们的值就好了……嗯？动态调试？

百度了一下，神器的 IDA 果然是可以动态调试的，不过都没咋看明白，最后艰难地学会了一个最简单的方法。就是在 ida 安装目录下找到这俩文件，分别对应 x86 和 x64 版，拖到 linux 服务器上运行就 OK 了。

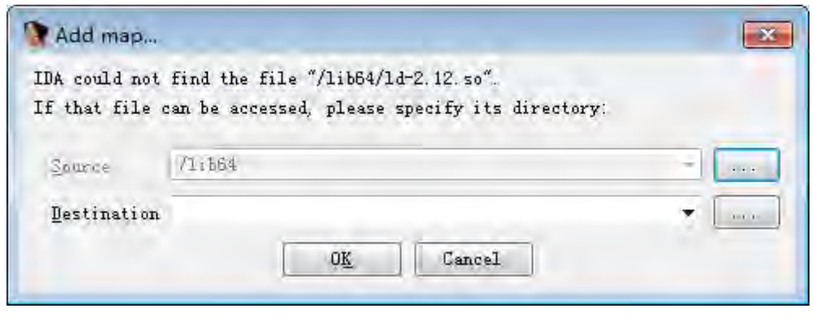




然后在对应的 IDA 客户端上选择好调试方式，点那个绿色的按钮。

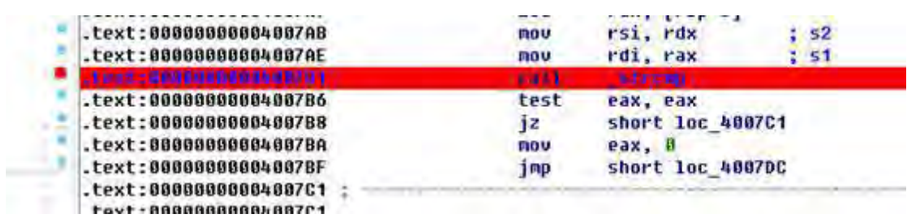


OK 之后走你，会弹出对话框让你设置一堆东西。然后不停取消，好像也没啥影响。



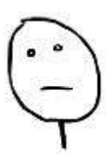
接着，查看远程 shell，程序在远程成功执行了。

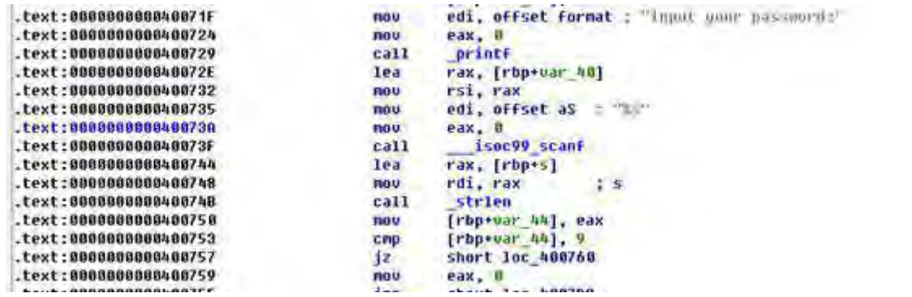
随便输入了一坨东西，然后就结束了。好，现在我们在 strcmp 那里打个断点，虽然不太懂但是多年的程序员生涯还是让我直觉地感到这是有用的。



输入一坨东西，这个时候我们就会来到断点处去看他们的字符串对比，卧槽咋又退出了？

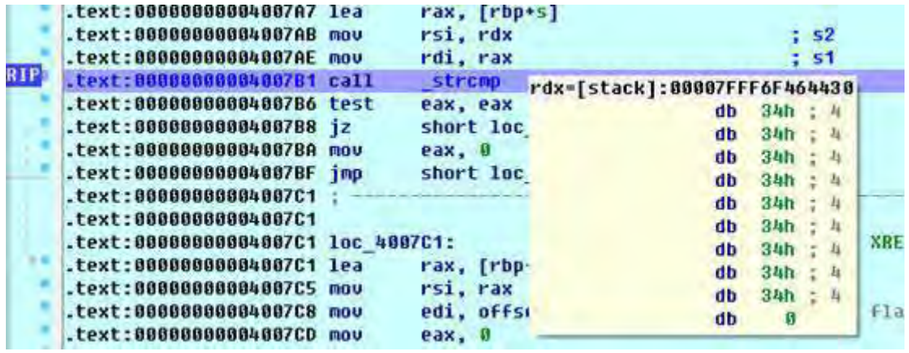
往上翻代码，看到了这里。

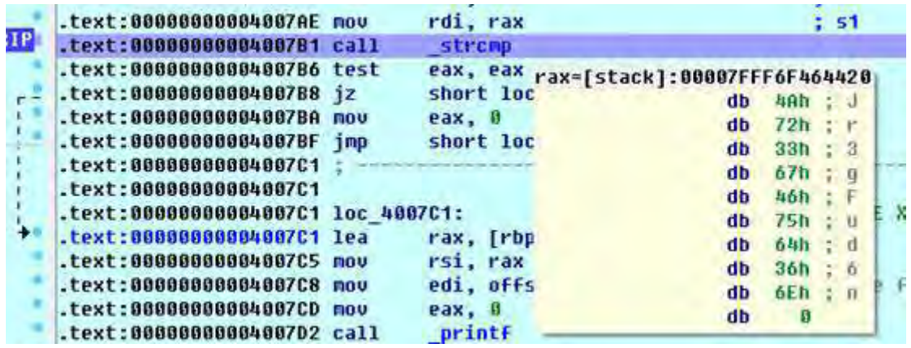




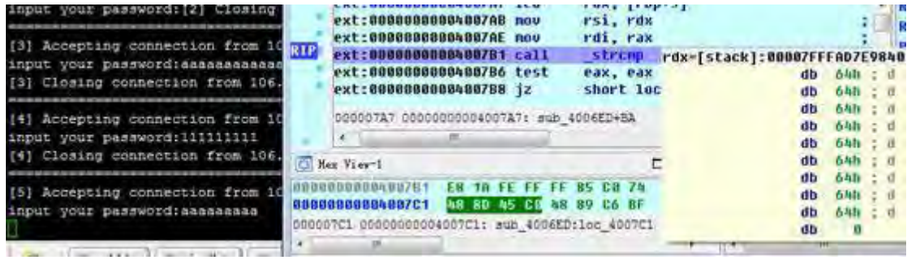
输完密码后有个啥啥 cmp 9 的，难道长度是九位？我试试。

输入了 9 个 1，果然断到 strcmp 处了，看看是咋 cmp 的。



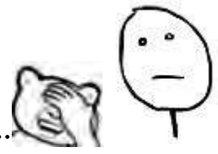


看了看离得最近的两个变量，一个是 9 个 4，另一个是 Jr3gFud6n，很明显那 9 个 4 就是我之前那 9 个 1，尼玛，谁让你给我变的，1 变成 4，我要是输入 9 个 a 你岂不是还要给我变成 9 个 d？



我读书少，这是凯撒加密么？

最后，我又花了半个小时的时间来破解 ascii 码加 3 过的 Jr3gFud6n……



0xffff 后记

卧槽真不敢相信我废话居然这么多，一个贪吃蛇我都能 BB 将近五千字，主要是第一次接触 IDA 这么高大上的东西玩儿激动了，真是第一次用，所以文章写的哪里不对请不要骂我，有能耐你拿辣条来抽我啊！我键盘两千多呢……看我 BB 了这么久多辛苦，一百块钱都没有人给我，还打我，我跟你们什么怨什么仇啊…… 

Made in China by woldy，@无所不能的魂大人

公元 2014 年 12 月 10 日 2:00

PS：除 IDA 外本文所提到文件地址在此→点击下载

（全文完）

http://ctf.idf.cn/index.php?g=portal&m=article&a=index&id=33

SCTF---- RE50 静态分析

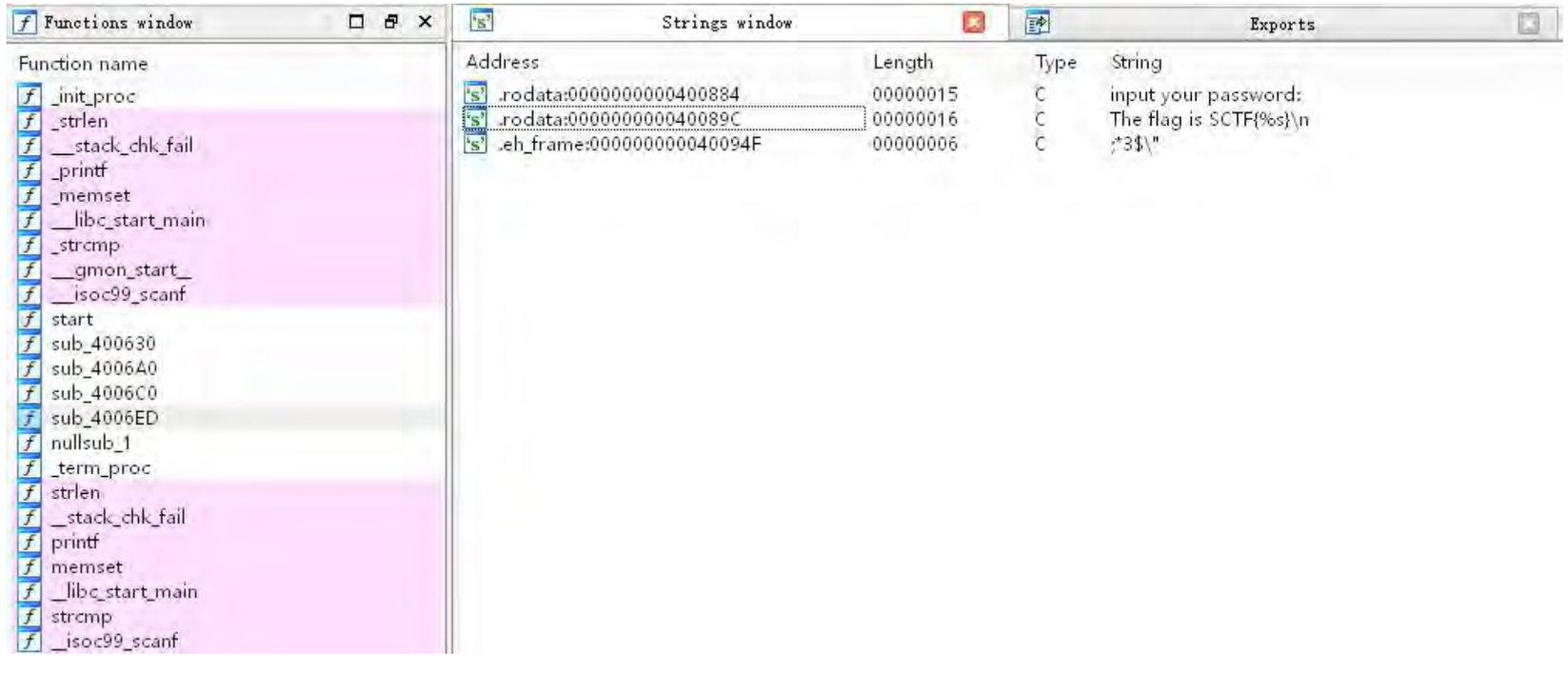
http://ctf.idf.cn/index.php?g=&m=article&a=index&id=34

天孤剑同学在《IDA 一日速成记》一文中介绍了 IDA 的使用方法，并对贪吃蛇和 RE50 进行了逆向分析。本文也是针对 RE50 的逆向分析，与天孤剑动态调试不同的是这里采用静态分析的方式，尝试通过汇编分析来还原出对应的 C 代码，希望对大家能有所帮助。

P.S. 许多 writeup 都给出了 RE50 的逆向分析且直指关键，但对于像我这样的小白，看到分析后只能是不明觉厉，下次遇到同样的问题可能依旧束手无策，所以本文不仅会介绍 RE50，而且还会更多地去介绍关联知识，这样后续对于逆向分析也许能更快地找到切入点。

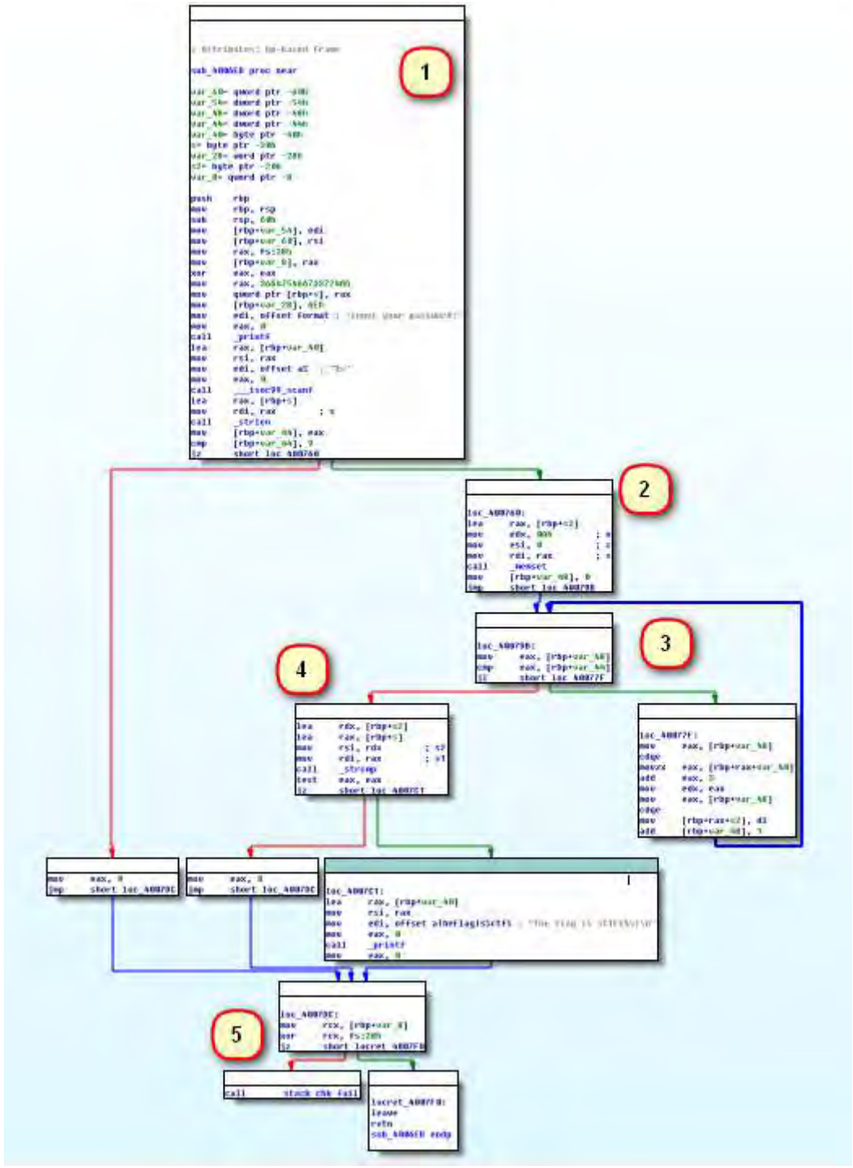
IDA 绝对是逆向分析的神器，二进制文件丢进去后直接 F5 反编译，后续的任何问题基本就很容易分析了。不过大家手头的 IDA 很多都是 6.6 之前的版本，这些版本 不支持 64 位程序的反编译，所以遇到 64 位程序时只能硬着头皮看汇编代码了。（IDA 6.6 有谁能提供一个不 ^_^）

废话少说，通过 IDA64 打开 RE50，通过 View->Open subviews 我们可以查看文件中的 functions、strings 信息：



RE50 symbols

"The flag is SCTF..."这绝对是敏感信息，我们可以查找哪个函数中使用了它，然后定位到了 sub_4006ED。IDA 进行汇编分析时，通过空格可以在汇编视图和 Graph 视图间切换，sub_4006ED 的逻辑图如下：



RE50 graph

sub_4006ED 的逻辑结构比较简单，这里将它拆分为 5 部分进行分析。当你去仔细查看和思考时，你会收获更多 :-)

在分析(1)之前，我们先来看看函数调用和栈空间等相关概念：

这里的栈是指函数执行时的栈空间，很多人都知道栈相关的概念，但有没思考过为什么用栈呢？

其实它与我们的编程语言紧密相关，如何限定局部变量的作用域、如何实现函数调用、如何解决递归问题，栈模型给出了很好的答案。

我们要知道是：

a.函数进行调用时，栈空间会扩张，函数结束时，栈空间会收缩

b.函数内部的局部变量，几乎全部都保存在栈上（static 的除外）

c.栈空间的平衡，由函数自身进行保障，与调用者无关

d.进入函数调用时的 call 指令会将下一条指令地址(EIP)保存在栈上

我们回到(1)，汇编如下：

```
sub_4006ED  proc  near

var_60=  qword  ptr  -60h                                    ---->IDA 生成的伪代码，用于索引当前栈帧的偏移
var_54=  dword  ptr  -54h
var_48=  dword  ptr  -48h
var_44=  dword  ptr  -44h
var_40=  byte  ptr  -40h
s=  byte  ptr  -30h
var_28=  word  ptr  -28h
s2=  byte  ptr  -20h
var_8=  qword  ptr  -8

push        rbp                                              ---->将调用者的栈底指针保存在栈上
mov         rbp,  rsp                                        ---->开辟当前函数的栈帧
sub         rsp,  60h                                        ---->开辟当前函数的栈空间，栈空间为 60 字节
mov         [rbp+var_54],  edi
mov         [rbp+var_60],  rsi
mov         rax,  fs:28h                                     ---->在第五部分会提到它
mov         [rbp+var_8],  rax
xor            eax,  eax
mov         rax,  366475466733724Ah                          ---->在 rbp+s 和 rbp+var_28 处，写入一些数据
mov         qword  ptr  [rbp+s],  rax
mov         [rbp+var_28],  6Eh
mov         edi,  offset  format  ;  "input  your  password:"
mov         eax,  0
call           _printf
lea            rax,  [rbp+var_40]                            ---->获取用户输入字符串，保存在 rbp+var_40 处
mov         rsi,  rax
mov         edi,  offset  aS    ;  "%s"
mov         eax,  0
call           ___isoc99_scanf
lea            rax,  [rbp+s]                                 ---->获取 rbp+s 处字符串的长度（这里可能是出题者弄错了）
```

```
mov         rdi,  rax                ;  s
call            _strlen
mov          [rbp+var_44],  eax
cmp          [rbp+var_44],  9
jz                  short  loc_400760
```

我们书写自己的函数，对上面代码进行还原：

```
int  sub_4006ED()
{
        char  s2[16];
        char  s[16];
        char  var_40[16];
        short  *var_28  =  (short  *)(s  +  8);
        int    var_44,  var_48;

        *(long  long  *)s  =  0x366475466733724A;
        *var_28  =  0x6e;

        printf("input  your  password:");
        scanf("%s",  var_40);
        var_44  =  strlen(s);
        if  (var_44  ==  9)  {

        }

        return  0;
}
```

栈上 buf 的大小取决于函数内部的局部变量:

1. 用 s[16]，并假设指针 s 对应栈上地址 rbp-30。var_28 - s = 8，那么 var_28 对应 s + 8。
2. 用 var_40[16]，并假设指针 var_40 对应栈地址 rbp-40。scanf 输入的字符串保存在 var_40 指向的 buf 中。
3. 经过前面的设置，strlen(s)结果保存到 var_44 中，它必然等于 9（除非 scanf 输入的数据超过 16 字节，将 s 尾部写坏了），所以我觉得这里是否是 strlen(var_40)，出题时这里弄错了吗？

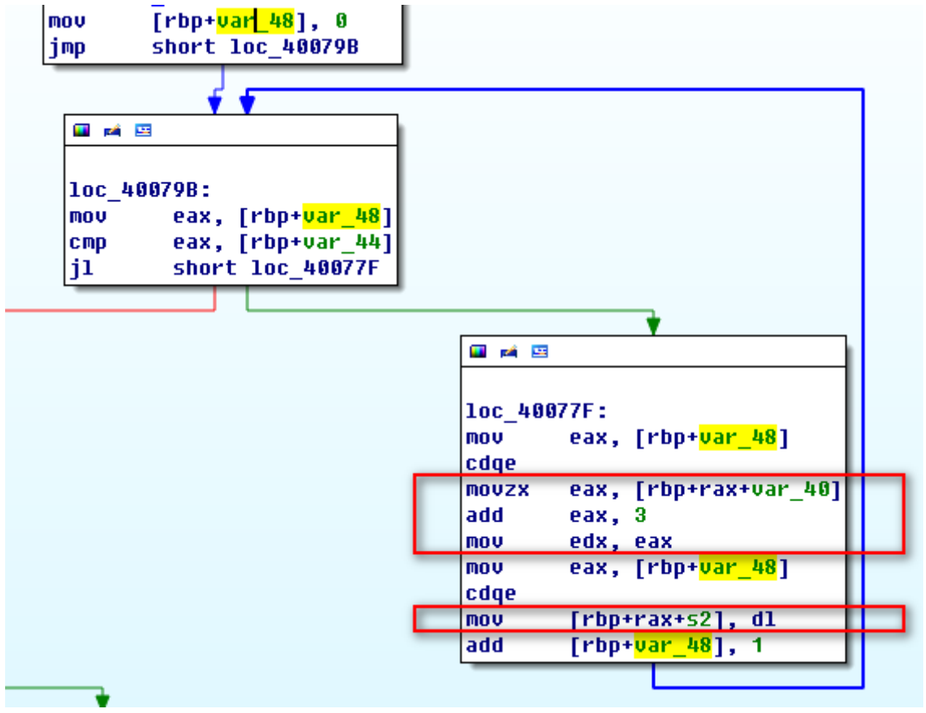
接下来我们看(2)，汇编如下：

```
loc_400760:                                             ;  CODE  XREF:  sub_4006ED+6Aj
                                lea          rax,  [rbp+s2]
                                mov          edx,  0Ah               ;  n
                                mov          esi,  0                   ;  c
                                mov          rdi,  rax               ;  s
```

```
call        _memset
mov         [rbp+var_48],  0
jmp         short  loc_40079B
```

它的作用是调用 memset 函数，其中内存地址为 rbp+s2，设置值为 0，长度为 10，即 memset(s2, 0, 0xa)。然后它将 var_48 设置为 0。

第(3)部分稍微长一点，也是很多 writeup 中指明的关键部分，这里我们看它的逻辑图进行分析：



var_48 初始化为 0，若小于 var_44（前面设置为了 9），则将 var_40 中的字符拷贝到 s2 中，然后 var_48 加一。
仔细思考下，这不是一个 for 循环吗：

```
for  (var_48  =  0;  var_48  <  var_44;  var_48++)
        *(s2+var_48)  =  *(var_40+var_48)  +  3;
```

ok，我们继续看第(4)部分：

```
lea         rdx,  [rbp+s2]
lea         rax,  [rbp+s]
mov         rsi,  rdx               ;  s2
mov         rdi,  rax               ;  s1
call        _strcmp
```

```
                                test        eax,  eax
                                jz            short  loc_4007C1
                                mov          eax,  0
                                jmp          short  loc_4007DC
```

很简单，比较字符串 s 与 s2，若 s 与 s2 相等，则跳转到最后的 printf 进行输出，若不相等，则返回。

至此，汇编基本分析完成，我们可以写出 sub_4006ED 的代码如下：

```
int  sub_4006ED()
{
        char  s2[16];
        char  s[16];
        char  var_40[16];
        short  *var_28  =  (short  *)(s  +  8);
        int    var_44,  var_48;

        *(long  long  *)s  =  0x366475466733724A;
        *var_28  =  0x6e;

        printf("input  your  password:");
        scanf("%s",  var_40);

        var_44  =  strlen(s);

        if  (var_44  ==  9)  {
                memset(s2,  0,  0xa);
                for  (var_48  =  0;  var_48  <  var_44;  var_48++)
                        *(s2  +  var_48)  =  *(var_40  +  var_48)  +  3;

                if(strcmp(s2,  s)  ==  0)
                        printf("The  flag  is  SCTF{%s}\n",  var_40);
        }

        return  0;
}
```

等等，第(5)部分呢？

```
loc_4007DC:                                                ;  CODE  XREF:  sub_4006ED+71j
                                                                ;  sub_4006ED+D2j
                                mov          rcx,  [rbp+var_8]
                                xor          rcx,  fs:28h
```

```
                                jz          short  locret_4007F0
                                call        ___stack_chk_fail
```

还记得(1)在开辟栈空间后做的一个特殊处理吗：

```
mov         rax,  fs:28h
mov         [rbp+var_8],  rax
```

我们都知道栈溢出漏洞，其最直观的利用方式就是覆盖栈顶的 EIP 指针，使其指向 JMP ESP 指令。其原理是：当函数返回时 retn 会 pop 前面压入栈中 EIP 的指针，通过栈溢出使得 EIP 被覆盖（由输入控制），从而实现对程序流程的劫持（比如 JMP ESP 会跳转到栈上的 shellcode 区域进行执行）。

这里通过将 fs:28h 处的随机值写到栈上，在函数返回时进行栈上保存值的校验，若栈被写坏那么这里的值会发生变化，会进入 stack_check_fail 函数做异常处理，达到对栈溢出攻击的防护。

P.S. 栈溢出检查是由编译器自动完成的，并不需要在代码中显示书写。

关于栈溢出攻击，可参考这篇文章：http://my.oschina.net/sincoder/blog/115944

受文章篇幅限制，许多细节处没进行扩展，读者若感兴趣可进行自行查阅，我觉得比较有趣的是 Linux 中的栈模型（栈的扩展和收缩，如何保持栈平衡等）、x64 相关的汇编指令和寄存器信息等。如果文章中有不对的地方，也欢迎大家进行指正，谢谢~

SCTF 相关题目和程序，我们进行了收集整理，见这里：http://pan.baidu.com/s/1eQCeitW

P.S. IDF 实验室目前在收集整理 CTF 相关的工具，感兴趣的朋友可以看这里：https://github.com/woldy/CTF-Tools
你可以将自己喜欢的工具告知我们，也可以将安装包或下载地址发送给我们。我们收集整理后，会维护一份 list，并将程序放到百度网盘中提供给对 CTF 感兴趣的小伙伴们~

Author：IDF 实验室 cumirror

Mail：tongjinam#qq.com

（全文完）

扩展 IDA 的知识

第 1 章 扩展 IDA 的知识

现在应该明确的是，高质量的反汇编远远超出了只对由字节序列生成助记符和操作数的列表。为使反汇编发挥更大的作用，我们需要用在处理各种与 API 有关的数据（如函数原型和标准数据类型）时获得的信息来扩充反汇编代码清单。在第 8 章中，我们讨论了 IDA 如何处理数据结构，包括如何访问标准 API 数据结构和如何定制数据结构。本章通过分析 IDA 的 idsutils 和 loadint 实用工具的用法，继续讨论 IDA 知识的扩展。你可以从 IDA 的产品光盘上获得这些实用工具，或从 Hex-Rays 的下载站点下载1[1]。

1[1]请参阅 http://www.hex-rays.com/idapro/idadown.htm；下载时需要提供有效的用户名和密码。

1.1 增强函数信息

IDA 获取了解函数的两个来源：类型库（.til）文件和 IDS 实用工具（.ids）文件。在初始分析阶段，IDA 使用存储在这些文件中的信息来提高反汇编过程的准确性及反汇编代码清单的可读性。它通过合并函数参数名称和类型，以及与各种库函数有关的注释完成这个任务。

第 8 章提到，类型库文件是 IDA 用于存储复杂数据结构布局的机制。同时，IDA 还使用类型库文件记录与函数调用约定和参数有关的信息。IDA 以各种方式使用函数签名信息。首先，当一个二进制文件使用共享库时，IDA 无法知道这些库中的函数使用的是什么调用约定。这时，IDA 会尝试根据一个类型库文件中的相关签名来匹配库函数。如果它发现一个匹配的签名，IDA 就可以知道这个函数使用的调用约定，并对栈指针进行必要的调整（如前所述，stdcall 函数自己对栈进行消理）。使用函数签名的第二种方式，是为传递给函数的参数提供注释。这些注释说明在调用函数之前，到底是哪一个函数被压入到栈上。注释提供的信息量，取决于 IDA 能够解析的函数签名所包含的信息量。下面两个签名都是有效的 C 声明，第二个签名提供了更多有关函数的信息，除数据类型外，它还提供形式参数名称。

```
LSTATUS _stdcall RegOpenKey(HKEY, LPCTSTR, PHKEY);
LSTATUS _stdcall RegOpenKey(HKEY hKey, LPCTSTR lpSubKey, PHKEY phkResult);
```

IDA 的类型库包含了大量常用 API 函数，其中包括许多 Windows API 的签名信息。调用 RegOpenKey 函数默认的反汇编如下所示：

```
.text:00401006    00C  lea eax, [ebp+YhKey]
.text:00401009    00C   push    eax       ✍; phkResult
.text:0040100A    010   push    offset ✍SubKey     ; "Software\\Hex-Rays\\IDA"
.text:0040100F    014   push    80000001h✍; hKey
.text:00401014    018   call    ds:RegOpenKeyA
.text:0040101A ✍00C   mov     [ebp+var_8], eax
```

值得注意的是，IDA 已经在右边缘✍添加了注释，指出在调用 RegOpenKey 之前，每个指令压入了什么参数。如果函数签名提供形式参数名称，那么 IDA 会更进一步，自动为与特定的参数对应的变量命名。在前面例子中的✍处，我们看到 IDA 已经根据 RegOpenKey 原型中对应的形式参数名称，对一个局部变量（hKey）和一个全局变量（SubKey）进行了命名。如果解析后的函数原型中仅包含类型信息，而没有形式参数名称，那么，前面例子中的注释将指出对应参数的数据类型，而非参数名称。至于 lpSubKey 参数，这个参数名称并不作为注释显示，因为这个参数碰巧指向一个字符串变量，而该字符串的内容则通过 IDA 的重复注将机制显示。最后，需要注意的是，IDA 已经将 RegOpenKey 识别为一个 stdcall 函数，并自动调整了栈指针✍，RegOpenKey 在返回时也会这样做。所有这些信息均源自该函数的签名，同时，IDA 将在反汇编代码清单中适当的导入表位置以注释的形式显示这些信息，如下面的代码段所示：

```
.idata:0040A000  ; LSTATUS __stdcall RegOpenKeyA(HKEY hKey, LPCSTR lpSubKey, PHKEY phkResult)
.idata:0040A000          extrn  RegOpenKeyA:dword ; CODE  XREF: _main+14p
.idata:0040A000                                   ; DATA  XREF: _main+14r
```

显示函数原型的注释来自 IDA 的一个.til 文件，该文件提供与 Windows API 函数有关的信息。

在什么情况下，生成自己的函数类型签名？2[2]如果你遇到一个链接到（无论是动态还是静态）IDA 并不包含其函数原型的二进制库文件，你可能希望为这个库中的所有函数生成类型签名信息，以便 IDA 能自动为你的反汇编生成注释。这类库包括常用的图形或加密库，虽然它们不属于标准 Windows 库，但却使用广泛。OpenSSL 加密库就是这样一个库。

第 8 章提到，我们可以在一个数据库的本地.til 文件中添加复杂的数据类型信息。同样，我们可以通过 File↴Load File↴Parse Header File 命令让 IDA 解析一个或几个函数原型，在同一个.til 文件中添加函数原型信息。同样，你可以使用 tilib.exe（见第 8 章）来解析头文件并创建独立的.TIL 文件，将它们复制到<IDADIR>/til，使其在全局范围内可以使用。

如果你拥有希望 IDA（tilib.exe）为你解析函数的源代码，这当然好。但多数情况下，你都无法访问相关源代码，并且你仍然希望获得高质量的反汇编。那么，在没有源代码可供参考的情况下，你如何为 IDA 提供信息？这正是 IDA 实用工具或 idsutils 的作用所在。这些 IDA 实用工具包括三个用于创建.ids 文件的实用程序。我们首先介绍.ids 文件的定义，然后说明如何创建我们自己的.ids 文件。

2[2] 这里，我们使用术语“签名”表示一个函数的参数类型、数量和顺序，而不是匹配已编译函数的代码模式。

1.1.1 IDS 文件

IDA 使用.ids 文件扩展它在库函数方面的知识。.ids 文件通过列举共享库中包含的每一个导出函数，来描述这个库的内容。与函数有关的详细信息包括函数名称、它的相关序号3[3]，还包括该函数是否使用 stdcall，如果使用，在返回时，该函数从栈上删除了多少字节的代码，另外也包括在反汇编代码清单中引用该函数时显示的可选注释。.ids 文件实际上是压缩后的.idt 文件，后者包含每个库函数的文本说明。

手动重写被删除字节

利用 stdcall 调用约定的库函数可能会给 IDA 的栈指针分析造成不良影响。缺乏任何类型库或.ids 文件信息，IDA 都无法知道导入函数是否使用 stdcall 约定。了解这一点非常重要，因为 IDA 可能无法在函数（IDA 并不了解它们的调用约定信息）调用中正确跟踪栈指针的行为。除了需要知道函数使用了 stdcall 外，IDA 还必须了解在完成操作时，这个函数到底从栈上删除了多少个字节。缺乏调用约定信息，IDA 将尝试使用一种叫做“单纯方法”（simplex method）*的算术分析技巧自动确定一个函数是否使用 stdcall。第二种技巧需要 IDA 用户手动干预。如图 13-1 所示是一个专门用于编辑导入函数的函数编辑对话模式。

图 13-1：编辑导入的的函数

导航到函数的导入表条目并选择编辑这个函数（ Eidt┐Functions┐Edit Function，或 ALT+P），即可打开该对话框。需要注意的是，这个特殊的对话框提供的功能有限（相对于图 7-7 所示的“编辑函数”对话框）。因为这是一个导入函数条目，IDA 无法访问该函数的已编译主体，因而也无法获得与该函数的栈帧结构有关的信息，以及它是否使用 stdcall 约定的直接证据。由于缺乏这些信息，IDA 只得将 Purged bytes 输入框设置为-1，表示它不知道该函数在返回时是否从栈中清除了字节。在这种情况下，要重写 IDA，需要输入已删除字节的正确数量，这样，在相关函数被调用时，IDA 会将获得的信息合并到它的栈指针分析中。如果 IDA 了解该函数的行为，Purged bytes 输入框可能已经填有数据（如图 13-1 所示）。注意，使用“单纯方法”分析时，这个输入框绝不可能会填入数据。

*Ilfak 在他的一篇博客文章中介绍了在 IDA5.1 版中引入的“单纯方法”，地址为 http://hexblog.com/2006/06。

初次在数据库中加载一个可执行文件时，IDA 将确定该文件所依赖的共享库。IDA 会在<IDADIR>/ids 目录中搜索与每一个共享库对应的.ids 文件，以获得有关该可执行文件可能引用的任何库函数的说明。需要记住的是，.ids 文件中不一定包含函数签名信息。因此，IDA 可能无法只根据.ids 文件中的信息提供函数参数分析。但是，如果.ids 文件能够正确指出函数所使用的调用约定，以及函数从栈中清除的字节数量，IDA 就能够进行准确的栈指针调整。如果一个 DLL 导出改编名称，IDA 就能够根据这个改编名称推断出一个函数的参数签名，在加载.ids 文件后，我们就可以利用这些信息。我们将在下一节中介绍.idt 文件的语法。在这方面，.til 文件包含与反汇编函数调用有关的更多有用的信息，不过，要想生成.til 文件，你需要使用源代码。

1.1.2 创建 IDS 文件

IDA 的 idsutils 实用工具用于创建.ids 文件。这些实用工具包括两个库解析器：从 Windows DLL 中提取信息的 dll2idt 和从 ar 风格库中提取信息的 ar2idt。无论使用哪一个解析器，其输出都是一个.idt 文本文件，它每行显示一个导出函数，并将导出函数的序号与函数名称对应起来。.idt 文件的语法非常简单，idsutils 自带的 readme.txt 文件介绍了这种语法。.idt 文件中的绝大多数行用于根据以下方案描述导出函数：

✍ 导出项以正数开头，这个数是导出函数的序号。

✍ 序号后是一个空格，后面接 Name=函数形式的 Name 指令，

例如，Name=RegOpenKeyA。如果使用零这个特殊的序号，则 Name 指令用于指定当前的.idt 文件所描述的库名称，如下所示：

```
0 Name=advapi32.dll
```

✍ 一个可选的 Pascal 指令可用于说明一个函数是否使用 stdcall 调用约定，并指出该函数返回时从栈中删除多少个字节的数据。例如：

```
483 Name=RegOpenKeyA  Pascal=12
```

✍ 可以在导出项后附加一个可选的 Comment 指令，指定一条注释，并在反汇编中每个引用该函数的位置与函数一起显示这条注释。一个完整的导出项如下所示：

```
483 Name=RegOpenKeyA  Pascal=12 Comment=Open  a registry key
```

此外，读者可以参阅 idsutils 的 readme.txt 文件了解其他可选的指令。idsutils 解析实用工具的目的是尽可能自动创建.idt 文件。创建.idt 文件的第一步，是获得你希望解析库的副本。然后，使用合适的解析实用工具解析这个副本。如果希望为与 OpenSLL 有关的 ssleay32.dll 库创建一个.idt 文件，可以使用以下命令：

```
$ ./dll2idt.exe  ssleay32.dll
```

3[3] 序号是与每个导出函数有关的整数索引。使用序号可通过整数查询表迅速定位一个函数。若通过将函数名称与字符串进行比较来定位函数，则很缓慢。

```
Convert DLL to IDT file. Copyright 1997 by Yury Haron. Version 1.5
File: ssleay32.dll ... ok
```

这时，如果解析成功，我们将得到一个名为 SSLEAY32.idt 的文件。由于 dll2idt.exe 基于从 DLL 库本身获得的信息生成输出文件名，因此，输入文件名与输出文件名之间存在大小写差异。生成的.idt 文件的前几行如下所示：

```
ALIGNMENT 4
;DECLARATION
;
0 Name=SSLEAY32.dll
;
121 Name=BIO_f_ssl
173 Name=BIO_new_buffer_ssl_connect
122 Name=BIO_new_ssl
174 Name=BIO_new_ssl_connect
124 Name=BIO_ssl_copy_session_id
```

请注意，解析器无法确定一个函数是否使用 stdcall，以及如果使用，它从栈上删除多少字节的数据。要想增加任何 Pascal 或 Comment 指令，你必须在创建最终的.ids 文件之前使用文本编辑器手动添加。创建.ids 文件的最后一个步骤，是使用 zipids.exe 实用工具压缩.idt 文件，并将得到的.ids 文件复制到<IDADIR>/ids 目录中。

```
$ ./zipids.exe SSLEAY32.idt
File: SSLEAY32.idt ... {219 entries [0/0/0]} packed
$ cp SSLEAY32.ids ../Ida/ids
```

这样，只要加载了一个链接到 ssleay32.dll 的二进制文件，IDA 就会加载 SSLEAY32.ids。如果你选择不将新建的.ids 文件复制到<IDADIR>/ids 目录中，你随时可以通过 File↴Load File↴IDS File 加载它们。

在使用.ids 文件时，可采用另一个步骤将.ids 文件链接到特定的.sig 或.til 文件。在选择.ids 文件时，IDA 会使用一个名为<IDADIR>/idsnames 的 IDS 配置文件。这个文本文件可执行以下操作：

✍ 将共享库的名称与它对应的.ids 文件名映射起来。如果共享库的名称不能完全转换成一个 MS-DOS 8.3 形式的文件名，这样做可帮助 IDA 定位正确的.ids 文件，如下所示：

```
libc.so.6 libc.ids +
```

✍ 将.ids 与.til 文件映射起来。这样，只要 IDA 加载指定的.ids 文件，它会自动加载指定的.til 文件。使用下面的命令，一旦 IDA 加载 SSLEAY32.ids，openssl.til 文件将会自动加载（请参阅 idsnames 文件了解相关语法信息）：

```
SSLEAY32.ids SSLEAY32.ids + openssl.til
```

✍ 将.sig 文件与对应的.ids 文件映射起来。这样，只要反汇编应用指定.sig 文件，IDA 将加载指定.ids 文件。下面的命令行指出：一旦用户应用 libssl.sig FLIRT 签名，IDA 应加载 SSLEAY32.ids 文件：

```
libssl.sig SSLEAY32.ids +
```

第 15 章将介绍如何使用脚本编写 idsutils 提供的库解析器。同时，我们将利用 IDA 的函数分析功能生成更加详细的.idt 文件。

1.2 用 loadint 扩充预定义注释

在第 7 章中，我们介绍了 IDA 的“自动注释”概念，如果启用了它，IDA 将显示描述每个汇编语言指令的注释。如下所示这种注释的两个例子：

```
.text:08048654 lea ecx, [esp+arg_0] ; Load Effective Address
.text:08048658 and esp, 0FFFFFFF0h ; Logical AND
```

这些预定义注释保存在<IDADIR>/ida.int 文件中，这些注释主要按 CPU 类型排序，其次按指令类型排序。如果启用自动注释，IDA 会在 ida.int 文件中搜索与每一条指令有关的注释，如果找到，它将在反汇编代码清单的右侧显示这些注释。

使用 loadint4[4]实用工具可以修改现有的注释，或在 ida.int 文件中添加新注释。如我们前面讨论的其他附加实用工具一样，loadint 发行版自带的 readme.txt 文件介绍了 loadint 的用法。loadint 发行版中还包含大量的.cmt 文件，它们是描述 IDA 的所有处理器模块的预定义注释。修改现有注释的过程非常简单，首先确定与处理器关联的注释文件（如用于 x86 处理器的 pc.cmt 文件），然后修改其中的注释，运行 loadint.exe 重新创建 ida.int 注释文件，最后将得到的 ida.int 文件复制到 IDA 主目录中，下次启动时，IDA 将从这个目录加载新建的 ida.int 文件。一段重建注释数据库的简单代码如下所示：

```
$ ./loadint comment.cmt ida.int
Comment base loader. Version 2.04. Copyright (c) 1991-2011 Hex-Rays

17566 cases, 17033 strings, total length: 580575
```

你希望进行的更改包括：修改现有注释，或为没有注释的指令添加注释。例如，在 pc.cmt 文件中，为使在启用自动注释时不会生成过多注释，IDA 并没有为几个比较常见的指令添加注释。下面的代码行取自 pc.cmt 文件，它们证实，默认情况下，x86 mov 指令并不生成注释：

```
NN_ltr:     "Load Task Register"
//NN_mov:   "Move  Data"
NN_movsp:   "Move  to/from  Special Registers"
```

如果你希望为 mov 指令添加注释，你可以删除中间一行注释，并根据详细步骤重建注释数据库。

loadint 文档资料中的一条提示指出：loadint.ext 必须能够找到 IDA 发行版自带的 ida.hlp 文件。如果你收到以下错误消息，应该将 ida.hlp 文件复到 loadint 目录，并重新运行 loadint。

```
$ ./loadint comment.cmt ida.int
Comment base loader. Version 2.04. Copyright (c) 1991-2011 Hex-Rays
Can't initialize help system.
File name: 'ida.hlp', Reason: can't find file (take it from IDA  distribution).
```

此外，你可以对 loadint 使用-n 选项，指定<IDADIR>的位置，如下面的命令行所示：

```
$ ./loadint -n <IDADIR> comment.cmt ida.int
```

comment.cmt 文件作为主输入文件提供给 loadint 的过程。该文件的语法在 loadint 文档中说明。简言之，Commnet.cmt 创建处理器类型与相关的注释文件之间的映射。特定于处理器的注释文件则反过来指定特定指令与每条指令的相关注释文本之间的对应关系。这整个过程由几个枚举（C 风格枚举）常量控制，它们定义所有处理器类型（位于 comment.cmt 文件中）以及每个处理器可能使用的所有指令（位于 allins.hpp 文件中）。

如果你希望给一个全新的处理器类型添加预定义注释，这个过程可能会比仅修改现有的注释要复杂一些。而且，这个过程还与创建新的处理器模块（见第 19 章）直接相关。如果不深入分析处理器模块，要给一个全新的处理器类型添加注释，首先，你需要在 allins.hpp 文件中创建一个新的枚举常量集合（与处理器模块共享），由它为指令集中的每条指令定义一个常量；其次，必须创建一个注释文件，将每个枚举指令常量与相关的注释文本关联起来；最后，必须为你的处理器类型定义一个新常量（同样，与处理器模块共享），并在 comment.cmt 中创建一个条目，将处理器类型与相关的注释文件对应起来。完成这些步骤后，必须运行 loadint，建立一个新的注释数据库，并将新的处理器类型及相关注释添加到其中。

1.3 小结

虽然 idsutils 和 loadint 现在似乎对你没有什么用处，但只要开始应用 IDA 的高级功能，你就需要用到这些实用工具。只需要花一点点时间创建一个.ids 或.til 文件，随后，如果你在将来的项目中遇到由这些文件描述的库，就可以节省大量的时间。记住，IDA 不可能为现有的每一个库提供注释。本章介绍的工具旨在帮助你全面了解 IDA 中的库。

4[4] 当前版本是 loadint61.zip

Reversing C++ programs with IDA pro and Hex-rays

Introduction

During my holidays, I had plenty of time to study and reverse a program, which was completely coded in C++. This was the first time I seriously studied a C++ codebase, using IDA as the only source of information, and found it quite hard.

Here’s a sample of what you get with Hex-rays when you start up digging into an interesting function:

```
v81 = 9;
v63 = *(_DWORD *)(v62 + 88);
if ( v63 )
{
     v64 = *(int (__cdecl **)(_DWORD, _DWORD, _DWORD,
     _DWORD, _DWORD))(v63 + 24);
     if ( v64 )
         v62 = v64(v62, v1, *(_DWORD *)(v3 + 16), *(_DWORD
         *)(v3 + 40), bstrString);
}
```

It’s our job to add symbol names, identify classes and set up all the information to help hex-rays in giving us a reliable and certainly understandable output:

```
padding = *Dst;
if ( padding < 4 )
   return -1;
buffer_skip_bytes(this2->decrypted_input_buffer, 5u);
buffer_skip_end(this2->decrypted_input_buffer, padding);
if ( this2->encrypt_in != null )
{
   if ( this2->compression_in != null )
   {
       buffer_reinit(this2->compression_buffer_in);
       packet_decompress(this2,
           this2->decrypted_input_buffer,
           this2->compression_buffer_in);
       buffer_reinit(this2->decrypted_input_buffer);
       avail_len = buffer_avail_bytes(this2->compression_buffer_in);
       ptr = buffer_get_data_ptr(this2->compression_buffer_in);
       buffer_add_data_and_alloc(this2->decrypted_input_buffer, ptr, avail_len);
  }
}
```

```
packet_type = buffer_get_u8(this2->decrypted_input_buffer);
*len = buffer_avail_bytes(this2->decrypted_input_buffer);
this2->packet_len = 0;
return packet_type;
```

Of course, Hex-rays is not going to invent the names for you, you'll still have to make sense of the code and what it means to you, but at least, being able to give a name to the classes will certainly help.

All my samples here have been compiled either with visual studio or Gnu C++. I have found the results to be similar, even if they may not be compatible. Fix it for your compiler of interest.

Structure of a C++ program

It is not my goal to teach you how OOP works, you already know that. We'll just see how it works (and is implemented) in the big lines.

Class = data structure + code (methods).

The data structure can only be seen in the source code, when the methods will appear in your favorite disassembler.

Object = memory allocation + data + virtual functions.

The object is an instantiation of a class, and something you can observe in IDA. An object needs memory, so you will see a call to new() (or a stack allocation), a call to a constructor and a destructor. You will see accesses to its member variables (embedded objects) and maybe calls to virtual functions.

Virtual functions are silly: it is hard to know, without running the program with breakpoints, what code is going to be executed at runtime (and disassemble it).

Member variables are a bit easier: they work like their counterpart in C (structs), and IDA has a very handy tool to declare structures, and hex-rays handles them very well in the disassembly. Let's go back to the bits and bytes.

Object creation

```
int __cdecl sub_80486E4()
{
  void *v0; // ebx@1
  v0 = (void *)operator new(8);
  sub_8048846(v0);
  (**(void (__cdecl ***)(void *))v0)(v0);
  if ( v0 )
    (*(void (__cdecl **)(void *))(*(_DWORD *)v0 + 8))(v0);
  return 0;
}
```

Here’s the decompilation of a small test program I compiled with G++. We can see the new(8), which means our object is 8 bytes long, even if that doesn’t mean we have 8 bytes of variables.

The function sub_8048846 called just after the new() takes the pointer as parameter, and certainly is the constructor.

The next function call is a little cryptic. It’s doing two pointer deferences on v0 before calling it. It’s a virtual function call.

All polymorphic objects have a special pointer in their variables, called the vtable. This table contains addresses of all the virtual methods, so the C++ program can call them when needed. In the compilers I could test, this vtable is always the first element of an object, and always stays at the same place, even in subclasses. (This could no stay true for multiple inheritance. I did not test).

Let’s do some IDA magic:

Rename the symbols

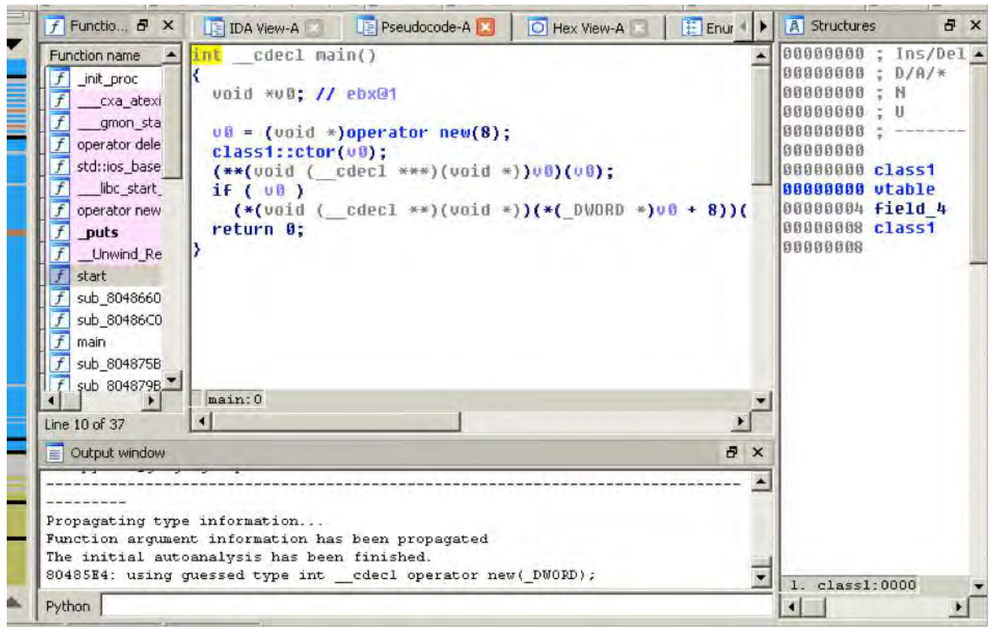
Just click on a name, press « n » and give a meaningful name. Since we don’t know yet what our class do, I suggest we name the class « class1 », and use this convention until we’ve understood what our class do. It’s very possible that we’re going to discover other classes before we finished digging class1, so I suggest we simply continue naming classes as we find them.

```
int __cdecl main()
{
  void *v0; // ebx@1
  v0 = (void *)operator new(8);
  class1::ctor(v0);
  (**(void (__cdecl ***)(void *))v0)(v0);
  if ( v0 )
    (*(void (__cdecl **)(void *))(*(_DWORD *)v0 + 8))(v0);
  return 0;
}
```

Create structures

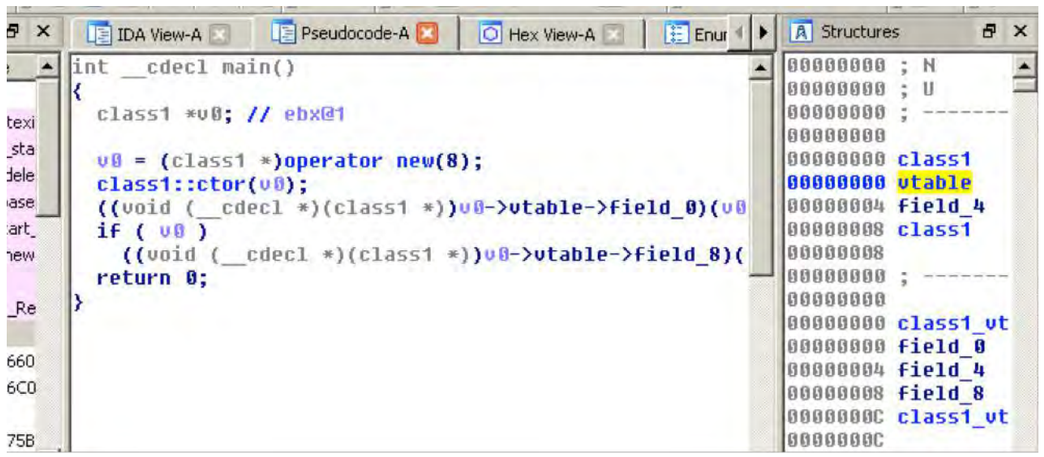
The « structures » window of IDA is very useful. Type Shift-F9 to make it appear. I suggest you pull it off (in the QT IDA version) and put it on the right of the IDA window, so you can see both the decompile window and the structures.

Press « ins » and create a new structure « class1 ». Since we know that this structure is 8 bytes long, add fields (using key « d ») until we have two « dd » fields. Rename the first to vtable, since yes, that’s what we got here !



Now, we're going to add typing information in our function. Right-click on v0, « Convert to struct * », select « class1 ». Alternatively, pressing « y » and typing in « class1 * » will give you the same result.

Create a new structure, of 12 bytes, and call it « class1_vtable ». At this state, we cannot really know how big that vtable is, but changing the structure size is very easy. Click on « vtable » in class1's declaration, and type « y ». Now, declare it as a « class1_vtable * » object. Refresh the pseudocode view, and watch the magic.



We can rename the few methods to « method1 » to « method3 ». Method3 is certainly the destructor. Depending on the programming convention and the compiler used, the first method often is the destructor, but here's a counterexample. It is time to analyze the constructor.

Analysis of the constructor

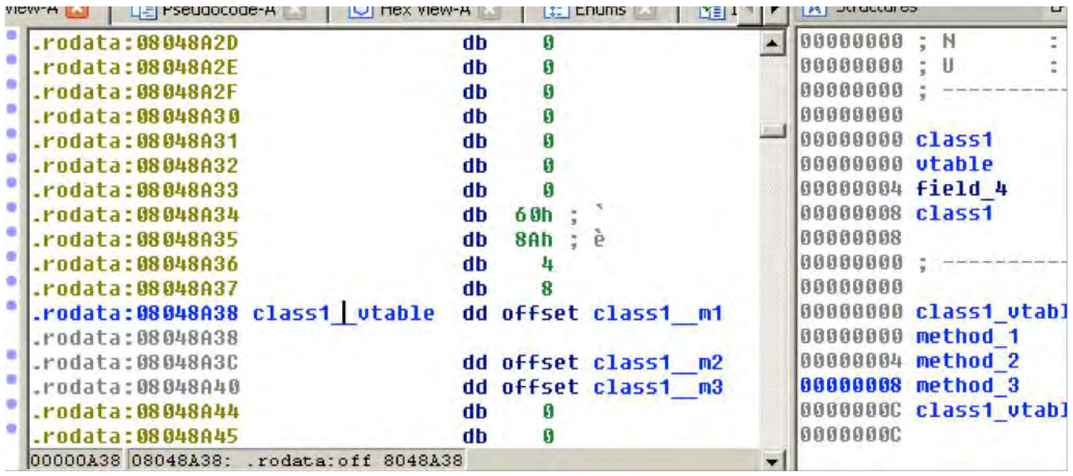
```
int __cdecl class1::ctor(void *a1)
{
  sub_80487B8(a1);
  *(_DWORD *)a1 = &off_8048A38;
  return puts("B::B()");
}
```

You can start by setting the typing information we already know on « a1 ». The puts() call confirms our thoughts that we are in a constructor, but here we even learn the name of the class.

« sub_80487B8() » is called directly in the constructor. This can be a static method of class1, but it can also be a constructor of a parent-class.

« off_8048A38 » is the vtable of class1. By looking there, you will be able to find out how big is our vtable (just watch the next pointer that has an Xref), and a list of the virtual methods of « class1 ». You can rename them to « class1_mXX », but beware that some of these methods may be shared with other classes.

It is possible to set typing information on the vtable itself (click on it, « y », « class1_vtable »), but I do not recommend it since you lose the classic view in IDA, and it doesn't provide anything you can't see in the classic view.



The strange call in the constructor

```
int __cdecl sub_80487B8(int a1)
```

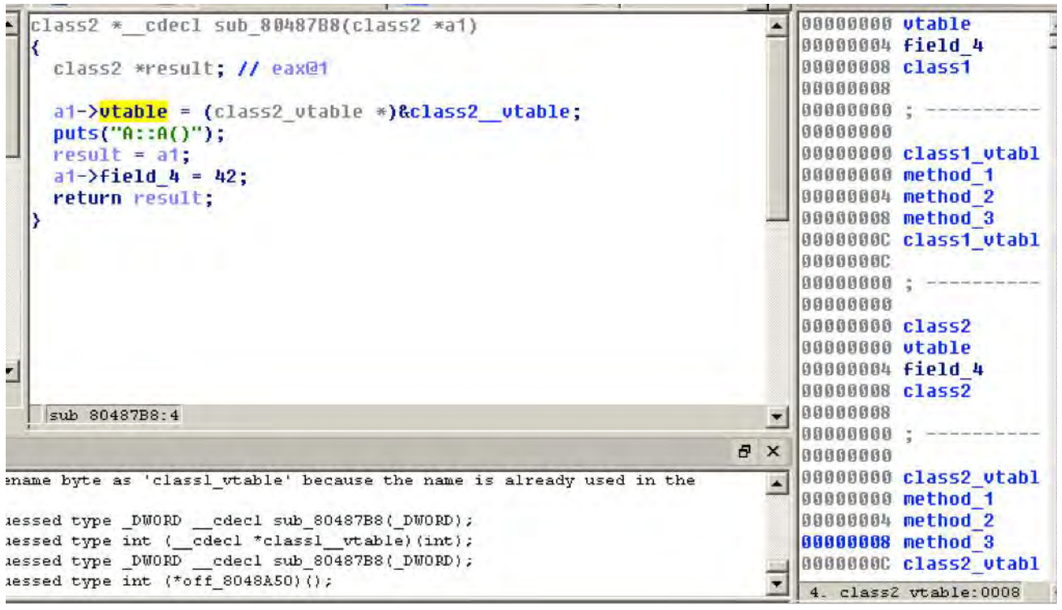
```
{
  int result; // eax@1
  *(_DWORD *)a1 = &off_8048A50;
  puts("A::A()");
  result = a1;
  *(_DWORD *)(a1 + 4) = 42;
  return result;
}
```

The call to the « sub_80487b8() » function in the constructor reveals us the same type of function: a virtual function table pointer is put in the vtable member, and a puts() tells us we're in yet another constructor.

Don't retype the type « class1 » for argument « a1 », since we're not dealing with class1. We found a new class, that we will call « class2 ». This class is a superclass of class1. Let's do the same work as in class1. The only difference it that we do not know exactly the size of its member. There are two ways of figuring it out:

- Look at the xrefs of class2 ::ctor. If we find a straight call to it after a new (i.e. an instantiation), we know the size of its members.
- Look at the methods in the vtable, and try to guess what's the highest member ever accessed.

In our case, « class2 ::ctor » accesses the 4 bytes after the 4 first ones and set it to 42. Since its child-class « class1 » is 8 bytes long, so is « class2 ».



Do the same procedure with all the subclasses, and give names to the virtual functions, starting from the parent classes to the children.

Study of the destructors

Let’s go back to our main function. We can see that the last call, before our v0 object becomes a memory leak, is a call to the third virtual method of class2. Let’s study it.

```
if ( v0 )
   ((void (__cdecl *)(class1 *))
       v0->vtable->method_3)(v0);
…
void __cdecl class1::m3(class1 *a1)
{
   class1::m2(a1);
   operator delete(a1);
}
…
void __cdecl class1::m2(class1 *a1)
{
   a1->vtable = (class1_vtable *)&class1__vtable;
   puts("B::~B()");
   class2::m2((class2 *)a1);
}
…
void __cdecl class2::m2(class2 *a1)
{
   a1->vtable = (class2_vtable *)&class2__vtable;
   puts("A::~A()");
}
```

What we can see here is the following: class1 ::m3 is a destructor, which calls class1 ::m2 which is the main destructor of class1. What this destructor do is ensure that we’re well in « class1 » context, by setting back the vtable to is « class1 » state. It then calls the destructor of « class2 », which also sets the vtable to « class2 » context. This method can also be used to walk through the whole class hierarchy, since the virtual destructors must always be called for all the classes in the way.

Hey, what are all these casts? Why do I have two structures defining the same fields?

What we have here is exactly the same problem that you get when doing OOP with C : You end up with several fields declared in all the subclasses. Here is what I do to avoid redefinition of fields:

- For each class, define a classXX_members, classXX_vtable, classXX structure.
- classXX contains
    - +++ vtable (typed to classXX_vtable *)

- +++ classXX-1_members (members of the superclass)
- +++ classXX_members, if any
  - classXX_vtable contains
  - +++classXX-1_vtable
  - +++classXX's vptrs, if any

Ideally, you should start from the main class to the children, until you end up in an edge class. In our exemple, here's the « solution » of our sample:

```
00000000 class1                  struc ; (sizeof=0x8)
00000000 vtable                  dd ?                                    ; offset
00000004 class2_members   class2_members ?
00000008 class1                  ends
00000008
00000000 ; ---------------------------------------------00000000
00000000 class1_members   struc ; (sizeof=0x0)
00000000 class1_members   ends
00000000
00000000 ; ---------------------------------------------00000000
00000000 class1_vtable     struc ; (sizeof=0xC)
00000000 class2_vtable     class2_vtable ?
0000000C class1_vtable     ends
0000000C
00000000 ; ---------------------------------------------00000000
00000000 class2                   struc ; (sizeof=0x8)
00000000 vtable                   dd ?                                     ; offset
00000004 members                 class2_members ?
00000008 class2                   ends
00000008
00000000 ; ---------------------------------------------00000000
00000000 class2_vtable     struc ; (sizeof=0xC)
00000000 method_1               dd ?                                    ; offset
00000004 dtor                      dd ?                                     ; offset
00000008 delete                   dd ?                                     ; offset
0000000C class2_vtable     ends
0000000C
00000000 ; ---------------------------------------------00000000
00000000 class2_members   struc ; (sizeof=0x4)
00000000 field_0                 dd ?
```

```
00000004 class2_members   ends
00000004
int __cdecl main()
{
   class1 *v0; // ebx@1
   v0 = (class1 *)operator new(8);
   class1::ctor(v0);
   ((void (__cdecl *)(class1 *)) v0->vtable->class2_vtable.method_1)(v0);
   if ( v0 )
       ((void (__cdecl *)(class1 *)) v0->vtable->class2_vtable.delete)(v0);
   return 0;
}
int __cdecl class1::ctor(class1 *a1)
{
   class2::ctor((class2 *)a1);
   a1->vtable = (class1_vtable *)&class1__vtable;
   return puts("B::B()");
}
class2 *__cdecl class2::ctor(class2 *a1)
{
   class2 *result; // eax@1
   a1->vtable = (class2_vtable *)&class2__vtable;
   puts("A::A()");
   result = a1;
   a1->members.field_0 = 42;
   return result;
}
```

In brief

- When you find a new class, give a symbolic name, and resolve the whole tree before figuring out what should be its real name
- Start from the ancestor and go up to the children
- Look at the constructors and destructors first, check out the references to new() and static methods.
- Often, the methods of a same class are located close to each other in the compiled file. Related classes (inheritance) may be far away from each other. Sometimes, the constructors are inlined in childclasses constructors, or even at the place of the instantiation.
- If you want to spare time when reversing huge inherited structures, use the struct inclusion trick to name variable only once.
- Use and abuse Hex-rays' typing system, it's very powerful.
- Pure virtual classes are hell : you can find several classes having similar vtables, but no code in common. Beware of them.

Sources

Try this at home !
The binary (elf32 stripped)

The source file. Don’t open it too fast !

http://blog.0xbadc0de.be/archives/67

静态分析

Android

Dex 静态

使用 IDA Pro 静态分析 Android 程序

摘至《Android 软件安全与逆向分析》

http://www.ituring.com.cn/article/26962

如何操作

以 5.2 节的 crackme0502.apk 为例，首先解压出 classes.dex 文件，然后打开 IDA Pro，将 classes.dex 拖放到 IDA Pro 的主窗口，会弹出加载新文件对话框，如图 5-4 所示，IDA Pro 解析出了该文件属于“Android DEX File”，保持默认的选项，点击 OK 按钮，稍等片刻 IDA Pro 就会分析完 dex 文件。

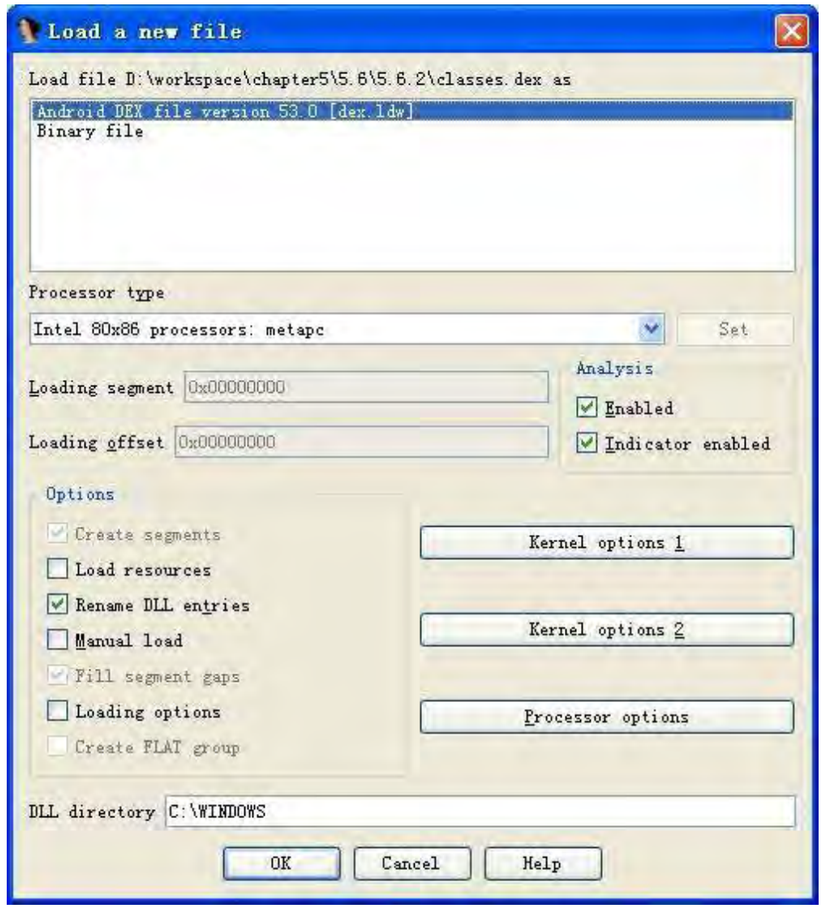


图 5-4　使用 IDA Pro 加载 classes.dex 文件

IDA Pro 支持结构化形式显示数据结构，因此，我们有必要先整理一下反汇编后的数据。dex 文件的数据结构大部分在 Android 系统源码中 dalvik\libdex\DexFile.h 文件中，笔者将其中的结构整理为 dex.idc 脚本，在分析 dex 文件时直接导入即可使用。导入的方法为 点击 IDA Pro 的菜单项“File→Script file”，然后选择 dex.idc 即可。点击 IDA Pro 主界面的 Structures 选项卡，如图 5-5 所示。

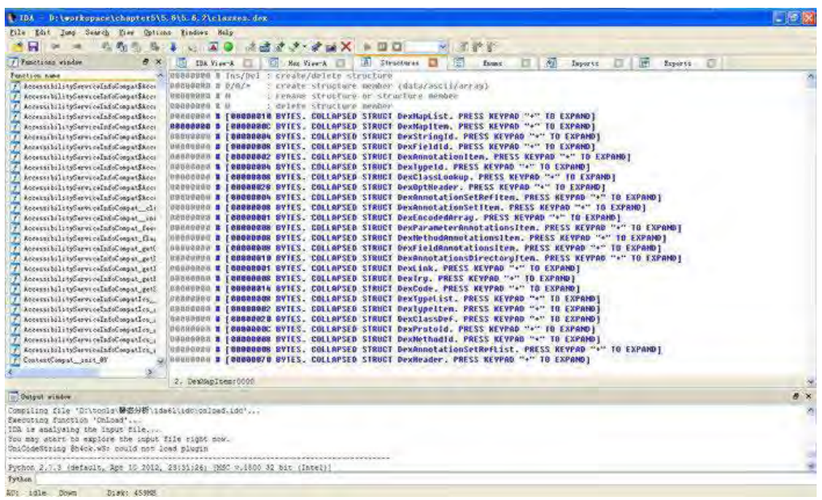


图 5-5　导入 dex.idc

点击 IDA View-A 选项卡，回到反汇编代码界面，然后点击菜单项“Jump→Jump to address”，或者按下快捷键 G，弹出地址跳转对话框，输入 0 让 IDA Pro 跳转到 dex 文件开头。将鼠标定位到注释“# Segment type: Pure data”所在的行，然后点击菜单项“Edit→Structs→ Struct var”，或者按下快捷键 ALT+Q，弹出选择结构类型对话框，如图 5-6 所示，选择 DexHeader 后点击 OK 按钮返回。

此时，dex 文件开头的 0x70 个字节就会格式化显示，效果如图 5-7 所示。同样，读者可以手动对 dex 文件中其它的结构进行整理，如 DexHeader 下面的 DexStringId 结构。

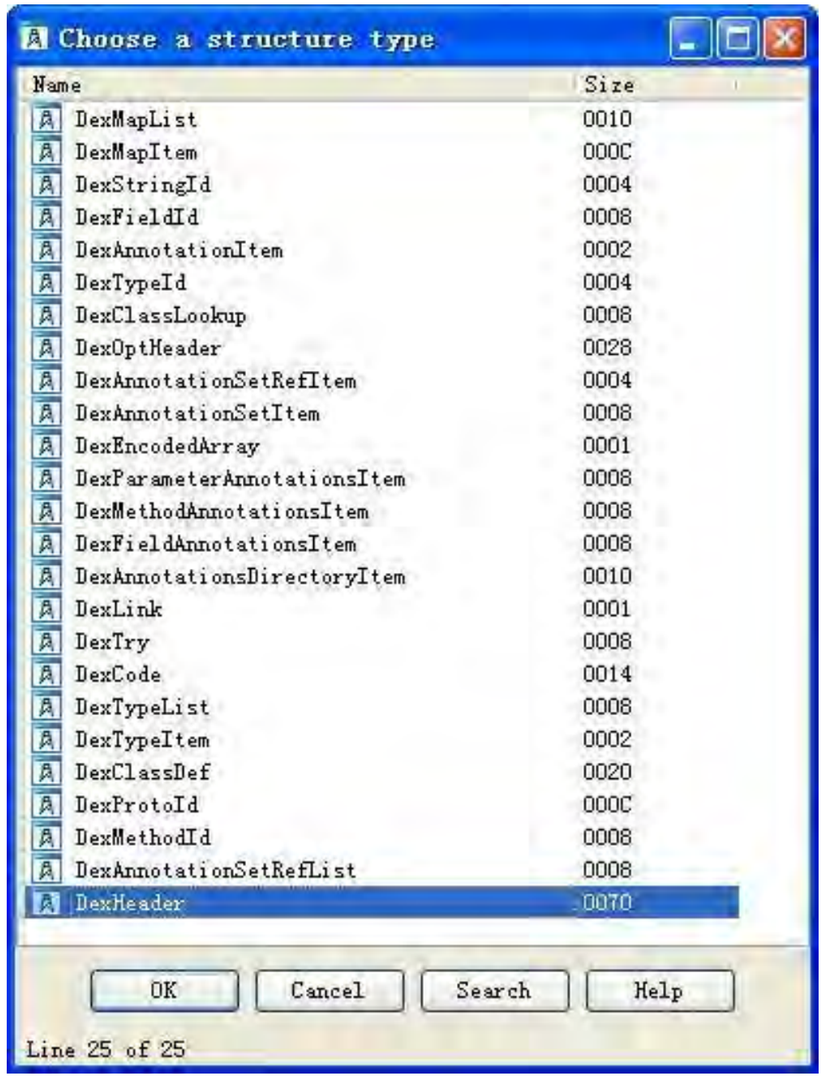

Choose a structure type
Name
Size
DexMapList 0010
DexMapItem 000C
DexStringId 0004
DexFieldId 0008
DexAnnotationItem 0002
DexTypeId 0004
DexClassLookup 0008
DexOptHeader 0028
DexAnnotationSetRefItem 0004
DexAnnotationSetItem 0008
DexEncodedArray 0001
DexParameterAnnotationsItem 0008
DexMethodAnnotationsItem 0008
DexFieldAnnotationsItem 0008
DexAnnotationsDirectoryItem 0010
DexLink 0001
DexTry 0008
DexCode 0014
DexTypeList 0008
DexTypeItem 0002
DexClassDef 0020
DexProtoId 000C
DexMethodId 0008
DexAnnotationSetRefList 0008
DexHeader 0070
OK
Cancel
Search
Help
Line 25 of 25

图 5-6　选择结构类型

图 5-7　格式化后的 DexHeader 结构

点击菜单项“Jump→Jump to segment”，或者按下快捷键 CTRL+S，弹出段选择对话框，如图 5-8 所示，IDA Pro 将 dex 文件一共分成了 9 个段，其中前 7 个段由 DexHeader 结构给出，最后 2 个段可以通过计算得出。仔细查看段名，可以发现 IDA Pro 对其命名不是很好，有 3 个 HEADER 段与 2 个 CODE 段，笔者觉得第 3 个段改名为 PROTOS 更合适一些，还有第 6 个段改名为 CLASSDEFS 更好，IDA Pro 为什么这样命名我们不得而知，不过，我们需要知道每个段具体所代表的含义。

图 5-8　dex 文件的 9 个段

dex 文件中所有方法可以点击 Exports 选项卡查看。方法的命名规则为“类名.方法名@方法声明”。在 Exports 选项卡中随便选择一项，如 SimpleCursorAdapter.swapCursor@LL，然后双击跳转到相应的反汇编代码处，该处的代码如下。

```
CODE:0002CFCC     Method 2589 (0xa1d):
CODE:0002CFCC    public android.database.Cursor
CODE:0002CFCC        android.support.v4.widget.SimpleCursorAdapter.swapCursor(
CODE:0002CFCC        android.database.Cursor p0)        #方法声明
CODE:0002CFCC         this = v2    #this 引用
CODE:0002CFCC         p0 = v3        #第一个参数
CODE:0002CFCC        invoke-super     {this, p0}, <ref ResourceCursorAdapter.
                        swapCursor(ref) imp. @ _def_ResourceCursorAdapter_
                        swapCursor@LL>
CODE:0002CFD2        move-result-object              v0
CODE:0002CFD4        iget-object  v1, this, SimpleCursorAdapter_mOriginalFrom
CODE:0002CFD8        invoke-direct    {this, v1}, <void SimpleCursorAdapter.
                    findColumns(ref) SimpleCursorAdapter_findColumns@VL>
CODE:0002CFDE
CODE:0002CFDE     locret:
CODE:0002CFDE        return-object                   v0
CODE:0002CFDE        Method End
```

IDA Pro 的反汇编代码使用 ref 关键字来表示非 Java 标准类型的引用，如方法第 1 行的 invoke-super 指令的前半部分如下。

```
invoke-super     {this, p0}, <ref ResourceCursorAdapter.swapCursor(ref)
```

前面的 ref 是 swapCursor()方法的返回类型，后面括号中的 ref 是参数类型。

后半部分的代码是 IDA Pro 智能识别的。IDA Pro 能智能识别 Android SDK 的 API 函数并使用 imp 关键字标识出来，如第 1 行的 invoke-super 指令的后半部分如下。

```
imp. @ _def_ResourceCursorAdapter_swapCursor@LL
```

imp 表明该方法为 Android SDK 中的 API，@后面的部分为 API 的声明，类名与方法名之间使用下划线分隔。

IDA Pro 能识别隐式传递过来的 this 引用，在 smali 语法中，使用 p0 寄存器传递 this 指针，此处由于 this 取代了 p0，所以后面的寄存器命名都依次减了 1。

IDA Pro 能识别代码中的循环、switch 分支与 Try/Catch 结构，并能将它们以类似高级语言的结构形式显示出来，这在分析大型程序时对了解代码结构 有很大的帮助。具体的代码反汇编效果读者可以打开 5.2 节使用到的 SwitchCase.apk 与 TryCatch.apk 的 classes.dex 文件 自行查看。

定位关键代码——使用 IDA Pro 进行破解的实例

使用 IDA Pro 定位关键代码的方法整体上与定位 smali 关键代码差不多。

第一种方法是搜索特征字符串。首先按下快捷键 CTRL+S 打开段选择对话框，双击 STRINGS 段跳转到字符串段，然后点击菜单项 “Search→text”，或者按下快捷键 ALT+T，打开文本搜索对话框，在 String 旁边的文本框中输入要搜索的字符串后点击 OK 按钮，稍等片刻 就会定位到搜索结果。不过目前 IDA Pro 对中文字符串的显示与搜索都不支持，如果字符串中的中文字符显示为乱码，需要编写相关的字符串处理插件来解决，这个工作就交给读者去完成了。

第二种方法是搜索关键 API。首先按下快捷键 CTRL+S 打开段选择对话框，双击第一个 CODE 段跳转到数据起始段，然后点击菜单项 “Search→text”，或者按下快捷键 ALT+T，打开文本搜索对话框，在 String 旁边的文本框中输入要搜索的 API 名称后点击 OK 按钮，稍等 片刻就会定位到搜索结果。如果 API 被调用多次，可以按下快捷键 CTRL+T 来搜索下一项。

第三种方法是通过方法名来判断方法的功能。这种办法比较笨拙，对于混淆过的代码，定位关键代码比较困难。比如，我们知道 crackme0502.apk 程序的主 Activity 类为 MainActivity，于是在 Exports 选项卡页面上输入 Main，代码会自动定位 到以 Main 开头的所在行，如图 5-9 所示，可粗略判断出每个方法的作用。

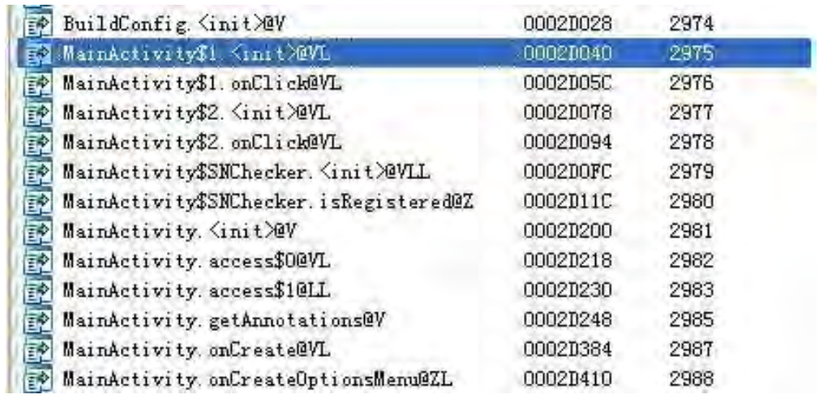


图 5-9 定位 MainActivity

下面我们来尝试破解一下 crackme0502.apk。首先安装运行 apk 程序，程序运行后有两个按钮，点击“获取注解”按钮会 Toast 弹出 3 条信息。在文本框中输入任何字符串后，点击“检测注册码”按钮，程序弹出注册码错误的提示信息。这里我们以按钮事件响应为突破口来查找关键代码，通过图 5-9 我们可以发现有两个名为 OnClick()的方法，那具体是哪一个呢？我们分别进去看看。前者调用了 MainActivity.access$0()方法，在 IDA Pro 的反汇编界面双击 MainActivity_access 可以看到它其实调用了 MainActivity 的 getAnnotations()方法， 看到这里应该可以明白，MainActivity$1.onClick()方法是前面按钮的事件响应代码。接下来查看 MainActivity$2.onClick()方法，双击代码行，来到相应的反汇编代码处，按下空格键切换到 IDA Pro 的流程视图，如图 5-10 所示，代码的“分水岭”就是“if-eqz v2, loc_2D0DC”。图中左边红色箭头表示条件不满足时执行的路线，右边的绿色箭头是条件满足时执行的路线。

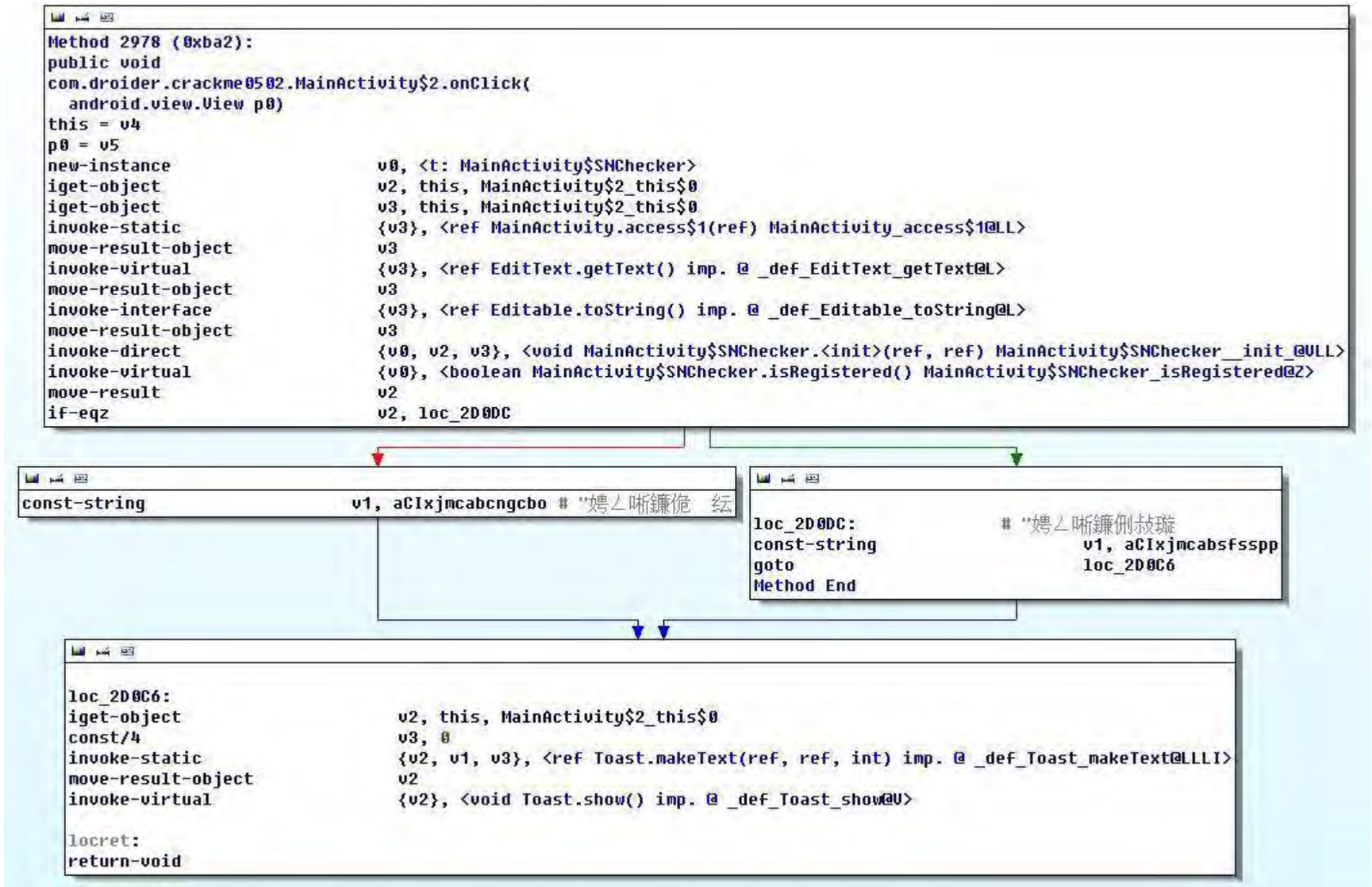


图 5-10　IDA Pro 的流程视图

虽然不知道这堆乱码字符串分别是什么，但通过最后调用的 Toast 来看，直接修改 if-eqz 即可将程序破解。将鼠标定位到指令“if-eqz v2, loc_2D0DC”所在行，然后点击 IDA Pro 主界面的“Hex View-A”选项卡，可看到这条指令所在的文件偏移为 0x2D0BE，相应的字节码为“38 02 0f 00”，通过前面的学习，我们知道只需将 if-eqz 的 OpCode 值 38 改成 if-nez 的 OpCode 值 39 即可。说干说干，使用 C32asm 打开 classes.dex 文件，将 0x2D0BE 的 38 改为 39，然后保存退出。接着按照本书 4.6 小节的介绍，将 dex 文件进行 Hash 修复后导入 apk 文件，对 apk 重新签名后安装测试发现程序已经破解成功了。

为了让读者看到一种常见的 Android 程序的保护手段，这里更换一下破解思路。通过图 5-10 可发 现，MainActivity$SNChecker.isRegistered()方法实际上返回一个 Boolean 值，通过判断它的返回值来确定注册码 是否正确。现在的问题是，如果该程序是一个大型的 Android 软件，而且调用注册码判断的地方可能不止一处，这种情况时，通常有两种解决方法：第一种是 使用 IDA Pro 的交叉引用功能查找到所有方法被调用的地方，然后修改所有的判断结果；第二种方法是直接给 isRegistered()方法“动手术”，让它的结果 永远返回为真。很显然，第二种方法解决问题更利落，而且一劳永逸。

下面尝试使用这种方法进行破解，首先按下空格键切换到反汇编视图，发现直接修改方法的第二条指令为“return v9 ”即可完成破解，对应机器码为“0F 09”，将其修改完成后重新修复与签名，安装测试发现程序启动后就立即退出了。这时最先怀疑的是程序是否修改正确，使用 IDA Pro 重新导入修改过的 classes.dex 文件，发现修改的地方没错，看来是程序采取了某种保护措施！回想一下前面提到的两种程序退出方法： Context 的 finish()方法与 android.os.Process 的 killProcess()方法，按下快捷键 CTRL+S 并双击 CODE 回到代码段，接着按下快捷键 ALT+T 搜索 finish 与 killProcess，最后在 MyApp 类的 onCreate()方法中找到了相应的调用，查 看相应的反汇编代码，发现这段代码使用 Java 的反射机制，手工调用 isRegistered()方法检查字符串“11111”是否为合法注册码，如果是 或者调用 isRegistered()失败都说明程序被修改过，从而调用 killProcess()来杀死进程。明白了保护手段，解决方法就简单多了，直 接将两处 killProcess()的调用直接 nop 掉（修改相应地方的指令为 0）就可以了。

利用 IDA Pro 分析 Android 软件基础教程

淡然出尘

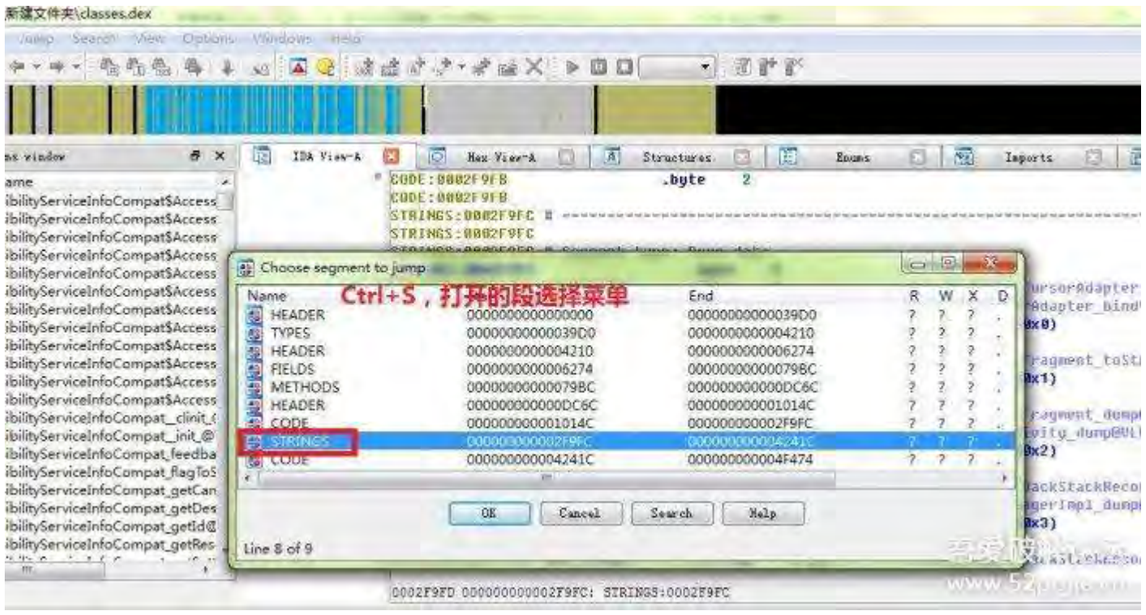
说明：本图文教程的方法及工具均分享自看雪移动区版主@非虫的原创图书《Android 软件安全与逆向分析》#5.6.3#
特为整理出来，新手教程 求高手指点啊 0(∩_∩)0

使用 IDA Pro 定位关键代码的方法：

1、搜索特征字符串。具体操作为：

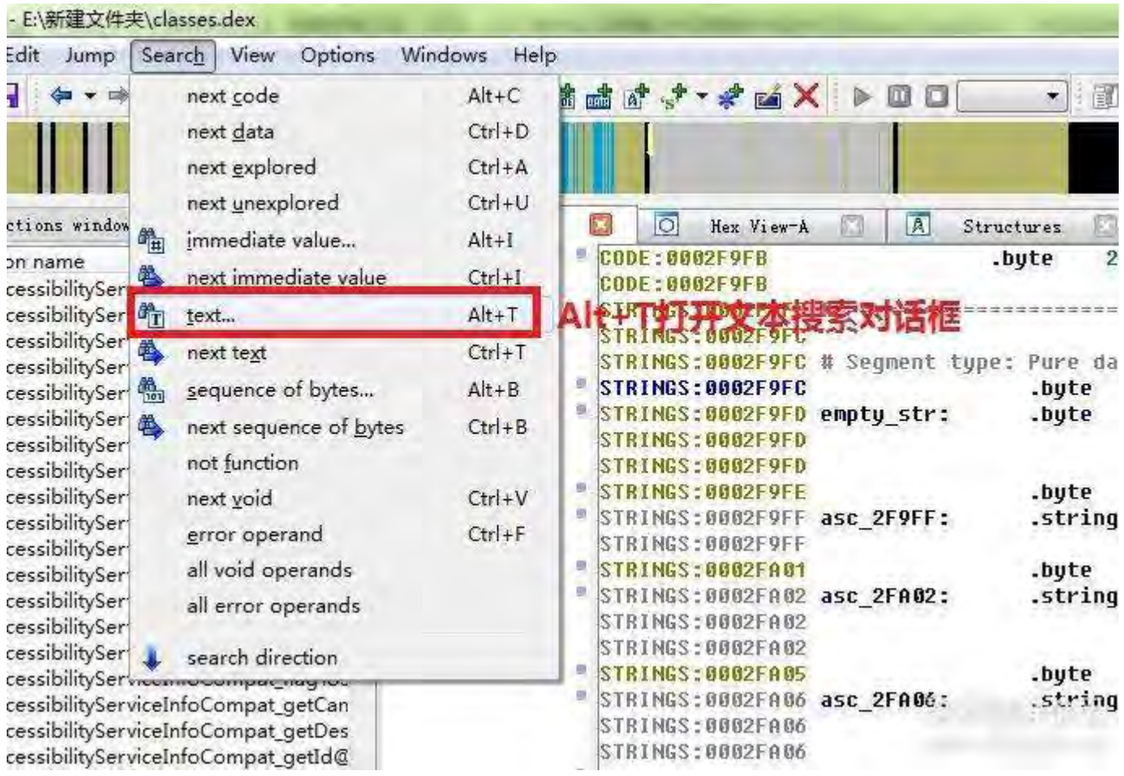
①快捷键 Ctrl+S

打开搜索类型选择对话框-->双击 Strings，跳到字符串段-->菜单项“Search-->Text”；



②快捷键 Alt+T

打开文本搜索对话框，在 String 文本框中输入要搜索的字符串点击 OK 即可；



不过 目前 IDA Pro 对中文字符串的显示和搜索都不支持，需要编写相关的字符串处理插件来解决。

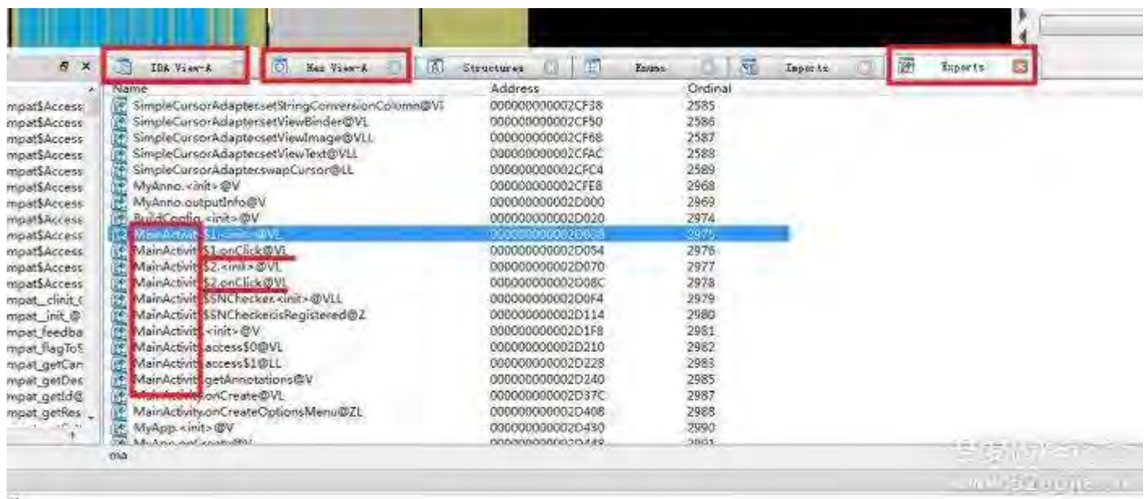
2、搜索关键的 API

搜索方法与上面的是一样的。
Tips：
Android API（可供调用的系统接口）可以分为：可选 API、Wi-Fi API、定位服务、多媒体 API、图形 API 等；

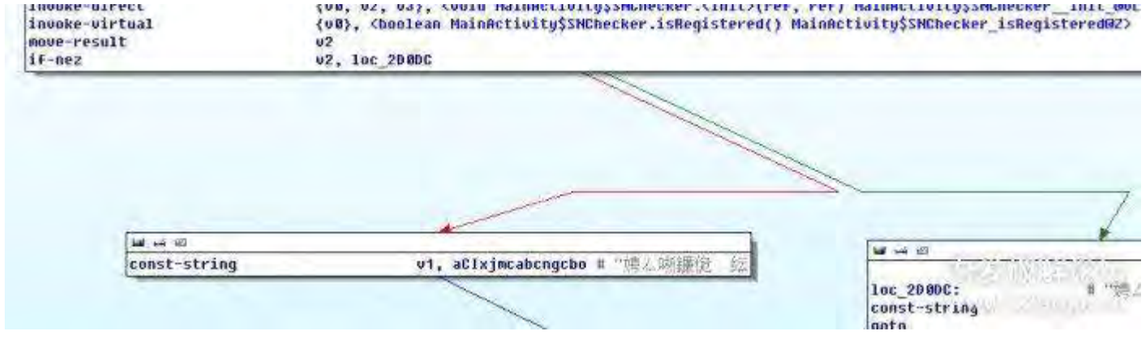
3、通过方法名来判断方法的功能

这种方法比较笨拙 对于混淆过的代码，定位关键代码比较困难。下面的 CrackMe 就可以通过方法名来定位关键代码。
下面我们就以一个 CrackMe 为实例来演示 IDA Pro 分析 Android 的流程。
首先安装运行 APK 程序 可以看到主界面上有两个按钮，点击第一个按钮“获取注解” 会 Toast 弹出三条信息；点击第二个按钮“检测注册码”
则会显示“注册码错误”。很显然 第一个按钮是混淆视听的。
Tips：
IDA Pro 是直接分析 Android 程序的主体 dex 文件 所以需要将 dex 文件从原程序中分离出来 可以用 RAR 压缩解压缩软件直接将 dex 从 apk 中拉出来，修改修复以后再借助它压缩进去即可。
所以我们就可以按钮事件的响应为突破口来查找关键代码。
很容易知道该 CrackMe.apk 的主 Activity 类为 MainActivity，于是在 Exports 选项卡页面上输入 Main，代码会自动定位到以 Main 开头的所在行 其中
的方法也就一目了然了。

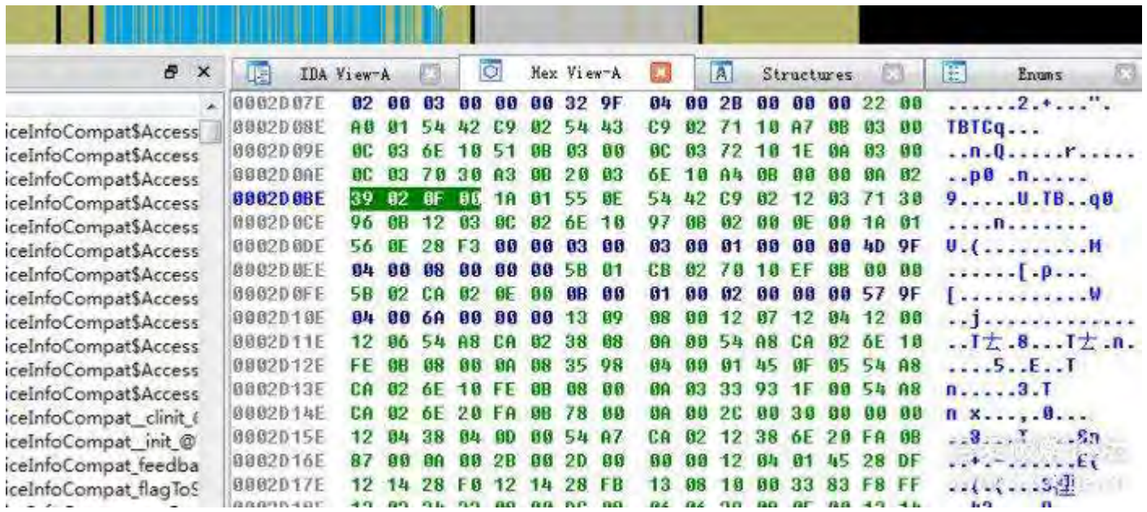
Tips:
一个 Android 程序是由一个或多个的 Activity 以及其他组件组成 每个 Activity 都是相同级别的，不同的 Activity 实现不同的功能。每个 Activity 都是 Android 程序的一个显示“页面” 主要负责数据的处理及展示工作。
每个 Android 程序有且仅有一个主 Activity（隐藏程序除外） 它是程序启动的第一个 Activity 通常标识为“android.intent.action.MAIN”.



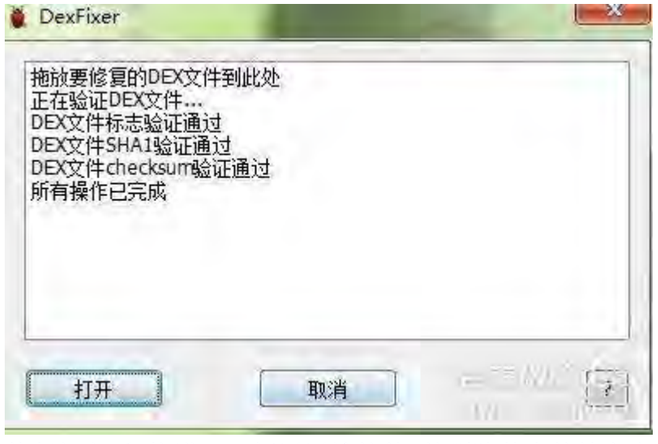
通过图我们可以看到有两个名为 OnClick()的方法 我们可以试试它们分别是对应哪一个。分别双击这两个代码行 会来到相应的反汇编代码（IDA View-A）处，
按下空格键切换到流程图。从而容易得出第二个 OnClick()方法是关键 且可以看到代码的分水岭就是“if-eqz v2,loc_2D0DC”。
左边红色箭头表示不满足时执行的路线；
右边表示条件满足时执行的路线。



将鼠标定位到指令“if-eqz v2,loc_2D0DC”，然后点击 IDA Pro 主界面上的“Hex View-A”选项卡，



知偏移地址为 0x2D0BE 相应的字节码为“38 02 0f 00”，
所以只需将 if-eqz 的 OpCode 值 38 改为 if-nez 的 OpCode 值 39 即可。
（说明：也就是利用十六进制编辑器 将偏移地址 0x2D0BE 的“38 02 0f ”改为“39 02 0f”，具体什么什么是 OpCode 我不明白了 o(╯□╰)o）
修改好之后的修复 dex 文件的合法性 所以就借助@非虫的工具



修复好之后 拖进 RAR 解压压缩软件 并删除其中的签名文件夹“META-INF”，这时 CrackMe.apk 的破解就完成了 签名安装就可以测试了。
http://www.52pojie.cn/thread-237886-1-1.html

IDA 破解 apk 的初次尝试

1、背景介绍

做 APP 开发一年多的时候，我突然对破解有了浓厚的兴趣，看了些资料，自己也动手写了写，老早之前有这样一篇文章，一直没空拿出来分享一下。最近又重新整理了一下，感觉写得还是比较详细的，拿出来和大家分享探讨一下。

2、工具和环境

工具：IDA Pro6.1，C32arm，apktool
环境：win7，jdk1.6
文件：crackme02.apk

上面的工具和文件大家自己准备一下，最后的 crackme02.apk 这个文件我会上传一份到 CSDN 让大家下载，好与我的文件保持一致。

实际上本文也是《Android 软件安全与逆向分析》一书中的总结，大家也可以找这本书看看。

3、加载 apk 到 IDA

首先使用 zip 解压缩文件取出 crackme02.apk 文件中的 classes.dex 文件，然后打开 IDA Pro 编译器，将程序拖入到 IDA Pro 编译器中，会有一个弹窗“Load a new file0”对话框，直接点击 OK 按钮。（注意选择 Android 文件）

这时，程序会进入反汇编的界面，稍等一下，IDA Pro 分析完 classes.dex 后会出现如下图所示的情形：

IDA - F:\classes.dex

File Edit Jump Search View Options Windows Help

Functions window

Function name

- AccessibilityServiceInfoCompat$AccessibilityS
- AccessibilityServiceInfoCompat$AccessibilityS
- AccessibilityServiceInfoCompat$AccessibilityS
- AccessibilityServiceInfoCompat$AccessibilityS
- AccessibilityServiceInfoCompat$AccessibilityS
- AccessibilityServiceInfoCompat$AccessibilityS
- AccessibilityServiceInfoCompat$AccessibilityS
- AccessibilityServiceInfoCompat$AccessibilityS
- AccessibilityServiceInfoCompat$AccessibilityS
- AccessibilityServiceInfoCompat$AccessibilityS
- AccessibilityServiceInfoCompat$AccessibilityS
- AccessibilityServiceInfoCompat__clinit_@V
- AccessibilityServiceInfoCompat__init_@V
- AccessibilityServiceInfoCompat_feedbackTyp
- AccessibilityServiceInfoCompat_flagToString(
- AccessibilityServiceInfoCompat_getCanRetrie
- AccessibilityServiceInfoCompat_getDescripti
- AccessibilityServiceInfoCompat_getId@LL
- AccessibilityServiceInfoCompat_getResolveIn
- AccessibilityServiceInfoCompat_getSettingsA
- AccessibilityServiceInfoCompatIcs__init_@V
- AccessibilityServiceInfoCompatIcs_getCanRet
- AccessibilityServiceInfoCompatIcs_getDescrip
- AccessibilityServiceInfoCompatIcs_getId@LL
- AccessibilityServiceInfoCompatIcs_getResolv
- AccessibilityServiceInfoCompatIcs_getSetting

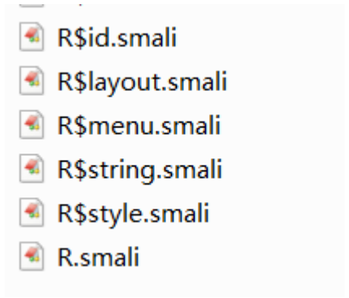
IDA View-A | Hex View-A | Structures | Enums | Imports

```
HEADER:00000000 #
HEADER:00000000 # +-------------------------------------------------------------------------+
HEADER:00000000 # |   This file has been generated by The Interactive Disassembler (IDA)    |
HEADER:00000000 # |           Copyright (c) 2011 Hex-Rays, <support@hex-rays.com>           |
HEADER:00000000 # |                         License info: 48-327F-7274-B7                   |
HEADER:00000000 # |                                ESET spol. s r.o.                        |
HEADER:00000000 # +-------------------------------------------------------------------------+
HEADER:00000000 #
HEADER:00000000 # Input MD5   : E208B5E127F47844E54E7CFBEE3A1ABA
HEADER:00000000 # Input CRC32 : 56DEB69C
HEADER:00000000
HEADER:00000000
HEADER:00000000                 DEX Module, Interface version 7
HEADER:00000000                 Input Dex File version 35
HEADER:00000000
HEADER:00000000 # ===========================================================================
HEADER:00000000
HEADER:00000000 # [00003938 BYTES: COLLAPSED SEGMENT HEADER. PRESS KEYPAD "+" TO EXPAND]
TYPES:00003938 # B
TYPES:00003938 # ===========================================================================
TYPES:00003938
TYPES:00003938 # Segment type: Pure data
TYPES:00003938                  .int unk_13E
TYPES:0000393C # C
TYPES:0000393C                  .int unk_155
TYPES:00003940 # D
TYPES:00003940                  .int unk_184
TYPES:00003944 # F
TYPES:00003944 off_3944:        .int unk_1A0            # DATA XREF: FIELDS:PagerTabStrip_mInitialMotionX↓i
TYPES:00003944                                          # FIELDS:PagerTabStrip_mInitialMotionY↓i ...
TYPES:00003948 # I
TYPES:00003948 off_3948:        .int unk_1DF            # DATA XREF: FIELDS:Notification_audioStreamType↓i
TYPES:00003948                                          # FIELDS:Notification_defaults↓i ...
TYPES:0000394C # J
TYPES:0000394C off_394C:        .int unk_208            # DATA XREF: FIELDS:Notification_when↓i
```

00000001 00000000: HEADER:00000000

4、定位资源选项

按照破解思路，我们找到资源文件中的 string.xml 文件，可以看到对应的字符串为 unsuccessed。然后我们使用 apktool 反编译 crackme02.apk 文件，从 smali 文件的 R 文件中找到 R$string.smali 文件中的 unsuccessed 所对应的 id 值，如下图所 示，我们得到的结果是：0x7f05000c



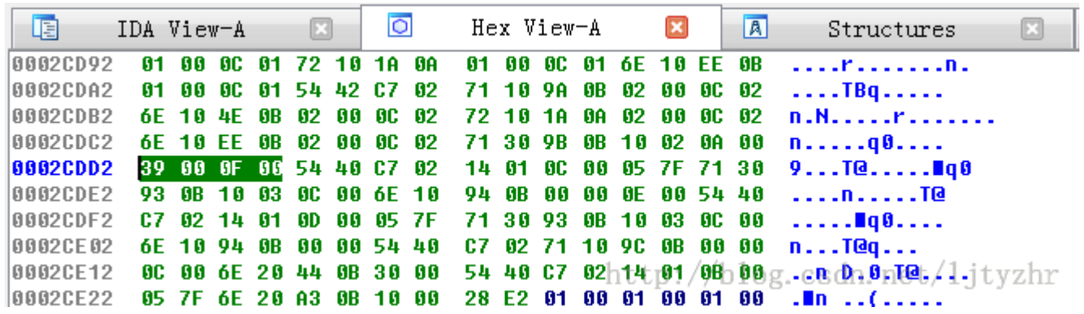
当找到字符串对应的 id 值之后，我们回到 IDA Pro 界面上来，在 IDA 的主窗口按下 Alt+T 快捷键，会弹出字符串搜索的窗口，我们在该窗口中输入刚才找到的 id 值 0x7f05000c，如图所示：

```
CODE:0002CDD2                if-nez                  v0, loc_2CDF0
CODE:0002CDD6                iget-object             v0, this, MainActivity$1_this$0
CODE:0002CDDA                const                   v1, 0x7F05000C
CODE:0002CDE0                invoke-static           {v0, v1, v3}, <ref Toast.makeText(ref, int, int) imp. @ _def_Toast_makeText@LLII>
CODE:0002CDE6                move-result-object      v0
```

然后会看到鼠标定位到了 0x7f05000c 所对应的地方。
这时候，我们向上查找最近的判断语句，很快会定位到 CODE：0002CDD2 行的 if-nez。

5、查找指令代码

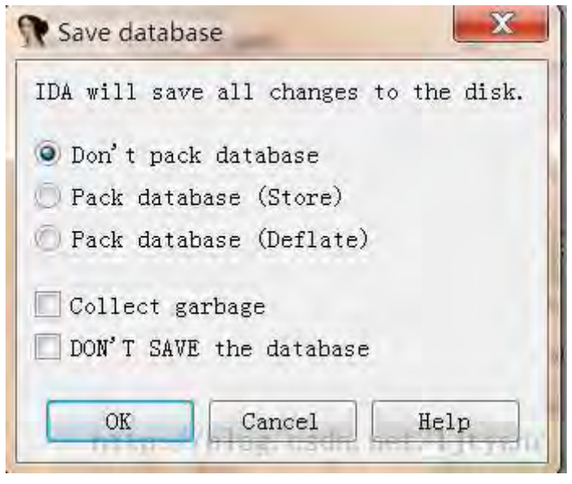
我们点击 IDA Pro 主界面上的“Hex View-A”选项卡，发现这行代码的指令为“39 00 0f 00”第一个字节 39 为 if-nez 的指令操作码，我们只需将其改为 if-eqz 的指令码就行了。查表如图所示：



38	if-eqz vx, *目标*	如果 vx == 0[注2]，跳转到*目标*。vx 是 int 型值。	3802 1900 - if-eqz v2, 0038 // +0019 如果 v2 == 0，跳转到当前位置+19H。0038 是目标指令标签。
39	if-nez vx, *目标*	如果 vx != 0[注2]，跳转到*目标*。	3902 1200 - if-nez v2, 0014 // +0012 如果 v2 != 0，跳转到当前位置+18(hex 12)。0014 是目标指令标签。

6、修改

关闭 IDA Pro，在弹出的 Save datebase 对话框中勾选“DON'T SAVE the database”然后点击 OK 退出程序。
使用十六进制编辑工具打开 classes.dex 文件，本处使用 C32asm，定位到 0X2CDD2 处，修改 39 为 38，保存退出。



7、签名

apk 程序在安装的时候会调用 dexopt 对 dex 进行验证与优化，dex 文件的 DexHeader 头的 checksum 字段标识了 dex 文件的合法 性。当我们修改了 classes.dex 文件之后，dex 文件在验证时计算 checksum 会失败，这样我们需要重新计算并写回 checksum 的值。

最后我们将修改后的 classes.dex 文件压缩到刚才解压的 zip 包中，然后删除 MAIN-INF 文件，使用 Auto-sign 签名文件对它签名即可。

8、测试

测试通过！

9、文件地址

http://download.csdn.net/detail/ljtyzhr/8071745

http://blog.csdn.net/ljtyzhr/article/details/40393471

SO 静态

一个 JNI 逆向传参数引发的血案分析

作者:PeterChen
时间:9/22/2015 11:44:46 AM
我一个提问整理

默认的两个参数

用 JNI 传参时，前面两个参数是默认
CrazyMonkey
第一个参数是 JNIEnv *，第二个参数的 jobject

什么是 ATPCS

转载请注明出处
作者:小马

很多人学了 ARM 好长时间也不清楚到底什么是 ATPCS。其实 ADS 安装文件里有一个英文 PDF 文档专门讲什么是 ATPCS。 我简单说一下自己对 ATPCS 的理解(看了原文的文档和网上的一些资料)，如果想要了解更深入的话，可以去查看原文档。

ATPCS 是 ARM-Thumb Procedure Call Standard 的缩写，也就是 ARM,Thumb 的程序调用标准. 总体来说该标准分为两部分，一是基本的 ATPCS，还有就是扩展的 ATPCS。 下面分这两部分来说明

一标准的 ATPCS.

1 通用寄存器

Register	Synonym	Special	Role in the procedure call standard
r15		**PC**	The Program Counter.
r14		**LR**	The Link Register.
r13		**SP**	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8	FP	ARM-state variable-register 8. ARM-state frame pointer.
r10	v7	**SL**	ARM-state variable-register 7. Stack Limit pointer in stack-checked variants.
r9	v6	**SB**	ARM-state v-register 6. Static Base in PID,/re-entrant/shared-library variants
r8	v5		ARM-state variable-register 5.
r7	**v4**	**WR**	Variable register (v-register) 4. Thumb-state Work Register.
r6	**v3**		Variable register (v-register) 3.
r5	**v2**		Variable register (v-register) 2.
r4	**v1**		Variable register (v-register) 1.
r3	**a4**		Argument/result/scratch register 4.
r2	**a3**		Argument/result/ scratch register 3.
r1	**a2**		Argument/result/ scratch register 2.
r0	**a1**		Argument/result/ scratch register 1.

根据这张表显示, r0~r3 一般用来传递函数的参数，r4~r7 则用来放置局部变量. 而 r12~r15 则可以有特别的用途.

2 浮点数寄存器

基本的 ATPCS 支持两种浮点数体系结构. FPA 和 AFP. FPA 体系有八个可以放置单精度和双精度数的浮点寄存器. AFP 体系有 16 个双精度寄存器.

3 参数调用

参数调用分两种情况，参数可变的调用和参数个数固定的调用. 有以下的规则

* 小于 32 位的参数值会被自动扩展为 32 位.
* 64 位的参数被当成两个 32 位数.
* 对于浮点数.如果芯片本身硬件上支持浮点运算，则浮点参数会放在浮点寄存器里传递. 如果硬件上不支持浮点数运算，则转化为整型放通用寄存器传递.
* 其它类型通通转为 32 位整型数传递
* 对于参数可变的程序调用，前四个参数放在 r0~r3 中传递，如果多于四个参则按相反的顺序进栈保存,所谓相反的顺序是指靠前参数后进栈.
* 对于固定参数的调用，如果有可以做浮点运算的硬件部件，各个浮点参数按顺序处理;为每个浮点参数分配 FP 寄存器;分配的方法是,满足该浮点参数需要的且编号最小的一组连续的 FP 寄存器.第一个整数参数通过寄存器 R0~R3 来

传递,其他参数通过数据栈传递.

4 子程序返回

* 一个字大小的结果放 r0
* 64 位可通过 r0, r1 返回.依次类推
* 如果返回的结果是一个浮点数,则可通过 f0, d0 或 s0 返回.

二扩展的 ATPCS

1 对 PIC 的支持.所谓 PIC 就是 position independent code. 当一个程序中的 RO(read only) segment 可以在任何地址被加载和使用的时候,它就是 ROPI(read only position independent). 通常在一个程序中只读往往是 code segment, 这就是 PIC 的由来.

2 对 RWPI(read only position independent).

3 支持数据栈限制检查
http://blog.csdn.net/pony_maggie/article/details/5270203

逆向函数的分析

```
private static native void native_get_suggestions(long arg0, String arg1, int arg2, List arg3, List
    arg4, int arg5)
.text:0000B6E8                  EXPORT Java_com_test_native_1get_1suggestions
.text:0000B6E8          Java_com_test_native_1get_1suggestions
.text:0000B6E8
.text:0000B6E8          var_58      = -0x58
.text:0000B6E8          var_54      = -0x54
.text:0000B6E8          var_50      = -0x50
.text:0000B6E8          var_4C      = -0x4C
.text:0000B6E8          var_44      = -0x44
.text:0000B6E8          var_38      = -0x38
.text:0000B6E8          arg_0       =  0
.text:0000B6E8          arg_4       =  4
.text:0000B6E8          arg_8       =  8
.text:0000B6E8          arg_C       =  0xC
.text:0000B6E8          arg_10      =  0x10
.text:0000B6E8
.text:0000B6E8 F0 B5                PUSH    {R4-R7,LR}
.text:0000B6EA 5F 46                MOV     R7, R11
.text:0000B6EC 56 46                MOV     R6, R10
.text:0000B6EE 4D 46                MOV     R5, R9
.text:0000B6F0 44 46                MOV     R4, R8
```

```
.text:0000B6F2 F0 B4                    PUSH    {R4-R7}
.text:0000B6F4 8D B0                    SUB     SP, SP, #0x34
.text:0000B6F6 90 46                    MOV     R8, R2
.text:0000B6F8 18 9A                    LDR     R2, [SP,#0x58+arg_8]
.text:0000B6FA 16 9E                    LDR     R6, [SP,#0x58+arg_0]
.text:0000B6FC A9 23                    MOVS    R3, #0xA9
.text:0000B6FE 92 46                    MOV     R10, R2
.text:0000B700 02 68                    LDR     R2, [R0]
.text:0000B702 9B 00                    LSLS    R3, R3, #2
.text:0000B704 31 1C                    MOVS    R1, R6
.text:0000B706 D3 58                    LDR     R3, [R2,R3]
.text:0000B708 00 22                    MOVS    R2, #0
.text:0000B70A 04 1C                    MOVS    R4, R0
.text:0000B70C 98 47                    BLX     R3
.text:0000B70C
.text:0000B70E 05 1E                    SUBS    R5, R0, #0
.text:0000B710 00 D1                    BNE     loc_B714
.text:0000B710
.text:0000B712 E8 E0                    B       loc_B8E6
```

Zrhai

R0 吧。

当参数小于 4 个时，ARM 首先会用 R0-R3 四个寄存器来传递参数，当参数个数大于 4 时，开始使用栈来传递参数.

Nermor

```
MOV     R7, R11
MOV     R6, R10
MOV     R5, R9
MOV     R4, R8
```

PUSH {R4-R7} 这里是保存 R8-R11 寄存器的值；

IsBerry

你这个参数很复杂,你的 r0 是 this 指针，也就是 env，不管你写没写，都会带过来，从上面看，你的 r2 传过来了，保持了，你 r1 被覆盖，传来的没有使用，

到底什么是你的传的第一个参数，如果你是才开始写程序的话，可能你程序写错了，没有写上 env 变量，

后面五个参数从 ida 上看应该是

```
.text:0000B6E8                  arg_0       = 0
.text:0000B6E8                  arg_4       = 4
.text:0000B6E8                  arg_8       = 8
.text:0000B6E8                  arg_C       = 0xC
.text:0000B6E8                  arg_10      = 0x10
```

但是这是不对的，如果你的是上面的那种定义的 7 个参数的话，你的 arg0->r2
push {R4-R7,LR}
5 个寄存器
第一个调用函数的 offset ＝ 0xA9 *2 = GetStringUTFChars
*4
2A4

从 ida 后面来说，你的是对的，5 *84 ＋ 4*4 ＋ 0x34 ＝ 0x58 ， [SP,#0x58+arg_0] 的确对应第一个参数
.text:0000B6F8 18 9A LDR R2, [SP,#0x58+arg_8]
.text:0000B6FA 16 9E LDR R6, [SP,#0x58+arg_0] // 的确对应第一个参数

```
suggestions(JNIEnv *env, jobject objc, int paramLong, int paramString, int paramInt1, int paramList1, int paramList3, int paramInt2)
{

 (*env)->GetStringUTFChars(env, (jstring)paramInt1, 0);

.text:0000B6F8          LDR     R2, [SP,#0x58+paramList3]
.text:0000B6FA          LDR     R6, [SP,#0x58+paramInt1]

}
```

r0 为 env 指针，r1 为 objc， 没有使用，所以可以被覆盖，r2 传到 r8 ，r8 也没有使用，r3 被覆盖，也没有使用，就是说六个参数中 前两个没有使用，加上一个系统的 r1 没有使用，共三个没有使用，为 r1，r2，r3，sp 的 0 位置为传递六个参数中的第三个参数即为 0000B6FA 处的参数。

Reverse Engineering 破解 Android NDK 程式(*.so)

本篇僅供教學用途】

為了要保護自己的 Android App，開發者有時會把「付費註冊、序號處理、加解密」等重要的程式碼使用 C/C++來撰寫，因為 C/C++無法反編譯，只能反組譯，因此較難破解

要破解 Java 程式碼較簡單，只要熟悉 dalvik bytecode 就好了，參考資料：

1. http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html
2. http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html

但要破解 C/C++程式碼其實也沒這麼難（若是程式的流程規劃不好，則破解 C/C++甚至比破解 Java 簡單），以下是必要基礎或工具：

1. 熟悉 ARM 指令集（https://www.tenlong.com.tw/items/1558608745?item_id=22049）
2. IDA Pro 6.4 或更新版本（逆向工程 ARM 組語用）

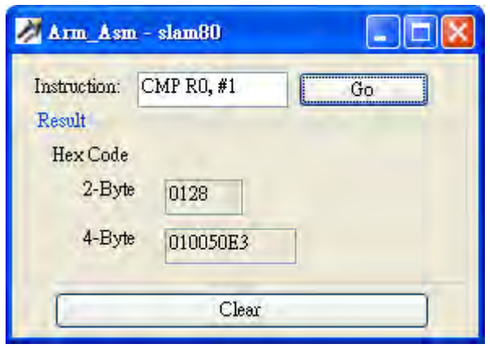
3. Arm_Asm - slam80（如果不會將 ARM 指令集轉為 hex，則該工具可以幫忙）
4. Hex Editor Neo（修改 hex 用）

破解的步驟大致上是：

1. 取得 apk 中的.so
2. 使用 IDA Pro 找到關鍵點（以下是範例）：

```
.text:00007414 7B 44                                    ADD     R3, PC ; pkg_zip_ptr
.text:00007416 1B 68                                    LDR     R3, [R3] ; pkg_zip
.text:00007418 18 60                                    STR     R0, [R3]
.text:0000741A 00 28                                    CMP     R0, #0
.text:0000741C 0D D0                                    BEQ     loc_743A
.text:0000741E 22 68                                    LDR     R2, [R4]
.text:00007420 C0 23 9B 00                              MOVS    R3, #0x300
.text:00007424 D5 58                                    LDR     R5, [R2,R3]
.text:00007426 20 1C                                    MOVS    R0, R4
.text:00007428 31 1C                                    MOVS    R1, R6
.text:0000742A 3A 1C                                    MOVS    R2, R7
.text:0000742C 00 23                                    MOVS    R3, #0
.text:0000742E A8 47                                    BLX     R5
.text:00007430 01 20                                    MOVS    R0, #1
.text:00007432
.text:00007432                          loc_7432                                ; CODE XREF: Java_com_
.text:00007432 02 B0                                    ADD     SP, SP, #8
.text:00007434 04 BC                                    POP     {R2}
.text:00007436 90 46                                    MOV     R8, R2
.text:00007438 F0 BD                                    POP     {R4-R7,PC}
.text:0000743A                          ; ---------------------------------------------------------
.text:0000743A
.text:0000743A                          loc_743A                                ; CODE XREF: Java_com_
.text:0000743A 22 68                                    LDR     R2, [R4]
.text:0000743C C0 23 9B 00                              MOVS    R3, #0x300
.text:00007440 D5 58                                    LDR     R5, [R2,R3]
.text:00007442 20 1C                                    MOVS    R0, R4
.text:00007444 31 1C                                    MOVS    R1, R6
.text:00007446 3A 1C                                    MOVS    R2, R7
.text:00007448 00 23                                    MOVS    R3, #0
.text:0000744A A8 47                                    BLX     R5
.text:0000744C 00 20                                    MOVS    R0, #0
.text:0000744E F0 E7                                    B       loc_7432
```

3.
4. 通常開發者要判斷「是否已註冊」或「是否正確」都會透過一個判斷式，若有判斷則會進行 Branch 指令，BEQ 則代表 condition 是 EQual 的
5. 找到對應的 hex，如「CMP R0, #0」對應「00 28」，BEQ 要進去的 branch 是當掉的判斷式，當然就可以將上一個指令「CMP R0, #0」改為「CMP R0, #1」，就不會進入判斷式或「就會進入付費版模式」 （僅舉例，實際情況依不同的.so 而定）
6. 透過工具算出 hex 為「01 28」

7. 

8. 當然您也可以自己算 hex，請參考：
   - http://www.nyx.net/~troddis/ARM.html
   - http://stackoverflow.com/questions/11785973/converting-very-simple-arm-instructions-to-binary-hex
9. 使用 Hex Editor NEO 修改為新的 hex，再儲存即可
10. 把修改後的.so 丟進去 apk，再重新 compile 與 sign
11. 找到的修改點不一定正確，因此可能需要嘗試數次，若成功了就把關鍵點拿掉了

http://www.dotblogs.com.tw/cheng/archive/2014/03/09/144307.aspx

google play 破解（一）：钻石修改

转载请注明文章出处：http://d-iao.weebly.com/2002739029/google-play

Blog 建立了有一段时间 都没空整理 今天抽出点时间写个教程
破解 play 验证 有 2 种方式
1.内购破解
2.修改数据

前者破解必须安装 play 商店 否则不能完成内购
国内很多手机都不自带 play 商店 所以有时候就算破解了内购
别人手机若没 play 商店 同样无法内购成功
所以大多数的破解者 都是选择后者直接修改游戏的钻石或金币达到想要的效果
最近看到一款游戏代码比较简单 很适合写个入门教程 所以就打算写篇文章
今天的主角是一款过关的 RPG 游戏 看说明是超过 100 万的下载量的游戏 我们来看看

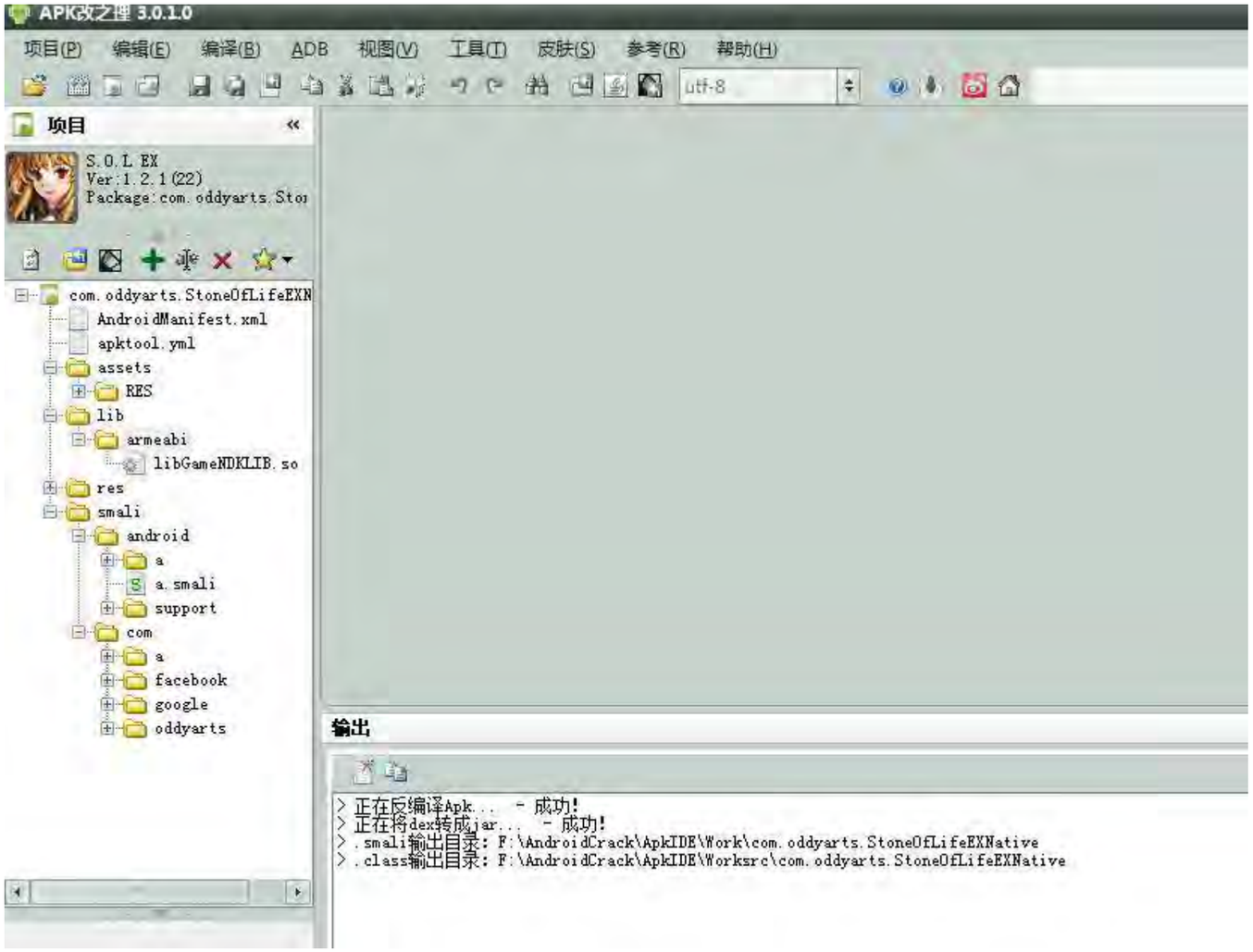

ver 1.2.1
TOUCH THE SCREEN
SOL Ex
STONE OF LIFE
© 2014 ODDY ARTS CORP.

我们来看看内容

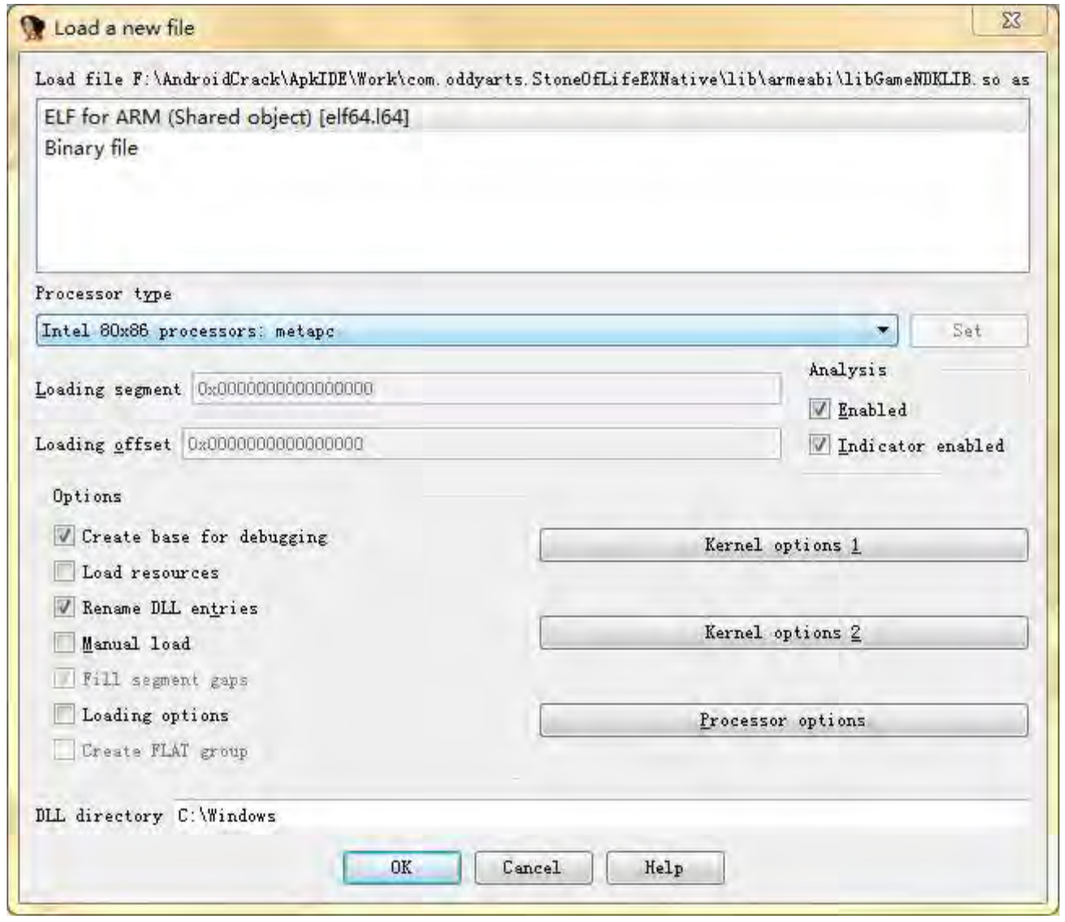
从上图里我们可以看到 游戏货币分为 3 种
1.ticket 门票
2.gold 金币
3.gem 钻石
经分析 ticket 和 gold 可以使用 gem 买 然而钻石则要用真钱买
所以 我们只要对 gem 的数量进行修改即可
可以从上图的右上角 gem shop 里看到 这游戏的钻石叫做 gem

点击 gem shop 后 可以看到如上图 gem 的购买是通过真钱购买

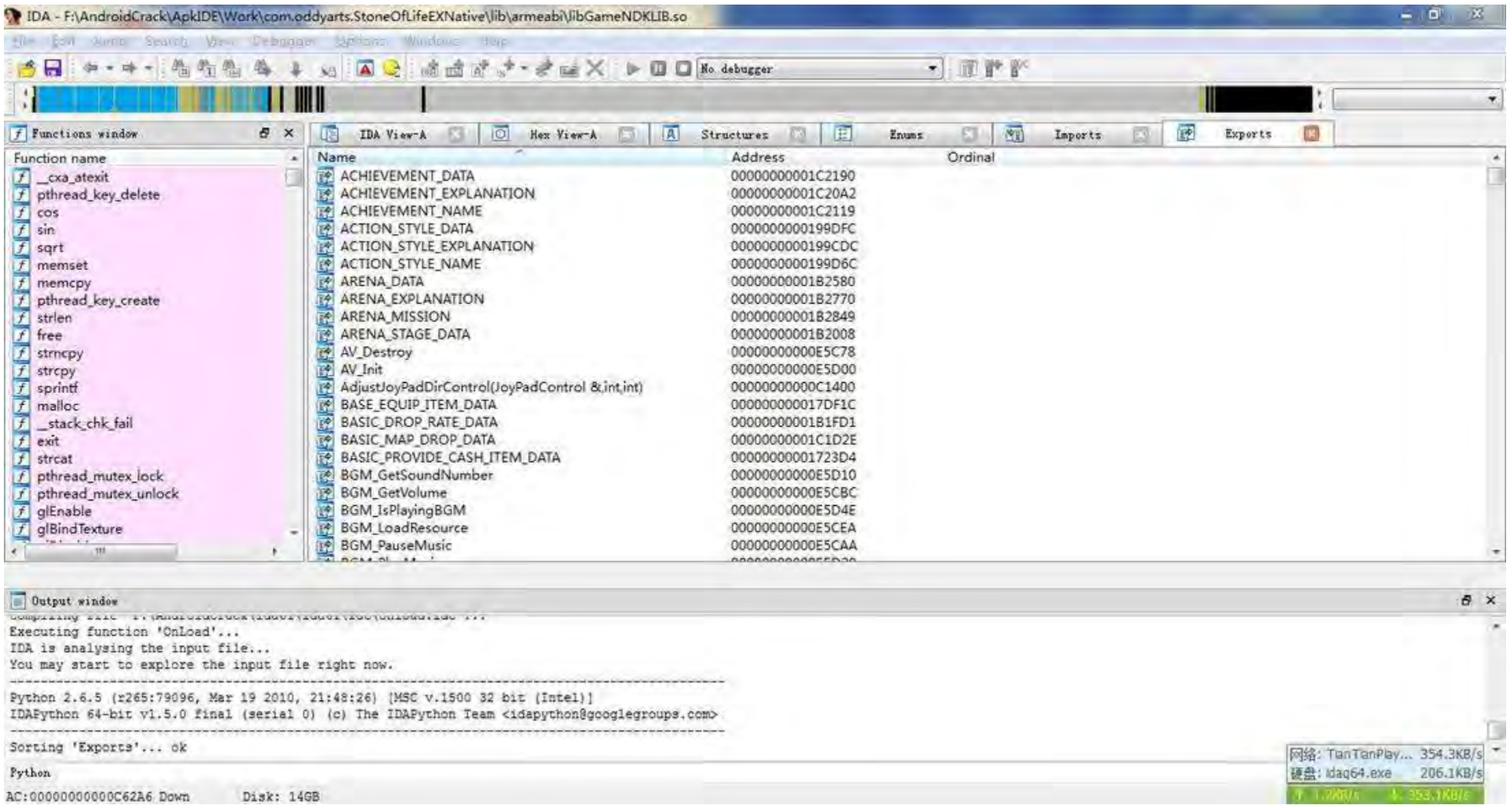
现在对游戏基本要修改的东西有了了解 我们可以开始反编译了
这里为了更好的让我们了解整个过程 我使用 IDE 进行反编译



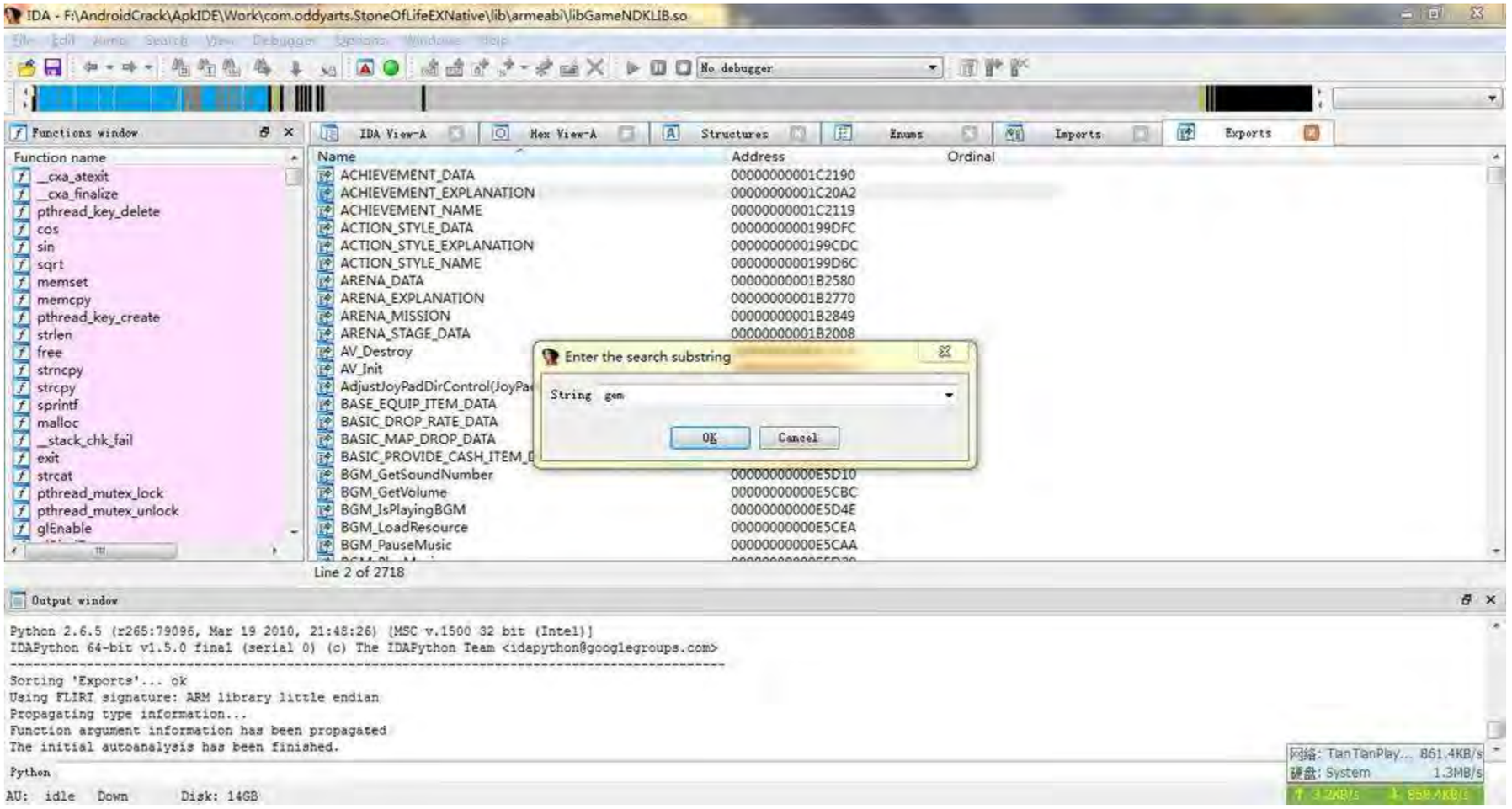
对于国外游戏 都很喜欢用 NDK 来开发 所以当我们看到如上图 lib 目录下有 SO 文件的时候 我们就应该先从 SO 文件下手 这时候要用 IDA 打开 SO 文件 IDA 的使用我就不多介绍了 不懂的人请看 IDA 使用相关指南



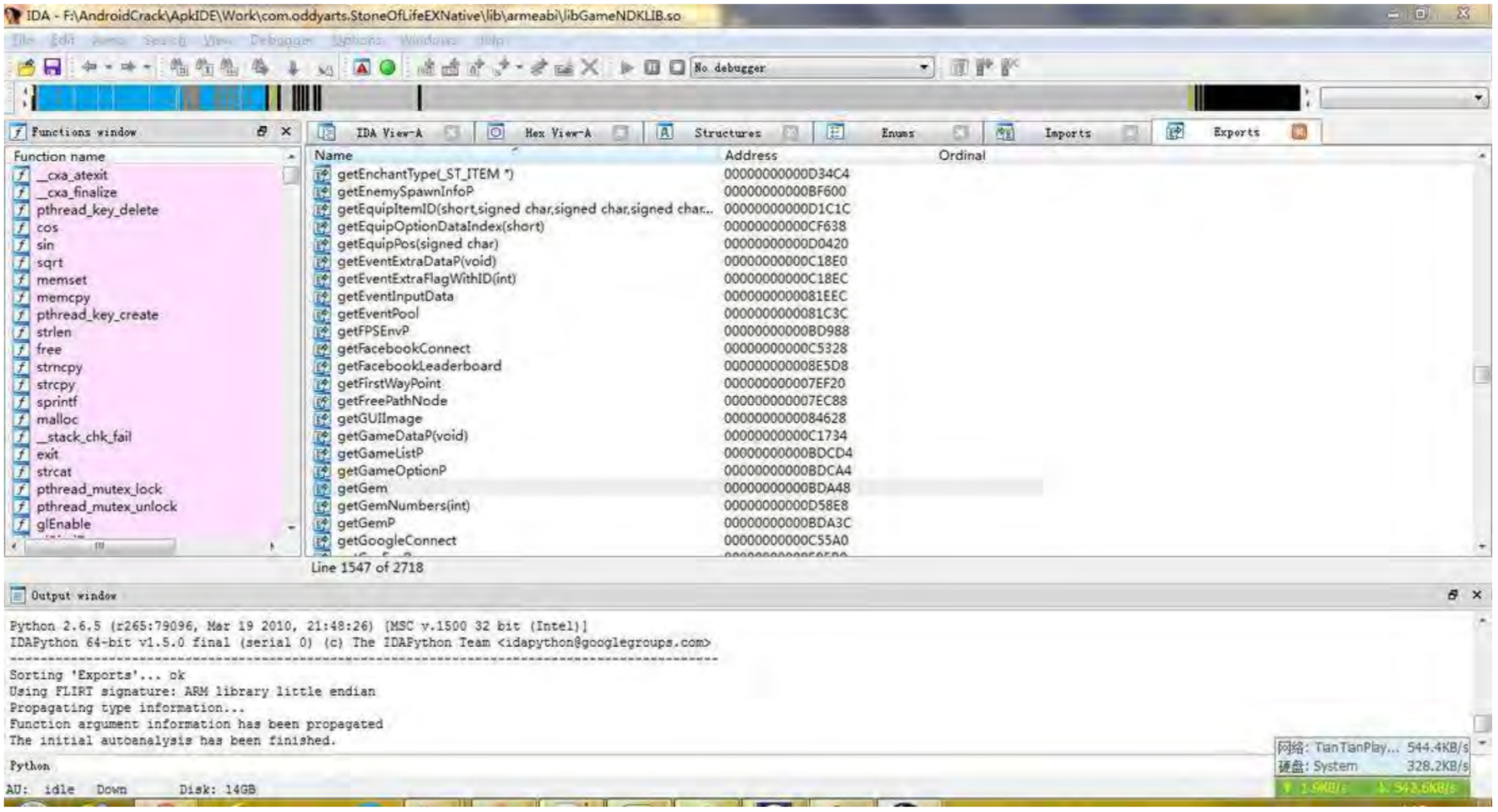
IDA 载入 SO 后 如上图 全部默认 无需设置 然后点 OK



打开后点击右上角的 Exports 观看函数



如上图 按 ALT+T 打开搜索窗口 然后输入“gem” 接着按 OK 开始搜索 要继续搜索下一个 就按 CTRL+T



CTRL+T 几次后 看到有个“getGem”的函数 就是获取钻石数量 我们直接双击 来看看代码

```
.text:000BDA48                 EXPORT getGem
.text:000BDA48 getGem                                  ; CODE XREF: draw_CraftMaterialInfo:loc_95794↑p
.text:000BDA48                                         ; draw_GUI_CUSTOM_TOOL_GEMBOX+2C↑p ...
.text:000BDA48                 LDR     R3, =(m_gem_ptr - 0xBDA4E) ;   //获取gem的基址
.text:000BDA4A                 ADD     R3, PC          ;   //gem基址+偏移
.text:000BDA4C                 LDR     R3, [R3]        ;  //获取gem存放位置
.text:000BDA4E                 LDR     R0, [R3]        ;  //获取gem数量 并将数量保存到R0里
.text:000BDA50                 BX      LR
.text:000BDA50 ; End of function getGem
```

从图中的解释 我们可以知道 gem 的数量在 R0 里 所以我们只要修改 R0 的数量即可

接着点击 Hex View-A 进入 16 进界面 我们来看看对应机器码

整句 getGem 的代码

000BDA48 02 4B 7B 44 1B 68 18 68 70 47

我们直接将

```
LDR     R3, =(m_gem_ptr - 0xBDA4E)
ADD     R3, PC
LDR     R3, [R3]
LDR     R0, [R3]
BX      LR
```

修改为

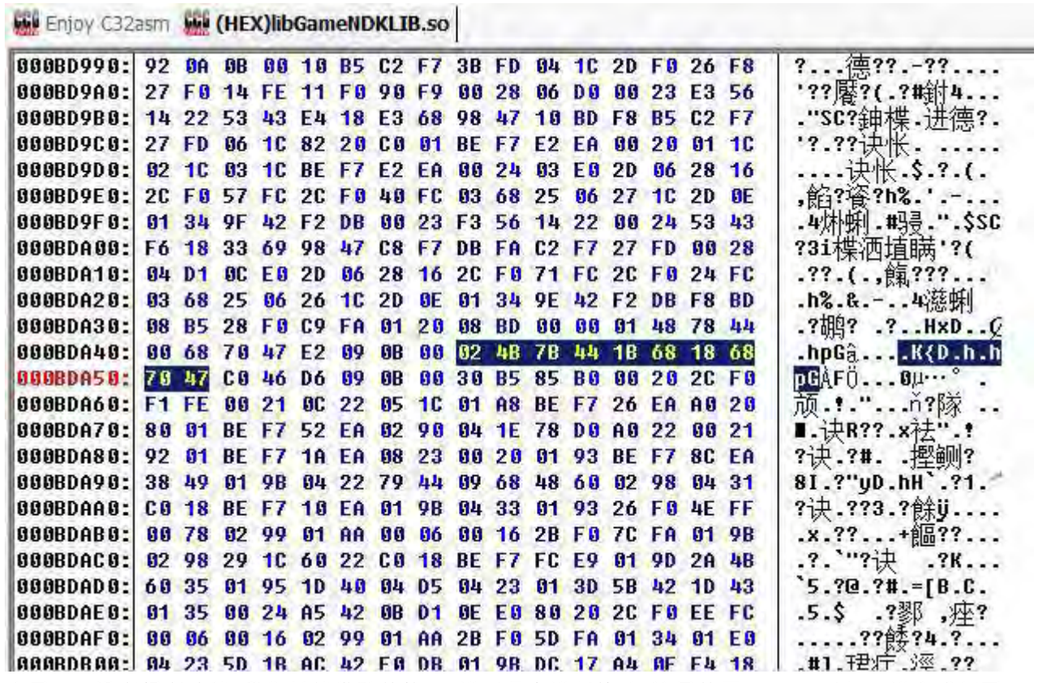
```
MOVS    R0,0x2800       //0x2800 =10240  也就是 gem 数量  =10240 个
BX          LR
```

对应机器码是

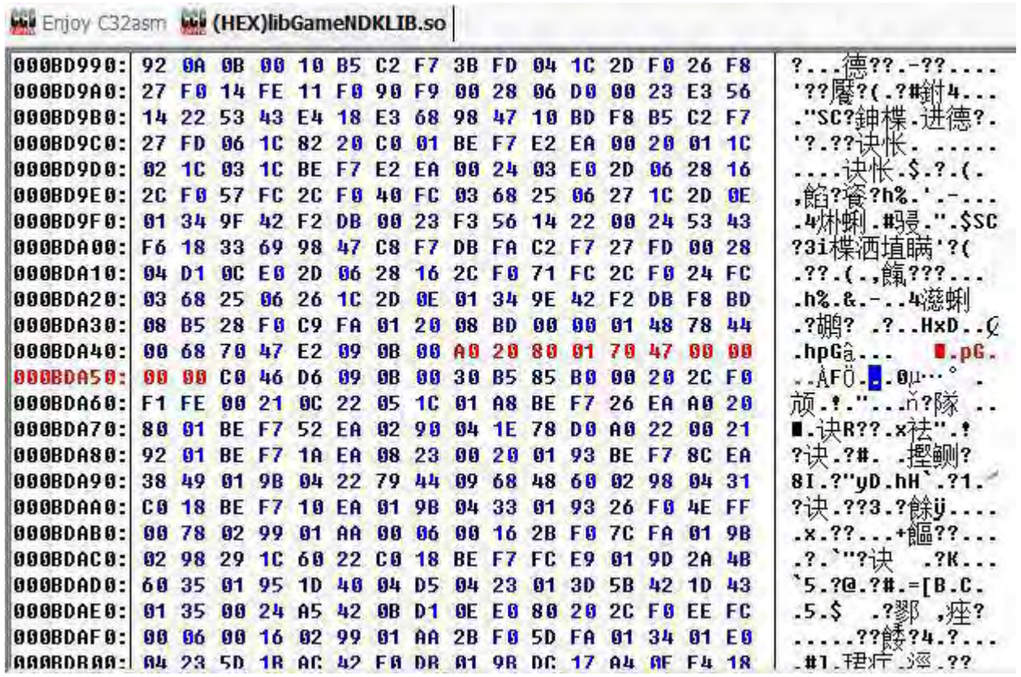
02 4B 7B 44 1B 68 18 68 70 47

改为

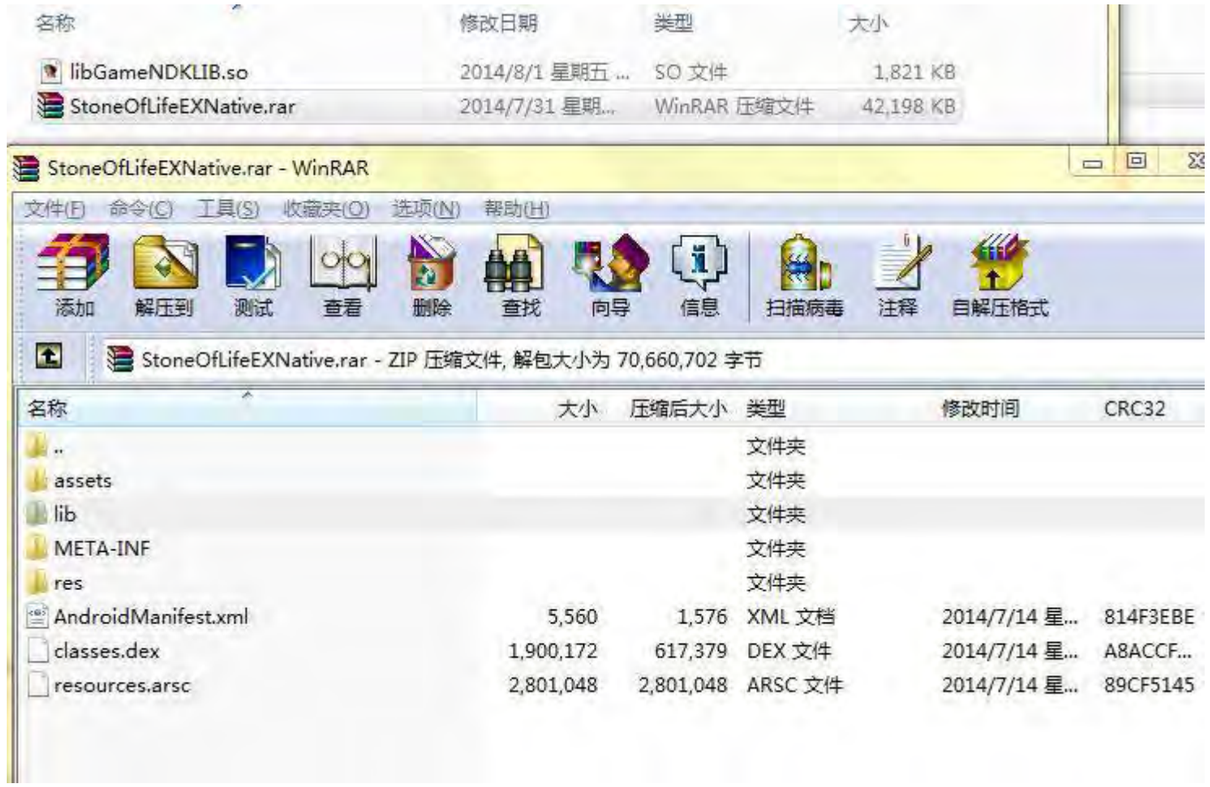
A0 20 80 01 70 47 00 00 00 00



由于 IDA 没有保存功能 我们只能借助其他 16 进工具来进行修改 这里使用 C32ASM 以 16 进方式打开 CTRL+G 搜索对应地址 “000BDA48" 如上图高亮的地方



修改为如上图的结果 然后点击左上角的保存 此时 我们已经完成修改了 现在可以进行打包签名了



由于游戏有保护 导致 IDE 无法直接回编 这里我们直接将游戏 APK 包后缀改为 rar 类型 如上图
然后把修改好的 SO 文件 拖到“lib\armeabi”目录里

修改好后再把后缀名改回 apk 接着用签名工具进行签名 apk 文件 签名好后会生成新的已签名文件

接着我们把机子上原版卸载 重新安装我们修改后的版本

如上图我们可以看到 gem 的数量=10240 证明我们修改成功了 真的成功了吗？ 那我们来验证下 随便购买个物品看看

嘿嘿 有购买成功的提示了 到这里 表示我们已经破解成功了
教程就到这里 感谢大家收看

跟着鬼哥学 so 修改，二，进行篇

图/文　听鬼哥说故事

继续上文的内容----------

0x1:测试文件的编写

经过上一篇文章的基础学习，现在我们开始进行是用的部分。

既然我们可以在 so 中定义 String 字符串了，那么我们当然也可以定义 int 类型的数据了，那么，我们在此定义一个 getCoin 方法，返回值为 int 类型。如下：

```
JNIEXPORT  jint  JNICALL  Java_com_ggndktest1_JniGg_getCoin

    (JNIEnv  *  env,  jobject  this)

{

int  c=100;

return  c;

}
```

同时编写 java 代码：

```
static  public  native  int  getCoin();
```

这两个方法的添加，可以继续在我们上一篇文章的基础上进行补充即可，当然，为了熟练创建 ndk 工程，也可以重新创建工程的。

然后我们在 shell 下切换到工程根目录：

```
E:\mmmz\GGNdkTest1>ndk-build
"Compile thumb  : gg-jni <= gg-jni.c
SharedLibrary   : libgg-jni.so
Install         : libgg-jni.so => libs/armeabi/libgg-jni.so
```


这样成功生成 so 文件。

然后我们简单在 xml 定义下布局，简单的，直接在 layout 下

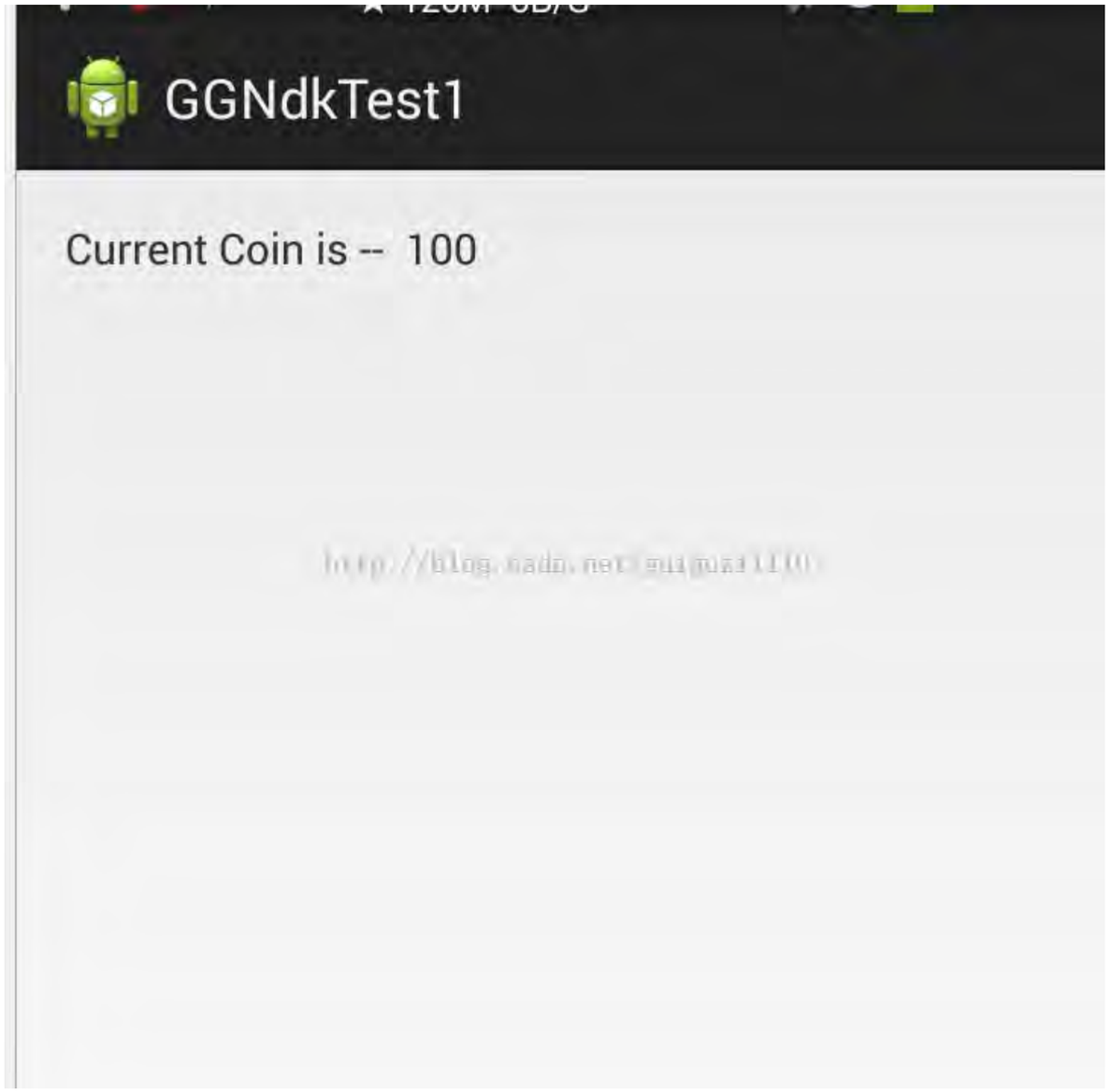
```
<TextView
```

```
android:id="@+id/coin"

android:layout_width="wrap_content"

android:layout_height="wrap_content"

android:text=""  />
```

添加一个 TextView，设置 id 为 coin，然后就可以在主类进行引用。

```
public class MainActivity extends Activity {


    TextView tview;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        tview=(TextView) this.findViewById(R.id.coin);

        tview.setText(JniGg.ggPrintHello()+JniGg.getCoin());

    }
}
```


这样我们运行一下程序，查看一下效果。


GGNdkTest1
Current Coin is -- 100

0x2：任务明确

我们现在需要做一个任务，就是修改上文程序中的金币数量。

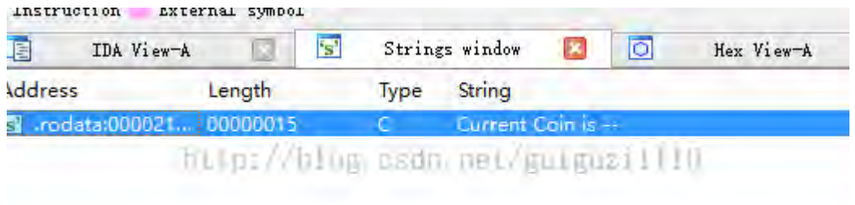
我们直接在工程的 bin 目录下拿出来未签名的 apk，分析时可以先不管签名，我们把 lib/armeabi 下的 so 文件拿出来，拖进 IDA 进行分析。

0x3:挂起 IDA，进行分析

（假设，当前我们的状态是不知道程序源码）

通过上文运行的截图，我们可以分析到，有关键字“Current Coin ”,我们在对反编译后的程序分析搜索，发现只存在于 so 文件中，这样，我们就直接在 IDA 中搜索字符串去。

快捷键　Shift + F12

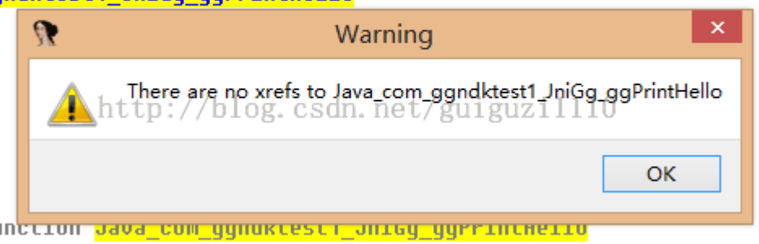


找到它的引用

```
                EXPORT Java_com_ggndktest1_JniGg_ggPrintHello
Java_com_ggndktest1_JniGg_ggPrintHello
                PUSH    {R3,LR}
                LDR     R2, [R0]
                LDR     R1, =(aCurrentCoinIs - 0xBFA)
                MOVS    R3, #0x29C
                ADD     R1, PC          ; "Current Coin is --  "
                LDR     R3, [R2,R3]
                BLX     R3
                POP     {R3,PC}
; End of function Java_com_ggndktest1_JniGg_ggPrintHello

; ---------------------------------------------------------------------------
```

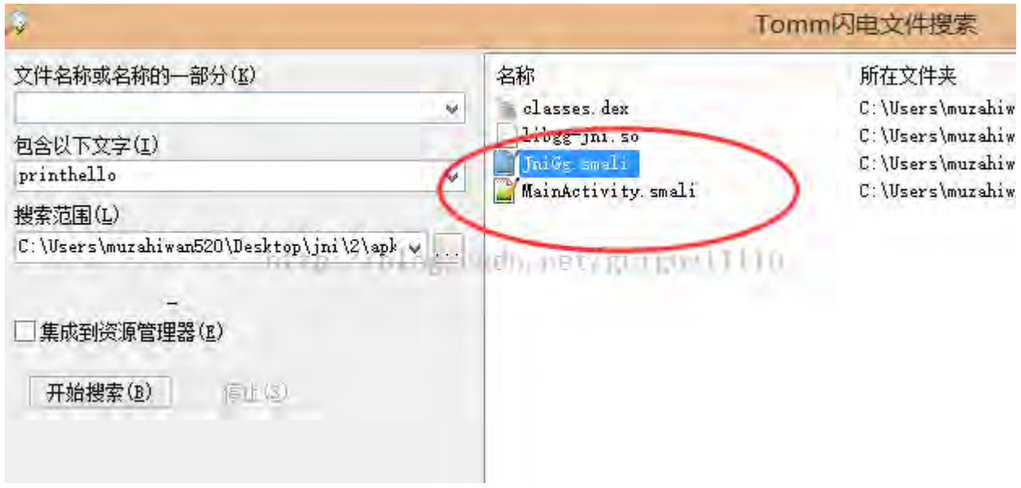

这个没啥好解释的，加载字符串而已，没啥重要信息，发现不出来金币在哪里定义了，那么我们就看下引用。

```
                EXPORT Java_com_ggndktest1_JniGg_ggPrintHello
Java_com_ggndktest1_JniGg_ggPrintHello
```




```
; End of function Java_com_ggndktest1_JniGg_ggPrintHello
```

发现在 so 文件中，并没有函数对此方法进行调用，那么我们就返回 java 中，看看 java 语句中是怎么对它进行的调用，其上下文都是什么东西。



经过查看 printhello 方法在 JniGg.smali 中是定义，在 MainActivity 是调用，我们详细看下代码：

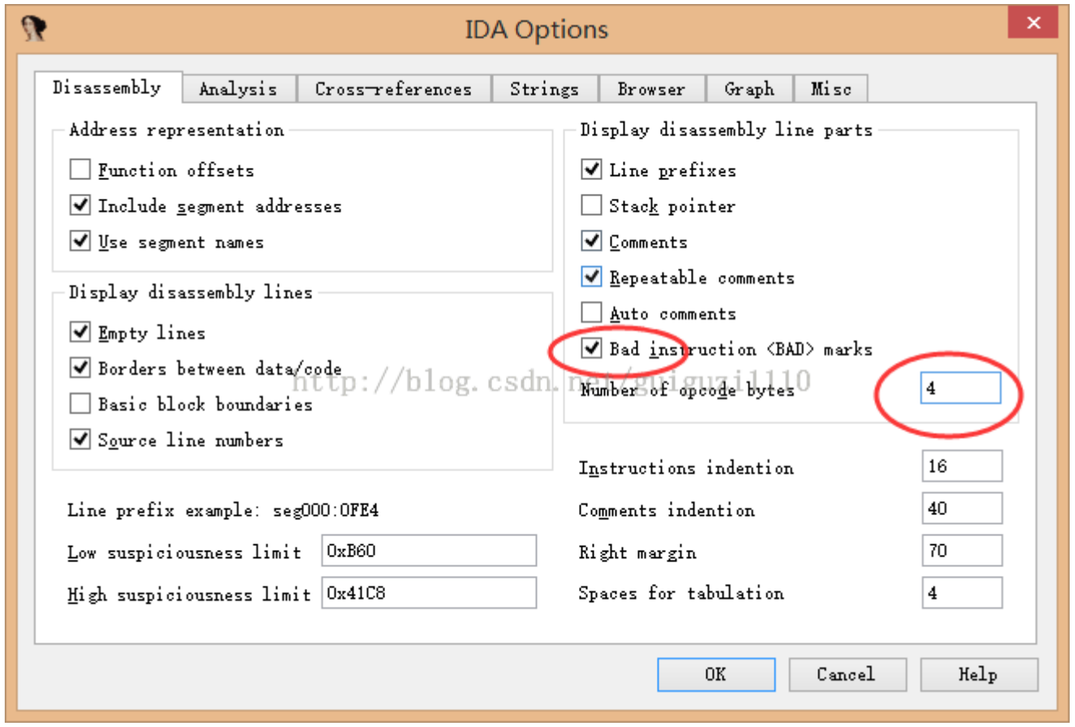
```
.line 22
iget-object v0, p0, Lcom/ggndktest1/MainActivity;->tview:Landroid/widget/TextView;

new-instance v1, Ljava/lang/StringBuilder;      #创建StringBUilder对象

invoke-static {}, Lcom/ggndktest1/JniGg;->ggPrintHello()Ljava/lang/String;    #调用so中的PrintHello方法，即 Current Coin 字符串

move-result-object v2

invoke-static {v2}, Ljava/lang/String;->valueOf(Ljava/lang/Object;)Ljava/lang/String;

move-result-object v2

invoke-direct {v1, v2}, Ljava/lang/StringBuilder;-><init>(Ljava/lang/String;)V

invoke-static {}, Lcom/ggndktest1/JniGg;->getCoin()I     #获取一个Int数值

move-result v2

invoke-virtual {v1, v2}, Ljava/lang/StringBuilder;->append(I)Ljava/lang/StringBuilder;

move-result-object v1

invoke-virtual {v1}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;

move-result-object v1

invoke-virtual {v0, v1}, Landroid/widget/TextView;->setText(Ljava/lang/CharSequence;)V     #给textview设置刚才的StringBuilder的toString()后的字符串

.line 24
return-void
```


贴上简单的说明，大家就可以看的清楚了。

我们在这里发现一个关键函数，getCoin 方法，通过向上追寻，发现也是在 so 文件中的，那么现在我们再次回到 IDA 中：

```
                EXPORT Java_com_ggndktest1_JniGg_getCoin
Java_com_ggndktest1_JniGg_getCoin
                MOVS    R0, #0x64
                BX      LR
; End of function Java_com_ggndktest1_JniGg_getCoin
```


在 Options 打开 General，也就是第一个选项，设置如下：

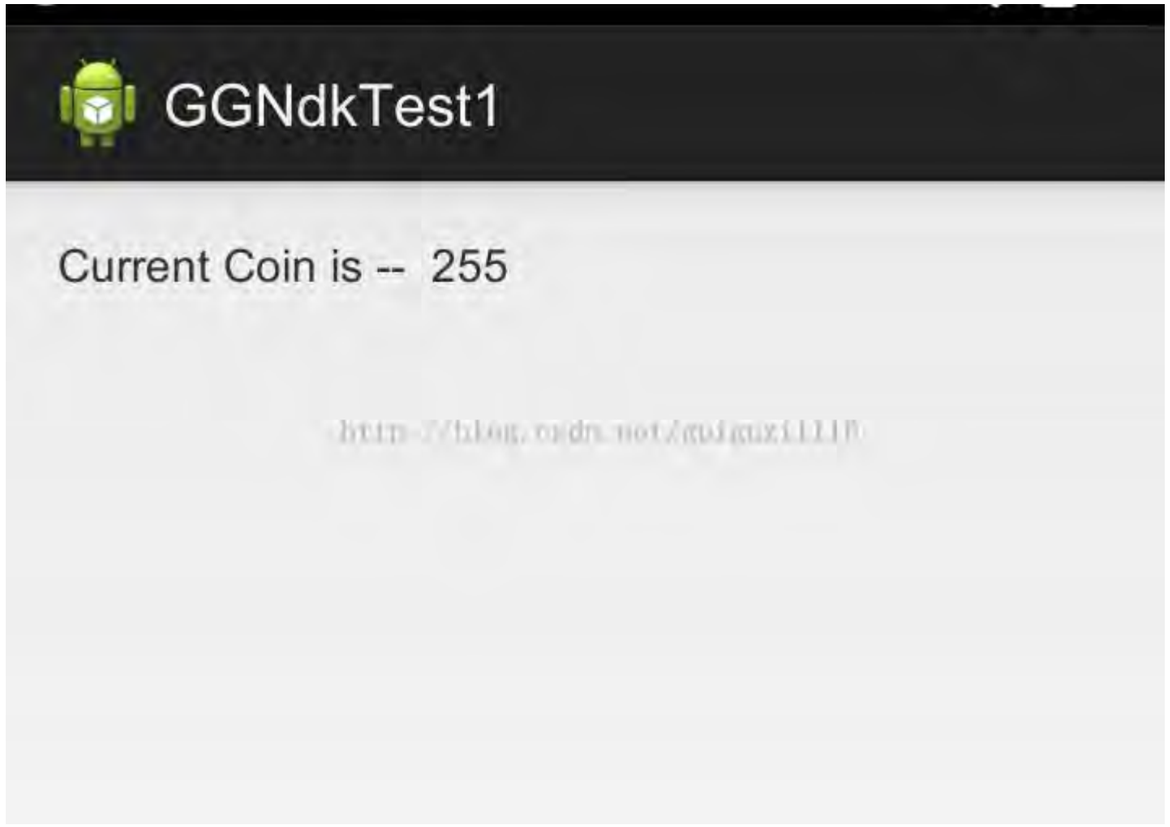


然后我们可以看到：

```
.text:00000C04
.text:00000C04
.text:00000C04                          EXPORT Java_com_ggndktest1_JniGg_getCoin
.text:00000C04              Java_com_ggndktest1_JniGg_getCoin
.text:00000C04 64 20                    MOVS    R0, #0x64
.text:00000C06 70 47                    BX      LR
.text:00000C06              ; End of function Java_com_ggndktest1_JniGg_getCoin
.text:00000C06
.text:00000C08                          CODE32
```


这个是 Thumb 的指令，那么我们修改，也要使用 Thumb。

一个字节来定义数值，我们只能用最大 ff 了，即 255，那么在 16 进制编辑器打开，跳转到 c04 的位置，修改为 FF 20，然后放回程序，签名，运行。



好了，说明我们的修改成功。

0x4:反思一下

既然我们是自己写的源码，那么我们就可以很好的来对比思考了，这个过程留给同学们自己来思考，请花上 10 分钟时间，对比一下代码里面编写的方法，与在 so 中反汇编的代码的联系，增强一下熟悉的感觉。

另：

上面的 apk，还有其他破解方法，即我们找到 java 层的调用了，那么直接修改 java 层调用方法的返回值，效果也一样，可以修改为无限大的数值。

相关附件地址：

http://pan.baidu.com/s/1i3wzetf

文章同时也交由 www.pd521.com 首发，转载请注明出处。

小黄人快跑之 so 文件的简单分析

游戏名称：小黄人快跑

测试类型：修改香蕉数量（类似金币）

主要方式：IDA 进行分析，修改 so 文件

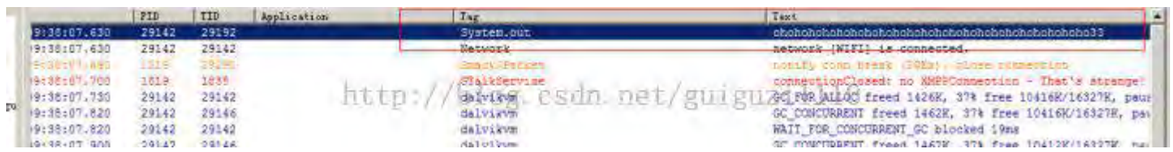
今天我们来分析的是小黄人快跑，小黄人的形象从电影上面一直都蛮好的，现在都拍到了第二部了，所以我们找到这个游戏来进行分析。

第一层次分析：

拿到游戏，首先我们得先安装看看有什么地方是值得破解的。

通过观察，我们发现游戏中主要的消费方式在香蕉和金币上面，既然这样，那么我们就先看看支付上面是否可以直接破解内购。购买截图如下图所示：

和游戏支付的方式，那么就查询相关资料，例如以前我的教程中对 onPayFailed 方法的修改，对 onBillingFinish 方法的修改，在这里可能是对移动支付进行了升级，没法直接那样操作。既然这样，那么我们就查看下 Logcat，看看有无有价值的信息。

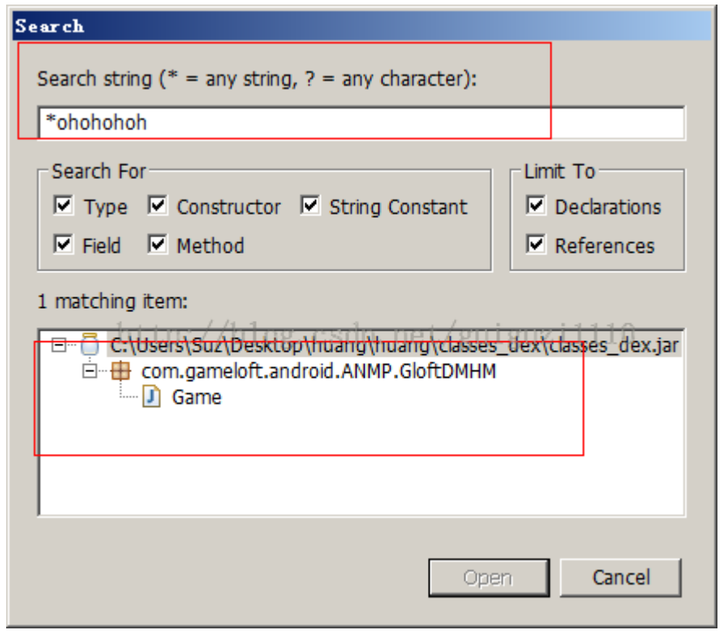


通过多次测试，我们发现在点击进入游戏支付的时候有这行输出，那么好，我们就跟进观察这行代码，这行代码后面肯定是启动支付流程的相关代码。

```
invoke-direct {v0, v1}, Ljava/text/SimpleDateFormat;-><init>(Ljava/lang/String;)V

sget-object v0, Lcom/gameloft/android/ANMP/GloftDMHM/Game;->N:Lcom/gameloft/android/ANMP/GloftDMHM/Game;

new-instance v1, Lcom/gameloft/android/ANMP/GloftDMHM/x;

invoke-direct {v1}, Lcom/gameloft/android/ANMP/GloftDMHM/x;-><init>()V

invoke-virtual {v0, v1}, Lcom/gameloft/android/ANMP/GloftDMHM/Game;->runOnUiThread(Ljava/lang/Runnable;)V

sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;

new-instance v1, Ljava/lang/StringBuilder;

const-string v2, "ohohohohohohohohohohohohohohohohohohohohoho"

invoke-direct {v1, v2}, Ljava/lang/StringBuilder;-><init>(Ljava/lang/String;)V

invoke-virtual {v1, p0}, Ljava/lang/StringBuilder;->append(I)Ljava/lang/StringBuilder;

move-result-object v1

invoke-virtual {v1}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;

move-result-object v1
```

这行代码被调用，所以打开观察它

对于 smali 代码不熟悉的同学，那么直接使用 jd-gui 打开 jar 文件查看吧，如下：



搜索找到我们发现的特征字符，如下：

```
public static void Billing(int paramInt1, int paramInt2)
{
  if ((paramInt1 > 33) || (paramInt2 == 1))
    v = true;
  if (!v)
  {
    v = true;
    m = paramInt1;
    new SimpleDateFormat("yyyyMMddHHmmss");
    N.runOnUiThread(new x());
    System.out.println("ohohohohohohohohohohohohohohohohohohohohohohoho" + paramInt1);
  }
}
```


这样，我们直接查看 x()类：

```
{
  return;
  GameInterface.doBilling(Game.N, true, true, "018", null, Game.u);
  continue;
  GameInterface.doBilling(Game.N, true, true, "005", null, Game.u);
  continue;
  GameInterface.doBilling(Game.N, true, true, "006", null, Game.u);
  continue;
  GameInterface.doBilling(Game.N, true, true, "014", null, Game.u);
  continue;
  GameInterface.doBilling(Game.N, true, true, "015", null, Game.u);
```

我们可以发现，doBilling 方法，按照名称来想，我们也感觉就是购买的时候使用的，所以我们追寻它去。

全文搜索 doBilling 效果不太理想，所以我们可以直接找 GameInterface 这个类的相关调用方式了。

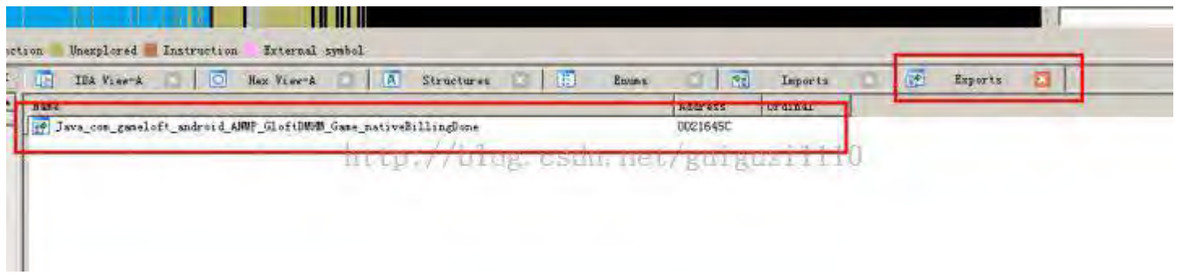
```
package com.gameloft.android.ANMP.GloftDMHM;

import cn.cmgame.billing.api.GameInterface.IPayCallback;

final class m
  implements GameInterface.IPayCallback
{
  public final void onResult(int paramInt, String paramString, Object
  {
    switch (paramInt)
    {
    default:
      Game.w = false;
      Game.nativeShowedBilling();
      Game.nativeActiveoffline(1);
    case 1:
    case 2:
    }
    while (true)
    {
      return;
      Game.w = false;
      Game.nativeBillingDone(Game.m);
      continue;
      Game.w = false;
      Game.nativeShowedBilling();
      Game.nativeActiveoffline(1);
    }
  }
```


好了，到了这个类，根据接口名称，我们可以猜到，这个是支付的回调函数。回调函数的意思，也就是判断是否支付了，然后返回相关数据，由其他算法来进行香蕉或金币数量的增加。

Game.nativeBillingDone,看到这个 native 方法，我们就需要对 so 文件进行分析操作了，我们找到 Game 类，看看是调用的哪个 so 文件，搜索 system.loadlibrary：

```
  }
  SUtils.setContext(this);
  System.loadLibrary("despicablemefree");
  if (!ay)
  {
    Device.init();
```


晓得了 so 的名称，那么我们就挂起 ida，java 可以直接调用使用的肯定都是 export 导出函数，所以我们在 export 中搜索 nativeBillingDone:



然后看到这个方法：

```
; ================ S U B R O U T I N E =======================================

                EXPORT Java_com_gameloft_android_ANMP_GloftDMHM_Game_nativeBillingDone
Java_com_gameloft_android_ANMP_GloftDMHM_Game_nativeBillingDone
                MOV             R0, R2
                B               addCash
; End of function Java_com_gameloft_android_ANMP_GloftDMHM_Game_nativeBillingDone
```

B 是调用方法的意思，这里调用 addCash 方法，所以我们更清楚的明白，这个就是增加金币阿，名字都这么通俗易懂了，再猜不到就对不起人家作者了。

先大致浏览一下方法是干什么的：

```
                EXPORT addCash
addCash                                 ; CODE XREF: Java_com_gameloft_android_ANMP_GloftDMHM_Game
                STMFD           SP!, {R3-R8,R10,LR}
                CMP             R0, #1  R0为传入的参数，这里跟1做比较
                LDR             R5, =(_GLOBAL_OFFSET_TABLE_ - 0x209014)
                MOV             R6, R0 将传入的参数R0赋值给R6
                ADD             R5, PC, R5 ; _GLOBAL_OFFSET_TABLE_
                BNE             loc_209028 上面跟1比较，比较完之后，看结果来进行方法的条换
                LDR             R3, =(off_A45CC4 - 0xA48944)
                MOV             R2, #2
                LDR             R3, [R5,R3] ; unk_A56BE0
                STR             R2, [R3]
                LDMFD           SP!, {R3-R8,R10,PC}
```

好，大致看懂第一段代码后，我们往下翻翻：

```
loc_2090D4                              ; CODE XREF: addCash+98↑j
                CMP             R6, #8  上面我们看的，将传入参数R0赋值给了R6，用R6跟立即数8比较
                BNE             loc_209118 比较完之后的跳转指令
                BL              sub_3CAC74
                CMP             R0, #0
                BEQ             loc_209100
                LDR             R0, [R4]
                MOV             R1, #0x49C8     这样代码要引起重视，因为在游戏中，我们购买见到了18888这个数
                BL              sub_208DAC      字也就是说，如果R6=8,那么就会执行这些代码。
                LDR             R0, [R4]
                MOV             R1, #0
                BL              sub_3CAC90

loc_209100                              ; CODE XREF: addCash+E8↑j
                LDR             R3, =(off_A45CD4 - 0xA48944)
                MOV             R2, #1
                MOV             R0, #8
                LDR             R3, [R5,R3] ; unk_A56BEC
                STR             R2, [R3]
                B               loc_209070
```


通过图片中的文字说明，我们也简单能够明白，我们需要做的，就是先测试一下，将这个 addcash(int)方法的传入参数设置为 8 会怎么样，是不是会增加金币。所以我们回到最初分析的时候，看这里：

```
Game.v = false;
Game.nativeBillingDone(Game.m);
continue;
```


我们通过查找可以知道 Game.m 就是一个 int 类型的数据，那么，我们就明白了，我们可以直接操作 Game.m 的数值，将其设置为 8 即可，同时将这行 Game.nativeBillingDone 调用方法放到一个我们明显能用按钮控制的位置。

```
0:10:13.540   29142   29192   System.out   soundofftrue
```

阿门，感谢作者，看到这行输出，这个是我们点击支付界面的返回按钮产生的，因为支付过程中会将游戏声音暂停，所以才会产生这个情况，方便开发者观察数据。

起来，我们是可以通过自己在主类 Game.smali 中自己寻找其他地方的，能看到 log 只是更加方便了我们的分析速度而已，我们手动寻找代码也是可以的。所以没有发现 log，只是在时间上面花费会多一点，其他影响是不大的。

好，那么我们就直接找到这行 log 的打印地方，在它上面调用 nativeBillingDone 方法：

```
.method public static isSoundOff()I
    .locals 3

    const/16 v1 , 0x8
    invoke-static {v1}, LILOVEGG;->LogInt(I)V

    invoke-static {v1}, Lcom/gameloft/android/ANMP/GloftDMHM/Game;->nativeBillingDone(I)V
```


第一行，设置 v1=8，第二行，调用我自己的方法打印 int 数据看是否赋值成功，同学们可以忽略掉的，第三行调用 natveBillingDone 方法。

然后回编译，打包测试.....

发现初始化金币真的为 18888，然后点击商店，支付页面，按返回键：

好了，到了这里，我们的第一层次的目的就完成了。

需要说明的是，IDA 的使用，同学们自己查资料，ARM 语法，自己查资料即可，网上有很多教程的，自己动手，掌握更深入。

第二层次分析：

进行完第一层次分析后，我们知道已经分析对了位置。So 文件中的算法也找对了，那么，我们初始化多增加点金币多好，于是，我们继续开始分析。

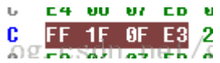
关键点就在 addcash 方法中的这一行对 R1 复制的代码上面，我们打开 16 进制看一眼：

只有四个字节，通过这 4 个字节实现将 18888 赋值给 R1,4 个字节最大的数字为 0xffff，要看这个函数接收的是无符号，还是有符号类型的 int16，现在最大数值可能就是 0xffff 对应换算到十进制为 65535。

好，那么我们就看看怎么来修改数据吧，立即数，我们就可以直接操作修改字节了。

C8 19 04 E3 ，这行代码原本为 MOV R1, #0x49C8

关于指令命令本人也不太熟悉，所以直接慢慢试着改一下



因为那个本身的立即数，通过观察发现原本的指令刚好反过来为 E3 04 19 C8 ，那么对照 0x49C8，我直接修改为如上图所示：

发现直接修改成功了，好了，这样就更给力了，我们直接使用 010Editor 或者 UE，操作 so 文件

前面蓝色部分为内存地址，我们使用 ctrl + g 命令跳转地址，直接对应修改这四个字节码即可。

然后重新打包，测试。。

结果自己可以看到，一切正常，初始化 65535。。

支付页面，返回，增加金币。。

我们这次测试的是香蕉的增加方式，所以还有金币的增加噢，这个就留给感兴趣的同学们自己尝试了，自己动手，多多练习~~~

仅供内部交流，请勿做其他商业用途！

听鬼哥说故事

TTX 连萌

听鬼哥说故事

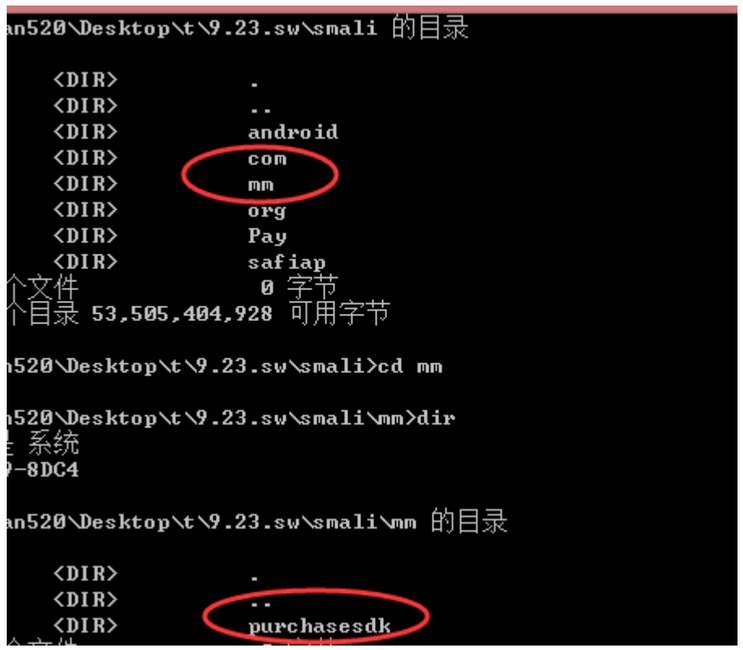
最近工作太忙，没啥时间写文章，刚好今天遇到一个小游戏，简单分析下，写下此文。

移动 MM 的游戏，前面我们已经写过很多文章，没有看过的朋友，自行查找即可，今天我们继续分析一个类似的游戏，不过使用多种方式来分析，同时，欢迎同学们自己补充新方式来扩展思路。

0x1:游戏试玩

打开游戏玩了会儿，最近这类游戏非常多，也谈不上喜欢玩不喜欢玩，那么直接查看商城吧。

配合查看游戏反编译的目录结构：



可以简单猜测游戏的内购是否为移动 MM 的。

然后配合我们自己简单查看下 smali 代码，确定内购支付方式为移动 MM .

0x2:分析破解

确定了游戏的支付方式，我们就有目的性的去分析如何来分析它的破解情况了。

在这里再次补充上：

移动 MM 的支付方式以及 SDK 的相关调用方法

http://wenku.it168.com/d_001271444.shtm

通过了解它的支付流程，来测试他在哪块地方存在被破解的可能，即欺骗支付。

在这篇文章中，已经有简单说明了，这里便不再重复：

http://www.52pojie.cn/thread-259909-1-1.html

然后，我们可以想到：

①.我们直接修改支付结果，用支付成功的方法替换支付失败

②.直接将判断支付是否成功的状态码锁定为支付成功

③.直接修改游戏金币

④.修改支付短信

0x3:第一种支付方式破解

还是上面说的，直接搜索寻找 onBillingFinish。

这里所说按照第一种方式，有点牵强，不过，支付失败的话，这里的 code 也将改变，我们让他走订购成功的方法，即修改方法内的那个判断语句。

```
public void onBillingFinish(int code, HashMap arg1) {
    Log.d("IAPListener", "billing finish, status code = " + code);
    String v0 = "订购结果：订购成功";
    if(code == 102 || code == 104 || code == 1001) {
        Log.d("IAPListener", "Success");
        SDKInvoker.getInstance(PopStar.mMain).onBillingFinish(0, "");
    }
    else {
        v0 = "订购结果：" + Purchase.getReason(code);
    }

    Log.d("IAPListener", v0);
}
```

关键修改点

对应 smali 代码自行修改即可。

0x4:第二种锁定状态码破解

我们在追踪 onBillingFinish 方法上下文的时候，发现 PurchaseCode.smali 中的 getStatusCode（）I 方法以及 MessageInfo.smali 中的 getPurchasecode()I 方法，应该是用于定义支付码状态，即支付成功，支付失败，取消支付的。

在 PurchaseCode.smali 文件中，我们发现：

.field public static final WEAK_ORDER_OK:I = 0x3e9

这个是定义支付订单成功的状态码。一般在移动 MM 里面，多数都是这个。

所以，我们可以看到上面所说 getStatusCode（）I 方法和 getPurchasecode()I 方法都是返回值为 int 类型的，那么我们直接将其返回值修改为 0x3e9，即数字 1001，即代表支付成功。

我们保存修改，回编译，查看到，点击购买按钮，直接提示下图：

在无卡模式下测试的，一切正常，所以肯定了我们的修改时正确的，也不会扣费。

0x5:直接修改金币

通过 OnBillingFinish()方法

```
String v0 = "订购结果：订购成功";
if(code == 102 || code == 104 || code == 1001) {
    Log.d("IAPListener", "Success");
    SDKInvoker.getInstance(PopStar.mMain).onBillingFinish(0, "");
}
else {
    v0 = "订购结果：" + Purchase.getReason(code);
```

我们进入到这个 PopStar 查看：

```
public void addCoin(int arg5) {
    LogUtil.log("addCoin.index = " + arg5);
    PopStar.nativeAddCoin(arg5);
}
```

当然，这个类里面还有很多方法，很多支持破解的方法，如:

```
private void payIAP(int arg5) {
    LogUtil.log("payIAP.index = " + arg5);
    this.setType(arg5);
    int v1 = 1500;
    switch(arg5) {
        case 0: {
            v1 = 10;
            break;
        }
        case 1: {
            v1 = 300;
            break;
        }
        case 2: {
            v1 = 600;
            break;
        }
        case 3: {
```

定义购买金币数量的，即购买成功后增加的数量。等等。。。

长话短说，咱们看到 PopStar.nativeAddCoin 方法后，确定是一个 native 方法，那么我们找到上文 `System.loadLibrary("xinxin");`

用 IDA 加载 libxinxin.so,然后定位到 nativeAddCoin 方法.

现在大家多数用的大佬的那个 IDA，带 F5 的，那么我们直接 f5

```
void __fastcall Java_com_PopStar_org_PopStar_nativeAddCoin(int a1, int a2, int a3)
{
  int v3; // r0@3
  signed int v4; // r3@3
  signed int v5; // r0@20

  if ( a3 != -1 && Shop::ArrayPriceSize )
  {
    v3 = 100;
    v4 = Shop::ArrayPrice[a3];
    if ( v4 == 1000 )
      goto LABEL_24;
    if ( v4 <= 1000 )
    {
      if ( v4 == 10 )
        goto LABEL_27;
      if ( v4 > 10 )
      {
        v3 = 25;
        if ( v4 != 300 )
        {
          if ( v4 == 600 )
            v3 = 55;
        }
        goto LABEL_24;
      }
      if ( v4 == 1 )
LABEL_27:
        v3 = 10;
LABEL_24:
      GameMain::useGameCoin(v3);
      return;
    }
    if ( v4 == 2000 )
    {
      v5 = 130;
000AA6C6 Java_com_PopStar_org_PopStar_nativeAddCoin:16
```

查看到这里的方法，useGameCoin 方法。打开这个方法查看:

```
; GameMain::useGameCoin(int)
                EXPORT _ZN8GameMain11useGameCoinEi
_ZN8GameMain11useGameCoinEi                ; CODE XREF: GameOver::GameOverRestryGame(cocos2d::CCObject *)+4A↓p
                                           ; Props::UseProps(int,int)+3E↓p ...
                PUSH    {R0-R2,R4-R7,LR}
                LDR     R5, =(aGamecoin - 0xA41E6)
                MOVS    R6, R0
                BL      _ZN7cocos2d13CCUserDefault17sharedUserDefaultEv ; 获取共享的数值，这里一般是游戏过关获取的金币
                ADD     R5, PC          ; "GameCoin"
                MOVS    R1, R5
                MOVS    R2, #0xA
                BL      _ZN7cocos2d13CCUserDefault16getIntegerForKeyEPKci ; 根据字符串GameCoin查找int数值，一般为本地储存的金币
                LDR     R4, =(_ZN8GameMain8GameCoinE_ptr - 0xA41F4)
                ADDS    R0, R0, R6      ; 这里是金币总数量
                ADD     R4, PC ; _ZN8GameMain8GameCoinE_ptr
                LDR     R4, [R4]        ; GameMain::GameCoin
                STR     R0, [R4]        ; 将金币数量再次储存
                BL      _ZN7cocos2d13CCUserDefault17sharedUserDefaultEv ; cocos2d::CCUserDefault::sharedUserDefault(void)
                LDR     R2, [R4]
                MOVS    R1, R5
                BL      _ZN7cocos2d13CCUserDefault16setIntegerForKeyEPKci ; cocos2d::CCUserDefault::setIntegerForKey(char const*,int)
                BL      _ZN7cocos2d13CCUserDefault17sharedUserDefaultEv ; cocos2d::CCUserDefault::sharedUserDefault(void)
                BL      _ZN7cocos2d13CCUserDefault5flushEv ; cocos2d::CCUserDefault::flush(void)
                CMP     R6, #0
```

这里有我加的注释，很清晰的，看不明白的，F5 一下也就知道了。

因为 getIntegerForKey，返回值为 R0，然后往下第二行代码，R0=R0+R6，所以我们关键是在这里修改 R0，即金币数量

Hex 查看一下二进制，这里为 Thumb 指令，那么我们可以操作局限性也很高了。

所以，这里提供一个思路，大家可以寻找更加简便的方法。

即，BL getIntegerForKey，我们直接复制 R0 一个数值，然后在下面 Adds R0,R0,R6 这里，直接对 R0 逻辑左移一下，即 LSL 一下，将其数值变大。

因为第一个 BL 是 4 个字节，我们直接修改 00 00 09 20 ，即 Movs R0,R0 代表没有任何操作，09 20 是将数字 9 赋值给 R0

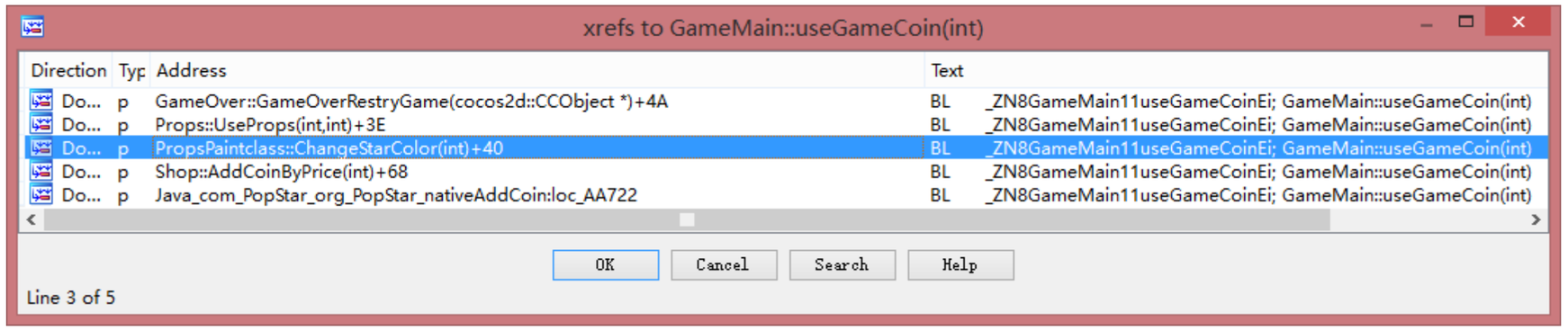
然后 ADDs 那一行，有两个字节，直接改为 00 04，即 LSLS R0，R0，#0x10

这样，我们即完成了对金币数量的赋值，如下图：

```
ext:000A41DA                LDR     R5, =(aGamecoin - 0xA41E6)
ext:000A41DC                MOVS    R6, R0
ext:000A41DE                BL      _ZN7cocos2d13CCUserDefault17sharedUserDefaultEv ; cocos2d::CCUserDefault::sharedUserDefault(void)
ext:000A41E2                ADD     R5, PC          ; "GameCoin"
ext:000A41E4                MOVS    R1, R5
ext:000A41E6                MOVS    R2, #0xA
ext:000A41E8                MOVS    R0, R0
ext:000A41EA                MOVS    R0, #9
ext:000A41EC                LDR     R4, =(_ZN8GameMain8GameCoinE_ptr - 0xA41F4)
ext:000A41EE                LSLS    R0, R0, #0x10
ext:000A41F0                ADD     R4, PC ; _ZN8GameMain8GameCoinE_ptr
ext:000A41F2                LDR     R4, [R4]        ; GameMain::GameCoin
ext:000A41F4                STR     R0, [R4]
ext:000A41F6                BL      _ZN7cocos2d13CCUserDefault17sharedUserDefaultEv ; cocos2d::CCUserDefault::sharedUserDefault(void)
ext:000A41FA                LDR     R2, [R4]
ext:000A41FC                MOVS    R1, R5
ext:000A41FE                BL      _ZN7cocos2d13CCUserDefault16setIntegerForKeyEPKci ; cocos2d::CCUserDefault::setIntegerForKey(char cons
ext:000A4202                BL      _ZN7cocos2d13CCUserDefault17sharedUserDefaultEv ; cocos2d::CCUserDefault::sharedUserDefault(void)
ext:000A4206                BL      _ZN7cocos2d13CCUserDefault5flushEv ; cocos2d::CCUserDefault::flush(void)
ext:000A420A                CMP     R6, #0
ext:000A420C                BLE     loc_A4218
```

然后，我们在 16 进制下修改，选用 010Editor 或 UE，ctrl + g，进行地址跳转，寻找到我们在 IDA 修改的地址，按照上文修改，保存，替换，回编译。

这里需要说明的是，这个方法何时被调用，即才能激活我们对金币的赋值呢？



直接在 IDA 里面对这个方法按下 x 键，查看调用，有 5 处。那么应该很容易被激活。

红色标注的，是使用金币的道具，按照我们看到方法被调用的地方，这里被调用了，然后金币数量即我们定义的。

0x6:修改短信

这里不多说了，根据系统发送短信函数进行查找修改，替换短信内容，替换短信发送人即可。

我直接使用 hook 的方式替换的，比较省事了。。

相关代码不难，看完我前面几篇 hook java 的文章的同学，应该很容易能够写出来代码，这里不再多写，有兴趣，自己动手去。

跟着鬼哥学 so 修改，六，实例第三篇

今天帮群里一个朋友看一个成语的游戏，游戏还行，学习传统文化知识嘛，顺手帮忙破解看了下，中午吃饭的时间写了下文章，在这里记录一下分析思路，还有很多种方式破解的，大家可以拓展思路继续写文章的。

0x1:寻找关键点

拿到游戏之后，我们需要先到游戏进行一下玩耍，寻找一下需要破解什么，推测在什么地方可以破解。

捕获到关键字：
①.金币不足 200!
②.当前金币为 0
③.有时间限制为 200s

这样，我们可以有以下思路:
①.寻找到提示金币不足 200 的地方，修改判断。

②.找到初始化定义金币的地方修改金币数值。
③.200s 是固定数值，所以找到定义地方，修改掉它。

0x2:进行分析

由步骤一，我们可以先从金币来进行寻找。
从 Unicode 编码的金币，和中文金币两种形式进行搜索。(因为如果是在 smali 代码中调用中文，是会转义成 Unicode 编码。如果是调用 xml 中定义的，一般是中文直接定义的)。
这样，我们在 assets/strings.xml 中可以找到：

```
<?xml version="1.0" encoding="UTF-8"?><dict>
<key>tip1</key>
    <string>选择的答案不正确
请重新选择</string>
<key>goal</key>
    <string>恭喜你，全部通关啦！ </string>
    <key>tip2</key>
    <string>提示</string>
    <key>tip3</key>
    <string>金币不够,试试
分享按钮求助朋友吧</string>
    <key>tip4</key>
    <string>今天已经领取过了！ </string>
    <key>tip5</key>
    <string>消耗金币200</string>
    <key>tip6</key>
    <string>金币不足200!</string>
<key>tip10</key><string>评论后请稍等片刻！ </strin
```


这里是以键值对的方式存在的，所以我们要找哪里调用了”金币不足 200!”,就要再去寻找这个 key 的调用地方，即 tip6。
经过搜索，发现只被 libcocos2dcpp.so 所调用，所以这个时候我们打开 IDA 来载入这个 so 文件。
通过 String 窗口搜索字符串 tip6，然后寻找到调用它的地方，这里我们 F5 一下，方便大家理解：

```
if ( v44 <= 199 )
{
  sub_382138(&v53, "tip6", &v49);
  v10 = cocos2d::CCDictionary::objectForKey(v4, &v53);
  CTipLayer::SetTipText((MainScene *)((char *)v3 + 3396), *(const char **)(v10 + 20));
  result = sub_380A3C(&v53);
  goto def_14966C;
}
```


通过代码，我们可以看到是通过判断 v44 是否大于 199，来判断是否输出”金币不足 200!”这个字符串的。

所以关键点我们需要找到 v44 的定义位置：

```
v5 = (cocos2d::CCUserDefault *)(*(int (__fastcall **)(cocos2d::CCObject *))(*(_DWORD *
v6 = v5;
v7 = (cocos2d::CCUserDefault *)cocos2d::CCUserDefault::sharedUserDefault(v5);
result = cocos2d::CCUserDefault::getIntegerForKey(v7, "total_scores");
v44 = result;
if ( (unsigned int)v6 <= 5 )
{
  result = (int)v6;
  switch ( v6 )
  {
    case 0u:
      v9 = (CocosDenshion::SimpleAudioEngine *)CocosDenshion::SimpleAudioEngine::share
      result = CocosDenshion::SimpleAudioEngine::playEffect(v9, "sounds/pressBtn.mp3")
      if ( *((_BYTE *)v3 + 3357) )
        goto def_14966C;
      CTipLayer::SetFadeOutTime((MainScene *)((char *)v3 + 3396), 4.0);
      cocos2d::CCPoint::CCPoint((cocos2d::CCPoint *)&v57, 240.0, 100.0);
      CTipLayer::SetTipPostion((char *)v3 + 3396, &v57);
      if ( v44 <= 199 )
      {
        sub_382138(&v53, "tip6", &v49);
```


红色指向的即是，所以到了这里，我们关键点修改这里的 result 数据，即 cocos2d::CCUserDefault::getIntegerForKey(v7, "total_scores");
因为这里是传入了数据 total_scores，所以我们得想办法，不能直接修改这个算法的返回值。
所以我们打开 hex 窗口：

```
A C1 49                         LDR     R1, =(aTotal_scores - 0x149660)
C 79 44                         ADD     R1, PC          ; "total_scores"
E 8B F0 E8 FF                   BL      _ZN7cocos2d13CCUserDefault16getIntegerForKeyEPKc ; 获取得到当前金币数值
2 02 90                         STR     R0, [SP,#0x680+var_678] ; 将获取得到的金币数值R0保存于金币存放的地址
4 05 2E                         CMP     R6, #5          ; switch 6 cases
6 00 D9                         BLS     loc_14966A
8 9F E2                         B       def_14966C      ; jumptable 0014966C default case
A             ; ---------------------------------------------------------------------------
```


这里我添加了注释，大家可以看到。
因为是有传入参数的，这个算法最后获得的数值现在看来只在这里使用，所以我们可以修改 Bl 的代码，即在这里为 R0 赋值即可，因为在下一行代码中会保存 R0 的数值到金币存放的具体位置。
因为是 4 个字节，所以我们可以参考前面系列文章的实例教程，

“以 TTX 连萌来多层次分析游戏破解”中的破解方式来修改 so 代码。

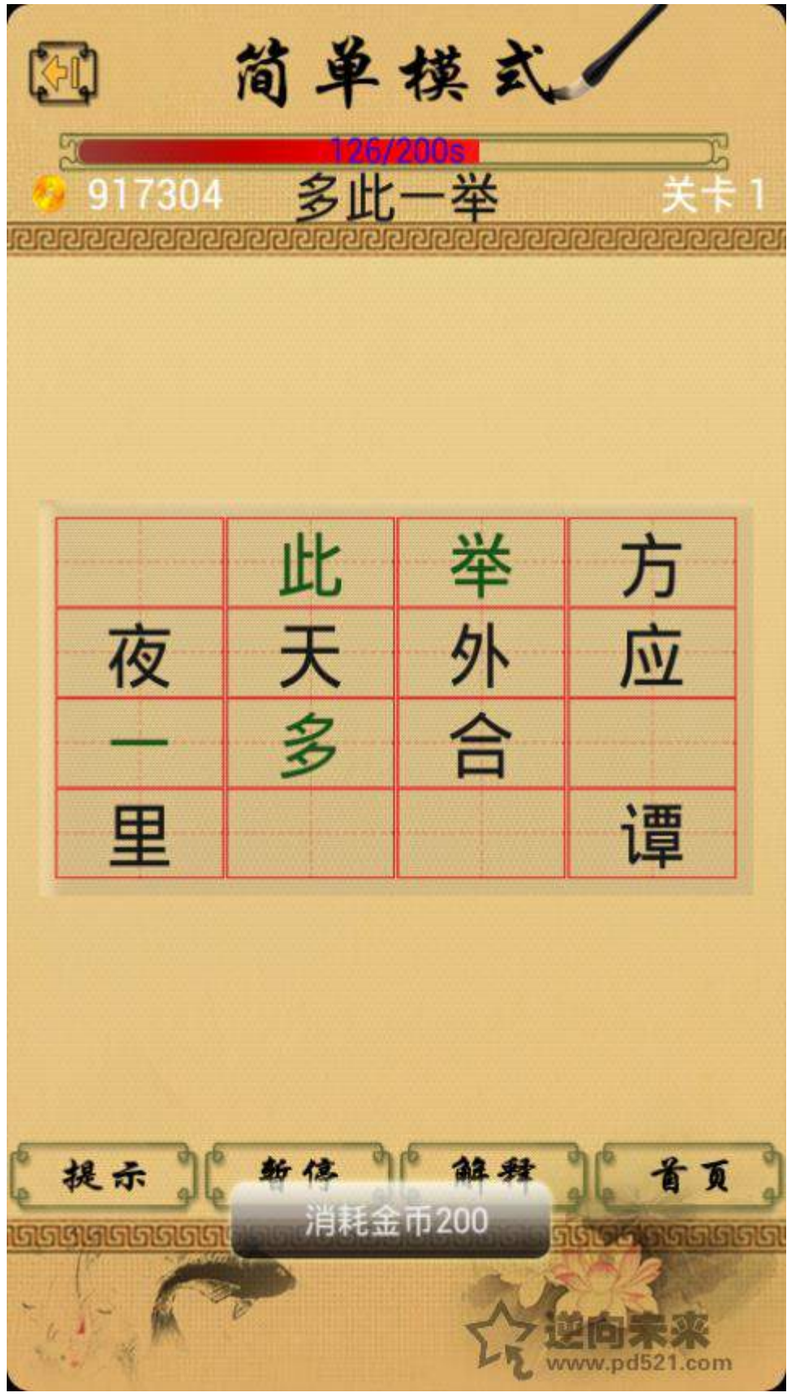
http://blog.csdn.net/guiguzi1110/article/details/40586339

摘录关键点：

即，BL getIntegerForKey，我们直接复制R0一个数值，然后在下面Adds R0,R0,R6这里，直接对R0逻辑左移一下，即LSL一下，将其数值变大。
因为第一个BL是4个字节，我们直接修改 00 00 09 20 ，即Movs R0,R0 代表没有任何操作，09 20是将数字9赋值给R0
然后ADDs那一行，有两个字节，直接改为 00 04，即 LSLS R0，R0，#0x10
这样，我们即完成了对金币数量的赋值，如下图：

我们这里换 010Editor 来加载 so 文件，跳转到对应 BL 代码的位置。
修改为 0E 20 00 04
修改成这个指令的原因，大家通过以前的文章自己看一看去理解，这里不多解释了。

0x3:进行测试


简单模式
126/200s
917304
多此一举
关卡 1
此 举 方
夜 天 外 应
一 多 合
里 谭
提示
新信
解释
首页
消耗金币200
逆向未来
www.pd521.com

我们只替换 so 文件，重新签名安装即可，发现一切正常。

所以到此，我们对这个游戏的分析已经可以了，剩下的几种破解方式，大家自己去研究，游戏也有广告，大家去一下，欢迎大家继续发帖交流。

游戏原包丢百度盘:

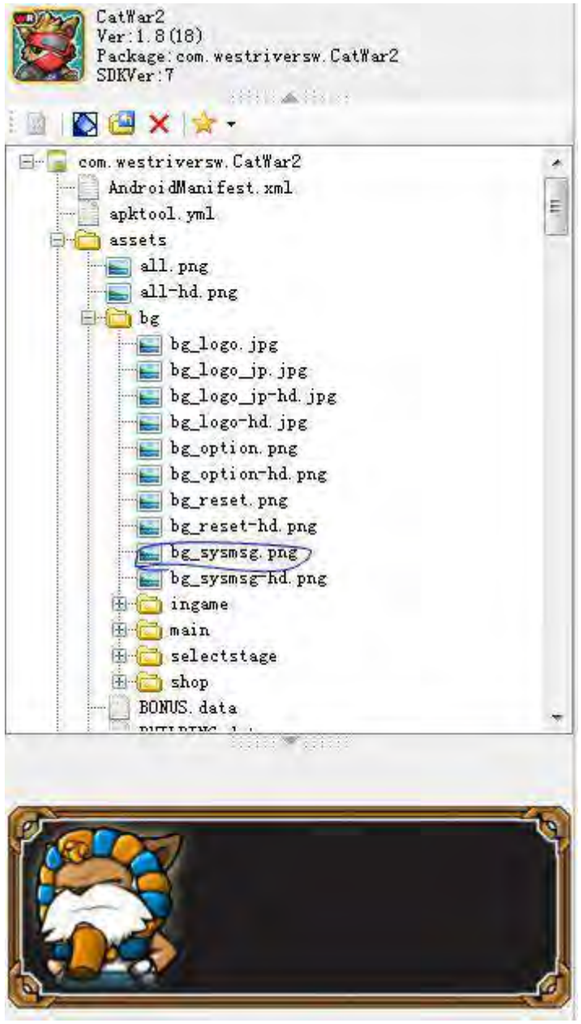
http://pan.baidu.com/s/1ntCc6Kh

转载请注明出处:www.pd521.com

[Android] 记一次安卓游戏破解（分析代码全过程）

IDA,">关于安卓破解思路分析贴
转帖地址 http://bbs.pediy.com/showthread.php?t=170132
起因：CatWar2，钻石不够，而且用八门神器修改之后，钻石数会自动清 0

线索：



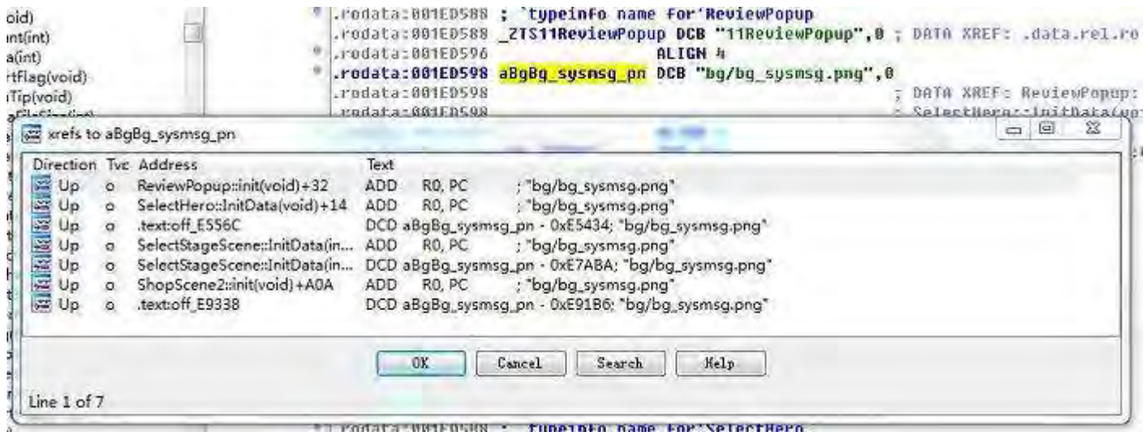
顺藤摸瓜，搜索 bg_sysmsg

发现其被引用于 libhelloworld.so 文件中

引用:

.so 文件是动态链接库文件,相当于 win 下的 .dll 文件

于是操起 IDA，ALT+T 搜索 bg_sysmsg

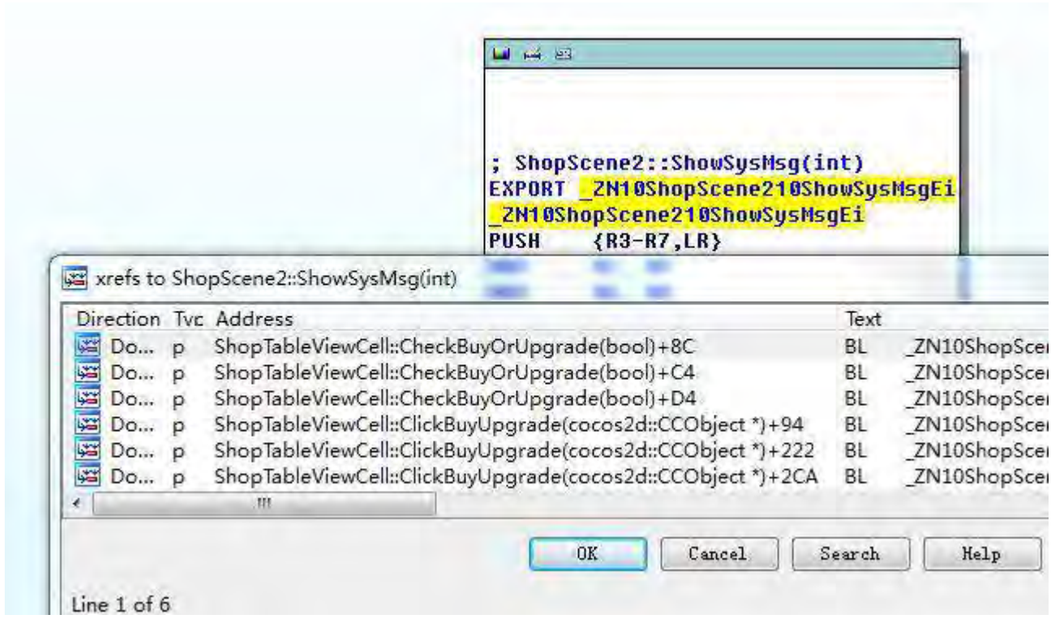


经过一番观察，根据函数名，来到了
引用:
ShopScene2::init(void)

出现在初始化里，然后在函数列表里找到 ShopScene2::ShowSysMsg(int)。
引用:
.text:000E864A MOVS R0, 0x3F800000
.text:000E864E BL _ZN7cocos2d11CCDelayTime18actionWithDurationEf ; cocos2d::CCDelayTime::actionWithDuration(float)

用了延迟，第一张图的提示，也是显示一段时间就消失了了。猜测就是调用这个函数，显示信息。

再次查找引用



进入第一个函数，这个函数引用了一个很可疑的函数：WRIntEncrypt::Get(void)

引用:

```
BL      _ZN12WRIntEncrypt3GetEv ; WRIntEncrypt::Get(void)
```

看一下这个类的所有函数，目测就是要找的了。

来到构造函数：

（PS:R0 是 this 指针）

引用:

```
.text:000F1260                  PUSH     {R4-R6,LR}
.text:000F1262                  MOVS     R5, #0
.text:000F1264                  STR      R5, [R0]
.text:000F1266                  MOVS     R4, R0
```

```
.text:000F1268                BLX     lrand48
.text:000F126C                STR     R5, [R4,#8]
.text:000F126E                STR     R0, [R4,#4]
.text:000F1270                BLX     lrand48
.text:000F1274                STR     R0, [R4,#0xC]
.text:000F1276                STR     R5, [R4,#0x10]
.text:000F1278                MOVS      R0, R4
.text:000F127A                POP     {R4-R6,PC}
```

确定这个类成员有 5 个，初始化之后
+0　0
+4　随机数 1
+8　0
+c　随机数 2
+10 0

再看 WRIntEncrypt::Set(int)
（PS：R1 是参数，就是要设置的数值）
引用:

```
.text:000F1118                LDR     R3, [R0,#4]
.text:000F111A                STR     R1, [R0,#0x10]
.text:000F111C                EORS      R3, R1
.text:000F111E                STR     R3, [R0]
.text:000F1120                LDR     R3, [R0,#0xC]
.text:000F1122                EORS      R1, R3
.text:000F1124                STR     R1, [R0,#8]
.text:000F1126                BX      LR
```

从这里就看到
+0　未加密值 异或 随机数 1
+4　随机数 1
+8　未加密值 异或 随机数 2
+c　随机数 2
+10 未加密值

```
struct EncryptInt
{
  int eint1;//加密数值
  int key1;//随机密钥
  int eint2;//加密数值
  int key2;//随机密钥
  int realint;//原始数值
```

};

也就是说，游戏保存了三份数据，二份是加密过，一次是未加密的。

再看看 WRIntEncrypt::Get(void)
引用:

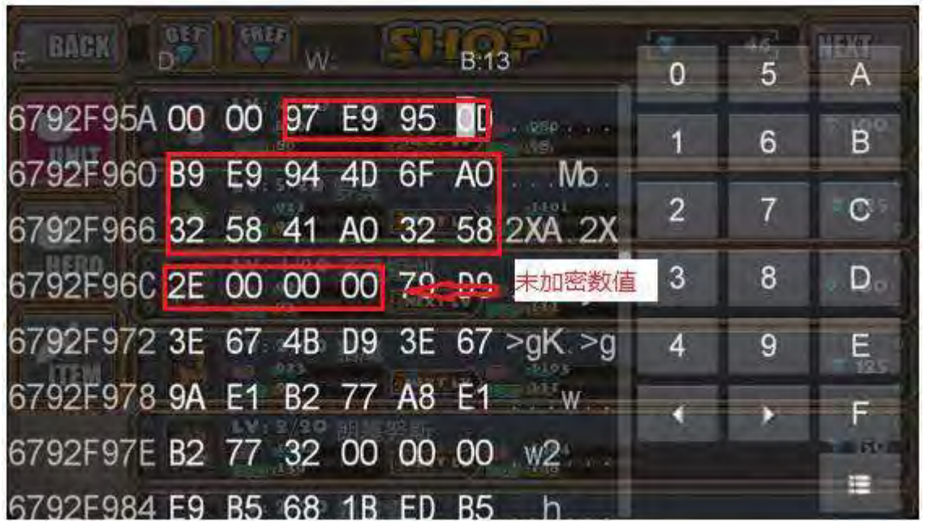
```
.text:000F120C                 PUSH     {R4,LR}
.text:000F120E                 LDR      R3, [R0,#EncryptInt]
.text:000F1210                 LDR      R2, [R0,#EncryptInt.key1]
.text:000F1212                 LDR      R1, [R0,#EncryptInt.key2]
.text:000F1214                 MOVS      R4, R0
.text:000F1216                 EORS      R2, R3
.text:000F1218                 LDR      R3, [R0,#EncryptInt.eint2]
.text:000F121A                 EORS      R3, R1
.text:000F121C                 LDR      R1, [R0,#EncryptInt.realint]
.text:000F121E                 MOVS      R0, #1
.text:000F1220                 CMP      R3, R1
.text:000F1222                 BEQ      loc_F123C
.text:000F1224                 CMP      R2, R3
.text:000F1226                 BEQ      loc_F1246
.text:000F1228
.text:000F1228 loc_F1228                               ; CODE XREF: WRIntEncrypt::Get(void)+38j
.text:000F1228                                         ; WRIntEncrypt::Get(void)+3Ej
.text:000F1228                 MOVS      R0, R4
.text:000F122A                 BL       _ZN12WRIntEncrypt8CheatingEv ; WRIntEncrypt::Cheating(void)
```

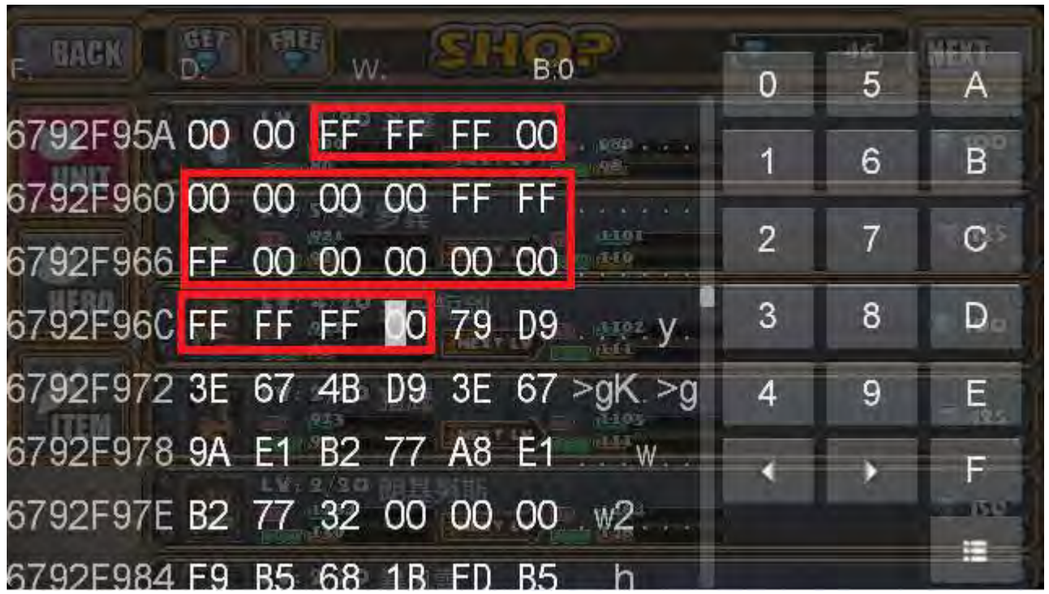
与加密的数值做了比较，不相等就进入作弊处理 WRIntEncrypt::Cheating(void)

知道了加密怎么处理，就很容易了。回到手机，八门神器定位到 realint

```
struct EncryptInt
{
  int eint1;
  int key1;
  int eint2;
  int key2;
  int realint;
};
```

然后 eint1，eint2，realint 都改为 0xFFFFFF,key1,key2 清零。任何数值异或 0 都不变。


BACK
GET
FREE
SHOP
B:13
6792F95A 00 00 97 E9 95 0D
6792F960 B9 E9 94 4D 6F A0
6792F966 32 58 41 A0 32 58
6792F96C 2E 00 00 00 79 D9
未加密数值
6792F972 3E 67 4B D9 3E 67
6792F978 9A E1 B2 77 A8 E1
6792F97E B2 77 32 00 00 00
6792F984 E9 B5 68 1B FD B5
0 5 A
1 6 B
2 7 C
3 8 D
4 9 E
F


BACK
GET
FREE
SHOP
B:0
6792F95A 00 00 FF FF FF 00
6792F960 00 00 00 00 FF FF
6792F966 FF 00 00 00 00 00
6792F96C FF FF FF 00 79 D9
6792F972 3E 67 4B D9 3E 67
6792F978 9A E1 B2 77 A8 E1
6792F97E B2 77 32 00 00 00
6792F984 E9 B5 68 1B FD B5
0 5 A
1 6 B
2 7 C
3 8 D
4 9 E
F

不会 APKDIY，只能先这样了。最后发现，钻石退出以后，重新进入游戏会清 0.可能还有别的地方有验证。不过修改完钻石，把英雄升级之后，基本就无敌了。也算是够用了。

对某一算法的详细分析（学习 ARM 下汇编指令）

前言

这次校赛出了一个安卓上的 crackme，算法比较简单，稍微难点的是将算法代码全都写在了一个 SO 中。增加了一点难度。

下面是利用 IDA Pro 6.4 对算法部分的静态分析：分析目标：xxxxx.so

0×1 定位到 jnicall1：

```
.text:000014FC                                  EXPORT Java_com_reverse_ilikewanniba_MainActivity_jnicall1
.text:000014FC                  Java_com_reverse_ilikewanniba_MainActivity_jnicall1
.text:000014FC
.text:000014FC                  var_14          = -0x14
.text:000014FC
.text:000014FC 30 B5                            PUSH    {R4,R5,LR}
.text:000014FE 01 68                            LDR     R1, [R0]        ; R1 = R0   R0 是 *env
.text:00001500 A9 23 9B 00                      MOVS    R3, #0x2A4      ; 调用函数的偏移 0x2A4
.text:00001504 CB 58                            LDR     R3, [R1,R3]     ; R3 是 env 偏移 0x2A4 处的一个函数
.text:00001506 83 B0                            SUB     SP, SP, #0xC    ; 分配栈空间
```

```
.text:00001508 11 1C                                    MOVS     R1, R2              ; str 给 R1 寄存器
.text:0000150A 00 22                                    MOVS     R2, #0              ; R2 寄存器赋 0
.text:0000150C 05 1C                                    MOVS     R5, R0              ; 因为 R0 要存函数返回值
.text:0000150C                                                                       ; 这里先将 R0 原来的值存入 R5 寄存器
.text:0000150E 98 47                                    BLX      R3                  ; 执行 GetStringUTFChars 函数
.text:00001510 00 28                                    CMP      R0, #0              ; 判断返回值（指针）是否为空
.text:00001512 2E D0                                    BEQ      loc_1572
.text:00001514 03 78                                    LDRB     R3, [R0]            ; R3 取 第一个
.text:00001516 01 AC                                    ADD      R4, SP, #0x18+var_14 ; 新定义一个数据空间
.text:00001518 23 70                                    STRB     R3, [R4]            ; paramStr[0] = inputstr[0]
.text:0000151A 42 78                                    LDRB     R2, [R0,#1]
.text:0000151C 43 1C                                    ADDS     R3, R0, #1
.text:0000151E 20 1C                                    MOVS     R0, R4
.text:00001520 62 70                                    STRB     R2, [R4,#1]         ; paramStr[1] = inputstr[1]
.text:00001522 5A 78                                    LDRB     R2, [R3,#1]
.text:00001524 A2 70                                    STRB     R2, [R4,#2]         ; paramStr[2] = inputstr[2]
.text:00001526 9B 78                                    LDRB     R3, [R3,#2]
.text:00001528 E3 70                                    STRB     R3, [R4,#3]         ; paramStr[3] = inputstr[3]
.text:0000152A 00 23                                    MOVS     R3, #0
.text:0000152C 23 71                                    STRB     R3, [R4,#4]         ; paramStr[4] = '\0'
.text:0000152E FF F7 EF FE                              BL       StringConvert       ; 调用 StringConvert 函数,参数是 paramStr[]
.text:00001532 10 4B                                    LDR      R3, =(__data_start_ptr - loc_153A)
.text:00001534 21 78                                    LDRB     R1, [R4]
.text:00001536 7B 44                                    ADD      R3, PC ; __data_start_ptr
.text:00001538 1B 68                                    LDR      R3, [R3]            ; "SycL0v3R"
.text:0000153A
.text:0000153A                          loc_153A                                     ; DATA XREF: .text:off_1574o
.text:0000153A 1A 78                                    LDRB     R2, [R3]
.text:0000153C 91 42                                    CMP      R1, R2
.text:0000153E 03 D1                                    BNE      loc_1548
.text:00001540 62 78                                    LDRB     R2, [R4,#1]
.text:00001542 5B 78                                    LDRB     R3, [R3,#1]
.text:00001544 9A 42                                    CMP      R2, R3
.text:00001546 0A D0                                    BEQ      loc_155E
.text:00001548
.text:00001548                          loc_1548                                     ; CODE XREF: Java_com_reverse_ilikewanniba_MainActivity_jnicall1+42j
.text:00001548 0B 4A                                    LDR      R2, =(ohno_ptr - loc_1554)
.text:0000154A 28 68                                    LDR      R0, [R5]
.text:0000154C A7 23 9B 00                              MOVS     R3, #0x29C
.text:00001550 7A 44                                    ADD      R2, PC ; ohno_ptr
.text:00001552 C3 58                                    LDR      R3, [R0,R3]
.text:00001554
```

```
.text:00001554                  loc_1554                                ; DATA XREF: .text:off_1578o
.text:00001554 11 68                            LDR     R1, [R2] ; ohno
.text:00001556 28 1C                            MOVS    R0, R5
.text:00001558 98 47                            BLX     R3
.text:0000155A
.text:0000155A                  loc_155A                                ; CODE XREF: Java_com_reverse_ilikewanniba_MainActivity_jnicall1+74j
.text:0000155A                                                          ; Java_com_reverse_ilikewanniba_MainActivity_jnicall1:loc_1572j
.text:0000155A 03 B0                            ADD     SP, SP, #0xC
.text:0000155C 30 BD                            POP     {R4,R5,PC}
.text:0000155E                  ; ---------------------------------------------------------------------------
.text:0000155E
.text:0000155E                  loc_155E                                ; CODE XREF: Java_com_reverse_ilikewanniba_MainActivity_jnicall1+4Aj
.text:0000155E 07 4A                            LDR     R2, =(yeah_ptr - loc_1568)
.text:00001560 28 68                            LDR     R0, [R5]
.text:00001562 A7 23                            MOVS    R3, #0xA7
.text:00001564 7A 44                            ADD     R2, PC ; yeah_ptr
.text:00001566 9B 00                            LSLS    R3, R3, #2
.text:00001568
.text:00001568                  loc_1568                                ; DATA XREF: .text:off_157Co
.text:00001568 C3 58                            LDR     R3, [R0,R3]
.text:0000156A 11 68                            LDR     R1, [R2] ; yeah
.text:0000156C 28 1C                            MOVS    R0, R5
.text:0000156E 98 47                            BLX     R3
.text:00001570 F3 E7                            B       loc_155A
.text:00001572                  ; ---------------------------------------------------------------------------
.text:00001572
.text:00001572                  loc_1572                                ; CODE XREF: Java_com_reverse_ilikewanniba_MainActivity_jnicall1+16j
.text:00001572 F2 E7                            B       loc_155A
.text:00001572                  ; End of function Java_com_reverse_ilikewanniba_MainActivity_jnicall1
.text:00001572
```

jnicall1 主要功能是调用 GetStringUTFChars 函数，获取到输入的注册码，然后将前四位字符，作为 StringConvert 函数的参数，来调用，最后返回值和固定字符串”syclover”进行相应的比较。达到验证的目的。

算法部分就是在 StringConvert 这个函数了。

0×2 StringConvert 函数：

```
.text:00001310                          EXPORT StringConvert
.text:00001310                  StringConvert                           ; CODE XREF: Java_com_reverse_ilikewanniba_MainActivity_jnicall4+30p
.text:00001310                                                          ; Java_com_reverse_ilikewanniba_MainActivity_jnicall3+30p ...
.text:00001310
```

```
.text:00001310                          var_24          = -0x24
.text:00001310
.text:00001310 F0 B5                            PUSH    {R4-R7,LR}
.text:00001312 4F 46                            MOV     R7, R9
.text:00001314 46 46                            MOV     R6, R8
.text:00001316 C0 B4                            PUSH    {R6,R7}         ; 继续压栈
.text:00001318 83 B0                            SUB     SP, SP, #0xC    ; 开辟空间
.text:0000131A 07 1C                            MOVS    R7, R0          ; 参数存入 R7
.text:0000131C FF F7 94 EF                      BLX     strlen          ; 去调用 strlen 得到参数的长度 len
.text:00001320 01 23                            MOVS    R3, #1          ; R3 = 1
.text:00001322 03 40                            ANDS    R3, R0          ; 利用 ands 操作判断字符串 paramStr 长度的奇偶性
.text:00001324 20 D1                            BNE     loc_1368        ; 奇数就跳走，偶数就继续执行
.text:00001326 01 AC                            ADD     R4, SP, #0x28+var_24 ; 开辟一个新空间 newStr，R4 指向 newStr[]
.text:00001328 23 80                            STRH    R3, [R4]
.text:0000132A 40 08                            LSRS    R0, R0, #1      ; 长度 len 右移一位，len/2
.text:0000132C 00 23                            MOVS    R3, #0          ; R3 = 0
.text:0000132E A3 70                            STRB    R3, [R4,#2]
.text:00001330 80 46                            MOV     R8, R0          ; len/2 存入 R8
.text:00001332 00 28                            CMP     R0, #0          ; len 和 0 比较 判断参数是否为空
.text:00001334 10 D0                            BEQ     loc_1358        ; 为空就跳走
.text:00001336 00 25                            MOVS    R5, #0          ; R5 = 0
.text:00001338 3E 1C                            MOVS    R6, R7          ; R6 指向 paramStr
.text:0000133A A9 46                            MOV     R9, R5          ; R9 = R5 =0
.text:0000133C
.text:0000133C                  loc_133C                                ; CODE XREF: StringConvert+46j
.text:0000133C 33 78                            LDRB    R3, [R6]        ; R3 = paramStr[0]
.text:0000133E 4A 46                            MOV     R2, R9          ; R2 = R9 = 0
.text:00001340 20 1C                            MOVS    R0, R4          ; R0 指向 一个新空间
.text:00001342 23 70                            STRB    R3, [R4]        ; newStr[0] = R3 = paramStr[0]
.text:00001344 73 78                            LDRB    R3, [R6,#1]     ; R3 = paramStr[1]
.text:00001346 A2 70                            STRB    R2, [R4,#2]     ; newStr[2] = '\0'
.text:00001348 02 36                            ADDS    R6, #2          ; R6 现在指向 paramStr[2]
.text:0000134A 63 70                            STRB    R3, [R4,#1]     ; newStr[1] = R3 = paramStr[1]
.text:0000134C FF F7 8C FF                      BL      FromStrToChar   ; 以 newStr[2] 作为参数,调用 FromStrToChar
.text:00001350 78 55                            STRB    R0, [R7,R5]     ; R0 是返回值,paramStr[0] = R0
.text:00001350                                                          ; 加入取的两位是 63,那么 R0=0x63
.text:00001352 01 35                            ADDS    R5, #1          ; R5 循环计数器+1
.text:00001354 45 45                            CMP     R5, R8          ; 判断循环是否结束
.text:00001356 F1 D1                            BNE     loc_133C        ; R3 = paramStr[0]
.text:00001358
.text:00001358                  loc_1358                                ; CODE XREF: StringConvert+24j
.text:00001358 00 23                            MOVS    R3, #0          ; R3 = 0
```

```
.text:0000135A 42 46                    MOV     R2, R8              ; R2 = R8 = LEN/2
.text:0000135C BB 54                    STRB    R3, [R7,R2]         ; 在末尾添加'\0'
.text:0000135E
.text:0000135E                  loc_135E                            ; CODE XREF: StringConvert+5Cj
.text:0000135E 03 B0                    ADD     SP, SP, #0xC
.text:00001360 0C BC                    POP     {R2,R3}
.text:00001362 90 46                    MOV     R8, R2
.text:00001364 99 46                    MOV     R9, R3              ; 一系列的出栈操作
.text:00001366 F0 BD                    POP     {R4-R7,PC}
.text:00001368                  ; ---------------------------------------------------------------------------
.text:00001368
.text:00001368                  loc_1368                            ; CODE XREF: StringConvert+14j
.text:00001368 00 23                    MOVS    R3, #0
.text:0000136A 3B 70                    STRB    R3, [R7]
.text:0000136C F7 E7                    B       loc_135E
.text:0000136C                  ; End of function StringConvert
```

StringConvert 函数首先判断输入参数长度是不是为奇数，不是就末尾补 0。每次循环依次取走参数的两位，作为 FromStrToChar 这个函数的参数。将计算后的值保存。

0×3 FromStrToChar 函数：

```
.text:00001268 F0 B5                    PUSH    {R4-R7,LR}
.text:0000126A 06 1C                    MOVS    R6, R0              ; 参数给 R6
.text:0000126C 97 B0                    SUB     SP, SP, #0x5C       ; 开辟空间
.text:0000126E 00 20                    MOVS    R0, #0              ; R0 清零
.text:00001270 00 2E                    CMP     R6, #0              ; 判断传入的参数是否为空
.text:00001272 3F D0                    BEQ     loc_12F4            ; 为空就跳走
.text:00001274 24 49                    LDR     R1, =(unk_32EC - loc_127E) ; R1  指向一个初始化 int 数组  int arr[16] =
.text:00001274                                                      ; 0000 1000 2000 .....
.text:00001276 06 AF                    ADD     R7, SP, #0x70+dest ; 定义了一个数组，R7 数组分配空间
.text:00001278 38 1C                    MOVS    R0, R7              ; dest
.text:0000127A 79 44                    ADD     R1, PC ; unk_32EC ; src
.text:0000127C 40 22                    MOVS    R2, #0x40           ; n
.text:0000127E
.text:0000127E                  loc_127E                            ; DATA XREF: .text:off_1308o
.text:0000127E FF F7 C0 EF              BLX     memcpy
.text:00001282 00 25                    MOVS    R5, #0              ; R5 = 0
.text:00001284 00 24                    MOVS    R4, #0              ; R4 = 0
.text:00001286
.text:00001286                  loc_1286                            ; CODE XREF: FromStrToChar+86j
.text:00001286 33 5D                    LDRB    R3, [R6,R4]         ; R3 = newStr[0]  取参数第一位
.text:00001288 1A 1C                    MOVS    R2, R3              ; R2 = R3 = newStr[0]
.text:0000128A 30 3A                    SUBS    R2, #0x30           ; R2 = R2 - 0X30 = newStr[0] - 0x30
```

```
.text:0000128C 11 06                                LSLS    R1, R2, #0x18     ; 左移 0x18
.text:0000128E 09 0E                                LSRS    R1, R1, #0x18     ; 右移 0x18    R1 是减去 0x30 的最后结果
.text:00001290 09 29                                CMP     R1, #9              ; R2 - 0X30  和  0x9  比较
.text:00001292 31 D8                                BHI     loc_12F8            ;  小于 9 就不跳走，大于 9 就跳转
.text:00001294 92 00                                LSLS    R2, R2, #2          ; newStr[0] - 0x30  左移 2 位
.text:00001294                                                                    ; (newStr[0] - 0x30) * 4
.text:00001296 B8 58                                LDR     R0, [R7,R2]         ;  以上面得到的值作为下标，去初始化数组中取值
.text:00001296                                                                    ; result = arr[(newStr[0] - 0x30) * 4]
.text:00001298
.text:00001298                      loc_1298                                      ; CODE XREF: FromStrToChar+96j
.text:00001298 00 F0 E0 EA                          BLX     __floatsidf
.text:0000129C 00 90                                STR     R0, [SP,#0x70+var_70]
.text:0000129E 01 91                                STR     R1, [SP,#0x70+var_6C]
.text:000012A0 20 1C                                MOVS    R0, R4
.text:000012A2 00 F0 CA EA                          BLX     __floatunsidf
.text:000012A6 02 1C                                MOVS    R2, R0              ; y
.text:000012A8 0B 1C                                MOVS    R3, R1
.text:000012AA 16 49                                LDR     R1, =0x40300000
.text:000012AC 14 48                                LDR     R0, =0              ; x
.text:000012AE FF F7 BA EF                          BLX     pow
.text:000012B2 02 90                                STR     R0, [SP,#0x70+var_68]
.text:000012B4 03 91                                STR     R1, [SP,#0x70+var_64]
.text:000012B6 28 1C                                MOVS    R0, R5
.text:000012B8 00 F0 BE EA                          BLX     __floatunsidf
.text:000012BC 02 9A                                LDR     R2, [SP,#0x70+var_68]
.text:000012BE 03 9B                                LDR     R3, [SP,#0x70+var_64]
.text:000012C0 04 90                                STR     R0, [SP,#0x70+var_60]
.text:000012C2 05 91                                STR     R1, [SP,#0x70+var_5C]
.text:000012C4 0F 49                                LDR     R1, =0x40300000
.text:000012C6 0E 48                                LDR     R0, =0
.text:000012C8 00 F0 6C EC                          BLX     __divdf3
.text:000012CC 02 1C                                MOVS    R2, R0
.text:000012CE 0B 1C                                MOVS    R3, R1
.text:000012D0 00 98                                LDR     R0, [SP,#0x70+var_70]
.text:000012D2 01 99                                LDR     R1, [SP,#0x70+var_6C]
.text:000012D4 00 F0 30 EB                          BLX     __muldf3
.text:000012D8 02 1C                                MOVS    R2, R0
.text:000012DA 0B 1C                                MOVS    R3, R1
.text:000012DC 04 98                                LDR     R0, [SP,#0x70+var_60]
.text:000012DE 05 99                                LDR     R1, [SP,#0x70+var_5C]
.text:000012E0 00 F0 54 E9                          BLX     __aeabi_dadd
.text:000012E4 00 F0 60 ED                          BLX     __fixunsdfsi      ; 上面一长串代码的功能我是通过几次动态跟踪才总结出来的
```

```
.text:000012E4                                                  ; 循环变量为 i,R0 =  上面减去 0x30/0x37 的值  * 16/pow(16,i)
.text:000012E8 01 34                    ADDS    R4, #1          ; R4 作为循环变量  + 1
.text:000012EA 05 1C                    MOVS    R5, R0          ; 转换之后的结果存入 R5
.text:000012EC 02 2C                    CMP     R4, #2          ; 判断循环结束么有
.text:000012EE CA D1                    BNE     loc_1286        ; R3 = newStr[0]  取参数第一位
.text:000012F0 05 06                    LSLS    R5, R0, #0x18
.text:000012F2 28 0E                    LSRS    R0, R5, #0x18
.text:000012F4
.text:000012F4                  loc_12F4                        ; CODE XREF: FromStrToChar+Aj
.text:000012F4 17 B0                    ADD     SP, SP, #0x5C
.text:000012F6 F0 BD                    POP     {R4-R7,PC}
.text:000012F8                  ; ---------------------------------------------------------------------------
.text:000012F8
.text:000012F8                  loc_12F8                        ; CODE XREF: FromStrToChar+2Aj
.text:000012F8 37 3B                    SUBS    R3, #0x37       ; 大于 9,R0 = R3 - 0x37 = newStr[0] - 0x37
.text:000012FA 9B 00                    LSLS    R3, R3, #2      ; 左移两位,乘以 4
.text:000012FC F8 58                    LDR     R0, [R7,R3]     ; 以上面得到的值作为下标，去初始化数组中取值
.text:000012FC                                                  ; result = arr[(newStr[0]  - 0x30) * 4]
.text:000012FE CB E7                    B       loc_1298
.text:000012FE                  ; End of function FromStrToChar
```

调用一次 FromStrToChar，参数有两个字符。对每一位都有一次运算。

取字符，然后分别和‘0‘和’9′比较，判断是否在 0×30 和 0×39 区间内

在：字符 - 0×30 * 16*(pow(16,i))

不在：字符 - 0×37 * 16*(pow(16,i))

最终结果：上面两次运算的结果的和。

0×4 小结

获取注册码字符串：31323334
获取前 4 位：3132
调用 StringConvert(“3132”) –> 返回 0×31 0×32
StringConvert 中
{
有一个初始化数组：int arr[] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
第一次循环，取前 2 位：31

调用 FromStrToChar(“31”) –> 返回 0×31

{

第一次循环

’3′ – 0×30 = 3

以 3 作为下标，去初始化数组中查表。

3 * 16 = 48 (R0 = 0×30)

第二次循环

’1′ – 0×30 = 1

以 1 作为下标，去初始化数组中查表。

1 * 1 = 1

R0 = 48 +1 = 49 (R0 = 0×31)

}

第二次循环，取前 2 位：32

调用 FromStrToChar(“32”) –> 返回 0×32

}

固定字符串与之逐位进行比较。

jnicall1 算法基本上就分析到这儿了。

还剩下的 jnicall2、jnicall3、jnicall4、算法都是一样的，区别只是在注册码中取的字符不同而已。

http://0nly3nd.sinaapp.com/?p=384

某 APK 注册机制分析与多思路破解方法

killer5

刚刚接触 android 逆向，请多多包涵

在论坛看到一篇帖子，说软件作者利用异常机制来实现注册，思路比较好。。。

于是简单分析了下，并对 so 文件里的算法进行了分析。

原帖地址：http://www.52pojie.cn/thread-196268-1-1.html

原帖的作者对破解技术进行了部分讲解。

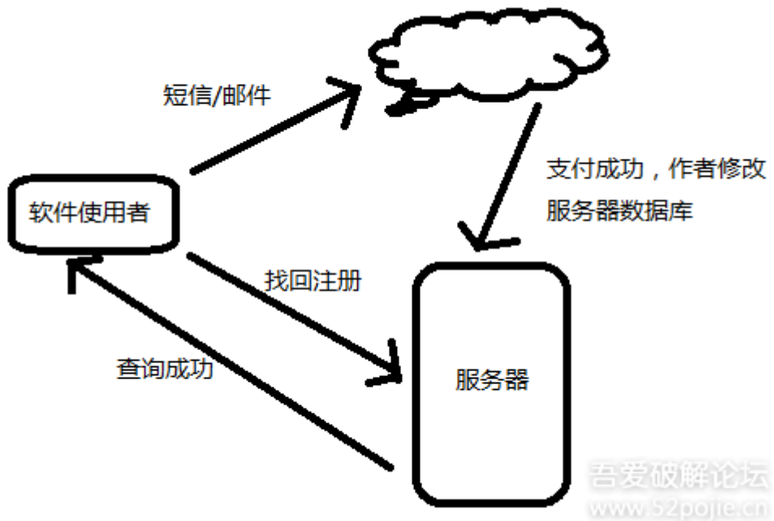
下面我将以软件注册机制和多思路破解两部分进行详细讲解

一软件注册机制分析篇

首先，软件安装后，如图所示

在右上角提示注册，点击后提示找回注册和注册说明，并提示了注册码(在不同的模拟器中显示不一样)

其中该软件的注册流程如下所示：



首先 APK 使用者支付成功后，利用短信或者邮件，将支付信息和机器码发送给作者->作者修改服务器的数据库->APK 使用者利用找回注册，去服务器查询，查询成功后，提示注册成功，否则提示不成功。

因此利用"不成功"字符串为突破口，进行进一步的分析。

JEB 载入 apk，在 BqimenDateInputActivity 中查找到如下的函数

```
static void s(BqimenDateInputActivity arg4) {
        String v0_2;
        try {
            v0_2 = "";
            HttpResponse v1 = new DefaultHttpClient().execute(new HttpGet(arg4.regetkey("QM" + arg4.
                    getIMEI())));
            if(v1.getStatusLine().getStatusCode() == 200) {
                v0_2 = EntityUtils.toString(v1.getEntity(), "UTF-8");
            }
        }
        catch(IOException v0) {
            arg4.aa.MyDialog(((Context)arg4), "不成功，连接超时，请开通网络重试！(错误代码：-403)", "提示");
            v0.printStackTrace();
            return;
        }
        catch(ClientProtocolException v0_1) {
            arg4.aa.MyDialog(((Context)arg4), "不成功，请开通网络重试或联系开发者！(错误代码：-327)", "提示");
            v0_1.printStackTrace();
            return;
        }

        v0_2 = arg4.regetkeyafter(v0_2);
        try {
            if(Integer.parseInt(v0_2) != 6) {
                return;
            }

            arg4.al = true;
            arg4.ag.setVisibility(8);
            arg4.writeFileData("config1my", arg4.writekey("QM" + arg4.getIMEI()));
            arg4.aa.ExecuteSQL(arg4.aa.MyDB, "insert into config values (\'qm\',\'" + arg4.getIMEI()
                    + arg4.getIMSI() + "\',\'1\',1)");
            arg4.aa.ShowMessage(((Context)arg4), "成功", "");
            arg4.m();
        }
        catch(Exception v0_3) {
            arg4.aa.ShowMessage(((Context)arg4), "不成功", "");
        }
    }
```

可以看到，首先利用 HttpGet 连接服务器，查询信息，然后利用动态库 so 里面的 regetkeyafter()方法，对查询的结果进行处理，最后将 regetkeyafter 方法返回的字符串转换成整数，

并判断值是否 6。在字符串转换成整数过程中，如果转换失败，则抛出异常，软件的作者也是在异常中，提示注册不成功的。

上面对 static void s(BqimenDateInputActivity arg4)整体流程分析后，我们再回过头来仔细看看该函数的细节。

该函数中首先调用 getIMEI()方法获得一个字符串（实际上获得的机器码，稍后我会讲解如何验证），并与 QM 拼接获得新的字符串“QMXXXX”，传递给动态库 so 文件里的 regetkey()方法，得到服务器的地址和查询参数（如何知道的？稍后会慢慢道来）。

下面我们可以先输出这几个关键函数的返回值，做个初步的判断。

因此，我们首先获得 getIMEI()方法的返回值、regetkey()方法的返回值、服务器的查询结果 v0_2 和 regetkeyafter()方法的返回值。

打开 apktool 反编译后的文件 BqimenDateInputActivity.smali，找到 S 函数

.method static synthetic s(Lcom/forace/Bqimen/BqimenActivity/BqimenDateInputActivity;)V

修改如下几处代码：

```
.locals 5                                                    改为5

:try_start_0
new-instance v0, Ljava/lang/StringBuilder;

const-string v1, "QM"

invoke-direct {v0, v1}, Ljava/lang/StringBuilder;-><init>(Ljava/lang/String;)V

invoke-virtual {p0}, Lcom/forace/Bqimen/BqimenActivity/BqimenDateInputActivity;->getIMEI()Ljava/lang/String;

move-result-object v1

const-string v4,"SN"
invoke-static{v4,v1},Landroid/util/Log;->v(Ljava/lang/String;Ljava/lang/String;)I
```

```
invoke-virtual {p0, v0}, Lcom/forace/Bqimen/BqimenActivity/BqimenDateInputActivity;->regetkey(Ljava/lang/String;)Ljava/lang/String;

move-result-object v1

invoke-static{v4,v1},Landroid/util/Log;->v(Ljava/lang/String;Ljava/lang/String;)I

const-string v0, ""


invoke-static {v0, v1}, Lorg/apache/http/util/EntityUtils;->toString(Lorg/apache/http/HttpEntity;Ljava/lang/String;)Ljava/lang/String;
:try_end_0
.catch Lorg/apache/http/client/ClientProtocolException; {:try_start_0 .. :try_end_0} :catch_0
.catch Ljava/io/IOException; {:try_start_0 .. :try_end_0} :catch_1

move-result-object v0

invoke-static{v4,v0},Landroid/util/Log;->v(Ljava/lang/String;Ljava/lang/String;)I

:cond_0
```



```
invoke-virtual {p0, v0}, Lcom/forace/Bqimen/BqimenActivity/BqimenDateInputActivity;->regetkeyafter(Ljava/lang/String;)Ljava/lang/Strin

move-result-object v0

invoke-static{v4,v0},Landroid/util/Log;->v(Ljava/lang/String;Ljava/lang/String;)I

:try_start_1
```

然后打包成 apk，签名运行后，点击注册->找回注册,并在 cmd 中输入 adb logcat -s SN:V

得到结果为

```
V/SN      (  677): 000000000000000
V/SN      (  677): http://www.d8soft.com/userc/checkID.asp?key='JIfR7OoQPh5yVMYf
R'
V/SN      (  677): False
V/SN      (  677): a
```


对比之前的注册界面，可以验证我们的想法是正确的。即 getIMEI()方法获得机器码、regetkey()方法的返回服务器的查询地址和参数、变量 v0_2 保存服务器查询的结果，regetkeyafter()对服务器的查询结果进行处理，

此处返回了字符 a，因此调用 Integer.parseInt 会产生异常，跳到如下的代码执行：

```
catch(Exception v0_3) {
                arg4.aa.ShowMessage(((Context)arg4), "不成功", "");
        }
```

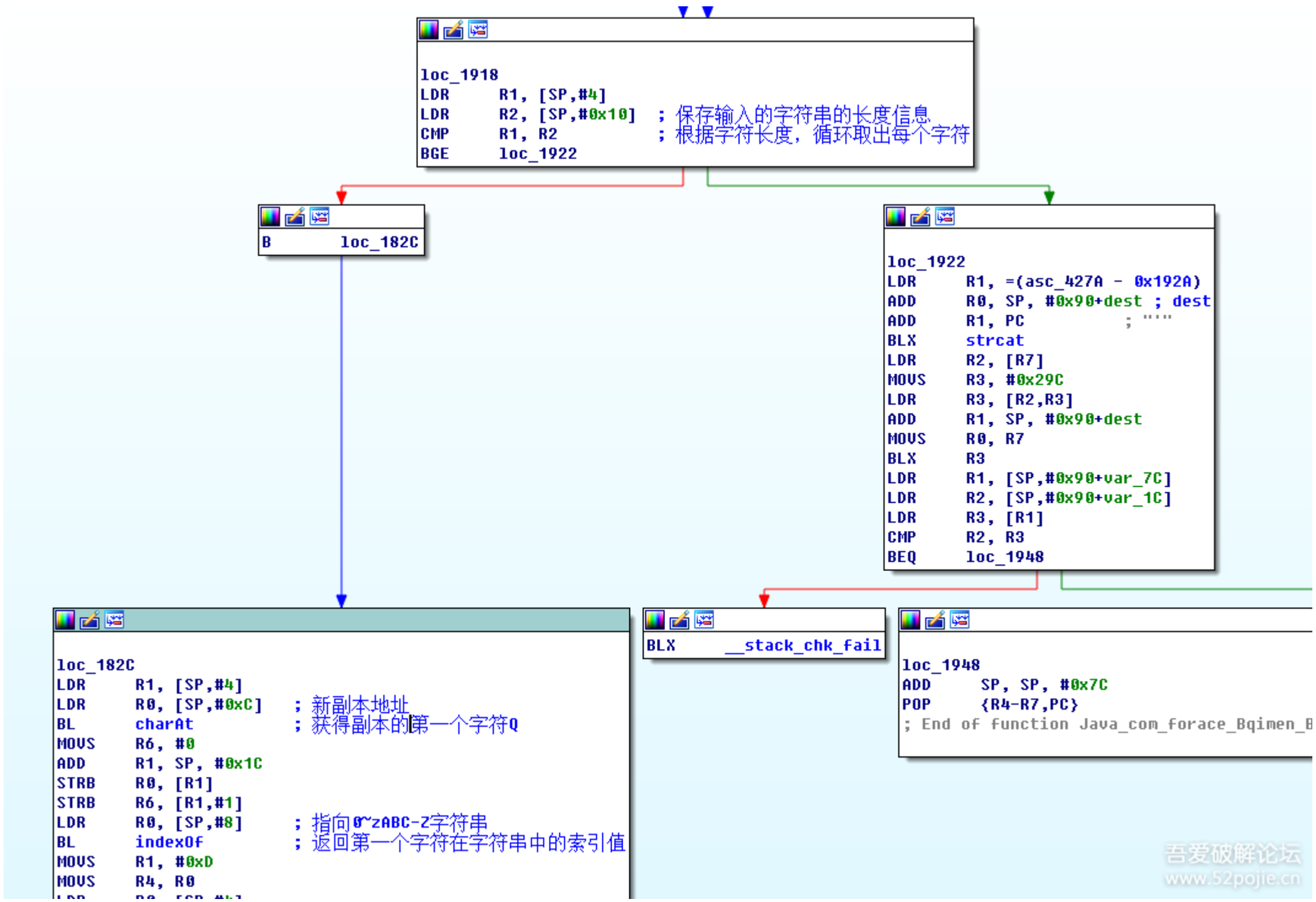
分析完软件注册的整体框架后，下面我们对动态库 so 文件里的 regetkey()方法和 regetkeyafter()方法进行分析。

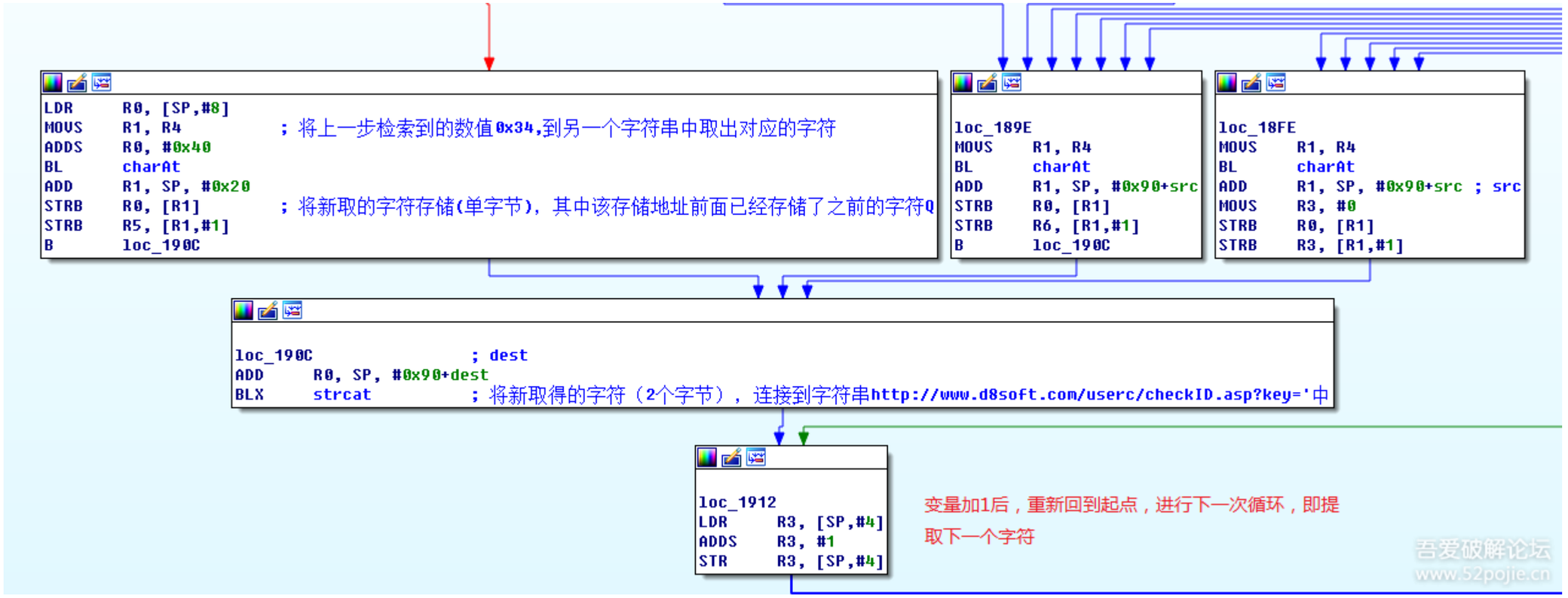
regetkey()方法根据机器码产生相应个服务器地址和查询参数，载入 IDA，找到该函数：

```
.text:000017E4
.text:000017E4                  EXPORT Java_com_forace_Bqimen_BqimenActivity_BqimenDateInputActivity_regetkey
.text:000017E4 Java_com_forace_Bqimen_BqimenActivity_BqimenDateInputActivity_regetkey
.text:000017E4
.text:000017E4 var_8C           = -0x8C
```


该函数的实现算法为：计算输入的注册码长度，并提取注册码中的每个字符，然后对每个字符进行查表的方式，获得一个新的字符，并添加到字符串"http://www.d8soft.com/userc/checkID.asp?key=""中的 Key 参数中。

```
EXPORT Java_com_forace_Bqimen_BqimenActivity_BqimenDateInputActivity_regetkey
Java_com_forace_Bqimen_BqimenActivity_BqimenDateInputActivity_regetkey

var_8C= -0x8C
var_88= -0x88
var_84= -0x84
var_80= -0x80
var_7C= -0x7C
var_74= -0x74
src= -0x70
dest= -0x6C
var_1C= -0x1C

PUSH    {R4-R7,LR}
LDR     R4, =(__stack_chk_guard_ptr - 0x17F0)
SUB     SP, SP, #0x7C
LDR     R1, [R0]
ADD     R4, PC ; __stack_chk_guard_ptr
LDR     R4, [R4] ; __stack_chk_guard
MOVS    R6, R0
MOVS    R7, R6
LDR     R3, [R4]
STR     R3, [SP,#0x74]
MOVS    R3, #0x2A4
LDR     R3, [R1,R3]
MOVS    R1, R2
MOVS    R2, #0
BLX     R3
MOVS    R5, R0          ; 对输入字符串QM000000000000000求长度
BLX     strlen
STR     R0, [SP,#0x10]  ; 长度保存地址
MOVS    R0, R5          ; s
BLX     strdup          ; 创建字符串的副本
LDR     R1, =(aHttpWww_d8soft - 0x181C)
STR     R0, [SP,#0xC]   ; 新副本的地址
ADD     R0, SP, #0x24   ; dest
ADD     R1, PC          ; "http://www.d8soft.com/userc/checkID.asp"...
BLX     strcpy
LDR     R2, =(a0123456789abcd - 0x1828)
MOVS    R1, #0
STR     R1, [SP,#4]
ADD     R2, PC          ; "0123456789abcdefghijklmnopqrstuvwxyzABC"...
STR     R4, [SP,#0x14]
STR     R2, [SP,#8]
B       loc_1918
```

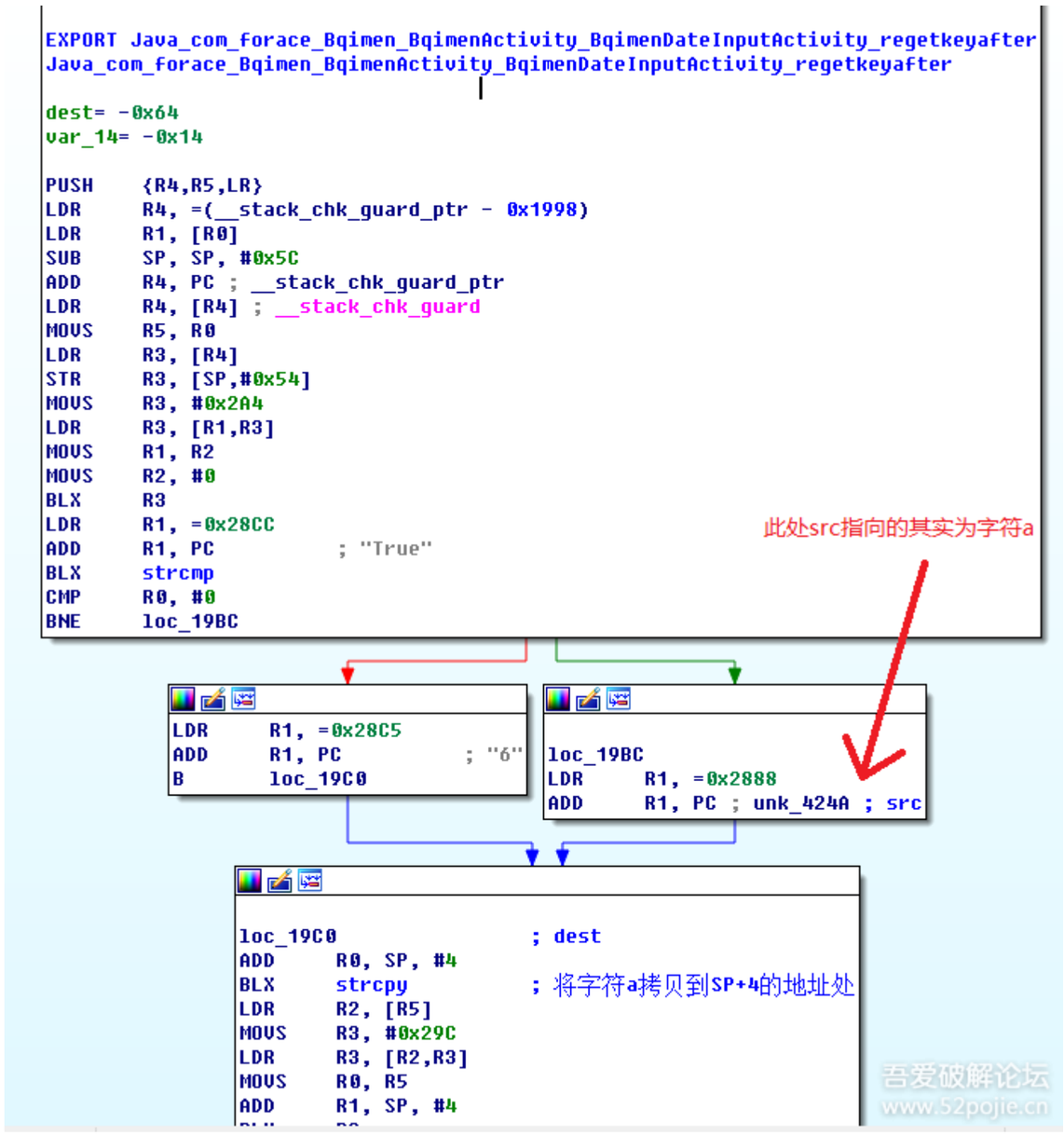




关于该算法的详细注释，我会打包到附件中，另外也可以通过动态调试的方式，获得该函数的算法。

下面对另一个函数 regetkeyafter()进行分析，其实该函数的实现是比较简单的，具体为：将查询结果与”true“进行比较，如果成功，则将返回值赋值为字符 6，否则赋值为字符 a。

如下：



上面便是整个软件的注册机制的实现和算法讲解。

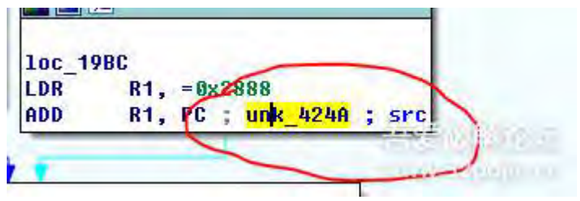
在了解了注册机制后，对与该软件的破解应该是很简单的了，下面简单的介绍两种破解方法。

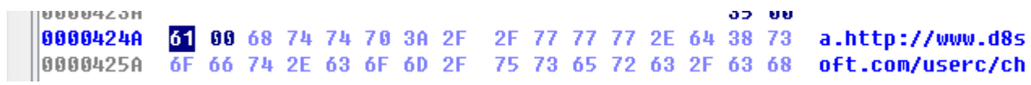
二软件多种思路破解篇

此处我会提供两种新的破解思路

1、第一种破解思路

通过前面的分析，发现其实只需要使 regetkeyafter()的返回值为字符 6 即可实现破解。而最简单的方法就是修改 src（即字符 a）处的原始字符为 6 即可，如下地址





利用 WinHex 将 61 改为 36 即可。然后打包，签名，注册->找回注册即可，注册成功。

2、第二种破解思路

既然软件需要访问服务器获得查询结果（true 或者 false），那同样在获得了完整的访问网址及参数后，可以自己搭建服务器，实现离线验证，此方法复杂，在此不在细讲，只提供一个思路。

另外此种方法适用于复杂的认证过程中，通过抓包获得地址，然后搭建伪服务器验证即可。

到此，本文的讲解就结束了。

http://www.52pojie.cn/thread-390537-1-1.html

360crackme 分析

万抽抽 myKanxue

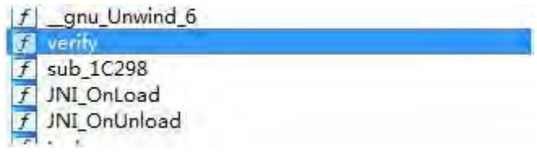
1 概述

该 crackme 的核心验证程序在 libqihoo.so 中。这个动态库经过了加密、清除 elf 文件的节区信息、花指令等处理，无法直接使用 IDA 进行静态分析。所以需要使用动态调试。不过如果仅仅使用动态跟踪的话，是很难分析出最后结果的。所以应当想法获得正确的 libqihoo.so 文件，然后结合动态调试、静态分析，最终分析出验证机制。

2 破解方法

2.1 获取较为正确的 libqihoo.so 文件

这里之所以使用较为正确，是因为我从内存 dump 出来的 libqihoo.so 文件并不完全正确，不过核心代码均有了，只是 IDA 没能自动识别出 export 函数(malloc,memset 等)，需要我们分析的时候自己加以判断，只要熟悉这些函数就很容易判断出来。
获取 so 文件的方法很简单：使用动态调试附加上 crackme 之后，运行几次确保 so 已经加载到内存，然后 ctrl+s 查看 so 在内存中的起始位置，再到 hex-view 窗口中 dump 出这段内存，另存为 libqihoo.so，然后另起一个 IDA 静态分析即可。IDA 分析后的可以发现在 Function widows 有了我们关心的 JNI_OnLoad 和 verify 函数：
如图所示：



2 开始分析

通过分析发现 JNI_OnLoad 函数将 JAVA 层的 verify 函数同 so 中的 verify 函数进行了关联。所以我们直接开始分析 verify 函数即可。Ps:方便大家查看，我将我分析过程中提取的关键代码也放出来，在 360crackmeARMcode.c 中，大家可以参照着看，那里面有详细注释。其中逻辑简单的我保留了汇编代码，逻辑复杂的就手工转换成了 c 代码。
360crakemeARMCode.c

```
typedef struct inputInfo{
	char *username;
	char *emailaddr;
	char *serialNum;
}inputInfo;


/*verify 的汇编代码如下*/
verify(){
/*

*/
	asm(
		STMFD SP!, {R4-R8,LR}
		MOV R5, R2
		SUB SP, SP, #8
		MOV R4, R0
		MOV R8, R3
		MOV R7, R1
		MOV R1, R5 //r1 = r5 = username
		MOV R0, R4
		BL __GNU_Unwind_8 //将 string 字符串转换为 C/C++的 char*
		MOV R1, R8    //r1 = r8 = emailaddr
```

```
        MOV R6, R0
        MOV R0, R4
        BL __GNU_Unwind_8
        LDR R1, [SP, #-8+arg_28]      //这是序列号
        MOV R5, R0
        MOV R0, R4
        BL __GNU_Unwind_8
        MOV R1, R7    ; r1 = r7 = jobject
        STR R0, [SP, #-8+arg_8] //将 java 层传递进来的序列号存储起来
        MOV R2, R6 //username
        MOV R0, R4
        MOV R3, R5 //emailaddr
        BL __GNU_Unwind_6
        ADD SP, SP, #8
        LDMFD SP!, {R4-R8, PC}
        STMFD SP!, {R0}
        STREQH R1, [R0], -LR
        ADD R0, R0, #4
        SUB R0, R0, #4
        BX R0
)
}

/*
此函数将 info 转换成 C/C++char[]
返回这个 char[]的第一个元素地址
*/
__GNU_Unwind_8(JNIEnv* env, jobject info){
        /*jni 函数我没太注意格式，当作伪代码看吧*/
        jclass jstringClass = env->FindClass("java/lang/String"); //返回值为 r0，赋给 r7
        jstring js_utf_8 = env->NewStringUTF("utf-8"); //返回值 r0  赋给 r6
        jmethodID jm_getBytes = env->GetMethodID(jstringClass, "getBytes", "(Ljava/lang/String;)[B"); //返回值赋给 r2
        jobject jbyteArray_object = env->CallObjectMethod(info, jm_getBytes, js_utf_8); //这里的 r5 是上层函数通过 r1 传递进来的,返回值赋给 r1,r6
        int length = env->GetArrayLength(jbyteArray_object); //返回值赋给 r7
        jbyte jb = env->GetByteElement(jbyteArray_object, 0); //返回值赋给 r8
        if(jb == null){
                goto byteElementIsNull //一般不会为空
        }
        //r0 = length + 1
        char* mychar = malloc((length+1)*sizeof(byte)); //r5
        if(mychar == null){
                goto mallo_error
```

```
        }
        memcpy(mychar, jb, length);
        mychar[length] = 0;
        env->ReleaseByteArrayElement(jbyteArray_object, jb, 0);

        return mychar;

}



__GNU_Unwind_6(env, jobject, char* username, char* emailaddr, char* serialNum){
        inputInfo input;//分配 12 字节的空间
        /*
        依次存储这三个变量的指针
        */
        input.username = username;
        input.emailaddr = emailaddr;
        input.serialNum = serialNum;

        int total_length = strlen(username) + strlen(emailaddr) + 1;
        char* total = malloc( total_length* sizeof(char));
        memset(total, 0, total_length);
        char sha1_output[40];
        memset(sha1_output, 0, 40);
        char usernma_first = username[0];
        if(usernma_first == null){
                return;
        }
        char    emailaddr_first = emailaddr[0];
        if(emailaddr_first == null){
                return;
        }
        if(strlen(serialNum) != 8 ){ //序列号必须为 8 字节 !
                return;
        }

        __GNU_Unwind_1(env, jobject, input, total, sha1_output);
}

/*
total  是 username 和 emailAddr 的连接字符串,不过被初始化为 0 了。
```

```
sha1_output 是一个大小为 40 字节的字符串，初始化全为 0
*/
__GNU_Unwind_1(JNIEnv* env, jobject, inputInfo input, char* total, char *sha1_output){

    asm{
        STMFD SP!, {R4-R10, LR}
        MOV R5, R0    ;r5 = r0 = env
        SUB SP, SP, #8
        MOV R4, R1 ;r4 = r1 = jobject
        MOV R10, R2 ; r10 = r2 = input
        MOV R7, R3 ; r7 = r3 = total
        LDR R8, [SP, #0X28]    ; r8 = sha1_output
        MOV R1, R4    ; r1 = r4 = jobject
        MOV R0, R5    ;r0 = r5 = env
        BL __GNU_Unwind_2
        MOV R1, R10 ; r1 = input
        MOV R6, R0    ;      r6 = jm_show
        MOV R0, R7    ; r0 = r7 = total
        BL __GNU_Unwind_3 ;    sprintf(total, "%s%s", input.username, input.emailaddr);
        ADD R7, SP, #8
        MOV R3, 0
        STR R3, [R7, #-4]! ;将 r7-4 ~ r7-1 的 4 字节空间置为 0，这里的！其实可以去掉
        MOV R9, R0    ;R0 = total
        MOV R1, #4
        BL __GNU_armfini_29 ； 完成对 total 的前四字节的转换
        MOV R0, R9    ;R0 = total （转换后的）
        MOV R1, R8    ; r1 = sha1_output，本来一直不知道这个变量的作用(暂命名为 temp)，在这里才发现它就是 sha1 的 output！所以赶快回溯，将变量名改为 sha1_output
        BL    __GNU_Unwind_4    ;sha1_hash
        MOV R0, R10 ; R0    = R10 = input
        MOV R1, R7    ; r1 = r7 = sp+8,前四字节全为 0
        MOV R2, R8    ; r2= r8 = sha1_output
        BL __GNU_Unwind_11 ； 对比验证
        MOV R0, R5    ; r0 = r5 = env
        MOV R1, R4    ; r1 = r4 = jobject
        MOV R2, R6    ; r2 = r6 = jm_show
        LDR R3, [SP, #4]
        BL __GNU_Unwind_10 ;判断注册码，正确的话就调用 show 方法显示"you passed..."
        ADD SP, SP, #8
        LDMFD SP!, {R4-R10, PC}

}
```

```
}

jmethodID     __GNU_Unwind_2(JNIEnv* env, jobject, 。。。){
//调用 armfini_25，
        char* activityName, activityName_base64;    //sp + 0xA4
        __GNU_armfini_25(activityName_base64); //解密，获取 base64 字符串
        __GNU_armfini_23(activityName_base64, activityName); //解码 base64 字符串
        char* show,show_base64;
        __GNU_armfini_25(show_base64);
        __GNU_armfini_23(show_base64, show);
        char* paraString, paraString_base64;
        __GNU_armfini_25(paraString_base64);
        __GNU_armfini_23(paraString_base64, paraString); //paraString = Ljava/lang/String;

        jclass jc_activity = env->FindClass("...MainActivity");
        jmethodID jm_show = env->GetMethodID(jc_activity, "show", "(Ljava/lang/String;)V");
        return jmshow;

}
```

```
/*
sprintf(total, "%s%s", input.username, input.emailaddr);
*/
__GNU_Unwind_3(char* total, inputInfo input){
        /*
        char *temp;
        temp = malloc(0x80 *sizeof(char));
        memset(temp, 0 , 0x80);
        char* username = input.username;
        char* emailaddr = input.emailaddr;
        char* format = "%s%s";
        memcpy(temp, format, strlen(format));
        */
        sprintf(total, "%s%s", input.username, input.emailaddr);


}
/*
这里涉及到解密 0x2c 字节的字符串，解密结果为一个 base64 编码的字符串。
真正的明文就是 com/qihoo/qhcrackme/MainActivity
*/
```

```
__GNU_armfini_25(char* activityName){
	asm{
		STMFD SP!, {R4, LR}
		MOV R4, R0
		MOV R0, R4
		BL Strlen   ；求出来长度为 0x2c，45 字节
		CMP R0, #0
		BLE strlenIs0
		ADD R0, R4, R0   ;r0 指向 para1 的最后一个元素的下一个。
		LDRB R3, [R4] ; r3 = 0xfc   ,★这是循环开始的地方！
		EOR R3, R3, #0X66 ;异或结果为 0x9a
		MOV R2, R3, LSR#7 ; r2 = r3 >> 7 ,逻辑右移 7 位，结果为 0x1
		AND R1, R3, #2   ; r1 = r3 & 2 = 2
		ORR R2, R2, R3, LSL#7 ; R3 逻辑左移 7 位后，再同 r2 相或, r2 |= (r3 <<7) = 0x4d01
		ORR R2, R2, R1, LSL#5 ; R2 |= (r1 << 5) = 0x4d41
		AND R1, R3, #4   ; R1 = R3&4
		ORR R1, R2, R1, LSR#3
		AND R1, R3, #8
		ORR R2, R2, R1, LSL#1
		AND R1, R3, #0X10
		ORR R2, R2, R1, LSR#1
		AND R1, R3, #0X20
		ORR R2, R2, R1, LSR#3
		AND R3, R3, #0x40
		ORR R3, R2, R3, LSR#5
		STRB R3, [R4], #1
		CMP R4, R0
		BNE loc_12340   ;如果不等就返回到循环开始的地方
		LDMFD SP!, {R4, PC}


	}
}
/*
base64 解码
*/
__GNU_armfini_23(char* base64char, char* p){

}
/*
total = input.username + input.emailaddr
length = 4  固定为 4！
```

```
*/
__GNU_armfini_29(char* total, int length ){

	unsigned char info[12] = { 0xC6, 0x96, 0xE6, 0x57,0x87, 0xF5, 0x66, 0x56, 0x96, 0x87, 0x57, 0x56 };
	unsigned char tramsformTable[0x102] = {0};
	int length = 12;
	__arm_aeabi_6(info, length, tramsformTable); //构造转换表
	__gnu_arm_message(total, 4, tramsformTable); //对 total 的前 4 字节进行变换
	memset(tramsformTable, 0, 0x102);
}

/*
info =    { 0xC6, 0x96, 0xE6, 0x57,0x87, 0xF5, 0x66, 0x56, 0x96, 0x87, 0x57, 0x56 }
length = 12
*/
__arm_aeabi_6(char* info, int length, char* transformTable){
	STMFD SP!, {R3-R8, R10, LR}
	MOV R4, R0    ;r4 = r0 = info
	MOV R5, R1    ;r5 = r1 = 0xc
	MOV R8, R2    ;r8 = r2 = transformTable
	MOV R3, #0
LABLE1:
	STRB R3, [R8, R3] ;LABLE1 的功能就是将 transformTable[0~0xff]的值依次置为 0x00~0xff
	ADD R3, R3, #1
	CMP R3, #0X100
	BNE LABLE1

	MOV R3, #0
	MOV R6, R3
	STRB R3, [R8, #0X100] ;transformTable[0x100~0x101] = null;
	STRB R3, [R8, #0X101]
	MOV R7, R8                ;r7 = r8 = transformTable
	ADD R10, R8, #0X100      ; r10 = transformTable[0x100]
	MOV R0, R3                ;r0 = r3 = 0
LABLE2:
	LDRB R2, [R4, R0]         ;r2 = info[i] , i = 0~0x100
	LDRB R3, [R7]             ;r3 = transformTable[i]
	ADD R0, R0, #1            ;i++
	MOV R1, R5                ;r1 = r5 = length
	ADD R2, R2, R3            ;r2 = info[i]+transformTable[i]
	ADD R6, R2, R6            ;r6 += r2 ,r6 初始化为 0x00
	AND R6, R6, #0XFF         ;取低位字节，其它位全置 0
```

```
        LDRB R2, [R8, R6]          ;r2 = transformTable[r6] = r6
        STRB R2, [R7], #1          ;transformTable[i] = r2, i++
        STRB R3, [R8, R6]          ;transformTable[r6] = r3 = transformTable[i-1]
        BL idivmod                    ; i%length      r0: 1->0, r1: 0xc->1, r2: r6->0xc ,r0 为商，r1 为余数,r2 为被除数
        CMP, R7, R10                  ; i <= 0x100
        AND R0, R1, #0XFF          ; i = r1 & 0xff
        BNE LABLE2
        LDMFD SP!, {R3,-R8, R10, PC}

}
/*
transformTable 共 0x102 字节,转换后的表如下图所示
  5   4b   e1   9f   78   eb   90   98   a5   a8   2c   dd   af   a0   46    1   43   49   c1   79
 d4   70   4d   3b   28   d7   19   37   da   30   15   e5   9b   10   c8   2f   59   bb   3d    3
 e4   45   ee   fc   be   b3   6b   31   2d   a2   b7   93   96   ae   17   a9   a4    a   c0   e2
 53   82   1f   e0   61   5e   c2   68   d6   67   e9   86   94   ce   47    8   14    b   de   6f
 55   4a   58   52    c   3c   32   27   74   e8   62   95   c3   f5   c4   97   7e   1c   64   38
 bd   22   57   84   2b   ad   80   b5   e7   8c   36   cf   89   65   c9   6a   f9   d2   25    0
 2a    4   ca   1a   8d   ff   c5   b6   4c   b8   f3   df   2e   ec   5f   fd   6d   7d    7   5d
 9e   35   8b   8a   b9   d1   56   69   29   cd   60   d9   ed   54   81   40   d5   8e   d3   e6
 cb   a3   b1   9c   dc   9d   24   f0    9   1b   77   b2   26   db   71    2   83   a1   39   b4
 72   fe   4e   d8   3f   f6   fb   50   21   18    6   23   bf   1e   85   87   6c   fa   d0   16
 bc   42   9a   76   51   13   92   34   44   7f   a7   ea   b0   ba   5b   f7   cc   63   75   41
  e   7c   73   f4   12   48   11   3e   5c   8f   99   66   c6   3a    f   a6   5a   6e   7b   f1
 7a   f2   aa   91   f8   ac   33   88   20   e3   1d   ab   ef   c7    d   4f    0    0
*/
__arm_aeabi_6(unsigned char* info, int length, unsigned char transformTable[0x102]){
        //先将 transformTable 初始化
        int i = 0;
        unsigned char uplevel_temp = 0x00;
        for(i; i < 0x100; i++){
                transformTable[i] = uplevel_temp;
                uplevel_temp += 0x01;
        }
        //然后开始对 info 进行变换
        unsigned char info_temp = null;
        unsigned char char_temp = 0x00;
        unsigned char char_temp2 = 0x00;
        int int_temp;
        for(i = 0; i < 0x100 ; i++){
                info_temp = info[i%length];
                uplevel_temp = transformTable[i];
```

```
            int_temp = info_temp + uplevel_temp + char_temp;
            char_temp = (unsigned char)int_temp & 0xff;
            char_temp2 = transformTable[char_temp];
            transformTable[i] = char_temp2;
            transformTable[char_temp] = uplevel_temp;
        }
}
/*

*/
__gnu_arm_message(char* total, int const_value /*4*/, unsigned char[] tramsformTable){
        asm{
        STMFD SP!, {R4-R6, LR}
        MOV R4, R1     ;r4 = r1 = 4
        MOV R5, R2     ;r5 = r2 = tramsformTable
        MOV R6, R0     ; r6 = r0 = total
        CMR R4, #0     ; r4 !=0
        LDRB R3, [R5, #0X100]     ;r3 = tramsformTable[0x100] = 0x00
        LDRB R2, [R5, #0X101]     ;r2 = tramsformTable[0x101] = 0x00
        BLE     TABLE_1
        MOV R0, R6                      ;r0 = r6 = total
        ADD R4, R6, R4                  ;r4 = r6 + r4 = total + 4    字符串加法
TABLE_2:
        ADD R3, R3, #1                  ;r3++ , r3 初始化为 0x00
        AND R3, R3, #0XFF               ;r3 &=0xff
        LDRB R1, [R5, R3]               ;r1 = tramsformTable[r3]
        ADD R2, R1, R2                  ;r2 += tramsformTable[r3] , r2 初始化为 0x00
        AND R2, R2, #0XFF               ;r2 &=0XFF
        LDRB R12, [R5, R2]              ;r12 = tramsformTable[r2]
        STRB R12, [R5, R3]              ;tramsformTable[r3] = r12
        STRB R1, [R5, R2]               ;tramsformTable[r2] = r1
        LDRB R6, [R5, R3]               ;r6 = tramsformTable[r3]
        LDRB R12, [R0]                  ;r12 = total[i], i 初始化为 0,后面会每轮加 1
        ADD R1, R1, R6                  ;r1 += r6
        AND R1, R1, #0XFF               ;r1 &=0xff
        LDRB R1, [R5, R1]               ;r1 = tramsformTable[r1]
        EOR R1, R12, R1                 ;r1 ^= r12
        STRB R1, [R0], #1               ;total[i] = r1 , i++
        CMP R0, R4                          ;r1 =? r4
        BNE TABLE_2
        STRB R3, [R5, #0X100]    ; tramsformTable[0x100] = r3
        STRB R2, [R5, #0X101]    ; tramsformTable[0x101] = r2
```

```
TABLE_1:
        LDMFD SP!, {R4-R6, PC}
}


}

/*
判断序列号
*/
__GNU_Unwind_11(char* input, char* uwind_temp1, char* sha1_result){
        serialNum[0] = sha1_result[0];
        serialNum[1] = sha1_result[2];
        serialNum[2] = sha1_result[5];
        serialNum[3] = sha1_result[9];
        serialNum[4] = sha1_result[14];
        serialNum[5] = sha1_result[20];
        serialNum[6] = sha1_result[27];
        serialNum[7] = sha1_result[35];
        serialNum[8] = 0x0;
        return  判断结果
}
/*
根据__GNU_Unwind_11 的对比结果，决定是否调用 show(“you passed。。。”)
*/
__GNU_Unwind_10{

}
```

2.1 JAVA 层传递的参数

JAVA 层传递三个参数 username, emailaddr, serialNum。这三个参数均为 String 类型。

2.2 so 文件的花指令模式

Libqihoo.so 中加入了大量的花指令，这无疑增加了逆向分析的工作量。不过只要理清了它的花指令模式，我们完全可以一劳永逸地解决这个麻烦。它的花指令模式如下：

```
LDMFD SP!, {R0}
真正指令
STMFD SP!, {R0}
ADRL R0, LOC_XXX
SUB R0, R0, #4
BX R0
```

为了方便分析，我是手工将花指令代码中的真正指令提取出来，然后进行分析的。看起来很费劲，其实是大大缩减了分析时间。

解决了花指令问题，就开始真正分析了。

2.3 本地验证流程

1)在 verify 函数中调用__GNU_Unwind_8，将三个参数(String 型的字符串)转换为 native 层的 char* username, * emailaddr, * serialNum。
2)在 verify 函数中调用__GNU_Unwind_6。此函数完成验证的核心功能。
下面开始对__GNU_Unwind_6 函数进行分析：
①新建一个结构体(命名为 inputinfo)，该结构体如下：

```
typedef struct inputInfo{
    char *username;
    char *emailaddr;
    char *serialNum;
}inputInfo;
```

然后给 inputInfo 内的三个成员赋值。
②判断 serialNum 长度是否为 8，是就继续，否则退出。
③调用__GNU_Unwind_1。在__GNU_Unwind_1 中依次调用__GNU_Unwind_2、__GNU_Unwind_3、__GNU_armfini_29、__GNU_Unwind_4、__GNU_Unwind_11、__GNU_Unwind_10。下面对这几个函数加以说明：
__GNU_Unwind_2： 此函数主要调用__GNU_armfini_25 和__GNU_armfini_23 两个函数来获取 com/qihoo/qhcrackme/MainActivity;->show(Ljava/lang/String;)V 方法 id。__GNU_armfini_25 是一个解密函数，解密出一个 base64 字符串，__GNU_armfini_23 是一个 base64 解码函数，将__GNU_armfini_25 解出的字符串转换成明文。
__GNU_Unwind_3：完成一个功能：sprintf(info, "%s%s", input.username, input.emailaddr);
★__GNU_armfini_29：这是本程序中最重要的两个函数之一。该函数首先将 username 和 emailaddr 连接在一起(这里为方便称之为 info)，然后调用__arm_aeabi_6 和__gnu_arm_message 完成对 info 信息的变换。__arm_aeabi_6 用于构造一个 0x102 字节的转换表 ， __gnu_arm_message 用于将 input 的前 4 字节进行变换(使用前面生成的转换表)。这里需要特别说明：由于程序固定转换 info 的前 4 字节，所以如果我们输入的 username+emailaddr 的字节数小于 4 的话就会发生预料不到的错误，因此一定要确保两者长度之和大于 4。
__GNU_Unwind_4：sha1_hash 函数，对变换后的 info 进行 hash，得到 sha1_result
★__GNU_Unwind_11：本程序最重要的两个函数之二。该函数功能如下：

```
serialNum[0] = sha1_result[0];
    serialNum [1] = sha1_result [2];
    serialNum [2] = sha1_result [5];
    serialNum [3] = sha1_result [9];
    serialNum [4] = sha1_result [14];
    serialNum [5] = sha1_result [20];
    serialNum [6] = sha1_result [27];
    serialNum [7] = sha1_result [35];
```

依次进行对比。
__GNU_Unwind_10：根据__GNU_Unwind_11 的对比结果来进行不同处理，如果对比成功，就调用前面获得的 show 方法，显示”you passed…”，否则就什么都不做。

至此整个程序的验证逻辑分析完毕。下面开始编写注册机代码：
呃…算了，时间有限…大家直接看附件 createSerialNum 中的代码吧~~
Ps：这个 crackme 有两个版本，我分析的是最新的版本(现今官方指定版)。听说第一个版本对 sha1 进行了部分变异，原帖参考 http://bbs.pediy.com/showthread.php?t=187906。不过最新版本并没有进行变异，在这里就偷个懒，直接将“风随雨行”大大的 sha1 代码 copy 过来，再把变异代码改成正常代码即可~()
附一组验证通过的序列号：
Username:wanchouchou
Emaildaar:1024@qq.com

SerialNum: 56c0e1fb

3 总结

以前都是静态分析，这是我第一次动态分析 so，第一次遇到花指令，第一次搞这么复杂的代码，断断续续花了 6 天的时间才搞定….不过收获还是很大的，通过 360 的第一个题，学习了 jni 的编写，第 3 个题初识了 so 的动态调试。
不过还是有一个疑惑，360 是如何清除 elf 的节区信息，使得直接 IDA 静态分析失效的呢？清除过后，又是如何在加载这个 so 文件的时候，还原的呢？望大牛们不吝赐教啊！谢谢！
转换表：
代码:

```
 5  4b  e1  9f  78  eb  90  98  a5  a8  2c  dd  af  a0  46   1  43  49  c1  79
d4  70  4d  3b  28  d7  19  37  da  30  15  e5  9b  10  c8  2f  59  bb  3d   3
e4  45  ee  fc  be  b3  6b  31  2d  a2  b7  93  96  ae  17  a9  a4   a  c0  e2
53  82  1f  e0  61  5e  c2  68  d6  67  e9  86  94  ce  47   8  14   b  de  6f
55  4a  58  52   c  3c  32  27  74  e8  62  95  c3  f5  c4  97  7e  1c  64  38
bd  22  57  84  2b  ad  80  b5  e7  8c  36  cf  89  65  c9  6a  f9  d2  25   0
2a   4  ca  1a  8d  ff  c5  b6  4c  b8  f3  df  2e  ec  5f  fd  6d  7d   7  5d
9e  35  8b  8a  b9  d1  56  69  29  cd  60  d9  ed  54  81  40  d5  8e  d3  e6
cb  a3  b1  9c  dc  9d  24  f0   9  1b  77  b2  26  db  71   2  83  a1  39  b4
72  fe  4e  d8  3f  f6  fb  50  21  18   6  23  bf  1e  85  87  6c  fa  d0  16
bc  42  9a  76  51  13  92  34  44  7f  a7  ea  b0  ba  5b  f7  cc  63  75  41
 e  7c  73  f4  12  48  11  3e  5c  8f  99  66  c6  3a   f  a6  5a  6e  7b  f1
7a  f2  aa  91  f8  ac  33  88  20  e3  1d  ab  ef  c7   d  4f   0   0
```

注：本帖由看雪论坛志愿者 PEstone 重新将文档整理排版，若和原文有出入，以原作者附件为准

Ida pro 查看函数调用关系

IDA 调试的时候，已经停在断点了，怎么看到函数调用关系
类似 gdb 的 bt 命令
司马(303345229) 10:32:07
貌似 ida 没这个功能.
司马(303345229) 10:32:24
看 lr 倒是可以追到上一级.

ARM

Bin 文件

IDA 如何识别 ARM 的 main 函数

不管是动态链接还是静态链接的程序（不包括.so 文件），其_start 函数都是一样的。
android 源码 crtbegin_static.S 中的_start 函数如下(抽取重要部分)：

```
_start:
    mov r0,sp         @取堆栈指针 sp 指向 main 函数的参数 argc(堆栈内容依次为 argc,argv[],argc[])
    mov r1,#0
    adr r2,0f         @标号 0 地址，即执行 main 函数的指针
    adr r3,1f         @标号 1 地址，即数组指针
    b __libc_init
    .....
```

可以看出 main 函数即_start 调用_libc_init 时，寄存器 R2 的值.

静态链接程序实例 IDA 的反汇编代码（加上自己的注释）：

```
.text:000080E0 ; Attributes: bp-based frame
.text:000080E0
.text:000080E0 EXPORT start
.text:000080E0 start
.text:000080E0
.text:000080E0 preinit_array= -0x14
.text:000080E0 init_array= -0x10
.text:000080E0 fini_array= -0xC
.text:000080E0 ctors= -8
.text:000080E0
.text:000080E0 LDR      R12, =(dword_1DFF4 - 0x80FC)
.text:000080E4 STMFD    SP!, {R11,LR}
.text:000080E8 LDR      R3, =0xFFFFFF68
.text:000080EC ADD      R11, SP, #4
.text:000080F0 SUB      SP, SP, #0x10
.text:000080F4 ADD      R12, PC, R12 ; dword_1DFF4
.text:000080F8 LDR      R3, [R12,R3]
.text:000080FC STR      R3, [R11,#preinit_array]        ; unk_1df00
```

```
.text:00008100 LDR      R3, =0xFFFFFF6C
.text:00008104 ADD      R0, R11, #4
.text:00008108 LDR      R3, [R12,R3]
.text:0000810C STR      R3, [R11,#init_array]          ; unk_1def4
.text:00008110 LDR      R3, =0xFFFFFF70
.text:00008114 MOV      R1, #0
.text:00008118 LDR      R3, [R12,R3]
.text:0000811C STR      R3, [R11,#fini_array]          ; unk_1deec
.text:00008120 LDR      R3, =0xFFFFFF74
.text:00008124 LDR      R3, [R12,R3]
.text:00008128 STR      R3, [R11,#ctors]               ; unk_1df08
.text:0000812C LDR      R3, =0xFFFFFF78
.text:00008130 LDR      R2, [R12,R3]                   ; r2 is main .
.text:00008130                                         ; r2 is 0x8240
.text:00008134 SUB      R3, R11, #-preinit_array
.text:00008138 BL       __libc_init
.text:0000813C SUB      SP, R11, #4
.text:00008140 LDMFD    SP!, {R11,PC}
.text:00008140 ; End of function start
.text:00008140
.text:00008140 ; ---------------------------------------------------------------------------
```

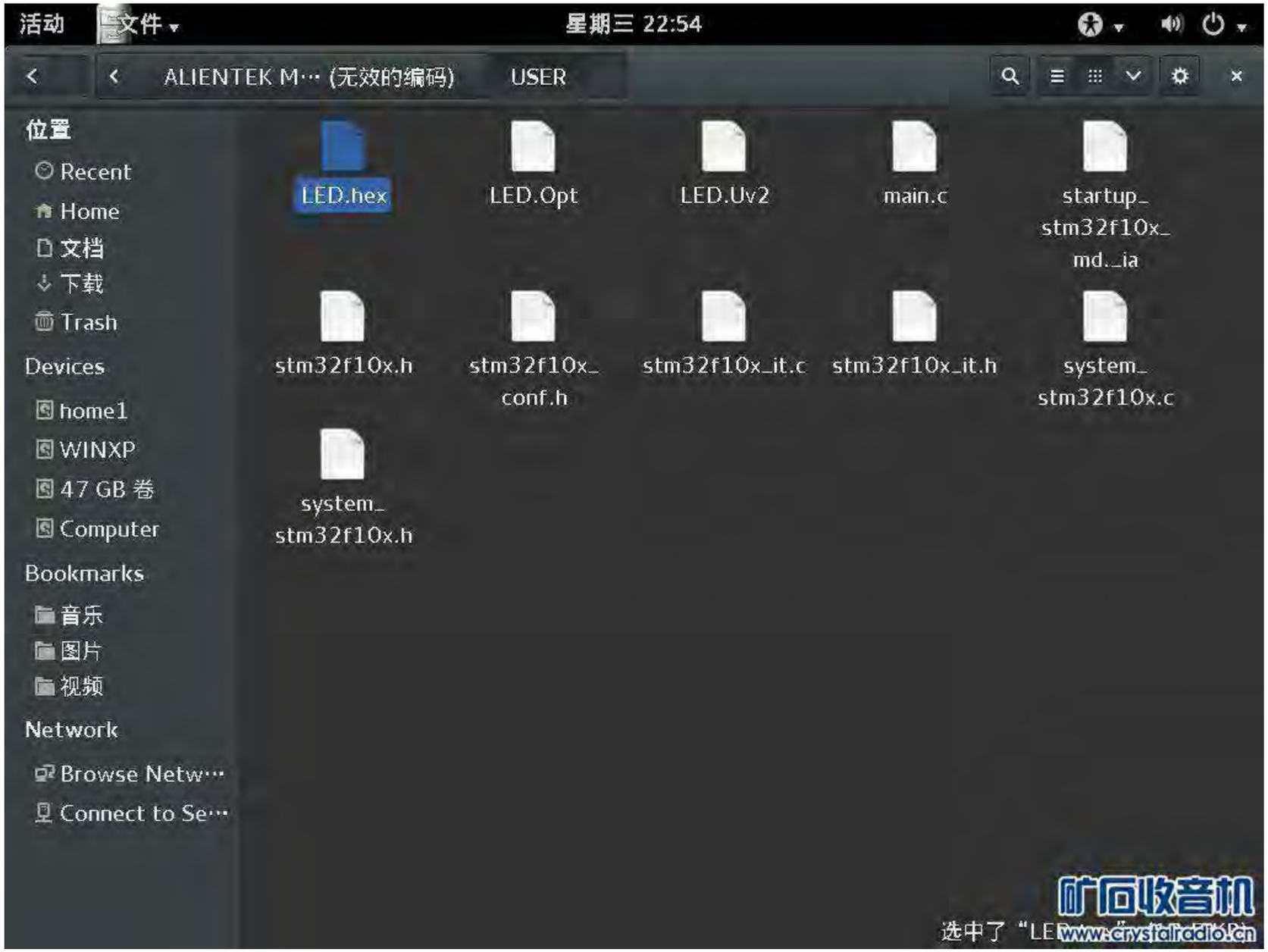
http://blog.csdn.net/zhangmiaoping23/article/details/39672757

用 IDA 反汇编个 STM32 小程序

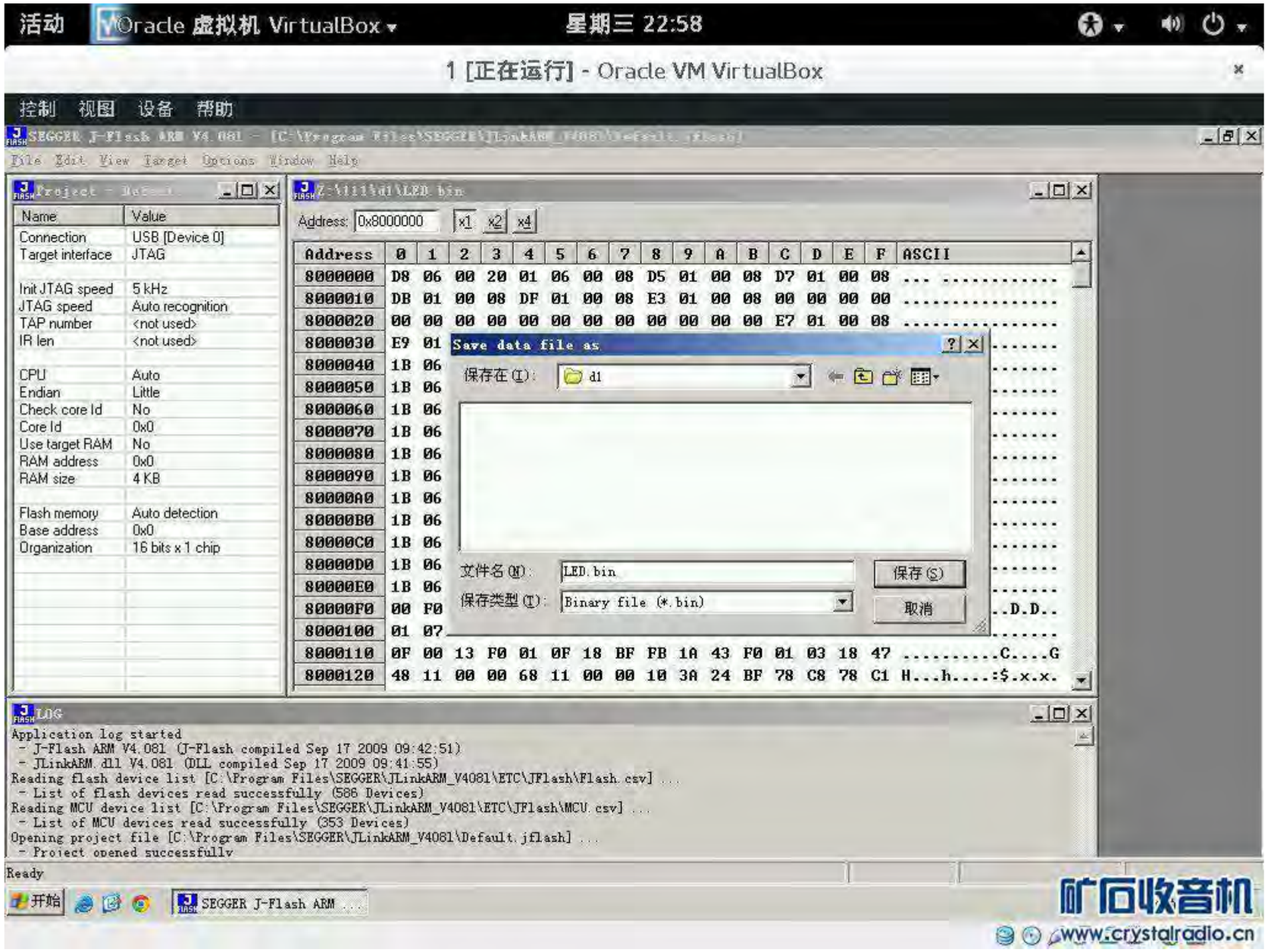
想仿制别人的一款产品，但是只有固件没有源码无法修改程序，于是研究了一下反汇编。

下面把过程贴出来，方法对错我也不知道，我小学文化不懂英文操作这款软件有困难。

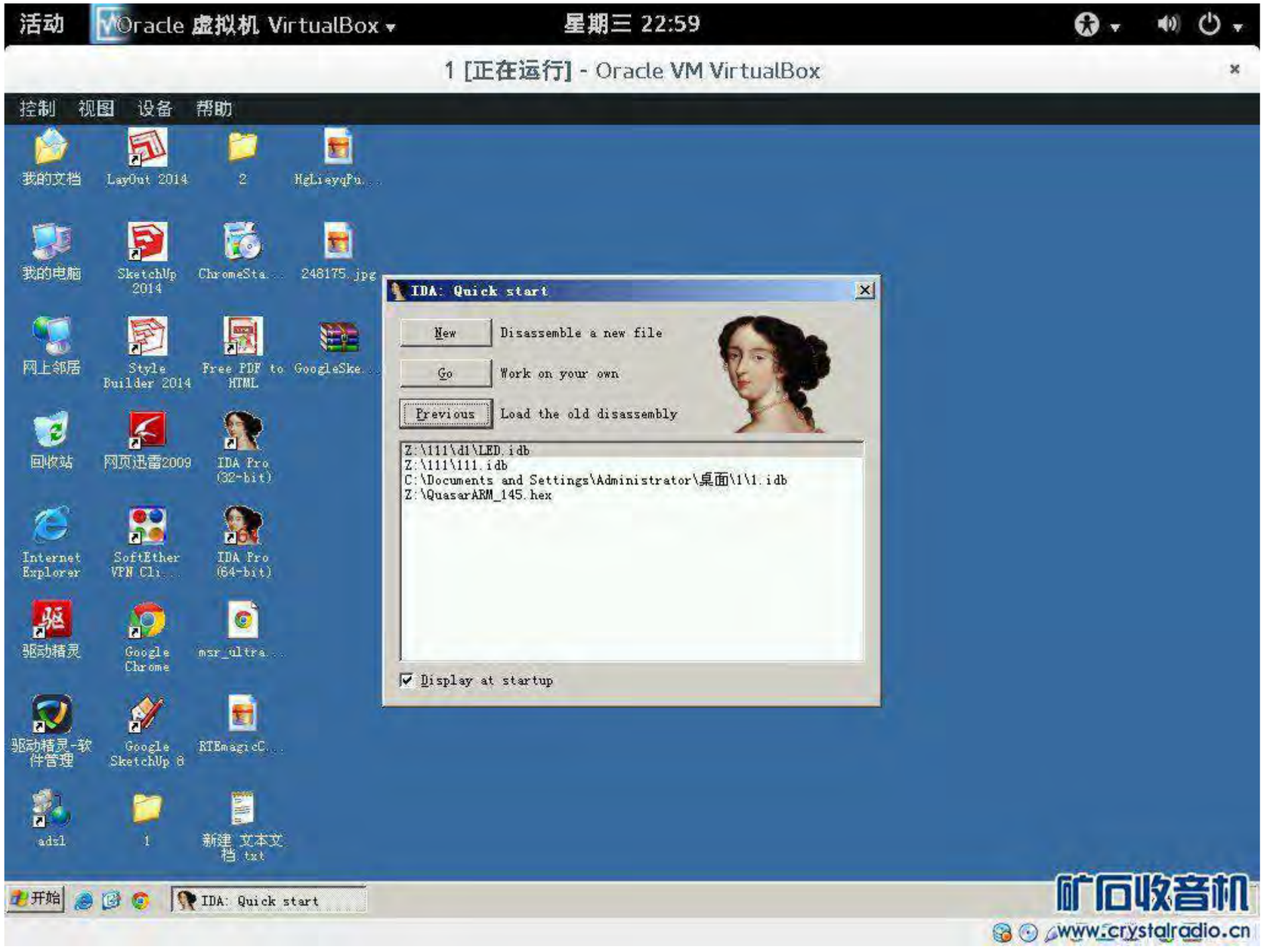
目标某开发的一个 LED 小程序



把 HEX 文件转为 BIN 文件


活动
Oracle 虚拟机 VirtualBox
星期三 22:58
1 [正在运行] - Oracle VM VirtualBox
控制 视图 设备 帮助
File Edit View Target Options Window Help
Name Value
Connection USB [Device 0]
Target interface JTAG
Init JTAG speed 5 kHz
JTAG speed Auto recognition
TAP number <not used>
IR len <not used>
CPU Auto
Endian Little
Check core Id No
Core Id 0x0
Use target RAM No
RAM address 0x0
RAM size 4 KB
Flash memory Auto detection
Base address 0x0
Organization 16 bits x 1 chip
Address: 0x8000000
x1 x2 x4
Address 0 1 2 3 4 5 6 7 8 9 A B C D E F ASCII
8000000 D8 06 00 20 01 06 00 08 D5 01 00 08 D7 01 00 08
8000010 DB 01 00 08 DF 01 00 08 E3 01 00 08 00 00 00 00
8000020 00 00 00 00 00 00 00 00 00 00 00 00 E7 01 00 08
8000030 E9 01
8000040 1B 06
8000050 1B 06
8000060 1B 06
8000070 1B 06
8000080 1B 06
8000090 1B 06
80000A0 1B 06
80000B0 1B 06
80000C0 1B 06
80000D0 1B 06
80000E0 1B 06
80000F0 00 F0
8000100 01 07
8000110 0F 00 13 F0 01 0F 18 BF FB 1A 43 F0 01 03 18 47
8000120 48 11 00 00 68 11 00 00 10 3A 24 BF 78 C8 78 C1
Save data file as
保存在(I): d1
文件名(N): LED.bin
保存类型(T): Binary file (*.bin)
保存(S)
取消
LOG
Application log started
- J-Flash ARM V4.08l (J-Flash compiled Sep 17 2009 09:42:51)
- JLinkARM.dll V4.08l (DLL compiled Sep 17 2009 09:41:55)
Reading flash device list [C:\Program Files\SEGGER\JLinkARM_V408l\ETC\JFlash\Flash.csv] ...
- List of flash devices read successfully (586 Devices)
Reading MCU device list [C:\Program Files\SEGGER\JLinkARM_V408l\ETC\JFlash\MCU.csv] ...
- List of MCU devices read successfully (353 Devices)
Opening project file [C:\Program Files\SEGGER\JLinkARM_V408l\Default.jflash] ...
- Project opened successfully
Ready
开始
SEGGER J-Flash ARM ...
矿石收音机
www.crystalradio.cn

打开 IDA


活动
Oracle 虚拟机 VirtualBox
星期三 22:59
1 [正在运行] - Oracle VM VirtualBox
控制 视图 设备 帮助
我的文档
LayOut 2014
2
HgLieyqFu...
我的电脑
SketchUp 2014
ChromeSta...
248175.jpg
网上邻居
Style Builder 2014
Free PDF to HTML
GoogleSke...
回收站
网页迅雷2009
IDA Pro (32-bit)
Internet Explorer
SoftEther VPN Cli...
IDA Pro (64-bit)
驱动精灵
Google Chrome
msr_ultra...
驱动精灵-软件管理
Google SketchUp 8
RTEmagicC...
ads1
1
新建 文本文档.txt
IDA: Quick start
New
Disassemble a new file
Go
Work on your own
Previous
Load the old disassembly
Z:\111\d1\LED.idb
Z:\111\111.idb
C:\Documents and Settings\Administrator\桌面\1\1.idb
Z:\QuasarARM_145.hex
Display at startup
开始
IDA: Quick start
矿石收音机
www.crystalradio.cn

打开刚刚的 BIN 文件

活动 Oracle 虚拟机 VirtualBox ▾ 星期三 23:00

1 [正在运行] - Oracle VM VirtualBox

控制 视图 设备 帮助

IDA v6.5.140116

File Edit Jump Search View Debugger Options Windows Help

New instance
Open...
Load file
Produce file
Script file... Alt+F7
Script command... Shift+F2
Save Ctrl+W
Save as...
Take database snapshot... Ctrl+Shift+W
Close

Quick start

0. Z:\111\d1\LED.idb
1. Z:\111\111.idb
2. C:\Documents and Settings\Administrator\桌面\1\1.idb
3. Z:\QuasarARM_145.hex

Exit Alt+X

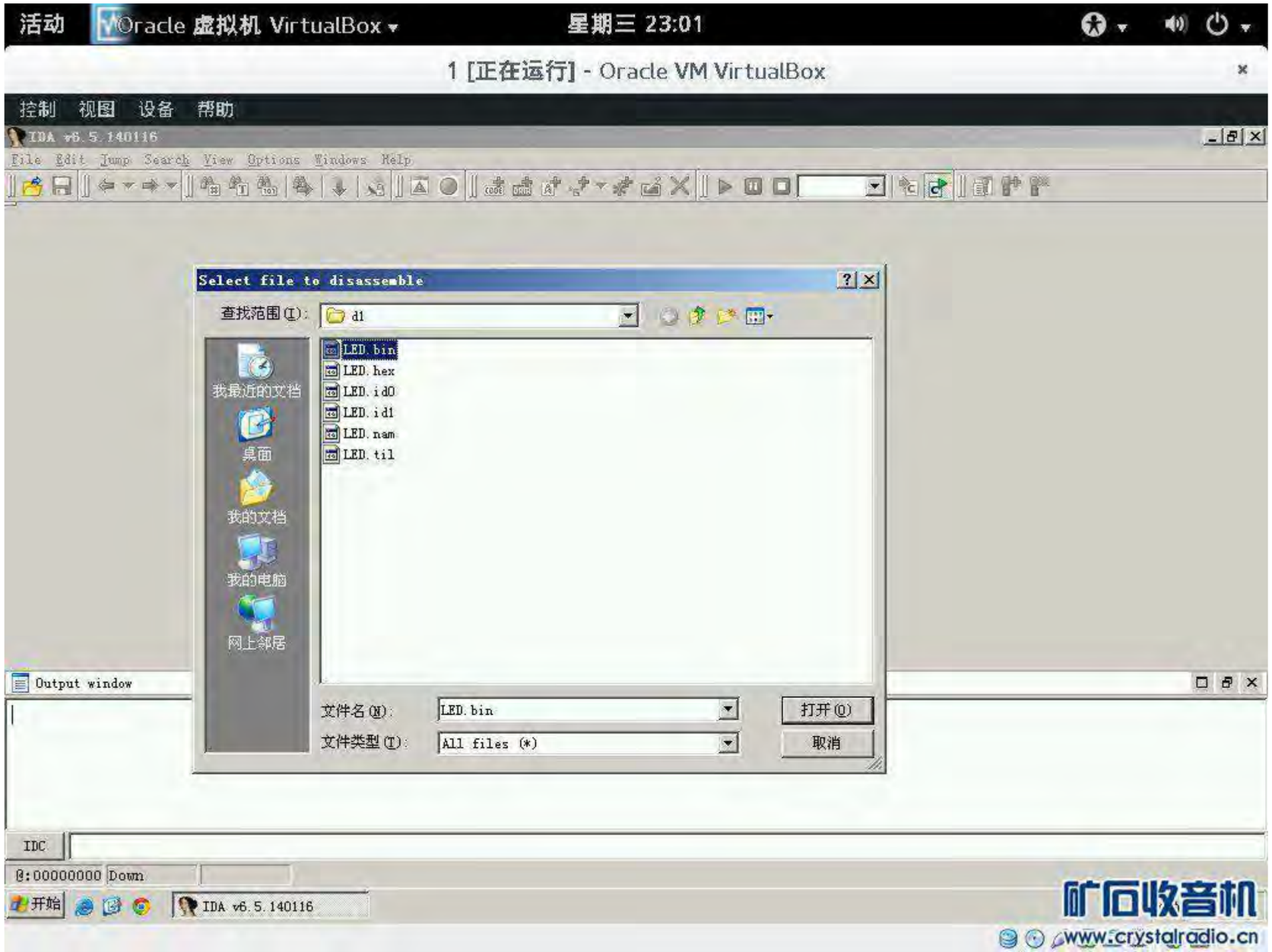
file here to disassemble it

Output window

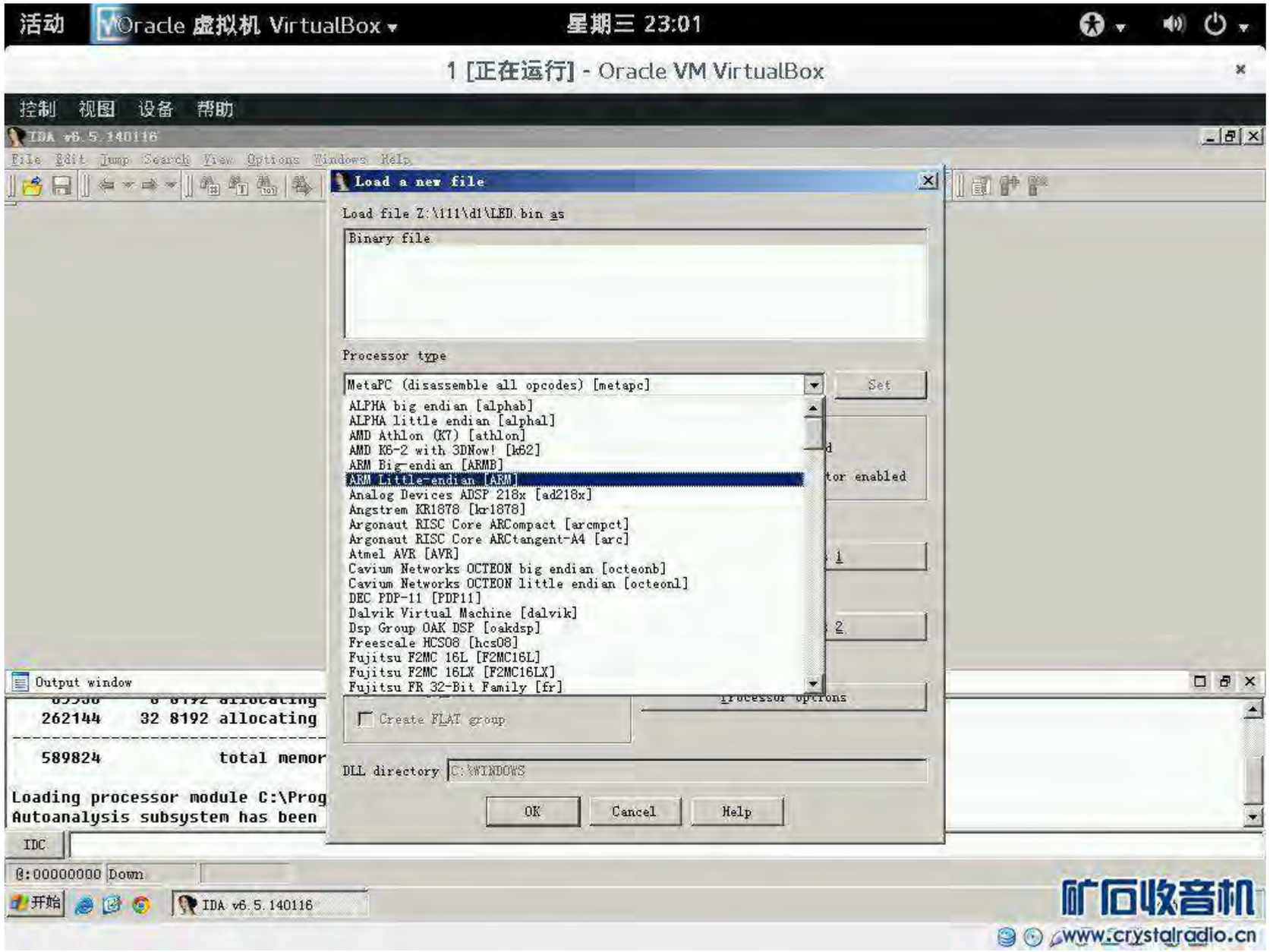
IDC

Load a new file or database

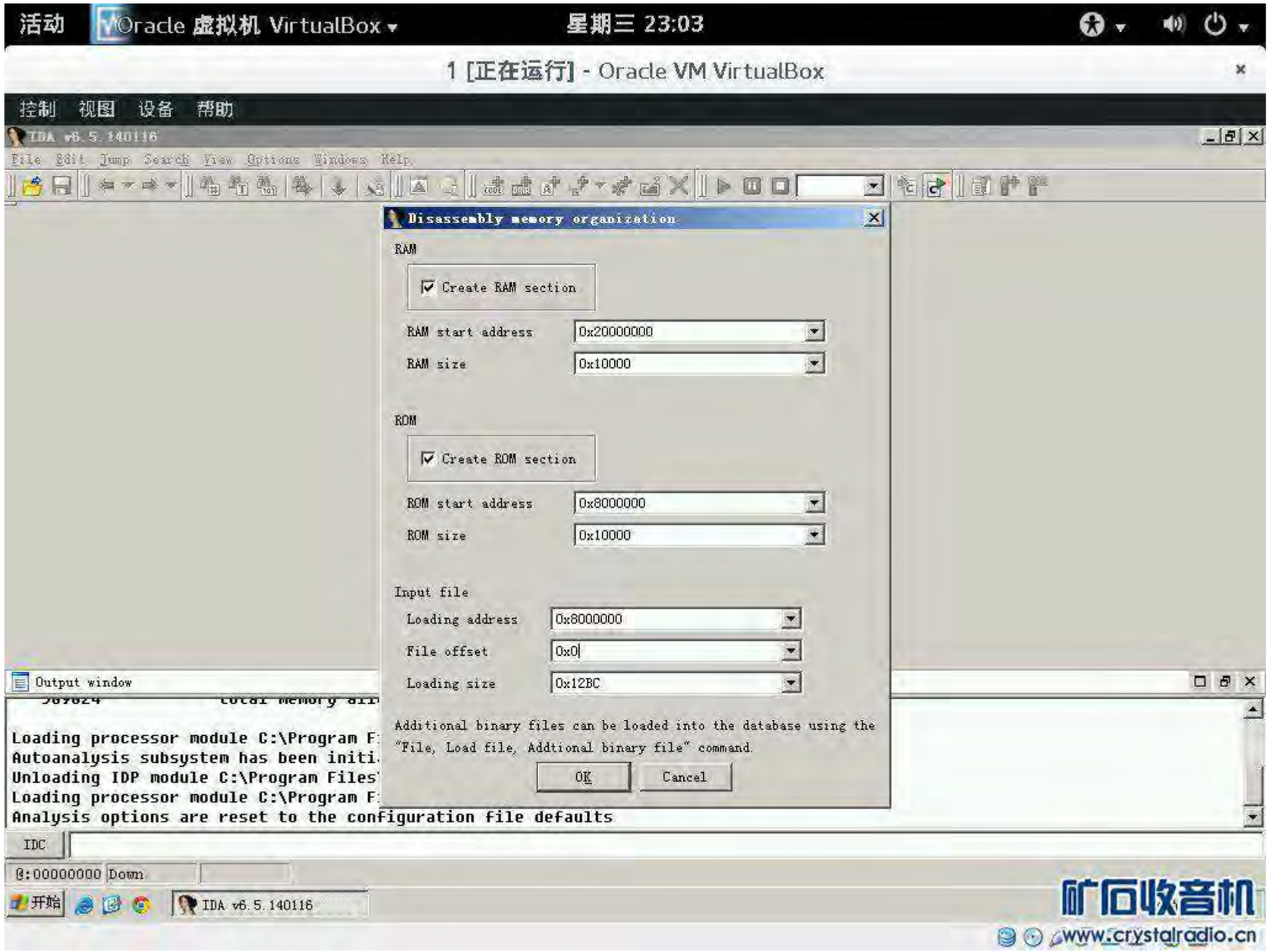
开始 IDA v6.5.140116



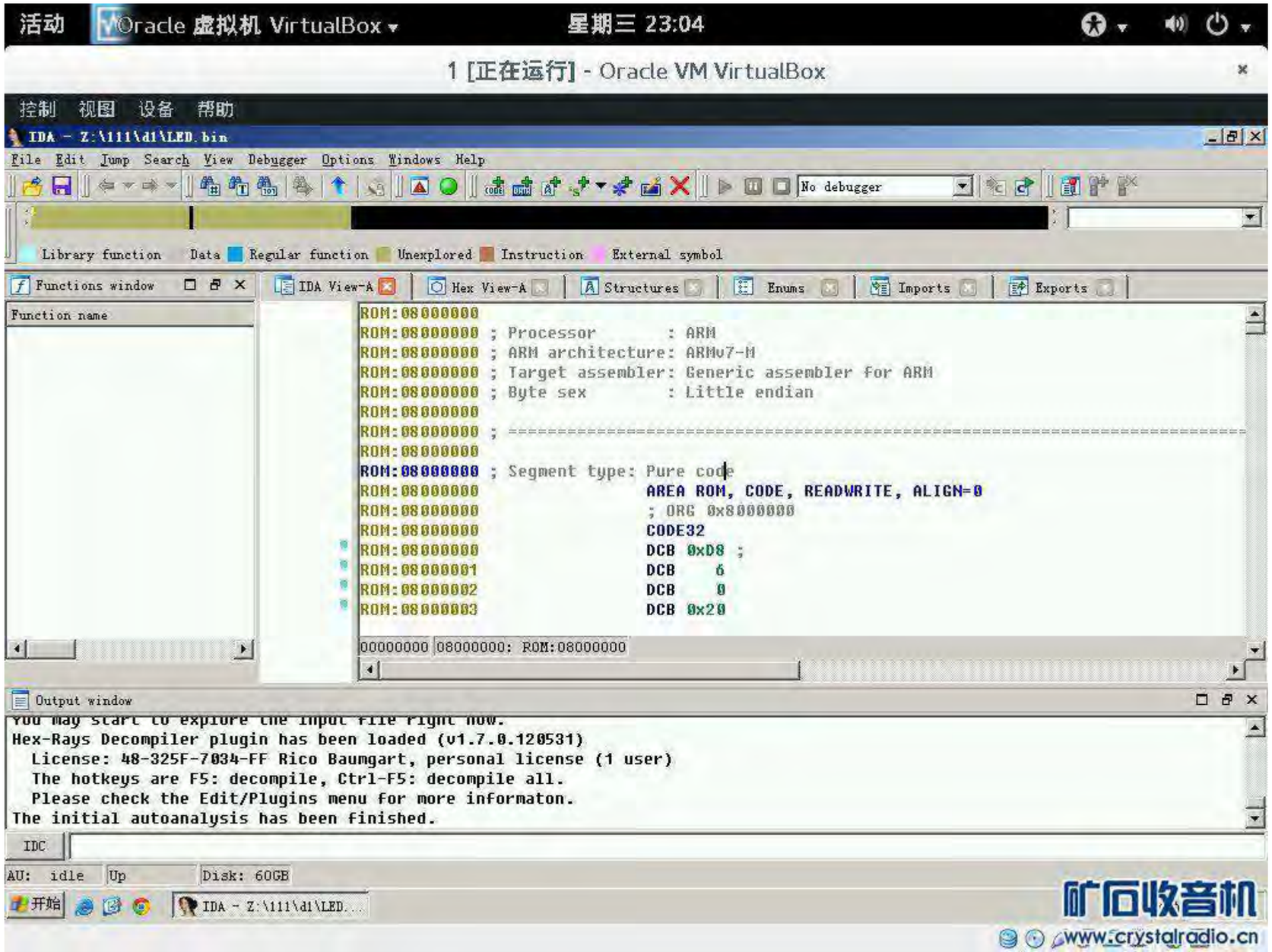
选择 ARM：
然后还有个高级选项选择 Cortex-M （忘记截图）


活动
Oracle 虚拟机 VirtualBox
星期三 23:01
1 [正在运行] - Oracle VM VirtualBox
控制 视图 设备 帮助
IDA v6.5.140116
File Edit Jump Search View Options Windows Help
Load a new file
Load file Z:\111\d1\LED.bin as
Binary file
Processor type
MetaPC (disassemble all opcodes) [metapc]
Set
ALPHA big endian [alphab]
ALPHA little endian [alphal]
AMD Athlon (K7) [athlon]
AMD K6-2 with 3DNow! [k62]
ARM Big-endian [ARMB]
ARM Little-endian [ARM]
Analog Devices ADSP 218x [ad218x]
Angstrem KR1878 [kr1878]
Argonaut RISC Core ARCompact [arcmpct]
Argonaut RISC Core ARCtangent-A4 [arc]
Atmel AVR [AVR]
Cavium Networks OCTEON big endian [octeonb]
Cavium Networks OCTEON little endian [octeonl]
DEC PDP-11 [PDP11]
Dalvik Virtual Machine [dalvik]
Dsp Group OAK DSP [oakdsp]
Freescale HCS08 [hcs08]
Fujitsu F2MC 16L [F2MC16L]
Fujitsu F2MC 16LX [F2MC16LX]
Fujitsu FR 32-Bit Family [fr]
Create FLAT group
DLL directory C:\WINDOWS
OK
Cancel
Help
Output window
262144 32 8192 allocating
589824 total memor
Loading processor module C:\Prog
Autoanalysis subsystem has been
IDC
@:00000000 Down
开始
IDA v6.5.140116
矿石收音机
www.crystalradio.cn

ROM 地址

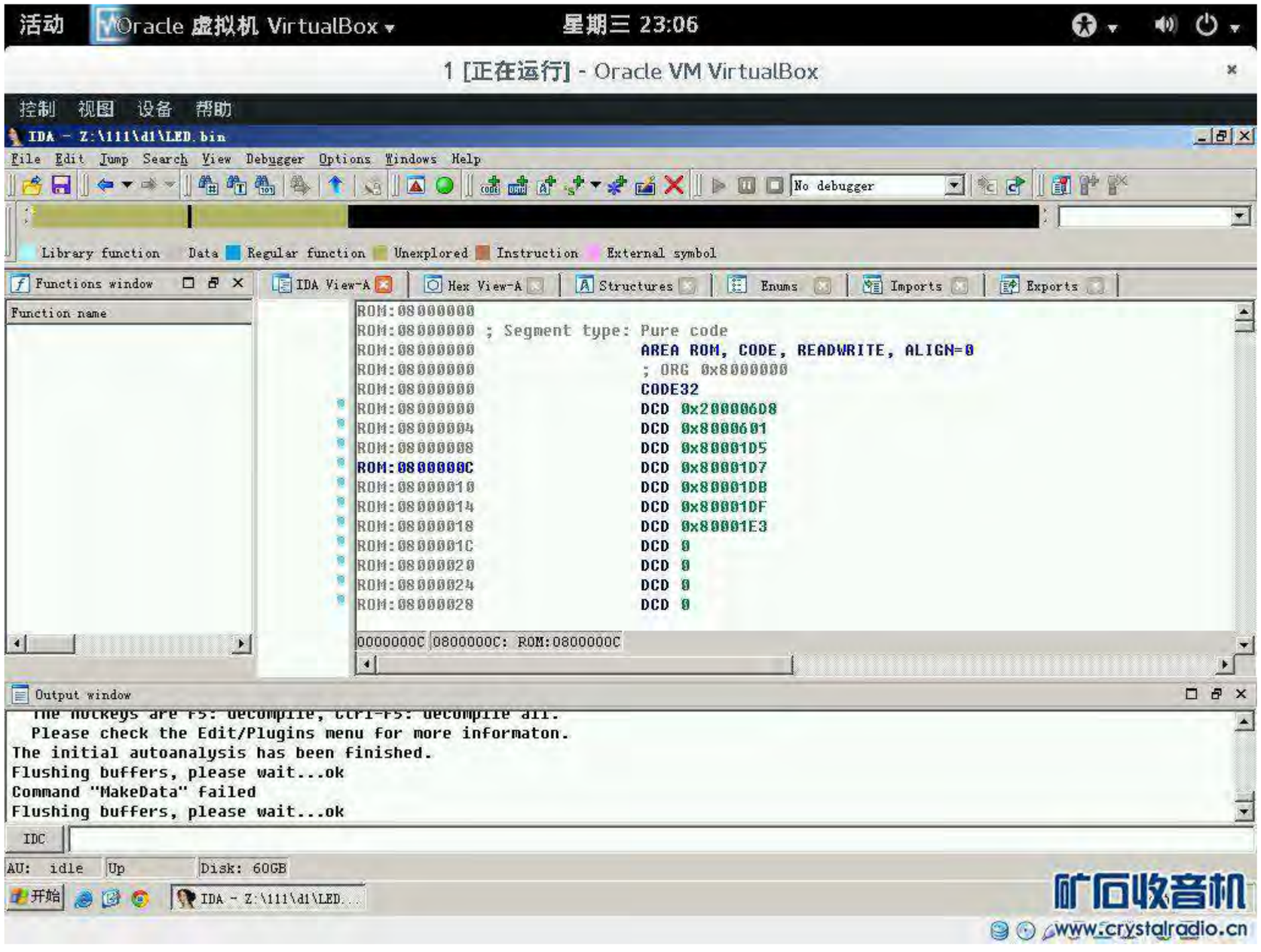

活动
Oracle 虚拟机 VirtualBox
星期三 23:03
1 [正在运行] - Oracle VM VirtualBox
控制 视图 设备 帮助
IDA v6.5.140116
File Edit Jump Search View Options Windows Help
Disassembly memory organization
RAM
Create RAM section
RAM start address 0x20000000
RAM size 0x10000
ROM
Create ROM section
ROM start address 0x8000000
ROM size 0x10000
Input file
Loading address 0x8000000
File offset 0x0
Loading size 0x12BC
Additional binary files can be loaded into the database using the "File, Load file, Addtional binary file" command.
OK
Cancel
Output window
Loading processor module C:\Program F
Autoanalysis subsystem has been initi
Unloading IDP module C:\Program Files
Loading processor module C:\Program F
Analysis options are reset to the configuration file defaults
IDC
@:00000000 Down
开始
IDA v6.5.140116
矿石收音机
www.crystalradio.cn

反汇编

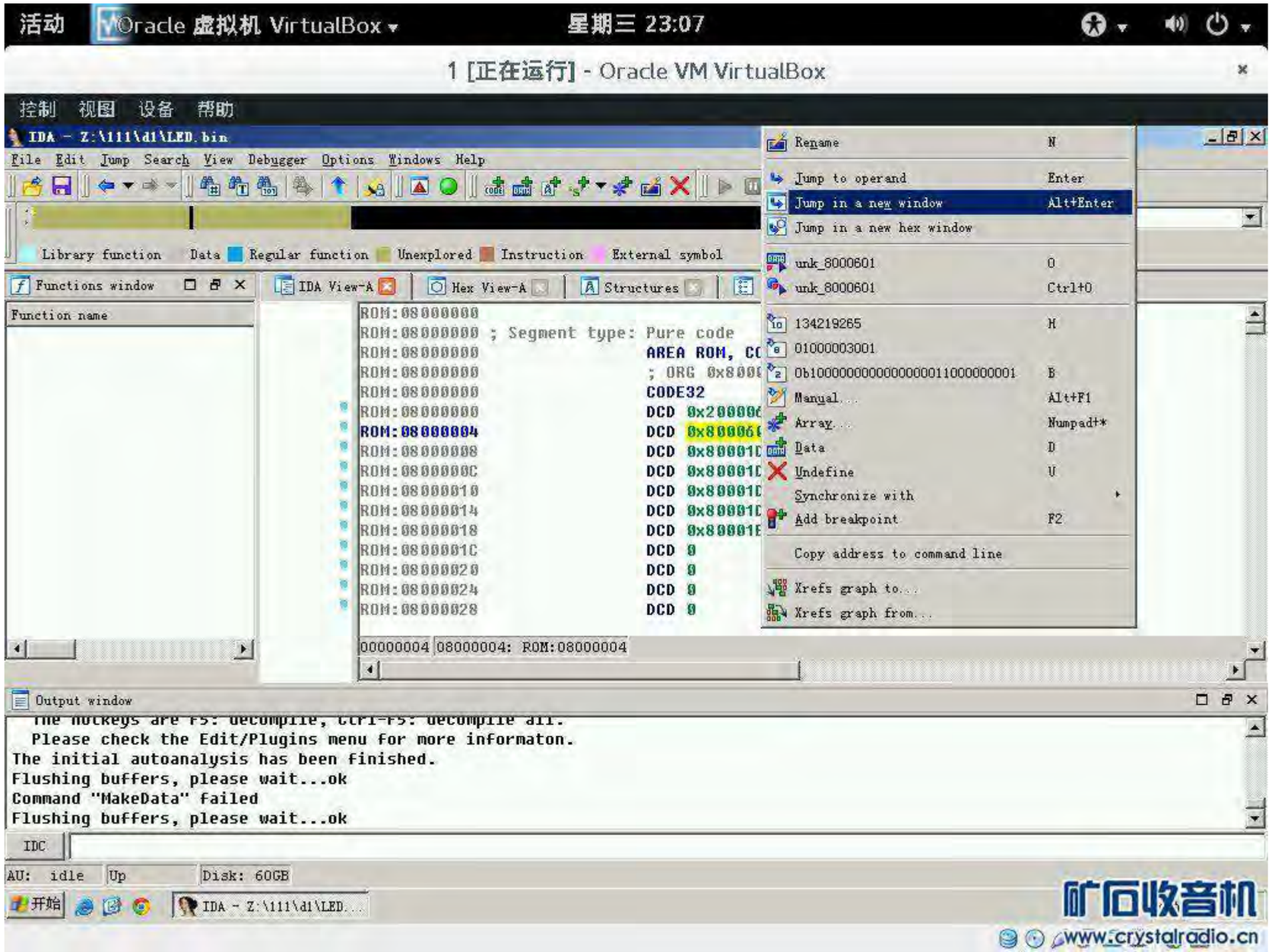


开头的是数据， 第一个是站顶，第二个是复位向量，我们从他解开程序

按 D 健转换



跳转到复位向量指向的地址


活动
Oracle 虚拟机 VirtualBox
星期三 23:07
1 [正在运行] - Oracle VM VirtualBox
控制 视图 设备 帮助
IDA - Z:\111\d1\LED.bin
File Edit Jump Search View Debugger Options Windows Help
Library function Data Regular function Unexplored Instruction External symbol
Functions window
Function name
IDA View-A
Hex View-A
Structures
ROM:08000000
ROM:08000000 ; Segment type: Pure code
ROM:08000000 AREA ROM, CO
ROM:08000000 ; ORG 0x800
ROM:08000000 CODE32
ROM:08000000 DCD 0x200000
ROM:08000004 DCD 0x800061
ROM:08000008 DCD 0x80001D
ROM:0800000C DCD 0x80001D
ROM:08000010 DCD 0x80001D
ROM:08000014 DCD 0x80001D
ROM:08000018 DCD 0x80001F
ROM:0800001C DCD 0
ROM:08000020 DCD 0
ROM:08000024 DCD 0
ROM:08000028 DCD 0
00000004 08000004: ROM:08000004
Rename N
Jump to operand Enter
Jump in a new window Alt+Enter
Jump in a new hex window
unk_8000601 O
unk_8000601 Ctrl+O
134219265 H
01000003001
0b10000000000000000011000000001 B
Manual... Alt+F1
Array... Numpad+*
Data D
Undefine U
Synchronize with
Add breakpoint F2
Copy address to command line
Xrefs graph to...
Xrefs graph from...
Output window
Please check the Edit/Plugins menu for more informaton.
The initial autoanalysis has been finished.
Flushing buffers, please wait...ok
Command "MakeData" failed
Flushing buffers, please wait...ok
IDC
AU: idle Up Disk: 60GB
开始
IDA - Z:\111\d1\LED...
矿石收音机
www.crystalradio.cn

按 C 键转换成代码

活动 Oracle 虚拟机 VirtualBox ▾ 星期三 23:08

1 [正在运行] - Oracle VM VirtualBox

控制 视图 设备 帮助

IDA - Z:\111\d1\LED.bin

File Edit Jump Search View Debugger Options Windows Help

No debugger

Library function Data Regular function Unexplored Instruction External symbol

Functions window | IDA View-A | IDA View-B | Hex View-A | Structures | Enums | Imports | Exports

Function name

```
ROM:080005FC                DCB 0x34 ; 4
ROM:080005FD                DCB    0
ROM:080005FE                DCB    0
ROM:080005FF                DCB 0x20
ROM:08000600                DCB    9
ROM:08000601 unk_8000601    DCB 0x48 ; H
ROM:08000602                DCB 0x80 ; €
ROM:08000603                DCB 0x47 ; G
ROM:08000604                DCB    9
ROM:08000605                DCB 0x48 ; H
ROM:08000606                DCB    0
ROM:08000607                DCB 0x47 ; G
ROM:08000608                DCB 0xFE ;
ROM:08000609                DCB 0xE7 ;
ROM:0800060A                DCB 0xFE ;
ROM:0800060B                DCB 0xE7 ;
```

00000601 08000601: ROM:unk_8000601

Output window

```
  The hotkeys are F5: decompile, Ctrl-F5: decompile all.
  Please check the Edit/Plugins menu for more informaton.
The initial autoanalysis has been finished.
Flushing buffers, please wait...ok
Command "MakeData" failed
Flushing buffers, please wait...ok
```

IDC

AU: idle Up Disk: 60GB

开始 IDA - Z:\111\d1\LED...

活动 Oracle 虚拟机 VirtualBox ▾ 星期三 23:15

1 [正在运行] - Oracle VM VirtualBox

控制 视图 设备 帮助

IDA - Z:\111\d1\LED.bin

File Edit Jump Search View Debugger Options Windows Help

No debugger

Library function Data Regular function Unexplored Instruction External symbol

Functions window | IDA View-A | IDA View-B | Hex View-A | Structures | Enums | Imports | Exports

Function name

- sub_8000160
- sub_8000168
- sub_8000188
- sub_80001F0
- sub_80002B0
- sub_80002B8
- sub_80003B8
- sub_800041C
- sub_8000444
- sub_80004B4
- sub_80004C0
- sub_800061C
- sub_8000640
- sub_80006E4
- sub_80007E0
- sub_8000930

```
ROM:080005FD                DCB    0
ROM:080005FE                DCB    0
ROM:080005FF                DCB 0x20
ROM:08000600 ; ---------------------------------------------------------------------------
ROM:08000600                CODE32
ROM:08000600                LDR     R0, =(sub_80002B8+1)
ROM:08000602                BLX     R0 ; sub_80002B8
ROM:08000604                LDR     R0, =(loc_80000EC+1)
ROM:08000606                BX      R0 ; loc_80000EC
ROM:08000606 ; ---------------------------------------------------------------------------
ROM:08000608                DCB 0xFE ;
ROM:08000609                DCB 0xE7 ;
ROM:0800060A                DCB 0xFE ;
ROM:0800060B                DCB 0xE7 ;
ROM:0800060C                DCB 0xFE ;
ROM:0800060D                DCB 0xE7 ;
```

00000600 08000600: ROM:08000600

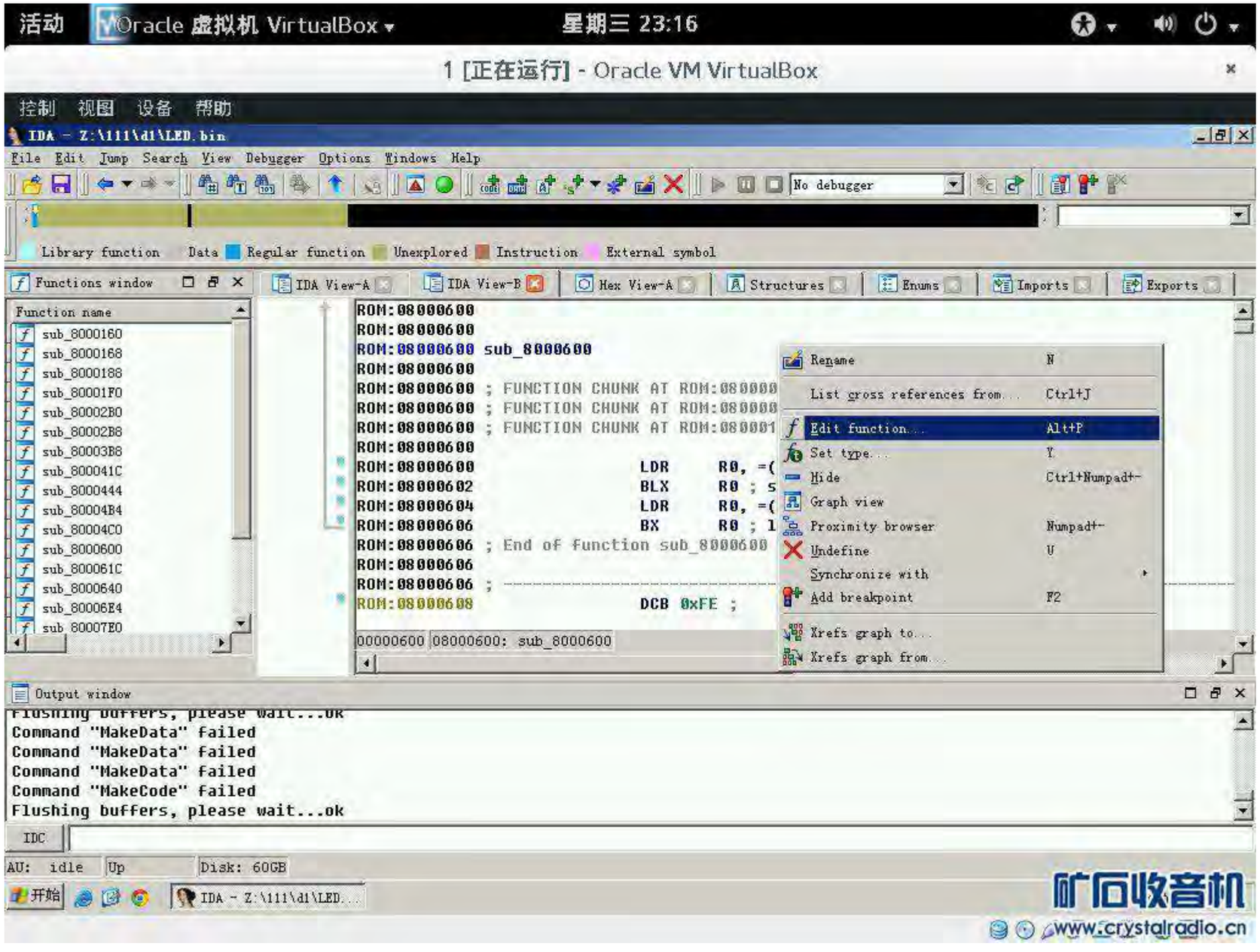
Output window

```
Flushing buffers, please wait...ok
Command "MakeData" failed
Command "MakeData" failed
Command "MakeData" failed
Command "MakeCode" failed
Flushing buffers, please wait...ok
```

IDC

AU: idle | Up | Disk: 60GB

开始 IDA - Z:\111\d1\LED...



反汇编完成，观看程序结构

活动 Oracle 虚拟机 VirtualBox ▾ 星期三 23:17

1 [正在运行] - Oracle VM VirtualBox

控制 视图 设备 帮助

IDA - Z:\111\d1\LED.bin

File Edit Jump Search View Debugger Options Windows Help

No debugger

Library function Data Regular function Unexplored Instruction External symbol

Functions window | IDA View-A | IDA View-C | IDA View-B | Hex View-A | Structures | Enums | Imports

Function name

sub_8000160
sub_8000168
sub_8000188
sub_80001F0
sub_80002B0
sub_80002B8
sub_80003B8
sub_800041C
sub_8000444
sub_80004B4
sub_80004C0
sub_8000600
sub_800061C
sub_8000640
sub_80006E4
sub_80007E0

```
ROM:08000600
ROM:08000600 ; =============== S U B R O U T I N E =======================================
ROM:08000600
ROM:08000600
ROM:08000600 sub_8000600
ROM:08000600
ROM:08000600 ; FUNCT                                          BYTES
ROM:08000600 ; FUNCT                                          BYTES
ROM:08000600 ; FUNCT                                          BYTES
ROM:08000600
ROM:08000600
ROM:08000602
ROM:08000604
ROM:08000606
ROM:08000606 ; End o
ROM:08000606
```

00000600 08000600: sub_

Rename N
Jump to operand Enter
Jump in a new window Alt+Enter
Jump in a new hex window
List cross references to... Ctrl+X
List cross references from... Ctrl+J
Edit function... Alt+P
Set type... Y
Hide Ctrl+Numpad+-
Graph view
Proximity browser Numpad+-
Undefine U
Synchronize with
Add breakpoint F2
Copy address to command line
Xrefs graph to...
Xrefs graph from...

Output window

```
Flushing buffers, please wait...ok
Command "MakeData" failed
Command "MakeData" failed
Command "MakeData" failed
Command "MakeCode" failed
Flushing buffers, please wait...ok
```

IDC

AU: idle Up Disk: 60GB

开始 IDA - Z:\111\d1\LED...

活动 Oracle 虚拟机 VirtualBox ▾ 星期三 23:18

1 [正在运行] - Oracle VM VirtualBox

控制 视图 设备 帮助

IDA - Z:\111\d1\LED.bin

File Edit Jump Search View Debugger Options Windows Help

No debugger

Library function Data Regular function Unexplored Instruction External symbol

Functions window

Function name

sub_8000160
sub_8000168
sub_8000188
sub_80001F0
sub_80002B0
sub_80002B8
sub_80003B8
sub_800041C

Line 1 of 20

Graph overview

IDA View-A IDA View-C IDA View-B Hex View-A Structures Enums Imports

```
sub_8000600

; FUNCTION CHUNK AT 080000EC SIZE 00000004 BYTES
; FUNCTION CHUNK AT 080000F4 SIZE 0000002C BYTES
; FUNCTION CHUNK AT 0800016C SIZE 0000000E BYTES

LDR     R0, =(sub_80002B8+1)
BLX     R0 ; sub_80002B8
LDR     R0, =(loc_80000EC+1)
BX      R0 ; loc_80000EC
; End of function sub_8000600
```

100.00% (-75,31) (604,164) 00000600 08000600: sub_8000600

Output window

```
Flushing buffers, please wait...ok
Command "MakeData" failed
Command "MakeData" failed
Command "MakeData" failed
Command "MakeCode" failed
Flushing buffers, please wait...ok
```

IDC

AU: idle Up Disk: 60GB

开始 IDA - Z:\111\d1\LED...

矿石收音机 www.crystalradio.cn

活动 Oracle 虚拟机 VirtualBox ▾ 星期三 23:19

1 [正在运行] - Oracle VM VirtualBox

控制 视图 设备 帮助

IDA View-C

```
CODE32

sub_8000600

; FUNCTION CHUNK AT 080000EC SIZE 00000000 BYTES
; FUNCTION CHUNK AT 080000FA SIZE 0000002C BYTES
; FUNCTION CHUNK AT 0800016C SIZE 0000000E BYTES

LDR     R0, =(sub_8000208+1)
BLX     R0 ; sub_8000208
LDR     R0, =(loc_80000EC+1)
BX      R0 ; loc_80000EC
; End of function sub_8000600
```

```
; START OF FUNCTION CHUNK FOR sub_8000600
CODE16

loc_80000EC
B1      loc_80000FA
; END OF FUNCTION CHUNK FOR sub_8000600
```

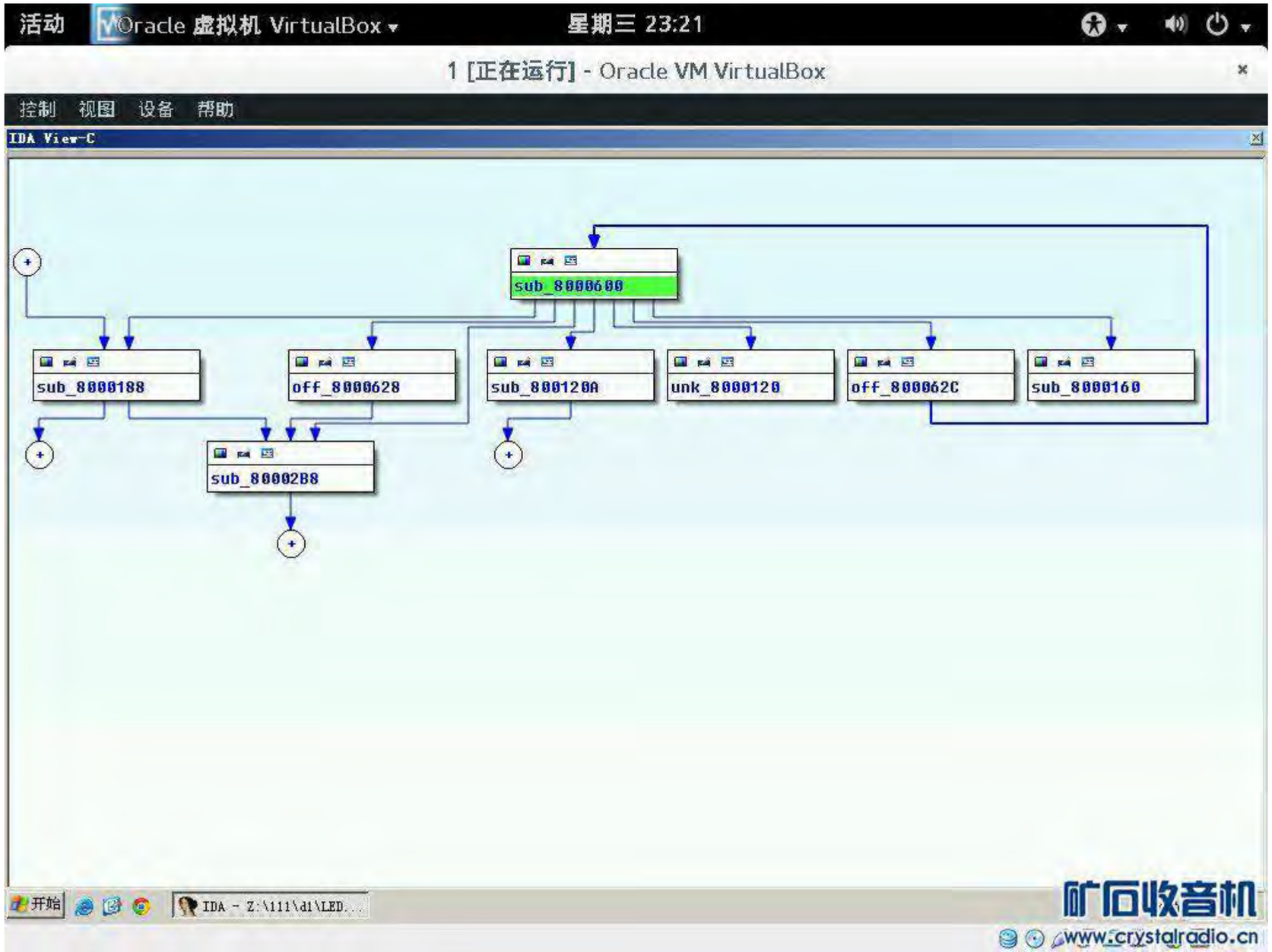
```
; START OF FUNCTION CHUNK FOR sub_8000600

loc_80000FA
ADR     R0, unk_8000120
LDMIA.W R0, {R10,R11}
ADD     R10, R0
ADD     R11, R0
SUB.W   R7, R10, #1

loc_8000102
CMP     R10, R11
BNE     loc_800010A
```

```
BL      loc_800016C
```

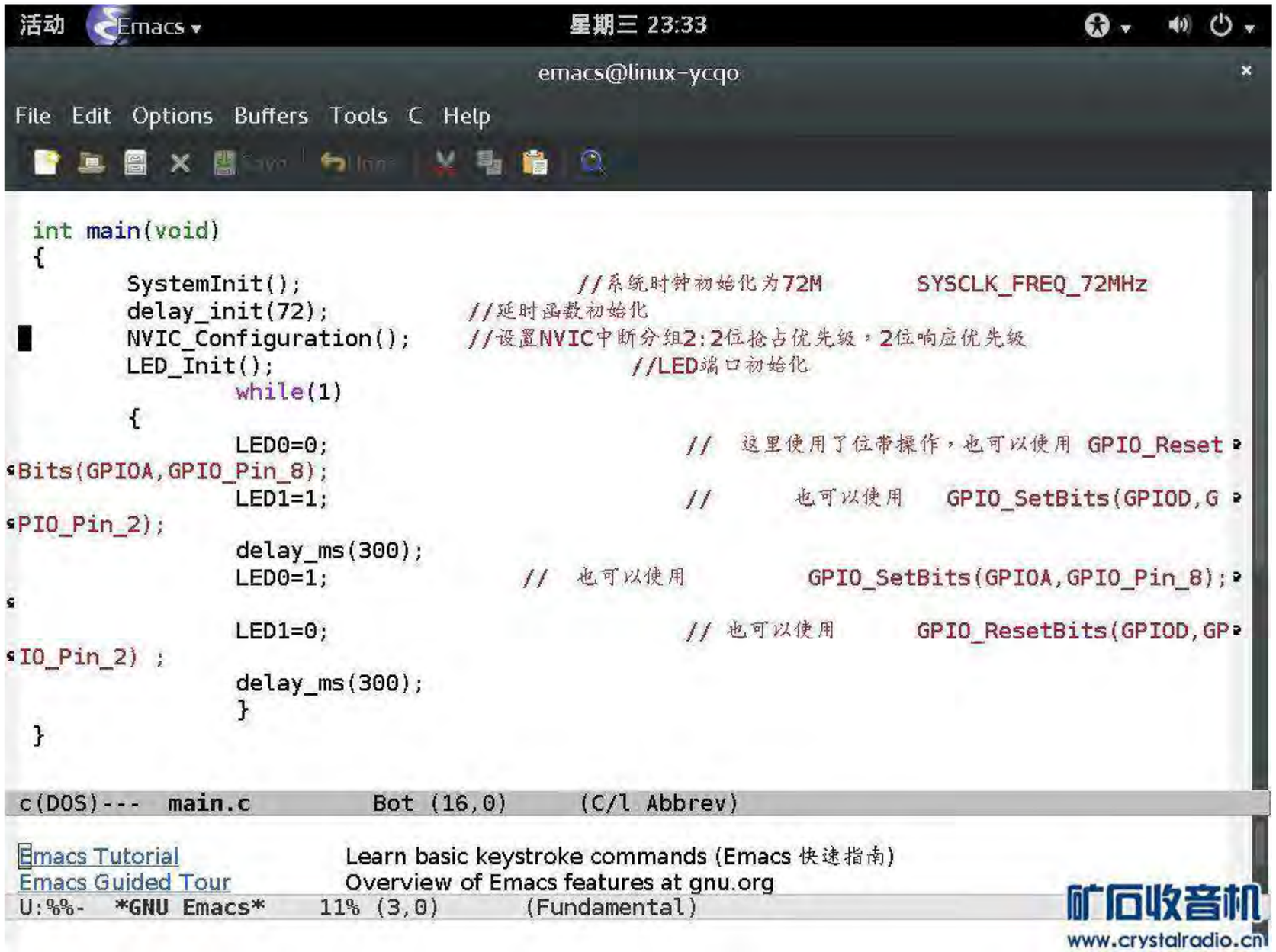
```
loc_800010A
ADR.W   LR, (loc_8000102+1)
LDMIA.W R101, {R0-R3}
TST.W   R3, #1
IT NE
SUBNE   R3, R7, R3
ORR.W   R3, R3, #1
BX      R3
; END OF FUNCTION CHUNK FOR sub_8000600
```

```
; START OF FUNCTION CHUNK FOR sub_8000600

loc_800016C
BL      sub_800120A
MOV     R1, R2
```

开始 IDA - Z:\111\d1\LED...


活动
Oracle 虚拟机 VirtualBox
星期三 23:21
1 [正在运行] - Oracle VM VirtualBox
控制 视图 设备 帮助
IDA View-C
sub_8000600
sub_8000188
off_8000628
sub_800120A
unk_8000120
off_800062C
sub_8000160
sub_80002B8
开始
IDA - Z:\111\d1\LED...
矿石收音机
www.crystalradio.cn

源程序代码



http://www.crystalradio.cn/thread-637028-1-1.html

A 库

IDA 反编译 ARM 静态链接程序

IDA 在反编译 ARM 的静态链接程序时，是没有输入符号表的，系统调用均采用 SVC 形式。如：

```
.text:0000B380 ; =============== S U B R O U T I N E =======================================
.text:0000B380
.text:0000B380
.text:0000B380 sub_B380                                ; CODE XREF: sub_8BB8+12p
.text:0000B380 STMFD      SP!, {R4,R7}
.text:0000B384 MOV         R7, #0x2A
.text:0000B388 SVC         0
.text:0000B38C LDMFD      SP!, {R4,R7}
.text:0000B390 MOVS        R0, R0
.text:0000B394 BXPL        LR
.text:0000B398 B           sub_168DC
.text:0000B398 ; End of function sub_B380
.text:0000B398
.text:0000B398 ; ---------------------------------------------------------------------------
```

此时需要根据 Andorid 源码对应平台的 unistd.h,来识别相应的系统调用.根据自己修改后，反汇编代码如下：

```
.text:0000B380 ; =============== S U B R O U T I N E =======================================
.text:0000B380
.text:0000B380
.text:0000B380 pipe                                    ; CODE XREF: sub_8BB8+12p
.text:0000B380 STMFD      SP!, {R4,R7}
.text:0000B384 MOV         R7, #0x2A
.text:0000B388 SVC         0
.text:0000B38C LDMFD      SP!, {R4,R7}
.text:0000B390 MOVS        R0, R0
.text:0000B394 BXPL        LR
.text:0000B398 B           __set_errno
.text:0000B398 ; End of function pipe
.text:0000B398
.text:0000B398 ; ---------------------------------------------------------------------------
```

arm/unistd.h 文件内容如下：

```
/*
 *    arch/arm/include/asm/unistd.h
```

```
 *
 *    Copyright (C) 2001-2005 Russell King
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 *
 * Please forward _all_ changes to this file to rmk@arm.linux.org.uk,
 * no matter what the change is.    Thanks!
 */
#ifndef __ASM_ARM_UNISTD_H
#define __ASM_ARM_UNISTD_H

#define __NR_OABI_SYSCALL_BASE      0x900000

#if defined(__thumb__) || defined(__ARM_EABI__)
#define __NR_SYSCALL_BASE 0
#else
#define __NR_SYSCALL_BASE __NR_OABI_SYSCALL_BASE
#endif

/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall        (__NR_SYSCALL_BASE+    0)
#define __NR_exit               (__NR_SYSCALL_BASE+    1)
#define __NR_fork               (__NR_SYSCALL_BASE+    2)
#define __NR_read               (__NR_SYSCALL_BASE+    3)
#define __NR_write                  (__NR_SYSCALL_BASE+    4)
#define __NR_open                   (__NR_SYSCALL_BASE+    5)
#define __NR_close                  (__NR_SYSCALL_BASE+    6)
                            /* 7 was sys_waitpid */
#define __NR_creat                  (__NR_SYSCALL_BASE+    8)
#define __NR_link               (__NR_SYSCALL_BASE+    9)
#define __NR_unlink                 (__NR_SYSCALL_BASE+ 10)
#define __NR_execve                 (__NR_SYSCALL_BASE+ 11)
#define __NR_chdir                  (__NR_SYSCALL_BASE+ 12)
#define __NR_time               (__NR_SYSCALL_BASE+ 13)
#define __NR_mknod                  (__NR_SYSCALL_BASE+ 14)
#define __NR_chmod                  (__NR_SYSCALL_BASE+ 15)
#define __NR_lchown                 (__NR_SYSCALL_BASE+ 16)
```

```
                        /* 17 was sys_break */
                        /* 18 was sys_stat */
#define __NR_lseek              (__NR_SYSCALL_BASE+ 19)
#define __NR_getpid             (__NR_SYSCALL_BASE+ 20)
#define __NR_mount              (__NR_SYSCALL_BASE+ 21)
#define __NR_umount             (__NR_SYSCALL_BASE+ 22)
#define __NR_setuid             (__NR_SYSCALL_BASE+ 23)
#define __NR_getuid             (__NR_SYSCALL_BASE+ 24)
#define __NR_stime              (__NR_SYSCALL_BASE+ 25)
#define __NR_ptrace             (__NR_SYSCALL_BASE+ 26)
#define __NR_alarm              (__NR_SYSCALL_BASE+ 27)
                        /* 28 was sys_fstat */
#define __NR_pause              (__NR_SYSCALL_BASE+ 29)
#define __NR_utime              (__NR_SYSCALL_BASE+ 30)
                        /* 31 was sys_stty */
                        /* 32 was sys_gtty */
#define __NR_access             (__NR_SYSCALL_BASE+ 33)
#define __NR_nice           (__NR_SYSCALL_BASE+ 34)
                        /* 35 was sys_ftime */
#define __NR_sync           (__NR_SYSCALL_BASE+ 36)
#define __NR_kill           (__NR_SYSCALL_BASE+ 37)
#define __NR_rename             (__NR_SYSCALL_BASE+ 38)
#define __NR_mkdir              (__NR_SYSCALL_BASE+ 39)
#define __NR_rmdir              (__NR_SYSCALL_BASE+ 40)
#define __NR_dup            (__NR_SYSCALL_BASE+ 41)
#define __NR_pipe           (__NR_SYSCALL_BASE+ 42)
#define __NR_times              (__NR_SYSCALL_BASE+ 43)
                        /* 44 was sys_prof */
#define __NR_brk            (__NR_SYSCALL_BASE+ 45)
#define __NR_setgid             (__NR_SYSCALL_BASE+ 46)
#define __NR_getgid             (__NR_SYSCALL_BASE+ 47)
                        /* 48 was sys_signal */
#define __NR_geteuid            (__NR_SYSCALL_BASE+ 49)
#define __NR_getegid            (__NR_SYSCALL_BASE+ 50)
#define __NR_acct           (__NR_SYSCALL_BASE+ 51)
#define __NR_umount2            (__NR_SYSCALL_BASE+ 52)
                        /* 53 was sys_lock */
#define __NR_ioctl          (__NR_SYSCALL_BASE+ 54)
#define __NR_fcntl          (__NR_SYSCALL_BASE+ 55)
                        /* 56 was sys_mpx */
#define __NR_setpgid            (__NR_SYSCALL_BASE+ 57)
                        /* 58 was sys_ulimit */
```

```
                        /* 59 was sys_olduname */
#define __NR_umask              (__NR_SYSCALL_BASE+ 60)
#define __NR_chroot             (__NR_SYSCALL_BASE+ 61)
#define __NR_ustat              (__NR_SYSCALL_BASE+ 62)
#define __NR_dup2               (__NR_SYSCALL_BASE+ 63)
#define __NR_getppid            (__NR_SYSCALL_BASE+ 64)
#define __NR_getpgrp            (__NR_SYSCALL_BASE+ 65)
#define __NR_setsid             (__NR_SYSCALL_BASE+ 66)
#define __NR_sigaction          (__NR_SYSCALL_BASE+ 67)
                        /* 68 was sys_sgetmask */
                        /* 69 was sys_ssetmask */
#define __NR_setreuid           (__NR_SYSCALL_BASE+ 70)
#define __NR_setregid           (__NR_SYSCALL_BASE+ 71)
#define __NR_sigsuspend             (__NR_SYSCALL_BASE+ 72)
#define __NR_sigpending             (__NR_SYSCALL_BASE+ 73)
#define __NR_sethostname        (__NR_SYSCALL_BASE+ 74)
#define __NR_setrlimit          (__NR_SYSCALL_BASE+ 75)
#define __NR_getrlimit          (__NR_SYSCALL_BASE+ 76)    /* Back compat 2GB limited rlimit */
#define __NR_getrusage          (__NR_SYSCALL_BASE+ 77)
#define __NR_gettimeofday       (__NR_SYSCALL_BASE+ 78)
#define __NR_settimeofday       (__NR_SYSCALL_BASE+ 79)
#define __NR_getgroups              (__NR_SYSCALL_BASE+ 80)
#define __NR_setgroups          (__NR_SYSCALL_BASE+ 81)
#define __NR_select             (__NR_SYSCALL_BASE+ 82)
#define __NR_symlink            (__NR_SYSCALL_BASE+ 83)
                        /* 84 was sys_lstat */
#define __NR_readlink           (__NR_SYSCALL_BASE+ 85)
#define __NR_uselib             (__NR_SYSCALL_BASE+ 86)
#define __NR_swapon             (__NR_SYSCALL_BASE+ 87)
#define __NR_reboot             (__NR_SYSCALL_BASE+ 88)
#define __NR_readdir            (__NR_SYSCALL_BASE+ 89)
#define __NR_mmap               (__NR_SYSCALL_BASE+ 90)
#define __NR_munmap             (__NR_SYSCALL_BASE+ 91)
#define __NR_truncate           (__NR_SYSCALL_BASE+ 92)
#define __NR_ftruncate          (__NR_SYSCALL_BASE+ 93)
#define __NR_fchmod             (__NR_SYSCALL_BASE+ 94)
#define __NR_fchown             (__NR_SYSCALL_BASE+ 95)
#define __NR_getpriority        (__NR_SYSCALL_BASE+ 96)
#define __NR_setpriority        (__NR_SYSCALL_BASE+ 97)
                        /* 98 was sys_profil */
#define __NR_statfs             (__NR_SYSCALL_BASE+ 99)
#define __NR_fstatfs            (__NR_SYSCALL_BASE+100)
```

```
                        /* 101 was sys_ioperm */
#define __NR_socketcall             (__NR_SYSCALL_BASE+102)
#define __NR_syslog                 (__NR_SYSCALL_BASE+103)
#define __NR_setitimer              (__NR_SYSCALL_BASE+104)
#define __NR_getitimer              (__NR_SYSCALL_BASE+105)
#define __NR_stat               (__NR_SYSCALL_BASE+106)
#define __NR_lstat              (__NR_SYSCALL_BASE+107)
#define __NR_fstat              (__NR_SYSCALL_BASE+108)
                        /* 109 was sys_uname */
                        /* 110 was sys_iopl */
#define __NR_vhangup                (__NR_SYSCALL_BASE+111)
                        /* 112 was sys_idle */
#define __NR_syscall                (__NR_SYSCALL_BASE+113) /* syscall to call a syscall! */
#define __NR_wait4                  (__NR_SYSCALL_BASE+114)
#define __NR_swapoff                (__NR_SYSCALL_BASE+115)
#define __NR_sysinfo                (__NR_SYSCALL_BASE+116)
#define __NR_ipc                (__NR_SYSCALL_BASE+117)
#define __NR_fsync                  (__NR_SYSCALL_BASE+118)
#define __NR_sigreturn              (__NR_SYSCALL_BASE+119)
#define __NR_clone                  (__NR_SYSCALL_BASE+120)
#define __NR_setdomainname          (__NR_SYSCALL_BASE+121)
#define __NR_uname                  (__NR_SYSCALL_BASE+122)
                        /* 123 was sys_modify_ldt */
#define __NR_adjtimex               (__NR_SYSCALL_BASE+124)
#define __NR_mprotect               (__NR_SYSCALL_BASE+125)
#define __NR_sigprocmask            (__NR_SYSCALL_BASE+126)
                        /* 127 was sys_create_module */
#define __NR_init_module            (__NR_SYSCALL_BASE+128)
#define __NR_delete_module              (__NR_SYSCALL_BASE+129)
                        /* 130 was sys_get_kernel_syms */
#define __NR_quotactl               (__NR_SYSCALL_BASE+131)
#define __NR_getpgid                (__NR_SYSCALL_BASE+132)
#define __NR_fchdir                 (__NR_SYSCALL_BASE+133)
#define __NR_bdflush                (__NR_SYSCALL_BASE+134)
#define __NR_sysfs              (__NR_SYSCALL_BASE+135)
#define __NR_personality            (__NR_SYSCALL_BASE+136)
                        /* 137 was sys_afs_syscall */
#define __NR_setfsuid               (__NR_SYSCALL_BASE+138)
#define __NR_setfsgid               (__NR_SYSCALL_BASE+139)
#define __NR__llseek                (__NR_SYSCALL_BASE+140)
#define __NR_getdents               (__NR_SYSCALL_BASE+141)
#define __NR__newselect                 (__NR_SYSCALL_BASE+142)
```

```
#define __NR_flock              (__NR_SYSCALL_BASE+143)
#define __NR_msync              (__NR_SYSCALL_BASE+144)
#define __NR_readv              (__NR_SYSCALL_BASE+145)
#define __NR_writev             (__NR_SYSCALL_BASE+146)
#define __NR_getsid             (__NR_SYSCALL_BASE+147)
#define __NR_fdatasync          (__NR_SYSCALL_BASE+148)
#define __NR__sysctl            (__NR_SYSCALL_BASE+149)
#define __NR_mlock              (__NR_SYSCALL_BASE+150)
#define __NR_munlock            (__NR_SYSCALL_BASE+151)
#define __NR_mlockall           (__NR_SYSCALL_BASE+152)
#define __NR_munlockall             (__NR_SYSCALL_BASE+153)
#define __NR_sched_setparam         (__NR_SYSCALL_BASE+154)
#define __NR_sched_getparam         (__NR_SYSCALL_BASE+155)
#define __NR_sched_setscheduler             (__NR_SYSCALL_BASE+156)
#define __NR_sched_getscheduler             (__NR_SYSCALL_BASE+157)
#define __NR_sched_yield        (__NR_SYSCALL_BASE+158)
#define __NR_sched_get_priority_max (__NR_SYSCALL_BASE+159)
#define __NR_sched_get_priority_min (__NR_SYSCALL_BASE+160)
#define __NR_sched_rr_get_interval  (__NR_SYSCALL_BASE+161)
#define __NR_nanosleep              (__NR_SYSCALL_BASE+162)
#define __NR_mremap             (__NR_SYSCALL_BASE+163)
#define __NR_setresuid          (__NR_SYSCALL_BASE+164)
#define __NR_getresuid          (__NR_SYSCALL_BASE+165)
                        /* 166 was sys_vm86 */
                        /* 167 was sys_query_module */
#define __NR_poll           (__NR_SYSCALL_BASE+168)
#define __NR_nfsservctl             (__NR_SYSCALL_BASE+169)
#define __NR_setresgid          (__NR_SYSCALL_BASE+170)
#define __NR_getresgid          (__NR_SYSCALL_BASE+171)
#define __NR_prctl          (__NR_SYSCALL_BASE+172)
#define __NR_rt_sigreturn       (__NR_SYSCALL_BASE+173)
#define __NR_rt_sigaction       (__NR_SYSCALL_BASE+174)
#define __NR_rt_sigprocmask             (__NR_SYSCALL_BASE+175)
#define __NR_rt_sigpending      (__NR_SYSCALL_BASE+176)
#define __NR_rt_sigtimedwait            (__NR_SYSCALL_BASE+177)
#define __NR_rt_sigqueueinfo            (__NR_SYSCALL_BASE+178)
#define __NR_rt_sigsuspend      (__NR_SYSCALL_BASE+179)
#define __NR_pread64            (__NR_SYSCALL_BASE+180)
#define __NR_pwrite64           (__NR_SYSCALL_BASE+181)
#define __NR_chown              (__NR_SYSCALL_BASE+182)
#define __NR_getcwd             (__NR_SYSCALL_BASE+183)
#define __NR_capget             (__NR_SYSCALL_BASE+184)
```

```
#define __NR_capset             (__NR_SYSCALL_BASE+185)
#define __NR_sigaltstack        (__NR_SYSCALL_BASE+186)
#define __NR_sendfile           (__NR_SYSCALL_BASE+187)
                                /* 188 reserved */
                                /* 189 reserved */
#define __NR_vfork              (__NR_SYSCALL_BASE+190)
#define __NR_ugetrlimit         (__NR_SYSCALL_BASE+191)   /* SuS compliant getrlimit */
#define __NR_mmap2              (__NR_SYSCALL_BASE+192)
#define __NR_truncate64             (__NR_SYSCALL_BASE+193)
#define __NR_ftruncate64        (__NR_SYSCALL_BASE+194)
#define __NR_stat64             (__NR_SYSCALL_BASE+195)
#define __NR_lstat64            (__NR_SYSCALL_BASE+196)
#define __NR_fstat64            (__NR_SYSCALL_BASE+197)
#define __NR_lchown32           (__NR_SYSCALL_BASE+198)
#define __NR_getuid32           (__NR_SYSCALL_BASE+199)
#define __NR_getgid32           (__NR_SYSCALL_BASE+200)
#define __NR_geteuid32              (__NR_SYSCALL_BASE+201)
#define __NR_getegid32          (__NR_SYSCALL_BASE+202)
#define __NR_setreuid32             (__NR_SYSCALL_BASE+203)
#define __NR_setregid32             (__NR_SYSCALL_BASE+204)
#define __NR_getgroups32        (__NR_SYSCALL_BASE+205)
#define __NR_setgroups32        (__NR_SYSCALL_BASE+206)
#define __NR_fchown32           (__NR_SYSCALL_BASE+207)
#define __NR_setresuid32        (__NR_SYSCALL_BASE+208)
#define __NR_getresuid32        (__NR_SYSCALL_BASE+209)
#define __NR_setresgid32        (__NR_SYSCALL_BASE+210)
#define __NR_getresgid32        (__NR_SYSCALL_BASE+211)
#define __NR_chown32            (__NR_SYSCALL_BASE+212)
#define __NR_setuid32           (__NR_SYSCALL_BASE+213)
#define __NR_setgid32           (__NR_SYSCALL_BASE+214)
#define __NR_setfsuid32             (__NR_SYSCALL_BASE+215)
#define __NR_setfsgid32             (__NR_SYSCALL_BASE+216)
#define __NR_getdents64             (__NR_SYSCALL_BASE+217)
#define __NR_pivot_root             (__NR_SYSCALL_BASE+218)
#define __NR_mincore            (__NR_SYSCALL_BASE+219)
#define __NR_madvise            (__NR_SYSCALL_BASE+220)
#define __NR_fcntl64            (__NR_SYSCALL_BASE+221)
                                /* 222 for tux */
                                /* 223 is unused */
#define __NR_gettid             (__NR_SYSCALL_BASE+224)
#define __NR_readahead              (__NR_SYSCALL_BASE+225)
#define __NR_setxattr           (__NR_SYSCALL_BASE+226)
```

```
#define __NR_lsetxattr          (__NR_SYSCALL_BASE+227)
#define __NR_fsetxattr          (__NR_SYSCALL_BASE+228)
#define __NR_getxattr           (__NR_SYSCALL_BASE+229)
#define __NR_lgetxattr          (__NR_SYSCALL_BASE+230)
#define __NR_fgetxattr          (__NR_SYSCALL_BASE+231)
#define __NR_listxattr          (__NR_SYSCALL_BASE+232)
#define __NR_llistxattr         (__NR_SYSCALL_BASE+233)
#define __NR_flistxattr         (__NR_SYSCALL_BASE+234)
#define __NR_removexattr        (__NR_SYSCALL_BASE+235)
#define __NR_lremovexattr       (__NR_SYSCALL_BASE+236)
#define __NR_fremovexattr       (__NR_SYSCALL_BASE+237)
#define __NR_tkill              (__NR_SYSCALL_BASE+238)
#define __NR_sendfile64             (__NR_SYSCALL_BASE+239)
#define __NR_futex              (__NR_SYSCALL_BASE+240)
#define __NR_sched_setaffinity      (__NR_SYSCALL_BASE+241)
#define __NR_sched_getaffinity      (__NR_SYSCALL_BASE+242)
#define __NR_io_setup           (__NR_SYSCALL_BASE+243)
#define __NR_io_destroy             (__NR_SYSCALL_BASE+244)
#define __NR_io_getevents       (__NR_SYSCALL_BASE+245)
#define __NR_io_submit              (__NR_SYSCALL_BASE+246)
#define __NR_io_cancel          (__NR_SYSCALL_BASE+247)
#define __NR_exit_group             (__NR_SYSCALL_BASE+248)
#define __NR_lookup_dcookie         (__NR_SYSCALL_BASE+249)
#define __NR_epoll_create       (__NR_SYSCALL_BASE+250)
#define __NR_epoll_ctl          (__NR_SYSCALL_BASE+251)
#define __NR_epoll_wait             (__NR_SYSCALL_BASE+252)
#define __NR_remap_file_pages       (__NR_SYSCALL_BASE+253)
                        /* 254 for set_thread_area */
                        /* 255 for get_thread_area */
#define __NR_set_tid_address        (__NR_SYSCALL_BASE+256)
#define __NR_timer_create       (__NR_SYSCALL_BASE+257)
#define __NR_timer_settime      (__NR_SYSCALL_BASE+258)
#define __NR_timer_gettime      (__NR_SYSCALL_BASE+259)
#define __NR_timer_getoverrun       (__NR_SYSCALL_BASE+260)
#define __NR_timer_delete       (__NR_SYSCALL_BASE+261)
#define __NR_clock_settime      (__NR_SYSCALL_BASE+262)
#define __NR_clock_gettime      (__NR_SYSCALL_BASE+263)
#define __NR_clock_getres       (__NR_SYSCALL_BASE+264)
#define __NR_clock_nanosleep        (__NR_SYSCALL_BASE+265)
#define __NR_statfs64           (__NR_SYSCALL_BASE+266)
#define __NR_fstatfs64          (__NR_SYSCALL_BASE+267)
#define __NR_tgkill             (__NR_SYSCALL_BASE+268)
```

```
#define __NR_utimes                 (__NR_SYSCALL_BASE+269)
#define __NR_arm_fadvise64_64       (__NR_SYSCALL_BASE+270)
#define __NR_pciconfig_iobase       (__NR_SYSCALL_BASE+271)
#define __NR_pciconfig_read         (__NR_SYSCALL_BASE+272)
#define __NR_pciconfig_write        (__NR_SYSCALL_BASE+273)
#define __NR_mq_open                (__NR_SYSCALL_BASE+274)
#define __NR_mq_unlink              (__NR_SYSCALL_BASE+275)
#define __NR_mq_timedsend           (__NR_SYSCALL_BASE+276)
#define __NR_mq_timedreceive        (__NR_SYSCALL_BASE+277)
#define __NR_mq_notify              (__NR_SYSCALL_BASE+278)
#define __NR_mq_getsetattr          (__NR_SYSCALL_BASE+279)
#define __NR_waitid                 (__NR_SYSCALL_BASE+280)
#define __NR_socket                 (__NR_SYSCALL_BASE+281)
#define __NR_bind               (__NR_SYSCALL_BASE+282)
#define __NR_connect                (__NR_SYSCALL_BASE+283)
#define __NR_listen                 (__NR_SYSCALL_BASE+284)
#define __NR_accept                 (__NR_SYSCALL_BASE+285)
#define __NR_getsockname            (__NR_SYSCALL_BASE+286)
#define __NR_getpeername            (__NR_SYSCALL_BASE+287)
#define __NR_socketpair             (__NR_SYSCALL_BASE+288)
#define __NR_send               (__NR_SYSCALL_BASE+289)
#define __NR_sendto                 (__NR_SYSCALL_BASE+290)
#define __NR_recv               (__NR_SYSCALL_BASE+291)
#define __NR_recvfrom               (__NR_SYSCALL_BASE+292)
#define __NR_shutdown               (__NR_SYSCALL_BASE+293)
#define __NR_setsockopt             (__NR_SYSCALL_BASE+294)
#define __NR_getsockopt             (__NR_SYSCALL_BASE+295)
#define __NR_sendmsg                (__NR_SYSCALL_BASE+296)
#define __NR_recvmsg                (__NR_SYSCALL_BASE+297)
#define __NR_semop                  (__NR_SYSCALL_BASE+298)
#define __NR_semget                 (__NR_SYSCALL_BASE+299)
#define __NR_semctl                 (__NR_SYSCALL_BASE+300)
#define __NR_msgsnd                 (__NR_SYSCALL_BASE+301)
#define __NR_msgrcv                 (__NR_SYSCALL_BASE+302)
#define __NR_msgget                 (__NR_SYSCALL_BASE+303)
#define __NR_msgctl                 (__NR_SYSCALL_BASE+304)
#define __NR_shmat                  (__NR_SYSCALL_BASE+305)
#define __NR_shmdt                  (__NR_SYSCALL_BASE+306)
#define __NR_shmget                 (__NR_SYSCALL_BASE+307)
#define __NR_shmctl                 (__NR_SYSCALL_BASE+308)
#define __NR_add_key                (__NR_SYSCALL_BASE+309)
#define __NR_request_key            (__NR_SYSCALL_BASE+310)
```

```
#define __NR_keyctl                (__NR_SYSCALL_BASE+311)
#define __NR_semtimedop                (__NR_SYSCALL_BASE+312)
#define __NR_vserver               (__NR_SYSCALL_BASE+313)
#define __NR_ioprio_set                (__NR_SYSCALL_BASE+314)
#define __NR_ioprio_get                (__NR_SYSCALL_BASE+315)
#define __NR_inotify_init          (__NR_SYSCALL_BASE+316)
#define __NR_inotify_add_watch         (__NR_SYSCALL_BASE+317)
#define __NR_inotify_rm_watch          (__NR_SYSCALL_BASE+318)
#define __NR_mbind                 (__NR_SYSCALL_BASE+319)
#define __NR_get_mempolicy             (__NR_SYSCALL_BASE+320)
#define __NR_set_mempolicy             (__NR_SYSCALL_BASE+321)
#define __NR_openat                (__NR_SYSCALL_BASE+322)
#define __NR_mkdirat               (__NR_SYSCALL_BASE+323)
#define __NR_mknodat               (__NR_SYSCALL_BASE+324)
#define __NR_fchownat              (__NR_SYSCALL_BASE+325)
#define __NR_futimesat             (__NR_SYSCALL_BASE+326)
#define __NR_fstatat64             (__NR_SYSCALL_BASE+327)
#define __NR_unlinkat              (__NR_SYSCALL_BASE+328)
#define __NR_renameat              (__NR_SYSCALL_BASE+329)
#define __NR_linkat                (__NR_SYSCALL_BASE+330)
#define __NR_symlinkat             (__NR_SYSCALL_BASE+331)
#define __NR_readlinkat                (__NR_SYSCALL_BASE+332)
#define __NR_fchmodat              (__NR_SYSCALL_BASE+333)
#define __NR_faccessat             (__NR_SYSCALL_BASE+334)
#define __NR_pselect6              (__NR_SYSCALL_BASE+335)
#define __NR_ppoll                 (__NR_SYSCALL_BASE+336)
#define __NR_unshare               (__NR_SYSCALL_BASE+337)
#define __NR_set_robust_list       (__NR_SYSCALL_BASE+338)
#define __NR_get_robust_list           (__NR_SYSCALL_BASE+339)
#define __NR_splice                (__NR_SYSCALL_BASE+340)
#define __NR_arm_sync_file_range       (__NR_SYSCALL_BASE+341)
#define __NR_sync_file_range2          __NR_arm_sync_file_range
#define __NR_tee               (__NR_SYSCALL_BASE+342)
#define __NR_vmsplice              (__NR_SYSCALL_BASE+343)
#define __NR_move_pages                (__NR_SYSCALL_BASE+344)
#define __NR_getcpu                (__NR_SYSCALL_BASE+345)
#define __NR_epoll_pwait           (__NR_SYSCALL_BASE+346)
#define __NR_kexec_load                (__NR_SYSCALL_BASE+347)
#define __NR_utimensat                 (__NR_SYSCALL_BASE+348)
#define __NR_signalfd              (__NR_SYSCALL_BASE+349)
#define __NR_timerfd_create            (__NR_SYSCALL_BASE+350)
#define __NR_eventfd               (__NR_SYSCALL_BASE+351)
```

```
#define __NR_fallocate              (__NR_SYSCALL_BASE+352)
#define __NR_timerfd_settime            (__NR_SYSCALL_BASE+353)
#define __NR_timerfd_gettime            (__NR_SYSCALL_BASE+354)
#define __NR_signalfd4              (__NR_SYSCALL_BASE+355)
#define __NR_eventfd2               (__NR_SYSCALL_BASE+356)
#define __NR_epoll_create1          (__NR_SYSCALL_BASE+357)
#define __NR_dup3                   (__NR_SYSCALL_BASE+358)
#define __NR_pipe2                  (__NR_SYSCALL_BASE+359)
#define __NR_inotify_init1          (__NR_SYSCALL_BASE+360)
#define __NR_preadv                 (__NR_SYSCALL_BASE+361)
#define __NR_pwritev                (__NR_SYSCALL_BASE+362)
#define __NR_rt_tgsigqueueinfo          (__NR_SYSCALL_BASE+363)
#define __NR_perf_event_open            (__NR_SYSCALL_BASE+364)
#define __NR_recvmmsg                   (__NR_SYSCALL_BASE+365)
#define __NR_accept4                (__NR_SYSCALL_BASE+366)
#define __NR_fanotify_init          (__NR_SYSCALL_BASE+367)
#define __NR_fanotify_mark          (__NR_SYSCALL_BASE+368)
#define __NR_prlimit64              (__NR_SYSCALL_BASE+369)
#define __NR_name_to_handle_at              (__NR_SYSCALL_BASE+370)
#define __NR_open_by_handle_at              (__NR_SYSCALL_BASE+371)
#define __NR_clock_adjtime          (__NR_SYSCALL_BASE+372)
#define __NR_syncfs                 (__NR_SYSCALL_BASE+373)
#define __NR_sendmmsg                   (__NR_SYSCALL_BASE+374)
#define __NR_setns                  (__NR_SYSCALL_BASE+375)
#define __NR_process_vm_readv           (__NR_SYSCALL_BASE+376)
#define __NR_process_vm_writev          (__NR_SYSCALL_BASE+377)

/*
 * The following SWIs are ARM private.
 */
#define __ARM_NR_BASE                   (__NR_SYSCALL_BASE+0x0f0000)
#define __ARM_NR_breakpoint             (__ARM_NR_BASE+1)
#define __ARM_NR_cacheflush             (__ARM_NR_BASE+2)
#define __ARM_NR_usr26                  (__ARM_NR_BASE+3)
#define __ARM_NR_usr32                  (__ARM_NR_BASE+4)
#define __ARM_NR_set_tls            (__ARM_NR_BASE+5)

/*
 * *NOTE*: This is a ghost syscall private to the kernel.    Only the
 * __kuser_cmpxchg code in entry-armv.S should be aware of its
 * existence.    Don't ever use this from user code.
 */
```

```
#ifdef __KERNEL__
#define __ARM_NR_cmpxchg		(__ARM_NR_BASE+0x00fff0)
#endif

/*
 * The following syscalls are obsolete and no longer available for EABI.
 */
#if !defined(__KERNEL__)
#if defined(__ARM_EABI__)
#undef __NR_time
#undef __NR_umount
#undef __NR_stime
#undef __NR_alarm
#undef __NR_utime
#undef __NR_getrlimit
#undef __NR_select
#undef __NR_readdir
#undef __NR_mmap
#undef __NR_socketcall
#undef __NR_syscall
#undef __NR_ipc
#endif
#endif

#ifdef __KERNEL__

#define __ARCH_WANT_IPC_PARSE_VERSION
#define __ARCH_WANT_STAT64
#define __ARCH_WANT_SYS_GETHOSTNAME
#define __ARCH_WANT_SYS_PAUSE
#define __ARCH_WANT_SYS_GETPGRP
#define __ARCH_WANT_SYS_LLSEEK
#define __ARCH_WANT_SYS_NICE
#define __ARCH_WANT_SYS_SIGPENDING
#define __ARCH_WANT_SYS_SIGPROCMASK
#define __ARCH_WANT_SYS_RT_SIGACTION
#define __ARCH_WANT_SYS_RT_SIGSUSPEND
#define __ARCH_WANT_SYS_OLD_MMAP
#define __ARCH_WANT_SYS_OLD_SELECT

#if !defined(CONFIG_AEABI) || defined(CONFIG_OABI_COMPAT)
#define __ARCH_WANT_SYS_TIME
```

```
#define __ARCH_WANT_SYS_IPC
#define __ARCH_WANT_SYS_OLDUMOUNT
#define __ARCH_WANT_SYS_ALARM
#define __ARCH_WANT_SYS_UTIME
#define __ARCH_WANT_SYS_OLD_GETRLIMIT
#define __ARCH_WANT_OLD_READDIR
#define __ARCH_WANT_SYS_SOCKETCALL
#endif

/*
 * "Conditional" syscalls
 *
 * What we want is __attribute__((weak,alias("sys_ni_syscall"))),
 * but it doesn't work on all toolchains, so we just do it by hand
 */
#define cond_syscall(x) asm(".weak\t" #x "\n\t.set\t" #x ",sys_ni_syscall")

/*
 * Unimplemented (or alternatively implemented) syscalls
 */
#define __IGNORE_fadvise64_64
#define __IGNORE_migrate_pages

#endif /* __KERNEL__ */
#endif /* __ASM_ARM_UNISTD_H */
```

通过 IDA 菜单 search->sequence of bytes...
搜索 00 00 00 ef
就能让到所有 SVC 语句。

http://blog.csdn.net/zhangmiaoping23/article/details/39672373

Windows

汇编语言里 eax, ebx, ecx, edx, esi, edi, ebp, esp 这些都是什么意思啊？

eax, ebx, ecx, edx, esi, edi, ebp, esp 等都是 X86 汇编语言中 CPU 上的通用寄存器的名称，是 32 位的寄存器。如果用 C 语言来解释，可以把这些寄存器当作变量看待。

比方说： add eax,-2 ; //可以认为是给变量 eax 加上-2 这样的一个值。
这些 32 位寄存器有多种用途，但每一个都有“专长”，有各自的特别之处。

EAX　　是"累加器"(accumulator)，它是很多加法乘法指令的缺省寄存器。
EBX　　是"基地址"(base)寄存器，在内存寻址时存放基地址。
ECX　　是计数器(counter)，是重复(REP)前缀指令和 LOOP 指令的内定计数器。
EDX　　则总是被用来放整数除法产生的余数。
ESI/EDI　分别叫做"源/目标索引寄存器"(source/destination index),因为在很多字符串操作指令中，DS:ESI 指向源串,而 ES:EDI 指向目标串.
EBP　是"基址指针"(BASE POINTER)，它最经常被用作高级语言函数调用的"框架指针"(frame pointer). 在破解的时候,经常可以看见一个标准的函数起始代码:

```
push ebp ;  保存当前 ebp
mov ebp,esp ;  EBP 设为当前堆栈指针
sub esp, xxx ;  预留 xxx 字节给函数临时变量.
...
```

这样一来,EBP 构成了该函数的一个框架，在 EBP 上方分别是原来的 EBP，返回地址和参数. EBP 下方则是临时变量. 函数返回时作　mov esp,ebp/pop ebp/ret　　即可.
ESP　　专门用作堆栈指针，被形象地称为栈顶指针，堆栈的顶部是地址小的区域，压入堆栈的数据越多，ESP 也就越来越小。在 32 位平台上，ESP 每次减少 4 字节。
http://blog.sina.com.cn/s/blog_45e2b66c0101bsfb.html

汇编中的 call 和 ret

反汇编经常看到的 CALL 指令的基本格式如下：
CALL　　地址 1
功能：调用地址 1 处的子程序

CALL 指令分为两种情况，一种是段内转移；另一种是段间转移。（这里的“段”指 PE 文件的.text，.data 映射在内存中以虚拟地址的方式呈现）

在 CALL 指令进行的是段内转移的情况时，跟在 CALL 后面的地址 1 为一个相对位移；而 CALL 指令进行的是段间转移的情况时，跟在 CALL 后面的地址 1 为一个绝对内存地址。

段内转移的 CALL 指令等价于两条指令：

```
push eip
jmp　　目的位置
```

RETN 指令用于从段内转移 CALL 进的子程序中返回：

```
RETN　　操作数
```

（等价于一条指令：POP　　eip，然后执行 ESP=ESP+操作数）

段间转移的 CALL 指令等价于三条指令：

```
    push CS
    push eip
    jmp     目的位置
```

RETF 指令用于从段间转移 CALL 进的子程序中返回：

```
    RETF     操作数

  （等价于两条指令：POP     eip
                    POP     CS ，然后执行 ESP=ESP+操作数  ）
```

http://www.cnblogs.com/mo-cuishle/p/3410975.html

IDA 反汇编深度总结

1,[eax]的歧义（其中 eax 指向 SourceString）：到底是*SourceString 还是 SourceString 所处的结构的第一个偏 移的结构。这个应该根据语境来，比如[eax]赋给的值的结构和第一偏移结构匹配，就是后面一种可能；反之就是第一种可能。

2，IDA 翻译的代码不一定是完全正确的，比如入口函数它会翻译为 DriverEntry(int,PUNICODE_STRING SourceString)。所以翻译的时候还是要坚持原则，能肯定的就不一定非要完全照抄 IDA 的结果。

3，PUNICODE_STRING 和 UNICODE_STRING 的选择：如果出现 DiskperfRegistryPath.Length 或 DiskperfRegistry.Buffer 则必定是 UNICODE_STRING.首先我们知道指针哪有长度和缓存呢。我们可以根据这样去记忆。其 次，这明显就是 UNICODE_STRING 的结构，可以查到。

4，局部变量和全局变量：关于一个新的变量，我们到底怎么去判断是前者还是后者呢。如果在函数中显视出现，则判定它为全局变量。如果是局部变量，则会根据偏移量制造出来。

5.关于头文件，我们可以在全部翻译完成读 C 代码的时候查看函数的 DDK。看他们都定义在哪些头文件里面。

6，函数后面的 eax。我们都知道，函数执行之后返回值放在 eax 寄存器里面。有时候不要脑子死掉了，看到 eax 就去找前面显视放进去的数据。不要犯这种低级的错误。

7，offset：它也是地址的标志，不要只记住 lea 而忘了这个 offset。

8，在 if 语句里面，如果 IDA 直译是 if(a=0)那么我们应该翻译成 if(a!=0)。

9，循环语句：我们都知道循环语句的初始化是在大方框的上面那一行代码。我们如果不太确定这行代码是不是初始化，可以先把他直译出来，不放到循环体里面。等到翻译出了判断式的条件语句，再回过头看下这行代码是不是该循环体的初始化语句。

10，PDRIVER_DISPATCH 定义为：PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION+1] 。

关 于 MajorFunction 我们要注意两个方面：首先，MajorFunction 数组是指针形式的，如果 DriverObject->MajorFunction 赋值给某一个变量，那么我们应该定义成 PDRIVER_DISPATCH *变量名的形式。其次，MajorFunction 是四个字节的，也就是说 MajorFunction[a]到
MajorFunction[a+1]其实增加了四个字节。

11，+4 真的就是+4 吗？？？这个问题其实接着上面的问题。我们应该对数据的大小有个了解，当碰到
ULONG 时+1 就是+1，但是当碰到上面的数据结构的时候，+4 就不是+4 了。

12,在 DriverEntry 中 0 就是 STATUS_SUCCESS。

13,另类函数调用：一般的函数调用是通过把参数压栈；现在的函数调用是通过 ecx 和 edx 调用，其中第一个参数是 ecx,第二个参数是 edx，其他的参数还是通过压栈实现。

14,数据结构 1 注释：IRP+60h 为=Irp->Tail.Overlay.CurrentStackLocation。那么它赋予的变量的结构是
PIO_STACK_LOCATION。[IRP+60h]-24h 就是 IoGetNextIrpStackLocation(Irp)，当然了，这个值被赋予的值还是 PIO_STACK_LOCATION 类型的。

15,数据结构 2 注释：DeviceExtension+24h 的结构是：DeviceExtension->DiskCounters。
DeviceExtension+34h 的结构是：DeviceExtension->CountersEnabled。

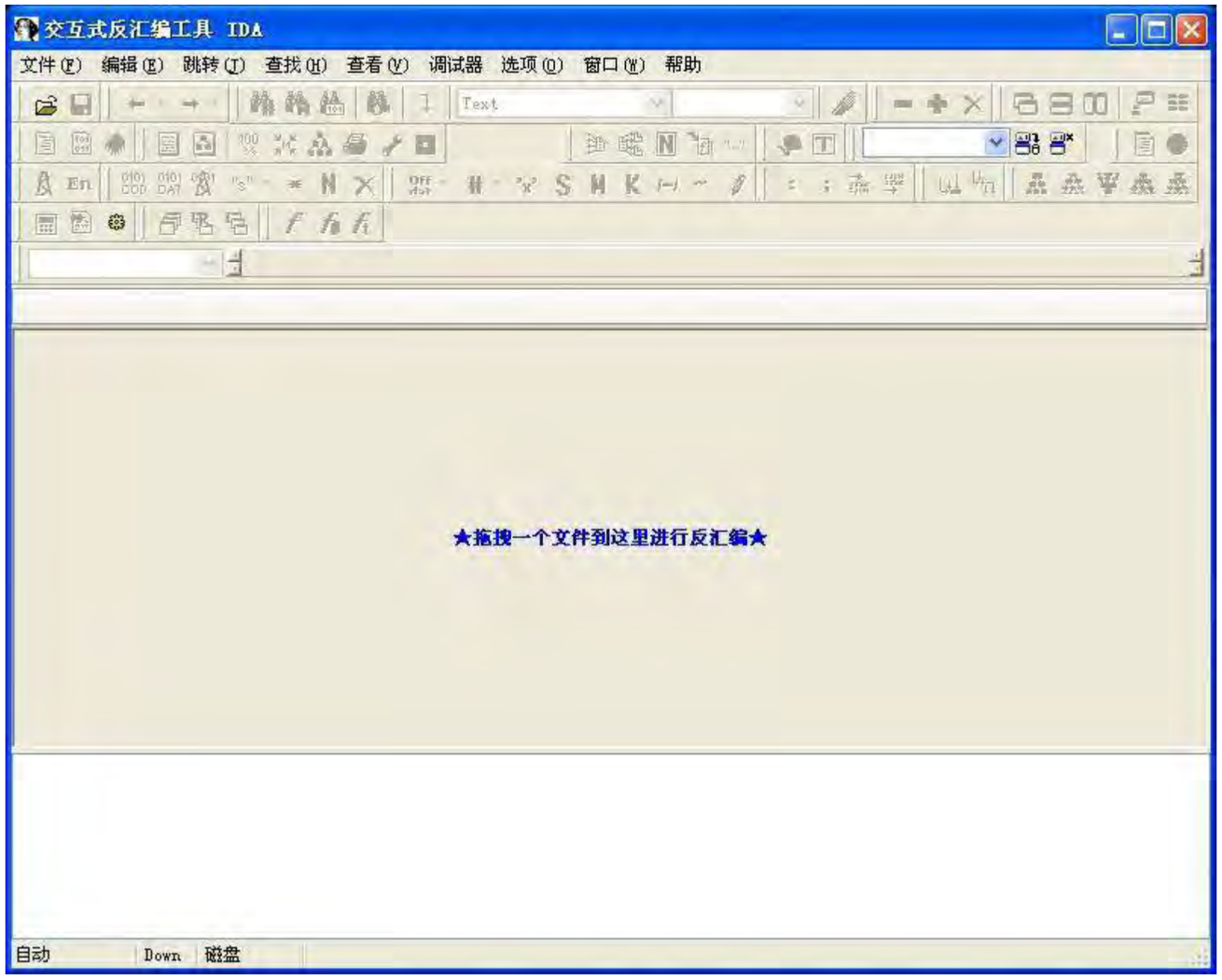
DeviceExtension+54h 的结构是：DeviceExtension->PhysicalDeviceNameBuffer[0]。
以上结论都市错误的，因为 DeviceExtension 是根据用户自己定义的。
16，或结构的 if 选择:如果有很多 if，并且有一个分支是指向同一个地址，那么这个结构可能就是或结构的 if 分支。这个时候就不能取反了，而是按 IDA 翻译的结构直接写出来。
17,mov esi,[ebp+a]
mov ecx,XXX
mov edi,[ebp+b]
rep movsd
正确的翻译方法是：*b=*a;
18，add eax,4
到底是 eax 加 4 还是 eax 的结构偏移 4 位呢。该怎么判断呢？
如果 eax 是整型的，当然是加 4 了；如果是别的 struct 结构，当然是后面哪种情况了。
19，数据结构 3 注释：IO_STACK_LOCATION+4h 的结构是 parameters 数组，该数组的第一个元素是 Read。
IO_STACK_LOCATION+0Ch 的结构是也是 parameters 数组，该偏移量的值是 DeviceIoControl.IoControlCode。以上说的是不正确的，因为 Parameters 是 union 结构。
20，分支结构中间反汇编什么时候停：在一条分支结构中，不需要全部反汇编到文件结尾。如果另一条分支和这条分支会在某一个地址相遇，就在这里停止。也就是在某一个地址分开，在某一个地址有相遇，就选择一个写就好了，不需要两个都写。
20,IO_STATUS_BLOCK 的第一个偏移量是 Status。
21，status 中的各种情况的值：
0C0000001h:STATUS_UNSUCCESSFUL
22,数据结构 4 注释：AssociatedIrp+0h 的结构是 AssociatedIrp 数组，该元素的第一个偏移是：SystemBuffer。
23，mov ecx,XXX1
xor eax,eax
mov edi,XXX2
rep stosd
该代码应该翻译成：RtlZeroMomery(XXX2,XXX1);
24,循环体:一般说在循环方框的前面代码即是循环体的初始化。如果出现 0 赋值某变量，那么应该要注意下该变量是不是循环体的变量。
25，函数的参数：不时说有几个 push 就有多少个参数，因为有些参数占两个 PUSH。怎么看：看看有没有两个 PUSH 压入的参数类型相同，但是只是数值差 4。
26,确定局部变量：找到一些系统的函数，查看 DDK，然后就可以确定有些未知变量的类型了。包括函数值的类型也可以去查了，如果返回值是放在一个未知变量里面的。
27,如果在同一个函数里面有两个相同类型的相同变量，那么他们可能一个是大局部变量，另一个是小局部变量。比如小局部变量位于 case 语句里面。
28,case 语句：如果对于同一个变量，比如 var_1C 有不同的选择分支，那么极可能是 case 语句。对于这类语句，IDA 怎么翻译就怎么翻译，不需要对判断条件求反。
29,ExAllocatePoolWithTag 的返回值的 buffer,它的类型是 PCHAR。
30,MajorFunction[]的结构：38h
+34h:Unload
+38h:IRP_MJ_CTREATE
+40h:IRP_MJ_CLOSE
+70h:IRP_MJ_DEVICE_CONTROL
+80h:IRP_MJ_CLEANUP
31,status 各值的含义：
0C000000Dh：STATUS_INVALID_PARAMETER
0C0000023h: STATUS_BUFFER_TOO_SMALL
0C0000120h: STATUS_CANCELLED
0C0000001h: STATUS_UNSUCCESSFUL
103h: STATUS_PENDING

32,continue 的使用（循环中的 continue）:假设 A 点是循环的开始，到 B 点后跳回 A 点，沿着 A 到 B 的选择的另一分支到达 C 点，又跳回 A 点。则 B 点是一个 continue.
33,while 语句循环：while 的条件判断是在循环方框里面的顶部。
34,DeviceExtensin->Flags 4:DO_BUFFERED_IO。
35,IofCompleteRequest(x,x)的标准式是：IoCompleteRequest(Irp, IO_NO_INCREMENT);
36,返回基地址：CONTAINING_RECORD(IN PCHAR Address,//现在的地址
IN TYPE Type,//返回的类型
IN PCHAR Field//现在的地址在返回类型中的偏移
);
一般这个适用于对一偏移结构出现 sub。
37,Irp->AssociatedIrp.SystemBuffer 的返回值是 PINPUT_DATA 类型的值。

天涯老大讲 IDA 入门

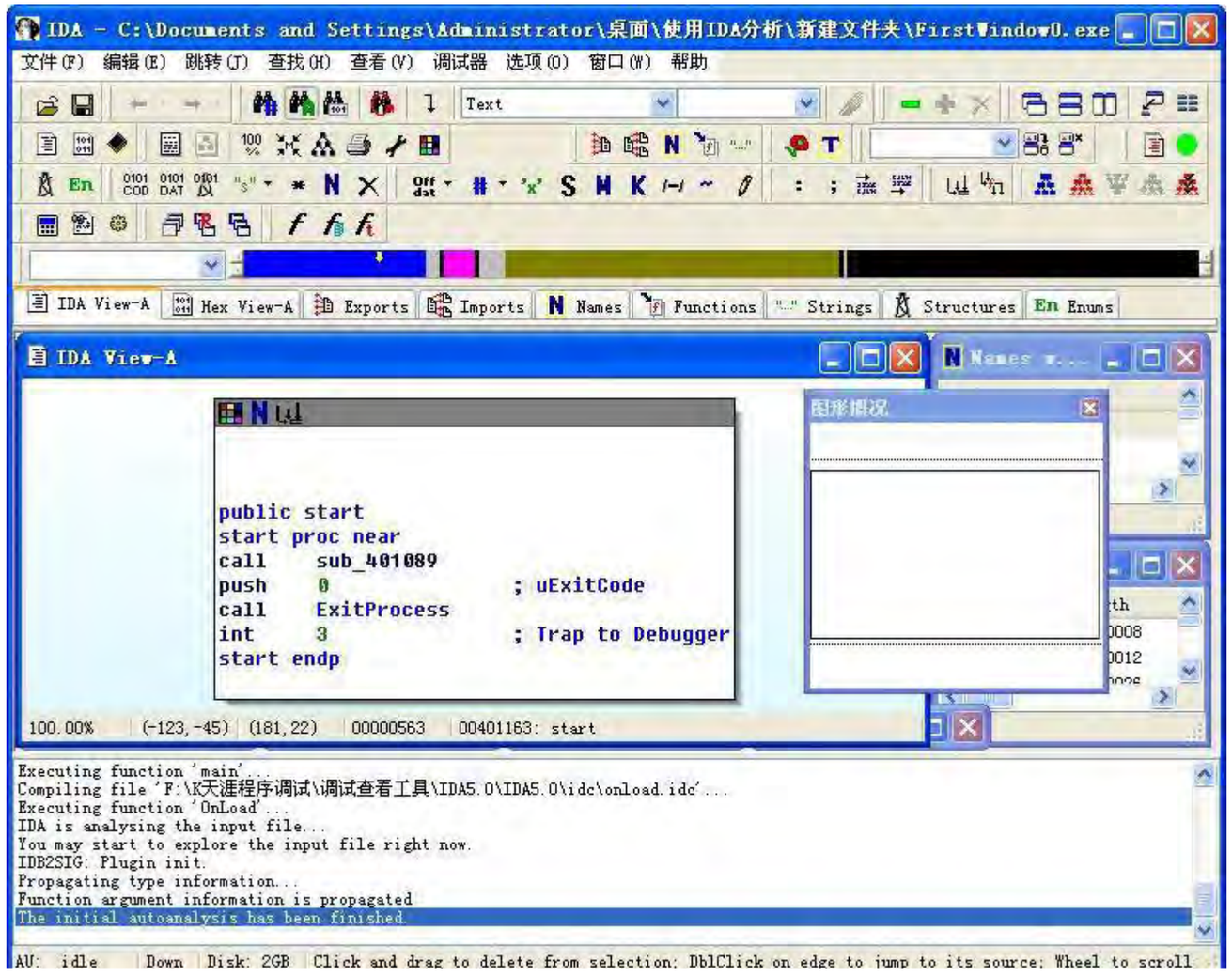
下好 IDA ，运行

会出现这个选择向导。
点新建会让你选择调试模式。
点运行，他只运行这个 IDA　不会让你去选择。
点载入，他会默认使用上一次，调试的程序载入。
我们什么都不做，直接选运行，就 ok。
进入 IDA 主界面：



用他是为了处理调试程序的。
这个时候 ，我们可以从文件菜单中装入你想看到的 exe，dll，驱动等
或按提示拖一个文件进去。大家下好 后，
随便写个小程序，拖进去看看会发生什么情况

会出现上面这样的一个窗口。
我们调试 PE 直接用第一项。
如果我们不想把 PE 当程序，直接用 Binary file。
顾名思义这个是二进制方式直接从文件头打开
注意里面有个手动载入和 Make i..
这两个多用于对付特殊处理过的程序，这个我们以后再说 ，我们今天先讲入门
点击确定按钮进入程序，让 IDA 为我们自动分析：



进入后会出现很多窗口，不要怕，惊喜在后面

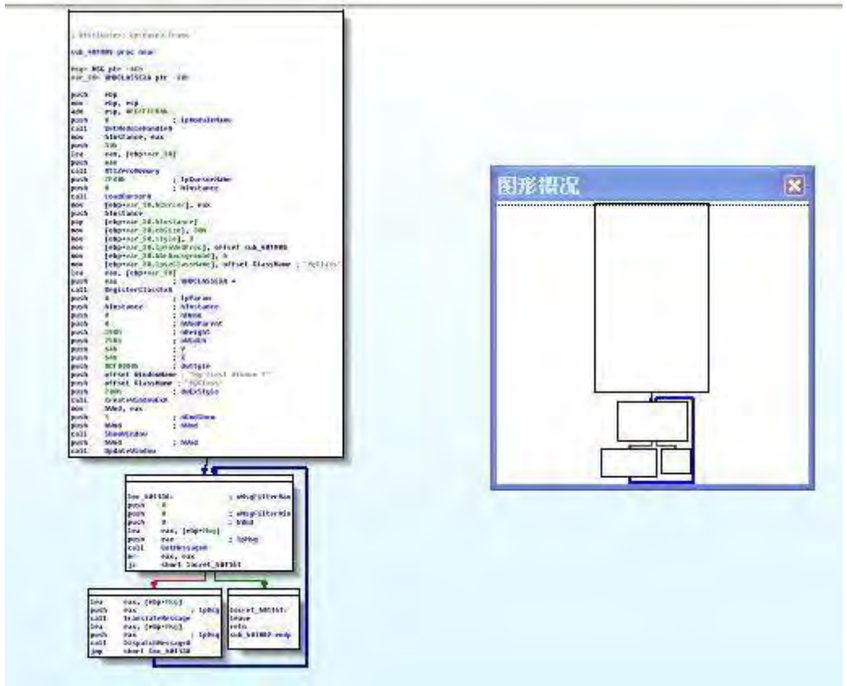
基础 IDA 分析程序是以数据库方式保存的，

它分析的所有东西，都会放进　相应的数据库中，我们在它界面上操作，只相当于对数据库的查询

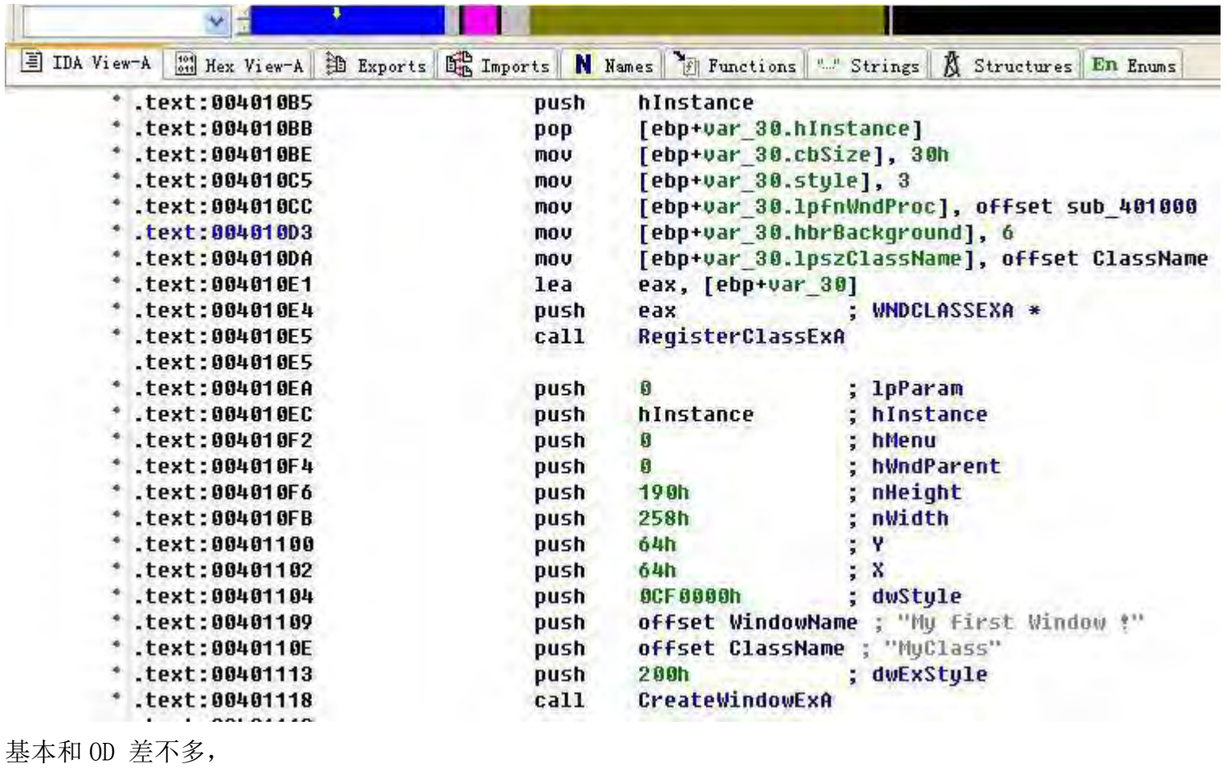
但这样已经非常好了，

IDA　　他是尽量让分析出的汇编代码程序更接近与高级语言的语法，他会更好的反出你真正想得到的高级语言程序原码而设计的，不只是为了调试，，调试不是他的宗旨　，但他的调试也不弱

欣赏几个界面

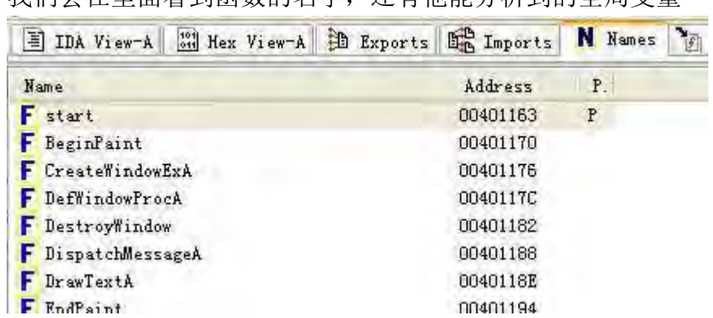


对代码来说，他所显示的模式有两种，
一就是这个图形模式的，带了流程结构的界面。
方便大家对程序的流程理解。
第二种：显示模式，就是列表模式：



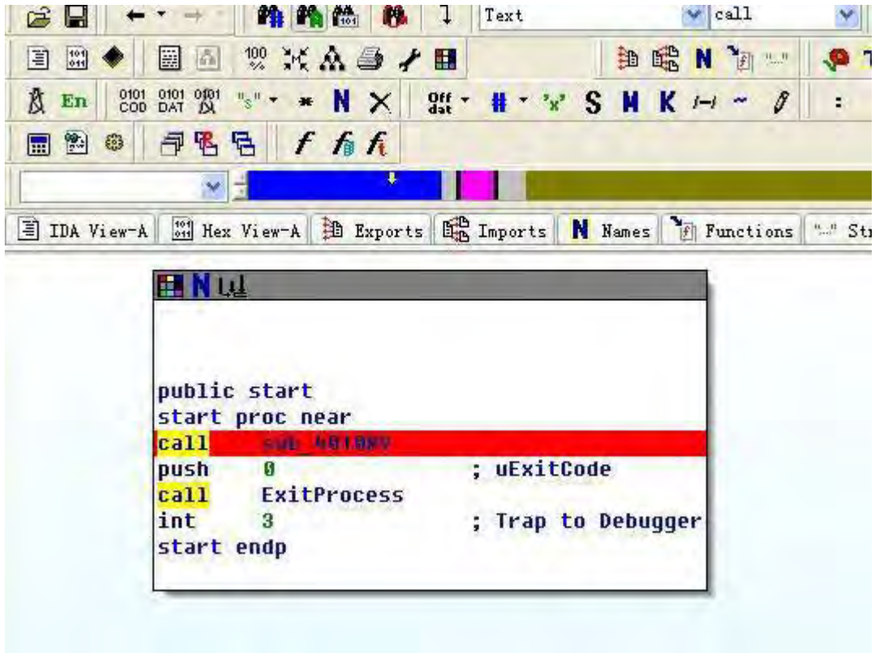
基本和 OD 差不多，
但它把一些变量，参数等，默默的给取名处理了
以方便你 ，反出 程序，做准备
：怎么弄出列表模式和图形模式
：：切换到代码视图
：：工具条找
切换用快截建空格 //ｃ ｔ ｒ ｌ ＋ ｓ ｈ ｉ ｆ ｔ+空格?
漫漫你就不会对 IDA 感到恐怖，漫漫你会爱上他的.....
大家还是尽量开 IDA
什么问题一操作，就方便理解了
先介绍下，他会给我们显示的窗口有哪些：

这是快速入门课，我不会详细讲 ，但 引导性的操作是必须的，
看上面的图。
点哪个你就会看到哪个窗口里面的内容。
看内容，你大概也会知道什么意思了 。
里面的子窗口一共有 18 个。
每个窗口都会透露不同的信息给我们。
我们看程序的时候，会在着 18 个中，不断的参考跳转。
比如 ，名字窗口打开，
我们会在里面看到函数的名字，还有他能分析到的全局变量

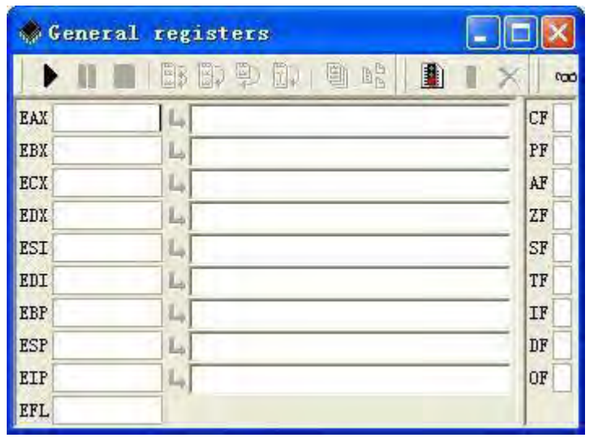


IDA 给我们提供的界面有点象 ie 的超连接，

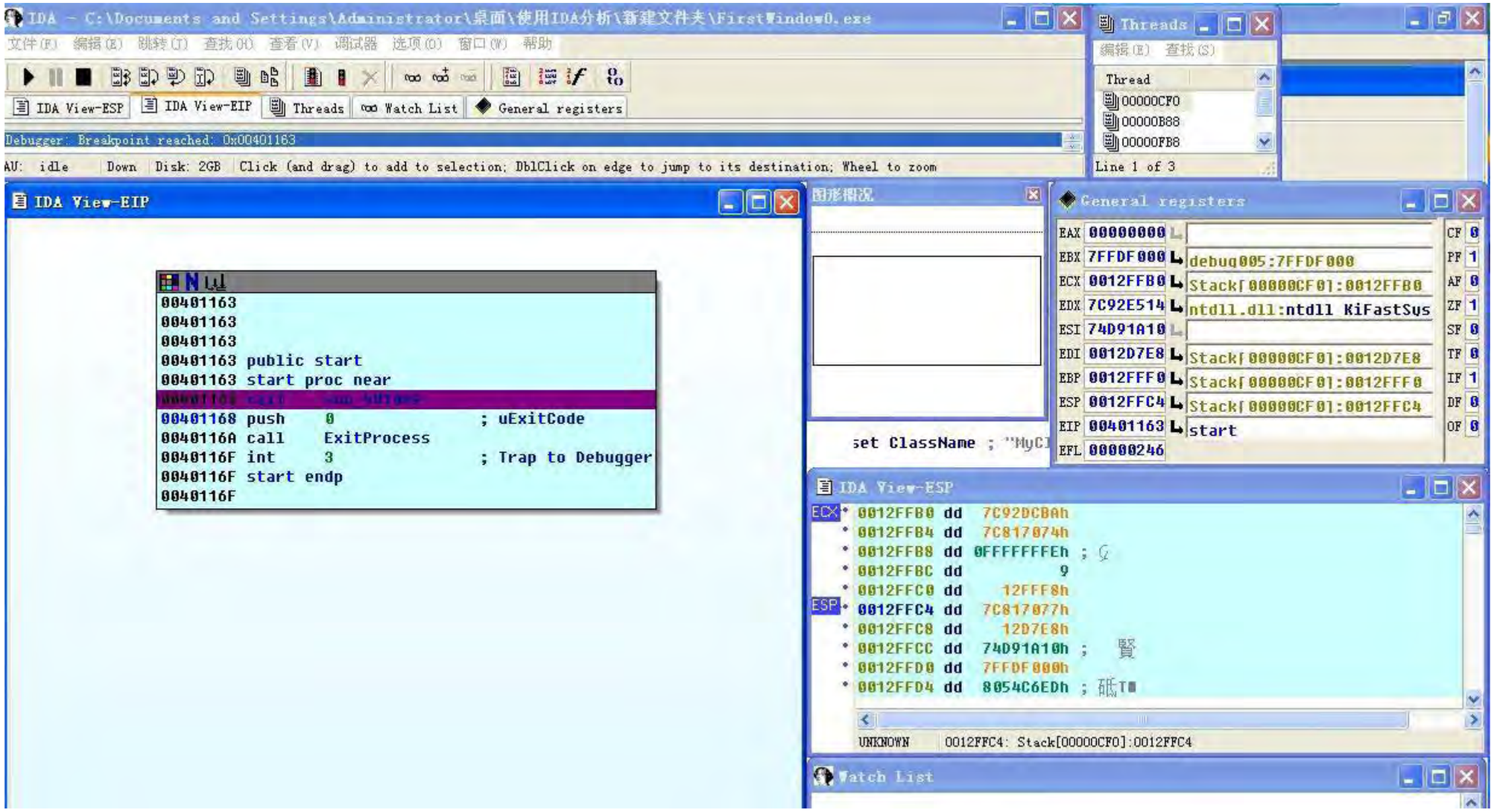
但需要双击来跳转。
比如双击 START，
我们会跑到程序的入口点。
这也是为了我们方便跳转做的。
IDA 窗口多，
显示感觉很乱。
但如果我们对各个窗口的内容理解了，掌握　IDA　，还是很简单的
里面还有输入表窗口，输出表窗口....
都是方便我们查看的，
也方便我们对文件的各种定位。
点那到那，点什么显示什么，非常直观，
大家注意下，以前的 IDA 纯是为了静态反汇编而做的。
发展到现在，他也可以象 OD 一样进行动态调试了
现在我演示让他动起来，看他显示出什么给我们
再入口点下个断点 F2：



点开上面的黑筐筐 

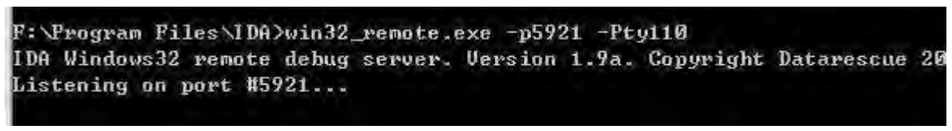


会出现个这样的一个窗口，我们点黑色的三角，我们调试的程序就会运行起来了

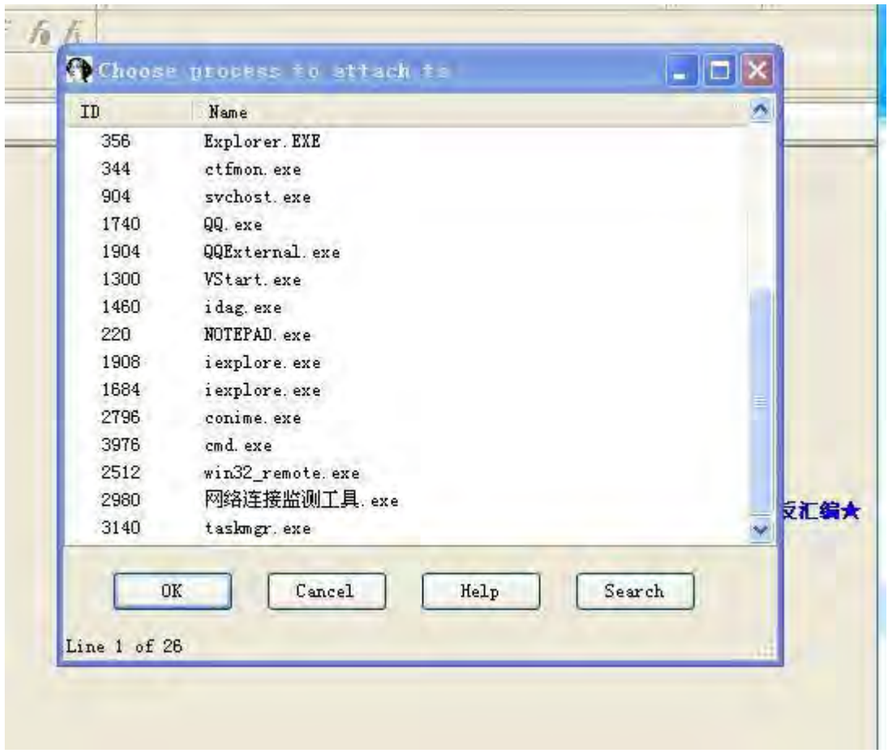


让程序运行起来是另一样状态。
这就是 IDA 的两种分析状态。
静态的，动态的，现在通吃。
讲入门功能都差不多。
版本不是问题。
以后大家熟悉了，随便用。
问题是让大家入门，知道这个优秀软件的存在。
这个软件的界面刚看时会感觉比 OD 乱，但是当你真的熟悉起来的时候，你会感觉他有些东西比 OD 的做法更能让我们理解整个程序的思想。
他就是为反编译而生的

反汇编与反编译是两个概念，大家要区分开。
简单的界面操作，我先不讲，我就带领大家进入不同的领域
因为界面操作的东西要建立在你不断使用中，漫漫都会通
静态的界面也看到了，动态的界面也看到了，
继续深入，
会深入到那里呢，！！？
给大家演示下 远程调试，
谁装 ida 了，跟我配合下
：：联网调试?
绝对的 联网调试
装 ida 的 ，有外网 IP 的跟我联系下
：俺
在水一方　我要你这么做
进入，IDA　目录
里面有个文件　win32_remote.exe
看到了吧
：看到
把他拷贝到一个根目录下
win32_remote.exe　/?
了解下参数
D:\win32_remote.exe -p5921 -Pty110
运行我上面的　指令
-p　表示开某个端口
-P　表示要使用连接密码
在水一方



ip 给我
等待
：123.180.151.20



：连上啦
我会用两种方式去调试你的电脑
就这么神气
：ok
一种方式就是，附加你的当前进程，
一种方式就是，打开你具体目录中的程序
这就是 IDA 给我们提供的两种远程调试方法
这个如果你们都有两台电脑，去调试，全屏游戏是很方便的事

在水一方，关了吧

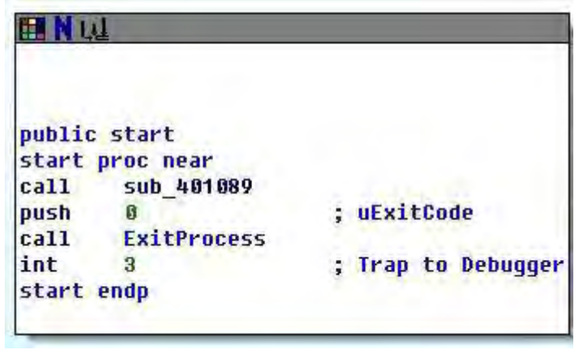
别被别人进入了
这个调试过程可以被利用，运行自己的 shellcode 代码，下载东西的。
看到上面的图没有，
这里就是进入远程调试的入口
第一个是调试他电脑里的，文件，
第二个是调试他电脑里的，进程
里面的分目录，第一个是本地，第二个是远程
看名字大家都应该清楚

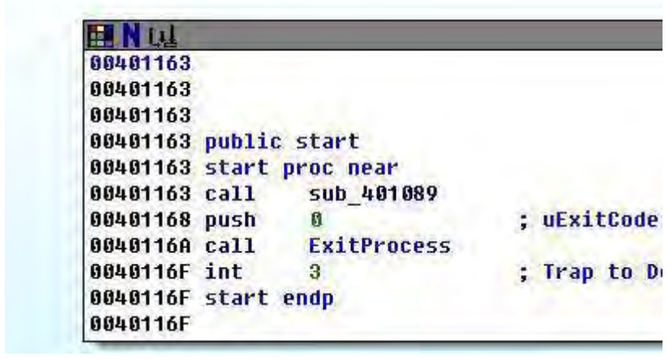
内容就这么添，不解释了

调试程序时还可以带参数，

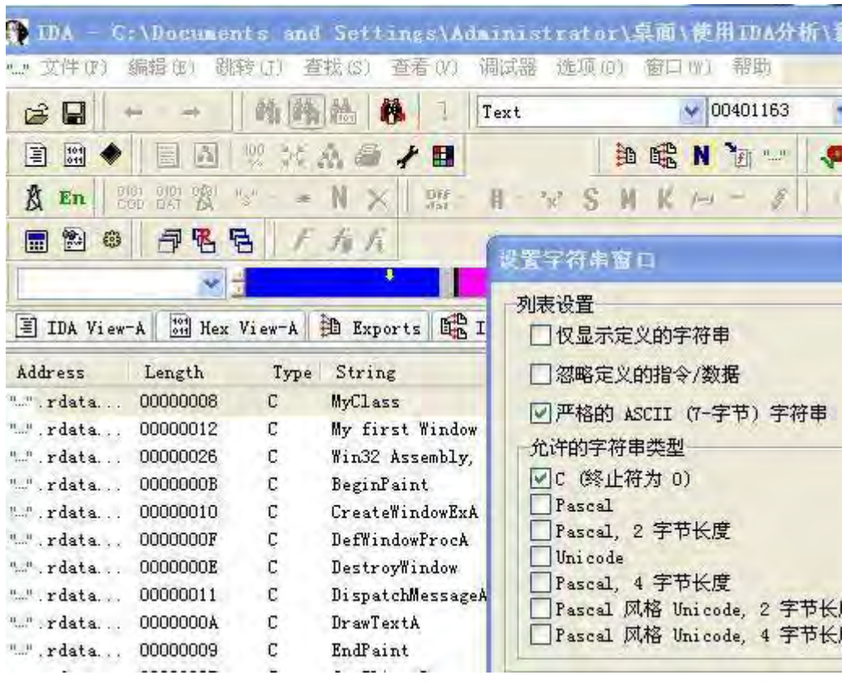
ok　没对　IDA 有点感觉没！！
：反汇编也很给力连填充结构都给出来了
嗯
即使他识别不出结构
你自己可以给他加载结构
程序会用你的结构去改写他的显示指令的
他的最大用处就是反编译

这个远程调试也挺给力
里面任何名字，你都可以修正的
多用这个，你会熟悉他的，反汇编也是很简单的工具
刚开始，你会感觉有点乱 ，比不上 OD
但他的功能远非 OD 能比
D B 快截键 改变数据显示方式！ C 转换为指令
空格切换图形模式与列表模式
箭头绿色去执行红色不执行，蓝色正常执行
按住 Ctrl + - 或鼠标滑轮改变图形模式大小
图形模式默认信息少,但可以配置出来,如带行前缀





去挖掘 IDA 中的各种设置吧，你会看到不一样的 IDA
右键字符串窗口可以设置重新查找各种类型的字符串



双击一个名称　IDA 将跳转到反汇编窗口中被引用的位置或展开内容

IDA 还提供了前进，后退，导航按钮。

用热键 ： ; 可写注释分别常规注释与可重复注释(蓝)

IDA 会创建一个数据库,4 个文件.id0 .id1 .nam .til

可以压缩成一个 IDB 文件

对于病毒，多人分析时可以只共享 IDB 文件，一样调试

IDA　分析不出的结构　，可以这样做

1　在结构体窗口中插入结构体　Insert

使用文本编辑器　以 c 表示法(.h)定义的结构体布局,比使用 IDA 繁琐的手动结构体定义一结构体布局方便　！！

2　使用 EDIT　Struct　Var(ALT　+　Q)　将光标开始处　转成　结构体布局

3　使用数字键盘上的　+　-　可以打开和折叠结构体

装载程序时使用手动载入用于自分析

还有很多很多，我就不一一列举了，漫漫熟悉软件吧，

真是不错的东西，这节课讲完了，此课只再于让大家热爱起工具，善用工具，看到更多精彩的内容

http://sxcode.tap.cn/index/article-21nf1p3cq0104

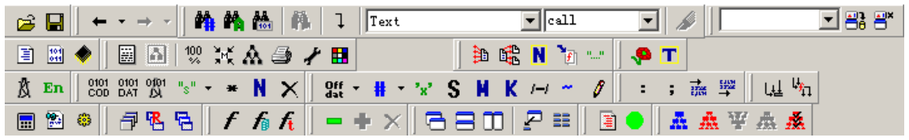
IDA 实例教程

作者：笨笨雄

邮箱：nemo314@gmail.com

1 软件环境

静态分析有很多好处，例如加壳的程序（尽管对于高手来说这并不会耗费太多时间），我们不需要寻找 OEP，也不需要解除自校验，只要修复 IAT，DUMP 下来就可以动手分析了。假如你需要修改程序，可以使用内存补丁技术。动态与静态，调试器与反汇编器结合可以简化分析任务，帮助我们理解代码。因此掌握一种反汇编器是非常必要的。IDA 可以说是这方面的首选工具，它为我们提供了丰富的功能，以帮助我们进行逆向分析。这从 IDA 复杂的工作界面便可以知道。



种类繁多的工具栏

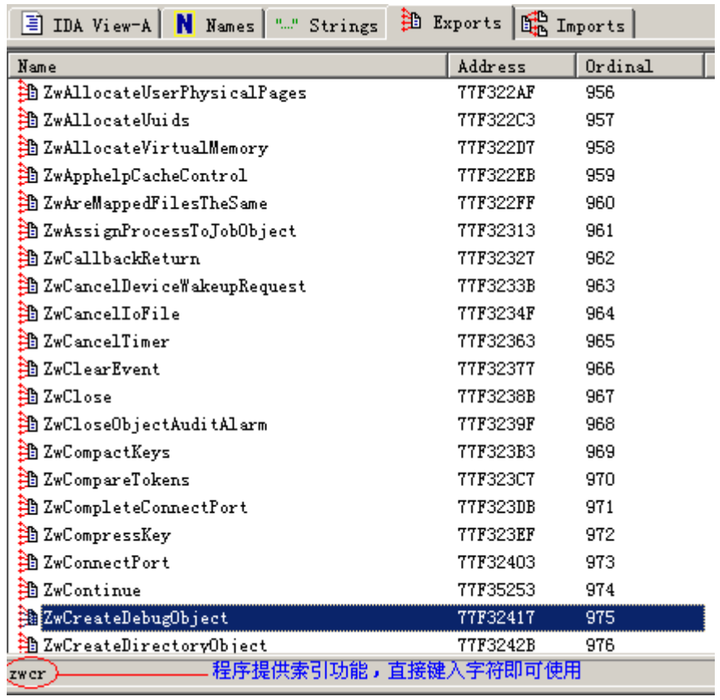
在分辨率不高的情况，这些工具栏与反汇编窗口挤在小屏幕里，看起来不爽。我一般把它关闭（查看=>工具栏=>主工具栏）以获得更好的视觉效果。当我们需要这些功能的时候，直接使用快捷键就可以了。下面是常用快捷键的清单：

快捷键清单

快捷键	功能	注释
C	转换为代码	一般在 IDA 无法识别代码时使用这两个功能整理代码
D	转换为数据	
A	转换为字符	
N	为标签重命名	方便记忆，避免重复分析。
;	添加注释	
R	把立即值转换为字符	便于分析立即值
H	把立即值转换为 10 进制	
Q	把立即值转换为 16 进制	
B	把立即值转换为 2 进制	
G	跳转到指定地址	
X	交叉参考	便于查找 API 或变量的引用
SHIFT+/	计算器	
ALT+ENTER	新建窗口并跳转到选中地址	这四个功能都是方便在不同函数之间分析（尤其是多层次的调用）。具体使用看个人喜好
ALT+F3	关闭当前分析窗口	
ESC	返回前一个保存位置	
CTRL+ENTER	返回后一个保存位置	

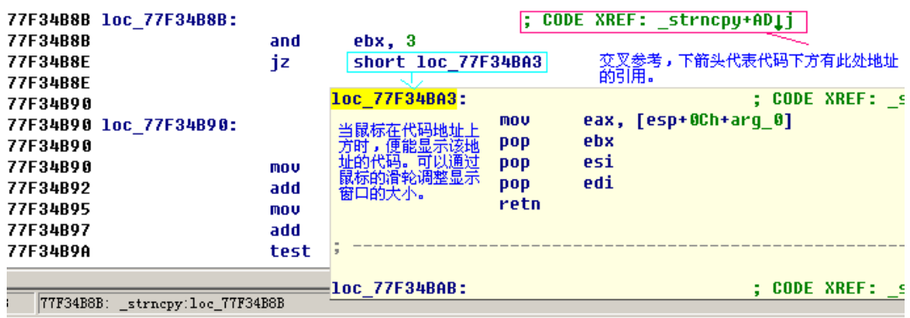
常用分析窗口

在工具栏下面的便是工作窗口。主要的窗口分页有“IDA View-A”、“Name”、“Strings”、“Exports”和“Imports”。对于后面 3 项相信大家都不会陌生了，它们分别是字符参考，输出函数参考和输入函数参考。Name 是命名窗口，在那里可以看到我们命名的函数或者变量。这四个窗口都支持索引功能，可以通过双击来快速切换到分析窗口中的相关内容，使用起来十分方便。



简单输入几个字符即可定位目标

IDA View-A 是分析窗口，支持两种显示模式，除了常见的反汇编模式之后，还提供图形视图以及其他有趣的功能。



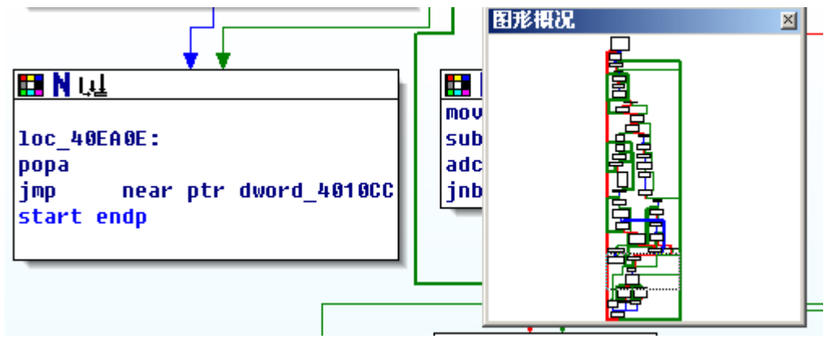
IDA 的反汇编窗口

一般我们在分析的时候，并不关心程序的机械码，所以 IDA 为我们自动隐藏了这些信息。如果你有需要，可以通过以下步骤来设置：

选项=>常规=>反汇编=>显示反汇编行部分=>机械码字节数=>修改为你允许显示的大小

现在让我们以论坛脱壳版块置顶帖的那个经典为例，看看图形视图的表现。首先我们到以下连接下载：http://bbs.pediy.com/upload/bbs/unpackfaq/notepad.upx.rar

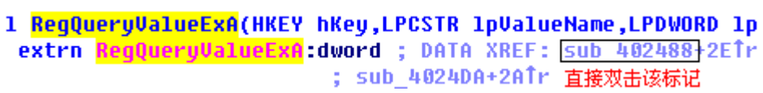
分析流程图



你能通过图形视图及其缩略图快速找到壳的出口吗？

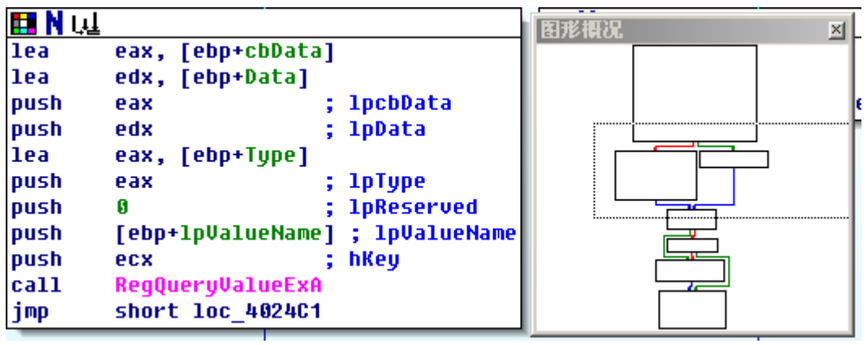
如图所示，标签 40EA0E 便是壳的出口代码的地址。在 OD 中直接跳到该地址，下断点，然后运行到该处，再单步便能看到 OEP 了。假如希望通过跳转法找 OEP，相信图形视图比你在 OD 一个一个跳转跟随，要快得多。

再来看看这个壳的另类脱法。直接运行该程序， DUMP 下来，再使用 IMPORTREC 的 IAT AutoSearch 功能修复输入表。用 IDA 打开修复了输入表的 DUMP 文件。在 IMPORT 窗口随便选一个 API，随便通过交叉参考跳转到一个函数的代码。



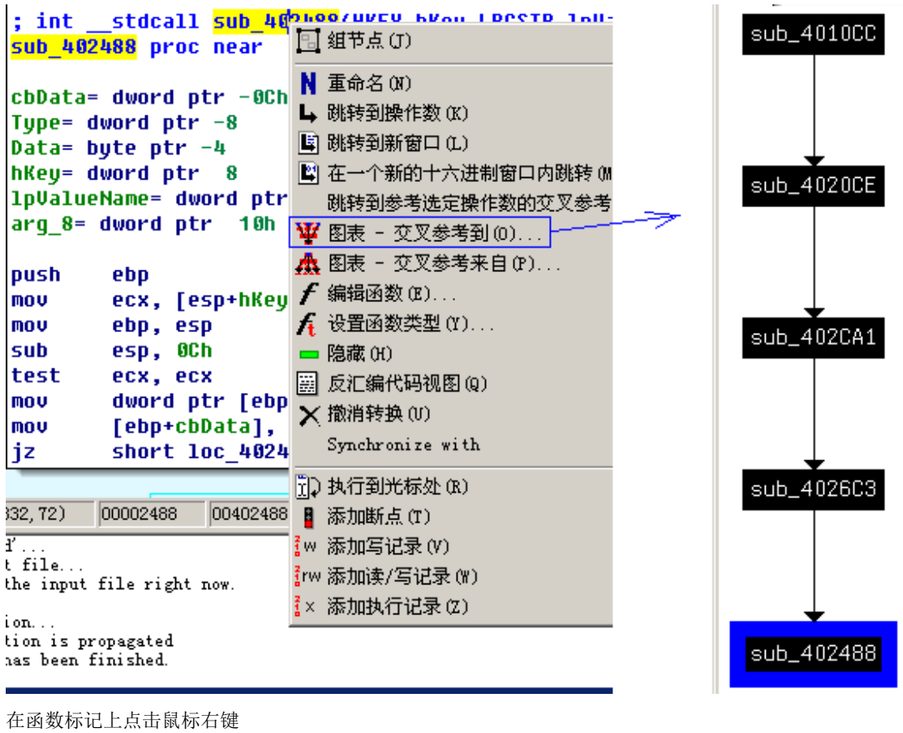
此处为文件输入表的位置

我选了 RegQueryValueExA，通过交叉参考，来到 Sub_402488 处的函数代码。



用鼠标拖动缩略图中的虚线框到上方，便能看到该 CALL 的头部了。然后按下图指示操作：

查找调用关系图



在函数标记上点击鼠标右键

处于最上层的函数，便是 OEP 了，使用 PE 工具修改文件入口为 10CC。现在函数可以正常工作了。这个方法的原理是通常我们写程序都有如下流程：

```
Main proc
        //代码
        CALL    FUN1
        //代码
        CALL    FUN2
        //代码
END proc
```

所以处于函数调用最上层的便是 MAIN 函数了。当然这个方法局限性很大，这里只是对该功能的一种介绍。我们留意到图表功能有两个选项，在上面的例子中，我们使用的是“交叉参考到”。我想细心的朋友大概能通过“交叉参考来自”左边的小图标猜出它的用途了。该功能可以显示目标函数调用了什么函数，当然也包括 API。这样除了观察函数的输入参数来判断是否关键 CALL 之外，又多了一个参考途径。

2 强大的 IDC

有时我们需要分析一些非文件格式的代码，例如 ShellCode，远线程注入和病毒。这些代码的特点便是动态获取 API，这给静态分析带来困难。尽管 IDA 支持分析 2 进制文件，但是缺少 IAT 的情况下，分析起来跟不方便。频繁的切换调试器查看并不是一个好方法。IDC 是 IDA 的脚本语言，它功能强大，为我们提供了另一条与调试器交互的途径。

如何使调试器获得 IDA 分析得出的符号？

IDA 提供多种文件格式输出，调试器可以通过解释这些文件获得一些符号。你可以通过文件菜单中的“创建文件”获得更多的信息。

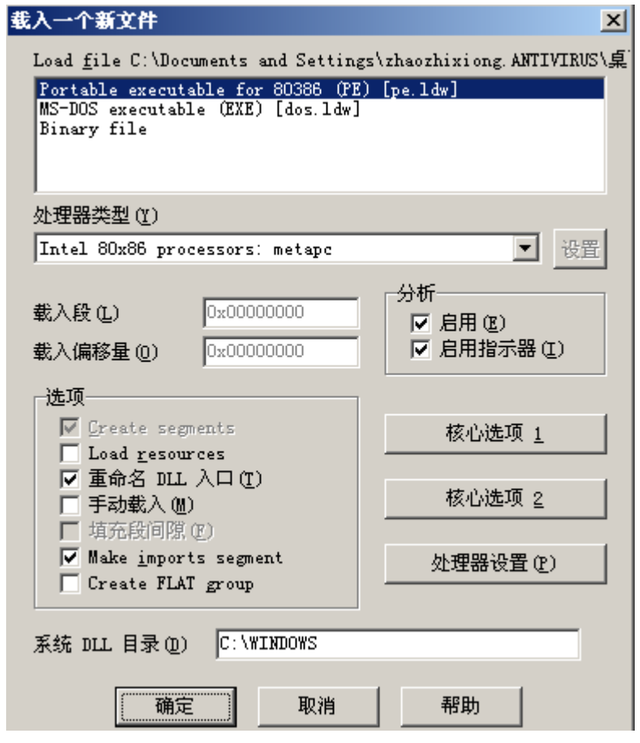
以 OD 为例，它的 GODUP 插件支持解释 MAP 文件（还能加载 IDA 的 SIG）。在 IDA 中使用如下步骤：

菜单：　　文件=>创建文件=>创建 MAP 文件

即可创建 MAP 文件，然后切换到 OD，使用如下步骤便能获得符号了：

菜单：　　插件=>GODUP Plugin=>Map Loader=>Load labels

仍然以那个经典的 UPX 加壳的 NOTEPAD 为例子，这次我们用 OD 打开，在到达 OEP 之后 DUMP 下来，不修复输入表，直接用 IDA 载入后看到下图：



丰富的文件载入选项

需要注意的是 Make imports segment 是 PE 文件特有的选项，该选项会隐藏输入表区域的所有数据，同时你获得的好处便是能在图表功能中看到 API 的调用。假如你希望查看在输入表的范围内的代码或者数据，你需要使用从菜单中选择“编辑”=>“区段”以删除遮挡数据的部分区段。

为了更真实的模拟从内存中截取代码的情况，在这里选择 Binary file，载入偏移量选 400000（根据实际代码在内存中的基址来选择），然后 IDA 就开始尝试分析可能存在于该文件中的代码了。对照 OD 中的 OEP 地址，在 IDA 中可以看到以下代码：

```
seg000:004010CC                 push    ebp
seg000:004010CD                 mov     ebp, esp
seg000:004010CF                 sub     esp, 44h
seg000:004010D2                 push    esi
seg000:004010D3                 call    ds:dword_4063E4
seg000:004010D9                 mov     esi, eax
seg000:004010DB                 mov     al, [eax]
seg000:004010DD                 cmp     al, 22h
seg000:004010DF                 jnz     short loc_4010FC
```

OEP 处的部分代码

OD 中对应的显示：

004010D3　　FF15 E4634000　　call　　dword ptr [4063E4]　　; kernel32.GetCommandLineA

使用以下 ollyscript （附件中的 ollyGetSym.txt）提取 IAT 的符号：

```
var ea
var Ecount                          //0 分隔号的记数器
var oFile

ask "请输入 IAT 起始地址"
cmp $RESULT, 0
je ECancel
mov ea, $RESULT
ask "输出文件？"
cmp $RESULT, 0
je ECancel
mov oFile, $RESULT

TryGetSym:
GN [ea]                             //获取该地址的符号
cmp $RESULT,00000000        //OLLYSCRIPT 是区分 00000000 和 0 的
je ETest
WRTA oFile,$RESULT_2
```

```
    mov Ecount,0
    add ea,4
    jmp TryGetSym

    ECancel:
    msg "无效输入"
    ret

    ETest:
    cmp Ecount,1                    //不同模块的地址以 0 分隔
    je Send                         //若存在两个 DWORD 的 0 则认为是末尾
    add Ecount,1
    add ea,4
    jmp TryGetSym

    SEnd:
    Ret
```

使用下面 IDC 脚本获取符号并对相应地址重命名：

```
#include <idc.idc>

static main() {
   auto Sbuffer,ea,zcount,filehandle,fileName,CustEa;
fileName = AskFile (0,"*.*","打开 IAT 符号文件");
CustEa = AskAddr(0,"目标 IAT 地址");
filehandle = fopen(fileName,"r");
for (ea = CustEa; zcount < 2; ea = ea + 4){
      if (Dword(ea) !=0){
      Sbuffer = readstr(filehandle);
            if(strlen(Sbuffer) < 2){          //ollyscript 的输出文件存在无效字符
            Sbuffer = readstr(filehandle); //如果字符无效则再取一次字符
                                    }
      MakeNameEx (ea,Sbuffer,SN_AUTO );      //为对应 DWORD 改名
      zcount = 0;
                        }
      else{
      zcount = zcount + 1;
            }
                                    }
fclose(filehandle) ;
}
```

GetSym.idc

正如 ollyscript 接近于 ASM，IDC 的函数及其语法也近似于 C 语言（详见 IDA 的帮助），在编写几个脚本之后，便能轻松掌握它的用法。

```
seg000:004010CC                 push    ebp
seg000:004010CD                 mov     ebp, esp
seg000:004010CF                 sub     esp, 44h
seg000:004010D2                 push    esi
seg000:004010D3                 call    ds:GetCommandLineA_
seg000:004010D9                 mov     esi, eax
seg000:004010DB                 mov     al, [eax]
seg000:004010DD                 cmp     al, 22h
seg000:004010DF                 jnz     short loc_4010FC
```

现在可以正常显示函数调用的 API 了

下面来看看另外一个例子中 IDC 的表现。附件中的 Exvirus.v 是一个木马程序。当然这里并不是要分析这个木马，更不会运行它，在静态分析的环境下，很安全。

Address	Length	Type	String
"…" CODE:0…	0000001C	C	SOFTWARE\\Borland\\Delphi\\RTL
"…" CODE:0…	0000000D	C	FPUMaskValue
"…" CODE:0…	0000000D	C	kernel32.dll
"…" CODE:0…	00000018	C	媧狒透镙扈展鸪灿铖痼栾
"…" CODE:0…	00000050	C	……
"…" CODE:0…	00000006	C	\xFF\xFF\xFF\xFF\a
"…" CODE:0…	00000006	C	Micro
"…" CODE:0…	00000013	C	GetCurrentThreadId
"…" CODE:0…	0000000D	C	Kernel32.dll
"…" CODE:0…	00000010	C	轷運xFF\xFF腚嬶[YY]鞍b
"…" CODE:0…	00000011	C	殒赵xFF\xFF腚嫧]宵xFF\xFF\xFF\xFF\n
"…" CODE:0…	0000000B	C	蔚鏊镱
"…" CODE:0…	00000006	C	\xFF\xFF\xFF\xFF\v
"…" CODE:0…	00000012	C	厌殺xFF\xFF腚蝻闢^[嫧]
"…" CODE:0…	00000011	C	藻蟮轭狒道蝻沐篌
"…" CODE:0…	0000000D	C	kernel32.dll
"…" CODE:0…	00000005	C	$$cd
"…" CODE:0…	00000014	C	檩蚺xFF\xFF豚_^[嫧]宵xFF\xFF\xFF\xFF\a
"…" CODE:0…	00000008	C	猃甬屮
"…" CODE:0…	0000000A	C	驳裟鼢祐
"…" CODE:0…	00000005	C	趄镱
"…" CODE:0…	00000005	C	TJPm
"…" CODE:0…	0000000D	C	物糸骋汜糸镱
"…" CODE:0…	00000011	C	?蔦xFF\xFF滕蟒_^[嫧]
"…" CODE:0…	0000001B	C	昨铙道蟮鲠翦序锨殖逵趄轭缃
"…" CODE:0…	00000006	C	\xFF\xFF\xFF\xFF\v
"…" CODE:0…	00000017	C	义玳簪驳渝蝤狙道蝻沐篌
"…" CODE:0…	00000041	C	JiF　騷sUk豍谒潬Yn]g墁揾牠U　　g嘉杠敲捋
"…" CODE:0…	00000021	C	桔翱函　鹇　搞锹　镭　般磔殖　簏

几乎都是乱码的窗口

加密了的字符，总要在使用之前解密。也就是说可以通过加密字符的交叉引用定位解密代码。

```
lea         edx, [ebp+var_4]
mov         eax, offset s_XsXQqSxUsSq ; "抿遽翦燥镬桢袱巢宇狃箬雉"
call        sub_404BEE
```

通过交叉引用定位的函数

由字符参考中的“SOFTWARE\Borland\Delphi\RTL”可以判断该木马是用 Delphi 编写的（也可从函数的参数传递约定判断）。在详细分析之前，先在菜单中进行如下步骤的操作：

文件=>加载文件=>加载 FLIRT 签名文件=>Delphi7 RTL/VCL/CLX

现在 IDA 将会根据 Delphi 的函数特征识别出一些库函数，这样可以减少很多工作量。

```
CODE:00404C2C                   mov         [ebp+var_8],1    //已处理字符记数器
CODE:00404C2C
CODE:00404C33
CODE:00404C33 loc_404C33:                   ; CODE XREF: sub_404BEE+6Aj
CODE:00404C33                   mov         eax, [ebp+var_4]
CODE:00404C36                   mov         edx, [ebp+var_8]
CODE:00404C39                   mov         bl, [eax+edx-1]  //单字节取字符解密
CODE:00404C3D                   add         bl, 80h
CODE:00404C40                   lea         eax, [ebp+var_C]
CODE:00404C43                   mov         edx, ebx
CODE:00404C45       call        @System@@LStrFromChar$qqrr17System@AnsiStringc
CODE:00404C45
CODE:00404C4A                   mov         edx, [ebp+var_C]
CODE:00404C4D                   mov         eax, edi
CODE:00404C4F                   call        @System@@LStrCat$qqrv
CODE:00404C4F
CODE:00404C54                   inc         [ebp+var_8]
CODE:00404C57                   dec         esi      //字符长度=0 跳出循环，解密完毕
CODE:00404C58                   jnz         short loc_404C33
```

函数较长，这里只列出关键代码。判断这部分为关键代码主要是因为整个函数就只有该处是循环。解密是对一定长度的数据进行运算，因此会有一个循环对字符中的数据逐一解密。然后从输入参数与寄存器或者堆栈的关联便可以理解函数的关键部分是如何工作的。由于 IDA 已经为我们识别出 Delphi 的库函数，所以这里很容易便知道解密的方便是对目标字符的每个字节都加上 80h。下面来看看我如何使用 IDC 来完成解密字符的工作。

Decode.idc

```
#include "idc.idc"

static main() {
    auto ea,x,y,z,zbyte,SRange,TStrLen,DeCodeBuffer,DeCodeCounter,NotTarget;

x = 0x404bee;

    for ( y=RfirstB(x); y != BADADDR; y=RnextB(x,y) ){    //通过交叉参考取得函数调用地址
        for (SRange = 4; SRange < 0x50; SRange++){
                z = y - SRange;
                zbyte = Byte(z);
                if (zbyte == 0xb8){      //mov eax,mem32 的机械码是 b8
                        zbyte = Dword(z + 1);
                        ea = Dword(zbyte);
                        if (ea != 0xFFFFFFFF){      //判断 mem32 是否有效，防止识别错指令
                                if (Byte(zbyte - 1) == 1){    //在字符指针前一个字节写入处理标记
                                        break;                        //避免重复处理
                                                }
                                PatchByte (zbyte - 1,1);
                                TStrLen = 0;
                                while (TStrLen < 0x30){      //解密的循环
                                DeCodeCounter = zbyte + TStrLen;
                                DeCodeBuffer = Byte(DeCodeCounter) + 0x80;
                                if (DeCodeBuffer == 0x80) break;
                                PatchByte (DeCodeCounter,DeCodeBuffer);
                                TStrLen++;
                                                }
                                MakeUnknown (zbyte,TStrLen,0);      //取消 IDA 原来的分析结果
                                MakeStr (zbyte, DeCodeCounter);      //把该位置标记为字符
                                break;
                                        }
                                }
                                        }
                                                                }
        }
```

既然可以通过加密字符定位目标函数，那么也可以通过加密函数定位加密字符。通过使用解密函数的交叉引用，往上搜索，解密第一条 mov eax,mem32 中的字符。当然这里个脚本写得有点简陋，并不能完全解决程序中的加密字符。这个就任务就留给读者来挑战吧。这里要注意的是我在编写 IDC 的过程中遇到很多 BUG，这是因为 IDA 区分大小写（调试了很久才知道）。此外要转换数据类型得先把原来的分析结果取消才可以。最后要看到下图的窗口，在运行脚本后，你需要重新打开字符参考窗口（不会自动刷新）。

Address	Length	Type	String
"..." CODE:0...	0000001C	C	SOFTWARE\\Borland\\Delphi\\RTL
"..." CODE:0...	0000000D	C	FPUMaskValue
"..." CODE:0...	0000000D	C	kernel32.dll
"..." CODE:0...	00000018	C	CreateToolhelp32Snapshot
"..." CODE:0...	0000000E	C	Process32First
"..." CODE:0...	0000000D	C	Process32Next
"..." CODE:0...	0000000D	C	Module32First
"..." CODE:0...	0000000C	C	Module32Next
"..." CODE:0...	0000000D	C	Thread32First
"..." CODE:0...	0000000C	C	Thread32Next
"..." CODE:0...	00000050	C	.....
"..." CODE:0...	00000006	C	\xFF\xFF\xFF\xFF\a
"..." CODE:0...	00000007	C	C:\\game
"..." CODE:0...	00000006	C	pFilil
"..." CODE:0...	00000006	C	Micro
"..." CODE:0...	00000013	C	GetCurrentThreadId
"..." CODE:0...	0000000D	C	Kernel32.dll
"..." CODE:0...	00000010	C	轧運xFF\xFF靛嬶[YY]菽b
"..." CODE:0...	00000009	C	ntdll.dll
"..." CODE:0...	0000000C	C	NtOpenThread
"..." CODE:0...	0000000C	C	Kernel32.dll
"..." CODE:0...	0000000B	C	ExitProcess
"..." CODE:0...	00000011	C	殞赻xFF\xFF靛嫗]肓xFF\xFF\xFF\xFF\u
"..." CODE:0...	0000000A	C	RavMon.exe
"..." CODE:0...	00000006	C	\xFF\xFF\xFF\xFF\v
"..." CODE:0...	0000000B	C	RavMonClass
"..." CODE:0...	0000000B	C	Ravmond.EXE
"..." CODE:0...	0000000B	C	IPARMOR.EXE
"..." CODE:0...	00000008	C	adam.exe

解密后的字符参考窗口

2 静态脱壳

上一节我们用 IDC 完成了字符解密的工作，既然脱壳的过程实际就是对源程序的解密，现在让我们来尝试在不运行壳的情况下把壳解决掉。首先到下面连接下载一个壳：

http://www.pediy.com/tools/PACK/Protectors/MSLRH/MSLRHv0.31a.rar

主页对这个壳的介绍是可以作为 Unpackme 练练手，现在就以该壳的主程序作为例子讲解如何静态脱壳。首先用 IDA 加载该壳的主程序。

```
seg005:004560FA loc_4560FA:               ; CODE XREF: start:loc_4560F4j
seg005:004560FA                           call    sub_456109
seg005:004560FA
seg005:004560FA start                     endp                    //入口函数的结尾
seg005:004560FA
seg005:004560FF
seg005:004560FF
seg005:004560FF
seg005:004560FF sub_4560FF    proc near     ; CODE XREF: seg005:00456104p
seg005:004560FF                                       ; sub_456109p     //红色
seg005:004560FF                           call    sub_456DEF
seg005:004560FF
seg005:004560FF sub_4560FF          endp
seg005:004560FF
seg005:00456104                           call    sub_4560FF
seg005:00456104
seg005:00456109
seg005:00456109
seg005:00456109
seg005:00456109 sub_456109        proc near    ; CODE XREF: start:loc_4560FAp
seg005:00456109         call      near ptr sub_4560FF+1    //+1 表示反汇编出现混乱
```

正常的交叉参考标记是绿色，当显示为红色时则证明与其他部分的反汇编代码产生冲突。另外在 jcc,jmp 和 call 后面出现“+X”的符号（X 为任意数字），一般也为反汇编出现混乱。在正式分析之前，我们必须找到花指令的规律，编写脚本，除去它的影响。现在我们从最初产生影响的地方开始。点击地址 4560FF，按 D

```
seg005:004560FF byte_4560FF          db 0E8h;          CODE XREF: seg005:00456p
seg005:00456100 unk_456100          db 0EBh ; ?    ; CODE XREF: sub_456109p
seg005:00456101                      db    0Ch
seg005:00456102                      db      0
seg005:00456103                      db      0
seg005:00456104                      call      near ptr byte_4560FF
```

注意 00456104 处也是花指令之一，它的作用就是让 IDA 误以为 004560FF 处为有效指令。因此也在该位置上按 D，将其转换为数据。而在 00456100 处按 C 转换为代码。

```
seg005:004560FA                         call        sub_456109
seg005:004560FA
seg005:004560FA start                   endp
seg005:004560FA
seg005:004560FA ; ---------------------------------------------------------------------------
seg005:004560FF                         db 0E8h
seg005:00456100 ; ---------------------------------------------------------------------------
seg005:00456100
seg005:00456100 loc_456100:                         ; CODE XREF: sub_456109p
seg005:00456100                         jmp         short loc_45610E
seg005:00456100
seg005:00456100 ; ---------------------------------------------------------------------------
seg005:00456102                         db        0
seg005:00456103                         db        0
seg005:00456104                         db 0E8h
seg005:00456105                         db 0F6h ; ?
seg005:00456106                         db 0FFh
seg005:00456107                         db 0FFh
seg005:00456108                         db 0FFh
seg005:00456109
seg005:00456109
seg005:00456109 sub_456109      proc near    ; CODE XREF: start:loc_4560FAp
seg005:00456109                             call        loc_456100
seg005:00456109
seg005:0045610E
seg005:0045610E loc_45610E:        ; CODE XREF: seg005:loc_456100j
seg005:0045610E                             add         esp, 8
```

现在我们手动修正了一处被花掉的代码。我们知道 OPCODE 的 E8 和 EB 后面的实际是一个相对地址偏移，而不是地址编码（反汇编翻译成地址是便于分析）。因此可能你已经想到通过搜索内存中的相应指令序列，然后告诉 IDA 什么是代码，什么则不是。读者可以先试试自己找出壳中花指令的规律，然后对比一下结果。

经过手动整理之后，发现壳使用了下面 4 种花指令代码：

```
        call       label1
        db 0E8h
label2:
        jmp        label3
        db 0
        db 0
        db 0E8h
        db 0F6h ;
        db 0FFh
```

```
    db 0FFh
    db 0FFh
label1:
    call    label2
label3:
    add     esp, 8
```

花指令 1

```
    Jz      label1
    Jnz     label1
    db 0EBh
    db  2
label1:
    jmp     label2
    db 81h
label2:
```

花指令 2

```
    push    eax
    call    label1
    db 29h
    db 5Ah
label1:
    pop     eax
    imul    eax, 3
    Call    label2
    db 29h
    db 5Ah
label2:
    add     esp, 4
    pop     eax
```

花指令 3

```
    Jmp label1
    db 68h
Label1:
    Jmp label2
    db 0CDh, 20h
Label2:
    Jmp label3
    db 0E8h
Label3:
```

花指令 4

在知道花指令结构之后，容易写出下面脚本用 NOP（0x90h）来代替干扰的反汇编器的数据：

```
static PatchJunkCode() {
        auto x,FBin,ProcRange;

    FBin = "E8 0A 00 00 00 E8 EB 0C 00 00 E8 F6 FF FF FF";
    // 花指令 1 的特征码
    for (x = FindBinary(MinEA(),0x03,FBin);x != BADADDR;x =
FindBinary(x,0x03,FBin)){

    x = x +5;
    PatchByte (x,0x90);
    x = x + 3 ;
    PatchByte (x,0x90);
    x++;
    PatchWord (x,0x9090);
    x =x +2 ;
    PatchDword (x,0x90909090);

                                                }

    FBin = "74 04 75 02 EB 02 EB 01 81";
    // 花指令 2 的特征码
    for (x = FindBinary(MinEA(),0x03,FBin);x != BADADDR;x =
FindBinary(x,0x03,FBin)){

    x = x + 4;
    PatchWord (x,0x9090);
    x = x + 4;
    PatchByte (x,0x90);

                                                }
    FBin = "50 E8 02 00 00 00 29 5A 58 6B C0 03 E8 02 00 00 00 29 5A 83 C4 04";
    // 花指令 3 的特征码
    for (x = FindBinary(MinEA(),0x03,FBin);x != BADADDR;x =
FindBinary(x,0x03,FBin)){

    x = x + 6;
    PatchWord (x,0x9090);
    x = x + 11;
```

```
        PatchWord (x,0x9090);
                                                        }

        FBin = "EB 01 68 EB 02 CD 20 EB 01 E8";
        //  花指令 4 的特征码
        for (x = FindBinary(MinEA(),0x03,FBin);x != BADADDR;x =
FindBinary(x,0x03,FBin)){

        x = x+2;
        PatchByte (x,0x90);
        x = x+3;
        PatchWord (x,0x9090);
        x = x+4;
        PatchByte (x,0x90);
                                                        }
                }
```

通过观察可知花指令中并不包含任何有意义的数据，在花指令的前后，堆栈是平衡的，各寄存器的数值也是不变的。IDC 提供了隐藏区域的命令，现在来看看以下脚本：

```
static HideJunkCode() {
          auto x,y,FBin;

        FBin = "E8 0A 00 00 00 E8 EB 0C 00 00 E8 F6 FF FF FF";

        for (x = FindBinary(MinEA(),0x03,FBin);x != BADADDR;x =
FindBinary(x,0x03,FBin)){
        MakeUnknown (x,0x17,1);
        y = x + 0x17;
        HideArea (x,y,atoa(x),atoa(x),atoa(y),-1);

                                                        }

        FBin = "74 04 75 02 EB 02 EB 01 81";

        for (x = FindBinary(MinEA(),0x03,FBin);x != BADADDR;x =
FindBinary(x,0x03,FBin)){
        MakeUnknown (x,0x09,1);
        y = x + 0x09;
        HideArea (x,y,atoa(x),atoa(x),atoa(y),-1);

                                                        }
```

```
        FBin = "50 E8 02 00 00 00 29 5A 58 6B C0 03 E8 02 00 00 00 29 5A 83 C4 04";

        for (x = FindBinary(MinEA(),0x03,FBin);x != BADADDR;x =
FindBinary(x,0x03,FBin)){
        MakeUnknown (x,0x17,1);
        y = x + 0x17;
        HideArea (x,y,atoa(x),atoa(x),atoa(y),-1);

                                                        }

        FBin = "EB 01 68 EB 02 CD 20 EB 01 E8";

        for (x = FindBinary(MinEA(),0x03,FBin);x != BADADDR;x =
FindBinary(x,0x03,FBin)){
        MakeUnknown (x,0x0a,1);
        y = x + 0x0a;
        HideArea (x,y,atoa(x),atoa(x),atoa(y),-1);
                                                        }
        }
```

由于花指令的关系，会使 IDA 错误识别指令，可能隐藏区域的边界刚好在一条指令的机械码中间，这样隐藏的操作便会失败。因此在隐藏指令执行之前，先使用 MakeUnknown 将目标代码设置为未识别的状态。在完成隐藏和替换之后，再使用分析引擎分析代码。

```
static main() {
            auto x,FBin,ProcRange;

        HideJunkCode();

        PatchJunkCode();

        AnalyzeArea (MinEA(),MaxEA());
                    }
```

CleanJunkCode.idc

在运行脚本之后，现在让我们看看修复的成果。

```
seg005:0045639F                         rdtsc
seg005:004563A1                         push    eax
seg005:004563A2                         rdtsc
seg005:004563A4 ; seg005:004563A4       //被隐藏的区域
seg005:004563BB                         sub     eax, [esp-8+arg_4]
seg005:004563BE ; seg005:004563BE
seg005:004563C7 ; ---------------------------------------------------------------------------
seg005:004563C7
seg005:004563C7 loc_4563C7:     ; CODE XREF: sub_4563B3:loc_4563C4j
seg005:004563C7                         add     esp, 4
seg005:004563CA ; seg005:004563CA
seg005:004563E1                         cmp     eax, 0FFFh
seg005:004563E6 ; seg005:004563E6
seg005:004563F0
seg005:004563F0 loc_4563F0:     ; CODE XREF: sub_4563D9:loc_4563EDj
seg005:004563F0                         jbe     short loc_45640D
seg005:004563F0
seg005:004563F2 ; seg005:004563F2
seg005:004563FC
seg005:004563FC loc_4563FC:     ; CODE XREF: sub_4563D9:loc_4563F9j
seg005:004563FC                         int     3       ; Trap to Debugger
seg005:004563FD                         mov     ax, 0FEh
seg005:00456401 ; seg005:00456401
seg005:0045640A
seg005:0045640A loc_45640A:     ; CODE XREF: sub_4563D9:loc_456407j
seg005:0045640A                 out     64h, ax         ; AT Keyboard controller 8042.
seg005:0045640A                                         ; Resend the last transmission
seg005:0045640A
seg005:0045640D ; seg005:0045640D
```

修复之后的代码

除了“sub eax, [esp-8+arg_4]”（实际上是 sub eax,[esp]）看起来有点怪之后，一切正常。作为一个壳，在解决了花指令之后，剩下的问题便只有反调试代码和解密（解压缩）代码了。例如上面列出的代码是通过时间校验检查调试器，一旦检查到，便使用特权级指令，让程序发生异常，无法继续运行下去。当然，我们在静态的环境下，反调试技巧对于我们来说，毫无意义。尽管如此，我们仍然需要知道程序会在什么时候运行到什么地方，最常见的利用系统的机制莫过于 SEH 了，现在来看看下面代码：

```
seg005:00456A9B                         call    $+5
seg005:00456AA0                         add     dword ptr [esp+0], 136Fh
seg005:00456AA7                         push    large dword ptr fs:0
seg005:00456AAE                         mov     large fs:0, esp
```

设置 SEH 的代码

“call $+5”指令后堆栈里的内容便是它的下一条指令在内存中的地址。这是病毒常用的重定位技巧。shift+/输入 0x00456AA0+0x136F 便能计算出异常处理函数的地址（457E0F）了。

```
seg005:0045745C                 xor         eax, eax
seg005:0045745E                 movzx       eax, byte ptr [eax]
```

产生异常的代码

现在我们应该跳到 457E0F 继续分析。我想你已经了解如何在静态环境下跟踪程序的流程，现在就让我们跟着程序的流程把解密相关的代码找出来。

```
seg005:00459191                 push        ecx
seg005:00459192                 xor         ecx, ecx
seg005:00459194                 call        $+5
seg005:00459199                 pop         edi
seg005:0045919A                 add         edi, 9C4h
seg005:004591A0                 pop         edx
seg005:004591A1                 add         edx, 15h
seg005:004591A4 loc_4591A4:                 ; CODE XREF: sub_459149+6Bj
seg005:004591A4                 movzx       eax, byte ptr [ecx+edi]
seg005:004591A8                 xor         eax, edx
seg005:004591AA                 mov         [ecx+edi], al
seg005:004591AD                 inc         ecx
seg005:004591AE                 cmp         ecx, 93h
seg005:004591B4                 jb          short loc_4591A4
```

解密代码

容易看出这就是解密代码，在循环之中，且有修改内存的指令。至于解密的 KEY，其实就是 00459191 处 ECX 的值+15h。我希望你还记得到达这里之前曾经看过下面代码：

```
seg005:004587B6                 mov         eax, [esp+0Ch]
seg005:004587BA                 xor         ecx, ecx
seg005:004587BC                 xor         ecx, [eax+4]
seg005:004587BF                 xor         ecx, [eax+8]
seg005:004587C2                 xor         ecx, [eax+0Ch]
seg005:004587C5                 xor         ecx, [eax+10h]
```

这一段是检查硬件断点的代码，假如没有设置硬件断点，那么 ECX 的结果应该是 0。假如你不能理解为什么，我建议你看看 SEH 以及关于反硬件断点的一些文章。
在知道解密代码的所有关键要素之后，就可以开始动手写脚本了。

```
#include "idc.idc"

static main() {
    auto StartAddr,cKey,Cbuffer,Counter;

StartAddr = 0x00459199 + 0x9c4;
cKey = 0x15;

for (Counter = 0 ; Counter < 0x93; Counter ++){
Cbuffer = Byte(StartAddr) ^cKey;                // movzx     eax, byte ptr [ecx+edi]
                                                 // xor       eax, edx
PatchByte(StartAddr,Cbuffer);                    // mov       [ecx+edi], al
StartAddr++;
                                    }
            }
```

Patch1.idc

在 00459BF7 和 0045B1FC 处可以看到类似的加密代码，就不把脚本给出来了，我把它放在附件中，分别为 PATCH2.idc 和 PATCH3.idc。在第三次解密之后，终于看到不同的解密代码了，代码比较多，我把隐藏区域的部分删掉：

```
seg005:00461F8D                 call    $+5
seg005:00461F92                 pop     ecx
seg005:00461F9D                 sub     ecx, 5
seg005:00461FAA                 xor     ebx, ebx
seg005:00461FB6                 mov     eax, 0BE9Ch
seg005:00461FC5                 mov     edi, ecx
seg005:00461FD1                 sub     edi, eax
seg005:00461FDD                 movzx   eax, byte ptr [edi]
seg005:00461FEA                 add     ebx, eax
seg005:00461FF6                 inc     edij
seg005:00462001                 cmp     edi, ecx
seg005:0046200D                 jb      short loc_461FDD
```

自校验代码

自校验代码的两个特征，一是读取代码，二是循环，对于那种单纯与校验结果比较控制流程的程序，我们是不需要理会自校验的。但是在这个例子里，紧跟后面的代码便是解密代码，并且自校验值作为解密 KEY，我们就得计算出它的校验值。

```
seg005:0046200F                 mov     edi, offset unk_447000
seg005:00462014                 mov     ecx, 0BC00h
seg005:00462019 ; seg005:00462019
seg005:00462023                 movzx   eax, byte ptr [edi]
seg005:00462030                 add     bl, bh
```

```
seg005:00462032                 xor         bl, bh
seg005:00462034                 xor         al, bl
seg005:00462040                 mov          [edi], al
seg005:0046204C                 inc         edi
seg005:00462057                 dec          ecx
seg005:00462062                 jnz         short loc_462019
```

自校验后的解密代码

相信有了前面的经验，要编写出以下脚本并不难。要注意的是由于之前修复花指令曾经修改过文件，因此在编写好脚本之后，必须重新加载程序，然后按顺序把解密脚本运行一次，确保解出正确的代码。此外还需注意下面代码：

```
seg005:00462064                 call        $+5
seg005:00462069                 pop          ecx
seg005:0046206A                 sub          [ecx+16h], ebx
seg005:0046206D                 popa
seg005:0046206E                 pusha
seg005:0046206F                 mov          esi, offset unk_447000
seg005:00462074                 lea          edi, [esi-46000h]
seg005:0046207A                 push         edi
seg005:0046207B                 or           ebp, 0FFFFFFFFh
seg005:0046207E                 push         offset sub_4528D0
seg005:00462083                 retn
```

自修改代码

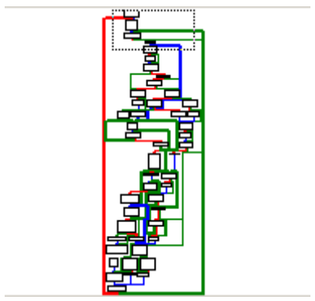
这里 0046206A 的代码实际就是以前面的校验值对 0046207E 处的指令修改，校验不正确便无法得出正确的返回地址。在写脚本的时候遇到一个问题是，解密代码使用了 BL 和 BH，即 BX 的低八位和高八位的寄存器。我们可以先将校验值写进一个 DWORD，然后获取其中第一个 BYTE 和第二个 BYTE，便可以得到它的值了。由此便可得出下面的脚本：

```
#include "idc.idc"

static main() {
        auto StartAddr,EndAddr,cKey,lKey,hKey,Cbuffer,Kbuffer,Counter;

    EndAddr = 0x00461F92 - 0x5;
    cKey = 0;

    for (StartAddr = EndAddr - 0x0BE9C; StartAddr < EndAddr; StartAddr ++){

    cKey = cKey + Byte(StartAddr);      // movzx     eax, byte ptr [edi]
                                        // add      ebx, eax
                                }

    Kbuffer = Dword(MinEA());           //从镜象基址借用 1 个 Dword
```

```
    PatchDword(MinEA(),cKey);
    lKey=Byte(MinEA());                    //转换成 bl
    hKey=Byte(MinEA()+1);                  //转换成 bh
    StartAddr = 0x447000;

    for (Counter = 0x0BC00 ; Counter !=0 ; Counter --){

    lKey=lKey + hKey;                      // add        bl, bh
    lKey=lKey ^ hKey;                      // xor        bl, bh
    Cbuffer = Byte(StartAddr) ^lKey;       // movzx      eax, byte ptr [edi]
                                           // xor        al, bl
    PatchByte(StartAddr,Cbuffer);          // mov        [edi], al
    StartAddr++;
                                    }

    StartAddr = 0x462069+0x16;
    PatchByte(MinEA(),lKey);
    cKey = Dword (MinEA());
    Cbuffer = Dword(StartAddr) - cKey;
    PatchDword(StartAddr,Cbuffer);
    PatchDword(MinEA(),Kbuffer);                  //恢复原来的数据
        }
```

Patch4.idc

在还原代码之后，容易看出 0046207E 处，PUSH + RET 相当于一个绝对跳转，现在让我们看看 4528D0 处的代码。在 4528D0 处按 P，IDA 将认为该处为函数的起点，并为函数建立图形视图。



流程的缩略图

看起来很复杂。或者的确复杂，但是我们只需要将它还原成 IDC 代码就可以了，甚至不需要我们理解算法的思想。可能你觉得在去除花指令影响之后，用 OD 改 EIP 直接运行相关代码也可以，内联汇编，写插件也可以。实际工作的时候，当然效率优先，选择最高效率的方法，但是将低级语言代码还原成高级语言代码，还是有一定意义的，例如你觉得 C 代码更容易理解一点，那么你可以先把汇编转成 C 代码，再理解。现在让我们切换到反汇编窗口再看代码：

```
seg001:004528D0                 jmp         short loc_4528E2    //跳到开始位置
seg001:004528D0
seg001:004528D2 ; ---------------------------------------------------------------------------
seg001:004528D2                 nop
seg001:004528D3                 nop
seg001:004528D4                 nop
seg001:004528D5                 nop
seg001:004528D6                 nop
seg001:004528D7                 nop
seg001:004528D7
seg001:004528D8
seg001:004528D8 loc_4528D8:               ; CODE XREF: sub_4528D0:loc_4528E9j
seg001:004528D8                 mov         al, [esi]             ; 1
seg001:004528DA                 inc         esi
seg001:004528DB                 mov         [edi], al
seg001:004528DD                 inc         edi
seg001:004528DD
seg001:004528DE
seg001:004528DE loc_4528DE:             ; CODE XREF: sub_4528D0+BAj
seg001:004528DE                                     ; sub_4528D0+D1j
seg001:004528DE                 add         ebx, ebx
seg001:004528E0                 jnz         short loc_4528E9
seg001:004528E0
seg001:004528E2
seg001:004528E2 loc_4528E2:                     ; CODE XREF: sub_4528D0j
seg001:004528E2                 mov         ebx, [esi]    //从这里开始
seg001:004528E4
seg001:004528E4 loc_4528E4:
seg001:004528E4                 sub         esi, -4
seg001:004528E7                 adc         ebx, ebx
seg001:004528E7
seg001:004528E9
seg001:004528E9 loc_4528E9:                     ; CODE XREF: sub_4528D0+10j
seg001:004528E9                 jb          short loc_4528D8
```

我们发现开始的地方需要访问 ESI 指向的内存，往回看发现解密代码需要的参数，在前面说的自修改代码部分（0046206F）已经处理过了。该处代码很容易转成高级语言，现在来看看如何重整代码的流程。跳转向上的时候，代表一个循环。这与高级语言是相通的，值得注意的是向下的跳转。达到某一条件，就绕过一部分代码，向后执行，这跟高级语言中的 IF 控制语句，即遇到某一条件就执行随后的代码。也就是说，我们得反转比较条件。

以给出的代码为例子，与自身相加，相当于乘 2，实际就是以个向左位移操作。想想十进制中，把 1 向左移动一位，实际就是将 1 乘以 10。在二进制中也是一样，将一个二进制数向左移动一位，则是乘以 2。汇编指令 jb 仅在进位标记 CF=1 时跳转，也就是说 004528E7 处的 adc　ebx, ebx 及后面的 jb　short loc_4528D8 的意义为，将 EBX 中的数向左移一位，并检查的最高位是否为 1，1 则向上跳转，也就是循环，0 则继续执行，即终止循环的条件。现在我们可以构造下面循环的框架：

```
auto EBX,HigtBitfla;

while (HigtBitflat != 0){

HigtBitflat = EBX & 0x80000000;       //与 0x80000000 进行 and 运算
                                        //最高位不为 0 则 HigtBitflat 为 0
                                        //0x80000000 最高位为 1，其他位 0
                                        //不明白的读者看将其展开计算
EBX = EBX + EBX;                      //向左位移

                            }
```

现在再来看看 004528DE 处的代码，jnz 在 ZF=0 时产生跳转，即当最高位之外任意一位不为 0 时产生跳转。正如上面说的，将跳转条件反转，我们便能使用 IF 语句了。

```
Auto EBX,IsNotZero;

IsNotZero = EBX & 0x7FFFFFFF;       //0x7FFFFFFF 最高位为 0
                                        //屏蔽最高位，以检查后面的位
                                        //仅当最高位外全为 0，IsNotZero 为
0
If （IsNotZero == 0）{
//   此处可以填上 004528E2 到 004528E7 的代码
                                }
EBX = EBX + EBX;                        //注意这里与汇编的区别
                                        //先判断，然后才移位
```

注意这里与汇编代码的区别，由于我们无法在 IDC 上访问标记寄存器，也无法使用跳转。这里只能先判断最高位，然后才进行位移。下面让我们来直接看最后得出的 IDC 脚本：

```
#include "idc.idc"

static main() {
   auto MyAddr,DeCodeAddr,HigtBitflat,EBX;
auto EAX,ECX,EBP,ESI,EDX,CF,IsNotZero,Counter;

MyAddr = 0x447000;
DeCodeAddr =0x447000 - 0x46000;
ESI=DeCodeAddr;
Counter = 0;            //初始化循环条件
CF = 0;                   //代表标志寄存器的 CF 位
EBX = Dword(MyAddr);
MyAddr = MyAddr + 4;
HigtBitflat = EBX & 0x80000000;
```

```
    EBX = EBX + EBX;
    EBX++;
//    为了统一循环入口，将部分代码移出循环执行。
    while (Counter != 1){
    while (HigtBitflat != 0){
        PatchByte (DeCodeAddr,Byte(MyAddr));
        MyAddr++;
        DeCodeAddr++;
        IsNotZero = EBX & 0x7FFFFFFF;
        if (IsNotZero == 0){
            CF=1;     //sub esi, -4 与 add esi,4 的区别就是前者 CF=1
            EBX = Dword(MyAddr);
                MyAddr = MyAddr + 4;
                    }
        HigtBitflat = EBX & 0x80000000;
        EBX = EBX + EBX;
            EBX = EBX + CF;          //加上 CF，模拟 ADC 指令
        CF = 0;
                                }

    EAX = 1;
    while (Counter != 1){
//4528F0 到 45291A，以 JMP 构成一个循环。因此使用 while 语句，构造
//一个无限循环。在符合终止循环条件处使用 break 指令结束循环。
    IsNotZero = EBX & 0x7FFFFFFF;
    if (IsNotZero == 0){
        CF=1;
        EBX = Dword(MyAddr);
            MyAddr = MyAddr + 4;
                }
    HigtBitflat = EBX & 0x80000000;
    EBX = EBX + EBX;
    EBX = EBX + CF;
    CF = 0;
    EAX = EAX + EAX;
    if (HigtBitflat != 0) EAX++;
    HigtBitflat = EBX & 0x80000000;
    if (HigtBitflat != 0){
        IsNotZero = EBX & 0x7FFFFFFF;
        if (IsNotZero != 0) {    //00452901     jnz    short loc_45291C
        EBX = EBX + EBX;
        break;
```

```
                        }
        EBX = Dword(MyAddr);
        MyAddr = MyAddr + 4;
        HigtBitflat = EBX & 0x80000000;
        if (HigtBitflat != 0) {         //0045290A     jb    short loc_45291C
        EBX = EBX + EBX;
        EBX++;
        break;
                        }
        CF = 1;
                            }
EBX = EBX + EBX;
EBX = EBX + CF;
CF = 0;
EAX--;
IsNotZero = EBX & 0x7FFFFFFF;
if (IsNotZero == 0){
        CF=1;
        EBX = Dword(MyAddr);
              MyAddr = MyAddr + 4;
                    }
HigtBitflat = EBX & 0x80000000;
EBX = EBX + EBX;
EBX = EBX + CF;
CF = 0;
EAX = EAX + EAX;
if (HigtBitflat != 0) EAX++;
                                        }

ECX = 0;              //xor ecx,ecx 常见的为寄存器赋值为 0 的语句。
//注意 00452921    jb    short loc_452934 处，程序分开两条路线
//在 loc_45293F 处汇合。因此这里使用 if  。。else 语句重整程序流程。
if (EAX < 3){       //此处直接使用减法指令作比较，而不是使用 CMP
EAX = EAX - 3;    //因此只能在比较之后再减
IsNotZero = EBX & 0x7FFFFFFF;
if (IsNotZero == 0){
        CF=1;
        EBX = Dword(MyAddr);
              MyAddr = MyAddr + 4;
                    }
HigtBitflat = EBX & 0x80000000;
EBX = EBX + EBX;
```

```
EBX = EBX + CF;
CF = 0;
                                }
else{
EAX = EAX - 3;
EAX = EAX << 8;
EAX = EAX + Byte(MyAddr);
MyAddr++;
EAX = EAX ^ 0xffffffff;
if (EAX == 0) break;
HigtBitflat = EAX & 1;    //检查 sar eax,1 是否影响 CF 位
EAX = EAX >> 1;           //检查结束再执行位移
EBP = EAX;
        }
ECX = ECX + ECX;
if (HigtBitflat != 0) ECX++;
IsNotZero = EBX & 0x7FFFFFFF;
if (IsNotZero == 0){
      CF=1;
      EBX = Dword(MyAddr);
            MyAddr = MyAddr + 4;
                  }
    HigtBitflat = EBX & 0x80000000;
    EBX = EBX + EBX;
    EBX = EBX + CF;
    CF = 0;
    ECX = ECX + ECX;
    if (HigtBitflat != 0) ECX++;
    if (ECX == 0 ){
    ECX++;
    HigtBitflat = 0;
//00452960     jnb         short loc_452951
//0045296B     jnb         short loc_452951
//此处有两个跳转指向循环入口，将 00452960 处的条件反转，翻译成 if
//语句。便可得到下面循环：
    while (HigtBitflat == 0){
    IsNotZero = EBX & 0x7FFFFFFF;
    if (IsNotZero == 0){
          CF=1;
          EBX = Dword(MyAddr);
                MyAddr = MyAddr + 4;
                      }
```

```
HigtBitflat = EBX & 0x80000000;
EBX = EBX + EBX;
EBX = EBX + CF;
CF = 0;
ECX = ECX + ECX;
if (HigtBitflat != 0) ECX++;
HigtBitflat = EBX & 0x80000000;
if (HigtBitflat != 0){
IsNotZero = EBX & 0x7FFFFFFF;
if (IsNotZero != 0) {
EBX = EBX + EBX;
break;
}
EBX = Dword(MyAddr);
MyAddr = MyAddr + 4;
CF = 1;
HigtBitflat = EBX & 0x80000000;
                    }
EBX = EBX + EBX;
EBX = EBX + CF;
CF = 0;
                                    }
ECX = ECX + 2;
                    }
//高级语言的比较为有符号数的比较，而 0045297F    jbe    short
loc_452990
//是无符数的比较。因此要先比较其最高位，模拟无符号数的比较
HigtBitflat = EBP & 0x80000000;
if (HigtBitflat !=0){
if (EBP < 0xfffffb00) CF =1;
                    }
else{
CF =1;
      }
ECX ++;
ECX = ECX + CF;
CF=0;
EDX = DeCodeAddr + EBP;
if (HigtBitflat !=0){
if (EBP > -4) CF=1;
                    }
//0045297F    jbe    short loc_452990 将此处分开两条路线，
```

```
//以 jmp    loc_4528DE 重新汇合。这里同样使用 if….else 语句。
      if (CF==1){
      CF=0;
      while (ECX !=0){
      PatchByte(DeCodeAddr,Byte(EDX));
      EDX ++;
      DeCodeAddr ++;
      ECX --;
                  }
                  }
      else{
      while(Counter != 1){
      PatchDword(DeCodeAddr,Dword(EDX));
      EDX = EDX + 4;
      DeCodeAddr = DeCodeAddr + 4;
      if (ECX <= 4){
      ECX= ECX -4;
      break;
      }
      ECX = ECX - 4;
      }
      DeCodeAddr = DeCodeAddr + ECX;
            }
//反汇编代码的循环入口（4528DE）与我们转换的循环入口不同
（4528E9）
//跟开始的时候一样，入口之前的代码放到循环外面。
      IsNotZero = EBX & 0x7FFFFFFF;
      if (IsNotZero == 0){
            CF=1;
            EBX = Dword(MyAddr);
                  MyAddr = MyAddr + 4;
                        }
      HigtBitflat = EBX & 0x80000000;
      EBX = EBX + EBX;
      EBX = EBX + CF;
      CF = 0;
                              }
                  }
```

Patch5.idc

至此，我们成功将 004528D0 到 004529A1 处的代码转换成 C 代码。在完成如此复杂的代码还原之后，004529A6 到 004529D8 处的反汇编代码只是小菜一碟。里面的代码也很好理解，将符合 E8 01 和 E9 01 的机械码解密。位移指令可以通过借用程序中的一个闲置的 Dword，使用 IDC 提供的 Pactch 系列指令来模拟,详见 Patch6.idc。在完成最后的解密代码后，便是 IAT 的修复了。现在看看下面代码：

```
004529DA                lea     edi, [esi+50000h]
004529E0 loc_4529E0:
004529E0                mov     eax, [edi]
004529E2                or      eax, eax
004529E4                jz      short loc_452A22
004529E4
004529E6                mov     ebx, [edi+4]
004529E9                lea     eax, [eax+esi+549B0h]
004529F0                add     ebx, esi
004529F2                push    eax
004529F3                add     edi, 8
004529F6                call    dword ptr [esi+54A3Ch]
004529FC                xchg    eax, ebp
004529FD loc_4529FD:
004529FD                mov     al, [edi]
004529FF                inc     edi
00452A00                or      al, al
00452A02                jz      short loc_4529E0
00452A02
00452A04                mov     ecx, edi
00452A06                push    edi
00452A07                dec     eax
00452A08                repne scasb
00452A0A                push    ebp
00452A0B                call    dword ptr [esi+54A40h]
00452A11                or      eax, eax
00452A13                jz      short loc_452A1C
00452A13
00452A15                mov     [ebx], eax
00452A17                add     ebx, 4
00452A1A                jmp     short loc_4529FD
```

在分析该处代码之前，显然应该先把 ESI 的值计算出来。鼠标点击 ESI，以高亮显示该寄存器，向上滚动反汇编窗口，发现从 004529A6　pop　esi 处开始，ESI 便没有被修改过，而该处对应于：

```
seg005:0046206F                 mov     esi, offset unk_447000
seg005:00462074                 lea     edi, [esi-46000h]
seg005:0046207A                 push    edi
```

可见 ESI=0x401000，容易计算出 004529F6 和 00452A0B 处 CALL 的地址分别为 455A3Ch 和 455A40h。跳转到该地址：

```
00455A3C ; HMODULE __stdcall LoadLibraryA(LPCSTR lpLibFileName)
00455A3C                extrn LoadLibraryA:dword
00455A40 ; FARPROC __stdcall GetProcAddress(HMODULE hModule,LPCSTR lpProcName)
00455A40                extrn GetProcAddress:dword
```

显然，这里便是壳填充 IAT 的地方了。那么 004529DA　lea　edi, [esi+50000h]中，EDI 便是保存 API 名字的数据表。做脱壳机的任务就留给读者作课后练习，正如前面介绍的那样，只需要 API 的名字为相关 IAT 地址重命名，便能分析了。也就是说 00452A0B 处，调用 GetProcAddress，跟踪它的参数 lpProcName (00452A06　push　edi)，以及它的返回值（00452A15　mov　[ebx], eax），当然这里的跟踪，可以象刚才那样手动确认，也可以通过与调试器配合快速得出结果。不难得出下面脚本：

```
#include "idc.idc"

static main() {
    auto ESI,EDI,EAX,EBX,Counter,cBuffer,BufLen,straa;

ESI = 0x447000 - 0x46000;
EDI = ESI + 0x50000;
Counter = MaxEA() - MinEA();
MakeUnknown(MinEA(),Counter,1);     //将整个程序标记未分析
AnalyzeArea (MinEA(),MaxEA());        //分析整个程序
Counter = 0;
while (Counter != 1){
EAX = Dword(EDI);
if (EAX == 0) break;
EBX = Dword(EDI+4);
EBX = EBX + ESI;
EDI = EDI + 8;
while (Counter != 1){
EAX = Byte(EDI);
EDI++;
if (EAX == 0) break;
cBuffer = GetString(EDI,-1,ASCSTR_C);
straa = cBuffer + "_";       //IDA 不允许重复命名，加上“_”避免重复
MakeNameEx(EBX,straa,SN_AUTO);
EBX = EBX + 4;
EDI = EDI + strlen(cBuffer);
EDI++;
                    }
                    }
}
```

IATPATCH.idc

注意解密后，必须将整个程序标记为未分析，并重新分析，然后才能进行重命名。

```
push    0               ; lpModuleName
call    j_GetModuleHandleA_

mov     hInstance, eax
call    j_InitCommonControls_

push    0               ; dwInitParam
push    offset loc_413D84 ; lpDialogFunc
push    0               ; hWndParent
push    65h             ; lpTemplateName
push    hInstance       ; hInstance
call    j_DialogBoxParamA_

push    0               ; uExitCode
call    j_ExitProcess_
```

程序的 OEP

到此，静态脱壳完毕。从这个例子也可以知道，对于掌握反汇编器的人来说，除非反调试机制与解密 KEY 关联，否则根本就没有强度可言。然而，IDA 博大精深，还有更多强大的功能，本文也只是抛砖引玉而已。下面给出几个链接，方便大家更进一步学习：

IDA 的官方网站：
www.datarescue.com

看雪论坛 9 月翻译专题：
http://bbs.pediy.com/showthread.php?s=&threadid=31023

IDA Pro 的插件开发 SDK：
http://bbs.pediy.com/showthread.php?s=&threadid=31441

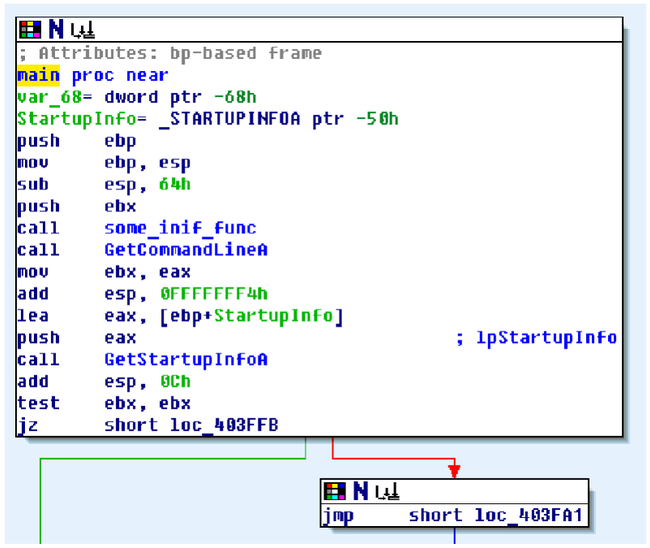
IDA 逆向工程入门：
http://bbs.pediy.com/showthread.php?s=&threadid=40765

IDA 简易教程：
www.pediy.com/practise/IDA.htm

http://www.pediy.com/kssd/pediy09/pediy09-265.htm

利用 IDA Pro 反汇编程序

IDA Pro 是一款强大的反汇编软件，特有的 IDA 视图和交叉引用，可以方便理解程序逻辑和快速定位代码片断，以方便修改。



IDA 视图

示例程序

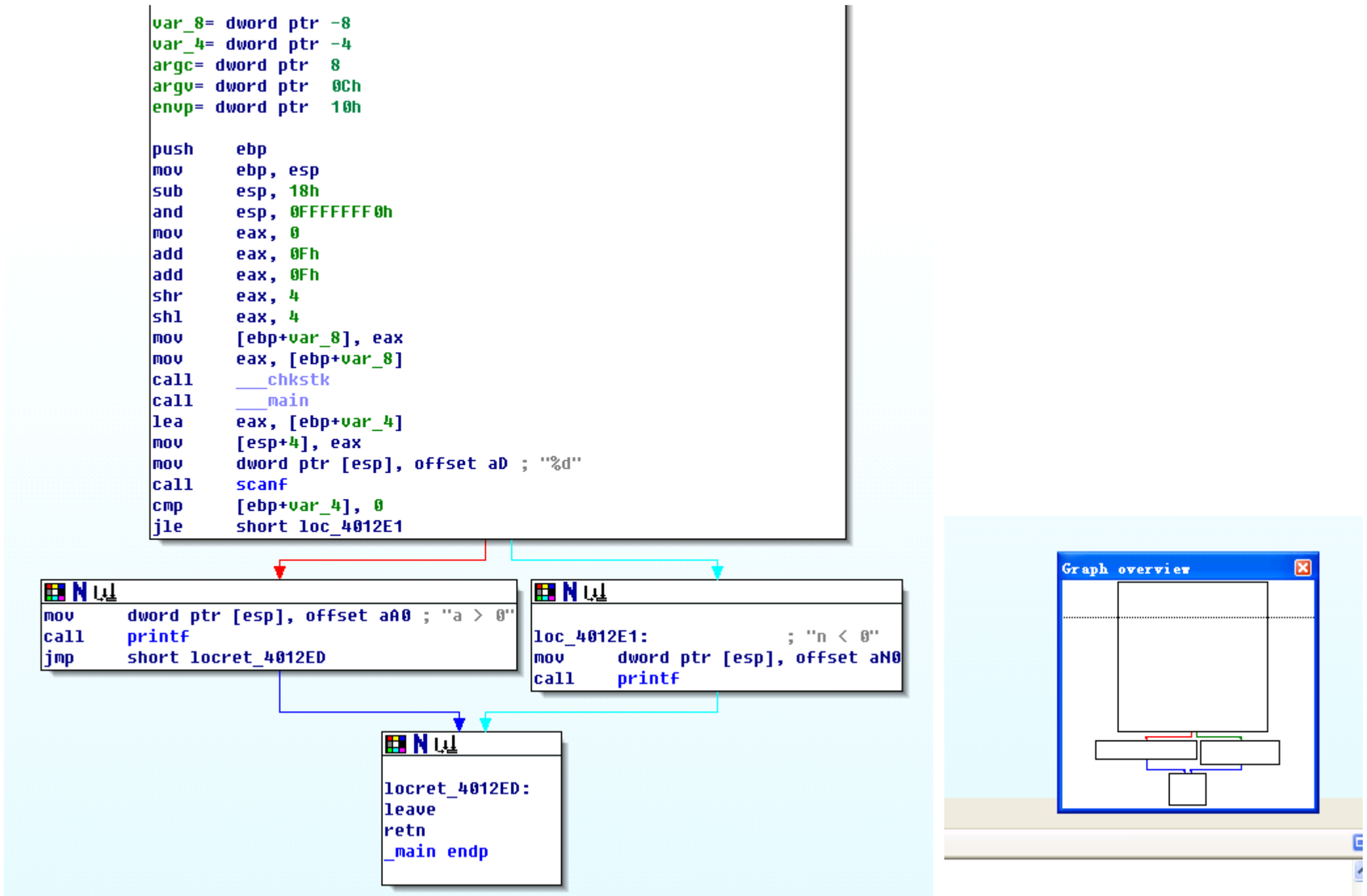
下面会通过修改示例程序的输出字符串，来讲解如何使用 IDA Pro。

```
#include

main()
{
    int n;
    scanf ("%d",&n);
    if (n > 0)
       printf("a > 0");  //后面会用 IDA Pro 把'a'改成'n'
    else
       printf("n < 0");
}
```

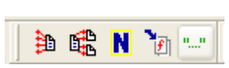
编译后的程序下载：demo

运行 IDA Pro

运行 IDA Pro，并使用 PE 文件的方式打开示例的 test.exe 文件。IDA Pro 会新建一个工程，并开始反汇编程序。反汇编完成后，在[IDA-View]窗口中，可以看到程序逻辑的树形图，如下：



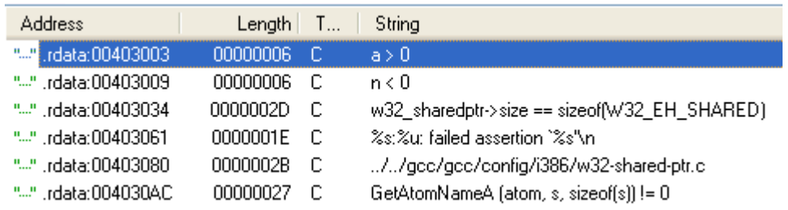
树形图把条件分支清晰地显示出来了，**绿色线**连着的表示条件为 true 时执行的逻辑，而**红色线**表示条件为 false 时执行的逻辑。**右下角**有 IDA 视图的缩略图，在上面点击可以快速定位到视图的指定位置。 IDA 的工具栏有几个按钮对定位代码很重要，如下图所示：



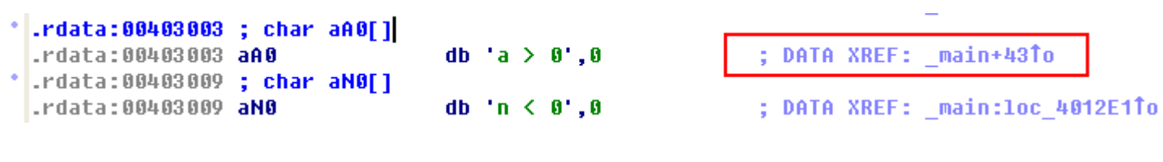
从左到右分别是： Open exports window:打开导出窗口 Open import window:打开导入窗口 *Open names window:函数和参数的命名列表 *Open functions window:程序调用的所有函数窗口 *Open strings window: 打开字符串显示窗口，会列出程序中的所有字符串，该窗口有助于你通过程序的运行输出逆向找出对应的代码片断。

定位代码片断

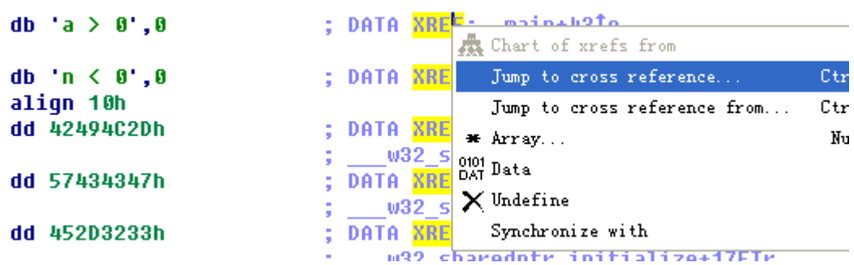
假设我们现在接到个任务，需修正程序，把输出“a > 0”修正为“n > 0”。示例程序比较简单，直接看 IDA 视图我们就能找到需修改的代码片断，但实际处理时，可能程序有几 m 大，通过一个个看 IDA 视图已没法有效找到相关的 执行代码片断，这时怎么办？ 使用字符串窗口和 IDA 强大的交叉引用！ 点击工具栏的[Open strings windows]按钮，可以看到如下的程序字符串：

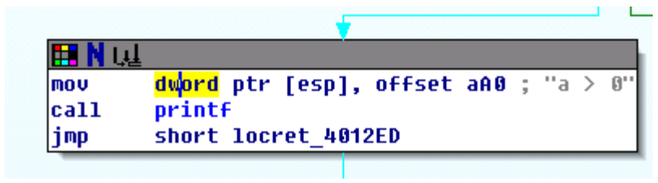


程序的字符串较少，可以很快地看到我们需要的字符串“a > 0”在数据段 00403003 位置。假如字符串多到已不能肉眼定位查找，因为字符串窗口是没有查找功能的，这时需要借助其他的文本编辑器，如 notepad，editplus 等。在字符串窗口内右键，选择菜单[copy]命令，会把字符串窗口的所有内容复制到剪贴板，再粘贴到记事本中查找就可以了。 **双击**字符串窗口的该行字符串，会跳转到 IDA 视图的 **00403003 位置，如下图所示：**

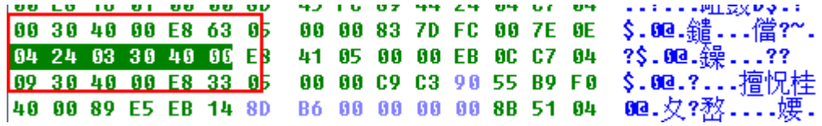


该位置的字符串后面会注释有 DATA XREF 的字样，这是程序中引用到该字符串的代码片断的地址！在该行上右键，选择[Jump to cross reference...]项，会立即跳转到引用该字符串的代码片断位置！





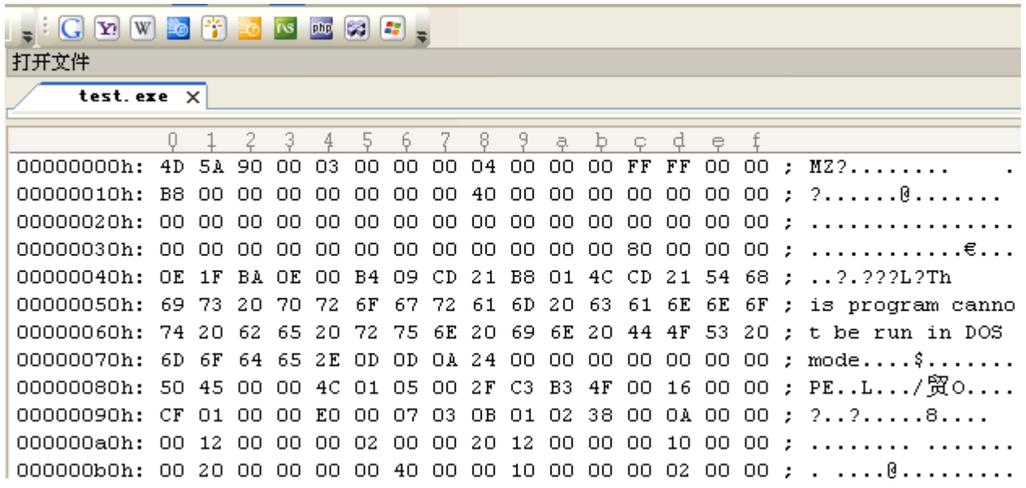
最后定位的代码片断 上图显示的汇编指令即是我们要找的代码片断，这时点击[Hex View-A]窗口，会切换到二进制浏览模式，并高亮了汇编代码的二进制格式指令，如下图所示：



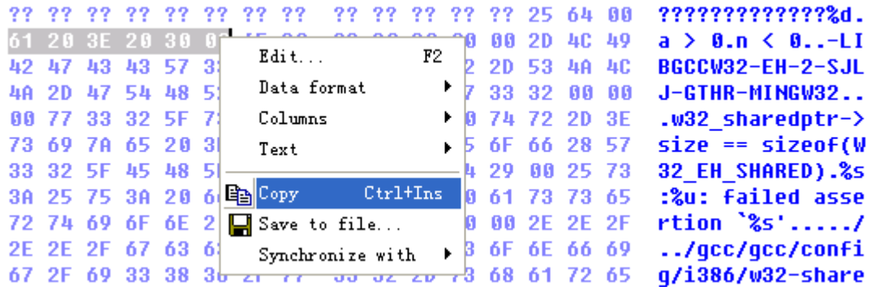
已找到需修改的代码片断，剩下的只需把 a 改成 n。

修改程序文件

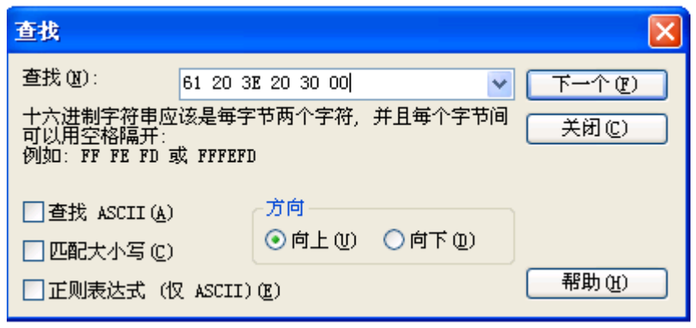
在 IDA 中，可以在[Hex View-A]窗口右键选择[Edit]来修改二进制指令。修改后通过右键选择[Commit Change]可以看到修改后的 IDA 视图。但需要注意的是，这种方式的修改并不会更新原始程序文件，实际只是修改了 IDA 的项目文件！IDA 中只适合做 一些验证性的修改，确保正确后再使用其他工具修改原始程序文件。 在 IDA 中验证修改正确后，可以使用 UltraEdit 或 Hex Workshop 来修改原始程序文件。下面会以 UltraEdit 为例来说明如何修改。



用 UltraEdit 直接打开程序文件，UltraEdit 会以 16 进制模式显示程序文件。**UltraEdit 显示的地址和 IDA 显示的地址是不同的**，为了找到对应代码片断在 UltraEdit 中的实际地址，需要使用到 UltraEdit 的查找功能。在 IDA 中复制需修改的 16 进制模式显示的指令，在 UltraEdit 中打开查找，粘贴并查找该 16 进制字符串，UltrEdit 会很快定位到该指令处，如下图所示：



在 IDA 中使得右键来复制



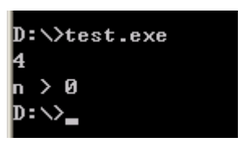
在 UltraEdit 打开查找功能

```
00000ff0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00001000h: 25 64 00 61 20 3E 20 30 00 6E 20 3C 20 30 00 00 ; %d.a > 0.n < 0..
00001010h: 2D 4C 49 42 47 43 43 57 33 32 2D 45 48 2D 32 2D ; -LIBGCCW32-EH-2-
00001020h: 53 4A 4C 4A 2D 47 54 48 52 2D 4D 49 4E 47 57 33 ; SJLJ-GTHR-MINGW3
00001030h: 32 00 00 00 77 33 32 5F 73 68 61 72 65 64 70 74 ; 2...w32_sharedpt
00001040h: 72 2D 3E 73 69 7A 65 20 3D 3D 20 73 69 7A 65 6F ; r->size == sizeo
```

找到了 UltraEdit 的对应位置 现在我们要把“a > 0”改成“n > 0”，a 对应的 ASCII 码是 61，而 n 对应的 ASCII 码是 6E，只需把 61 改成 6E 就可以了，修改后保存。

```
00001000h: 25 64 00 6e 20 3E 20 30 00 6E 20 3C 20 30 00 00 ; %d.n > 0.n < 0..
```

再次运行，可以看到结果已改变！



示例只是修改了字符串，只需更改数据段内容就可以了，不用更改指令。假如需要更改指令，需要参考< a href="http://courses.engr.illinois.edu/ece390/resources /opcodes.html#Main">8086 指令操作表写出对应指令的 16 进制形式，再修改。

参考资料：

http://blog.csdn.net/liquanhai/article/details/5479141 http://www.youtube.com/watch?v=Gl2S0YPRb9s http://www.woodmann.com/crackz/Tutorials/Flores1.htm http://courses.engr.illinois.edu/ece390/resources/opcodes.html#Main
http://faydoc.tripod.com/cpu/conventions.htm
http://www.cnblogs.com/vento/archive/2013/02/09/2909579.html

IDA 的 F5 功能真的好强大

今天在 pediy 看到一贴子，自己用 IDA F5 试了一下，真的好强大。

IDA 载入目标程序后，按 F5 键，得到伪代码，置于 VC6 编译器中，添加相关头文件，稍作修改，编译得到注册机。

下面是 IDA 的 F5 键得到的：

```
int __cdecl main(int argc, const char **argv, const char *envp)
{ signed int v4;
// [sp+1Ch] [bp-3Ch]@1
char v5;
// [sp+30h] [bp-28h]@1
unsigned int v6;
// [sp+2Ch] [bp-2Ch]@1
unsigned int v7;
// [sp+28h] [bp-30h]@1
v4 = 16;
__main();
printf("Enter your login:");
gets(&v5);
printf("Enter password:");
scanf("%ld", &v6);
v7 = 32 * strlen(&v5) + 2226449;
if ( v6 == v7 )
printf("Good job!, now make a keygen\n");
```

```
else

printf("Keep trying you'l get it");

sub_401980(10);

return 0;

}
```

然后我们制作注册机：

```
//-------------------------------------
//IDA 不会加上头文件的，呵呵，我们加上。
//然后把没有用的两句注释掉。就可编译通过，与原 crackme 一样的。
#include<stdio.h>
#include<string.h>
//-------------------------------------
int __cdecl main(int argc, const char **argv, const char *envp)
{
signed int v4; // [sp+1Ch] [bp-3Ch]@1
char v5; // [sp+30h] [bp-28h]@1
unsigned int v6; // [sp+2Ch] [bp-2Ch]@1
unsigned int v7; // [sp+28h] [bp-30h]@1

v4 = 16;
// __main();
printf("Enter your login:");
gets(&v5);
printf("Enter password:");
scanf("%ld", &v6);
v7 = 32 * strlen(&v5) + 2226449;
//-------------------------------------
//加上这一句，让我们看看正确的注册码：
printf("the sn is: %d\n",v7);
//-------------------------------------
if ( v6 == v7 )
    printf("Good job!, now make a keygen\n");
else
    printf("Keep trying you'l get it");
// sub_401980(10);
return 0;
}
```

//输入用户名：diype
//我们会输出序列号：2226609
//注册码只与注册名的位数有关，晕，这也叫 crackme ？？
//IDA 的 F5 功能真的好强大！！

http://hi.baidu.com/npzsqdjvbmelrte/item/8e874f9bff48b7f7291647a1

[IDA] 分析 for 循环的汇编代码

在程序设计语言里面，循环是三种语言流程之一（顺序，分支，循环），这其中循环又是编程这件事中最具魅力的，它发挥了人在思维和计算机在计算方面 的双方优势，体现了程序员的技巧和智慧，也体现了代码的简洁，优雅和优美。循环中最常用的应该是 for 循环，其他几种例如 while，do while 又基本上可以等效写成 for 循环。同时 for 循环又可以被等效改写为递归函数。本文首先通过 VC 创建一个含有 for 循环的简单函数的工程，然后用 IDA 工具分析其汇编代码。

for 循环主要由以下形式组成，例如：

```
for( i = 0; i < imax; i++) { ... }
```

可以将其看做有四个基本部分构成，即初始化语句（ i = 0 ），条件语句（i < imax），主体后续语句 （i++），和供循环重复执行的主体（{。。。}）。首先用 VC 创建一个 Win32 Console 程序，我们输入一个含有 for 循环的简单函数，代码如下：

```
#include "stdafx.h"
#include <string.h>

int getstring(char* s)
{
   int length = strlen(s);
   int i;
   for(i=0; i<length; i++)
   {
      //取余（%）优先级比加减（+-）高！
      s[i] = (s[i] - 'a' + 1)%26 + 'a';
   }
   return length;
}
```

```
int main(int argc, char* argv[])
{
   char s[] = "abcdefg\0";
   int result = getstring(s);
   printf("%s\n", s);
   return 0;
}
```

简要介绍以下上面的代码，函数 getstring 用于对输入的字符串（假设输入的字符串全部有小写字母 a-z 组成），把字符串中每个字符改为其下一个英文字符（例如 a 改为 b，b 改为 c，...，z 改为 a）。因此上面的程序我们在 main 里面初始化一个字符串为“abcdefg”，输出“bcdefgh”。

下面我们用 IDA 查看其反汇编代码（win32 debug）：在汇编代码中，getstring 函数和 main 函数和我们的 C++代码中出现的顺序相同，可以看到：

getstring 函数从 00401020H ~ 00401095H （代码量： 118 bytes，按 16bytes 对齐后实际占据 160bytes）；

main 函数从 004010C0H ~ 00401124H （代码量：101 bytes， 对齐后实际占据 128 bytes）；

在函数和函数之间，用 0xCC 进行填充，以使函数开始地址都位于 16bytes 整数倍处；

我们先简要看下 main 函数，很显然，字符串 s 是在 main 函数内 的栈上空间：汇编代码如下：

```
.text:004010C0 main          proc near           ; CODE XREF: j_mainj
.text:004010C0
.text:004010C0 var_50        = dword ptr -50h
.text:004010C0 var_10        = dword ptr -10h
.text:004010C0 var_C         = dword ptr -0Ch
.text:004010C0 var_8         = dword ptr -8
.text:004010C0 var_4         = byte ptr -4
.text:004010C0
.text:004010C0               push    ebp
```

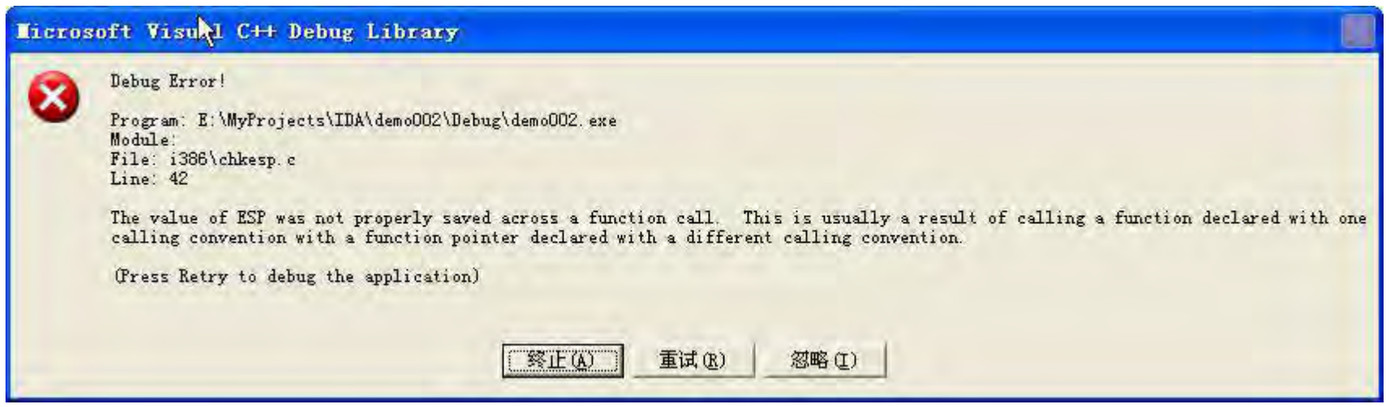
```
.text:004010C1          mov     ebp, esp

                ;main 在栈上申请了 80 bytes 临时空间
.text:004010C3          sub     esp, 50h ; main 在栈上申请了 80 bytes 临时空间
.text:004010C6          push    ebx     ; 保存寄存器当前值（esp 继续减小，存储于 80bytes 相邻的低地址处）
.text:004010C7          push    esi
.text:004010C8          push    edi

                ;把 80bytes 临时空间初始化成全部用 0xCC 填充
.text:004010C9          lea     edi, [ebp+var_50]
.text:004010CC          mov     ecx, 14h
.text:004010D1          mov     eax, 0CCCCCCCCh
.text:004010D6          rep stosd           ; stosd：串存储

                  ;把.rdata 中的数据放到 char s[]，s: ebp + var_C
                ;eax <- "abcd"
.text:004010D8          mov     eax, ds:??_C@_08PIFP@abcdefg?$AA?$AA@
.text:004010DD          mov     [ebp+var_C], eax

                ;ecx <- "efg0"
.text:004010E0          mov     ecx, ds:dword_422024
.text:004010E6          mov     [ebp+var_8], ecx
.text:004010E9          mov     dl, ds:byte_422028
.text:004010EF          mov     [ebp+var_4], dl

                ;调用 getstring(char* s);
.text:004010F2          lea     eax, [ebp+var_C]
.text:004010F5          push    eax
.text:004010F6          call    j_getstring
.text:004010FB          add     esp, 4  //调用方复原堆栈。
.text:004010FE          mov     [ebp+var_10], eax ;result = getstring(s);

                ;调用 printf("%s", s);
.text:00401101          lea     ecx, [ebp+var_C]
.text:00401104          push    ecx
.text:00401105          push    offset ??_C@_03HHKO@?$CFs?6?$AA@
.text:0040110A          call    printf
.text:0040110F          add     esp, 8 ;复原堆栈
.text:00401112          xor     eax, eax
.text:00401114          pop     edi
.text:00401115          pop     esi
.text:00401116          pop     ebx  ;复原寄存器内容（导致 esp 增加）
```

```
                释放栈上申请的 80 bytes  临时空间
.text:00401117          add     esp, 50h

                ;检查栈是否被破坏？
.text:0040111A          cmp     ebp, esp
.text:0040111C          call    __chkesp
.text:00401121          mov     esp, ebp
.text:00401123          pop     ebp
.text:00401124          retn
.text:00401124 main     endp
```

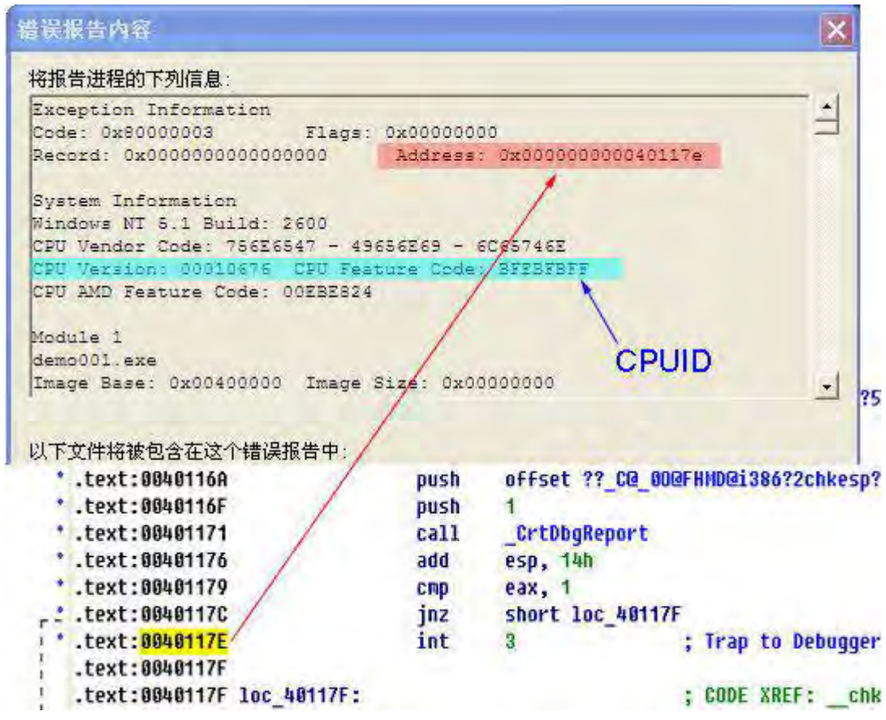

main 函数并不是重点，不过我们还是可以看出一些基本的东西，比如函数是如何利用栈的。从上面的代码可以看出，进入函数的第一件事是把 esp 复制给 ebp，这相当于对刚进入函数时刻的栈做了一个“快照”（由于 esp 是始终处于动态变化的，所以需要缓存进入函数时刻的栈顶地址）。此后对参 数的访问都通过“快照” ebp 完成，即（ ebp + 参数偏移量）去访问参数， 栈由调用方负责复原（复原 esp 的值，因为下一条指令地址和参数等信息入栈（push）时 esp 会减小），这属于调用约定的范畴。

然后函数一次性“申请”到足够函数内的临时变量使用的栈上空间（在 main 函数里是 80bytes），并把他们初始化全部用 0xCC 填充。此后的函数内部声明的那些临时变量就都处于在这一块栈上的空间之内。在函数返回之前，这一块空间将被函数“释放”（**【备注】请注意这里的申请和释放指的是对 esp 即栈顶地址的加减操作，减对应于申请，加对应释放，请注意和堆上的内存管理并不是相同概念**）。 例如我们在 main 函数中声明的 char s[] 就位于这 80bytes 之内，请注意汇编代码中是如何初始化 s 数组的，由于我设置的是 char s[] = "abcdefg\0", 字符串"abcdefg\0"的内容存储在.rdata 节中，由于它正好是 8 个字符，所以就可以认为是相当于两个 DWORD 数据，所以在汇编代码里是用了 两条 mov 指令完成对 s 数组 的初始化的。

在函数即将返回之前，再次将当前栈的状态（esp）应该和刚进入函数时的“快照”（ebp）进行比较，查看两者是否相符，否则栈可能在函数 执行期间遭到了破坏，或者调用方和函数之间没有遵循同样的调用约定。我们使用 IDA 在调试时对 esp 或者 ebp 进行修改，使两者不等，则进入 __chkesp 函数以后，我发现会弹出下图所示的一个对话框，观察对话框上输出的信息大意是，“ESP 的值在函数调用期间没有被正确的存储，这通常是由于使用和函数不同 的调用约定声明的函数指针去调用函数导致的。”在我的印象中 chkesp 好像是在某个 VisualStdio 版本之后才开始加入并成为了默认打开的选项，已使得应用程序更加安全，好像在 IDE 中有一个编译开关可以关闭编译器生 成检查堆栈的代码，我们可以关闭这个开关使生成的应用程序更加精简，运行更加高效，但毫无疑问有可能降低其安全性。关于它的后续我们就不继续分析了。



假如用 16 进制编辑器把释放临时变量空间的汇编代码（add esp, **h）全部填充为 NOP 指令（0x90），运行时也会弹出上面的对话框，如果我们让程序接着运行，就会弹出另一个 XP 中常见的对话框：...exe 遇到问题需要 关闭，然后我们点击查看它发送的错误报告的技术信息，就会看到下面的对话框，在这里我查看了里面的 Address，是触发中断的汇编代码（int 3），然后是 CPUID（通过 cpuid 指令得到），后面是一些模块，栈，内存信息。由于我们是在运行时有意破坏了 esp 校验的条件，所以这些数据发给 MS 公司显然也是不太可能有什么结果的。



下面我们再看以下 getstring 的汇编代码，这里含有一个基本的 for 循环：

```
.text:00401020 getstring      proc near               ; CODE XREF: j_getstringj
.text:00401020
.text:00401020 var_48         = dword ptr -48h
.text:00401020 var_8          = dword ptr -8
.text:00401020 var_4          = dword ptr -4
.text:00401020 arg_0          = dword ptr  8
.text:00401020
.text:00401020                push    ebp
.text:00401021                mov     ebp, esp  ;ebp = esp;
.text:00401023                sub     esp, 48h  ;在栈上申请 72bytes（供临时变量使用）
.text:00401026                push    ebx
.text:00401027                push    esi
.text:00401028                push    edi ;保存寄存器数据
.text:00401029                lea     edi, [ebp+var_48]
.text:0040102C                mov     ecx, 12h
.text:00401031                mov     eax, 0CCCCCCCCh
.text:00401036                rep stosd

                          ;调用 strlen(s);
.text:00401038                mov     eax, [ebp+arg_0] ;取出参数 s 的地址（在 ebp 下面）
.text:0040103B                push    eax   ;参数入栈
.text:0040103C                call    strlen
.text:00401041                add     esp, 4 ;调用方复原 栈指针
.text:00401044                mov     [ebp+var_4], eax ; length = strlen(s);

                          ; 从这里开始进入 for 循环， 下面是初始化部分(i = 0)
.text:00401047                mov     [ebp+var_8], 0   ; i = 0;
.text:0040104E                jmp     short loc_401059 ; 跳转到条件判断部分

                            ; 这里是后续语句（i++; ）
.text:00401050
.text:00401050 loc_401050:                            ; CODE XREF: getstring+60j
```

```
.text:00401050          mov     ecx, [ebp+var_8]
.text:00401053          add     ecx, 1
.text:00401056          mov     [ebp+var_8], ecx ; i = i + 1;

                ; 这里是条件判断语句 (i < length? )
                ;jge: 在 >= 时跳转到指定地址；
.text:00401059
.text:00401059 loc_401059:                      ; CODE XREF: getstring+2Ej
.text:00401059          mov     edx, [ebp+var_8]
.text:0040105C          cmp     edx, [ebp+var_4]
.text:0040105F          jge     short loc_401082 ; i<length?

                ; 以下是循环主题部分 {... }
.text:00401061          mov     eax, [ebp+arg_0] ; eax = s;
.text:00401064          add     eax, [ebp+var_8] ; eax = s + i;
.text:00401067          movsx   eax, byte ptr [eax] ; eax = s[i];
.text:0040106A          sub     eax, 60h
                ; eax = s[i] - 'a' + 1; ('a' = 0x61)

.text:0040106D          cdq
                ;CDQ: Convert Double to Quad (386+)
                ;把 edx 扩展为 eax 的高位，也就是说变为 64 位。

.text:0040106E          mov     ecx, 1Ah        ; ecx = 26;
.text:00401073          idiv    ecx
                ; idiv: 有符号除法, eax 为结果，edx 为余数；

.text:00401075          add     edx, 61h        ; edx += 'a';
.text:00401078          mov     eax, [ebp+arg_0]
.text:0040107B          add     eax, [ebp+var_8]
.text:0040107E          mov     [eax], dl       ; s[i] = edx low
.text:00401080          jmp     short loc_401050 ; 跳转到（i++）处


                ; 以下开始循环体之后的后续代码
.text:00401082
.text:00401082 loc_401082:                      ; CODE XREF: getstring+3Fj
.text:00401082          mov     eax, [ebp+var_4] ; return length;
.text:00401085          pop     edi
.text:00401086          pop     esi
.text:00401087          pop     ebx     ; 恢复寄存器数据
.text:00401088          add     esp, 48h ; 释放栈上申请的临时空间
```

```
.text:0040108B            cmp    ebp, esp ; 检查栈
.text:0040108D            call   __chkesp
.text:00401092            mov    esp, ebp
.text:00401094            pop    ebp
.text:00401095            retn
.text:00401095 getstring      endp
```

在上面的代码中，取参数 s 的地址是 [ebp + 8]？ 这是为什么呢，可以看调用函数的过程：

```
.text:004010F2                          lea          eax,   [ebp+var_C]
.text:004010F5                          push         eax                          // esp 就是 s 的地址（位于栈上）
.text:004010F6                          call         j_getstring                  //注意 call 指令会把下一条指令地址（ 004010FB ）入栈， esp 增加 4；
.text:004010FB                          add          esp,   4                     //调用方复原堆栈。
```

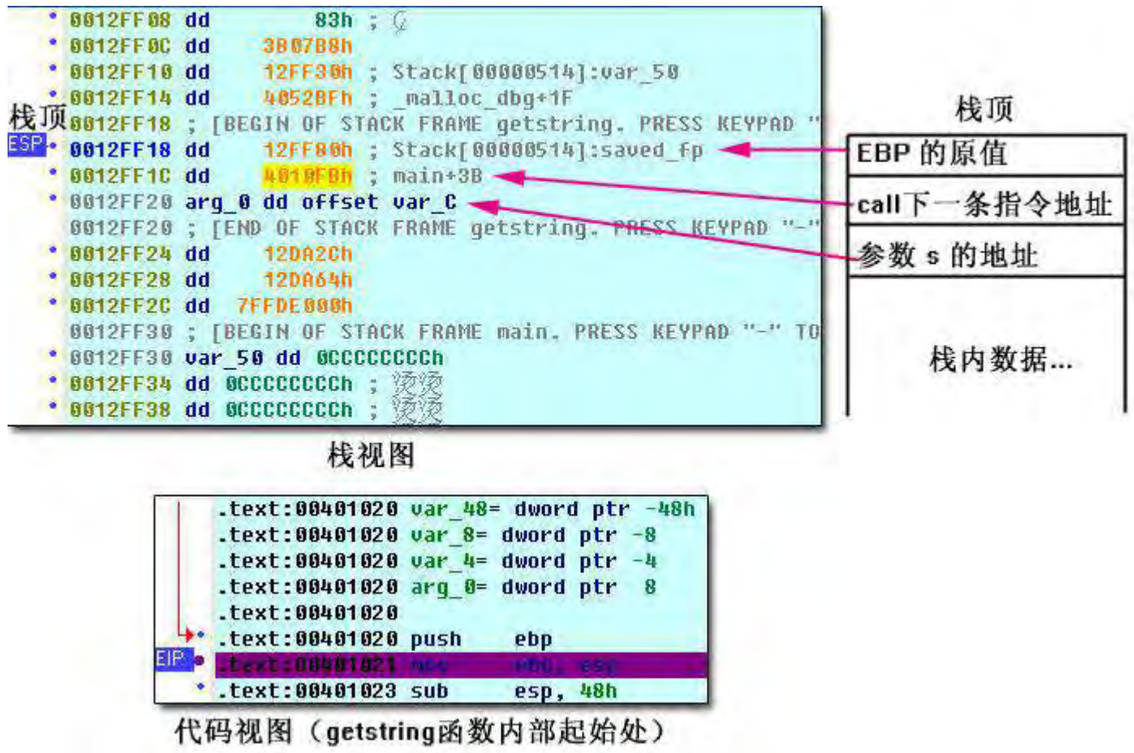
call 指令然后跳转到 getstring 函数：

```
.text:00401020  getstring               proc  near

.text:00401020  arg_0                       =  dword  ptr    8
.text:00401020
.text:00401020                              push         ebp                          //为了保存 ebp 的值，esp 再次增加 4；
.text:00401021                              mov          ebp,   esp                   //因此这时的 ebp  /  esp 距离参数的距离就是 8 bytes；
```

ebp 入栈后，栈内的数据如下：

```
-------------------------------------------------------------------------------

 esp--->|  ebp 的原值 （4bytes）

                    | 函数返回时的跳转地址（call j_getstring 的下一条指令地址） （4 bytes）

                    | 参数地址（第一个/最左侧参数），即 char s[] 的地址（位于栈上）；
```

因此函数的参数距离 ebp 的距离是 8 bytes，如果是按照从右到左的顺序入栈参数，则越靠左侧的参数距离栈顶越近（深度越浅）。以上分析内容在 IDA 的运行时截图如下所示：



在这里我们再总结以下 VC 编译器为一个函数生成的典型汇编代码：（注：这里的调用约定内容是：参数从右到左入栈，调用方负责参数出栈。）

（1）push ebp；　　ebp 入栈，保存 ebp 的原值；

（2）mov ebp，esp；　　ebp 指向当前栈顶；（此后 ebp 用作访问参数和函数临时变量的依据）

（3）sub esp，**h；　　为函数内的临时变量在栈上申请空间。（大于等于临时变量的精确需求，以 32bits 内存对齐）

（4）编译器产生的开场白（prolog）。（注：如果使用 MS 的关键字 naked 可以自己编写这部分汇编代码。）

包括：入栈保护寄存器的值，ESI，EDI，EBX，EBP （假如在函数里面用到了他们）；“初始化”临时变量分配的栈上空间；（备注：用 0xCC 填充）

（5）函数主体；（返回值被放到 eax）

（6）编译器产生的收场白（epilog）。（注：同上，可以用 naked 关键字自己来编写这部分汇编代码）

包括：复原被保护的寄存器；

（7）add esp, **h;　　释放为临时变量在栈上申请的空间。

（8）cmp ebp, esp

call __chkesp

mov esp, ebp　　检查栈指针

（9）pop ebp　　恢复 ebp 的值

（10） retn　　返回。

【备注】注意，这里讲的初始化有两种级别，一种是**编译器级别**（用 0xcc 填充栈上的临时变量空间），这一步骤对程序员透明；另一种是程序员编码时对临时变量的初始化，这是在**高级语言编码级别**的。

所有没经过**程序员初始化**的函数临时变量的数据都是 0xCC；这样可以很容易辨认出变量是否没有经过**编码级别初始化**， 例如没有程序员初始化的数据，字符串都是这样的特殊值，对于程序员由于粗心而忘记初始化的情况很容易发现。如果编译器没有做这一个动作，则临时变量的值都 是随机数据，它们可能是由过去的“函数”使用后留下的痕迹，我们很难辨别它们是的确被程序员有意初始化了还是被遗忘了（从来就没有被赋予过初值），所以编 译器的这个初始化动作尽管并非程序运行所必须，但却是我们排错调试所必要的。

【补充说明】hoodlum1980 于 2011-8-27

我在这里提到的“初始化过程”（对函数内申请的栈上空间用 0xCC 填充）针对的是 VC 在生成 Debug 版本时，编译器会产生这些代码。因此 Debug 版本中在程序中未初始化的栈上数据、缓冲区将被体现为这些值。有一种说法是因为 0xCC 是 int3 指令，这样一旦 PC 意外跳转到栈上，调试时就可以触发编译器中断。这种说法有一定道理，但 PC 从代码段跳转到进程的栈上空间的可

能性几乎没有，且各个 section 上都有段特性（相当于权限和性质）的定义。所以这种说法我还从未见到在现实中被验证过。当然，凡事不排除一个意外，至于为什么 VC 采用 0xCC 来填充栈上的空间的这个问题？可能需要开发编译器的更内部的人士来解答。

其中栈内的数据分布如下：

栈顶 esp--->　　临时变量空间（保证内存对齐）

受保护的寄存器值

ebp------->　　ebp 的旧值

函数返回时的跳转地址

参数（从上到下的顺序：左侧参数，右侧参数）

这里我们可以看到 for 循环中包含的四个基本部分，在汇编代码中按下面的顺序排列：

1. 初始化语句（i=0）；　然后跳转到 3；

2. 后续工作语句（i++）；

3. 条件语句（i<imax ？）；条件不满足时跳转到 5；

4. 循环体内部主体（{...}）；然后跳转到 2；

5. 继续向下执行后面的代码...

其中的 2，3，5 的起始地址在汇编代码中有标签以供跳转，即一个 for 循环的汇编代码主要由四部分组成，同时产生三个供跳转的地址标签。

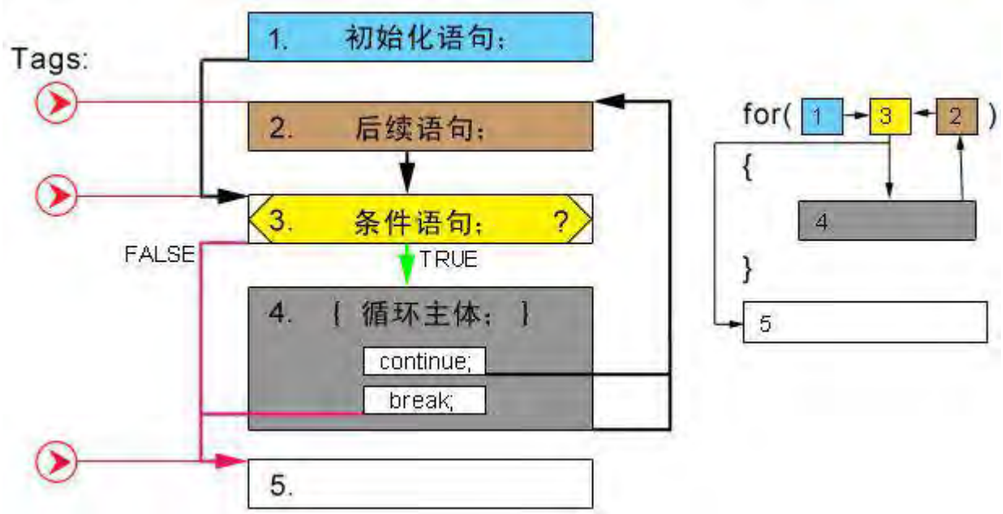
在循环体控制中，还有两个主要的控制流程的高级语言代码是 continue 和 break；这两者对循环的影响是很多人都非常清楚的，但是这 两者对后续语句（i++）的影响恐怕就不是每个人都能分辨的清楚了。例如假如有如下的考题，请写出他们的输出，你是否能够搞得清楚？

```
//题目 A：
int result;
for( i= 0, result = 0; i<10; i++, result++)
{
	if( i == 5 ) continue;
}
printf("%d", result);
```

```
//题目 B：
int result;
for( i= 0, result = 0; i<10; i++, result++)
{
	if( i == 5 ) break;
}
printf("%d", result);
```

实际上 continue 将跳转到 2， break 将跳转到 5；请注意这两者的区别，continue 是对 continue 之后的循环主体进行跳过（仅仅是循环主体不能够完整，其他三个部分的执 行是完整的。），所有的 后续语句（i++）则都会完整执行；而 break 是直接离开循环体 （属于本次循环后续的 i++ 则不会被执行）。因此弄明白这一点同样是非常重要的。

下面我们给出一个 for 循环的示意图作为结束，从下图可以看出 2，3，4 组成一个回环（loop），此回环唯一的出入口位于 3（条件语句），如果条件语句永远为 TRUE，则无休止的在回环中运行，永远跳不出，也就是所谓的死循环。



--The end; by hoodlum1980; on 2010.07.31;

【题外话】strlen 在 VC 中是用汇编语言实现的（可以在 VC 中查到它的汇编代码），非常有趣，它的高效之处在于不是逐个 char 去比较，而是以 4 个 char 为一组（DWORD）进行判断。第一次看恐怕很难看懂 strlen 的汇编代码，可以参考下面的文章：

《strlen 源码剖析 》, ant

http://www.cnblogs.com/hoodlum1980/archive/2010/07/30/1789092.html

IDA 学习笔记--VS2008 按钮事件捕捉

用到工具：

IDA Proc

C32Asm

Rescope

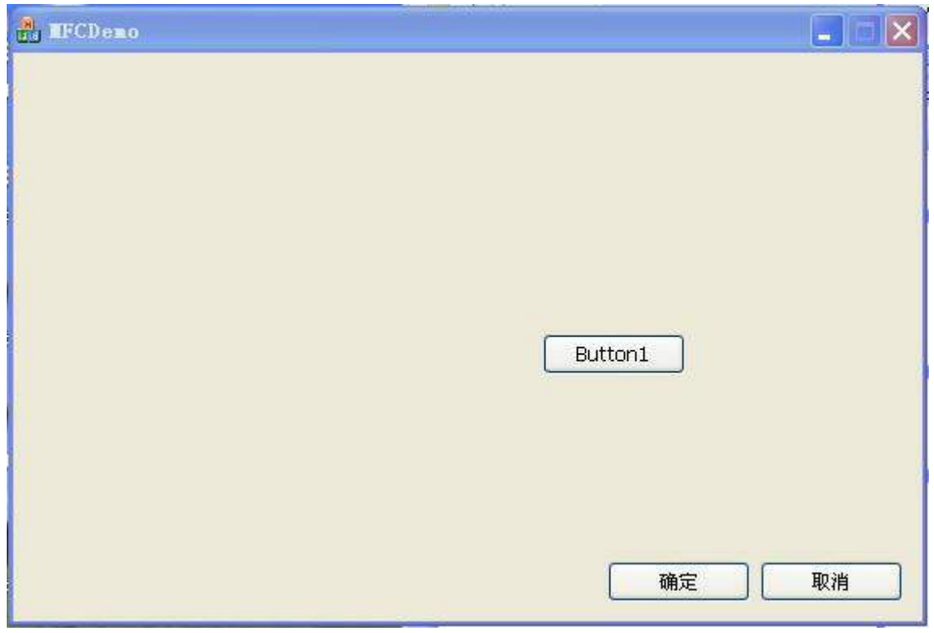
VS2008

LordPE

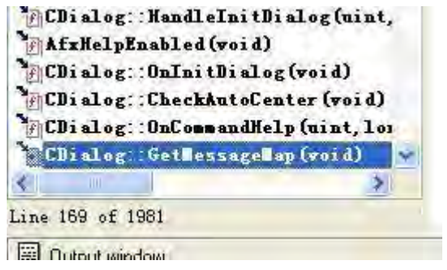
实例程序:MFCDemo.exe（附下载链接），我们目标是找到 Button1 对应的函数的地址处

附件包含：博客文章原文文档，文章中用到的 MFCDemo 程序。

下载链接：http://download.csdn.net/detail/ccnyou/5012040

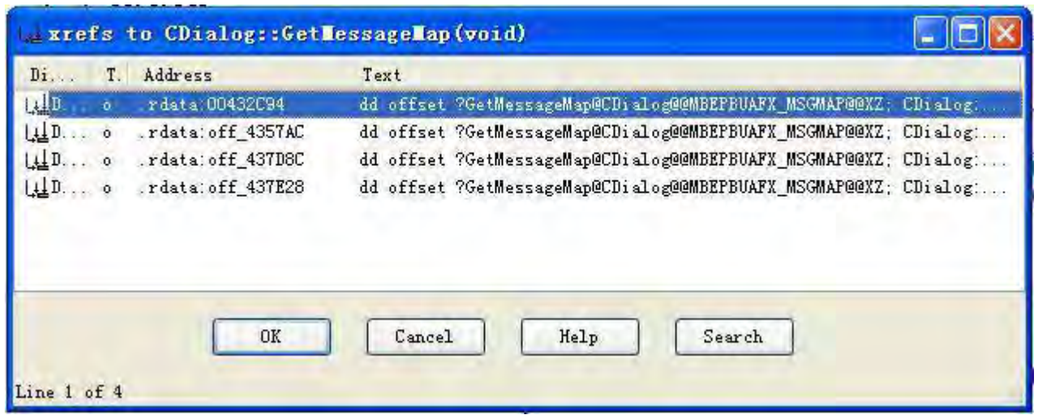


1，首先，IDA 载入程序，在左侧 Function Name 中输入　CDialog::GetMessageMap（没有输入框，只是定位到这个函数而已，其实书一部分就出来了 PS,如果能找到 GetThisMessageMap 优选选择 GetThisMessageMap），如图



2，在 XREF 中右键，选择“跳到交叉参考”，如图：

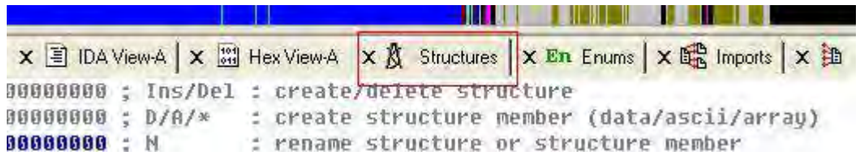
```
.text:00404207
.text:00404207 ; AFX_MSGMAP *__thiscall CDialog__GetMessageMap(CDialog *this)
.text:00404207 ?GetMessageMap@CDialog@@MBEPBUAFX_MSGMAP@@XZ proc near
.text:00404207                                         ; DATA XREF: .rdata:00432C94↓o
.text:00404207                                         ; .rdata:off_4357AC↓o ...
.text:00404207 this = ecx
.text:00404207                 mov     eax, offset messageMap_0
.text:0040420C                 retn
.text:0040420C ?GetMessageMap@CDialog@@MBEPBUAFX_MSGMAP@@XZ endp
.text:0040420C
```



点进去后第一个不是我们要找的，第二个开始才是我们目标，第二个如图：

```
.rdata:004357AC off_4357AC      dd offset ?GetMessageMap@CDialog@@MBEPBUAFX_MSGMAP@@XZ
.rdata:004357AC                                         ; DATA XREF: CCommonDialog::GetMessageMap(void)↑o
.rdata:004357AC                                         ; CDialog::GetMessageMap(void)
.rdata:004357B0                 dd offset unk_4357B8
.rdata:004357B4                 align 8
.rdata:004357B8 unk_4357B8      db  53h ; S             ; DATA XREF: .rdata:004357B0↑o
.rdata:004357B9                 db    0
.rdata:004357BA                 db    0
.rdata:004357BB                 db    0
.rdata:004357BC                 db    0
.rdata:004357BD                 db    0
.rdata:004357BE                 db    0
.rdata:004357BF                 db    0
.rdata:004357C0                 db    0
.rdata:004357C1                 db    0
.rdata:004357C2                 db    0
.rdata:004357C3                 db    0
.rdata:004357C4                 db    0
.rdata:004357C5                 db    0
.rdata:004357C6                 db    0
.rdata:004357C7                 db    0
.rdata:004357C8                 db    7
.rdata:004357C9                 db    0
.rdata:004357CA                 db    0
.rdata:004357CB                 db    0
```

3，此处，点击 IDA 上面的 Structures 打开结构体窗口，按键盘 Insert 增加一个结构体



名字使用 AFX_MSGMAP_ENTRY，然后依次按 D 增加几个成员并一一改名，最终如图

```
0000000
0000000 AFX_MSGMAP_ENTRY struc ; (sizeof=0x18, standard type)
0000000 nMessage        dd ?
0000004 nCode           dd ?
0000008 nID             dd ?
000000C nLastID         dd ?
0000010 nSig            dd ?
0000014 pfn             dd ?                    ; offset
0000018 AFX_MSGMAP_ENTRY ends
0000018
```

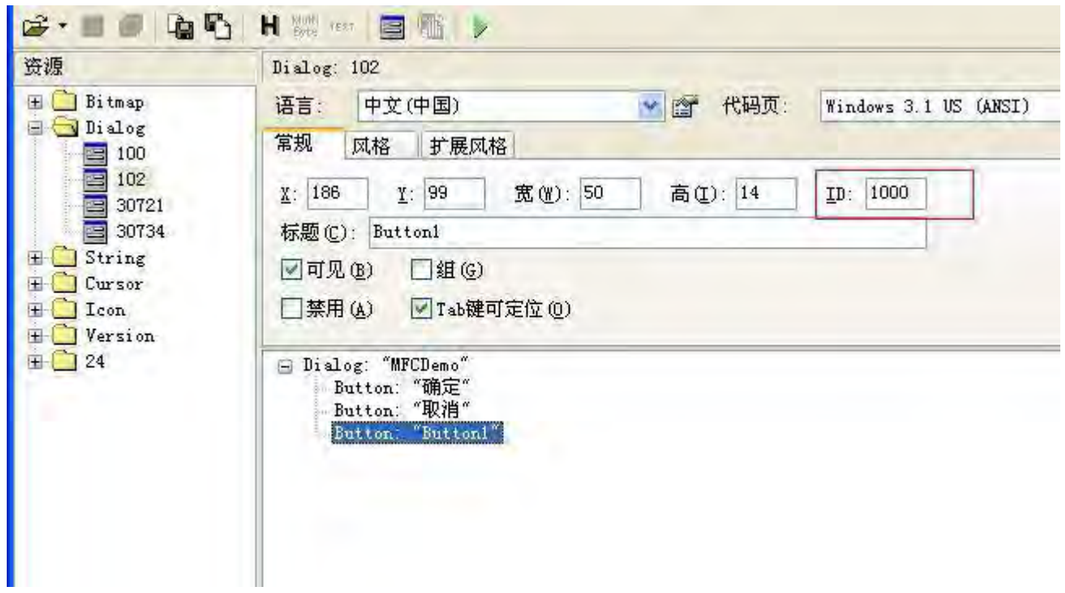
接下来，回到刚才汇编窗口，从第一个 unk_4357B8 开始，按 Alt+Q，将此处转为结构体变量，在弹出来的窗口选择 AFX_MSGMAP_ENTRY，转化之后如图：

```
data:004357AB                   db    0
data:004357AC off_4357AC        dd offset ?GetMessageMap@CDialog@@MBEPBUAFX_MSGMAP@@XZ
data:004357AC                                   ; DATA XREF: CCommonDialog::GetMessageMap(void)↑o
data:004357AC                                   ; CDialog::GetMessageMap(void)
data:004357B0                   dd offset stru_4357B8
data:004357B4                   align 8
data:004357B8 stru_4357B8       AFX_MSGMAP_ENTRY <53h, 0, 0, 0, 7, \
data:004357B8                                   ; DATA XREF: .rdata:004357B0↑o
data:004357B8                                offset ?OnHelpInfo@CCommonDialog@@IAEHPAUtagHELPINFO@@@Z> ; CCo
data:004357D0                   AFX_MSGMAP_ENTRY <0Fh, 0, 0, 0, 13h, \ ; CCommonDialog::OnCancel(void)
data:004357D0                                offset ?OnCancel@CCommonDialog@@MAEXXZ>
data:004357E8                   AFX_MSGMAP_ENTRY <0>
```

4，到这里，其实已经可以发现一些信息，VS2008 中的 AFX_MSGMAP_ENTRY 结构体，对于按钮消息都是这样子的

```
{  WM_COMMAND,  (WORD)BN_CLICKED,  (WORD)IDC_BUTTON1,  (WORD)IDC_BUTTON1,  AfxSigCmd_v,  \
(static_cast<  AFX_PMSG  >  (&CMFCDemoDlg::OnBnClickedButton1))  },
```

我们用 Rescope 打开目标程序，打开 Dialog 窗口，找到目标按钮 Button1，拿到按钮 ID 如图

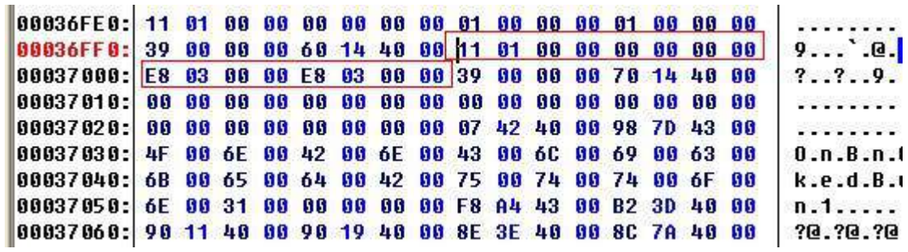


这里 ID = 1000 = 0x03E8，然后根据上面的信息，构造一串 HEX 常量

1101000000000000E8030000E8030000

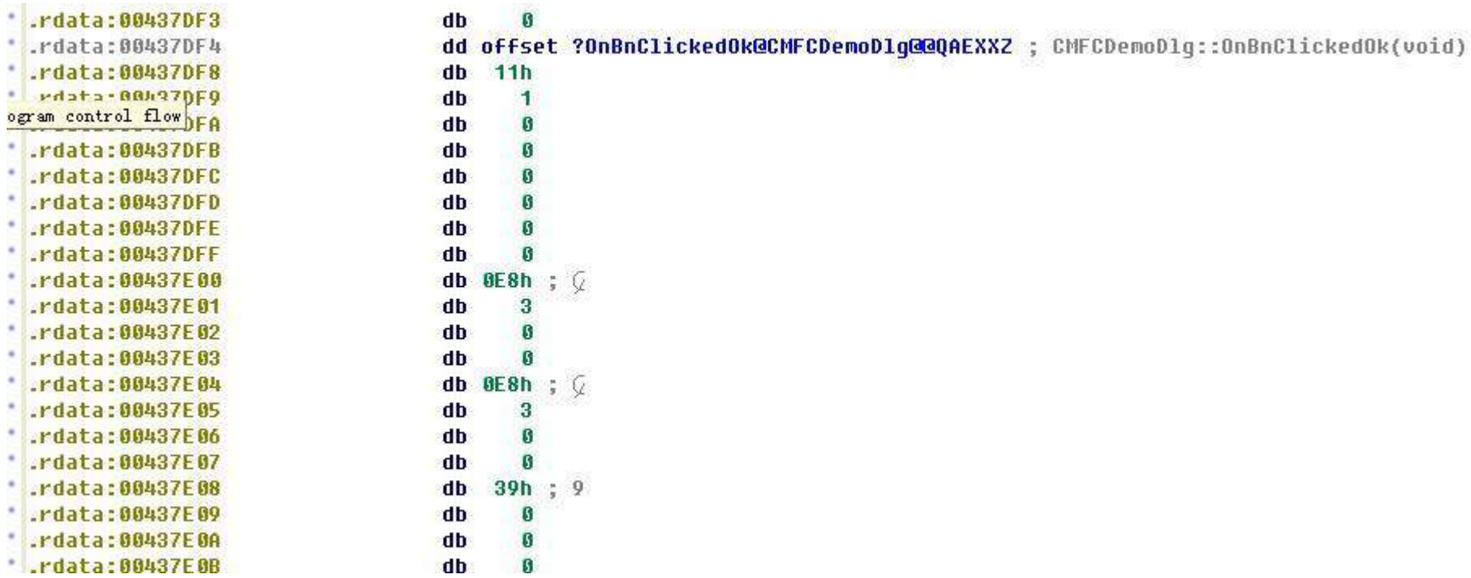
其中，1101 就是 0x0111，对应 WM_COMMAND， E803 即是 0x03E8，就是按钮 ID。

将程序载入 C32Asm，搜索 HEX,找到一个：



右键，选择【转到对应汇编模式编辑】，拿到指令地址 00436FF8，减掉基址 00400000 得到 036FF8，将程序载入 LordPE，使用位置计算器，在偏移量中输入 036FF8，点击转换，得到 VA=00437DF8。

4，在 IDA 中反汇编窗口，按 G 转到 00437DF8，如图

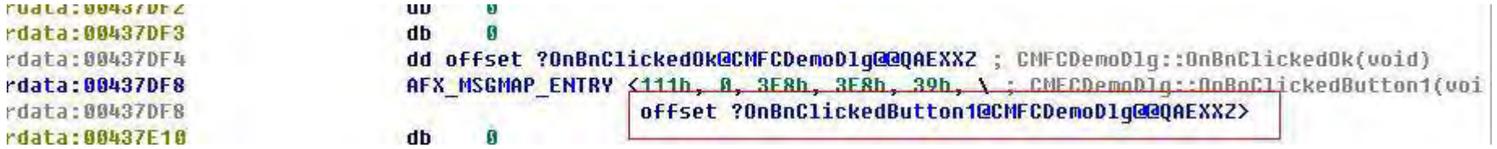


这里要说明下，上面那串

dd offset ?OnBnClickedOk@CMFCDemoDlg@@QAEXXZ ; CMFCDemoDlg::OnBnClickedOk(void)

估计是由于 IDA 找到了调试信息，但是一般逆向过程是没有调试信息的，这里忽略它。

我们对着 db 11h 按 Alt+Q，选择刚才的 AFX_MSGMAP_ENTRY，目标出来，如图：



在没有调试信息的情况下，这里是一个函数偏移量，双击就能到达函数代码处。文章至此结束。有问题请给我留言^_^。

http://blog.csdn.net/ccnyou/article/details/8521611

Windows 消息大全----------IDA 使用

MSG 结构体原型：

```
1 typedef struct{
2         HWND hwnd;
3         UINT message;
4         WPARAM wParam;
5         LPARAM lParam;
6         DWORD time;
7         POINT pt;
8 } MSG, *PMSG;
```

```
WM_NULL = $0000;

WM_CREATE = $0001;
应用程序创建一个窗口
WM_DESTROY = $0002;
一个窗口被销毁
WM_MOVE = $0003;
移动一个窗口
WM_SIZE = $0005;
改变一个窗口的大小
WM_ACTIVATE = $0006;
一个窗口被激活或失去激活状态;
WM_SETFOCUS = $0007;
获得焦点后
WM_KILLFOCUS = $0008;
失去焦点
WM_ENABLE = $000A;
```

改变 enable 状态
WM_SETREDRAW = $000B;
设置窗口是否能重画
WM_SETTEXT = $000C;
应用程序发送此消息来设置一个窗口的文本
WM_GETTEXT = $000D;
应用程序发送此消息来复制对应窗口的文本到缓冲区
WM_GETTEXTLENGTH = $000E;
得到与一个窗口有关的文本的长度（不包含空字符）
WM_PAINT = $000F;
要求一个窗口重画自己
WM_CLOSE = $0010;
当一个窗口或应用程序要关闭时发送一个信号
WM_QUERYENDSESSION = $0011;
当用户选择结束对话框或程序自己调用 ExitWindows 函数
WM_QUIT = $0012;
用来结束程序运行或当程序调用 postquitmessage 函数
WM_QUERYOPEN = $0013;
当用户窗口恢复以前的大小位置时，把此消息发送给某个图标
WM_ERASEBKGND = $0014;
当窗口背景必须被擦除时（例在窗口改变大小时）
WM_SYSCOLORCHANGE = $0015;
当系统颜色改变时，发送此消息给所有顶级窗口
WM_ENDSESSION = $0016;
当系统进程发出 WM_QUERYENDSESSION 消息后，此消息发送给应用程序，
通知它对话是否结束
WM_SYSTEMERROR = $0017;
WM_SHOWWINDOW = $0018;
当隐藏或显示窗口是发送此消息给这个窗口
WM_ACTIVATEAPP = $001C;
发此消息给应用程序哪个窗口是激活的，哪个是非激活的；
WM_FONTCHANGE = $001D;
当系统的字体资源库变化时发送此消息给所有顶级窗口
WM_TIMECHANGE = $001E;
当系统的时间变化时发送此消息给所有顶级窗口
WM_CANCELMODE = $001F;
发送此消息来取消某种正在进行的摸态（操作）
WM_SETCURSOR = $0020;
如果鼠标引起光标在某个窗口中移动且鼠标输入没有被捕获时，就发消息给某个窗口
WM_MOUSEACTIVATE = $0021;
当光标在某个非激活的窗口中而用户正按着鼠标的某个键发送此消息给当前窗口
WM_CHILDACTIVATE = $0022;

发送此消息给 MDI 子窗口当用户点击此窗口的标题栏，或当窗口被激活，移动，改变大小
WM_QUEUESYNC = $0023;
此消息由基于计算机的训练程序发送，通过 WH_JOURNALPALYBACK 的 hook 程序
分离出用户输入消息
WM_GETMINMAXINFO = $0024;
此消息发送给窗口当它将要改变大小或位置;
WM_PAINTICON = $0026;
发送给最小化窗口当它图标将要被重画
WM_ICONERASEBKGND = $0027;
此消息发送给某个最小化窗口，仅当它在画图标前它的背景必须被重画
WM_NEXTDLGCTL = $0028;
发送此消息给一个对话框程序去更改焦点位置
WM_SPOOLERSTATUS = $002A;
每当打印管理列队增加或减少一条作业时发出此消息
WM_DRAWITEM = $002B;
当 button，combobox，listbox，menu 的可视外观改变时发送
此消息给这些空件的所有者
WM_MEASUREITEM = $002C;
当 button, combo box, list box, list view control, or menu item 被创建时
发送此消息给控件的所有者
WM_DELETEITEM = $002D;
当 the list box 或 combo box 被销毁 或 当 某些项被删除通过 LB_DELETESTRING, LB_RESETCONTENT, CB_DELETESTRING, or CB_RESETCONTENT 消息
WM_VKEYTOITEM = $002E;
此消息有一个 LBS_WANTKEYBOARDINPUT 风格的发出给它的所有者来响应 WM_KEYDOWN 消息
WM_CHARTOITEM = $002F;
此消息由一个 LBS_WANTKEYBOARDINPUT 风格的列表框发送给他的所有者来响应 WM_CHAR 消息
WM_SETFONT = $0030;
当绘制文本时程序发送此消息得到控件要用的颜色
WM_GETFONT = $0031;
应用程序发送此消息得到当前控件绘制文本的字体
WM_SETHOTKEY = $0032;
应用程序发送此消息让一个窗口与一个热键相关连
WM_GETHOTKEY = $0033;
应用程序发送此消息来判断热键与某个窗口是否有关联
WM_QUERYDRAGICON = $0037;
此消息发送给最小化窗口，当此窗口将要被拖放而它的类中没有定义图标，应用程序能返回一个图标或光标的句柄，当用户拖放图标时系统显示这个图标或光标
WM_COMPAREITEM = $0039;
发送此消息来判定 combobox 或 listbox 新增加的项的相对位置
WM_GETOBJECT = $003D;
WM_COMPACTING = $0041;

显示内存已经很少了
WM_WINDOWPOSCHANGING = $0046;
发送此消息给那个窗口的大小和位置将要被改变时，来调用 setwindowpos 函数或其它窗口管理函数
WM_WINDOWPOSCHANGED = $0047;
发送此消息给那个窗口的大小和位置已经被改变时，来调用 setwindowpos 函数或其它窗口管理函数
WM_POWER = $0048;（适用于 16 位的 windows）
当系统将要进入暂停状态时发送此消息
WM_COPYDATA = $004A;
当一个应用程序传递数据给另一个应用程序时发送此消息
WM_CANCELJOURNAL = $004B;
当某个用户取消程序日志激活状态，提交此消息给程序
WM_NOTIFY = $004E;
当某个控件的某个事件已经发生或这个控件需要得到一些信息时，发送此消息给它的父窗口
WM_INPUTLANGCHANGEREQUEST = $0050;
当用户选择某种输入语言，或输入语言的热键改变
WM_INPUTLANGCHANGE = $0051;
当平台现场已经被改变后发送此消息给受影响的最顶级窗口
WM_TCARD = $0052;
当程序已经初始化 windows 帮助例程时发送此消息给应用程序
WM_HELP = $0053;
此消息显示用户按下了 F1，如果某个菜单是激活的，就发送此消息个此窗口关联的菜单，否则就
发送给有焦点的窗口，如果当前都没有焦点，就把此消息发送给当前激活的窗口
WM_USERCHANGED = $0054;
当用户已经登入或退出后发送此消息给所有的窗口，当用户登入或退出时系统更新用户的具体
设置信息，在用户更新设置时系统马上发送此消息；
WM_NOTIFYFORMAT = $0055;
公用控件，自定义控件和他们的父窗口通过此消息来判断控件是使用 ANSI 还是 UNICODE 结构
在 WM_NOTIFY 消息，使用此控件能使某个控件与它的父控件之间进行相互通信
WM_CONTEXTMENU = $007B;
当用户某个窗口中点击了一下右键就发送此消息给这个窗口
WM_STYLECHANGING = $007C;
当调用 SETWINDOWLONG 函数将要改变一个或多个 窗口的风格时发送此消息给那个窗口
WM_STYLECHANGED = $007D;
当调用 SETWINDOWLONG 函数一个或多个 窗口的风格后发送此消息给那个窗口
WM_DISPLAYCHANGE = $007E;
当显示器的分辨率改变后发送此消息给所有的窗口
WM_GETICON = $007F;
此消息发送给某个窗口来返回与某个窗口有关连的大图标或小图标的句柄；
WM_SETICON = $0080;
程序发送此消息让一个新的大图标或小图标与某个窗口关联；
WM_NCCREATE = $0081;
当某个窗口第一次被创建时，此消息在 WM_CREATE 消息发送前发送；

WM_NCDESTROY = $0082;
此消息通知某个窗口，非客户区正在销毁
WM_NCCALCSIZE = $0083;
当某个窗口的客户区域必须被核算时发送此消息
WM_NCHITTEST = $0084;//移动鼠标，按住或释放鼠标时发生
WM_NCPAINT = $0085;
程序发送此消息给某个窗口当它（窗口）的框架必须被绘制时；
WM_NCACTIVATE = $0086;
此消息发送给某个窗口 仅当它的非客户区需要被改变来显示是激活还是非激活状态；
WM_GETDLGCODE = $0087;
发送此消息给某个与对话框程序关联的控件，widdows 控制方位键和 TAB 键使输入进入此控件
通过响应 WM_GETDLGCODE 消息，应用程序可以把他当成一个特殊的输入控件并能处理它
WM_NCMOUSEMOVE = $00A0;
当光标在一个窗口的非客户区内移动时发送此消息给这个窗口 //非客户区为：窗体的标题栏及窗
的边框体
WM_NCLBUTTONDOWN = $00A1;
当光标在一个窗口的非客户区同时按下鼠标左键时提交此消息
WM_NCLBUTTONUP = $00A2;
当用户释放鼠标左键同时光标某个窗口在非客户区十发送此消息；
WM_NCLBUTTONDBLCLK = $00A3;
当用户双击鼠标左键同时光标某个窗口在非客户区十发送此消息
WM_NCRBUTTONDOWN = $00A4;
当用户按下鼠标右键同时光标又在窗口的非客户区时发送此消息
WM_NCRBUTTONUP = $00A5;
当用户释放鼠标右键同时光标又在窗口的非客户区时发送此消息
WM_NCRBUTTONDBLCLK = $00A6;
当用户双击鼠标右键同时光标某个窗口在非客户区十发送此消息
WM_NCMBUTTONDOWN = $00A7;
当用户按下鼠标中键同时光标又在窗口的非客户区时发送此消息
WM_NCMBUTTONUP = $00A8;
当用户释放鼠标中键同时光标又在窗口的非客户区时发送此消息
WM_NCMBUTTONDBLCLK = $00A9;
当用户双击鼠标中键同时光标又在窗口的非客户区时发送此消息

WM_KEYFIRST = $0100;
WM_KEYDOWN = $0100;
//按下一个键
WM_KEYUP = $0101;
//释放一个键
WM_CHAR = $0102;
//按下某键，并已发出 WM_KEYDOWN， WM_KEYUP 消息
WM_DEADCHAR = $0103;

当用 translatemessage 函数翻译 WM_KEYUP 消息时发送此消息给拥有焦点的窗口
WM_SYSKEYDOWN = $0104;
当用户按住 ALT 键同时按下其它键时提交此消息给拥有焦点的窗口；
WM_SYSKEYUP = $0105;
当用户释放一个键同时 ALT 键还按着时提交此消息给拥有焦点的窗口
WM_SYSCHAR = $0106;
当 WM_SYSKEYDOWN 消息被 TRANSLATEMESSAGE 函数翻译后提交此消息给拥有焦点的窗口
WM_SYSDEADCHAR = $0107;
当 WM_SYSKEYDOWN 消息被 TRANSLATEMESSAGE 函数翻译后发送此消息给拥有焦点的窗口
WM_KEYLAST = $0108;
WM_INITDIALOG = $0110;
在一个对话框程序被显示前发送此消息给它，通常用此消息初始化控件和执行其它任务
WM_COMMAND = $0111;
当用户选择一条菜单命令项或当某个控件发送一条消息给它的父窗口，一个快捷键被翻译
WM_SYSCOMMAND = $0112;
当用户选择窗口菜单的一条命令或当用户选择最大化或最小化时那个窗口会收到此消息
WM_TIMER = $0113; //发生了定时器事件
WM_HSCROLL = $0114;
当一个窗口标准水平滚动条产生一个滚动事件时发送此消息给那个窗口，也发送给拥有它的控件
WM_VSCROLL = $0115;
当一个窗口标准垂直滚动条产生一个滚动事件时发送此消息给那个窗口也，发送给拥有它的控件 WM_INITMENU = $0116;
当一个菜单将要被激活时发送此消息，它发生在用户菜单条中的某项或按下某个菜单键，它允许程序在显示前更改菜单
WM_INITMENUPOPUP = $0117;
当一个下拉菜单或子菜单将要被激活时发送此消息，它允许程序在它显示前更改菜单，而不要改变全部
WM_MENUSELECT = $011F;
当用户选择一条菜单项时发送此消息给菜单的所有者（一般是窗口）
WM_MENUCHAR = $0120;
当菜单已被激活用户按下了某个键（不同于加速键），发送此消息给菜单的所有者；
WM_ENTERIDLE = $0121;
当一个模态对话框或菜单进入空载状态时发送此消息给它的所有者，一个模态对话框或菜单进入空载状态就是在处理完一条或几条先前的消息后没有消息它的列队中等待
WM_MENURBUTTONUP = $0122;
WM_MENUDRAG = $0123;
WM_MENUGETOBJECT = $0124;
WM_UNINITMENUPOPUP = $0125;
WM_MENUCOMMAND = $0126;
WM_CHANGEUISTATE = $0127;
WM_UPDATEUISTATE = $0128;
WM_QUERYUISTATE = $0129;
WM_CTLCOLORMSGBOX = $0132;
在 windows 绘制消息框前发送此消息给消息框的所有者窗口，通过响应这条消息，所有者窗口可以通过使用给定的相关显示设备的句柄来设置消息框的文本和背景颜色
WM_CTLCOLOREDIT = $0133;
当一个编辑型控件将要被绘制时发送此消息给它的父窗口；通过响应这条消息，所有者窗口可以通过使用给定的相关显示设备的句柄来设置编辑框的文本和背景颜色

WM_CTLCOLORLISTBOX = $0134;
当一个列表框控件将要被绘制前发送此消息给它的父窗口；通过响应这条消息，所有者窗口可以通过使用给定的相关显示设备的句柄来设置列表框的文本和背景颜色
WM_CTLCOLORBTN = $0135;
当一个按钮控件将要被绘制时发送此消息给它的父窗口；通过响应这条消息，所有者窗口可以通过使用给定的相关显示设备的句柄来设置按纽的文本和背景颜色
WM_CTLCOLORDLG = $0136;
当一个对话框控件将要被绘制前发送此消息给它的父窗口；通过响应这条消息，所有者窗口可以通过使用给定的相关显示设备的句柄来设置对话框的文本背景颜色
WM_CTLCOLORSCROLLBAR= $0137;
当一个滚动条控件将要被绘制时发送此消息给它的父窗口；通过响应这条消息，所有者窗口可以通过使用给定的相关显示设备的句柄来设置滚动条的背景颜色
WM_CTLCOLORSTATIC = $0138;
当一个静态控件将要被绘制时发送此消息给它的父窗口；通过响应这条消息，所有者窗口可以通过使用给定的相关显示设备的句柄来设置静态控件的文本和背景颜色
WM_MOUSEFIRST = $0200;
WM_MOUSEMOVE = $0200;
// 移动鼠标

// 移动鼠标
WM_LBUTTONDOWN = $0201;
//按下鼠标左键
WM_LBUTTONUP = $0202;
//释放鼠标左键
WM_LBUTTONDBLCLK = $0203;
//双击鼠标左键
WM_RBUTTONDOWN = $0204;
//按下鼠标右键
WM_RBUTTONUP = $0205;
//释放鼠标右键
WM_RBUTTONDBLCLK = $0206;
//双击鼠标右键
WM_MBUTTONDOWN = $0207;
//按下鼠标中键
WM_MBUTTONUP = $0208;
//释放鼠标中键
WM_MBUTTONDBLCLK = $0209;
//双击鼠标中键
WM_MOUSEWHEEL = $020A;
当鼠标轮子转动时发送此消息个当前有焦点的控件
WM_MOUSELAST = $020A;
WM_PARENTNOTIFY = $0210;
当 MDI 子窗口被创建或被销毁，或用户按了一下鼠标键而光标在子窗口上时发送此消息给它的父窗口
WM_ENTERMENULOOP = $0211;
发送此消息通知应用程序的主窗口 that 已经进入了菜单循环模式
WM_EXITMENULOOP = $0212;
发送此消息通知应用程序的主窗口 that 已退出了菜单循环模式

WM_NEXTMENU = $0213;
WM_SIZING = 532;
当用户正在调整窗口大小时发送此消息给窗口；通过此消息应用程序可以监视窗口大小和位置也可以修改他们
WM_CAPTURECHANGED = 533;
发送此消息 给窗口当它失去捕获的鼠标时；
WM_MOVING = 534;
当用户在移动窗口时发送此消息，通过此消息应用程序可以监视窗口大小和位置也可以修改他们；
WM_POWERBROADCAST = 536;
此消息发送给应用程序来通知它有关电源管理事件；
WM_DEVICECHANGE = 537;
当设备的硬件配置改变时发送此消息给应用程序或设备驱动程序
WM_IME_STARTCOMPOSITION = $010D;
WM_IME_ENDCOMPOSITION = $010E;
WM_IME_COMPOSITION = $010F;
WM_IME_KEYLAST = $010F;
WM_IME_SETCONTEXT = $0281;
WM_IME_NOTIFY = $0282;
WM_IME_CONTROL = $0283;
WM_IME_COMPOSITIONFULL = $0284;
WM_IME_SELECT = $0285;
WM_IME_CHAR = $0286;
WM_IME_REQUEST = $0288;
WM_IME_KEYDOWN = $0290;
WM_IME_KEYUP = $0291;
WM_MDICREATE = $0220;
应用程序发送此消息给多文档的客户窗口来创建一个 MDI 子窗口
WM_MDIDESTROY = $0221;
应用程序发送此消息给多文档的客户窗口来关闭一个 MDI 子窗口
WM_MDIACTIVATE = $0222;
应用程序发送此消息给多文档的客户窗口通知客户窗口激活另一个 MDI 子窗口，当客户窗口收到此消息后，它发出 WM_MDIACTIVE 消息给 MDI 子窗口（未激活）激活它；
WM_MDIRESTORE = $0223;
程序 发送此消息给 MDI 客户窗口让子窗口从最大最小化恢复到原来大小
WM_MDINEXT = $0224;
程序 发送此消息给 MDI 客户窗口激活下一个或前一个窗口
WM_MDIMAXIMIZE = $0225;
程序发送此消息给 MDI 客户窗口来最大化一个 MDI 子窗口；
WM_MDITILE = $0226;
程序 发送此消息给 MDI 客户窗口以平铺方式重新排列所有 MDI 子窗口
WM_MDICASCADE = $0227;
程序 发送此消息给 MDI 客户窗口以层叠方式重新排列所有 MDI 子窗口
WM_MDIICONARRANGE = $0228;
程序 发送此消息给 MDI 客户窗口重新排列所有最小化的 MDI 子窗口

WM_MDIGETACTIVE = $0229;
程序 发送此消息给 MDI 客户窗口来找到激活的子窗口的句柄
WM_MDISETMENU = $0230;
程序 发送此消息给 MDI 客户窗口用 MDI 菜单代替子窗口的菜单

WM_ENTERSIZEMOVE = $0231;
WM_EXITSIZEMOVE = $0232;
WM_DROPFILES = $0233;
WM_MDIREFRESHMENU = $0234;
WM_MOUSEHOVER = $02A1;
WM_MOUSELEAVE = $02A3;
WM_CUT = $0300;
程序发送此消息给一个编辑框或 combobox 来删除当前选择的文本
WM_COPY = $0301;
程序发送此消息给一个编辑框或 combobox 来复制当前选择的文本到剪贴板
WM_PASTE = $0302;
程序发送此消息给 editcontrol 或 combobox 从剪贴板中得到数据
WM_CLEAR = $0303;
程序发送此消息给 editcontrol 或 combobox 清除当前选择的内容;
WM_UNDO = $0304;
程序发送此消息给 editcontrol 或 combobox 撤消最后一次操作
WM_RENDERFORMAT = $0305;

WM_RENDERALLFORMATS = $0306;
WM_DESTROYCLIPBOARD = $0307;
当调用 ENPTYCLIPBOARD 函数时 发送此消息给剪贴板的所有者
WM_DRAWCLIPBOARD = $0308;
当剪贴板的内容变化时发送此消息给剪贴板观察链的第一个窗口; 它允许用剪贴板观察窗口来
显示剪贴板的新内容;
WM_PAINTCLIPBOARD = $0309;
当剪贴板包含 CF_OWNERDIPLAY 格式的数据并且剪贴板观察窗口的客户区需要重画;
WM_VSCROLLCLIPBOARD = $030A;
WM_SIZECLIPBOARD = $030B;
当剪贴板包含 CF_OWNERDIPLAY 格式的数据并且剪贴板观察窗口的客户区域的大小已经改变是此消息通过剪贴板观察窗口发送给剪贴板的所有者;
WM_ASKCBFORMATNAME = $030C;
通过剪贴板观察窗口发送此消息给剪贴板的所有者来请求一个 CF_OWNERDISPLAY 格式的剪贴板的名字
WM_CHANGECBCHAIN = $030D;
当一个窗口从剪贴板观察链中移去时发送此消息给剪贴板观察链的第一个窗口;
WM_HSCROLLCLIPBOARD = $030E;
此消息通过一个剪贴板观察窗口发送给剪贴板的所有者 ; 它发生在当剪贴板包含 CFOWNERDISPALY 格式的数据并且有个事件在剪贴板观察窗的水平滚动条上; 所有者应滚动剪贴板图象并更新滚动条的值;
WM_QUERYNEWPALETTE = $030F;

此消息发送给将要收到焦点的窗口，此消息能使窗口在收到焦点时同时有机会实现他的逻辑调色板
WM_PALETTEISCHANGING= $0310;
当一个应用程序正要实现它的逻辑调色板时发此消息通知所有的应用程序
WM_PALETTECHANGED = $0311;
此消息在一个拥有焦点的窗口实现它的逻辑调色板后发送此消息给所有顶级并重叠的窗口，以此来改变系统调色板
WM_HOTKEY = $0312;
当用户按下由 REGISTERHOTKEY 函数注册的热键时提交此消息
WM_PRINT = 791;
应用程序发送此消息仅当 WINDOWS 或其它应用程序发出一个请求要求绘制一个应用程序的一部分；
WM_PRINTCLIENT = 792;
WM_HANDHELDFIRST = 856;
WM_HANDHELDLAST = 863;
WM_PENWINFIRST = $0380;
WM_PENWINLAST = $038F;
WM_COALESCE_FIRST = $0390;
WM_COALESCE_LAST = $039F;
WM_DDE_FIRST = $03E0;
WM_DDE_INITIATE = WM_DDE_FIRST + 0;
一个 DDE 客户程序提交此消息开始一个与服务器程序的会话来响应那个指定的程序和主题名；
WM_DDE_TERMINATE = WM_DDE_FIRST + 1;
一个 DDE 应用程序（无论是客户还是服务器）提交此消息来终止一个会话；
WM_DDE_ADVISE = WM_DDE_FIRST + 2;
一个 DDE 客户程序提交此消息给一个 DDE 服务程序来请求服务器每当数据项改变时更新它
WM_DDE_UNADVISE = WM_DDE_FIRST + 3;
一个 DDE 客户程序通过此消息通知一个 DDE 服务程序不更新指定的项或一个特殊的剪贴板格式的项
WM_DDE_ACK = WM_DDE_FIRST + 4;
此消息通知一个 DDE（动态数据交换）程序已收到并正在处理 WM_DDE_POKE, WM_DDE_EXECUTE, WM_DDE_DATA, WM_DDE_ADVISE, WM_DDE_UNADVISE, or WM_DDE_INITIAT 消息
WM_DDE_DATA = WM_DDE_FIRST + 5;
一个 DDE 服务程序提交此消息给 DDE 客户程序来传递个一数据项给客户或通知客户的一条可用数据项
WM_DDE_REQUEST = WM_DDE_FIRST + 6;
一个 DDE 客户程序提交此消息给一个 DDE 服务程序来请求一个数据项的值；
WM_DDE_POKE = WM_DDE_FIRST + 7;
一个 DDE 客户程序提交此消息给一个 DDE 服务程序，客户使用此消息来请求服务器接收一个未经同意的数据项；服务器通过答复 WM_DDE_ACK 消息提示是否它接收这个数据项；
WM_DDE_EXECUTE = WM_DDE_FIRST + 8;
一个 DDE 客户程序提交此消息给一个 DDE 服务程序来发送一个字符串给服务器让它象串行命令一样被处理，服务器通过提交 WM_DDE_ACK 消息来作回应；
WM_DDE_LAST = WM_DDE_FIRST + 8;
WM_APP = $8000;
WM_USER = $0400;
此消息能帮助应用程序自定义私有消息；

http://www.cnblogs.com/tk091/archive/2012/05/11/2496787.html

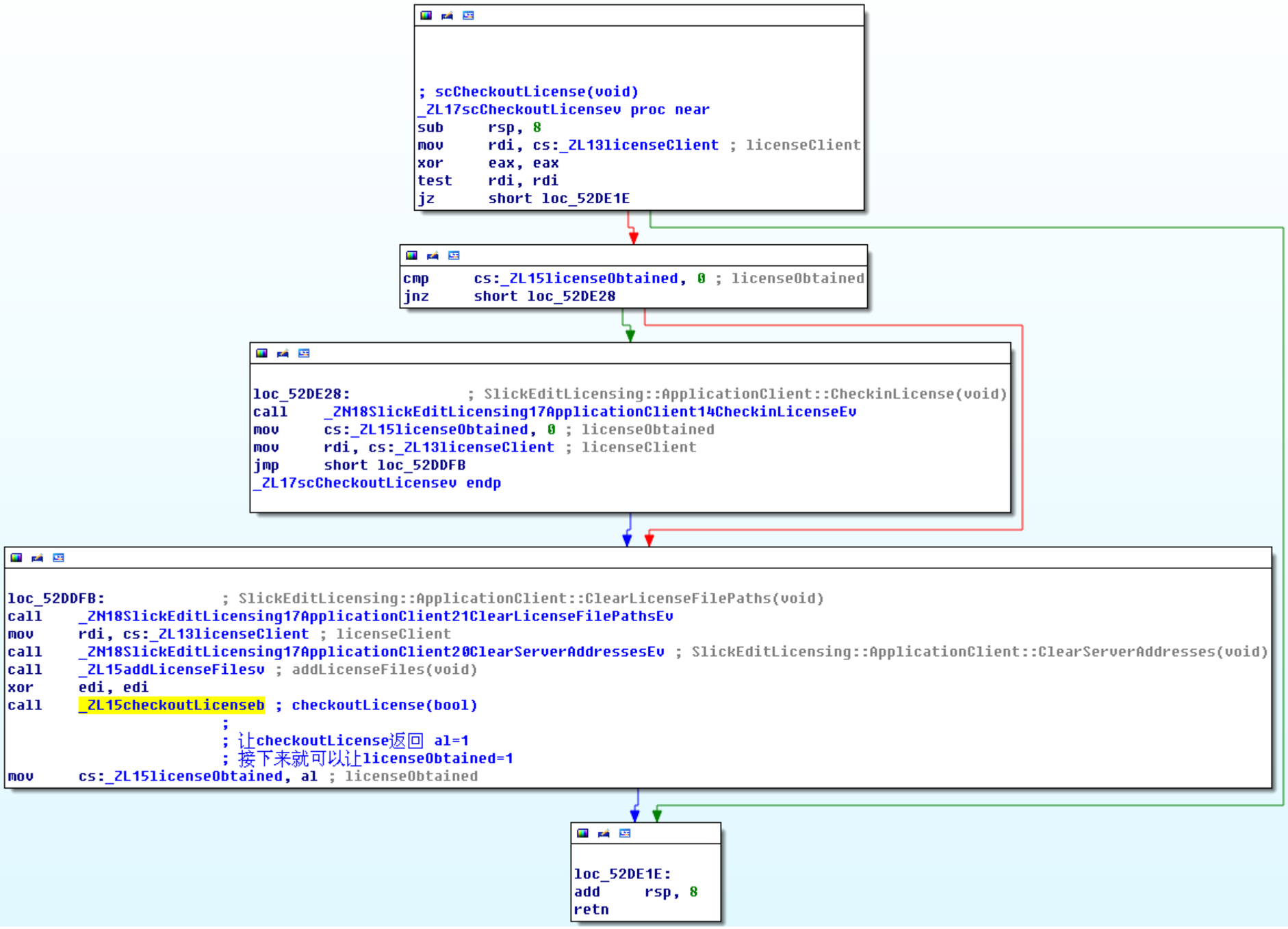
SlickeEdit 2014

SlickEdit 2014 不停地 Beta，让很多人从初夏（以往都是 5 月份左右发布新版)等到了初冬.

刘日红 和 刘雪 两位同学 告诉我 它终于发布了

看了一下， License 跟以为没有特别大的区别，感谢 SlickEdit 提供试用 License(我的修改就是让 SlickEditd 调用 vsTrialSet)

通过字符串 slickedit.lic 找到 addLicenseFile() , 从而找到 scCheckoutLicense()


; scCheckoutLicense(void)
_ZL17scCheckoutLicensev proc near
sub rsp, 8
mov rdi, cs:_ZL13licenseClient ; licenseClient
xor eax, eax
test rdi, rdi
jz short loc_52DE1E
cmp cs:_ZL15licenseObtained, 0 ; licenseObtained
jnz short loc_52DE28
loc_52DE28: ; SlickEditLicensing::ApplicationClient::CheckinLicense(void)
call _ZN18SlickEditLicensing17ApplicationClient14CheckinLicenseEv
mov cs:_ZL15licenseObtained, 0 ; licenseObtained
mov rdi, cs:_ZL13licenseClient ; licenseClient
jmp short loc_52DDFB
_ZL17scCheckoutLicensev endp
loc_52DDFB: ; SlickEditLicensing::ApplicationClient::ClearLicenseFilePaths(void)
call _ZN18SlickEditLicensing17ApplicationClient21ClearLicenseFilePathsEv
mov rdi, cs:_ZL13licenseClient ; licenseClient
call _ZN18SlickEditLicensing17ApplicationClient20ClearServerAddressesEv ; SlickEditLicensing::ApplicationClient::ClearServerAddresses(void)
call _ZL15addLicenseFilesv ; addLicenseFiles(void)
xor edi, edi
call _ZL15checkoutLicenseb ; checkoutLicense(bool)
;
; 让checkoutLicense返回 al=1
; 接下来就可以让licenseObtained=1
mov cs:_ZL15licenseObtained, al ; licenseObtained
loc_52DE1E:
add rsp, 8
retn

所以，我们让 checkoutLicnese 返回 1

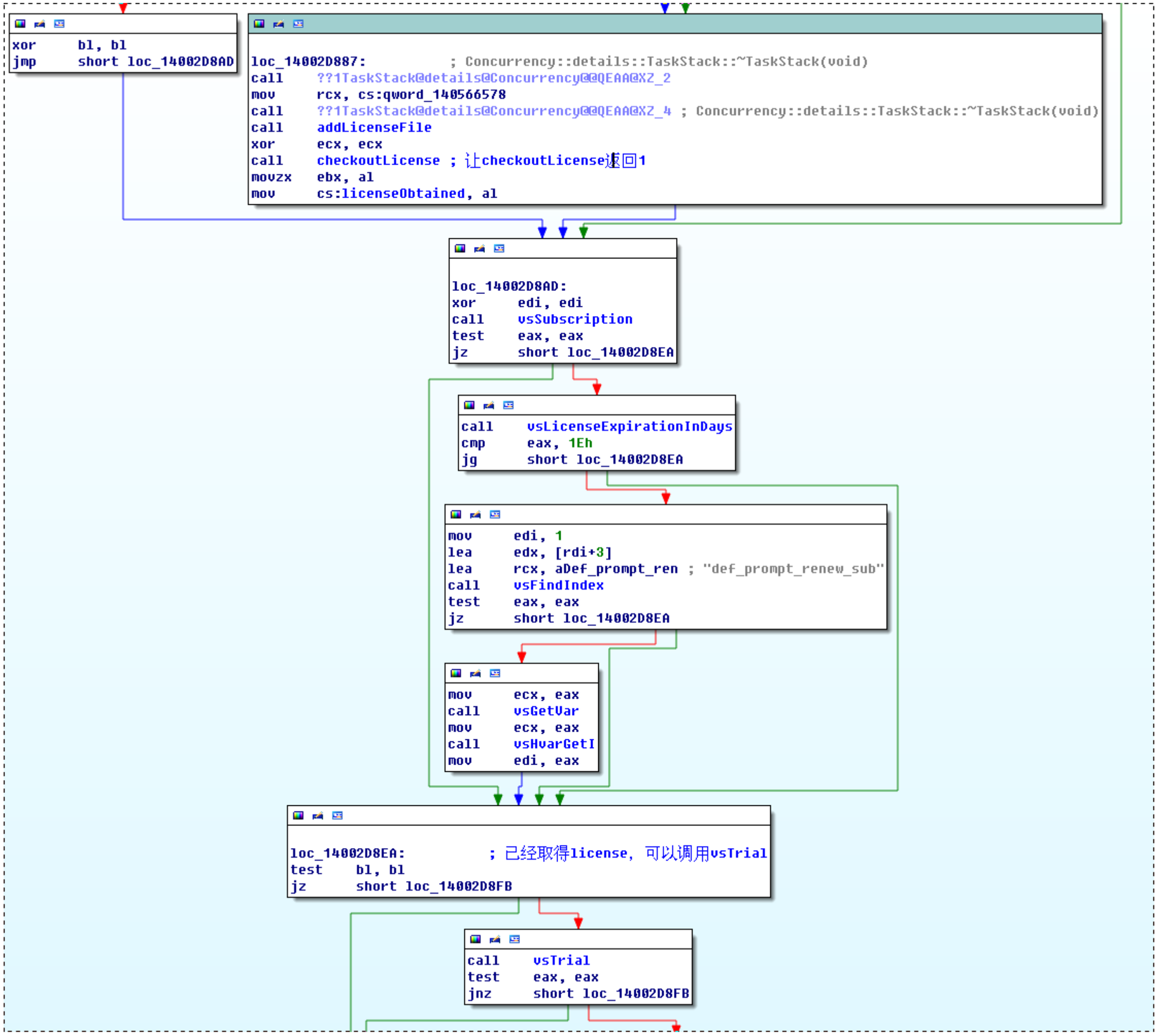
所以 Linux 64 的修改方法是

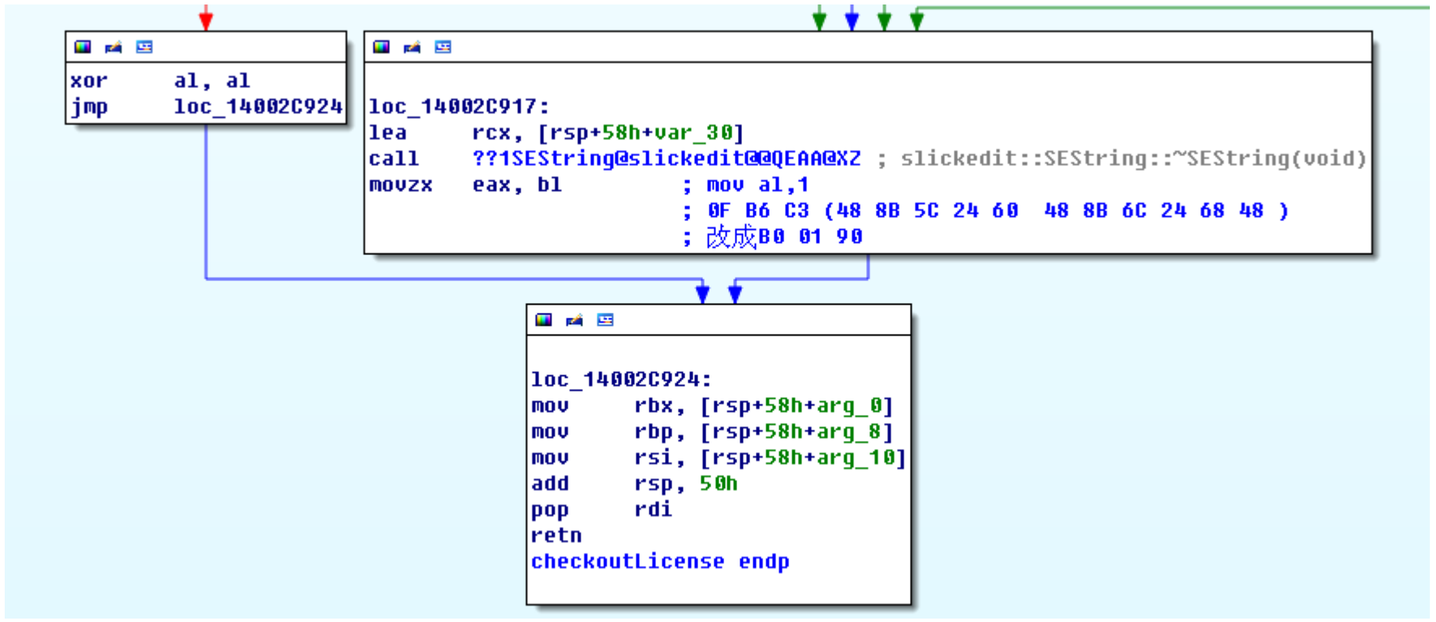
mov eax, ebx 改成 mov al, 1

也就是 89 D8 (5B 5D 41 5C 41 5D 41 5E C3 48 83 C4 20 31) 改成 B0 01

因为我在 VirtualBox 里运行了 Windows XP，所以也顺带看了下 Win32，修改方法是

8A C3 (8B 4C 24 30 64 89 0D 00 00 00 00) 改成 B0 01


xor bl, bl
jmp short loc_14002D8AD
loc_14002D887: ; Concurrency::details::TaskStack::~TaskStack(void)
call ??1TaskStack@details@Concurrency@@QEAA@XZ_2
mov rcx, cs:qword_140566578
call ??1TaskStack@details@Concurrency@@QEAA@XZ_4 ; Concurrency::details::TaskStack::~TaskStack(void)
call addLicenseFile
xor ecx, ecx
call checkoutLicense ; 让checkoutLicense返回1
movzx ebx, al
mov cs:licenseObtained, al
loc_14002D8AD:
xor edi, edi
call vsSubscription
test eax, eax
jz short loc_14002D8EA
call vsLicenseExpirationInDays
cmp eax, 1Eh
jg short loc_14002D8EA
mov edi, 1
lea edx, [rdi+3]
lea rcx, aDef_prompt_ren ; "def_prompt_renew_sub"
call vsFindIndex
test eax, eax
jz short loc_14002D8EA
mov ecx, eax
call vsGetVar
mov ecx, eax
call vsHvarGetI
mov edi, eax
loc_14002D8EA: ; 已经取得license，可以调用vsTrial
test bl, bl
jz short loc_14002D8FB
call vsTrial
test eax, eax
jnz short loc_14002D8FB



Win64 的修改方法，Chris 已经留言给出。看上图也知道改哪里

用的是 IDA Pro 6.5, 如果分析 64 位的二进制程序，需要用它的 idaq64.exe, 而不是 idaq.exe
我上面的图，是分析 SlickEdit Linux 64 的， 所以你分析 Win64 的程序找不到完全对应的，很正常
因为 Linux 64 位的 SlickEdit 带了调试信息，所以函数名都能看到
Win64 的我没看， 但是 Win32 的对应的信息少很多

SlickEdit Win64: 0F B6 C3 (48 8B 5C 24 60 48 8B 6C 24 68) 改成 B0 01 90 (48 8B 5C 24 60 48 8B 6C 24 68)

Linux 32

也就是 0F B6 44 24 1C (83 C4 5C 5B 5E 5F 5D C3 C6 44) 改成 B0 01 90 90 90

Win64

0F B6 C6 (4C 8D 9C 24 80 00 00 00) replace B0 01 90 (4C 8D 9C 24 80 00 00 00)

linux 64 v19.0.1

89 D8 5B 5D 41 5C 41 5D 41 5E 41 5F C3 0F 1F 80 00 00 00 00 (84 DB) -> 40 B0 01 5B 5D 41 5C 41 5D 41 5E 41 5F C3 90 90 90 90 90 90 (84 DB)

http://zhiwei.li/text/2014/11/slickeedit-2014/

Hopper Disassembler 修改一字节解除限制

'You cannot produce ASM with the demo version.'
'You cannot export a patched executable with the demo version'
'You cannot save with the demo version'

```
.text:000000000042E4CB                  call    sub_4DCC30
.text:000000000042E4D0                  test    al, al
.text:000000000042E4D2                  jnz     loc_42E560
.text:000000000042E4D8                  lea     rdi, aYouCannotProdu ; "You cannot produce ASM with the demo ve"...
.text:000000000042E4DF                  mov     esi, 2Dh
.text:000000000042E4E4                  call    __ZN7QString16fromAscii_helperEPKci ; QString::fromAscii_helper(char const*,int)
```

```
.text:00000000004DCC30 sub_4DCC30       proc near               ; CODE XREF: sub_42E4C0+Bp
.text:00000000004DCC30                                          ; sub_42E5E0+Bp ...
.text:00000000004DCC30                  sub     rsp, 8
.text:00000000004DCC34                  call    sub_4DC930
.text:00000000004DCC39                  and     eax, 0FFFFFFFDh
.text:00000000004DCC3C                  cmp     eax, 1          ; 改成 cmp, eax, 0
.text:00000000004DCC3C                                          ; 也就是  83 F8 01 改成  83 F8 00
.text:00000000004DCC3F                  setz    al
.text:00000000004DCC42                  add     rsp, 8
.text:00000000004DCC46                  retn
.text:00000000004DCC46 sub_4DCC30       endp
```

http://zhiwei.li/text/2013/10/hopper-disassembler%E4%BF%AE%E6%94%B9%E4%B8%80%E5%AD%97%E8%8A%82%E8%A7%A3%E9%99%A4%E9%99%90%E5%88%B6/

Genymotion 破解

genyshell

提示：检查有效的 License

Prompt::checkValidLicense(void)

关键字符串
‘No valid license found. This command is not available in Free mode.’

会被
Prompt::cmdDevicesResetFactory(QStringList const&)
调用

==========
PlayerApp::createPlayer(QString const&)
Player::Player(QSize const&, int, VMToolsEngine *, VMToolsMachine *)
Player::checkToken(void)
==========
ActivationToken::getTokenValidity(void)const

WidgetToolbar::checkToken(void)　　改 IMEI 等其他工具
DeviceToolbar::checkToken(void)　　像素对齐
DeviceScreen::checkToken(void)　　修改后去水印

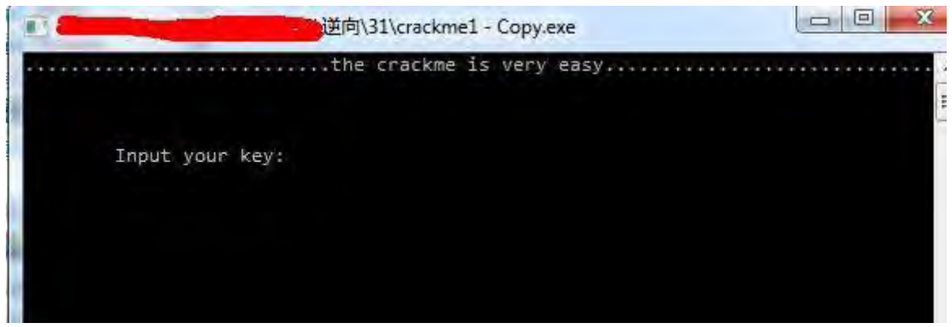
==========
ActivationToken::getLicenseType(void)

http://zhiwei.li/text/2014/06/genymotion%E7%A0%B4%E8%A7%A3/注意：下面的代码都是 thumb 指令，每条指令占 2 个字节

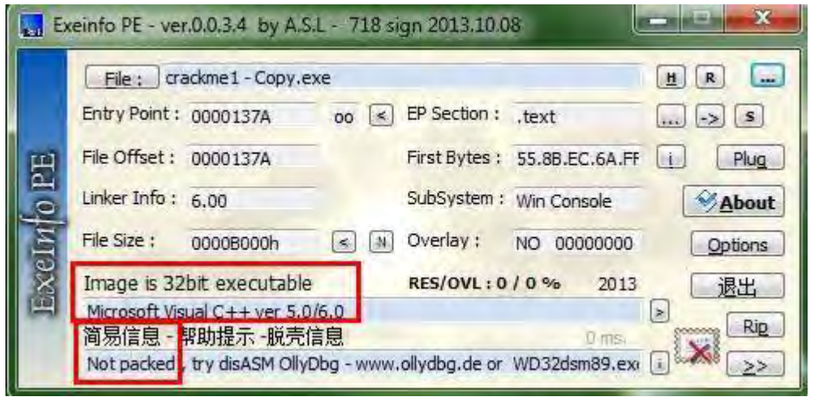
IDA Pro 逆向实战之 Crackme（简单篇）

今天手闲的很，没事就拿出了以前没解决的逆向题来做一下，具体的源程序在附件里，废话少说，直接上菜：

0. 源程序运行效果(输入不对的，直接退出)：

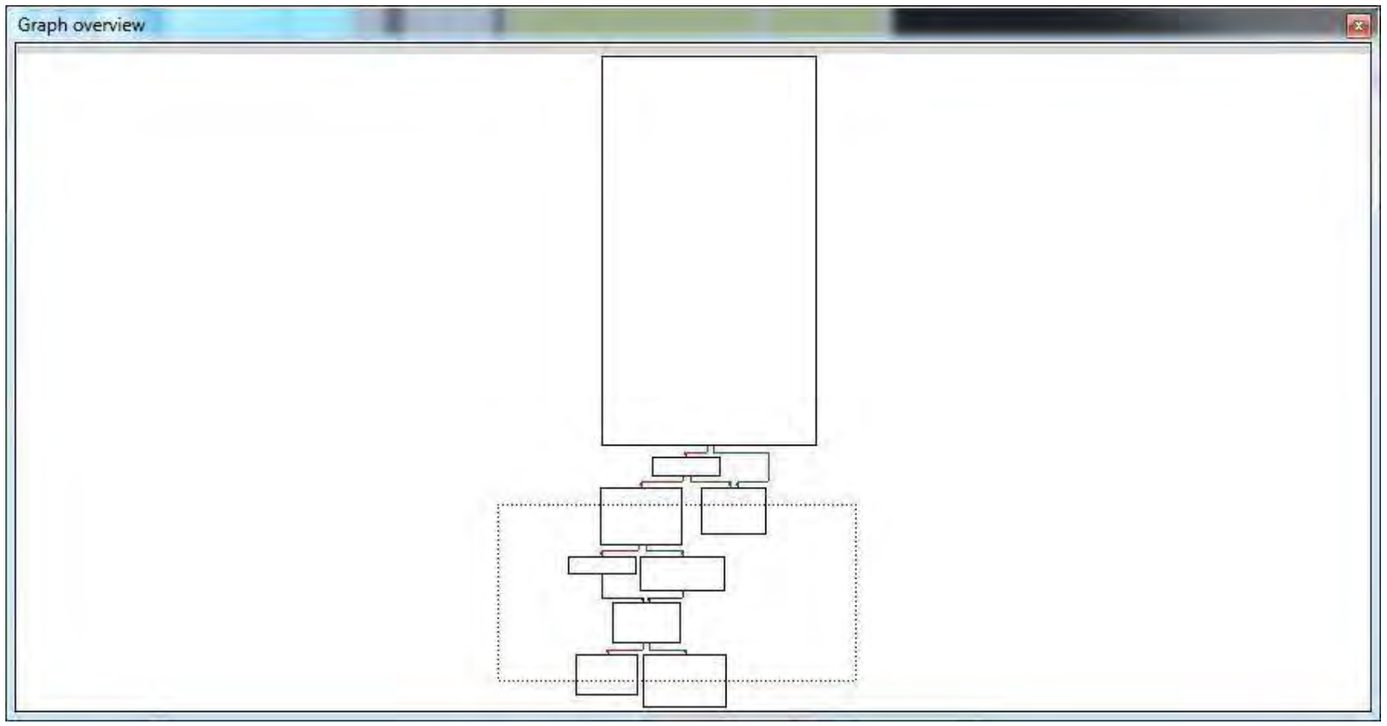


1. Exeinfo PE 查壳：



看到是无壳的小程序，看来练手一定会很容易！！！那接下来直接载入到 IDA 中看看程序执行逻辑吧！

2. IDA Pro 查看程序执行逻辑



这里截的是一张白图，不过对于分析程序流程已经足够了，顺便解释下图里的红线是 False 分支执行的，绿线是 True 分支执行的，大致看下这里的分支不是很多，所以这个程序逆起来应该不是很复杂！

3. 神奇的 IDA F5 直接看源码吧

Pseudocode-B

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v3; // ST08_4@1
  int v4; // ecx@1
  int result; // eax@7
  int v6; // [sp+0h] [bp-68h]@8
  char v7; // [sp+4h] [bp-64h]@1

  sub_401349((int)".....the crackme is very easy.....\n\n\n", v6);
  sub_401349((int)"\tInput your key: ", v3);
  scanf("%s", &v7);
  v4 = strlen(&v7);
  if ( v4 < 8 || v4 > 15 )
  {
    result = -1;
  }
  else
  {
    sub_4010C0(v4, &v7);
    --File._cnt;
    if ( File._cnt < 0 )
      _filbuf(&File);
    else
      ++File._ptr;
    --File._cnt;
    if ( File._cnt < 0 )
    {
      _filbuf(&File);
      result = 0;
    }
    else
    {
      ++File._ptr;
      result = 0;
    }
  }
  return result;
}
```

打印输出文字

接受数据

计算数据长度

判断输入的字符串的长度是在小于8或者大于15的情况下直接返回

关键的要执行的验证函数，顺便说一下啊，这里的sub_开头的函数都是用户自己写的函数，如果是有规范名字的函数都是C语言的库函数或者API之类的，所以我们主要关注点应该放在用户自定义函数上，当然这个代码前后有一定数量的冗余代码，需要人为判断过滤掉，如这个程序，实际就这一个函数有作用！！

我晕啊，这里的辛辛苦苦编辑的高清图片，最后只能是上面这个样子，讲究这看吧，分析出了关键函数，下面是男人就跟进来吧！

4. 点击关键函数，进入验证段代码，还是直接看反编译源码吧

Pseudocode-A

```
char __cdecl sub_4010C0(int a1, const char *a2)
{
  int v2; // eax@1
  int i; // eax@2
  char v5; // [sp+0h] [bp-14h]@1
  char v6; // [sp+1h] [bp-13h]@1
  char v7; // [sp+2h] [bp-12h]@1
  char v8; // [sp+3h] [bp-11h]@1
  char v9; // [sp+4h] [bp-10h]@1
  char v10; // [sp+5h] [bp-Fh]@1
  char v11; // [sp+6h] [bp-Eh]@1
  char v12; // [sp+7h] [bp-Dh]@1
  char v13; // [sp+8h] [bp-Ch]@4
  char v14; // [sp+9h] [bp-Bh]@4
  char v15; // [sp+Ah] [bp-Ah]@4
  char v16; // [sp+Bh] [bp-9h]@4
  char v17; // [sp+Ch] [bp-8h]@4
  char v18; // [sp+Dh] [bp-7h]@4
  char v19; // [sp+Eh] [bp-6h]@4
  char v20; // [sp+Fh] [bp-5h]@4
  char v21; // [sp+10h] [bp-4h]@4
  char v22; // [sp+11h] [bp-3h]@4
  char v23; // [sp+12h] [bp-2h]@4

  v7 = 67;
  v8 = 67;
  LOBYTE(v2) = 83;
  v5 = 83;
  v6 = 85;
```

```
  v9 = 69;
  v10 = 83;
  v11 = 83;
  v12 = 0;
  if ( a2 )
  {
    for ( i = 0; i < a1; ++i )
      a2[i] = (a2[i] - 5) ^ 0x11;
    v13 = 82;
    v14 = 101;
    v15 = 118;
    v16 = 101;
    v17 = 114;
    v18 = 115;
    v19 = 101;
    v20 = 32;
    v21 = 77;
    v22 = 101;
    v23 = 0;
    v2 = strcmp(a2, &v13);
    if ( !v2 )
      LOBYTE(v2) = sub_401349((int)"\n\t%s", (int)&v5);
  }
  return v2;
}
```

关键变换

内置将要比较的字符串"Reverse Me"

如果相同就进入下一个函数，这里下一个函数就是打印成功字样！

sub_4010C0:25

看到这里手都痒了，赶紧用 C 语言写个反向算法，把他的关键字符串逆回去！

5. C 语言编写的的简单逆向算法

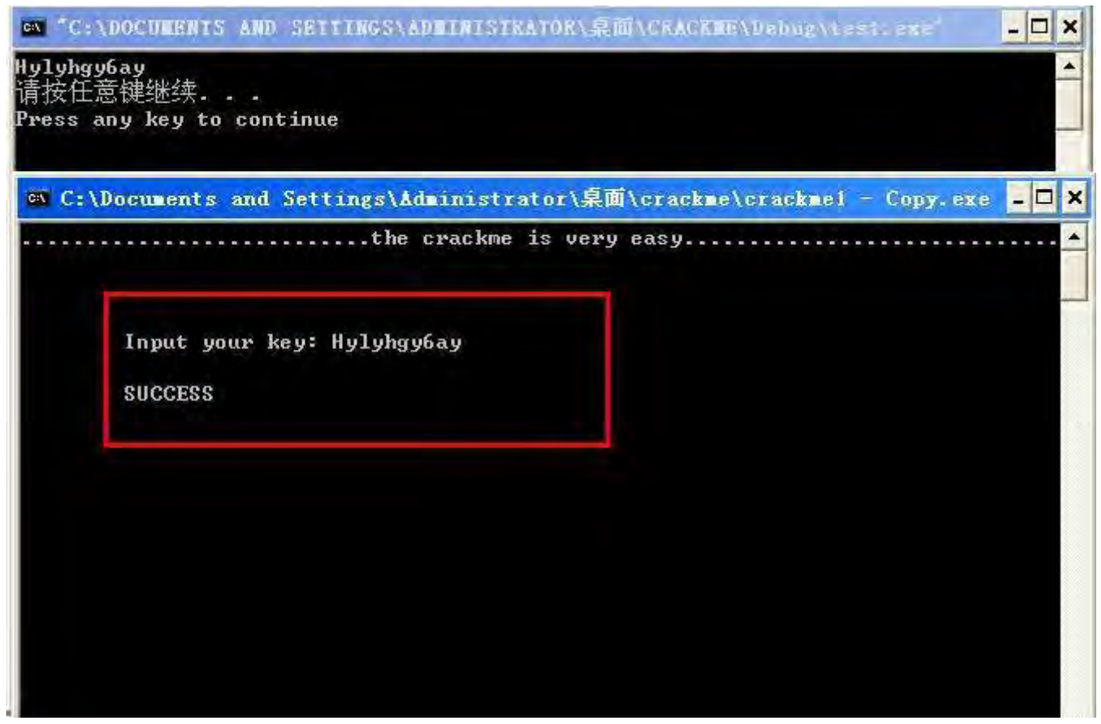

test - Microsoft Visual C++ - [test.c]
文件(F) 编辑(E) 查看(V) 插入(I) 工程(P) 组建(B) 工具(T) 窗口(W) 帮助(H)
(Globals)
(All global members
main
test classes
ClassV...
FileView
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
char v13[] ="Reverse Me";
unsigned int i = 0;
for (; i < strlen(v13); i++)
{
v13[i] = (v13[i]^0x11)+5;
}
printf("%s\n",v13);
system("Pause");
}
逆向算法，虽然只有一句
大同小异！

执行结果：


"C:\DOCUMENTS AND SETTINGS\ADMINISTRATOR\桌面\CRACKME\Debug\test.exe"
Hylyhgy6ay
请按任意键继续. . .
Press any key to continue

填进去看看结果呐！

6. 最后的胜利！



最后，这么搓的小程序都不好意思拿出手，但还是希望大家多多讨论。

（原创文章，作者：ByteWay）这里是我自己的论坛,欢迎大家来逛：http://itpark.sinaapp.com/thread-index-fid-4-tid-121.htm

http://blog.csdn.net/l_f0rm4t3d/article/details/37747285?utm_source=tuicool

Applied Cracking & Byte Patching with IDA Pro

In the previous IDA Pro article, we took a look at the basics of reverse engineering source code and binary files. This rare dissertation committed to impart cracking and byte patching in a binary executable using IDA Pro with the intention of subverting various security constraints as well as generating or producing the latest modified version (Patched) of that particular binary. IDA Pro is typically utilized to analyze the disassembled code of a binary so that the internal mechanism could be comprehended and identify the inherent vulnerability in the source code.

This article comprises subsequent contents:

- Binary Sample
- Patching Configuration in IDA Pro
- Binary Analysis
- Binary Cracking & Patching with IDA Pro
- Script Patching Substitute
- Final Note

IDA Pro appears to have managed mystical potentials in the reverse engineer's mind by having the impression that merely opening a binary with IDA will reveal all the secrets of a target file. IDA Pro is intended to assist you in considering the behavior of a binary by offering us disassembled code. IDA Pro is in fact not designed to modify or patch the binary code to suit your needs like other tools such as OllyDbg and CFF Explorer. It is really only a static-analysis disassembler tool. It can only facilitate your attempts to locate software vulnerabilities, bugs and loopholes which are typically utilized by both white hat and black hat professionals. Ultimately, it is up to your skills and how you apply them as to whether IDA makes your search for vulnerabilities easier.

Essential

This tutorial requires thorough knowledge of Assembly Programming and Hex Code manipulation because patching binary with IDA Pro especially deals with assembly opcode instructions. Besides that, the reverse engineer is supposed to operate the IDA Pro Software IDE features perfectly. This operation lists the following tools of the trade:
The Target Binary (C /C++ code)
IDA Pro Interactive Dissembler
IDA-Script File (*.idc files)
Assembly Language skills
ASCII Converter

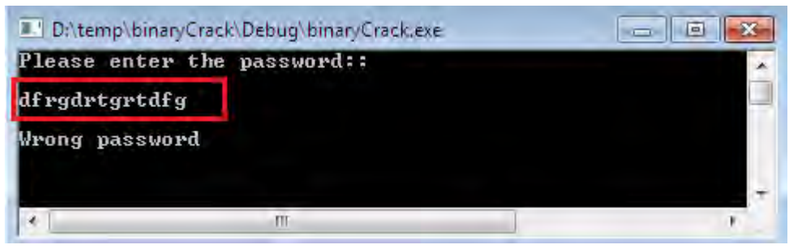
Binary Sample

This article exposes the demonstration of byte patching over a typical C++ binary which essentially required a user password to validate his identity and let him log into the system, and such confidential information is only provided to the registered person indeed. There is of course no direct method to breach into this application without being authenticated, except to reverse engineer or patch the critical bytes which are responsible for performing validation. The following code will make the binary executable live as binaryCrack.exe:

```
#include "stdafx.h"
```

```
#include
#include
#include
#include
#include

#define password "ajay"

int _tmain(int argc, _TCHAR* argv[])
{
char pwd[100];
printf("Please enter the password::\n\n");
scanf("%s", pwd);
if ( strcmp(pwd, password) == 0 )
{
printf("Congratulation!!\n\n");
printf("Ready to login with: %s",password);
}
else
{
printf("Wrong password");
}
getch();
return 0;
}
```

We can use any compiler to execute the aforementioned binary source code which makes an executable as binaryCrack.exe, and when we run that file, it will prompt to enter the password. If we enter the correct password as "*ajay*" then it shows the congratulations message, otherwise it will issue the wrong password message as following.



It is probable that we might not have the real password and in such circumstances we can't proceed without this information. So the only option left is Reverse Engineering this binary and manipulating the sensitive bytes to suit our needs as we shall see in the next sections.

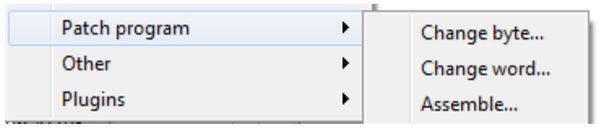
Patching Configuration in IDA Pro

The patching or editing assembly code features are normally invisible in the IDA Pro. You can double check from the *Edit* menu that no *Patch program* options appeared. Thus, in order make this option visible, open the *idagui.cfg* configuration file of IDA Pro which is located at *Drive: \Program Files\IDA PRO Advanced Edition\cfg* folder and scroll down to find the DISPLAY_PATCH_SUBMENU option which is typically set to NO. So, do the following changes and save this file.

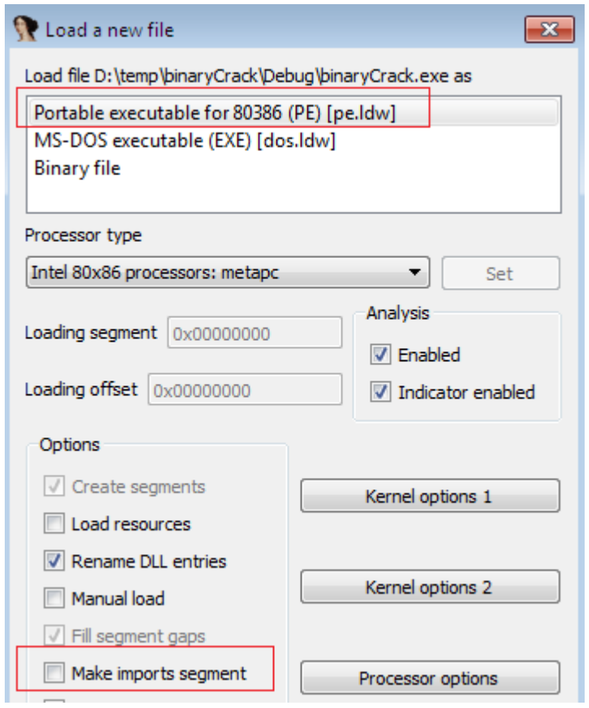
```
DISPLAY_PATCH_SUBMENU     = YES
DISPLAY_COMMAND_LINE      = NO
```

```
// Display the Edit,Patch submenu
// Display the expressions/IDC command line
// To turn on/off the command line,
// right click on the main toolbar after
// setting this parameter to YES
```

After saving this file, re-launch the IDA PRO and the moment you change the submenu option in the configuration file, you can notice that *Patch program* option becomes visible in the *Edit* menu.

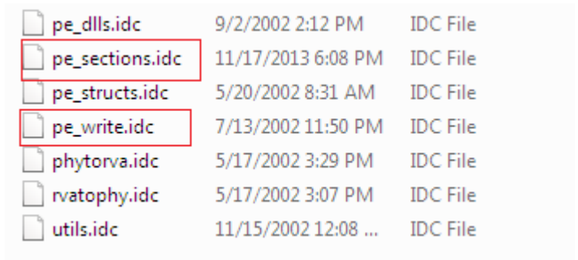


Now, load the target binary into IDA Pro. It will ask to create a new database as usual, then we select the PE file option as shown in the following figure. An important point to remember: don't forget to uncheck the *Make imports segment* option, because some useful data can be lost from the database if this option is enabled.

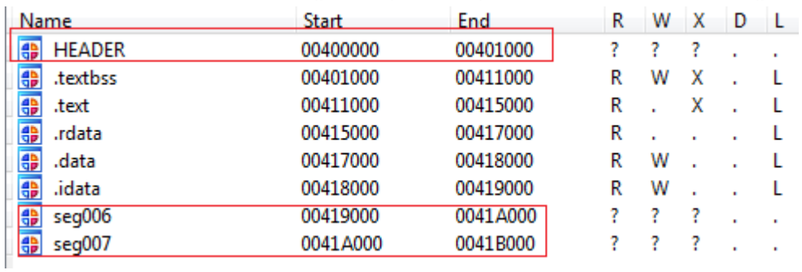


The target file will be loaded into the IDA Pro, but we could still not modify the byte sequence of the binary file even if enabling the *Patch program* option in the *Edit* menu earlier. So here, the role of special IDA script files comes into light as they are able to modify the byte like OllyDbg as well as write the changes into the executable to make the effect permanent.

The IDC script files can be downloaded from this URL as http://www.openrce.org/downloads/details/57/PE_Scripts. When downloaded, extract the files in a separate folder on the file system of your machine. A few script files are provided, but mainly two script files are significant:



After loading the target binary into the IDA Pro, open the folder where the aforesaid IDA script files are located and execute the *pe_sections.idc* file in order to extend new functionality into IDA Pro such as binary patching and writing. You can ensure the new specification from the Segments (shift + F7) windows that certain new segments are automatically added.



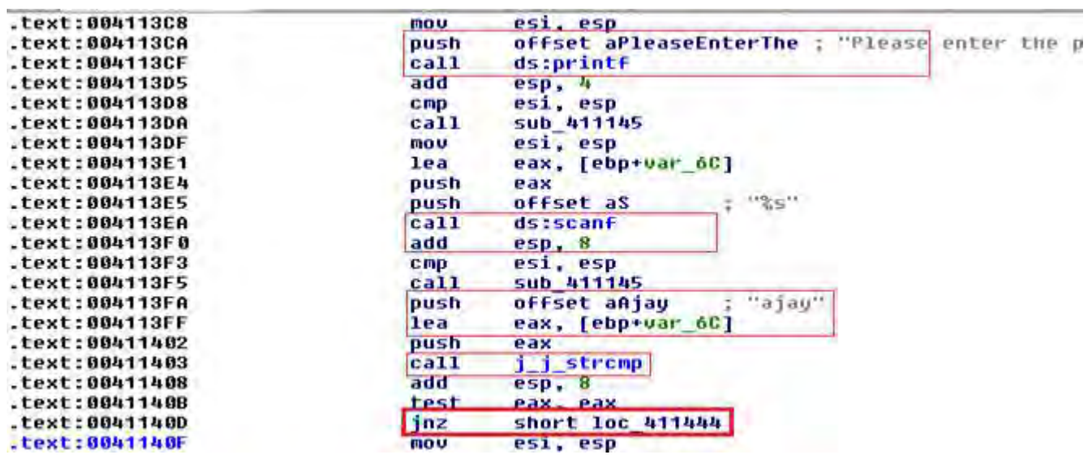
Name	Start	End	R	W	X	D	L
HEADER	00400000	00401000	?	?	?	.	.
.textbss	00401000	00411000	R	W	X	.	L
.text	00411000	00415000	R	.	X	.	L
.rdata	00415000	00417000	R	.	.	.	L
.data	00417000	00418000	R	W	.	.	L
.idata	00418000	00419000	R	W	.	.	L
seg006	00419000	0041A000	?	?	?	.	.
seg007	0041A000	0041B000	?	?	?	.	.

After successfully completion of such aforesaid operations, we can modify as well as write the byte sequence into binary file.

Binary Analysis

We have only the binary executable and it is almost impossible to know about the logic implementation without the source code. But we can disassemble the source code of any binary by employing IDA Pro, because unless we are aware of logic flow, how can we subvert any security mechanism? Hence, let's analyze the proper logic flow path of the binary file.

As we can consider the following image, IDA Pro disassembles the binary into raw assembly instruction sets. This program first prompts the user to enter the password by displaying a string message, then compares this value to a predefined value "ajay" which might be the real password. The comparison happens via string class *strcmp* method, and if the value is 0 then the entered value is correct; otherwise it is incorrect.



```
.text:004113C8          mov     esi, esp
.text:004113CA          push    offset aPleaseEnterThe ; "Please enter the p
.text:004113CF          call    ds:printf
.text:004113D5          add     esp, 4
.text:004113D8          cmp     esi, esp
.text:004113DA          call    sub_411145
.text:004113DF          mov     esi, esp
.text:004113E1          lea     eax, [ebp+var_6C]
.text:004113E4          push    eax
.text:004113E5          push    offset aS       ; "%s"
.text:004113EA          call    ds:scanf
.text:004113F0          add     esp, 8
.text:004113F3          cmp     esi, esp
.text:004113F5          call    sub_411145
.text:004113FA          push    offset aAjay    ; "ajay"
.text:004113FF          lea     eax, [ebp+var_6C]
.text:00411402          push    eax
.text:00411403          call    j_j_strcmp
.text:00411408          add     esp, 8
.text:0041140B          test    eax, eax
.text:0041140D          jnz     short loc_411444
.text:0041140F          mov     esi, esp
```

If the value of *eax* register is 1 then the execution is shifted towards the loc_411444 block by *jnz* statement, where the "wrong password" message would be echoed in the screen as follows:

```
text:00411444 loc_411444:                             ; CODE XREF: sub_4113A0+6D↑j
text:00411444                 mov     esi, esp
text:00411446                 push    offset aWrongPassword ; "Wrong password"
text:0041144B                 call    ds:printf
text:00411451                 add     esp, 4
text:00411454                 cmp     esi, esp
text:00411456                 call    sub_411145
```

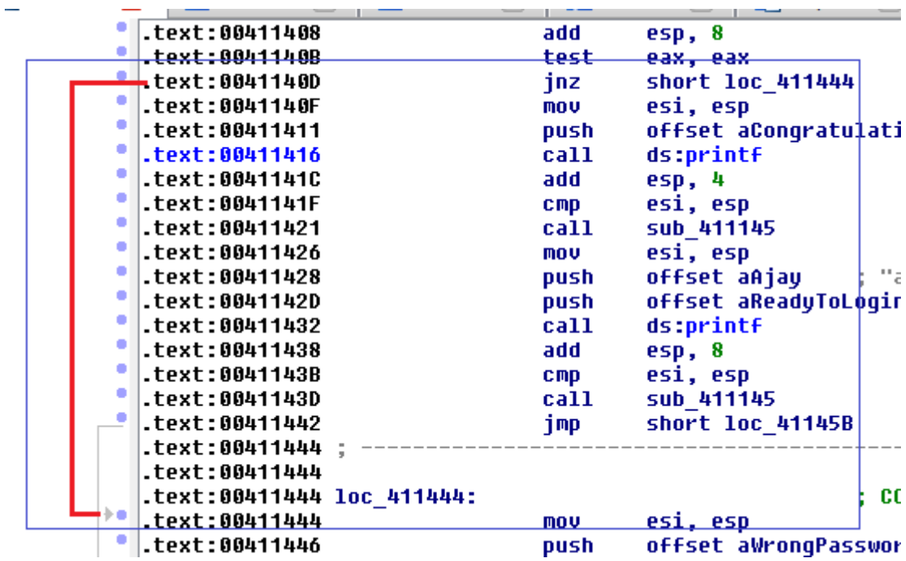
And if the value of eax is 0 then jump won't throw the execution anywhere. The program informs us that we have entered the correct password:

```
mov     esi, esp
push    offset aCongratulation ; "Congratulation?!\n\n"
call    ds:printf
add     esp, 4
cmp     esi, esp
call    sub_411145
mov     esi, esp
push    offset aAjay    ; "ajay"
push    offset aReadyToLoginWi ; "Ready to login with: %s"
call    ds:printf
add     esp, 8
cmp     esi, esp
call    sub_411145
jmp     short loc_41145B
```

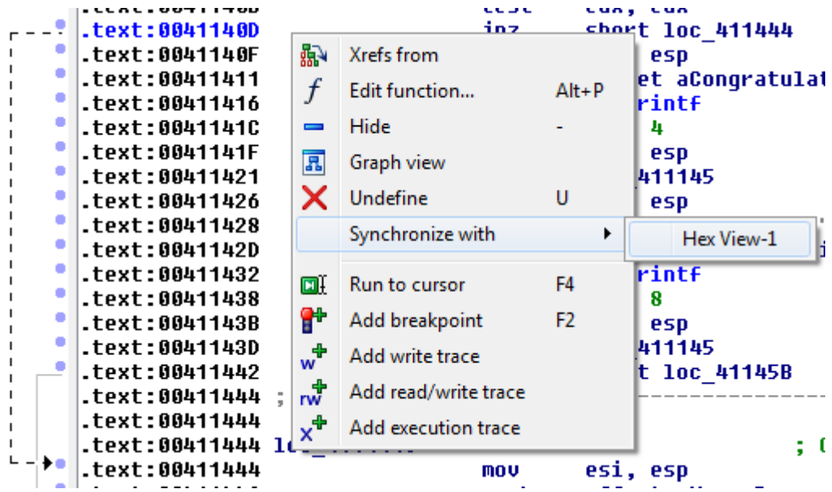
We can easily anticipate that the Jump instruction determines either the password is correct or incorrect. If the *eax* register has value 1 then execution is diverted towards the "wrong password" box flow, otherwise the "congratulations" message would be displayed. As we are implying from execution flow, everything depends on *jnz* instruction. So, a reverse engineer would surely be interested in this instruction because manipulation of this instruction can produce different results.

Binary Cracking and Patching

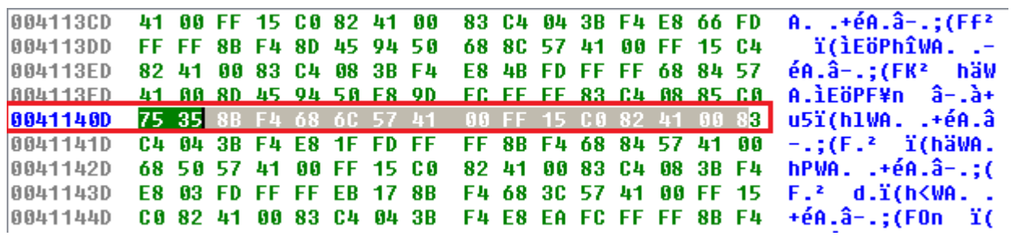
As we have concluded from the analysis, the outcome of this binary application is regulated by the *jnz short 1oc_411444* statements. The 1oc_411444 instruction belongs to the unmatched password display block, and execution is directly jumped to this block, as you can notice by the RED arrow.



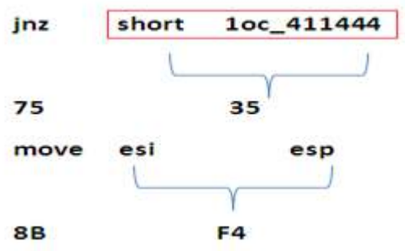
However, all we have to do is to change the corresponding *jnz* statement related bytes. So right click on the current location of this statement and synchronize it with Hex Editor View in order to examine this statement byte sequence:



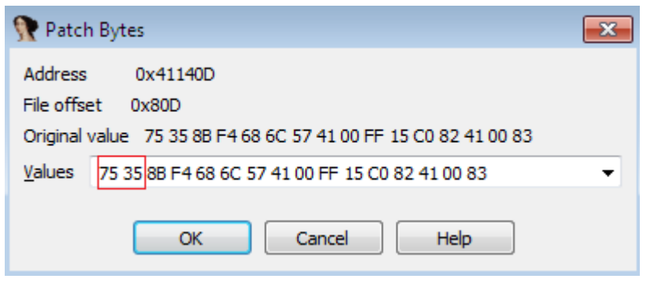
The hex view offers a 16 bytes sequence in one line and each two bytes represent only one assembly instruction set such that the hex value 75 35 belongs to .text: 0041140D address location where assembly code jnz short 10c_411444 is implemented as following.



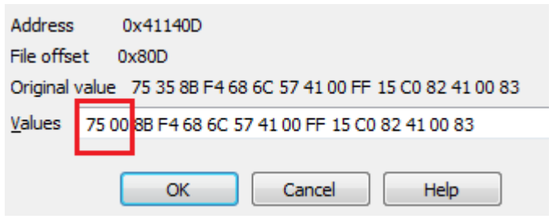
So we have to identify the correct bytes to the specific instructions so that we can modify them to suit our needs. The following figure showcasing the hex code refers to which instruction.



Finally, we have concluded that hex code 35 is the key value that directing the execution flow of the program. However, select the address location 0041140D into text view and go to *Edit* menu, choose *Patch program* and select *Patch Bytes* over there. It will show the entire 16 hex bytes sequence alike to hex view:



In order to diffuse the effect of the *jnz* statement where the execution is shifting towards the 1oc_411444 block, we have to change its corresponding hex value 35 to 00 which typically fills a nothing action in the memory.



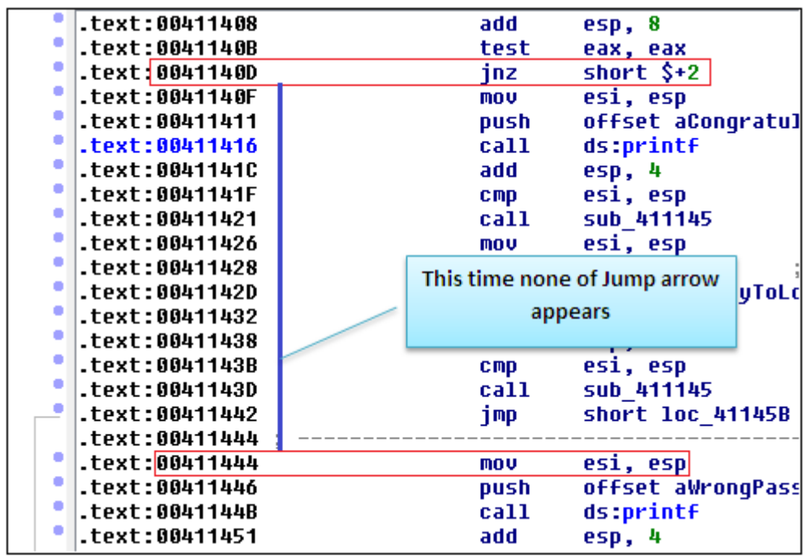
The moment you change the 1oc_411444 instruction code to 00, it will also reflect in the assembly:

```
.text:0041140B                test    eax, eax
.text:0041140D                jnz     short $+2
.text:0041140F                mov     esi, esp
.text:00411411                push    offset aCongratulation ; "Congratulation!
```

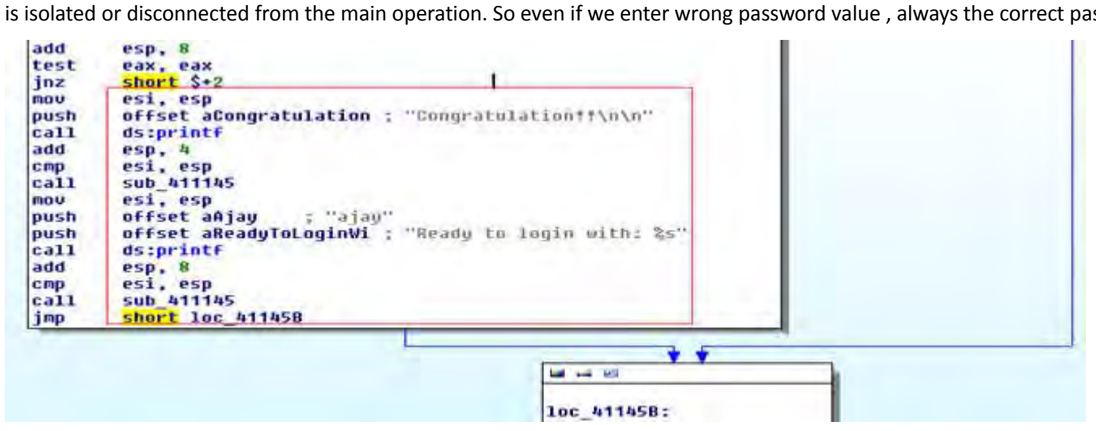
You can also notice the modification in the hex view:

```
004113FD  41 00 8D 45 94 50 E8 9D
0041140D  75 00 8B F4 68 6C 57 41
0041141D  C4 04 3B F4 E8 1F FD FF
```

One of the important changes to notice is that after making the value 00, no jump arrow is showing, which was pointing the execution divert to 1oc_411444 earlier.



Now open the graph view and notice that the “congratulations” block is merged into the main code rather than being separated as before editing the hex value. The BLUE arrow finally goes to end of statement block and the “wrong password” block is isolated or disconnected from the main operation. So even if we enter wrong password value , always the correct password block would be executed because *jnz* statement code is diffused to 00 and the 1oc_411444 block is disconnected:

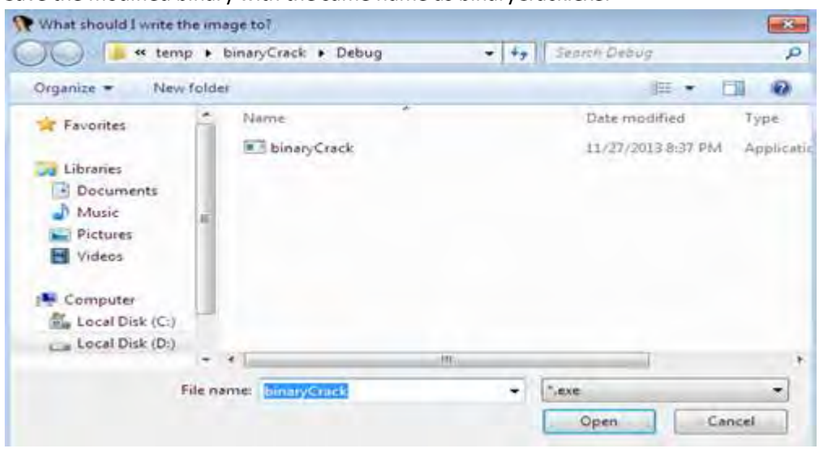


Okay, we have done the byte editing, so it’s time save the effect permanently into memory, but IDA Pro is not capable of writing bytes of the binary file into memory; instead it can write altered bytes into a database. So for this purpose, IDA special script file are used here. Go to *File* and select *Script file* and choose *pe_write.idc* which makes the perpetual effects in the memory:

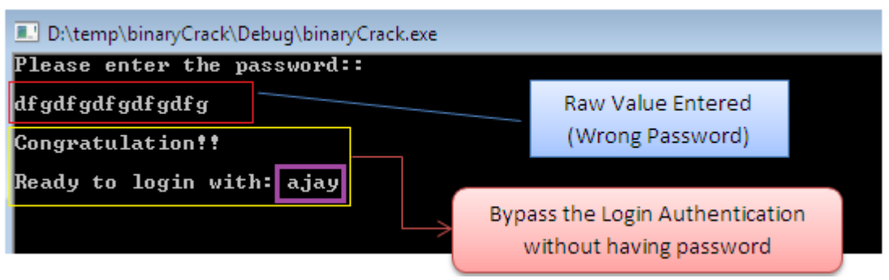
The moment you run the pe_write.idc file, you will notice that bytes have been written to segments successfully, and lastly, IDA will prompt to re-save the binary file:

```
 Sections written out:

    HEADER : [00400000] --> [00000000, 00000400]
  .textbss : [00401000] --> [00000000, 00000000]
     .text : [00411000] --> [00000400, 00003600]
    .rdata : [00415000] --> [00003A00, 00001E00]
     .data : [00417000] --> [00005800, 00000200]
    .idata : [00418000] --> [00005A00, 00000A00]
    seg006 : [00419000] --> [00006400, 00000600]
    seg007 : [0041A000] --> [00006A00, 00000600]

 The file has been written out
--------------------------------------------------
```

Save the modified binary with the same name as binaryCrack.exe:

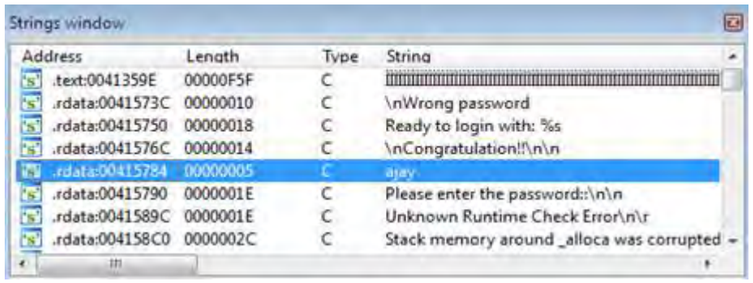


Now, load the binaryCrack.exe and it will prompt to enter the password, merely enter any value and BINGO!!!!!!! Congratulations message along with original password appearing. We have successfully bypassed or subverted the password mechanism by patching some related critical bytes using IDA Pro.



Patching String Bytes

As we can observe in the binaryCrack.exe, couples of strings messages are showing. We can access all these strings into place via String window (shift + F12) and can directly reach to its assembly code merely by clicking the string.



It is not a good programming practice to show the sensitive strings likes serial keys or passwords directly. Such information should be hidden because after patching (reverse engineering) the password's assembly code, the hacker can easily aware that the password is "*ajay*" as it is showing after the "congratulations" message. So we can hide or rename the password on the screen by patching its corresponding bytes.

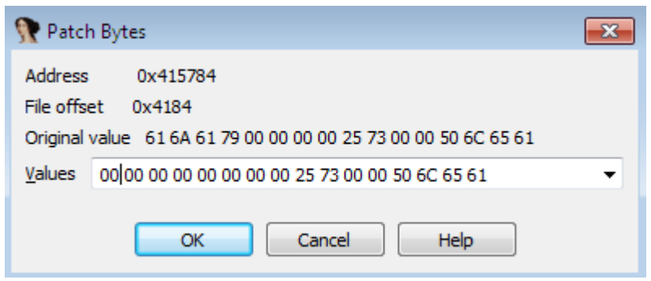
So, double click over the *"ajay"* in the String windows, and IDA will let us reach its assembly code where we can modify its visibility:

```
.rdata:00415781                 db    0
.rdata:00415782                 db    0
.rdata:00415783                 db    0
.rdata:00415784 aAjay           db 'ajay',0
.rdata:00415784
.rdata:00415789                 align 4
```

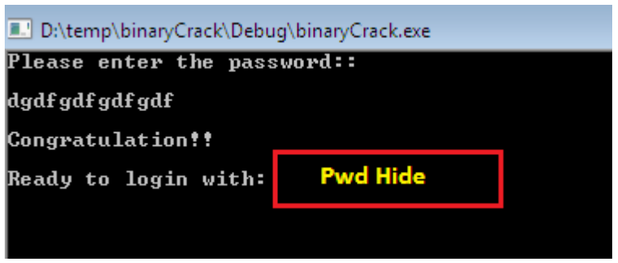
Thereafter open the Hex view and we can observe the "*ajay*" byte code sequence as 61 6A 61 79 00 as following:

```
61 73 73 77 6F 72 64 00  00 00 00 00 52 65 61 64
79 20 74 6F 20 6C 6F 67  69 6E 20 77 69 74 68 3A
20 25 73 00 00 00 00 00  0A 43 6F 6E 67 72 61 74
75 6C 61 74 69 6F 6E 21  21 0A 0A 00 00 00 00 00
61 6A 61 79 00 00 00 00  25 73 00 00 50 6C 65 61
73 65 20 65 6E 74 65 72  20 74 68 65 20 70 61 73
73 77 6F 72 64 3A 3A 0A  0A 00 00 00 00 00 00 00
```

Again open the *Patch Bytes* from the *Patch program* in the *Edit* menu and replace all these bytes with 00 as following:



Now once again, run the pe_write.idc to make changes perpetual in the binary file and the binaryCrack.exe. Enter any value as the password and observe that this time the real password is not showing:

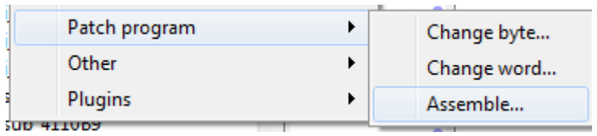


Removing Segments

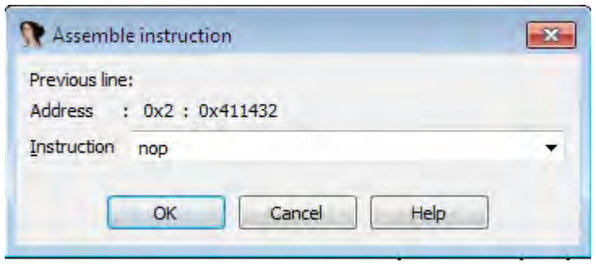
A question comes into mind: why should we display the string message after the “Congratulations” message when we subverted the password mechanism? We cannot execute a particular assembly code by using IDA Pro. Here, we can easily figure out that the code instruction after address location 00411411 is senseless to be displayed as:

```
.text:0041140F          mov     esi, esp
.text:00411411          push    offset aCongratulation ; "\nCongratulation!!
.text:00411416          call    ds:printf
.text:0041141C          add     esp, 4
```

This time we are not patching the bytes, instead we are integrating new assembly code so that any string message won’t display after the “Congratulations” message. Hence, go to *Patch program* and choose *Assemble* as following:



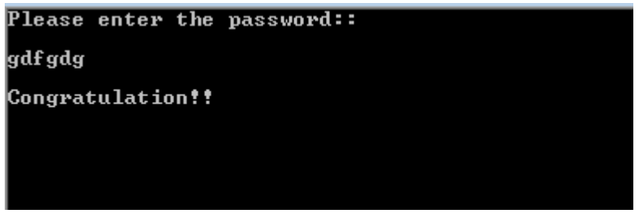
Now place *nop* assembly instruction which stands for no operation and it decides that none of the code would be executed as following:



After this, you can notice that *nop* is placed after 0041132 locations:

```
.text:0041142D          push    offset aReadyToLoginW:
text:00411432           nop
.text:00411433          adc     eax, offset printf
```

Now, run pe_write.idc again and notice that none of code is executing after the message:



Script Patching Substitute

It is not necessary that we only patch bytes by using the IDA Script file which exports the current IDA's database into the EXE binary executable. Rather, we can opt for another approach. First make changes in the byte or hex code just by editing them and produce a new DIF file as *test.dif*. Now open the test.dif file and it shows the original hex code and patched code:

```
This difference file has been created by IDA Pro
binaryCrack.exe
0000080E: 35      00
```

Patched Byte

Later compile the following C program by using any editor (I suggest use GCC in the linux platform).

```
#include

int main(int argc, char **argv) {
char line[256];
FILE *patch = stdin;
FILE *input = NULL;
unsigned int offset;
int orig;
int newval;

int i;

for (i = 1; i &lt; argc; i += 2) {
if (!strcmp(argv[i], "-p")) {
if ((i + 1) &lt; argc) {
FILE *f = fopen(argv[i+1], "r");
if (f) {
patch = f;
}
else {
fprintf(stderr, "Failed to open patch file %s\n", argv[i+1]);
exit(0);
}
}
}
else if (!strcmp(argv[i], "-i")) {
if ((i + 1) &lt; argc) {
fprintf(stderr, "Opening %s\n", argv[i+1]);
input = fopen(argv[i+1], "rb+");
```

```
if(input == NULL) {
fprintf(stderr, "Failed to open input file %s\n", argv[i+1]);
exit(0);
}
}
}
else{
fprintf(stderr, "usage:\n\t%s [-i ] [-p ]\n", argv[0]);
fprintf(stderr, "\t%s [-p ]\n", argv[0]);
fprintf(stderr, "\t%s [-i ] &amp;lt; \n", argv[0]);
fprintf(stderr, "\t%s &amp;lt; \n", argv[0]);
exit(0);
}
}

if(patch == stdin) {
fprintf(stderr, "Reading patch data from stdin.\n");
}
fgets(line, sizeof(line), patch); /* eat dif file intro line */
fgets(line, sizeof(line), patch); /* eat blank line */

if(input == NULL) {
fprintf(stderr, "Inferring input file name from patch file data.\n");
fscanf(patch, "%256s", line);
input = fopen(line, "rb+");
if(input == NULL) {
fprintf(stderr, "Failed to open input file %s\n", line);
exit(0);
}
}
else{ /* don't need input file name, but need to skip it in dif file */
fgets(line, sizeof(line), patch);
}

while(fscanf(patch, "%x: %x %x", &amp;amp;offset, &amp;amp;orig, &amp;amp;newval) == 3) {
fseek(input, offset, SEEK_SET);
if(fgetc(input) == orig) {
fseek(input, offset, SEEK_SET);
fputc(newval, input);
}
else{
//original bytes don't match expected?
```

```
}
}
fclose(input);
if (patch != stdin) {
fclose(patch);
}
}
```

After compiling this code successfully, execute the following command on DOS which requires the DIF file and new patched file name as:

```
ida_patcher.exe -i PatchedApp.exe -p binaryCrackPatched.dif
```

This approach also fulfills the same objective and produces a patched binary as defined earlier in the papers.

Final Note

So, we have learnt one of the amazing tactics of patching the binary and producing new executable files using IDA Pro, for which crackers used to struggle because as of the IDA limitation, it can only disassemble the binary code to analyze the vulnerabilities or bugs in the code. We can temporary divert the instruction code by modifying the ZF register during debugging just by placing the breakpoint, but can't patch or modify the bytes sequence into the memory directly to produce new binary executable, because IDA Pro makes change in the database, not in the binary executable. This paper taught us how to make visible the hidden features of the Patch program in IDA Pro IDE. We have also come across with string patching and new way to modify bytes by producing DIF file, which later passed as an argument to a custom C patch code.

http://resources.infosecinstitute.com/applied-cracking-byte-patching-ida-pro/

Linux

动态分析

Android

IDA 配套真机 ROM 修改教程

codegod

小菜最近开始学安卓破解，无奈市面上模拟器都不能很好的支持 ida 调试，下面简单介绍下主流的几款模拟器：

1、android 自带 emulator，基于 ARM 架构，缺点启动慢，IDA 附加经常下不了断点，权限不够等

2、droid4x（中文名海马玩），基于 ARM 架构，缺点同上

3、Genymotion，基于 X86 架构，启动快，缺点对于不支持 x86 平台的 apk 无法运行和调试

相信如果搭建过模拟器和 IDA 调试环境的肯定被一些问题折腾的够呛，当然 IDA 附加不上进程，可以用 GDB 命令行调试，个人测试过是没有问题的。

废话不多说，还是真机给力，小菜最近拿到一个三星手机，折腾完可以任意进程 IDA 附加，以下是步骤，大神绕道，不喜勿喷

以下操作在 ubuntu 12.04 64 位下

1、拆包

perl split_bootimg.pl boot.img

2、查看 img 信息

unpackbootimg -i boot.img

3、解压

$ mkdir ramdisk
$ cd ramdisk
$ gzip -dc ../boo.img-ramdisk.gz | cpio -i

4、修改 default.prop
ro.secure=0
ro.allow.mock.location=0
ro.debuggable=1
ro.adb.secure=0

5、将 ramdisk 打包，mkbootfs 为 32 位程序，需要安装 32 位库
sudo apt-get install lib32s
tools/mkbootfs ./ramdisk | gzip > ramdisk-new.gz

6、重新生成 boot.img，参数参考步骤 2 中输出信息

tools/mkbootimg --cmdline 'console=ttyDCC0 androidboot.hardware=xxx' --kernel
boot.img-kernel --ramdisk ramdisk-new.gz --base 0x00200000 --pagesize 4096 -o boot-new.img
7、将 boot.img 重新打包加 md5 校验

tar -cf boot.tar boot.img
md5sum -t boot.tar >> boot.tar

8、手机关机重新进入挖煤模式后使用 odin 将 boot.tar 刷入手机
重启手机后，使用 DDMS 就可以看到所有的进程都可以使用 Logcat 查看，IDA 附加，后面就可以 XXOO 了
http://www.pd521.com/thread-630-1-1.html

修改 Nexus5 的 boot.img- 打开系统调试

介绍

当打开 Android 系统调试标志时，手机内的 APP 都可以被调试，调试的是 dex 内的 Java 代码。

打开系统调试标志的好处是：当调试 APP 时，不需要在 APP 的 AndroidManifest.xml 文件中添加 android:debuggable="true"，从而省了对 APP 重打包的过程。

修改 boot.img 本以为是很简单的事情：解包-修改-打包-刷入-完事儿。可是实践中把新的 boot.img 刷入后，手机一直启动不起来。甚至解包后 直接打包，然后刷入手机，手机依旧起不来。经过摸索，现在成功的将修改后的 boot.img 刷入手机并且手机运行良好，所以总结了这篇文章。

不直接修改手机里的 default.prop，是因为当手机重启后这个文件就会复原。

所有需要的工具见附件，其中 mkbootimg 和 unpackbootimg 是基于源码编译的，网上下载下来的不靠谱。

所有操作基于 linux x64 系统。

步骤：

一、下载“Google Nexus 5 谷歌最新官方原厂安卓 4.4.4 固件”，从这个 ROM 中提取出 boot.img 文件。

二、boot.img 解包

运行下面的命令将会对 boot.img 解包，得到 boot.img-kernel 和 boot.img-ramdisk.gz 两个文件：

```
split-bootimg.pl boot.img
```

三、处理 boot.img-ramdisk.gz

运行下面的命令，对 boot.img-ramdisk.gz 进行解压：

```
mkdir ramdisk
cd ramdisk
gzip -dc ../boot.img-ramdisk.gz | cpio -i
```

四、修改 default.prop，打开系统调试标志

找到解压出来的 default.prop 文件，将其中的 ro.debuggable=0 修改为 ro.debuggable=1

五、ramdisk 目录打包

返回 ramdisk 的上层目录，输入命令：

mkbootfs ./ramdisk | gzip > ramdisk.img

六、打包出新的 boot.img

命令：

mkbootimg --base 0x00000000 --ramdisk_offset 0x02900000 --second_offset 0x00F00000 --tags_offset 0x02700000 --cmdline 'console=ttyHSL0 androidboot.hardware=hammerhead user_debug=31 maxcpus=2 msm_watchdog_v2.enable=1 earlyprintk' --kernel boot.img-kernel --ramdisk ramdisk.img -o newboot.img

七、将新的 boot.img 刷入手机

这时，在 windows 下还是 linux 下就无所谓了。将手机连上电脑输入下面的命令，重启手机进入 fastboot：

adb reboot bootloader

当重启手机后，将新的 boot.img 刷入手机：

fastboot flash boot newboot.img

如果出现类似下图的输出，就说明刷入成功了：

```
sending 'boot' (8702 KB)...
OKAY [  0.479s]
writing 'boot'...
OKAY [  0.742s]
finished. total time: 1.224s
```

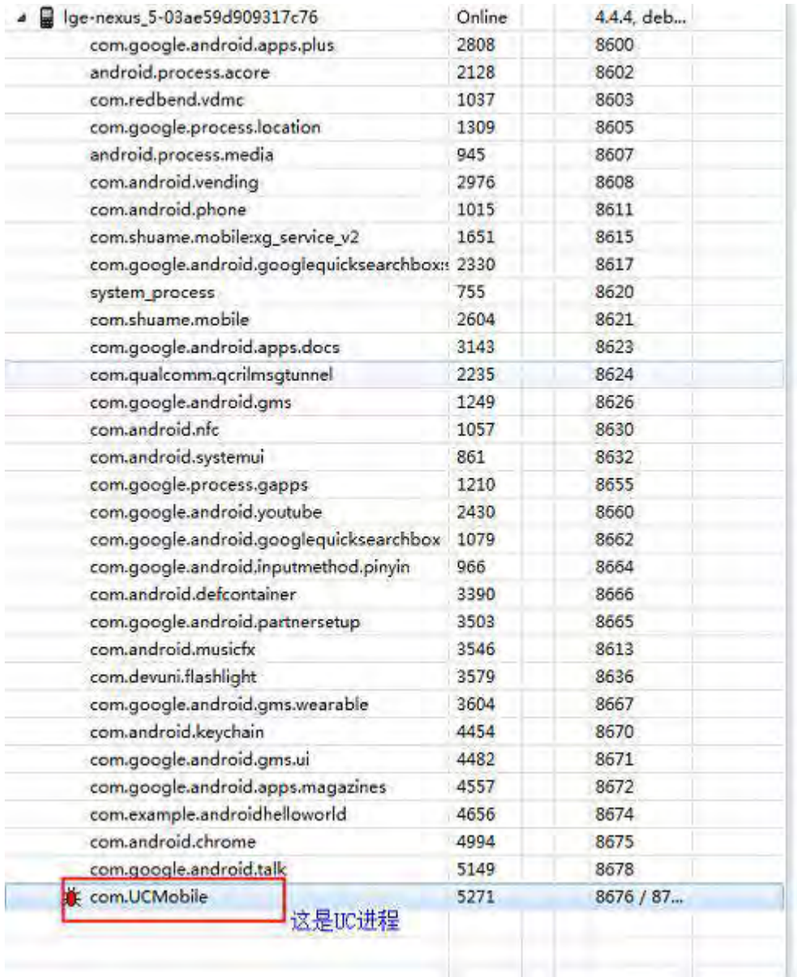
刷入成功并不代表你的手机可以正常使用~~不过只要严格按照上面的方法，在 Nexus5 这个手机下还是没有问题的。现在输入下面的命令重启手机吧：

fastboot reboot

八、系统中的 APP 都可以调试了！

APP 调试命令：adb shell am start -D -n <包名>/<Activity 名>

我的 eclipse 的 DDMS 视图：



后记：

如果手机刷成砖怎么办？

我的 Nexus5 本身就是“Google Nexus 5 谷歌最新官方原厂安卓 4.4.4”，所以我在手机因为刷 boot.img 而无法启动的时候，直接进入 fastboot，然后把官方的 boot.img 重新刷到手机里面，然后砖就满血满状态复活了。

手动进入 fastboot 模式（用于变成砖以后使用）：

同时按【音量下键+电源键】。

工具备注：

mkbootimg 和 unpackbootimg 可能只能用在 linux x86 的系统上，那我的 linux x64 怎么能使用这两个工具哪？因为我的 linux x64 系统下载了支持 x86 程序运行的库。

http://bbs.pediy.com/showthread.php?p=1350174

Android 双机（网络和 USB）调试及其完美 ROOT

本文主要讲如下两个内容。

1. 如何让一部 Android 手机带两台 PC 进行调试。
2. 如何完美 Root Android 手机。

我曾经遇到过这样的问题，Mac OS X 上测试 Android 的系统程序，大家都知道，Android 源代码的编译通常需要 Ubuntu Linux，所以在 Mac OS X 上安装了 Vmware ubuntu。这样本没什么问题。将 Android 手机与 MBP 相连，系统会提示连接 Mac OS X，还是 Ubuntu。不过这有一个问题，就是 ubuntu 和 mac os x 同时只能连接一个。由于某些需要，要用到 Mac OS X 上的 Eclipse 编写 Android 应用，也能安装在同一部 Android 手机上，这就很郁闷，单单通过 USB 线只能连接一台 PC（包括虚拟机）。

从技术上说，adb 完全有这个能力使一部 Android 手机连接到两台 PC 上，不过可能大多数读者用的 ROM 是官方，为了安全起见，将该功能关了。如果读 者使用了最新的 CM ROM（cm10.1，可能老的版本也有该功能，这个还没测试），就会发现在设置的“开发者选项”中多了一个“网络 ADB 调试”。如图 1 所示。这可是梦寐 以求的功能。

图 1

选择“网络 ADB 调试”选项。然后记住下面的 ip 和端口号。在一台机器上用 USB 线连接手机，Eclipse 的 devices 列表就会显示该手机已连接成功。如果仍然用当前 PC 通过网络连接 ADB，需要执行下面的命令（PC 与手机在同一网段）。

```
adb kill-server

adb connect 192.168.17.103:5555
```

如果一切正常，就已经连上了，打开 Eclipse，会在 devices 列表看到如图 2 的设备。

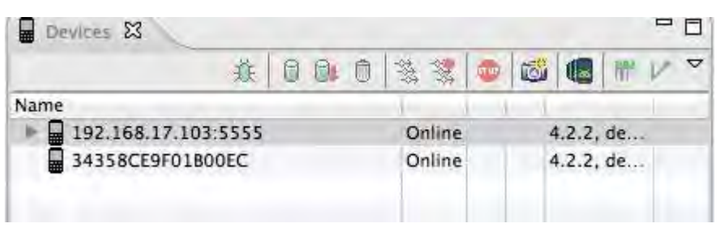


图 2

其实这两个设备是一个，只是上边那个通过网络连接的手机，下边那个通过 USB 线连接的手机。现在运行程序，仍然会看到要求选择运行设备，如图 3 所示。

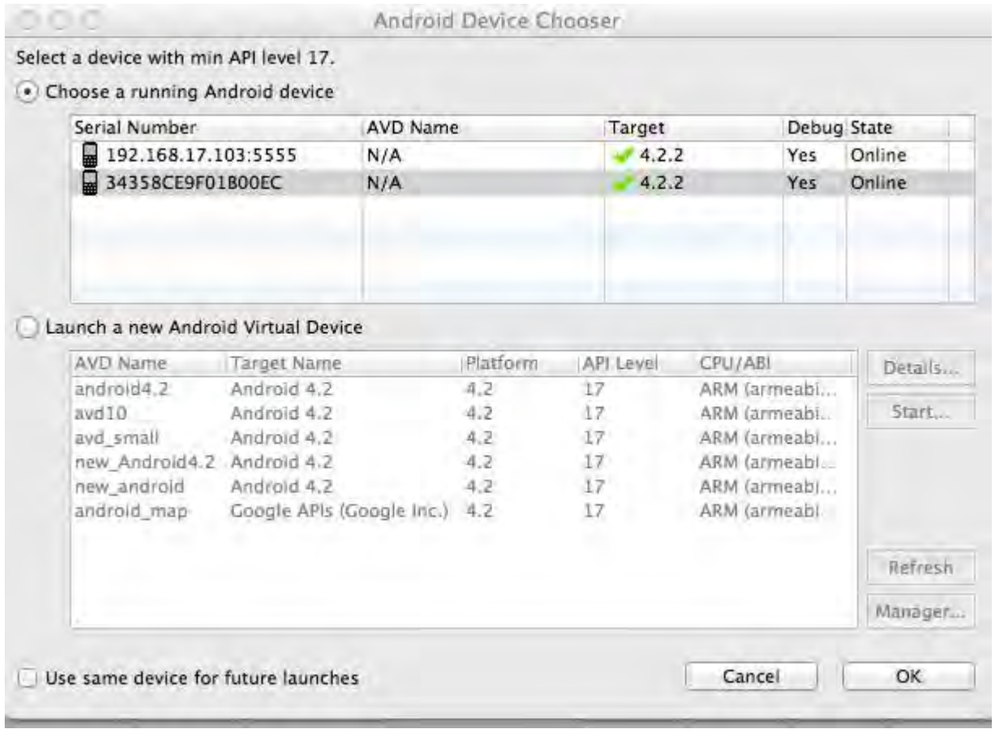


图 3

其实现在选哪个都行，因为这两个设备只是通过不同方式与 PC 连接。

如果在不同的 PC 上（或 PC 和虚拟机），一个通过 USB 线，一个通过网络。两台机器中的 eclipse 就可以在同一部手机上调试程序了，是不是很爽呢！

如果要断开网络 adb 连接，执行 adb disconnect 命令。

要注意，这一操作比较危险，一旦选择“网络 adb 调试”，只要同一网段的其他用户获知 IP 和端口号，就可以任意操作你的手机，而且不会有任何提示。甚至是在地球的另一端（只要你们在同一个网段，包括 VPN）。

现在进入第二个主题，如果完美获取 Android 手机的 ROM。由于前面使用的是 CMROM（不知道 CM ROM 是什么的上网自己查），所以这里仍然讨论 CM ROM 的 root 过程。

可能刚一接触 Android 手机的读者会感觉 root 一部手机很复杂，其实再简单不过了。Root 的基本原理就是在 Android 系统的/system /xbin 目录放一个 su 命令。用过 su 的 linux 用户都知道 su 是什么东西，一个提取 root 权限的命令。如果读者用的是 CM ROM，在/system/xbin 目录已经有了 su 命令。而且在设置里可以打开 root 权限（cm 的低版本直接就打开了 root 权限），所以使用 CM ROM，这一步可以省了。当然，如果使用的不是 CM ROM，也好吧。下一个 CM ROM 压缩包，将里面的杂七杂八的东西都删掉（包括内核镜像 boot.img），只保留 system/xbin 目录中的 su 文件

（该文件也可以上网找一个 现成的，或直接编译 android 源代码中的 su 也可以，默认是不带这个命令的）和 META-INF 目录，然后重新打包成 update.zip。然后最好 刷一个 clockworkmod recovery，这个比较好，update.zip 不需要签名验证就可以刷。之后就可以在 recovery 模式下刷 update.zip 了。这个 update.zip 并不是完整的 ROM，只是一个升级包，目的是将 su 文件放到 /system/xbin 目录中（该目录没有 root 权限是只读的）。刷 完后，进入手机的 shell（adb shell），然后执行 su，发现原来的$变成了#，表明已经是 root 了，当然，有些目录还是不可写，例如，为了使 system 及其子目录可写，需要再 次执行下面的命令。

```
mount  -o rw,remount /system
```

现在还有一个问题，就是通过 adb shell 进入 Shell，默认仍然是$，这就有些麻烦，因为对于程序员来说，需要调试系统程序，要用 adbremount 命令使/system/app 目录变成可写的，然后可用 adb push 命令直接将 apk 文件上传到该目录。但如果默认不是 root，是不能执行该命令的（权限不允许）。所以我们还需要做另外一件事，就是修改 Android 系统根目录的 default.prop 的内容，通常可以改成如下内容。

```
ro.adb.secure=0
ro.secure=0
ro.allow.mock.location=1
ro.debuggable=1
persist.sys.usb.config=mass_storage,adb
persist.service.adb.enable=1
```

最大的问题就是 default.prop 是内存文件，改了也没用，一重启就会恢复原样，而且只有重启才能生效（好像进入了二难推理）。所以修改 default.prop 文件的方法是直接修改 boot.img 文件。该文件由两部分组成：zImage 和 ramdisk.img。其中 zImage 就是 linux 内核的二进制文件。ramdisk.img 是内存磁盘镜像。该镜像中就包含了被称为 Android 第一个运行的程序 init。 default.prop 文件也包含在该镜像中。下面就需要 android 源代码和 linux 内核源代码了。强烈建议使用 CM 提供的源代码，因为 CM 团队 已经为我们进行了完美的适配，所以只需要修改与业务相关的内容即可。现在让我们向 CM 团队致敬。

Android 源代码编译后，在 out 目录的相关子目录录生成了一个 root 目录，该目录中的内容就是 ramdisk.img 文件解压后的内容。现在找到 default.prop 文件，并安着上面的内容修改该文件。然后使用下面的命令重新生成 ramdisk.img 文件。

```
mkbootfs root | minigzip > /ramdisk.img
```

接下来使用下面的命令重新生成 boot.img 文件。

```
mkbootimg --kernel kernel   --ramdisk ramdisk.img -o boot.img
```

其中 kernel 是内核二进制文件，与 zImage 完全一样。只是编译 android 源代码时将其命名为

ok， 现在可以进入 bootloader 模式，然后重新执行 fastboot flash bootboot.img 刷内核镜像，然后再重启手机，现在进入 shell，ok，默认就是#了。退出 shell，执行 adb remount。就可以用 adb push 上传文件到/system 的其他目录了，如/system/app。

http://www.cnblogs.com/nokiaguy/archive/2013/05/05/3061431.html

刷机包获取 ROOT 权限方法

大家应该都知道安卓手机的 ROOT 权限，官方的 ROM 是不带 root 权限的，为的是安全问题！而现在很多机友都需要 root 权限，因为 root 权限可以做很多东西，比如卸载系统程序等！

一、root 权限是指手机上有一个名为授权管理的程序 superuser.apk，可以授予程序 root 用户的权限。

root 权限的授权程序文件路径：

- 1、/system/app/superuser.apk
- 2、/system/bin/su 或者 /system/xbin/su

所以，有兴趣的朋友，自己可以做一个 recovery 模式的 root 刷机补丁。刷机脚本如下：

- mount("ext4", "EMMC", "/dev/block/mmcblk0p25", "/system"); （挂载 system 分区）
- package_extract_dir("system", "/system"); （拷贝文件到 system 分区）
- symlink("/system/bin/su", "/system/xbin/su"); （把 bin 和 xbin 的路径连在一起，su 放 bin 或者 xbin 里面都可以）
- set_perm(0, 0, 06755, "/system/bin/su"); （设置 su 的执行权限）
- unmount("/system"); （卸载 system 分区）

二、内核 root，估计了解内核 root 的机友比较少了，这里由我来给大家说一下

说到内核，大家都会想到 boot.img，没错，内核 root，就是将 boot.img 进行 root，具体操作如下：

将 boot.img 用厨房软件解包，得出 boot.img-ramdisk 文件夹和内核 zImage，打开 boot.img-ramdisk 文件夹。里面有个 default.prop，用文本编辑打开 default.prop 如下：

- ro.secure=1
- ro.allow.mock.location=1
- ro.debuggable=1
- persist.service.adb.enable=1

如果 ro.secure=0 就是允许我们运行 adb root 命令，通常内核 root 就是指 ro.secure=0，将 ro.secure=1 修改成 ro.secure=0，之后打包回来 boot.img，这样就成功 root 了内核了！

三、内核 root 延伸（降级 hboot）

内核 root 有什么用？一般装了 superuser.apk 授权程序，用户也能卸载系统程序了，那么 root 内核其实也没什么用！确实是这样！

这里我就说一个内核 root 比较有用的东西，不知道大家有没有试过降级 hboot，有些人很顺利就可以把 hboot1.28 降级到 hboot1.25，有 些人死活都不能降级 hboot，原因就是在这个内核了，hboot 降级不了的机友很大可能就是所装的系统内核是没有 root 的，内核没有 root，那么 adb 命令就没有 root 权限了，因为降级 hboot 都是使用 adb 命令操作的，而且需要 adb root 权限。所以 hboot 降级失败的机友，自己 root 一下内核 boot.img，这样降级 hboot 应该就没有问题了。

http://www.muzisoft.com/news/86783.html

使用 IDA 调试 APK 中动态加载的.so 库

最近在逆向 apk 的时候发现关键的加密算法被放进了 so 库里，毕竟直接用 java 写这种代码太容易被破了，放在 so 里面更安全些。既然这样，那也只好动态调 so 了，查资料研究了几个小时终于搞定了，现记录如下：

1.基本配置

IDA，adb，android_server，模拟器。这里用的是 IDA6.5。

android_server 在 IDA 的目录下面，需要用 adb push 至模拟器：

```
adb push android_server /data/
```

还要增加权限：

```
adb shell chmod 777 /data/android_server
```

然后到 data 目录下执行 android_server：

```
./android_server
```

默认开启 23946 端口：

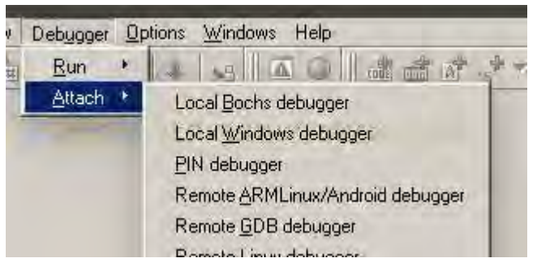
```
root@android:/data # ./android_server
IDA Android 32-bit remote debug server(ST) v1.17. Hex-Rays (c) 2004-2013
Listening on port #23946...
```

打开另一个 shell 设置端口转发：

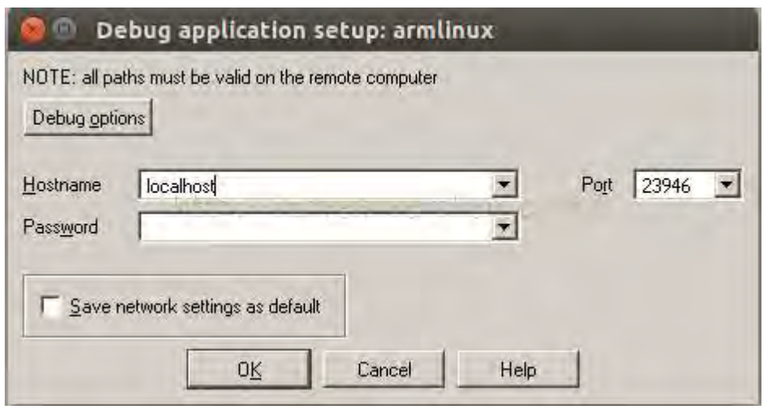
adb forward tcp:23946 tcp:23946

2.IDA 动态调试

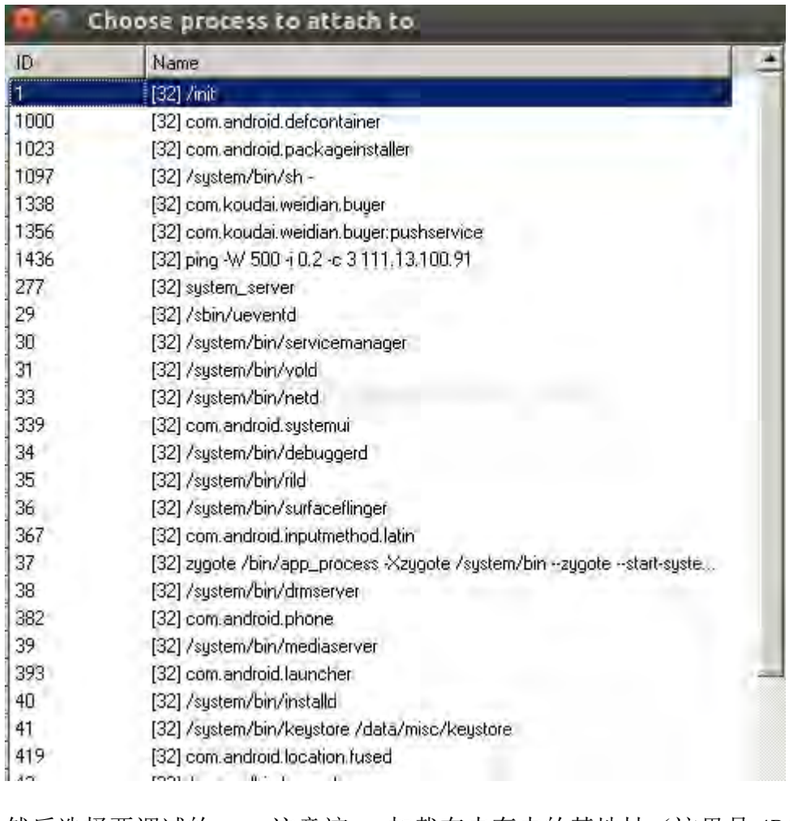
在模拟器中运行要被调试的 apk，确定要被调试的 so 加载进内存后打开 Remote ARMLinux/Android debugger：



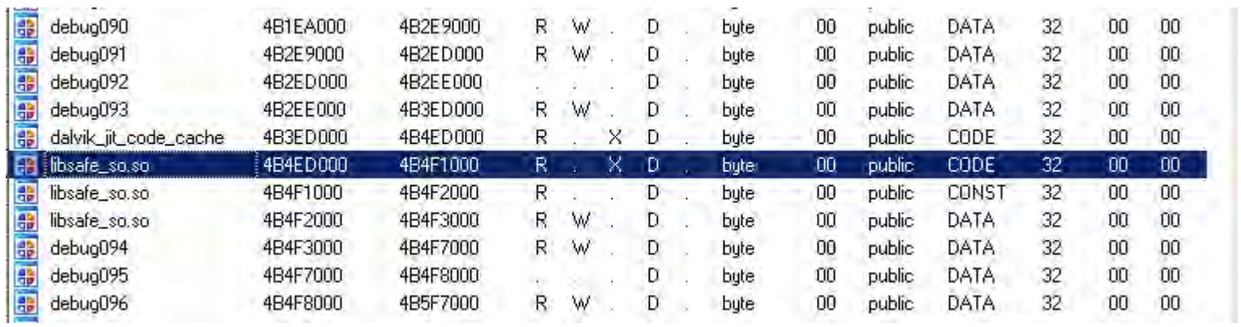
hostname 中填写 localhost，端口号添 android_server 打开的端口 23946，无需密码：



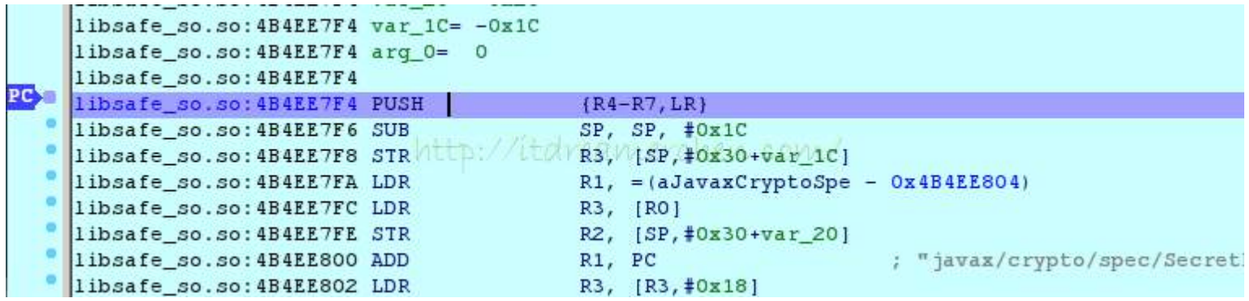
选择 attach 至调试的 apk 进程：



然后选择要调试的 so，注意该 so 加载在内存中的基地址（这里是 4B4ED000）：



这时会跳转至 so 模块的基地址，开启另一个 IDA 静态加载该 so 模块查看要跟踪的函数的偏移，基地址+偏移 就是这个函数在这个进程中的实际地址。点快捷键 g，选择要跳转的地址，然后 F2 下断点，运行程序，继续运行模拟器中的 apk，停在断点处，接下来就是一般 的逆向分析了。



http://itdreamerchen.com/%E4%BD%BF%E7%94%A8ida%E8%B0%83%E8%AF%95apk%E4%B8%AD%E5%8A%A8%E6%80%81%E5%8A%A0%E8%BD%BD%E7%9A%84-so%E5%BA%93/

IDA 调试 APK 中加载的 so 库之方法二

在 IDA 动态调试 so 库这 篇文章中讲了基本的动态调试方法，但是这种方法有一个缺点，就是只能在进程运行起来之后才能 attach 上，也就是说这时进程已经将 so 库加载进内存并且 已经执行完 JNI_OnLoad，所以只需要在 JNI_OnLoad 函数中完成一些反调试措施，就无法使用上篇文章所介绍的方法来动态调试。这篇文章就介 绍动态调试的另一种方法：直接在 JNI_OnLoad 函数上下断点。

另外，上篇文章是在模拟器下进行调试，这篇文章直接在真机上完成动态调试。

1.工具准备

IDA6.6，adb，jdb，固件工具箱，android_server，ddms。

首先打开 ddms。

2.运行 android_server

在 IDA 的 dbgsrv 目录下找到 android_server，并且拷贝至手机的/data/目录下，然后修改 android_server 的权限为 777，然后以 su 的身份运行 android_server。

3.设置端口转发

打开另一个窗口设置端口转发：

```
adb forward tcp:23946 tcp:23946
```

4.adb am 启动主 activity

首先打开固件工具箱->程序管理器->选中要调试的 apk->高级->高级冻结->这里可以查看 apk 中所有类的完整路径（也可以直接从 AndroidManifest.xml 中找），找到主 activity 完整路径。在终端中运行：

adb shell am start -D -n ｛包名(package)｝/｛包名｝.{activity 名称}

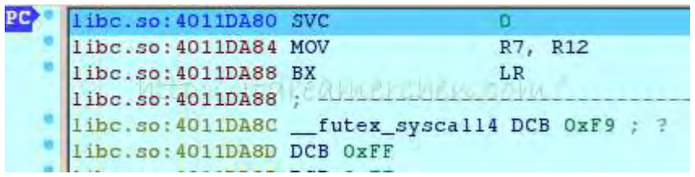
比如：

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

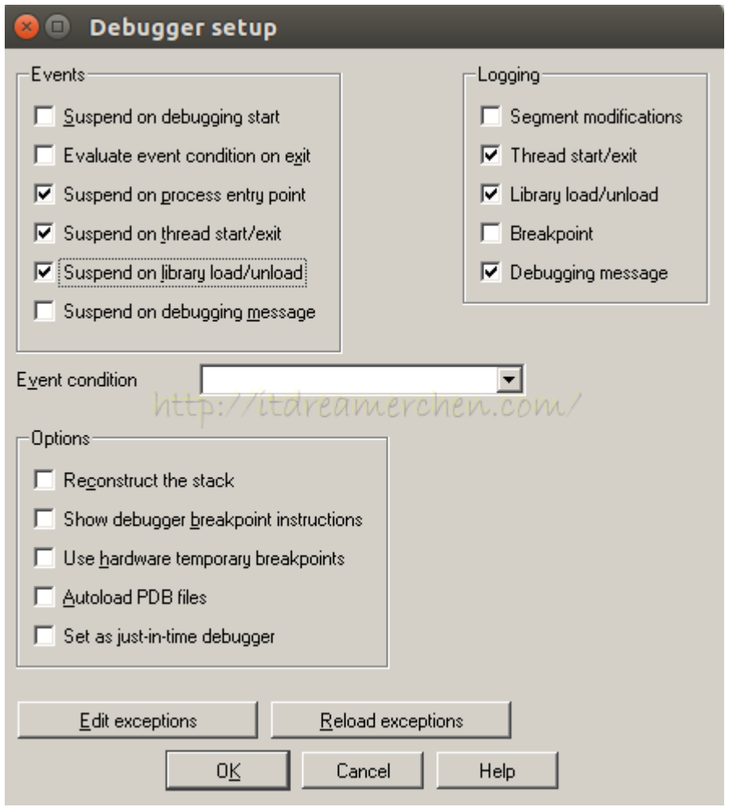
这时手机会进入 waiting for debugger 状态。

5.IDA 附加

点击菜单 Debugger->Attach->Remote ArmLinux/Android debugger ，打开调试程序对话框，在 hostname 一栏输入 localhost。选中要被调试的 apk，这时会停在 libc.so：



设置调试选项：点击菜单 Debugger->Debugger Opitions 在弹出的 Debugger setup 窗口的 Events 中选择 suspend on thread start/exit 以及 suspend on library load/unload，然后 OK 退出。通过此操作可以使得程序在创建新线程和加载 so 时自动中断。



6.jdb 连接调试器

在 ddms 中查看相应的进程端口号（一般情况下为 8700），然后执行如下命令：

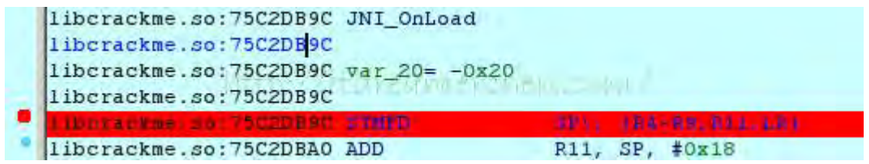
```
jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700
```

然后在 IDA 中点击 F9 继续运行，然后手机上的 waiting for debugger 提示消失。

7.在 JNI_OnLoad 上下断点

IDA 中 Ctrl+s 查找要调试的 so，一般情况下会有多个同名 so，选择具有 rx 权限的 so（代码段），其他的可能为数据段之类的，比如：

用另一个 IDA 打开这个 so 库，找到 JNI_OnLoad 起始地址，加上 so 库的偏移地址即为 JNI_OnLoad 函数在运行时的真实地址。快捷键 g，转至该地址，即可在 JNI_OnLoad 函数内下断点：



接下来 F9 继续运行，就可以动态调试了。

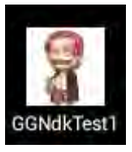
http://itdreamerchen.com/ida%E8%B0%83%E8%AF%95apk%E4%B8%AD%E5%8A%A0%E8%BD%BD%E7%9A%84so%E5%BA%93%E4%B9%8B%E6%96%B9%E6%B3%95%E4%BA%8C/

Ida Pro 6.6 的 Android 详细调试步骤

调试 so 文件

作者：PeterDocter
时间：2015 年 5 月 5 日 3:36:51 PM

调试对像



http://blog.csdn.net/guiguzi1110/article/details/42027109

调试环境

Google 原生模拟器

调试前准备

1.查看调试设备

adb devices

List of devices attached

emulator-5554　　device

2.创建存放 Ida Pro 的 android_server

adb -s emulator-5554 shell
cd /data/
mkdir tmp
exit

注意：“如果要想在 system 这样目录直接创建会提示只读，请用以下命令解决！”
adb -s emulator-5554 remount

3.传上 Ida Pro 的 android_server

Ida Pro 的目录(文件夹)
IDAPro6.6\dbgsrv\ android_server
cd /d D:\\IDAPro6.6\dbgsrv
adb -s emulator-5554 push android_server /data/tmp

4.执行 android_server

adb -s emulator-5554 shell
cd /data/tmp
chmod 755 android_server

./android_server
IDA Android 32-bit remote debug server(ST) v1.17. Hex-Rays (c) 2004-2014
Listening on port #23946...

5.端口转发

adb -s emulator-5554 forward tcp:23946 tcp:23946

//以下命令可能不用
adb forward tcp:8899 jdwp:进程 ID

6.启动 Android Deubg Monitor

Android sdk 目录中 tools
旧版是 ddms.bat
新版是 monitor.bat

APK 运行前准备

关键文件

查看 AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.ggndktest1">
    <application android:allowBackup="true" android:icon="@drawable/ggg" android:label="@string/app_name" android:theme="@style/AppTheme">
        <activity android:label="@string/app_name" android:name="com.ggndktest1.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

红色字体是关键！

libgg-jni.so 文件

要调试函数偏移地址

```
.text:00000C40                                  EXPORT Java_com_ggndktest1_JniGg_VipLevel
.text:00000C40                  Java_com_ggndktest1_JniGg_VipLevel
.text:00000C40 08 B5                                    PUSH        {R3,LR}
```

APK 三种挂起

Apktool+注入代码方式

AndroidManifest.xml 文件修改(可以不修改)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.ggndktest1">
    <application android:allowBackup="true" android:debuggable="true" android:icon="@drawable/ggg" android:label="@string/app_name" android:theme="@style/AppTheme">
        <activity android:label="@string/app_name" android:name="com.ggndktest1.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

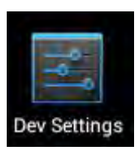
MainActivity.smali 文件修改

```
# virtual methods
.method protected onCreate(Landroid/os/Bundle;)V
    .locals 4
    .param p1, "savedInstanceState"    # Landroid/os/Bundle;

    .prologue
    .line 15
    invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V

    invoke-static {},Landroid/os/Debug;->waitForDebugger()V
```

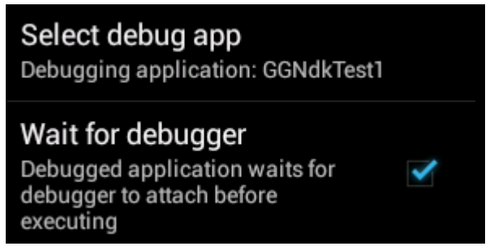
am 方法

adb –s emulator-5554 shell am start –n –D com.ggndktest1/ com.ggndktest1.MainActivity

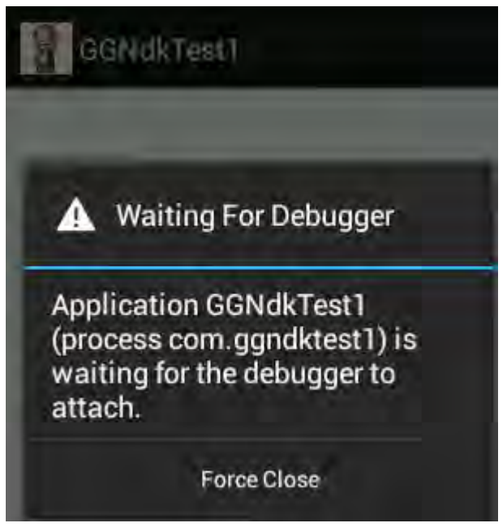
Dev Settings 方法



在 Android 移动设备上叫开发者模式



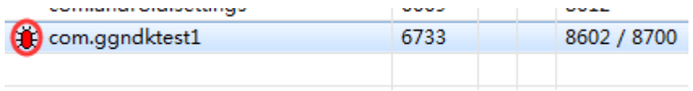
退出运行



同样可以挂起！

查看挂起状态

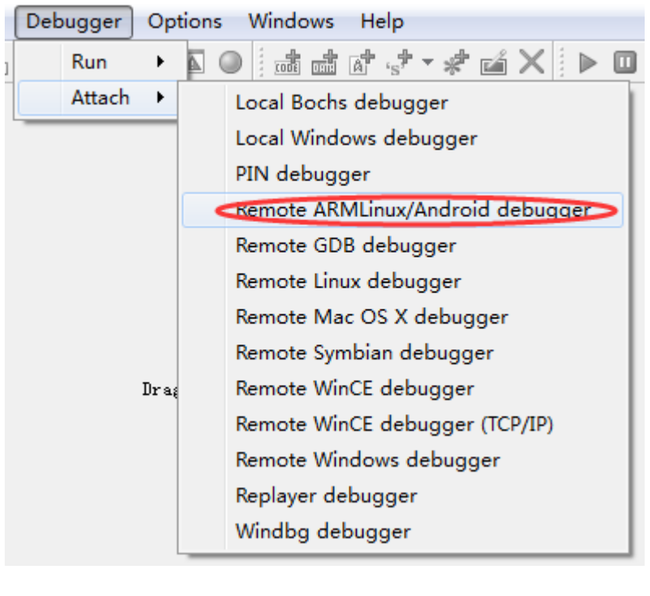
Android Deubg Monitor



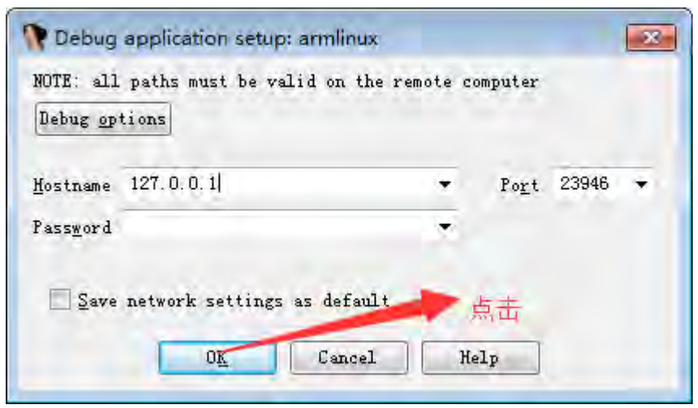
Ida Pro 两种附加

方法 A 附加

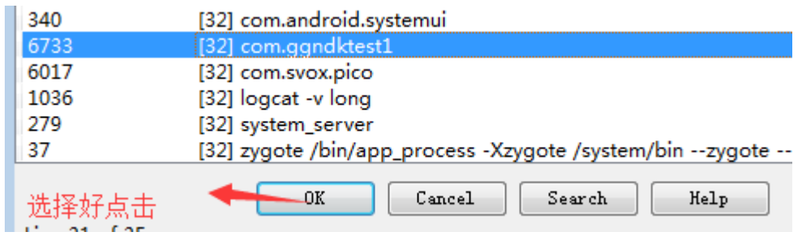
附加选择



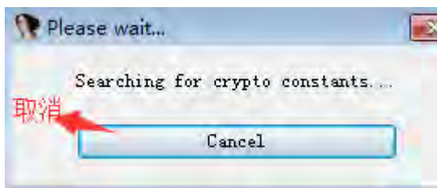
连接调试



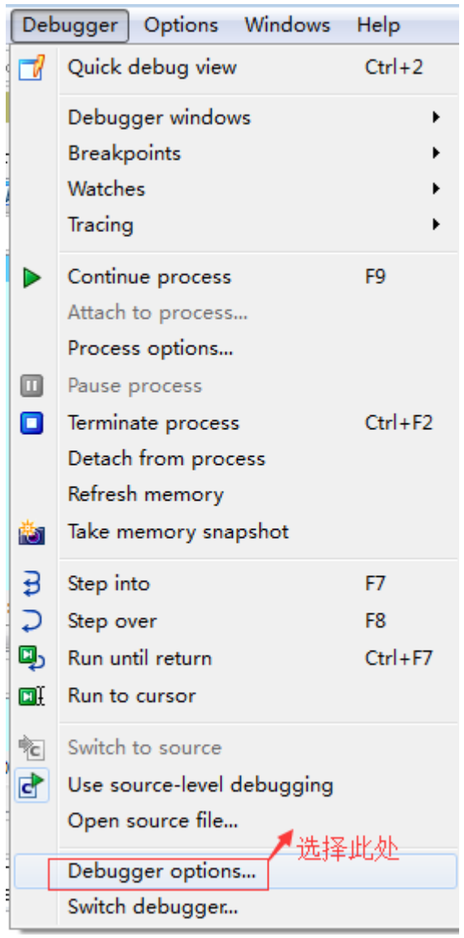
选择对像

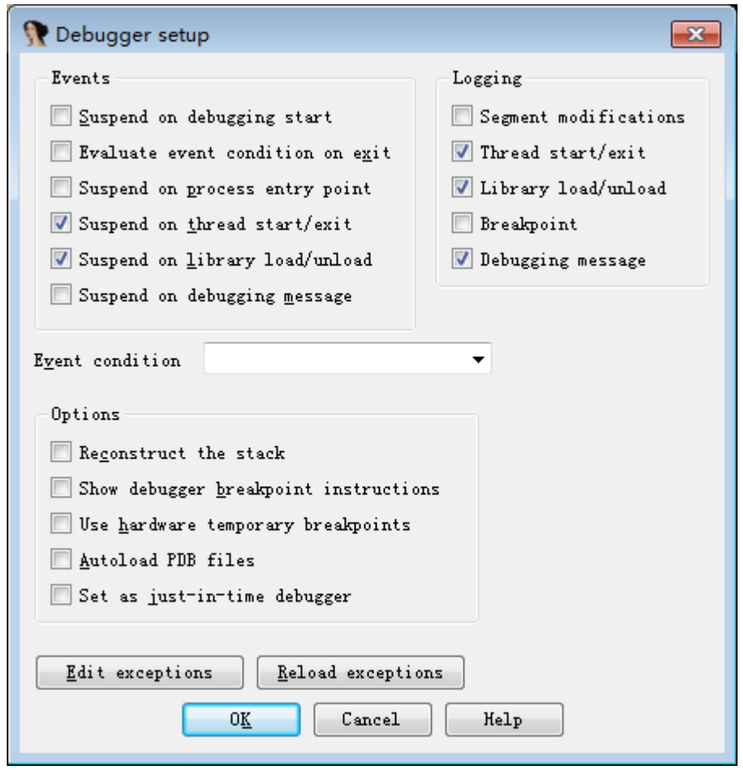

340 [32] com.android.systemui
6733 [32] com.ggndktest1
6017 [32] com.svox.pico
1036 [32] logcat -v long
279 [32] system_server
37 [32] zygote /bin/app_process -Xzygote /system/bin --zygote --
选择好点击
OK
Cancel
Search
Help

取消分析


Please wait...
Searching for crypto constants...
取消
Cancel

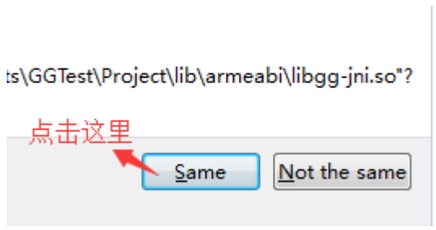
中断设置


Debugger
Options
Windows
Help
Quick debug view Ctrl+2
Debugger windows
Breakpoints
Watches
Tracing
Continue process F9
Attach to process...
Process options...
Pause process
Terminate process Ctrl+F2
Detach from process
Refresh memory
Take memory snapshot
Step into F7
Step over F8
Run until return Ctrl+F7
Run to cursor
Switch to source
Use source-level debugging
Open source file...
选择此处
Debugger options...
Switch debugger...



方法 B 附加

在静态分析 so 文件状态时调试不同之处：



```
.text:4A8A2C40
.text:4A8A2C40                          EXPORT Java_com_ggndktest1_JniGg_VipLevel
.text:4A8A2C40                  Java_com_ggndktest1_JniGg_VipLevel
.text:4A8A2C40 08 B5                    PUSH    {R3,LR}
.text:4A8A2C42 02 2A                    CMP     R2, #2
.text:4A8A2C44 00 00                    MOVS    R0, R0   方法B不同之处
.text:4A8A2C46 03 2A                    CMP     R2, #3
.text:4A8A2C48 00 00                    MOVS    R0, R0
.text:4A8A2C4A 01 2A                    CMP     R2, #1
.text:4A8A2C4C 00 00                    MOVS    R0, R0
.text:4A8A2C4E 08 49                    LDR     R1, =(aGoldVip - 0x4A8A2C54)
.text:4A8A2C50 79 44                    ADD     R1, PC          ; "Gold Vip"
.text:4A8A2C52 07 E0                    B       loc_4A8A2C64
.text:4A8A2C54                  ; ---------------------------------------------------------
.text:4A8A2C54 07 49                    LDR     R1, =(aCopperVip - 0x4A8A2C5A)
.text:4A8A2C56 79 44                    ADD     R1, PC          ; "Copper Vip"
.text:4A8A2C58 04 E0                    B       loc_4A8A2C64
.text:4A8A2C5A                  ; ---------------------------------------------------------
.text:4A8A2C5A 07 49                    LDR     R1, =(aNormalUser - 0x4A8A2C60)
.text:4A8A2C5C 79 44                    ADD     R1, PC          ; "Normal User"
.text:4A8A2C5E 01 E0                    B       loc_4A8A2C64
.text:4A8A2C60                  ; ---------------------------------------------------------
.text:4A8A2C60 06 49                    LDR     R1, =(aSilveryVip - 0x4A8A2C66)
.text:4A8A2C62 79 44                    ADD     R1, PC          ; "Silvery Vip"
.text:4A8A2C64
.text:4A8A2C64                  loc_4A8A2C64                            ; CODE XREF: Java_com
.text:4A8A2C64                                                          ; Java_com_ggndktest1
.text:4A8A2C64 02 68                    LDR     R2, [R0]
.text:4A8A2C66 A7 23 9B 00              MOVS    R3, #0x29C
.text:4A8A2C6A D3 58                    LDR     R3, [R2,R3]
.text:4A8A2C6C 98 47                    BLX     R3
.text:4A8A2C6E 08 BD                    POP     {R3,PC}
.text:4A8A2C6E                  ; End of function Java_com_ggndktest1_JniGg_VipLevel
.text:4A8A2C6E
```

其它步骤与方法一样 A！

APK 运行恢复

Jdb 方法

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

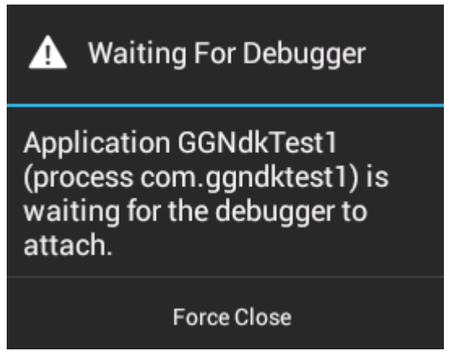
Eclipse 或其它 java 的 ide 工具都行

com.ggndktest1	6733	8602 / 8700

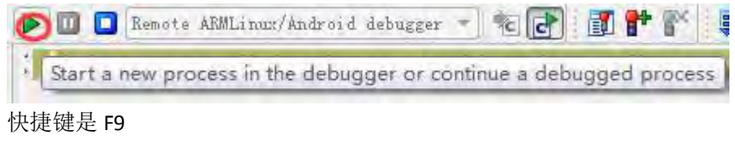
到这里 apk 运行状态已经恢复了！

SO 文件调试开始

1.让 apk 中的 so 加载



仍然出现这个提示？
是由于之前恢复的是 JDWP 协议调试的中断(Java 或 Smali)，现在的中断是 ida pro 接管的。

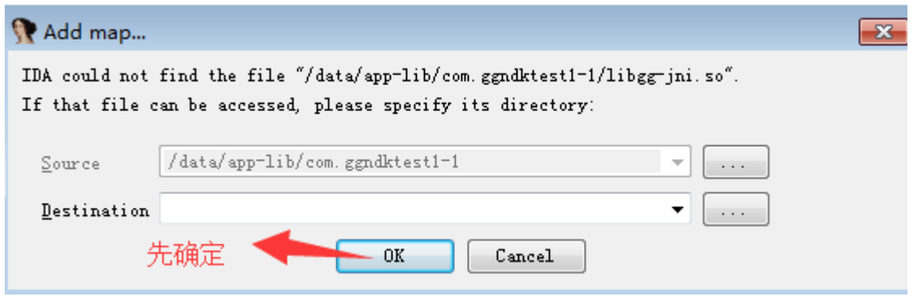


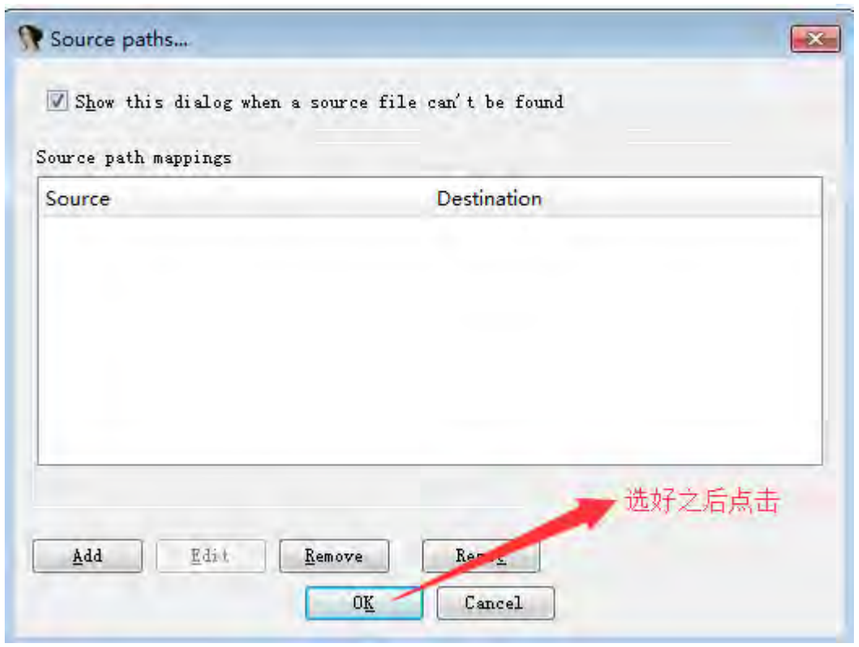
快捷键是 F9

这时提示窗口没有了，之前设置的中断起作用

```
E:\android_reverse\作业 1>jdb -connect com.sun.jdi.SocketAttach:port=8700,hostnam
e=localhost
设置 未捕捉到 java.lang.Throwable
设置延迟的 未捕捉到 java.lang.Throwable
正在初始化 jdb...
>

The initial autoanalysis has been finished.
40066408: thread has started (tid=7312)
4A9A6000: loaded /data/app-lib/com.ggndktest1-1/libgg-jni.so
```

2.可能出现的问题





3.两种定位断点

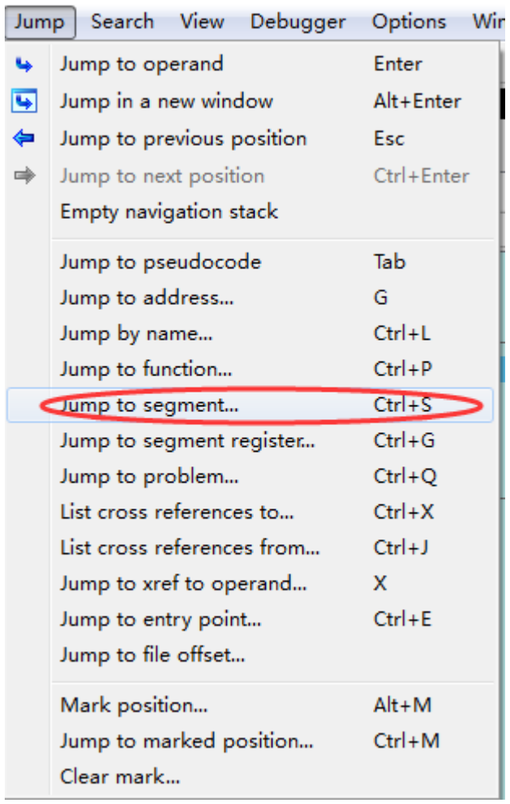
A.静态移偏地址+内存初化地址

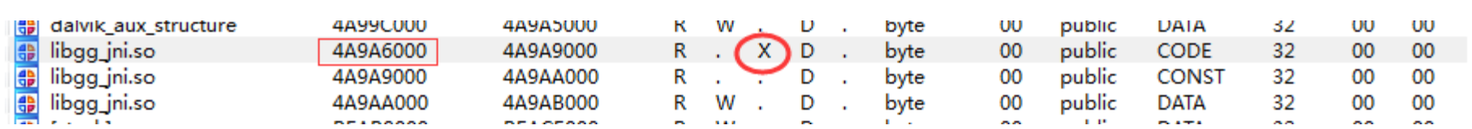
断点地址计算

静态移偏地址

.text:00000C40 Java_com_ggndktest1_JniGg_VipLevel

内存初化地址

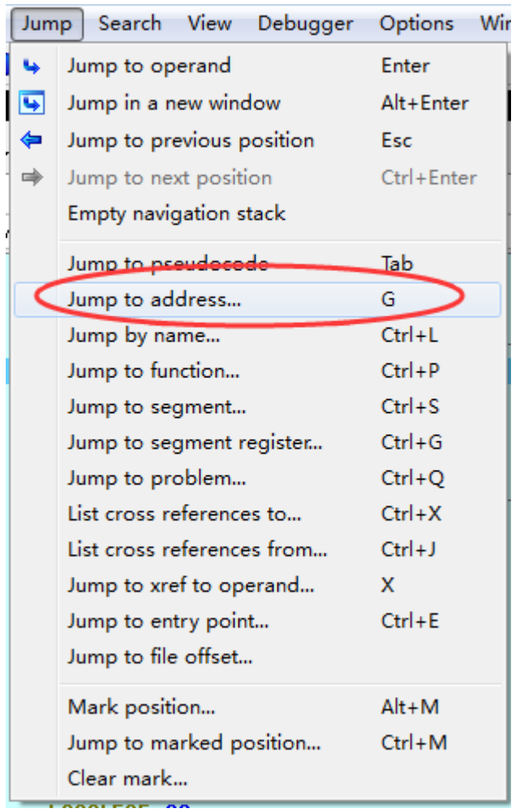


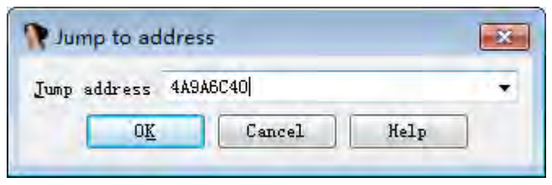


跳到断点地址

断点地址=静态移偏地址+内存初化地址

00000C40+4A9A6000=4A9A6C40

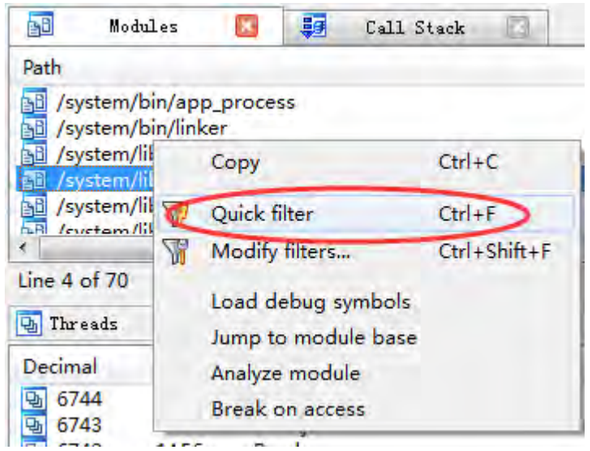


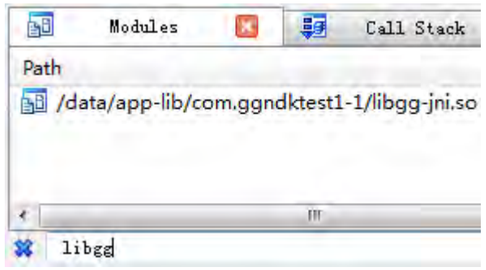



```
libgg_jni.so:4A9A6C40                   Java_com_ggndktest1_JniGg_VipLevel
libgg_jni.so:4A9A6C40 08 B5                     PUSH            {R3,LR}
libgg_jni.so:4A9A6C42 02 2A                     CMP             R2, #2
libgg_jni.so:4A9A6C44 0C D0                     BEQ             loc_4A9A6C60
libgg_jni.so:4A9A6C46 03 2A                     CMP             R2, #3
libgg_jni.so:4A9A6C48 04 D0                     BEQ             loc_4A9A6C54
libgg_jni.so:4A9A6C4A 01 2A                     CMP             R2, #1
libgg_jni.so:4A9A6C4C 05 D1                     BNE             loc_4A9A6C5A
libgg_jni.so:4A9A6C4E 08 49                     LDR             R1, =(unk_4A9A81CA - 0x4A9A6C54)
libgg_jni.so:4A9A6C50 79 44                     ADD             R1, PC ; unk_4A9A81CA
libgg_jni.so:4A9A6C52 07 E0                     B               loc_4A9A6C64
libgg_jni.so:4A9A6C54                   ; ---------------------------------------------------------------------------
```

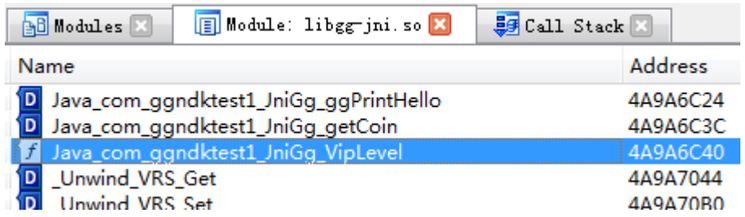
B.Ida Pro 6.6 的模块窗口定位

快速过滤器

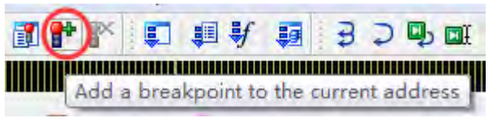




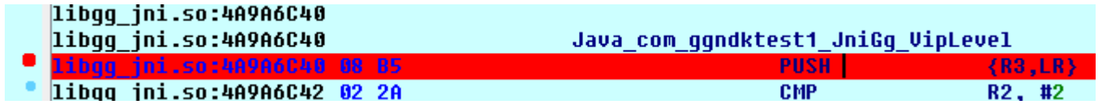
双击断点定位



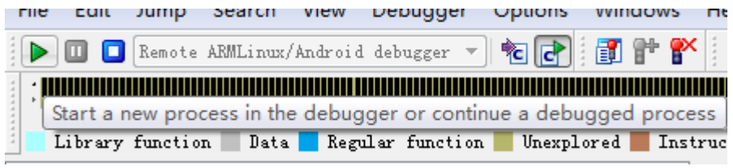
4.进行下断点



快捷键 F2

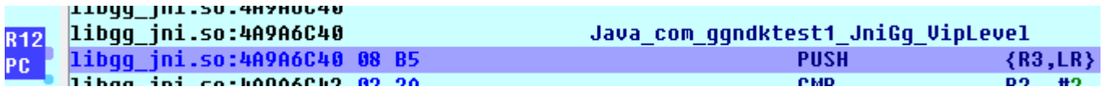


5.快乐调试

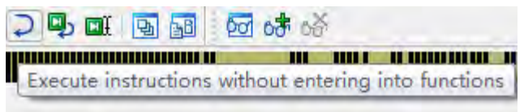


快捷键 F9

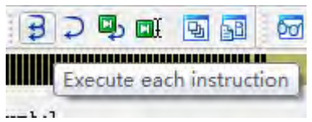
中断到这里来了！



之后就是以下步骤：



快捷键 F8



快捷键 F7

安卓 APP 动态调试技术

0x00 前言

随着智能手机的普及，移动 APP 已经贯穿到人们生活的各个领域。越来越多的人甚至已经对这些 APP 应用产生了依赖，包括手机 QQ、游戏、导航地图、微博、微信、手机支付等等，尤其 2015 年春节期间各大厂商推出的抢红包活动，一时让移动支付应用变得异常火热。

然后移动安全问题接憧而至，主要分为移动断网络安全和客户端应用安全。目前移动 APP 软件保护方面还处于初级阶段，许多厂商对 APP 安全认识不够深 入，产品未经过加密处理，使得逆向分析者能够通过逆向分析、动态调试等技术来破解 APP，这样 APP 原本需要账号密码的功能可以被破解者顺利绕过，使得厂 商利益严重受损。

对未加壳的 APP 进行动态调试，通常可以非常顺利且快速地绕过一些登陆限制或功能限制。本文将以安卓 APP 为例，来详细介绍一下移动 APP 动态调试技术。

0x01 调试环境搭建

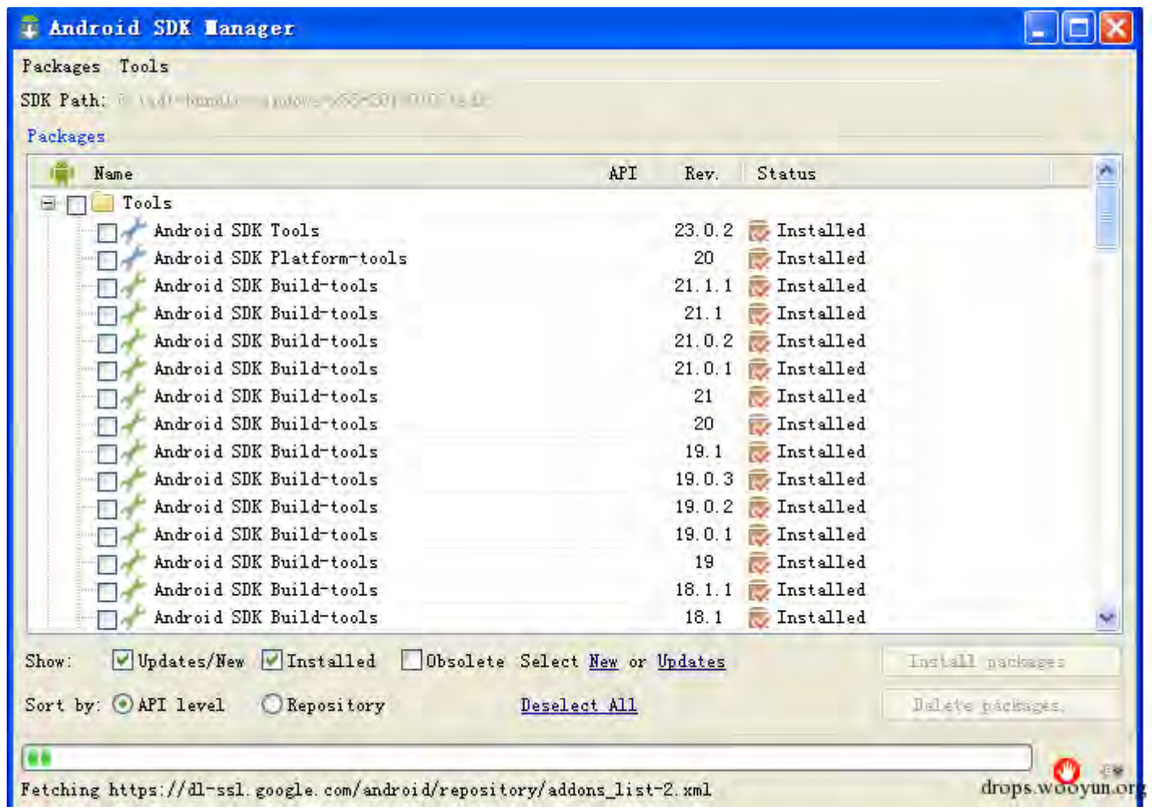
1.1 安装 JDK

JAVA 环境的搭建请自行查找资料，这里不做详述。

1.2 安装 Android SDK

下载地址：http://developer.android.com/sdk/index.html。

下载完安装包后解压到任意一目录，然后点击运行 SDK Manager.exe，然后选择你需要的版本进行安装，如图：



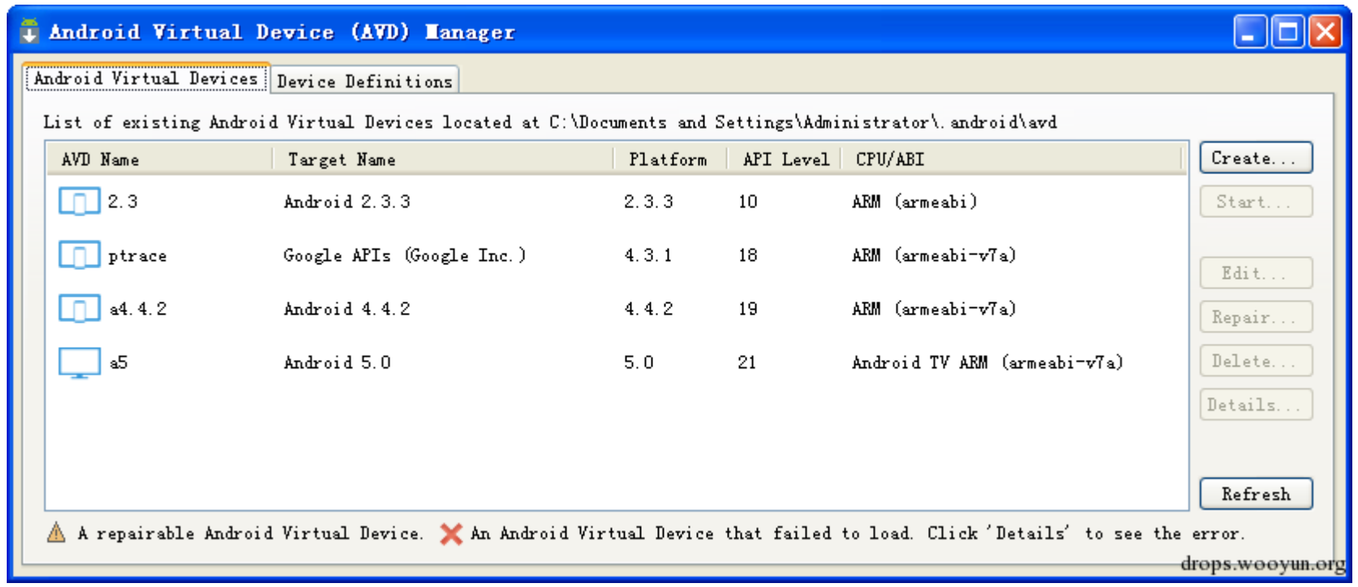
1.3 安装 Eclipse 集成开发环境

下载地址：http://www.eclipse.org/downloads。选择 Eclipse for Mobile Developers，解压到任意目录即可。

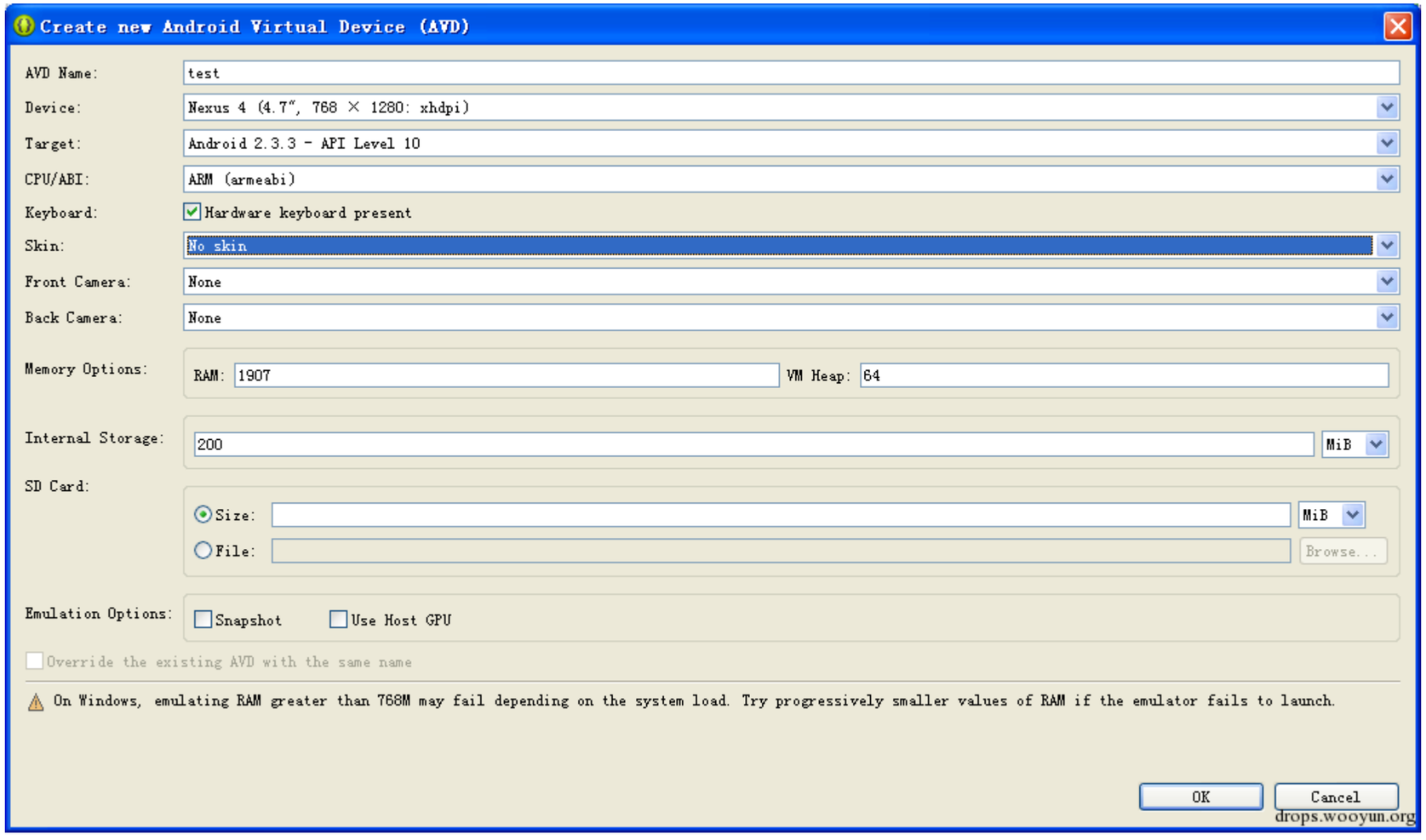
1.4 创建 Android Virtual Device

动态调试可以用真实的手机来做调试环境，也可以用虚拟机来做调试环境，本文采用虚拟机环境。因此创建虚拟机步骤如下：

1 打开 Eclipse ->windows->Android Virtual Device



2 点击 Create，然后选择各个参数如图：



这里 Target 就是前面步骤中安装的 SDK 选择任意你觉得喜欢的版本就可以。点击 OK 就创建完毕。

1.5 安装 APK 改之理

这个是一个很好用的辅助调试的软件，请自行搜索下载。

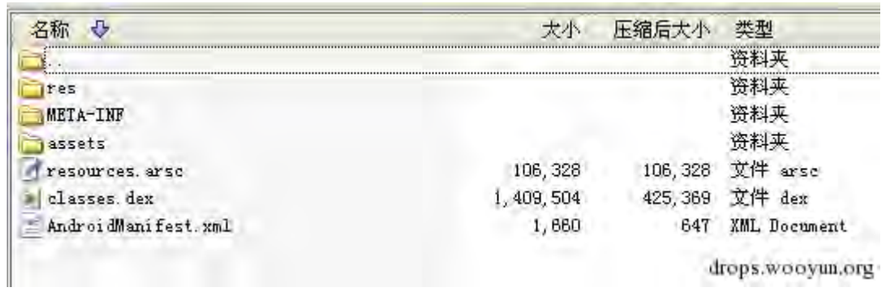
1.6 安装 IDA6.6

IDA6.6 开始支持安卓 APP 指令的调试，现该版本已经提供免费下载安装，请自行搜搜。

0x02 Dalvik 指令动态调试

2.1 准备工作

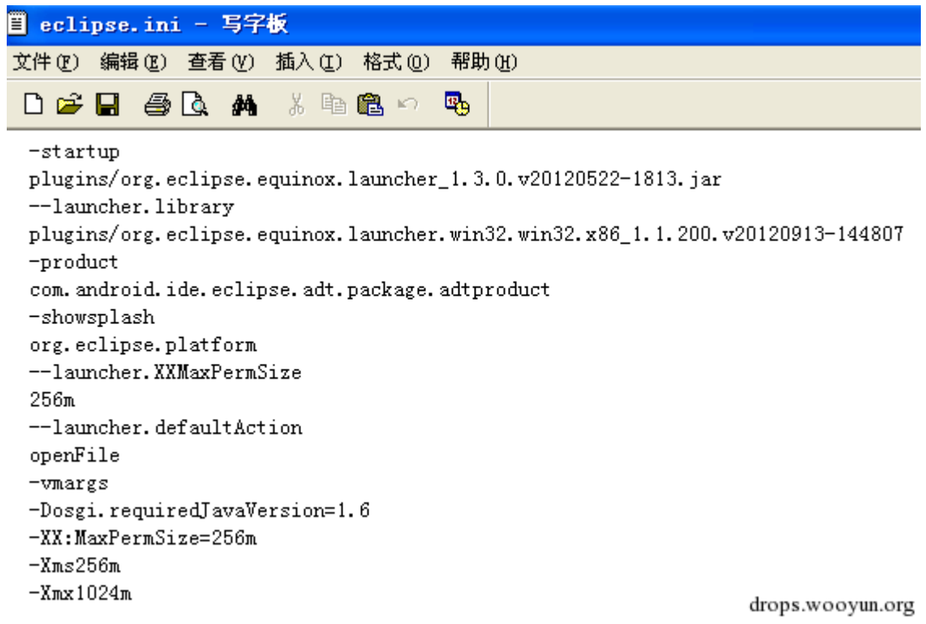
安卓 APP 应用程序后缀为 apk，实际上是一个压缩包，我们把它改后缀为 rar 打开如图：



其中 classes.dex 是应用的主要自行程序，包含着所有 Dalvik 指令。我们用 APK 改之理打开 apk，软件会自动对其进行反编译。反编译后会有很多 smail 文件，这些文件保存的就是 APP 的 Dalvik 指令。

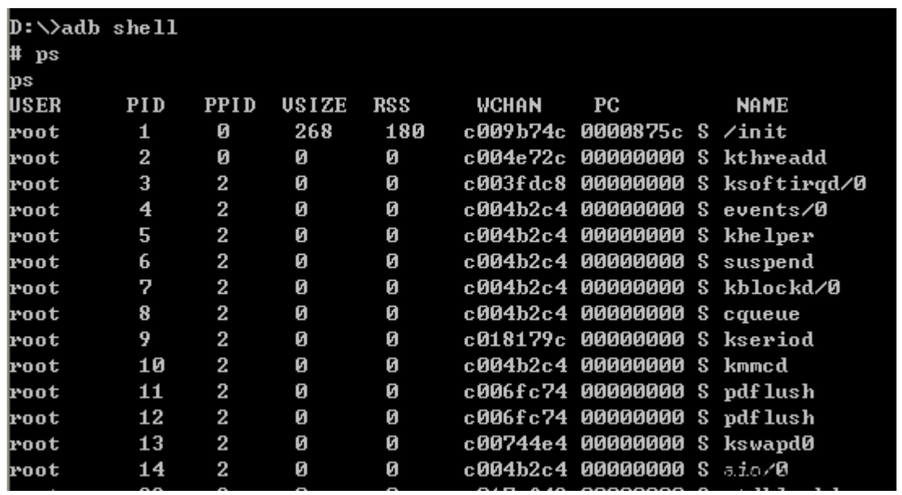
在 APK 改之理里双击打开 AndroidManifest.xml，为了让 APP 可调试，需要在 application 标签里添加一句 android:debuggable=”true” 如图：

然后点击保存按钮，然后编译生成新的 apk 文件。接着打开 Eclipse ->windows->Android Virtual Device，选择刚才创建的虚拟机，然后点击 start，虚拟机便开始运行。偶尔如果 Eclipse 启动失败，报错，可以同目录下修改配置文件：



把配置参数原本为 512 的改为 256 原本为 1024 的改为 512，然后再尝试启动。

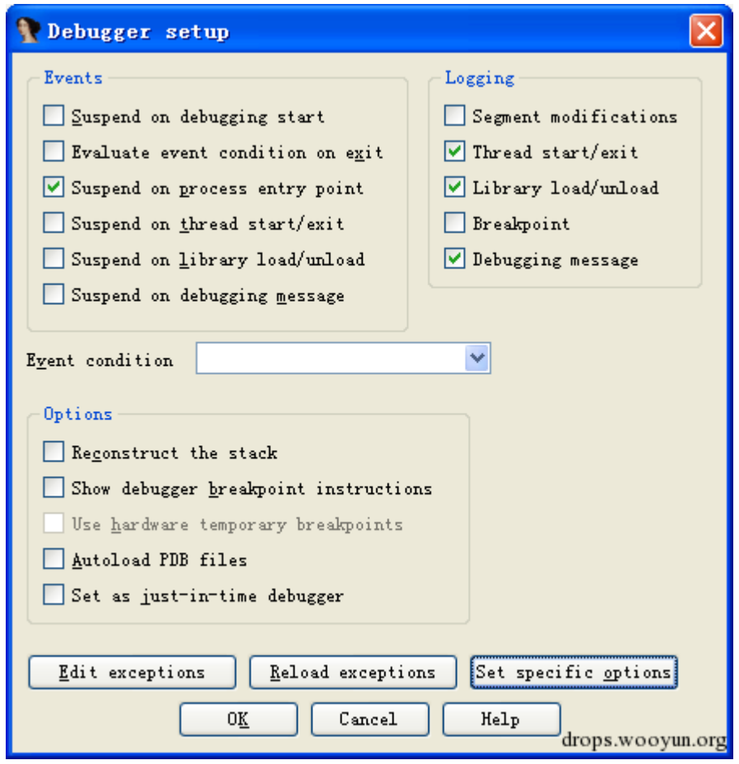
在 SDK 安装目录有个命令行下的调试工具 adb shell，本机所在目录为 E:\adt-bundle-windows-x86-20140702\sdk\platform-tools，把 adb.exe 注册到系统环境变量中，打开 dos 命令行窗口执行 adb shell 就可以进入 APP 命令行调试环境，或者切换到 adb 所在目录来执行 adb shell。



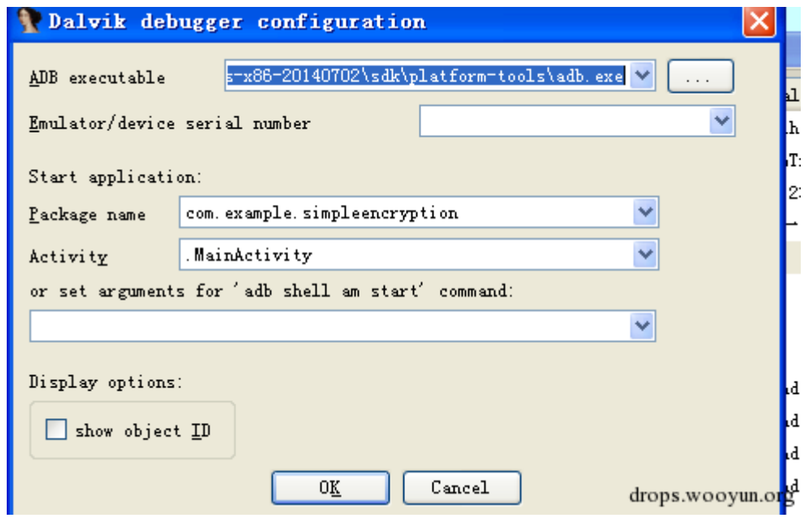
这里先不进入 adb shell，在 DOS 命令行下执行命令：adb install d:\1.apk 来安装我们刚才重新编译好的 APK 文件。安装完毕会有成功提示。

2.2 利用 IDA 动态调试

将 APP 包里的 classes.dex 解压到任意一目录，然后拖进 IDA。等待 IDA 加载分析完毕，点击 Debugger->Debugger Options 如图



按图所示勾选在进程入口挂起，然后点击 Set specific options 填入 APP 包名称和入口 activity 如图：



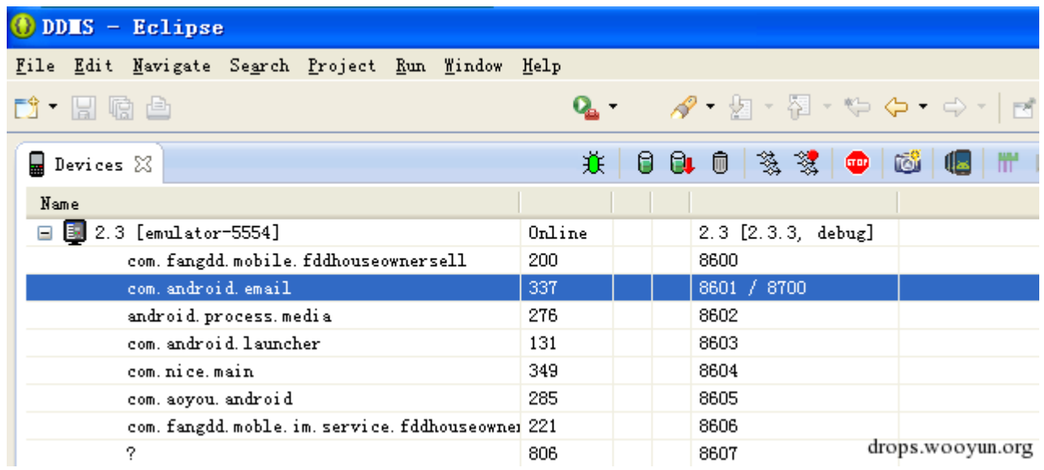
其中包的名称和入口 activity 都可以通过 APK 改之理里的 AndroidManifest.xml 文件获取：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.example.simpleencryption">
        <application android:allowBackup="true" android:debuggable="true" android:icon="@drawable/creakme_bg2" android:label="@string/app_name" android:theme="@style/AppTheme">
            <activity android:label="@string/app_name" android:name=".MainActivity">
```

然后在 IDA 点击 Debugger->Process Options

其他默认不变，端口这里改为 8700。这里默认端口是 23946，我在这里困扰了很久，就是因为这个端口没有改为 8700 所致。然后我们看看这个 8700 端口是怎么来的。在 Android SDK 里提供了一款工具 DDMS，用来监视 APP 的运行状态和结果。在 SDK 的 TOOLS 目录有个 DDMS.BAT 的脚步，运行后就会启动 DDMS。由于 我的本机安装了 SDK 的 ADT 插件，DDMS 集成到了 Eclips 中，打开 Eclips->Open perspective->ddms 就启动了 DDMS。

如图所示：

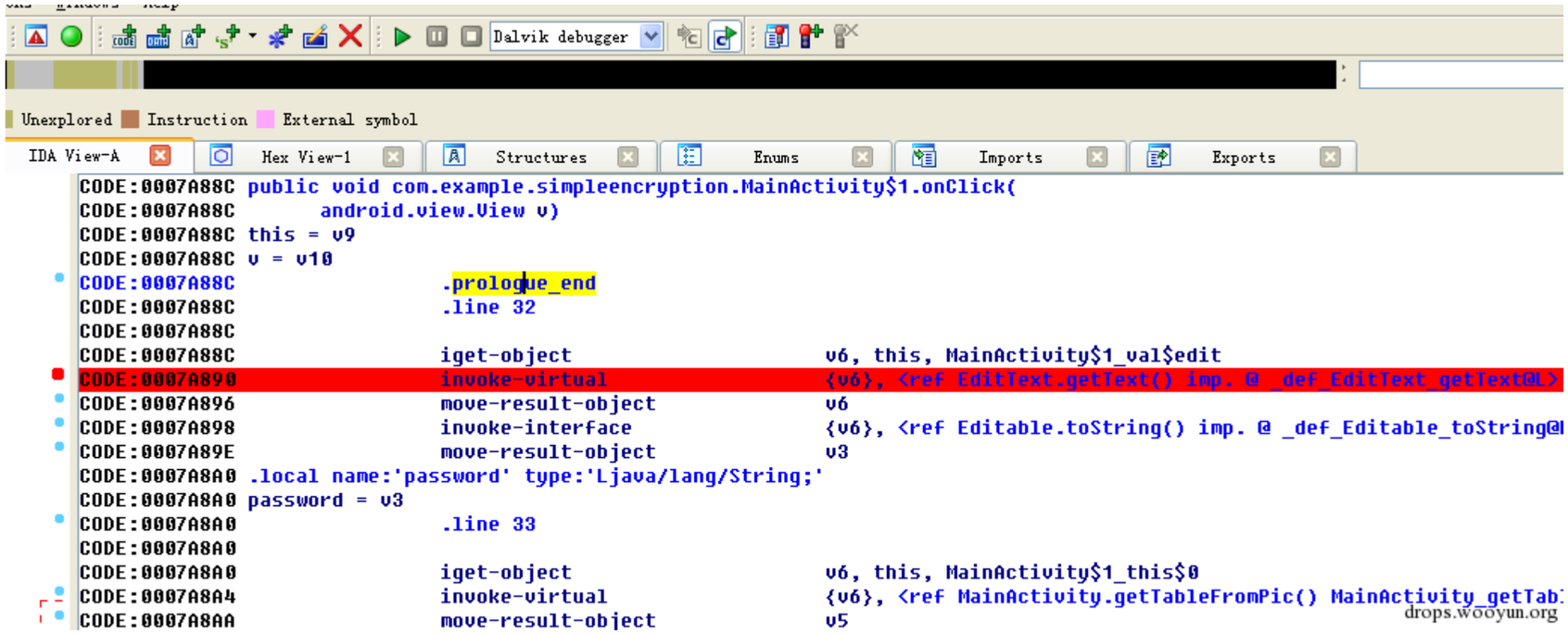


在 DDMS 选中某个进程后面就会注释出它的调试端口,本机这里是 8700。

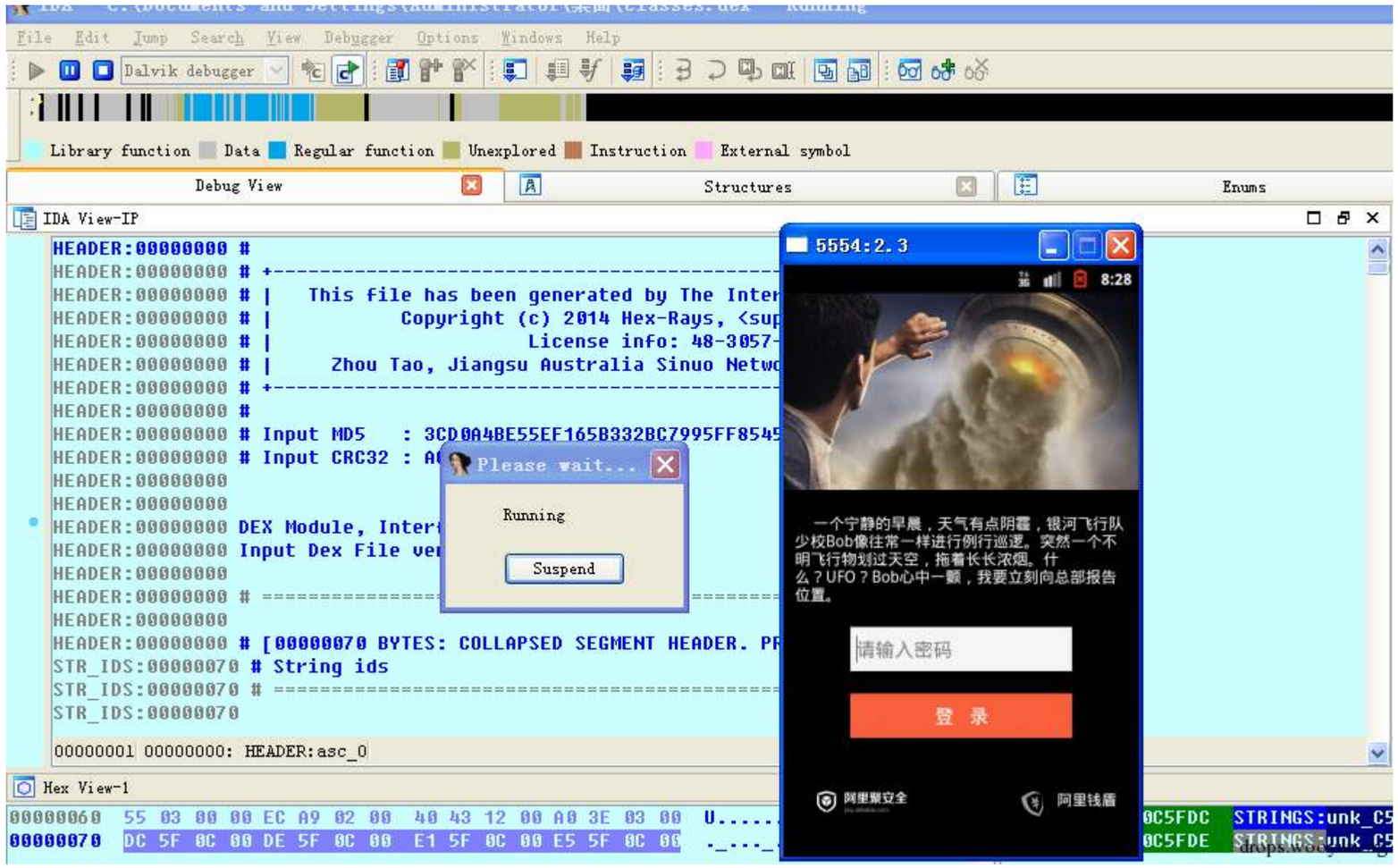
到此所有的工作就准备就绪，然后就可以下断点来调试该 APP 了。我们在 APK 改之理中在 com 目录下查看 smali 文件 发现 MainActivity.smali 里有一个感兴趣的函数 getPwdFromPic()，那么我们就对它下断以跟踪 APP 的运行。

在 IDA 里搜索字符串 getPwdFromPic，发现 onClick 有调用该函数

我们在 onClick 函数开始位置按 F2 下断如图：

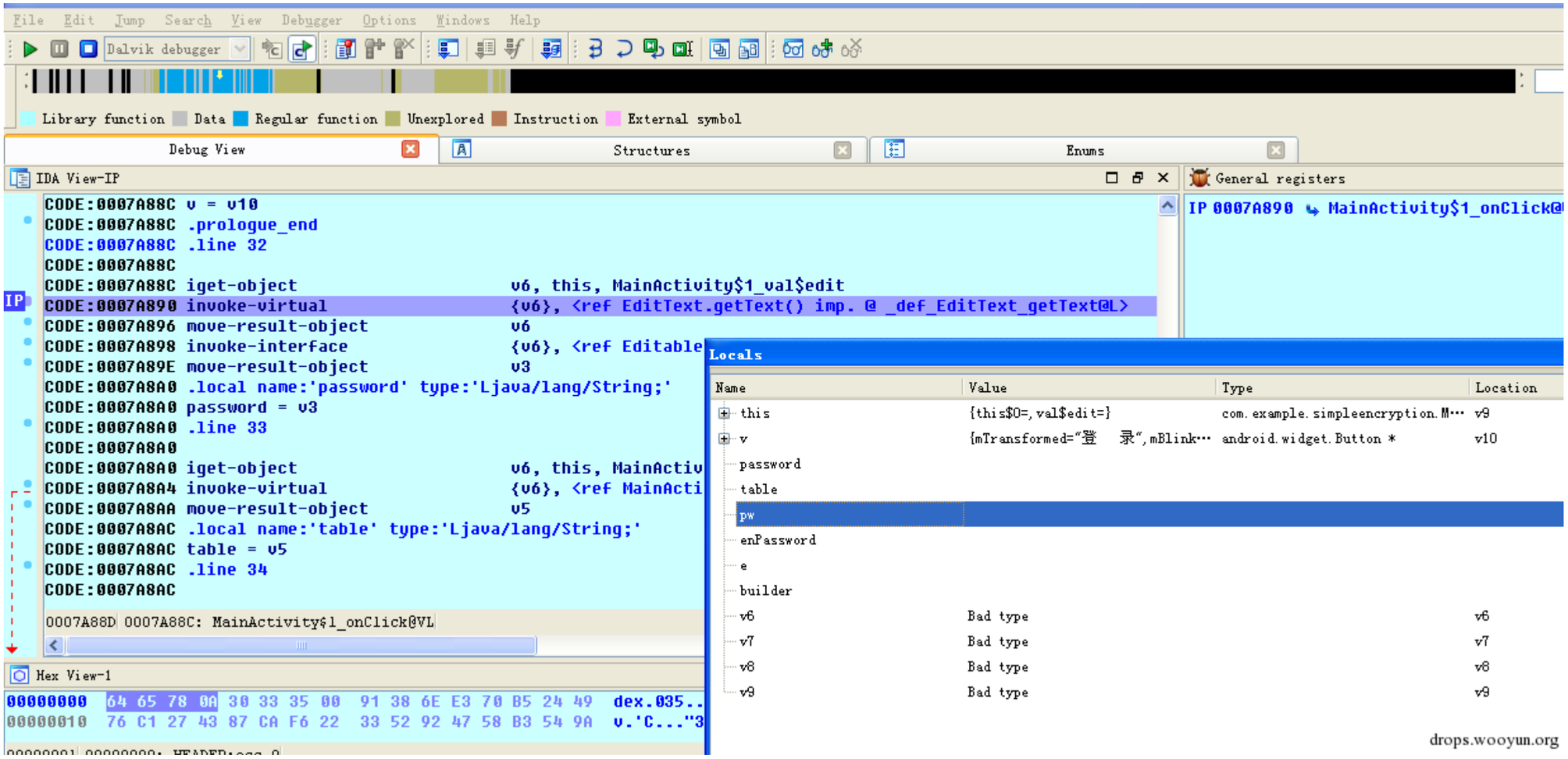


然后点击上图中绿色三角形按钮启动调试如图：

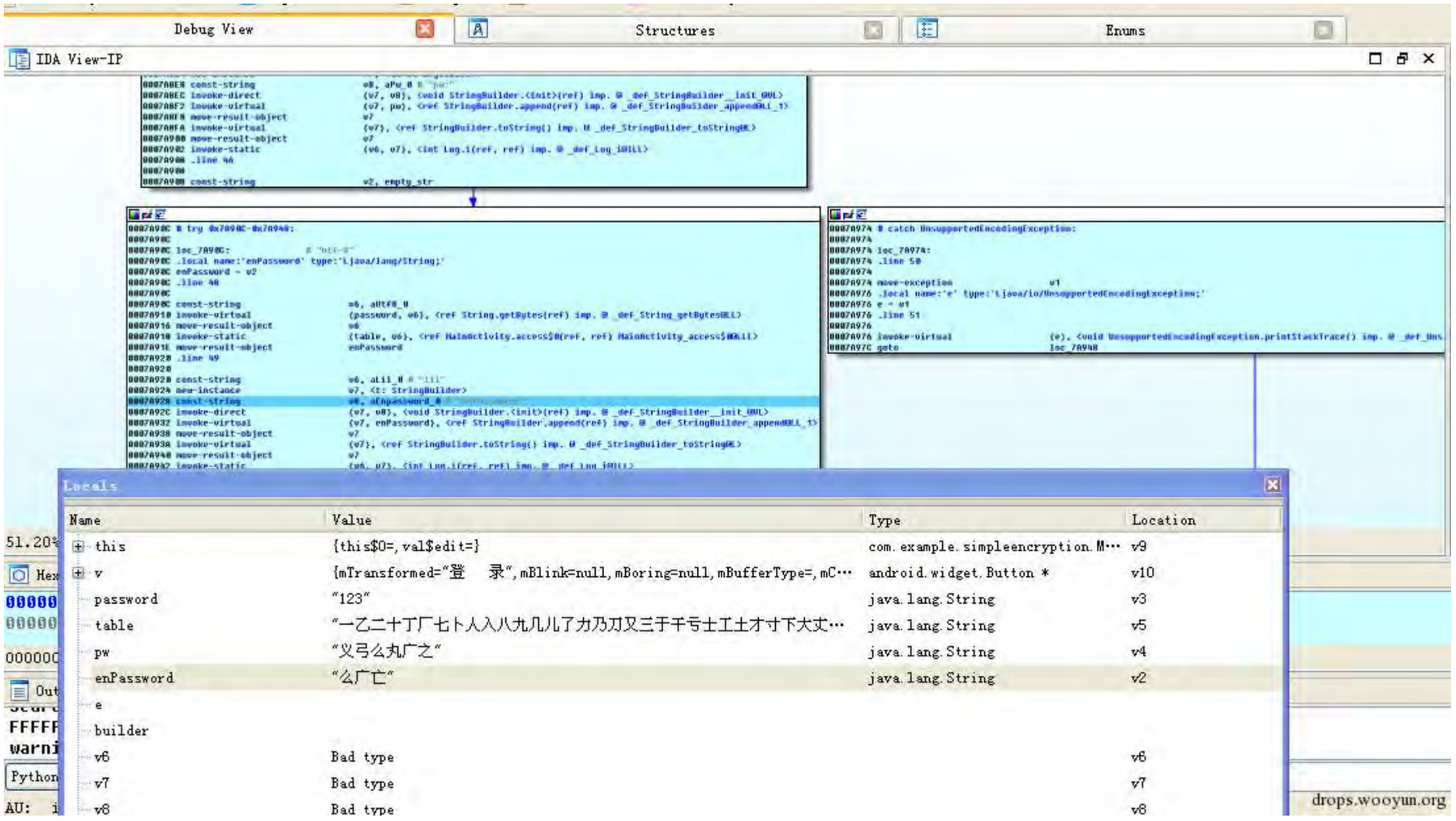


调试过程中有一个问题出现了很多次，浪费了我大量的时间，就在写文章的时候，操作时还是遇到了这样的问题。就是点击启动后 IDA 提示 can’t bind socket，琢磨了很久终于找到原因了，当打开过一次 DDMS 后 每次启动 Eclips 都会启动 DDMS 而 8700 端口正是被这个 DDMS 给占用了，然后每次都会启动失败，解决办法就是 虚拟机运行起来后关闭掉 Eclips，这时一切就正常了！

事例中是一个 APP crackme 提示输入密码才能进入正确界面。这个时候我们输入 123，点击登陆，IDA 中断在了我们设置断点的地方，这时选中 ida->debugger->use source level debugger，然后点击 ida->debugger->debugger windows->locals 打开本地变量窗口，如图：



然后按 F7 或 F8 单步跟踪程序流程，同时可以观察到变量值的变化，也可以在 IDA 右键选择图形视图，可以看到整个 APP 执行的流程图:



如上图所示 变量窗口中我们输入了 123 被转化成的密码是么广亡，pw 变量也显示出了正确的密码，其实这个时候已经很容易判断出正确密码了。

0x03 Andoid 原生动态链接库动态调试

通常为了加密保护等措施，有时 dex 执行过程中会调用动态链接库文件，该文件以 so 为后缀，存在于 APP 文件包里。

这里我们以动态附加的方式来调试原生库。

3.1 准备工作

1、将 IDA->dbgsrv 目录下的 android_server 拷贝到虚拟机里，并赋予可执行权限

DOS 命令分别为：

```
adb shell pull d:\ android_server /data/data/sv
adb shell chmod 755 /data/data/sv
```

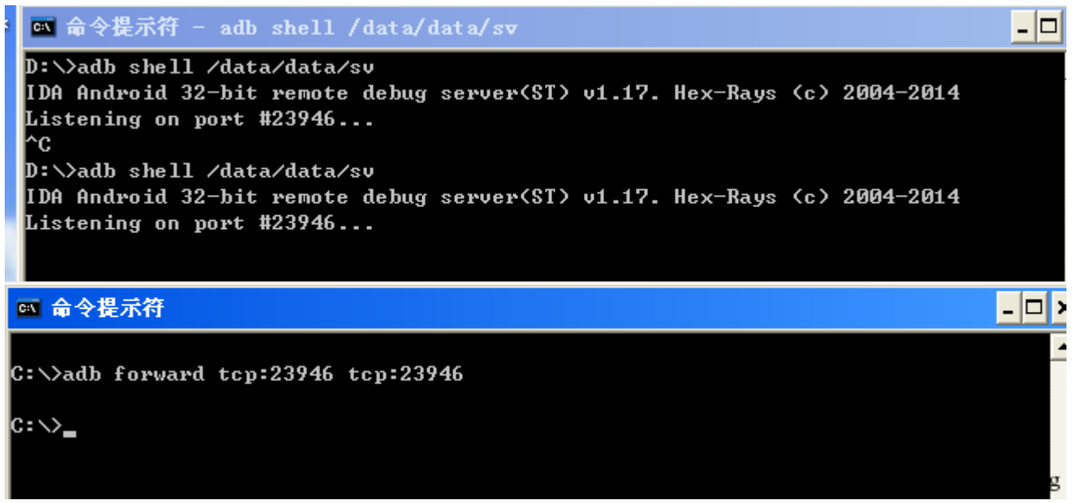
2、启动调试服务器 android_server

命令：adb shell /data/data/sv

服务器默认监听 23946 端口。

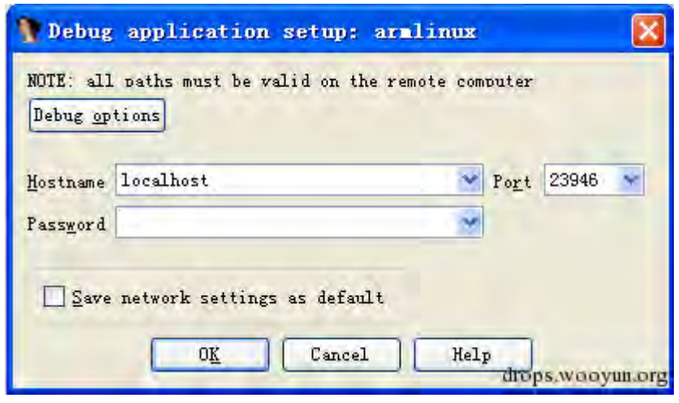
3、重新打开 DOS 窗口进行端口转发，命令：

adb forward tcp:23946 tcp:23946 如图：

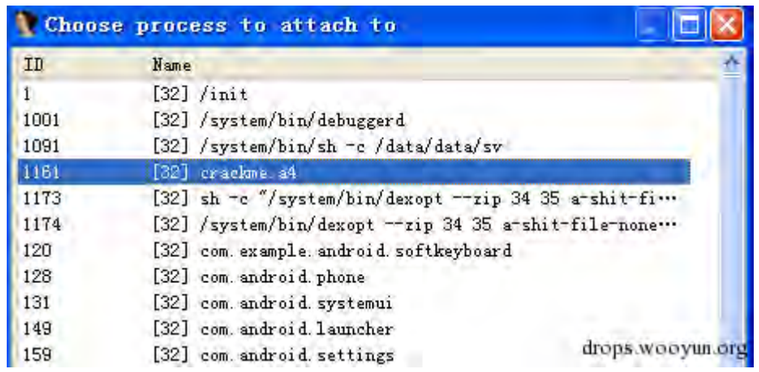


3.2 利用 IDA 进行动态调试

1、虚拟机里启动要调试的 APP 2、启动 IDA，打开 debugger->attach->remote Armlinux/andoid debugger



端口改为 23946 其他保持不变，点击 OK



如上图，选中要调试的 APP 的数据包名，然后点击 OK。

正常情况下，IDA 会把 APP 进程挂起。

3、由于当前程序不是在动态链接库领空，这时我们要重新打开一个 IDA，用它打开需要调试的 so 文件，找到需要下断的位置的文件偏移，并做记录，然后关闭后面打开的这个 IDA。

4、在原 IDA 界面按下 ctrl+s 键，找到并找到需要调试的 so，同时记录该文件的加载基址。然后点击 OK 或者 cancel 按钮关闭对话框。

5、按下快捷键 G 输入基址+文件偏移所得地址，点击 OK 就跳转到 SO 文件需要下断的地方，这时按下 F2 键设置断点。当 APP 执行到此处时便可以断下来。

3.3 在反调试函数运行前进行动态调试

程序加载 so 的时候，会执行 JNI_OnLoad 函数，做一系列的准备工作。通常反调试函数也会放到 JNI_OnLoad 函数里。进行 4.2 中第 2 步时也许会遇到如下情况：

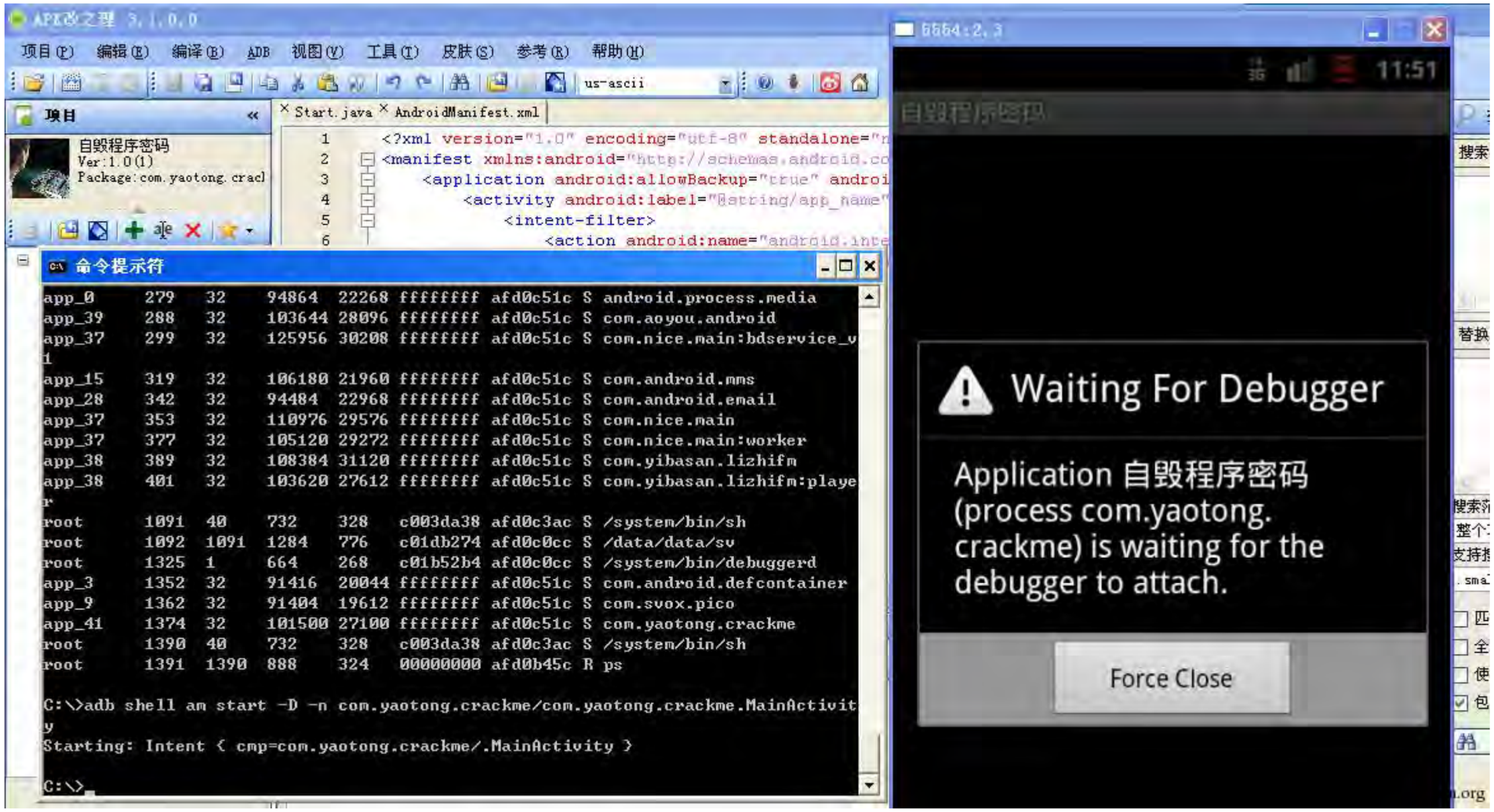
这时 APP 检测到了调试器，会自动退出，那么这时调试策略需要有所改变。

接着 4.1 第 3 步后，在 DOS 命令行执行命令：

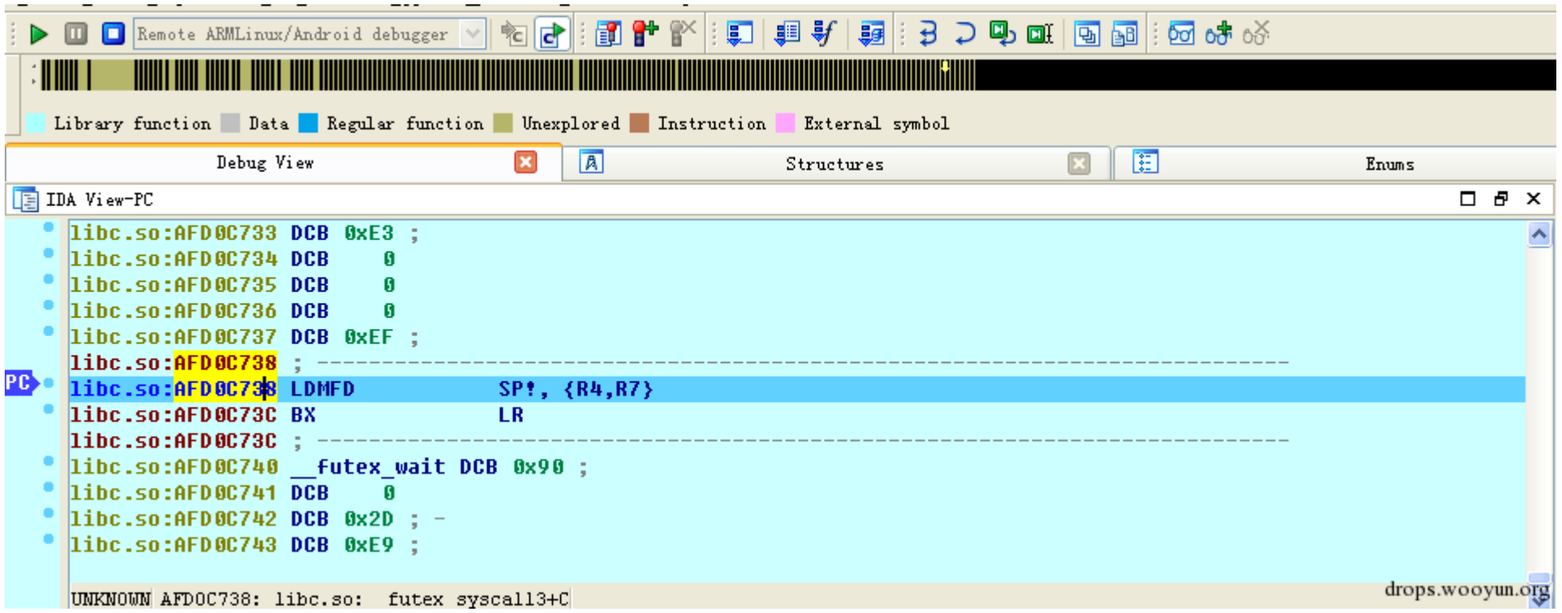
```
adb shell am start -D -n com.yaotong.crackme/com.yaotong.crackme.MainActivity
```

来以调试模式启动 APP 如图：

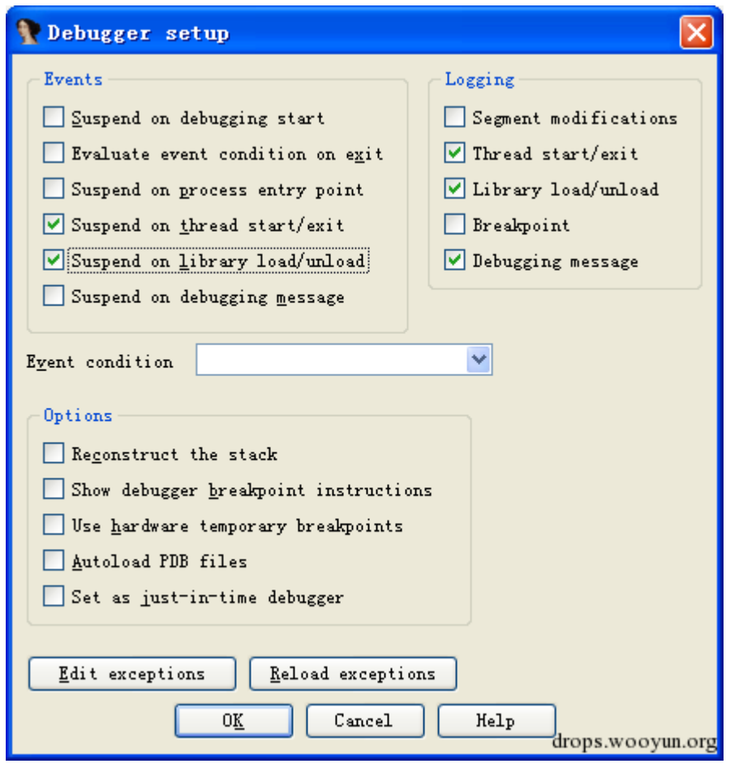
com.yaotong.crackme 是 APP 包名称，com.yaotong.crackme.MainActivity 是执行入口 这些可以用 APK 改之理查看。



这时由于 APP 还未运行，那么反调试函数也起不了作用，按照 4.2 中第 2 步把 APP 挂起。这时 IDA 会中断在某个位置



然后点击 debugger->debugger opions 设置如下：



点击 OK 后按 F9 运行 APP，然后再 DOS 命令下执行命令：

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

这时 APP 会断下来，然后按照 4.2 中的 3、4、5 补找到 JNI_OnLoad 函数的地址并下断，然后按 F9 会中断下来。然后便可以继续动态跟踪调试分析。

0x04 主要参考资料

1、《Andoroid 软件安全与逆向分析》

2、看雪论坛安卓安全版

3、吾爱破解论坛安卓版

感谢看雪论坛好友：我是小三、QEver、非虫等的热心指教！

转载自：http://drops.wooyun.org/mobile/5942　作者:netwind

Android 逆向经验总结

1.有选择性选择设备

adb devices -l
List of devices attached
emulator-5554　　device product:sdk model:sdk device:generic
adb -s emulator-5554 shell
root@android:/ #

2.调试模式启动程序

am start -D -n com.yaotong.crackme/com.yaotong.crackme.MainActivity
此时应用程序会等待被调试。此时可以做 IDA 附加等动作。
jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8604
流程
执行 android_server
端口转发 adb forward tcp:23946 tcp:23946
调试模式启动程序 adb shell am start -D -n 包名/类名
IDA 附加,Debugger->Debugger Options 可以设置调试选项。
静态找到目标函数对应所在模块的偏移地址
Ctrl+S 找到对应模块的基地址，两个地址相加得到最终地址
G 跳转至地址，然后下断
F9 运行
执行 jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700
断下，进行调试

3.kill 用法

3.1 kill -19 <pid> 暂停进程
3.2 IDA 附加，下断点
3.3 kill -18 <pid> 运行进程

4.脱壳

```
if(dvmDexFileOpenPartial(addr,len,&pDvmDex) != 0)
{
  ALOGE("Unable to create DexFile");
  goto bail;
}
```

libvm.so　载入 IDA，搜索字符串：Unable to create DexFile
http://blog.csdn.net/zhangmiaoping23/article/details/44154265

android 动态调试笔记

低调的天空

1、修改 apk 配置文件 AndroidManifest.xml

修改配置文件<application>子项，添加 android:debuggable="true" 属性，使 apk 处于可调式状态，打包回编即可。

```
<application android:allowBackup="true" android:debuggable="true" android:icon="@drawable/ic_launcher" android:label="@string/app_name">
```

2、调试启动 apk（两种方式）

1)使用 adb　shell am（没用过）

使用格式为 【adb shell am start -D -n 包名/包名＋类名】的命令，就可以从控制台启动一个 Activity
该命令的具体用法，参见 adb shell am

2)使用手机自带的调试功能（简单方便）

在手机中的开发者选项中，选择待调试的程序，然后正常单击程序，此时会弹出一个 title 为“Waiting For Debugger”的对话框，提示用户该应用程序正在等待 debugger attach

3、IDA 附加 app

在手机中以 root 身份启动 android_server
在主系统中,使用 adb forward tcp:23946 tcp:23946 命令转发端口数据
完成以上两步后，就可以使用 ida 附加到应用程序。附加完成后，在 ida->Debugger->Debugger Options 中，设置 event 断点，添加 library load/unload 断点,，这样，在 app 启动之后，每 load/unload 一个 so 的时候，都会断住，此时，用户就可以去设置断点，一遍后续调试

4、启动 app

经过以上三步,app 处于待调试状态，此时需要启动 app，让它跑起来。
在主系统中，使用 ddms 查看 app 的调试端口号（一般为 8700），然后使用如下命令：

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

此时，app 调试状态解除，开始正常运行。
但是你会发现，“Waiting For Debugger”对话框不见了，但是 app 还是没有运行，这是因为在 ida 附加的时候，自动断住了。只要在 ida 中，按下 F9，就可以让 app 真正的跑起来了。

使用 ida 调试 android 的 c 程序

普通调试

1. 用 ida 读入 libcocos2dcpp.so
2. 将 ida 目录下的 dbgsrv/android_server 拷贝到 android 手机
3. adb shell 进入, 并以 root 运行 android_server
4. 手机运行 app, ida 使用 Remote ARM Linux/Android debugger, 并 attach 对应的进程
5. 下断点调试即可

app 在 attach 之前不运行

一般先运行 app，这个时候比如 lib 就会被加载并运行，想断点某些入口点怎么办?

1. 在 adb 的 shell 中使用 am start -D -n 包名/Activity 类名运行 app, 这个时候 app 会显示 waiting debugger to attach
2. ida 慢慢 attach;在想要断的地方下好断子 (这个 attach 不是上面的 attach)
3. 使用 jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700 真正 attach, 此时游戏运行

attach 之后只看到一个进程:

android_server 没有用 root 运行，或者 shell 并不是 root，比如魅族 mx3，虽然有 root 权限，但是 shell 不是真 root

魅族 MX3 获得 shell root

mx3 的 rom 默认可以开启系统权限，但是使用 shell 还是无法切换到 root 用户；解决办法是安装 SuperSU 软件

放在/sdcard 里不能运行

sdcard 默认会格式化为 FAT32，所以在这个里面的所有文件都没有-x 权限，解决办法就是放到/data/local/tmp 目录下

mac 怎么调试

在 Paralle 跑 win8，win8 跑 ida，ida 是通过 ip 去 attach 的，那就得让 win8 连到外界能看到的网络，虚拟网卡使用桥接，接到比如公司 wifi 上

如果用真机，真机也连到 wifi 上就行了

如果是 Genymotion，VitrualBox 里的虚拟网卡同样设置成桥接，连到 wifi，那 win8 和 Genymotion 就在同一个网络了

然后在路由器把 ip 固定住

http://www.zhaoxiaodan.com/android/%E4%BD%BF%E7%94%A8IDA%E8%B0%83%E8%AF%95android%E7%9A%84c%E7%A8%8B%E5%BA%8F.html

让 VirtualBox 中 Windows 版的 IDA Pro 6.8 调试 Genymotion 或者 Android Emulator 的 APK

宿主机：
i7-6700K
64G RAM
240G SSD
4T HDD
Debian GNU/Linux stretch

0） 准备工作
安装了 Genymotion 2.6.0
virtualbox-qt 5.0.14

然后用 VirtualBox 安装了 Windows XP SP3，在 XP 里安装了 IDA Pro 6.8 和 ADB （可以在 http://adbshell.com/下载单独的 adb.zip 或者 直接完成安装 android sdk）

1）运行 Genymotion，开启一个 Android 4.1.2

adb devices
192.168.56.101:5555 device

adb 是通过 tcp 通道传输数据的 （genymotion 是通过 virtualbox 虚拟了另外一个网络接口）

2）我们希望 windows 里的 adb 也能访问 genymotion 的
在 cmd 里 ping 一下 192.168.56.101， 可以访问，然后在 cmd 控制台运行

adb.exe connect 192.168.56.101
adb devices

如果是 Android Emulator, 则需要配置下 iptables

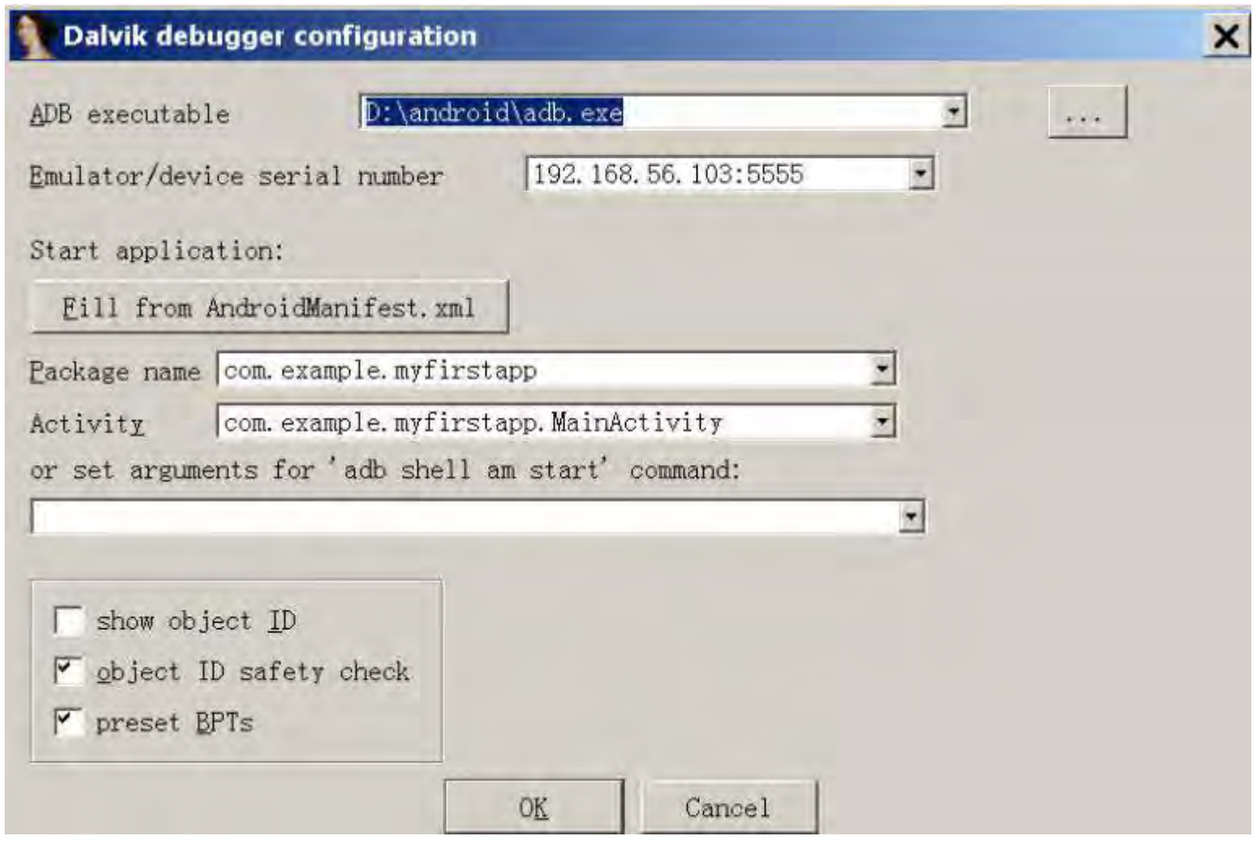
iptables -t nat -A OUTPUT -p tcp -d 192.168.1.112 --dport 5555 -j DNAT --to 127.0.0.1:5555

然后在 windows 里运行

adb.exe connect 192.168.1.112
adb devices

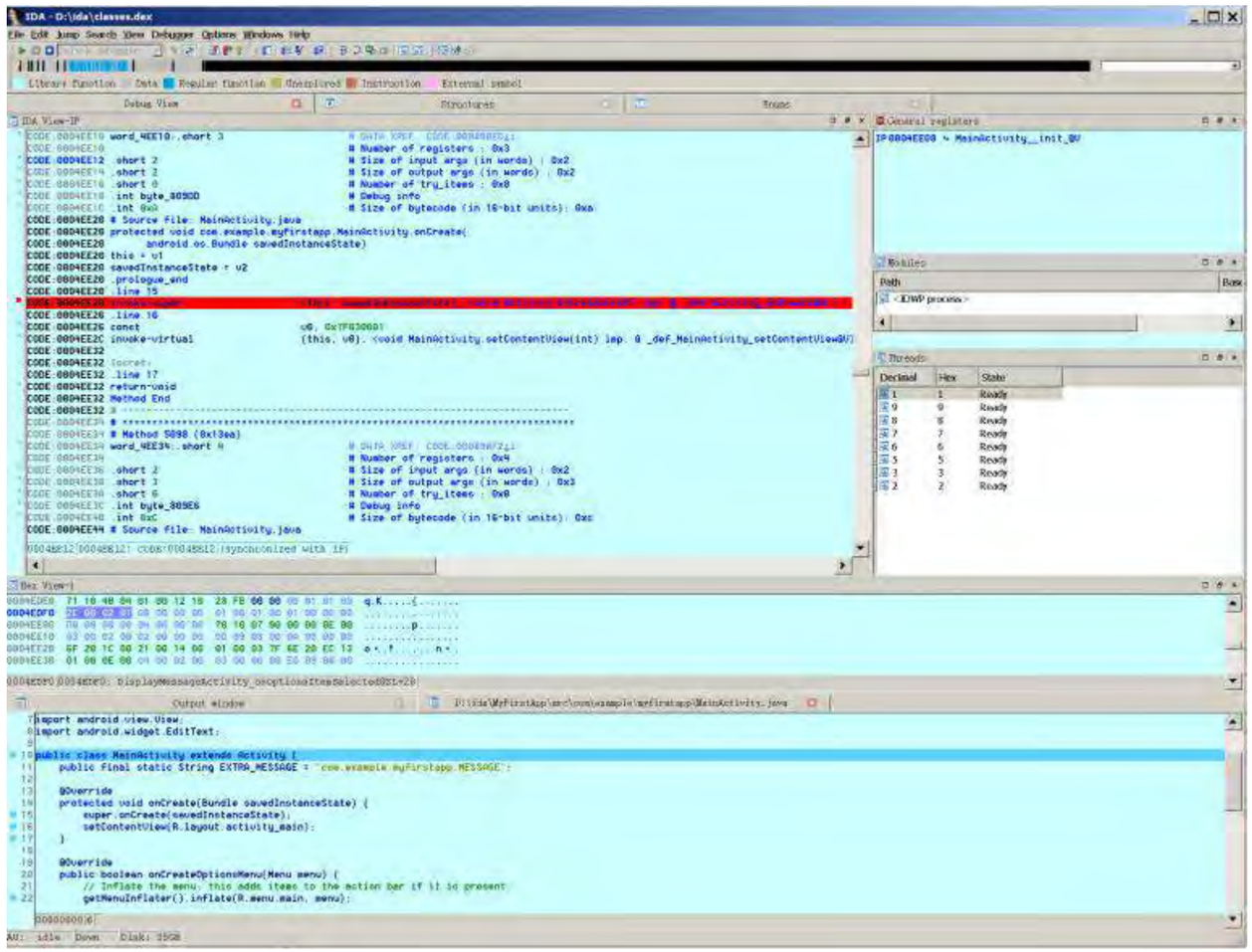
3）用 IDA 打开 apk

4)在 IDA 里设置 Debugger/Debugger Options/Set specific options
其中 ADB executable 设置 adb.exe
Fill from AndroidManifest.xml 按钮点开后，选中 apk 文件
Preset BPTs 勾选上

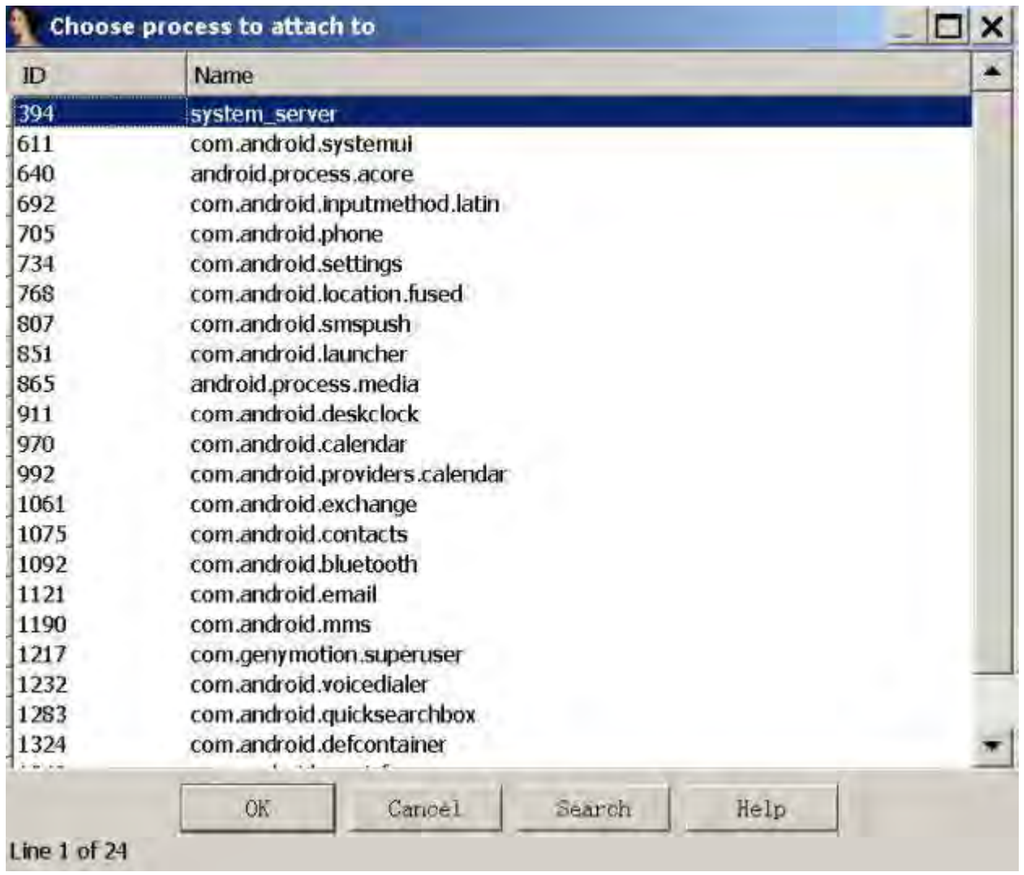


5）如果需要源代码级别的调试，请在菜单里设置 Options/Sources path 为源代码所在的目录

6）开启进程 “Debugger/Start process” 或者 直接按 F9



7）或者 attache 到一个进程



http://zhiwei.li/text/category/reverse_engineering/

Dex 调试

IDA Pro 6.6 调试 dex 代码初体验

作 者： 火翼[CCG]
转载：大老

IDA 6.6 新添加了对 dex 文件的调试支持，由于工作原因，我第一时间拿到了这个版本，下面就是针对这个功能的一些简单体验。
IDA 对这个新功能提供了一个 PDF 文档进行说明，按照教程一步步来就可以完成对例子程序的调试，我这里选 sina 微博作例子简单讲一下实际调试的流程。
准备工作

根据 android 的官方文档，如果要调试一个 App 里面的 dex 代码，必须满足以下两个条件中的任何一个：

1） App 的 AndroidManifest.xml 中 Application 标签包含属性 android:debuggable=true

2） /default.prop 中 ro.debuggable 的值为 1

由于正常的软件发布时都不会把 android:debuggable 设置为 true，所以要达成条件 1）需要对 app 进行重新打包，这不仅每次分析一个 App 都重复操作，而且很多软件会对自身进行校验，重打包后执行会被检测到，所以想办法满足第 2）个条件是个一劳永逸的办法，我实际使用的方法就是满足第 二个条件。由于 default.prop 是保存在 boot.img 的 ramdisk 中，这部分每次重新启动都会重新从 rom 中加载，所以要到目的必须修改 boot.img 中的 ramdisk 并重新刷到设备中。我测试使用的设备为 Nexus 7，修改步骤如下：

a) 从 Google 官方网站下载到 boot.img,

b) 使用工具（abootimg，gunzip, cpio）把 boot.img 完全解开，获取到 default.prop

c) 修改 default.prop

d) 把修改后的文件重新打包成 boot_new.img

e) 使用 fastboot 工具把 boot_new.img 刷入设备（fastboot flash boot boot_new.img）

```
root@flo:/ # cat default.prop
cat default.prop
#
# ADDITIONAL_DEFAULT_PROPERTIES
#
ro.secure=0
ro.allow.mock.location=0
ro.debuggable=1
ro.adb.secure=0
persist.sys.usb.config=adb
```

图 1 修改后的 default.prop 内容

实际调试

解压缩 sina 微博的 apk，把 dex 文件拖到 ida 中进行分析（非常慢，文件会到 6G 多，估计 IDA 的 dex 加载器有 BUG），分析完成后按照 IDA 的教程对 Debugger Options 进行设置。

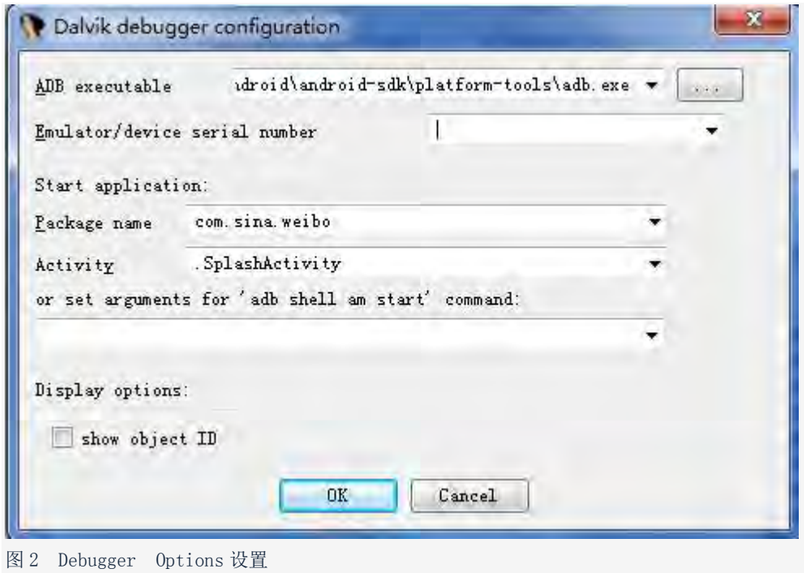


图 2 Debugger Options 设置

设置后找到 App 的入口 Activity——.SplashActivity, 在 SplashActivity_OnCreate 方法设置断点.

```
<activity android:theme="@style/Splash" android:name=".SplashActivity"
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter>
```

图 3　sina 微博入口 Activity 定义

```
CODE:00339704 # Source file: SplashActivity.java
CODE:00339704 protected void com.sina.weibo.SplashActivity.onCreate(
CODE:00339704       android.os.Bundle savedInstanceState)
CODE:00339704 this = v13
CODE:00339704 savedInstanceState = v14
CODE:00339704                 .line 93
CODE:00339704
CODE:00339704                 const/16                      v8, 0x400
CODE:00339708                 const/4                       v12, 1
CODE:0033970A                 const/4                       v11, 0
CODE:0033970C                 .prologue_end
CODE:0033970C                 .line 94
CODE:0033970C
CODE:0033970C                 invoke-static                 {}, <long System.curr
CODE:00339712                 move-result-wide              v5:v6
CODE:00339714 .local name:'startTime' type:'J'
CODE:00339714 startTime = v5
CODE:00339714                 .line 95
CODE:00339714
CODE:00339714                 invoke-static                 {}, <int Process.myPi
CODE:0033971A                 move-result                   v7
CODE:0033971C                 sput                          v7, SplashActivity_a
003396F4 003396F3: CODE:003396F3
```

图 4　在 SpalshActivity 的 OnCreate 方法设置断点

点击执行或者按 F9 即可开始运行 App，由于在入口 Activity 的 OnCreate 方法设置了断点，所以 App 执行后就会停在刚才设置的断点处等待用户操作。

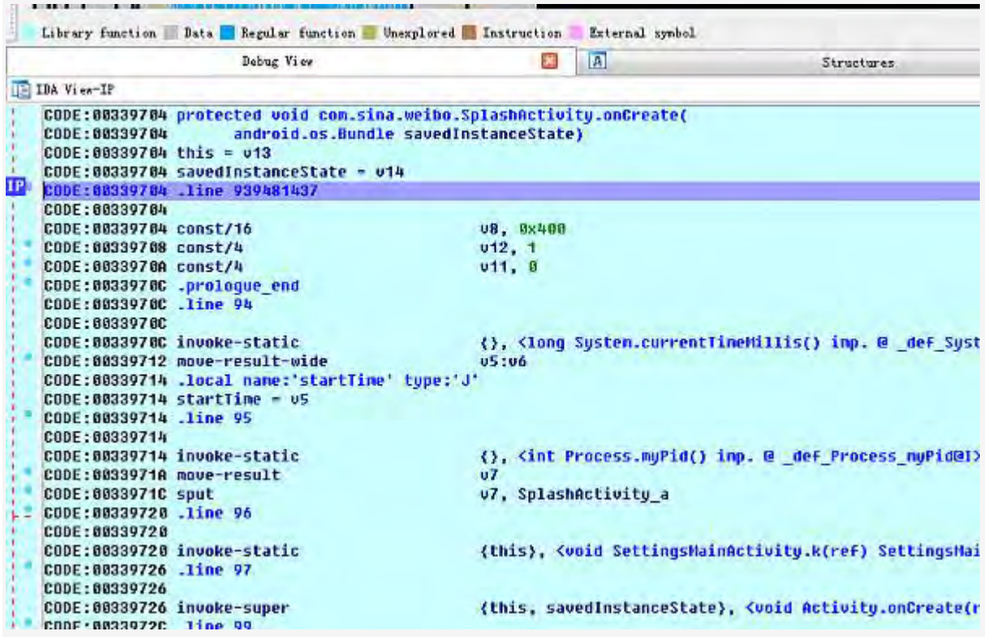


图 5　执行后程序中断在设置的断点处

利用 IDA6.6 进行 apk dex 代码动态调试

网上公开 IDA6.6 已经有一段时间，这个版本有个好处就是可以动态调试 java 代码。正好现在需要动态调试，所以顺便练习一下。

根据 android 的官方文档，如果要调试一个 apk 里面的 dex 代码，必须满足以下两个条件中的任何一个：

1.apk 中的 AndroidManifest.xml 文件中的 Application 标签包含属性 android:debuggable=”true”

2./default.prop 中 ro.debuggable 的值为 1

由于一般软件发布时都会把 android:debuggable 设置为 false，所以要达成条件 1 需要反编译原 apk，修改 AndroidManifest.xml 并进行重新打包，这样不仅麻烦，而且很多软件进行了加固，要破解很难。所以想办法满足第 2 个条件是个一劳永逸的办 法。

由于 default.prop 是保存在 boot.img 的 ramdisk 中，这部分每次重新启动都会重新从 rom 中加载，所以要到目的必须修改 boot.img 中的 ramdisk 并重新刷到设备中。修改步骤如下：

1.从 Google 官方网站下载到 boot.img

2.使用工具（abootimg，gunzip, cpio）把 boot.img 完全解开，获取到 default.prop

3. 修改 default.prop

4. 把修改后的文件重新打包成 boot_new.img

5. 使用 fastboot 工具把 boot_new.img 刷入设备（fastboot flash boot boot_new.img）

具体可参考：http://www.cnblogs.com/goodhacker/p/4106139.html

由于我们这篇文章的重点是如何动态调试 dex 代码，所以我们就假设该 dex 是可调试的。

1. 用 ida 打开 apk 文件，选择 dex 文件进行加载
2. 设置 debugger 选项，Debugger->Debugger options->Set specific options，按如图 1 所示进行设置，然后一路确定返回
3. 找到要下断点的位置，光标移到要下断点的那一行，按 f2 下断点
4. 手机开启调试选项，连接到电脑，运行 apk
5. 选中 IDA pro 窗口，按 f9 运行，如果出现如图 2 的画面，就说明设置成功，可以进行动态调试了。

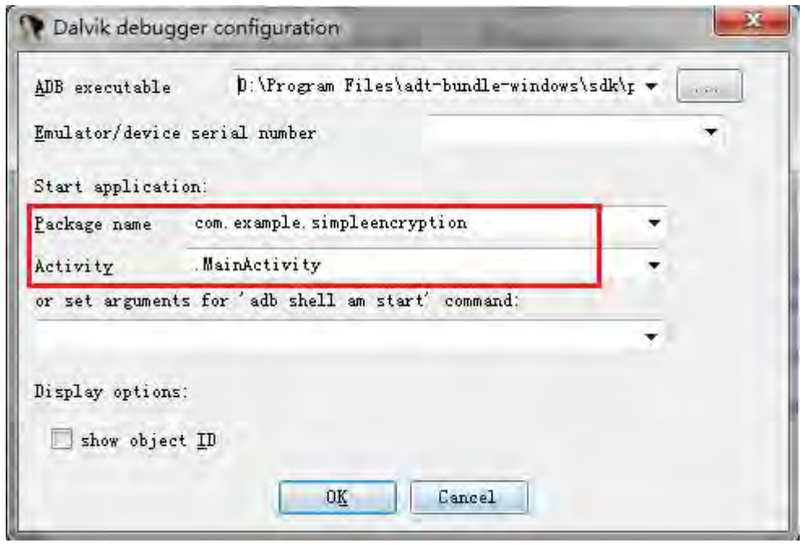


图 1 调试设置

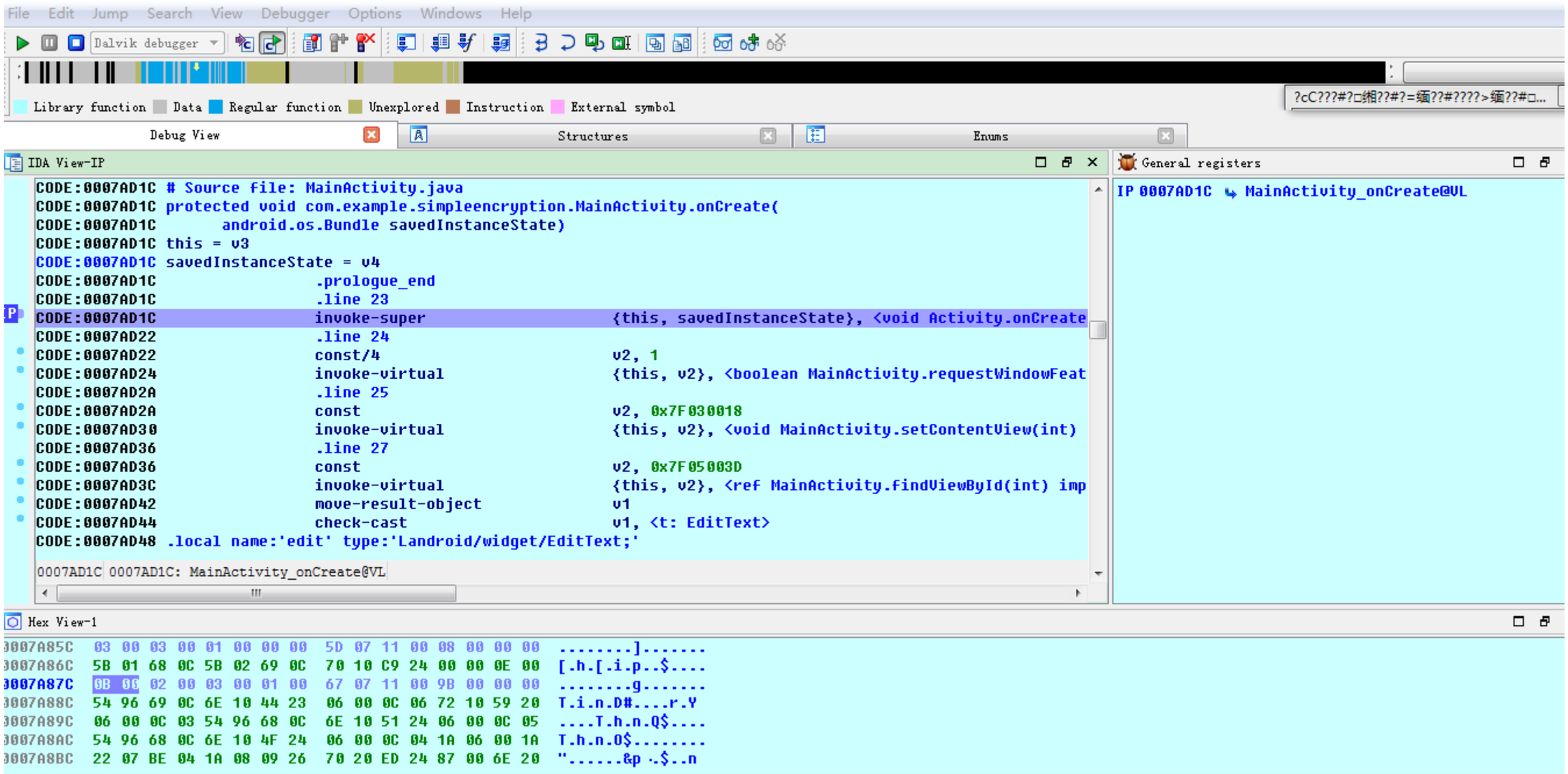


图 2 调试界面

注意：如果运行过程中一直显示如图 3 所示窗口，就需要关注一下你的手机屏幕，看是否是需要与用户进行交互了。

图 3

http://www.cnblogs.com/goodhacker/p/4257433.html

IDA 动态调试 Android 的 DEX 文件

本文涉及到的 apk，请在 github 下载 https://github.com/jltxgcy/AliCrack/AliCrackme_1.apk。

0x00

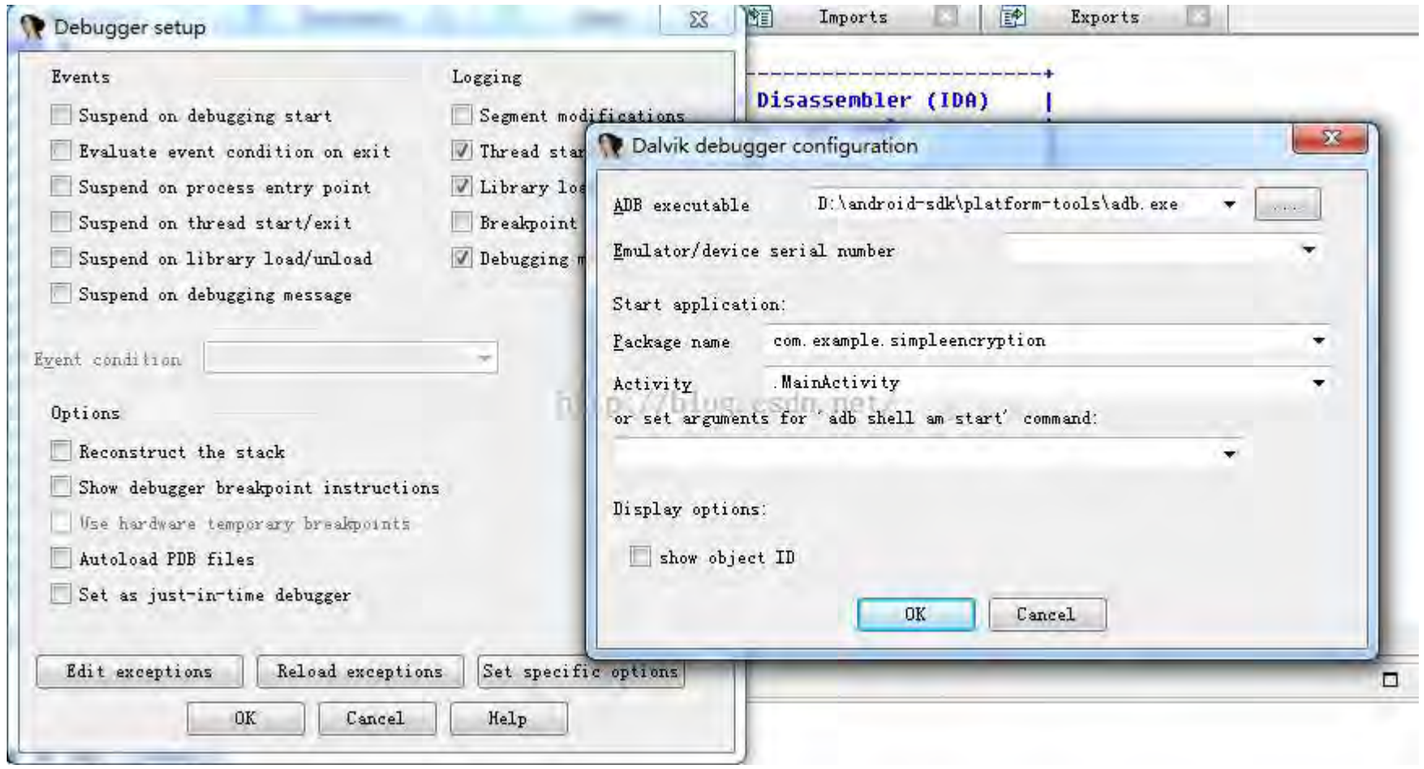
我们以阿里比赛第一题为例，来动态调试 dex 文件。参考 IDA 动态调试 Android 的 DEX 文件一文，首先 AndroidManifest.xml 里面 Android:debuggable="true"。

0x01

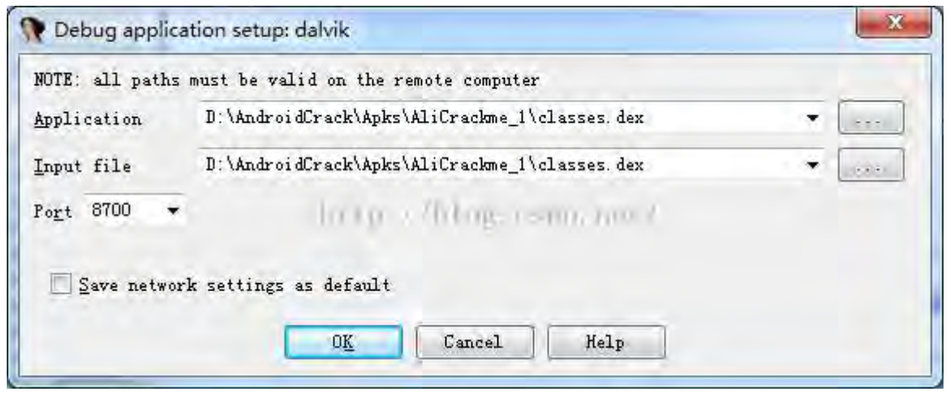
下面详细说步骤。

1、将 classes.dex 拖入 IDA 中。

2、选择 Debugger->Debugger options->Set specific options，在 Package Name 和 Activity 中填入主 Activity 的包名和类名。

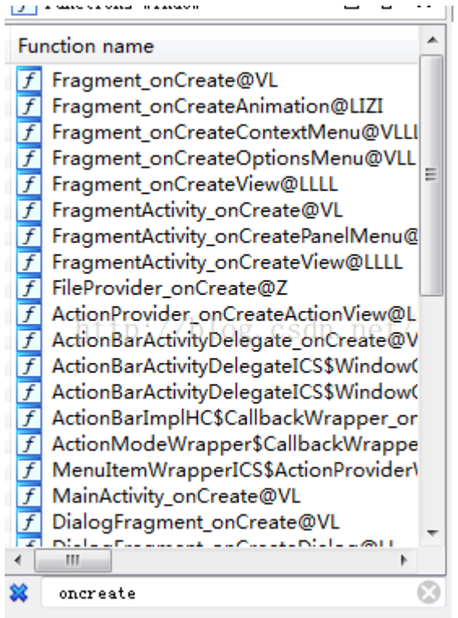


3、选择 Debugger->Process options，将 Port 改为 8700。



4、下断点

在左边的侧边栏搜索 onCreate。



点击 MainActivity_onCreate@VL，设置断点。

```
CODE:0007AD1C protected void com.example.simpleencryption.MainActivity.onCreate(
CODE:0007AD1C       android.os.Bundle savedInstanceState)
CODE:0007AD1C this = v3
CODE:0007AD1C savedInstanceState = v4
CODE:0007AD1C                   .prologue_end
CODE:0007AD1C                   .line 23
```

5、手机端开启 android_server

```
Youku@Youku-PC /cygdrive/d/视频书籍
$ adb shell
root@generic:/ # cd /data/local/tmp
cd /data/local/tmp
root@generic:/data/local/tmp # ./android_server
./android_server
IDA Android 32-bit remote debug server(ST) v1.17. Hex-Rays (c) 2004-2013
Listening on port #23946...
```

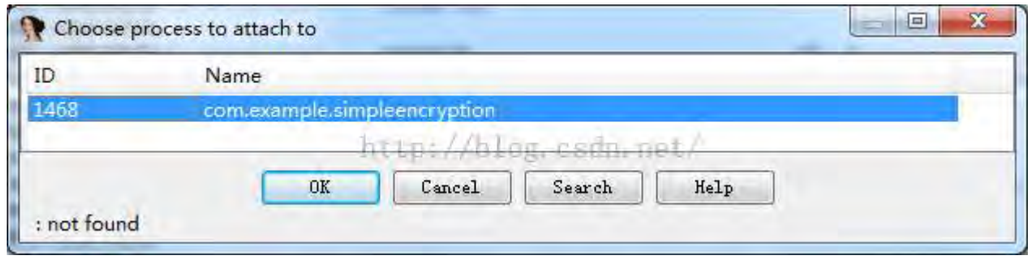
6、端口转发，打开要调试的 Activity

```
Youku@Youku-PC /cygdrive/d/视频书籍
$ adb forward tcp:23946 tcp:23946

Youku@Youku-PC /cygdrive/d/视频书籍
$ adb shell am start -D -n com.example.simpleencryption/.MainActivity
Starting: Intent { cmp=com.example.simpleencryption/.MainActivity }
```

此时应用的界面处于等阶段，如下：

注意此时在第 7 步前一定要关闭 eclipse，也就是关闭 ddms，否则 IDA 提示 can’t bind socket。

7、选择 Debugger->attach to process。出现如下界面：



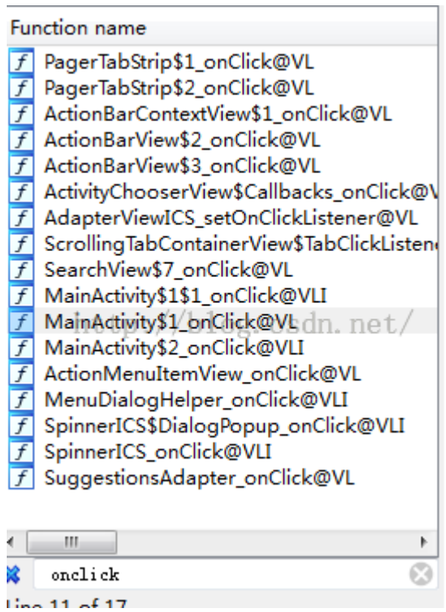
接下来，出现的界面就一直 Cancel 就好了。

我们看到程序停在的 MainActivity 的 onCreate 方法。按 F8 单步执行。

```
CODE:0007AD1C this = v3
CODE:0007AD1C savedInstanceState = v4
CODE:0007AD1C .prologue_end
CODE:0007AD1C .line 23
CODE:0007AD1C
CODE:0007AD1C invoke-super                {this, savedInstanceState}, <void Activity.onCreate(ref) imp. @ _de
CODE:0007AD22 .line 24
CODE:0007AD22
CODE:0007AD22 const/4                     v2, 1
CODE:0007AD24 invoke-virtual              {this, v2}, <boolean MainActivity.requestWindowFeature(int) imp. @
CODE:0007AD2A .line 25
CODE:0007AD2A
CODE:0007AD2A const                       v2, 0x7F030018
```


8、怎么在 onClick 方法下断点呢？

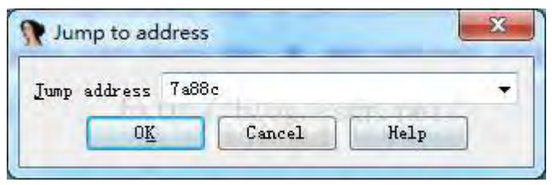
我们使用双开 IDA ，在静态分析中，左侧边栏搜索 onClick。



点击 MainActivity$1_onClick@VL，如下图：

```
CODE:0007A88C public void com.example.simpleencryption.MainActivity$1.onClick(
CODE:0007A88C       android.view.View v)
CODE:0007A88C this = v9
CODE:0007A88C v = v10
CODE:0007A88C                .prologue_end
CODE:0007A88C                .line 32
CODE:0007A88C
CODE:0007A88C                iget-object             v6, this, MainActivity$1_val$edit
CODE:0007A890                invoke-virtual          {v6}, <ref EditText.getText() imp. @ _def_EditText_getText@L>
CODE:0007A896                move-result-object      v6
CODE:0007A898                invoke-interface        {v6}, <ref Editable.toString() imp. @ _def_Editable_toString@L
CODE:0007A89E                move-result-object      v3
CODE:0007A8A0 .local name:'password' type:'Ljava/lang/String;'
```

所以在动态调试的 IDA 中，按 G，调到 7A88C 的地址。



然后下断点。

```
CODE:0007A88C public void com.example.simpleencryption.MainActivity$1.onClick(
CODE:0007A88C       android.view.View v)
CODE:0007A88C this = v9
CODE:0007A88C v = v10
CODE:0007A88C .prologue_end
CODE:0007A88C .line 32
```


然后点击 F8，让程序继续运行，刚才停在 onCreate 方法。

此时 IDA 界面如下：

应用程序界面如下：

此时输入密码，点击登录，程序会停留在刚才下的断点位置。

```
CODE:0007A88C public void com.example.simpleencryption.MainActivity$1.onClick(
CODE:0007A88C       android.view.View v)
CODE:0007A88C this = v9
CODE:0007A88C v = v10
CODE:0007A88C .prologue_end
CODE:0007A88C .line 32
CODE:0007A88C
CODE:0007A88C iget-object                   v6, this, MainActivity$1_val$edit
CODE:0007A890 invoke-virtual                {v6}, <ref EditText.getText() imp. @ _def_EditText_getText@L>
CODE:0007A896 move-result-object            v6
CODE:0007A898 invoke-interface              {v6}, <ref Editable.toString() imp. @ _def_Editable_toString@L>
CODE:0007A89E move-result-object            v3
CODE:0007A8A0 .local name:'password' type:'Ljava/lang/String;'
CODE:0007A8A0 password = v3
CODE:0007A8A0 .line 33
CODE:0007A8A0
```

此时按 F8，就可以单步调试 onClick 方法了。

和 IDA 动态调试 so 中少了一步 jdb -connect com.sun.jdi.SocketAttach:port=8700,hostname=localhost。

http://blog.csdn.net/jltxgcy/article/details/50600241

SO 调试

Android 逆向之动态调试总结

一、在 SO 中关键函数上下断点

刚学逆向调试时。大多都满足于在 SO 中某关键函数上下断点。然后通过操作应用程序，去触发这个断点，然后进行调试

详细的步骤可以参见非虫大大的《Android 软件安全与逆向分析》

简单说：在 libsyclover.so 文件中有一个函数 jnicall1。每次单击按钮的时候，便会调用此函数。

1.静态载入此 so 文件，找到函数的偏移地址为：0x132C

```
.text:0000132C                  EXPORT jnicall1
.text:0000132C jnicall1                                 ; DATA XREF: .data:00005058↓o
.text:0000132C
.text:0000132C var_14           = -0x14
.text:0000132C
.text:0000132C                  PUSH    {R0,R1,R4-R6,LR}
.text:0000132E                  LDR     R1, [R0]
.text:00001330                  MOVS    R3, #0x2A4
.text:00001334                  LDR     R3, [R1,R3]
.text:00001336                  MOVS    R1, R2
.text:00001338                  MOVS    R2, #0
.text:0000133A                  MOVS    R5, R0
```

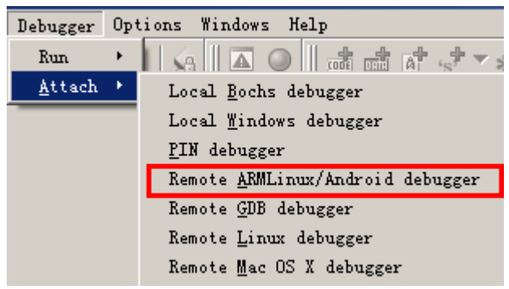
2.执行 android_server

3.端口转发

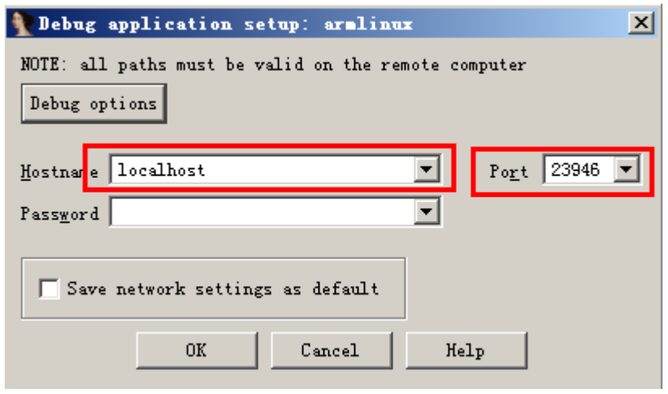
adb forward tcp:23946 tcp:23946

4.运行程序

5.IDA 附加



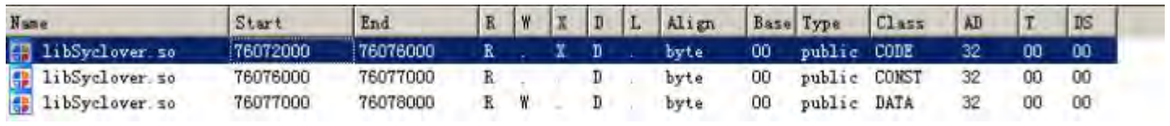
然后会弹出



点击 OK 之后，在弹出的列表框中选择需要附加的进程即可

6.下断点

附加完成之后，会停在 libc.so 这个模块中。此时按下 Ctrl + S，弹出模块列表框，搜索 so 文件名。

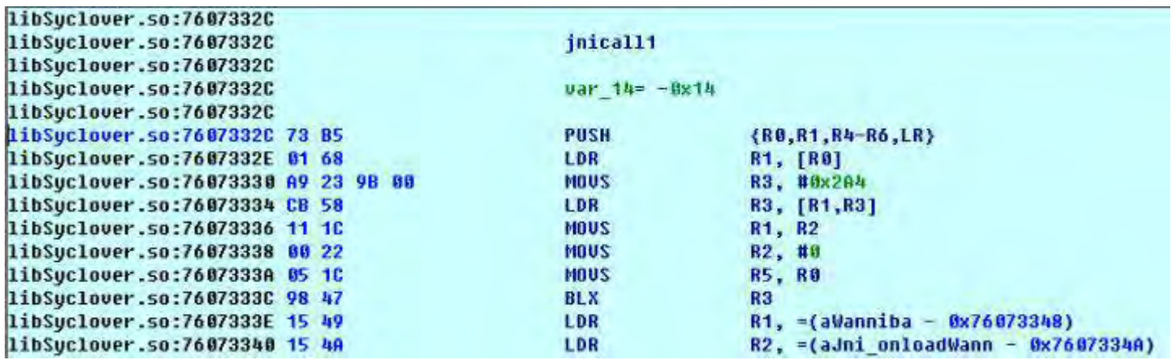


Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	T	DS
libSyclover.so	76072000	76076000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libSyclover.so	76076000	76077000	R	.	.	D	.	byte	00	public	CONST	32	00	00
libSyclover.so	76077000	76078000	R	W	.	D	.	byte	00	public	DATA	32	00	00

记录下基地址：0×76072000 （RX 权限）

和静态分析时得到的偏移地址 0x132C 相加得到 0x7607332C

G 跳转到此位置



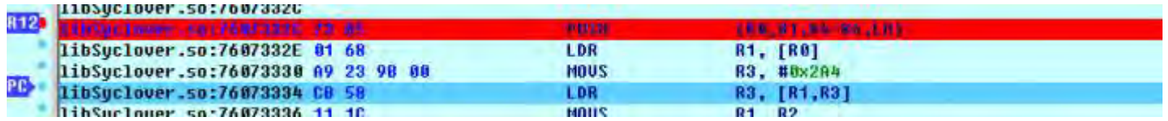
```
libSyclover.so:7607332C
libSyclover.so:7607332C                         jnicall1
libSyclover.so:7607332C
libSyclover.so:7607332C                         var_14= -0x14
libSyclover.so:7607332C
libSyclover.so:7607332C 73 B5                   PUSH            {R0,R1,R4-R6,LR}
libSyclover.so:7607332E 01 68                   LDR             R1, [R0]
libSyclover.so:76073330 A9 23 9B 00             MOVS            R3, #0x2A4
libSyclover.so:76073334 CB 58                   LDR             R3, [R1,R3]
libSyclover.so:76073336 11 1C                   MOVS            R1, R2
libSyclover.so:76073338 00 22                   MOVS            R2, #0
libSyclover.so:7607333A 05 1C                   MOVS            R5, R0
libSyclover.so:7607333C 98 47                   BLX             R3
libSyclover.so:7607333E 15 49                   LDR             R1, =(aWanniba - 0x76073348)
libSyclover.so:76073340 15 4A                   LDR             R2, =(aJni_onloadWann - 0x7607334A)
```

F2 下好断点！

7.触发断点

下好断点，便 F9 执行，此时状态是 runing

此时，去应用中单击按钮，程序便会断在刚刚下好的断点处~



ok~ 这种调试方法局限性很大，适合于比较初级的调试。这种调试手法在现在已经满足不了需求了。

二、在 JNI_OnLoad 函数上下断点

JNI_OnLoad 函数大概功能就是在程序加载 so 的时候，会执行 JNI_OnLoad 函数，做一系列的准备工作。
很多时候，程序猿们会将一些重要信息放在此函数中，而不是通过某种事件来重复触发。包括说将反调试函数放置在此函数中。因此，调试手段发生了改变，上述调试方法基本上被淘汰。

1.静态分析，找到 JNI_OnLoad 函数的偏移：0×1504

```
.text:00001504
.text:00001504                  EXPORT JNI_OnLoad
.text:00001504 JNI_OnLoad
.text:00001504
.text:00001504 var_30           = -0x30
.text:00001504 maxlen           = -0x24
.text:00001504 var_1C           = -0x1C
.text:00001504
.text:00001504                  PUSH    {R4-R7,LR}
.text:00001506                  SUB     SP, SP, #0x1C
.text:00001508                  MOVS    R3, #0
.text:0000150A                  STR     R3, [SP,#0x30+var_1C]
.text:0000150C                  LDR     R3, [R0]
.text:0000150E                  ADD     R1, SP, #0x30+var_1C
.text:00001510                  LDR     R2, =0x10004
.text:00001512                  LDR     R3, [R3,#0x18]
.text:00001514                  BLX     R3
.text:00001516                  STR     R0, [SP,#0x30+maxlen]
.text:00001518                  CMP     R0, #0
```

2.执行 android_server

3.端口转发

adb forward tcp:23946 tcp:23946

4.以调试模式启动程序

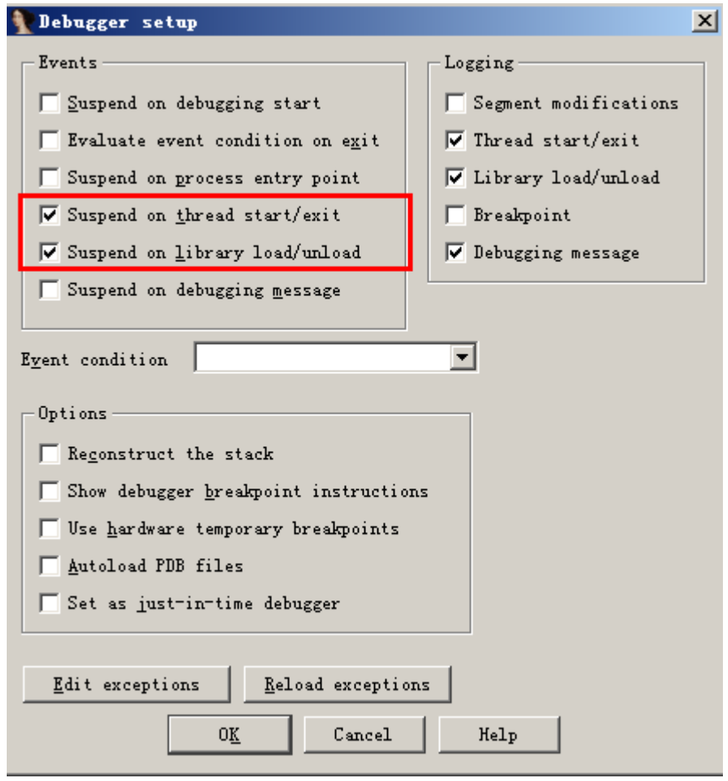
adb shell am start -D -n com.example.mytestcm/.MainActivity
此时，手机界面会出现 Waiting For Debugger 页面

5.打开 ddms 或者 Eclipse （必要，为了使用 jdb 命令）

6.IDA 附加

7.设置调试选项

Debugger — Debugger Options



8.F9 运行程序

IDA 中，F9 运行程序，此时是 runing 状态。

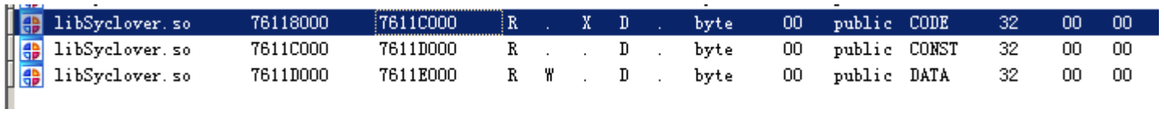
在命令行中执行：**jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700** 其中 port=8700 是从 ddms 中看到的。

此时程序会断下来

```
linker:40044118                         ; ---------------------------------------
linker:40044118                         CODE16
linker:40044118 65 48                   LDR             R0, =0xEBD6
linker:4004411A 78 44                   ADD             R0, PC
linker:4004411C 06 F0 28 EB             BLX             unk_4004A770
linker:40044120 01 20                   MOVS            R0, #1
linker:40044122 07 F1 14 07             ADD.W           R7, R7, #0x14
linker:40044126 BD 46                   MOV             SP, R7
linker:40044128 BD E8 F0 8F             POP.W           {R4-R11,PC}
```

9.下断点

Ctrl + S 然后搜索到 so 文件名



libSyclover.so	76118000	7611C000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libSyclover.so	7611C000	7611D000	R	.	.	D	.	byte	00	public	CONST	32	00	00
libSyclover.so	7611D000	7611E000	R	W	.	D	.	byte	00	public	DATA	32	00	00

记录下基地址是：0×76118000

加上 JNI_OnLoad 函数的偏移地址 0×1504 为 0×76119504

G 跳转到 0×76119504，下断点

```
libSyclover.so:76119504
libSyclover.so:76119504                         JNI_OnLoad
libSyclover.so:76119504
libSyclover.so:76119504                         var_30= -0x30
libSyclover.so:76119504                         var_24= -0x24
libSyclover.so:76119504                         var_1C= -0x1C
libSyclover.so:76119504
libSyclover.so:76119504 F0 B5                   PUSH            {R4-R7,LR}
libSyclover.so:76119506 87 B0                   SUB             SP, SP, #0x1C
libSyclover.so:76119508 00 23                   MOVS            R3, #0
libSyclover.so:7611950A 05 93                   STR             R3, [SP,#0x30+var_1C]
libSyclover.so:7611950C 03 68                   LDR             R3, [R0]
libSyclover.so:7611950E 05 A9                   ADD             R1, SP, #0x30+var_1C
libSyclover.so:76119510 4D 4A                   LDR             R2, =0x10004
libSyclover.so:76119512 9B 69                   LDR             R3, [R3,#0x18]
```

10.触发断点

下好断点之后，直接 F9 运行吧，就能断在 JNI_OnLoad 函数处~

```
libSyclover.so:76119504
libSyclover.so:76119504 F0 B5                   PUSH            {R4-R7,LR}
libSyclover.so:76119506 87 B0                   SUB             SP, SP, #0x1C
libSyclover.so:76119508 00 23                   MOVS            R3, #0
libSyclover.so:7611950A 05 93                   STR             R3, [SP,#0x30+var_1C]
```

当这种调试手法出现之后，将特殊函数，或者反调试函数放在 JNI_OnLoad 中也不是那么 的安全了。此时，程序猿们通过分析系统对 SO 文件的加载链接过程发现，JNI_OnLoad 函数并不是最开始执行的。在 JNI_OnLoad 函数执行之前，还会执行 init 段和 init_array 中的一系列函数。

因此，现在的调试方法，都是将断点下在 init_array 中~

至于下断点的方法，可以类比于在 JNI_OnLoad 中下断点的方法，在 init_array 的函数中下断点。还有一种方法便是通过在 linker 模块中，通过对其中函数下断点，然后也能单步到 init_array 中下面便详细介绍下如何给任意系统函数

下断点

三、给任意系统函数下断点

1.需要准备的有：

与你调试环境一致的系统源码，这个也可以在 http://androidxref.com/网站上在线查阅。
root 之后的手机，方便将系统的一些 so 文件 dump 至本地，静态获取到系统函数的偏移地址

2.流程

执行 android_server
端口转发 adb forward tcp:23946 tcp:23946
调试模式启动程序 adb shell am start -D -n 包名/类名
IDA 附加
静态找到目标函数对应所在模块的偏移地址
Ctrl+S 找到对应模块的基地址，两个地址相加得到最终地址
G 跳转至地址，然后下断
F9 运行
执行 jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700
断下，进行调试

四、在 dvmDexFileOpenPartial 函数下断点，dump 出明文 dex

发展至今，从去年到现在，apk 的加解密发展非常迅速。国内出现了很多针对 apk 的加壳保护方案。主要也体现在对 dex 的保护和对 so 的保护！
针对 dex 的保护，很长一段时间，都能通过对 dvmDexFileOpenPartial 函数下断点，从而 dump 出明文 dex 文件。
以这次 alictf 的第三题为例子，展示下如何对 dvmDexFileOpenPartial 函数下断点！
其他步骤都是一样的，这儿主要说下如何找到 dvmDexFileOpenPartial 函数位置

1.查看源码

dvmDexFileOpenPartial 函数在 rewriteDex 这个函数中被调用。

```
/*
 * Now that the DEX file can be read directly, create a DexFile struct
 * for it.
 */
if (dvmDexFileOpenPartial(addr, len, &pDvmDex) != 0) {
    ALOGE("Unable to create DexFile");
    goto bail;
}
```

可以看到关键字符串信息是：Unable to create DexFile

此时，从手机的/system/lib 目录下得到 libdvm.so

2. 载入 IDA，搜索字符串：Unable to create DexFile

```
text:0005AE8A loc_5AE8A                                 ; CODE XREF: sub_5AE64+1E↑j
text:0005AE8A                 MOV             R0, R9
text:0005AE8C                 MOV             R1, R7
text:0005AE8E                 ADD             R2, SP, #0x68+var_3C
text:0005AE90                 BL              _Z21dvmDexFileOpenPartialPKviPP6DvmDex ; dvmDexFileOpenPartial(void const*,int,DvmDex **
text:0005AE94                 CBZ             R0, loc_5AEA6
text:0005AE96                 LDR             R1, =(aDalvikvm_6 - 0x5AEA0)
text:0005AE98                 MOVS            R0, #6
text:0005AE9A                 LDR             R2, =(aUnableToCrea_1 - 0x5AEA2)
text:0005AE9C                 ADD             R1, PC  ; "dalvikvm"
text:0005AE9E                 ADD             R2, PC  ; "Unable to create DexFile"
text:0005AEA0                 BLX             __android_log_print
text:0005AEA4                 B               loc_5AE84
```

得到偏移地址是：0x0005AE8A

3.下断点

搜索模块 libdvm.so

libdvm.so	41492000	41539000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libdvm.so	41539000	4153D000	R	.	.	D	.	byte	00	public	CONST	32	00	00
libdvm.so	4153D000	41542000	R	W	.	D	.	byte	00	public	DATA	32	00	00
debug037	41542000	41544000	R	W	.	D	.	byte	00	public	DATA	32	00	00

基地址是 0×41492000
加上偏移地址为 0x414ECE8A
G 跳转至此位置，下好断点，即可

4.dump 明文 dex 文件

下好断点之后，F9 运行，执行 jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700
程序断下

```
libdvm.so:414ECE8C 39 46      MOV             R1, R7
libdvm.so:414ECE8E 0B AA      ADD             R2, SP, #0x2C
libdvm.so:414ECE90 E8 F7 78 FE BL              _Z21dvmDexFileOpenPartialPKviPP6DvmDex
```

此时，看到寄存器窗口中的值为：

```
R0  7624B008  [anon:libc_malloc]:7624B008
R1  000941FC
```

R0 保存 dex 的起始地址，R1 便是 dex 的长度

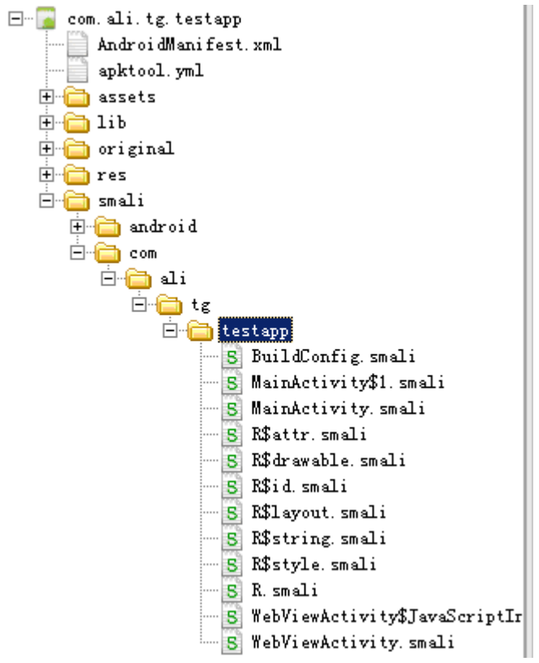
Hex View-1

```
7624B008  64 65 78 0A 30 33 35 00  6C 50 34 D8 82 04 15 79  dex.035.lP4....y
7624B018  EB 32 41 72 B7 D9 CE A3  6D 51 E6 41 06 56 BC 9D  .2Ar....mQ.A.V..
7624B028  FC 41 09 00 70 00 00 00  78 56 34 12 00 00 00 00  .A..p...xV4.....
7624B038  00 00 00 00 20 41 09 00  F0 18 00 00 70 00 00 00  .... A......p...
7624B048  02 04 00 00 30 64 00 00  2A 05 00 00 38 74 00 00  ....0d..*...8t..
7624B058  96 06 00 00 30 B2 00 00  08 19 00 00 E0 E6 00 00  ....0...........
7624B068  86 02 00 00 20 AF 01 00  1C 42 07 00 E0 FF 01 00  .... ....B......
7624B078  E0 FF 01 00 E2 FF 01 00  E5 FF 01 00 E9 FF 01 00  ................
7624B088  EF FF 01 00 F4 FF 01 00  0B 00 02 00 33 00 02 00  ............3...
7624B098  43 00 02 00 66 00 02 00  80 00 02 00 9D 00 02 00  C...f...■.......
7624B0A8  BE 00 02 00 D6 00 02 00  FD 00 02 00 25 01 02 00  ............%...
```

直接 dump 即可！

5.后续

dump 出来的 dex 就可以进行反编。
效果如下：



五、写在最后

随着现在技术的发展，对 apk 的保护是越来越好！大大增加了逆向分析人员的分析难度。同时，在整个攻防的过程中，对攻防两端的人都带来了非常棒体验。双方都取得了长足的进步！
也促使了整个加固方向水平的提升！
其中，动态调试手法在整个过程中是必不可少的。
http://www.52pojie.cn/thread-293648-1-1.html

安卓动态调试七种武器之孔雀翎 – Ida Pro

蒸米 · 2015/07/01 10:31

0x00 序

随着移动安全越来越火，各种调试工具也都层出不穷，但因为环境和需求的不同，并没有工具是万能的。另外工具是死的，人是活的，如果能搞懂工具的原理 再结合上自身的经验，你也可以创造出属于自己的调试武器。因此，笔者将会在这一系列文章中分享一些自己经常用或原创的调试工具以及手段，希望能对国内移动 安全的研究起到一些催化剂的作用。

目录如下:

0x01 孔雀翎

天下的暗器共有三百六十余种，但其中最成功、最可怕的就是孔雀翎。它使用简单，却威力无边。据说，孔雀翎发动之时，暗器四射，有如孔雀开屏，辉煌灿 烂，而就在敌人目眩神迷之际，便已魂飞魄散。这武器的描述与 Ida 是何其的相似啊！所以说安卓动态调试七种武器中的孔雀翎非 Ida 莫属。因为 Ida 太有名 了，相应的教程也是漫天飞，但很多并不是安卓相关的内容，所以笔者决定将一些经典的安卓调试技巧总结归纳一下。因为篇幅原因，笔者并不能保证本文能够覆盖 到 ida 调试的方方面面，看官如有兴趣可以再继续深入研究学习。

0x02 还原 JNI 函数方法名

在 android 调试中，你会经常见到这种类型的函数：

```
v5 = (*(int (__fastcall **)(int, int, _DWORD))(*(_DWORD *)v3 + 676))(v3, v4, 0);
```

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 v3 指针，我们选中后按一下”y”键，然后将类型声明为”JNIEnv*”。

随后 IDA 就会自动查找对应的方法并且显示出来了：

```
v5 = ((int (__fastcall *)(JNIEnv *, int, _DWORD))(*v3)->GetStringUTFChars)(v3, v4, 0);
```

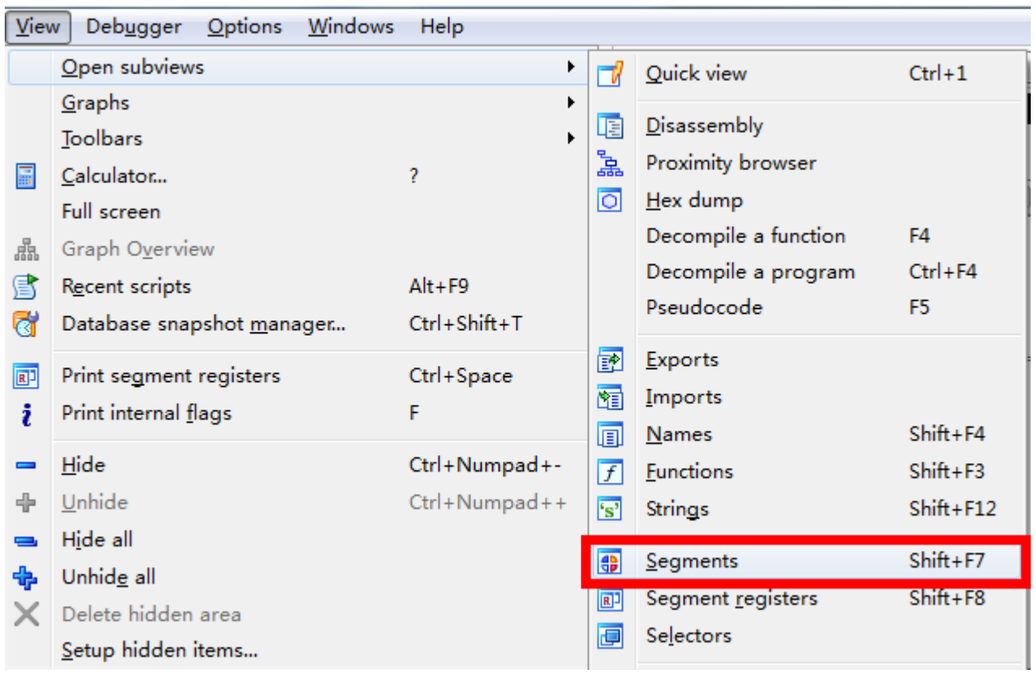
是不是瞬间清晰了很多？另外有人（ 貌似是看雪论坛上的）还总结了所有 JNIEnv 方法对应的数字，地址以及方法声明：

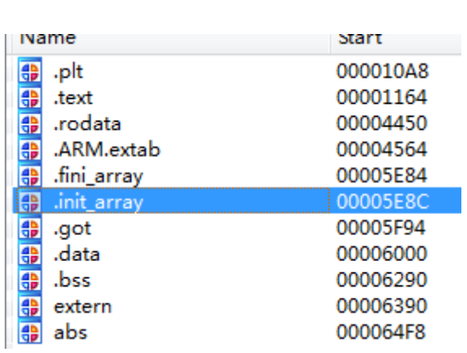
672	GetStringUTFLength	jsize (*)(JNIEnv*, jstring)
676	GetStringUTFChars	const char* (*)(JNIEnv*, jstring, jboolean*)
680	ReleaseStringUTFChars	void (*)(JNIEnv*, jstring, const char*)
684	GetArrayLength	jsize (*)(JNIEnv*, jarray)
688	NewObjectArray	jobjectArray (*)(JNIEnv*, jsize, jclass, jobject)

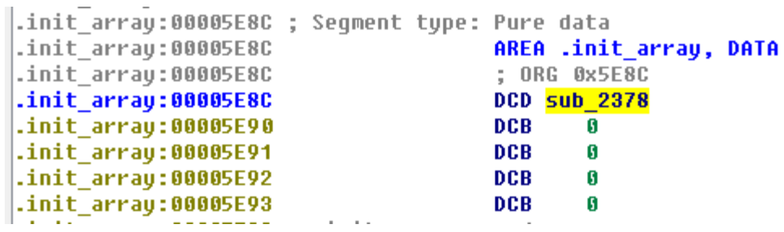
有兴趣的同学可以去我的 github 下载。

0x03 调试.init_array 和 JNI_OnLoad

我们知道 so 文件在被加载的时候会首先执行.init_array 中的函数，然后再执行 JNI_OnLoad()函数。JNI_Onload()函 数因为有符号表所以非常容易找到，但是.init_array 里的函数需要自己去找一下。首先打开 view ->Open subviews->Segments。然后点击.init.array 就可以看到.init_array 中的函数了。







但一般当我们使用 ida 进行 attach 的时候，.init_array 和 JNI_Onload()早已经执行完毕了，根本来不急调试。这时候我们 可以使用 jdb 这个工具来解决，这个工具是安装完 jdk 以后自带的，可以在 jdk 的 bin 目录下找到。在这里我们使用阿里移动安全挑战赛 2014 的第二题 作为例子讲解一下如何调试 JNI_OnLoad()。

打开程序后，界面是这样的：


7:46
自毁程序密码
当Bob带领银河飞行队赶到时，飞碟已坠落在小山谷里，驾驶员在坠落前启动了自毁程序，飞碟中的一切已化为灰烬，唯一幸免的是一部手机，但需要开机密码。
输入密码

我们的目标就是获取到密码。使用 ida 反编译一下 so 文件会看到我们输入后的密码会和 off_628c 这个指针指向的字符串进行比较。

```
v6 = off_628C;
while ( 1 )
{
  v7 = (unsigned __int8)*v6;
  if ( v7 != *(_BYTE *)v5 )
    break;
  ++v6;
  ++v5;
  v8 = 1;
  if ( !v7 )
    return v8;
}
return 0;
```

于是我们查看 off_628c 这个地址对应的指针，发现对应的字符串是”wojiushidaan”。

```
.data:0000628C off_628C        DCD aWojiushidaan

.rodata:00004450 aWojiushidaan   DCB "wojiushidaan",0
```

于是我们把这个密码输入一下，发现密码错误。看样子 so 文件在加载的时候对密码字符串进行了动态修改。既然动态修改了那我们用 ida 动态调试一下好 了，我们打开程序，然后再用 ida attach 一下，发现程序直接闪退了，ida 那边也没有任何有用信息。原来这就是自毁程序的意思啊。既然如此我们动态调试一下 JNI_OnLoad() 来看一下程序究竟做了什么吧。步骤如下：

1 ddms

一定要打开 ddms，否则调试端口是关闭的，就无法在程序刚开始的暂停了。我之前不知道要打开 ddms 才能用 jdb，还以为 android 系统或者 sdk 出问题了，重装好几次。汗。

2 adb push androidserver /data/local/tmp/

adb shell
su
chmod 777 /data/local/tmp/androidserver
/data/local/tmp/androidserver

这里我们把 ida 的 androidserver push 到手机上，并以 root 身份执行。

3 adb forward tcp:23946 tcp:23946

将 ida 的调试端口进行转发，这样 pc 端的 ida 才能连接手机。

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

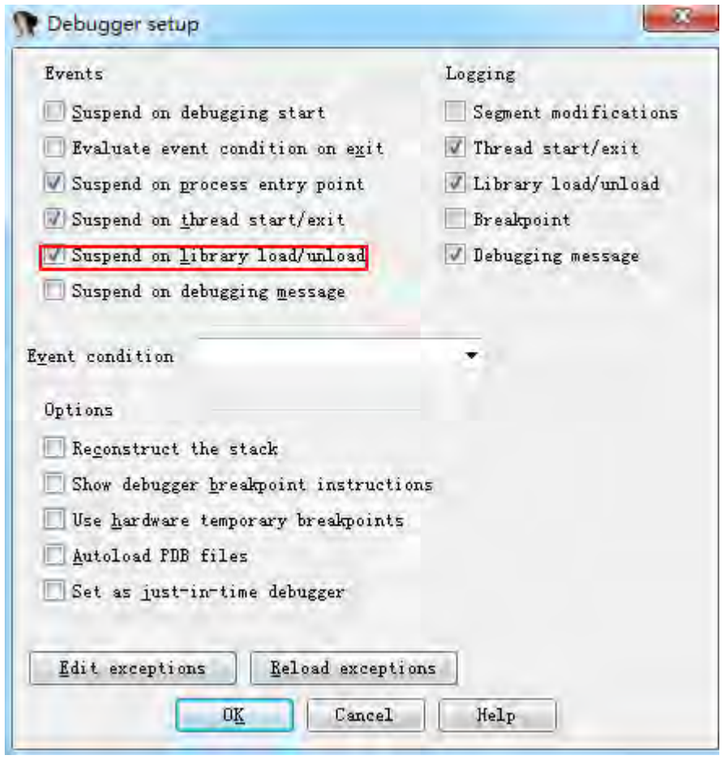
这里我们以 debug 模式启动程序。程序会出现 waiting for debugger 的调试界面。

5 ida attach target app

这时候我们启动 ida 并 attach 这个 app 的进程。

6 suspend on libary loading

我们在 debugger setup 里勾选 suspend on library load。然后点击继续。



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

用 jdb 将 app 恢复执行。

8 add breakpoint at JNI_OnLoad

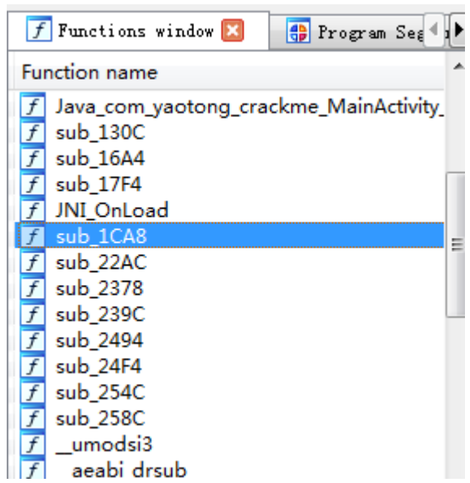
随后程序会在加载 libcrackme.so 这个 so 文件的时候停住。这时候 ida 会出现找不到文件的提示，不用管他，点取消即可。随后就能在 modules 中看到 libcrackme.so 这个 so 文件了，我们点进去，然后在 JNI_OnLoad 处下个断点，然后点击执行，程序就进入了 JNI_OnLoad()这个函数。

PS：有时候你明明在一个函数中却无法 F5，这时候你需要先按一下”p”键，程序会将这段代码作为函数分析，然后再按一下”F5”，你就能够看到反汇编的函数了。

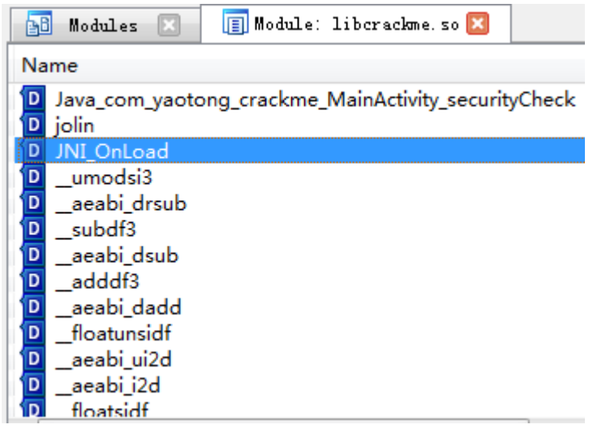
因为过程有点繁琐，我录制了一个调试 JNI_OnLoad()的视频在我的 github，有兴趣的同学可以去下载观看。因为涉及到其他的技巧，我们将会在随后的”ida 双开定位”章节中继续讲解如何调试.init_array 中的函数。

0x04 Ida 双开定位

Ida 双开定位的意思是先用 ida 静态分析 so 文件，然后再开一个 ida 动态调试 so 文件。因为在动态调试中 ida 并不会对整个动态加载的 so 文件进行详细的分析，所以很多函数并无法识别出来。比如静态分析中有很多的 sub_XXXX 函数：

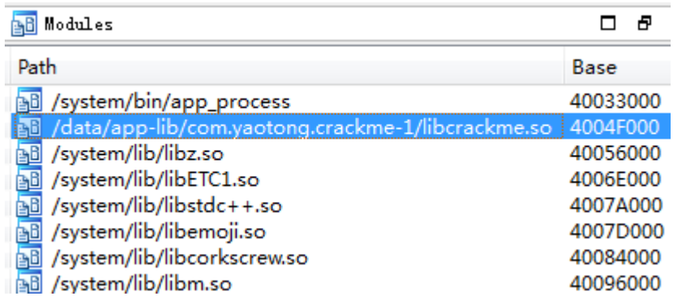


但动态调试中的 ida 是没有这些信息的。



所以我们需要双开 ida，然后通过 ida 静态分析的内容来定位 ida 动态调试的函数。当然很多时候我们也需要动态调试的信息来帮助理解静态分析的函数。

在上一节中，我们提到.init.array 中有个 sub_2378()，但当 ida 动态加载 so 后我们并无法在 module 中找到这个函数。那该 咋办呢？这时候我们就要通过静态分析的地址和 so 文件在内存中的基址来定位目标函数。首先我们看到 sub_2378()这个函数在静态分析中的地址 为.text:00002378。而在动态加载中这个 so 在内存中的基址为：4004F000。



因此 sub_2378()这个函数在内存中真正的地址应该为 4004F000 + 00002378 =40051378。下面我们在动态调试窗口输入”g”，跳转到 40051378 这个地址。然后发现全是乱码的节奏：

```
libcrackme.so:40051378                DCB    0
libcrackme.so:40051379                DCB 0x48 ; H
libcrackme.so:4005137A                DCB 0x2D ; -
libcrackme.so:4005137B                DCB 0xE9 ;
libcrackme.so:4005137C                DCB 0x10
libcrackme.so:4005137D                DCB    0
libcrackme.so:4005137E                DCB 0x9F ;
libcrackme.so:4005137F                DCB 0xE5 ;
libcrackme.so:40051380                DCB 0x10
libcrackme.so:40051381                DCB 0x10
libcrackme.so:40051382                DCB 0x9F ;
libcrackme.so:40051383                DCB 0xE5 ;
libcrackme.so:40051384                DCB    0
libcrackme.so:40051385                DCB    0
libcrackme.so:40051386                DCB 0x8F ;
libcrackme.so:40051387                DCB 0xE0 ;
libcrackme.so:40051388                DCB    0
libcrackme.so:40051389                DCB    0
libcrackme.so:4005138A                DCB 0x81 ;
libcrackme.so:4005138B                DCB 0xE0 ;
libcrackme.so:4005138C                DCB 0xC6 ;
libcrackme.so:4005138D                DCB 0xFF
libcrackme.so:4005138E                DCB 0xFF
libcrackme.so:4005138F                DCB 0xEB ;
```

不要担心，这是因为 ida 认为这里是数据段。这时候我们只要按”P”或者选中部分数据按”c”，ida 就会把这段数据当成汇编代码进行分析了：

```
libcrackme.so:40051378 ; =============== S U B R O U T I N E =======================================
libcrackme.so:40051378
libcrackme.so:40051378
libcrackme.so:40051378 sub_40051378
libcrackme.so:40051378                 STMFD           SP!, {R11,LR}
libcrackme.so:4005137C                 LDR             R0, =(unk_40054FBC - 0x4005138C)
libcrackme.so:40051380                 LDR             R1, =0xFFFFBCEC
libcrackme.so:40051384                 ADD             R0, PC, R0 ; unk_40054FBC
libcrackme.so:40051388                 ADD             R0, R1, R0
libcrackme.so:4005138C                 BL              unk_400512AC
libcrackme.so:40051390                 LDMFD           SP!, {R11,PC}
libcrackme.so:40051390 ; End of function sub_40051378
```

我们随后还可以按”F5”,将汇编代码反编译为 c 语言。

```
int sub_40051378()
{
  return ((int (__fastcall *)(_DWORD))unk_400512AC)(&unk_40050CA8);
}
```

是不是和静态分析中的 sub_2378()长的差不多？

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

我们随后可以在这个位置加入断点，再结合上一节提到的调试技巧就可以对 init.array 中的函数进行动态调试了。

我们接下来继续分析自毁程序这道题，当我们在对 init.array 和 JNI_OnLoad()进行调试的时候，发现程序在执行完 dowrd_400552B4()后就挂掉了。

```
  v8[-2] = 0;
  v5 = dword_400552B4(v8, 0, &unk_400506A4, 0);
  ((void (__fastcall *)(int))unk_400507F4)(v5);
  v6 = 65540;
```

于是我们在这里按”F7”进入函数看一下：

```
libc.so:40140A24 pthread_create
libc.so:40140A24                 STMFD           SP!, {R4-R11,LR}
libc.so:40140A28                 SUB             SP, SP, #0x14
```

原来是 libc.so 的 phread_create()函数，估计是 app 本身开了一个新的线程进行反调试检测了。

有意思的是在静态分析中我们并不清楚 dword_62B4 这个函数是做什么的，因为这个函数的地址在.bss 段还没有被初始化：

```
handle[-2] = 0;
dword_62B4(handle, 0, sub_16A4, 0);
sub_17F4();
```

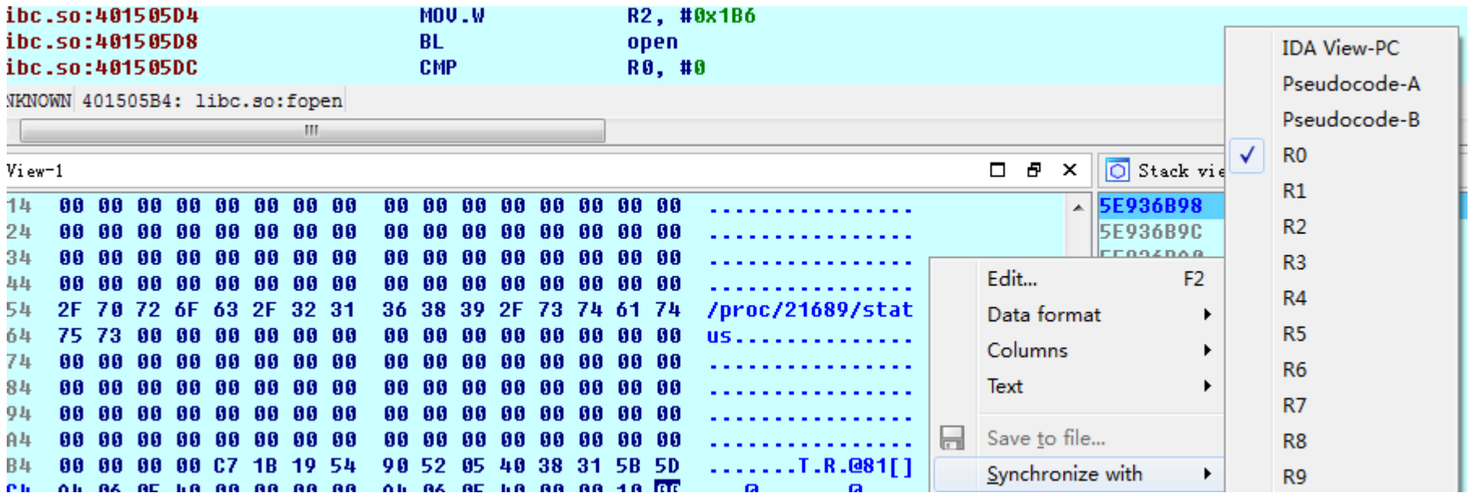
```
.bss:000062B4 ; int (__fastcall *dword_62B4)(_DWORD, _DWORD, _DWORD, _DWORD)
.bss:000062B4 dword_62B4      % 4                       ; DATA XREF: sub_16D4+28↑r
```

但是当我们动态调试的时候，这个地址的值已经修改为了 phread_create()这个函数的地址了。：

```
libcrackme.so:400552B4 ; int (__fastcall *dword_400552B4)(_DWORD, _DWORD, _DWORD, _DWORD
libcrackme.so:400552B4 dword_400552B4  DCD 0x40140A24
```

```
libc.so:40140A24 pthread_create
libc.so:40140A24                 STMFD           SP!, {R4-R11,LR}
libc.so:40140A28                 SUB             SP, SP, #0x14
```

所以说自毁程序密码这个 app 会用 pthread_create()开一个新的线程对 app 进行反调试检测。线程会运行 sub_16A4()这个函 数。于是我们对这个函数进行分析，发现里面的内容有大量的混淆，看起来十分吃力。这里我介绍个小 trick：常见的反调试方法都会用 fopen 打开一些文 件来检测自己的进程是否被 attach，比如说 status 这个文件中的 tracerpid 的值是否为 0，如果为 0 说明没有别的进程在调试这个进程，如果 不为 0 说明有程序在调试。所以我们可以守株待兔，在 libc.so 中的 fopen()处下一个断点，然后我们在 hex view 窗口中设置数据与 R0 的值同步：



这样的话，当函数在 fopen 处停住的时候我们就能看到程序打开了哪些文件。果不其然，程序打开了/proc/[pid]/status 这个文件。 我们”F8”继续执行 fopen 函数，看看返回后的地址在哪。然后发现我们程序卡在了某个函数中间，PC 上面都是数据，PC 下面才是汇编。这该咋办呢？

```
libcrackme.so:40050414                DCB 0xAF
libcrackme.so:40050415                DCB  0xF
libcrackme.so:40050416                DCB 0x8D ;
libcrackme.so:40050417                DCB 0xE2 ;
libcrackme.so:40050418                DCB 4
libcrackme.so:40050419                DCB 0x10
libcrackme.so:4005041A                DCB 0xA0 ;
libcrackme.so:4005041B                DCB 0xE1 ;
libcrackme.so:4005041C                DCB 0x35
libcrackme.so:4005041D                DCB 0xFF
libcrackme.so:4005041E                DCB 0x2F ; /
libcrackme.so:4005041F                DCB 0xE1 ;
libcrackme.so:40050420 ; ---------------------------------------------------
libcrackme.so:40050420                ADD             R6, SP, #0xBC
libcrackme.so:40050424                MOV             R5, R0
libcrackme.so:40050428                MOV             R1, #0x200
libcrackme.so:4005042C                MOV             R2, #0
libcrackme.so:40050430                MOV             R0, R6
libcrackme.so:40050434                STR             R5, [SP,#0x34]
libcrackme.so:40050438                BL              unk_400500E0
libcrackme.so:4005043C                LDR             R3, [R4,#0x10]
```

解决办法还是 ida 双开，我们知道现在 PC 的地址为 40050420，libcrackme.so 文件的基址为 4004F000。因此这段代码在 so 中的位置应该是：40050420 - 4004F000 = 1420。因此我们回到 ida 静态分析界面，就可以定位到我们其实是在 sub_130C()这个函数中。于是我们猜测这个函数就是用来做反调试检测的。

```
.text:00001408                STR     R8, [SP,#0x348+var_338]
.text:0000140C                ADD     R0, PC, R0 ; _GLOBAL_OFFSET_TABLE_
.text:00001410                ADD     R4, R7, R0
.text:00001414                ADD     R0, SP, #0x348+var_8C
.text:00001418                MOV     R1, R4
.text:0000141C                BLX     R5
.text:00001420                ADD     R6, SP, #0x348+var_28C
.text:00001424                MOV     R5, R0
.text:00001428                MOV     R1, #0x200
.text:0000142C                MOV     R2, #0
.text:00001430                MOV     R0, R6
.text:00001434                STR     R5, [SP,#0x348+var_314]
```

因此我们可以通过基址来定位 sub_130C()这个函数在内存中的地址：40050420 + 130C = 4005030C。然后我们在 4005030C 这个地址处按”P”， ida 就可以正确的识别整个函数了。

```
libcrackme.so:4005030C ; ---------------------------------------------------------------------------
libcrackme.so:4005030C                 STMFD           SP!, {R4-R11,LR}
libcrackme.so:40050310                 SUB             SP, SP, #0x324
libcrackme.so:40050314                 LDR             R0, =(unk_40054FBC - 0x40050328)
libcrackme.so:40050318                 MOV             R1, #0x64
libcrackme.so:4005031C                 MOV             R2, #0
libcrackme.so:40050320                 ADD             R4, PC, R0 ; unk_40054FBC
libcrackme.so:40050324                 LDR             R0, =0xFFFFFFD8
libcrackme.so:40050328                 LDR             R0, [R0,R4]
libcrackme.so:4005032C                 LDR             R0, [R0]
libcrackme.so:40050330                 STR             R0, [SP,#0x320]
libcrackme.so:40050334                 ADD             R0, SP, #0x2BC
libcrackme.so:40050338                 BL              unk_400500E0
libcrackme.so:4005033C                 LDR             R7, =0x2D4
```

所以说动态调试的时候可以帮我们了解到很多静态分析很难获取到的信息。这也就是 ida 双开的意义所在：静态帮助动态定位函数地址，动态帮助静态获取运行时信息。

0x05 Ida 动态修改内存数据和寄存器数值

我们继续分析自毁程序密码这个 app，我们发现该程序会用 fopen ()打开/proc/[pid]/status 这个文件，随后会用 fgets()和 strstr()来获取，于是我们在 strstr()处下个断点，然后 让 hex view 的数据与 R0 同步。每次点击继续，我们都会看到 strstr 传入的参数。当传入的参数变为 TracerPid:XXXX 的时候我们停一下。因为在 正常情况下，TracerPid 的值应该是 0。但是当被调试的时候就会变成调试器的 pid。

```
R5
PC  libc.so:4015821C strstr
    libc.so:4015821C                 PUSH.W          {R4-R8,LR}
    libc.so:40158220                 MOV             R4, R0
```

```
54 72 61 63 65 72 50 69  64 3A 09 32 34 38 35 34  TracerPid:.24854
```

为了防止程序发现我们在调试，在这里我们需要把值改回 0。我们在 hex view 的 2 那里点击右键，然后选择 edit。随后我们输入 30 和 00，再点击”apply changes”。就可以把 TracerPid 改为 0 了。然后就可以 bypass 这一次的反调试的检测。

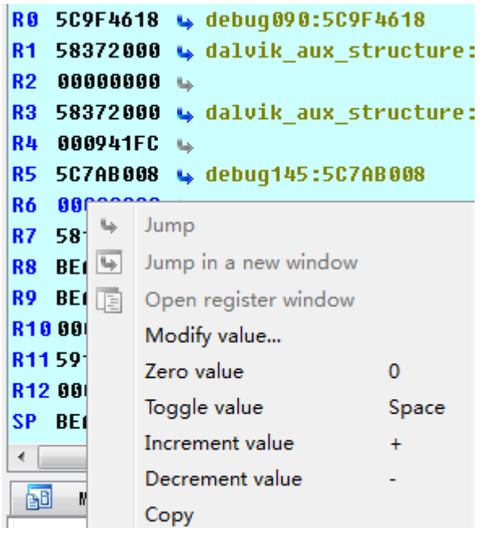
```
5EA36C54  54 72 61 63 65 72 50 69  64 3A 09 30 00 38 35 34  TracerPid:.0.854
5EA36C64  0A 00 29 0A 00 0A 00 00  00 00 00 00 00 00 00 00  ..).............
```

但这个程序检测 TracerPid 的次数非常频繁，我们要不断的修改 TracerPid 的值才行，这种方法实在有点治标不治本，所以我们会在下一节介绍 patch so 文件的方法来解决这个问题。

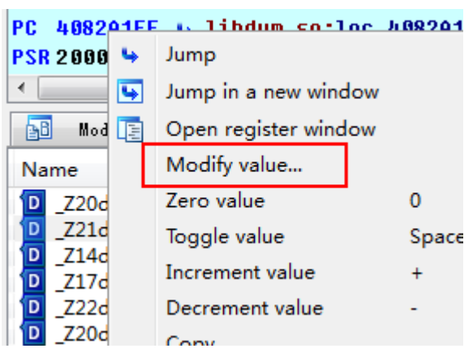
另外在 ida 动态调试过程中，除了内存中的数据可以修改，寄存器的数据也是可以动态修改的。比如说程序执行到 CMP R6, #0。本来 R6 的值是 0，经过比较后，程序会跳转到 4082A3FC 这个地址。

```
libdvm.so:4082A1F8                 CMP             R6, #0
libdvm.so:4082A1FA                 BEQ.W           loc_4082A3FC
```

但是如果我们在 PC 执行到 4082A1F8 这条语句的时候，将 R6 的值动态修改为 0。程序就不会进行跳转了。



你甚至可以修改 PC 寄存器的值来控制程序跳转到任何想要跳转到的位置，简直和 ROP 的原理一样。但记得要注意栈平衡等问题。

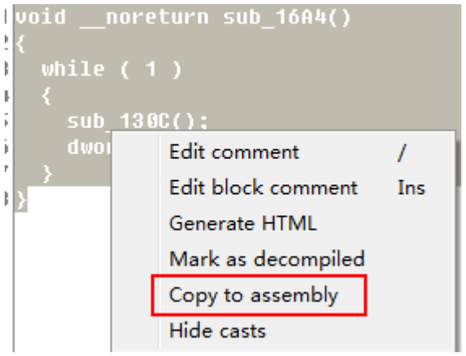


0x06 Patch so 文件

在上文中，我们通过分析定位到 sub_130C()这个函数有很大可能性是用来做反调试检测的，并且作者开了一个新的线程，并且用了一个 while 来不断执行 sub_130C()这个函数，所以说我们每次手动的修改 TracerPid 实在是不现实。

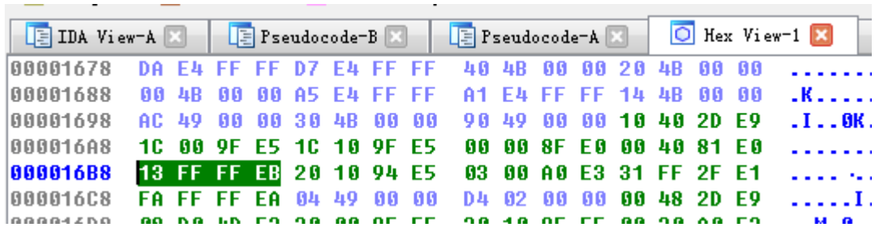
```
void __noreturn sub_16A4()
{
  while ( 1 )
  {
    sub_130C();
    dword_62B0(3);
  }
}
```

既然如此我们何不把 sub_130C()这个函数给 nop 掉呢？为了防止 nop 出错，我们先在”F5”界面选择所有代码，然后用”Copy to assembly”功能，就可以把 c 语言代码注释到汇编代码里。

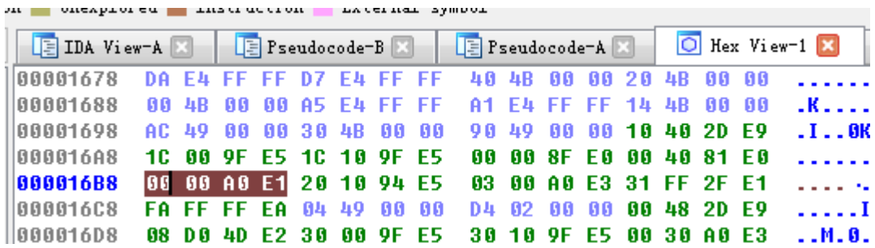



```
.text:000016A4 sub_16A4                                ; DATA XREF: JNI_OnLoad+A8↓o
.text:000016A4                                         ; .text:off_1CA4↓o
.text:000016A4                 STMFD   SP!, {R4,LR}
.text:000016A8                 LDR     R0, =(_GLOBAL_OFFSET_TABLE_ - 0x16B8)
.text:000016AC                 LDR     R1, =(unk_6290 - 0x5FBC)
.text:000016B0                 ADD     R0, PC, R0 ; _GLOBAL_OFFSET_TABLE_
.text:000016B4                 ADD     R4, R1, R0 ; unk_6290
.text:000016B8 ; 2:   while ( 1 )
.text:000016B8 ; 4:     sub_130C();
.text:000016B8
.text:000016B8 loc_16B8                                ; CODE XREF: sub_16A4+24↓j
.text:000016B8                 BL      sub_130C
.text:000016BC ; 5:     dword_62B0(3);
.text:000016BC                 LDR     R1, [R4,#(dword_62B0 - 0x6290)]
.text:000016C0                 MOV     R0, #3
.text:000016C4                 BLX     R1
.text:000016C8                 B       loc_16B8
.text:000016C8 ; End of function sub_16A4
```

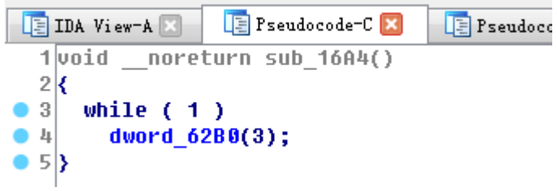
在这里我们看到如果想要注释掉 sub_130C()函数，只需要注释掉 000016B8 这个位置上的代码即可，如果我们想要注释掉 dword_62B0(3)这个函数，我们则需要注释掉 000016BC-000016C4 这三个位置上的代码。接下来我们选中 000016B8 这一行， 然后再点击 HexView。HexView 会帮我们自动定位到 000016B8 这个位置。



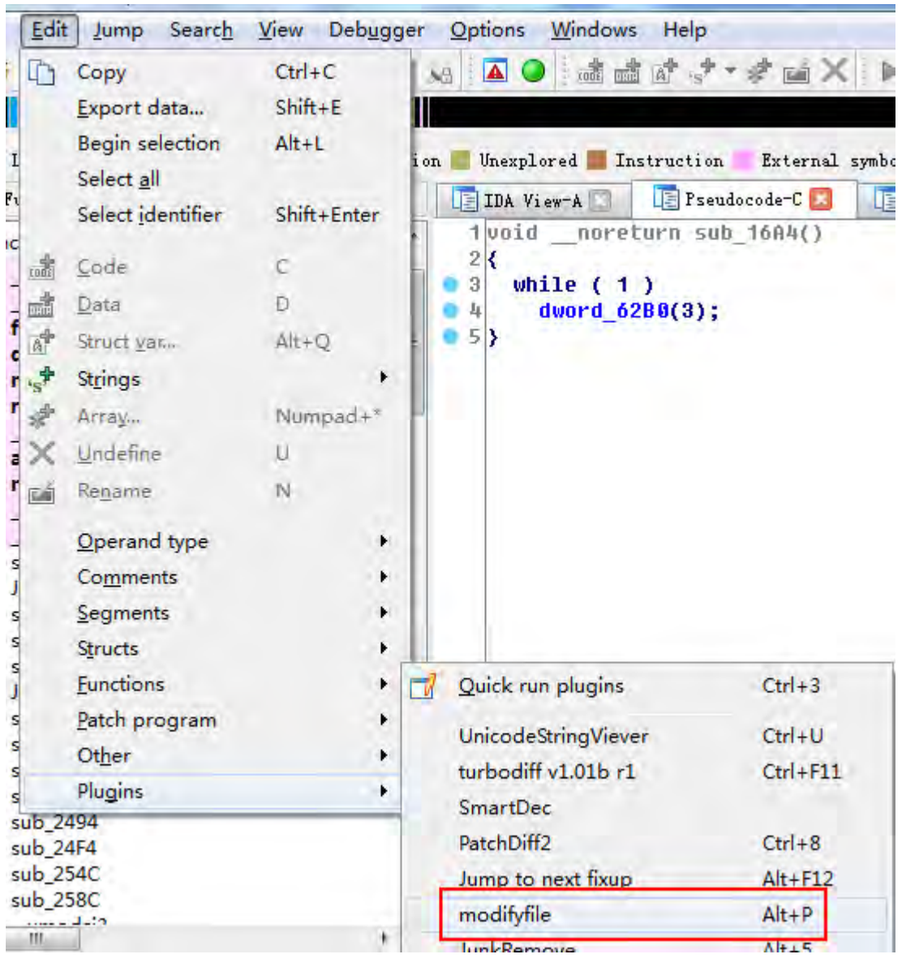
因为 ARM 是没有单独的 NOP 指令的。于是我们采用 movs r0,r0 作为 NOP。对应的机器码为”00 00 A0 E1”。所以我们把”13 FF FF EB”这段内容修改为”00 00 A0 E1”。



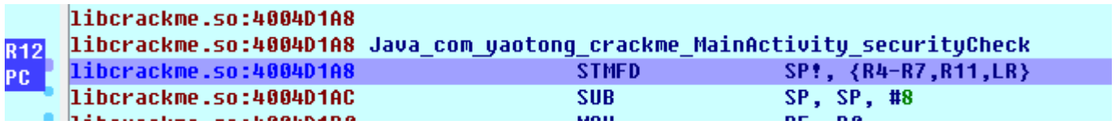
我们再回”F5”界面，就会发现 sub_130C()函数已经没有了。



最后我们点击”Edit->Plugins->modifyfile”，然后就可以保存新的 so 文件了。我们将这个 so 文件覆盖原 apk 中的 so 文件，然后再重新签名。



这次我们先运行程序，再用 ida 加载，app 并没有闪退，说明我们 patch 成功了。于是我们先 在”Java_com_yaotong_crackme_MainActivity_securityCheck”处下断点。然后在 app 随便输入一个密码，点击 app 上的”输入密码”按钮。



程序就会暂停在”Java_com_yaotong_crackme_MainActivity_securityCheck”处。我们先按”P”再按”F5”，就可以看到反汇编的 c 语言了。而这里的 unk_4005228C 就是保存了密码字符串指针的指针。

```
31  v5 = (*(int (__fastcall **)(int
32  v6 = unk_4005228C;
33  while ( 1 )
34  {
35    v7 = *(_BYTE *)v6;
36    if ( v7 != *(_BYTE *)v5 )
37      break;
38    ++v6;
39    ++v5;
40    v8 = 1;
41    if ( !v7 )
42      return v8;
```

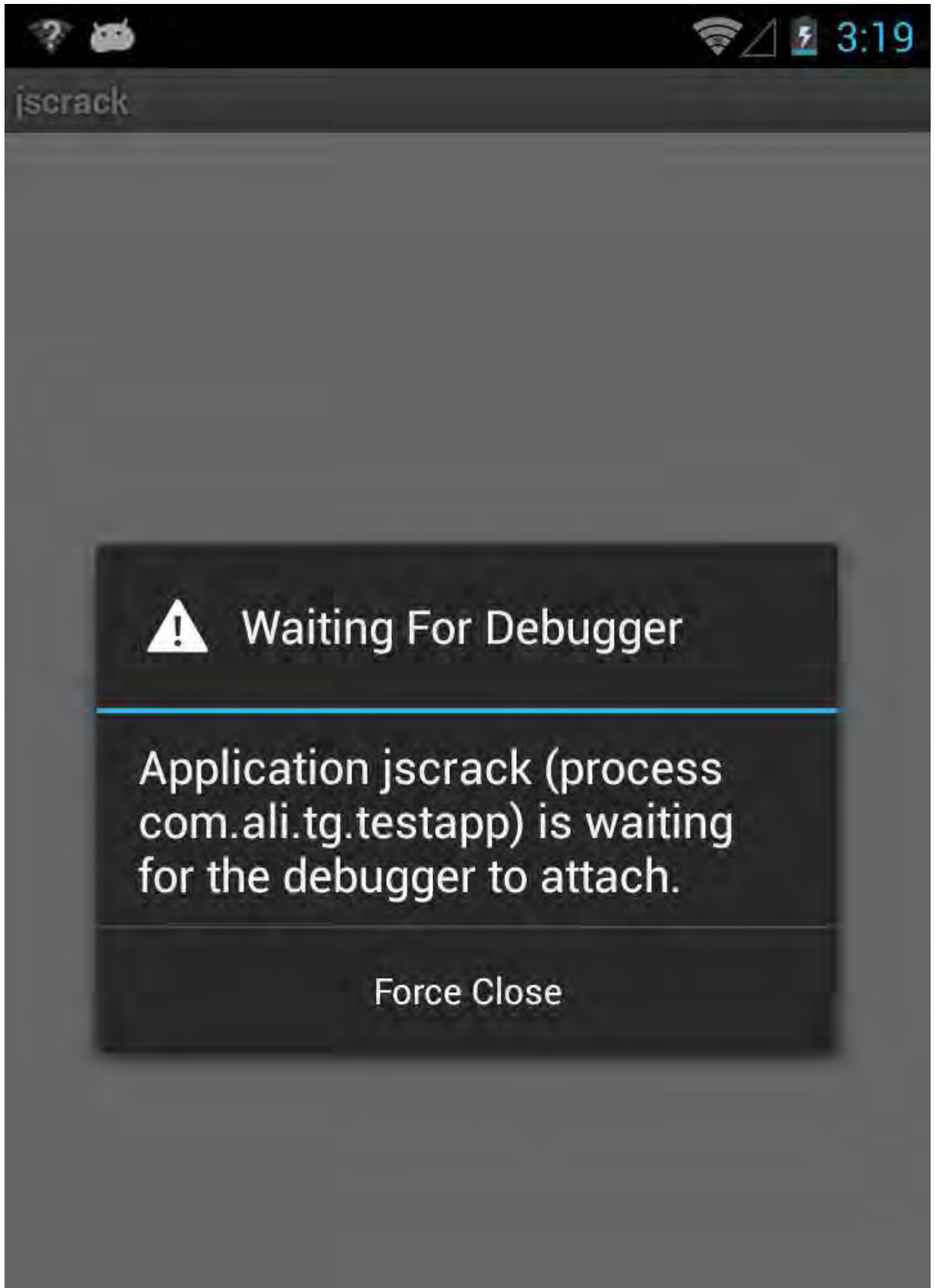
因为是指针的指针，所以我们先双击进入这个地址。

```
libcrackme.so:4005228C unk_4005228C    DCB 0x50 ; P
libcrackme.so:4005228D                 DCB    4
libcrackme.so:4005228E                 DCB    5
libcrackme.so:4005228F                 DCB 0x40 ; @
libcrackme.so:40052290                 DCB    0
libcrackme.so:40052291                 DCB    0
```

然后在这个地址上按三下”D”，将这里的数据格式从字符转化为指针形式。

```
libcrackme.so:4005228B                 DCB 0x56 ; V
libcrackme.so:4005228C dword_4005228C  DCD 0x40050450
libcrackme.so:40052290                 DCB    0
```

然后我们再双击进入这个地址，就可以看到最后的 flag 了。答案是”aiyou,bucuoo”。


3:19
jscrack
Waiting For Debugger
Application jscrack (process com.ali.tg.testapp) is waiting for the debugger to attach.
Force Close

这道题里我们只是用到了很简单的 patch so 技巧，在实战中我们不光可以 NOP，我们还可以改变条件判断语句，比如将”BNE”变为” BEQ”。我们甚至可以修改跳转地址，比如直接让程序 B 到某个地址去执行，这样的话就不需要挨个的 NOP 很多语句了。要注意的是，ARM 中的跳转指令是根 据相对地址计算的，所以你要根据当前指令地址和目标地址来计算出相对跳转的值。

比如说 00001BCC: BEQ loc_1C28 对应的汇编代码为”15 00 00 0A”。

```
.text:00001BCC                  BEQ     loc_1C28

00001BCC  15 00 00 0A
```

0x0A 代表 BEQ，”15 00 00”代表跳转的相对地址，因为在 arm 中 pc 的值是当前指令的下两条（下一条的下一条）指令的地址，所以我们需要将 0x15 再加上 2。随后就可以计算出 最后跳转到的地址： (0x15 + 0x2)*4 + 0x1BCC = 0x1C28。Ida 反汇编后的结果也验证了结果是 BEQ loc_1C28。

接下来我们想修改汇编代码为 00001BCC: BNE loc_1C2C。只需要将”0A”变成”1A”，将”15”变成”16”即可。

```
.text:00001BCC                  BNE     loc_1C2C

00001BCC  16 00 00 1A
```

0x0A 代表 BEQ，”15 00 00”代表跳转的相对地址，因为在 arm 中 pc 的值是当前指令的下两条（下一条的下一条）指令的地址，所以我们需要将 0x15 再加上 2。随后就可以计算出 最后跳转到的地址： (0x15 + 0x2)*4 + 0x1BCC = 0x1C28。Ida 反汇编后的结果也验证了结果是 BEQ loc_1C28。

接下来我们想修改汇编代码为 00001BCC: BNE loc_1C2C。只需要将”0A”变成”1A”，将”15”变成”16”即可。

0x07 Kill 调试技巧

该技巧是 QEver 在《MSC 的伪解题报告》中提到的。利用 kill 我们可以让程序挂起，然后用 ida 挂载上去，获取有用的信息，然后可以再用 kill 将程序恢复运行。我们还是拿自毁程序密码这个应用举例，具体实行方法如下：

1 首先用 ps 获取运行的 app 的 pid。

```
shell     1208  1204  1936   1368  c015fa58 40146c0c S logcat
u0_a56    1223  129   502832 56368 ffffffff 40076ee4 S com.yaotong.crackme
```

2 然后用 kill -19 [pid] 就可以将这个 app 挂起了。

```
shell@android:/ # kill -19 1223
kill -19 1223
```

3 随后我们用 ida attach 上这个 app。因为整个进程都挂起了，所以这次 ida 挂载后 app 并没有闪退。然后就可以在内存中找到答案了。

```
592AE450  61 69 79 6F 75 2C 62 75  63 75 6F 6F 00 4C 37 39  aiyou,bucuoo.L
```

4 如果想要恢复 app 的运行，需要将 ida 退出，然后再使用 kill -18 [pid]即可。

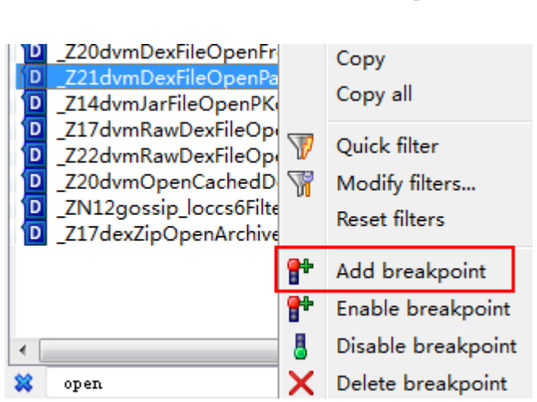
```
shell@android:/ # kill -18 1223
kill -18 1223
```

0x08 在内存中 dump Dex 文件

在现在的移动安全环境中，程序加壳已经成为家常便饭了，如果不会脱壳简直没法在破解界混的节奏。ZJDroid 作为一种万能脱壳器是非常好用的，但 是当作者公开发布这个项目后就遭到了各种加壳器的针对，比如说抢占 ZJDroid 的广播接收器让 ZJDroid 无法接收命令等。我们也会在”安卓动态调试 七种武器之多情环 - Customized DVM”这篇文章中介绍另一种架构的万能脱壳器。但工具就是工具，当我们发布的时候可能也会遭到类似 ZJDroid 那样的针对。所以说手动脱壳这项技能还 是需要学习的。在这一节中我们会介绍一下最基本的内存 dump 流程。在随后的文章中我们会介绍更多的技巧。

这里我们拿 alictf2014 中的 apk300 作为例子来介绍一下 ida 脱简单壳的基本流程。 首先我们用调试 JNI_OnLoad 的技巧将程序在运行前挂起：

adb shell am start -D -n com.ali.tg.testapp/.MainActivity



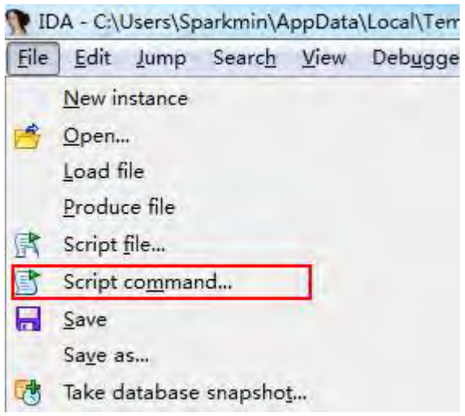
然后在 libdvm.so 中的 dvmDexFileOpenPartial 函数上下一个断点：

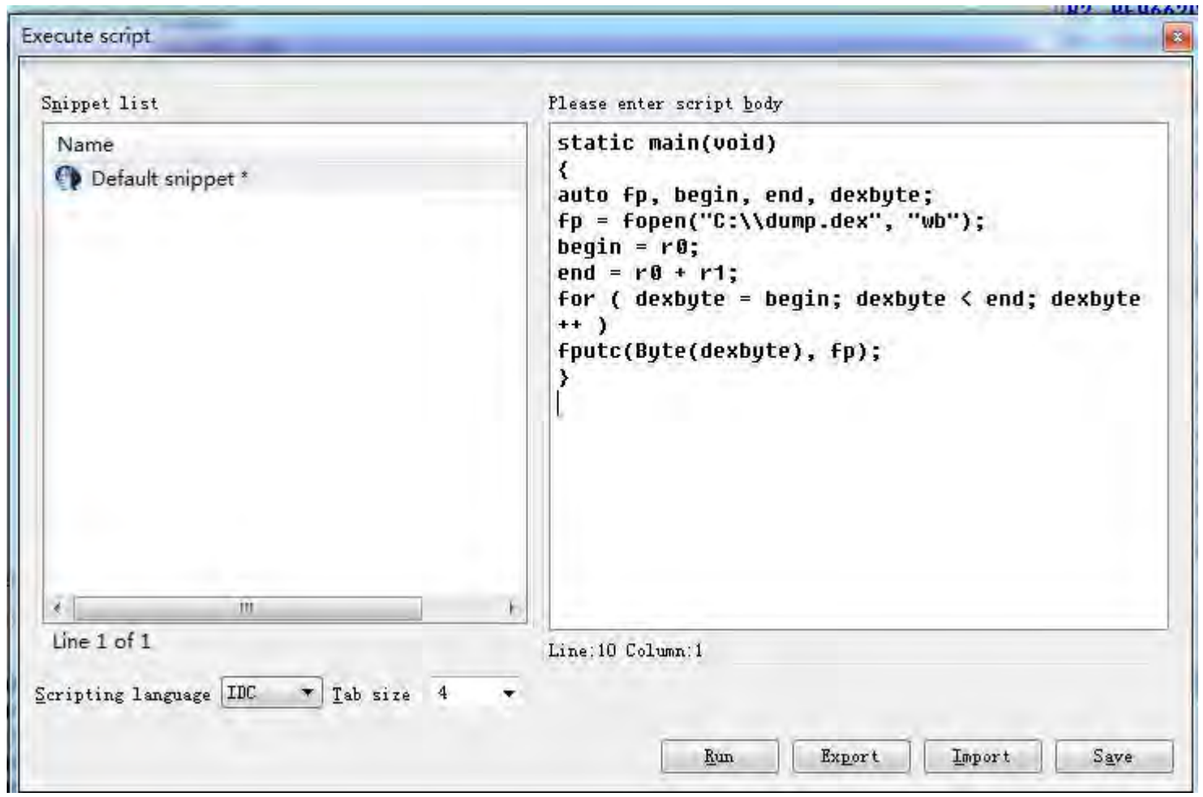
然后我们点击继续运行，程序就会在 dvmDexFileOpenPartial()这个函数处暂停，R0 寄存器指向的地址就是 dex 文件在内存中的地址，R1 寄存器就是 dex 文件的大小：

```
R0  5C137008 ↳ debug158:5C137008
R1  000941FC ↳
```

```
5C137008  64 65 78 0A 30 33 35 00  6C 50 34 D8 82 04 15 79  dex.035.lP4....y
5C137018  EB 32 41 72 B7 D9 CE A3  6D 51 E6 41 06 56 BC 9D  .2Ar....mQ.A.V..
5C137028  FC 41 09 00 70 00 00 00  78 56 34 12 00 00 00 00  .A..p...xV4.....
5C137038  00 00 00 00 20 41 09 00  F0 18 00 00 70 00 00 00  .... A......p...
5C137048  02 04 00 00 30 64 00 00  2A 05 00 00 38 74 00 00  ....0d..*...8t..
5C137058  96 06 00 00 30 B2 00 00  08 19 00 00 E0 E6 00 00  ....0...........
5C137068  86 02 00 00 20 AF 01 00  1C 42 07 00 E0 FF 01 00  .... ....B......
```

然后我们就可以使用 ida 的 script command 去 dump 内存中的 dex 文件了。






```
1static main(void)
2{
3    auto fp, begin, end, dexbyte;
4    fp = fopen("C:\\dump.dex", "wb");
5    begin = r0;
6    end = r0 + r1;
7    for ( dexbyte = begin; dexbyte < end; dexbyte ++ )
8            fputc(Byte(dexbyte), fp);
9}
```

Dump 完 dex 文件后，我们就可以用 baksmali 来反编译这个 dex 文件了。

```
c:\7weapons>java -jar baksmali-2.0.3.jar dump.dex
```

因为过程有点繁琐，我录制了一个 dump dex 文件的视频在我的 github，有兴趣的同学可以去下载观看。

当然这只是最简单脱壳方法，很多高级壳会动态修改 dex 的结构体，比如将 codeoffset 指向内存中的其他地址，这样的话你 dump 出来的 dex 文件其实是不完整的，因为代码段保存在了内存中的其他位置。但你不用担心，我们会在随后的文章中介绍一种非常简单的解决方案，敬请期待。

0x09 Function Rewrite 函数重写

有时我们想要将 app 中的某个函数的逻辑提取出来，用 gcc 重新编译一个可执行文件，比如我们想要写一个注册机，就需要把 app 生成 key 的逻辑提 取出来。但是 ida ”F5”过后的 c 语言直接编译经常会有很多错误，比如未定义的宏，未定义的声明等。这是因为这些宏都在 ida 的一个头文件里。里面定义了所有 ida 自定义 的宏和声明，比如说经常见到的 BYTEn()宏：

```
1#define BYTEn(x, n)     (*((_BYTE*)&(x)+n))
2#define BYTE1(x)     BYTEn(x,    1)                  // byte 1 (counting from 0)
3#define BYTE2(x)     BYTEn(x,    2)
```

加上这个”defs.h”头文件后就可以正常的编译 ida ”F5”后的 c 语言了。

另外我们还可以自己创建一个 NDK 项目，然后自己编写一个 so 或者 elf 利用 dlopen()和 dlsym()调用目标 so 中的函数。比如我们想要调用 libdvm.so 中的 dvmGetCurrentJNIMethod()函数，我们就可以在我们的 NDK 项目中这么写：

```
1typedef void* (*dvmGetCurrentJNIMethod_func)();
2dvmGetCurrentJNIMethod_func dvmGetCurrentJNIMethod_fnPtr;
3dvm_hand= dlopen("libdvm.so", RTLD_NOW);
4dvmGetCurrentJNIMethod_fnPtr =dlsym(dvm_hand, "_Z22dvmGetCurrentJNIMethodv");
5dvmGetCurrentJNIMethod_fnPtr();
```

0x10 小结

还是那句话，写了这么多依然不能保证本文能够覆盖到 ida 调试的方方面面，因为 ida 实在是太博大精深了。看官如有兴趣可以继续深入研究学习。另外文章中所有提到的代码和工具都可以在我的 github 下载到，地址是：

https://github.com/zhengmin1989/TheSevenWeapons

0x06 参考文章

---

1. MSC 解题报告 http://bbs.pediy.com/showthread.php?t=197235
2. 伪·MSC 解题报告 http://bbs.pediy.com/showthread.php?p=1349632

http://drops.wooyun.org/tips/6840

IDA pro 调试 Android APK 的动态链接库[基础]

憨搞逆向

IDA pro 调试 Android APK 的动态链接库

这年头 Android 逆向是原来越难，好多关键逻辑都是用 ndk 编写了，不学习 ARM 汇编，不学习 IDA 不行了。 采用的是远程调试方法。

1 复制 android_server 调试服务器到调试手机上

android_server 在 IDA pro 安装目录下的 dbgsrv 目录下。

1.1 在调试机的/data/local 下新建目录 ida

在命令行中依次输入:

adb shell su cd /data/local mkdir ida

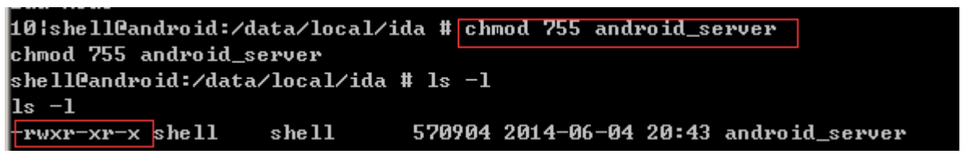
1.2 进入 android_server 所在目录

将 android_server 复制到调试机的/data/local/ida 目录下:

adb push android_server /data/local/ida

1.3 在 adb shell 中修改 android_server 为可执行的权限

chmod 755 android_server

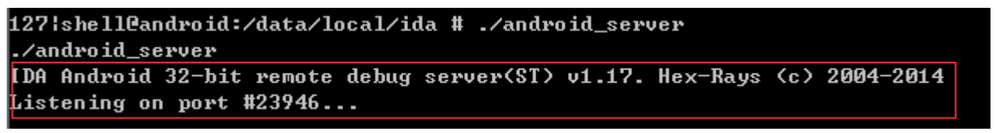


1.4 测试调试服务器 android_server 能否正常运行

在 adb shell 中的/data/local/ida 目录下，输入：

./android_server

如下图所示则成功，调试服务器正在监听 23496 端口



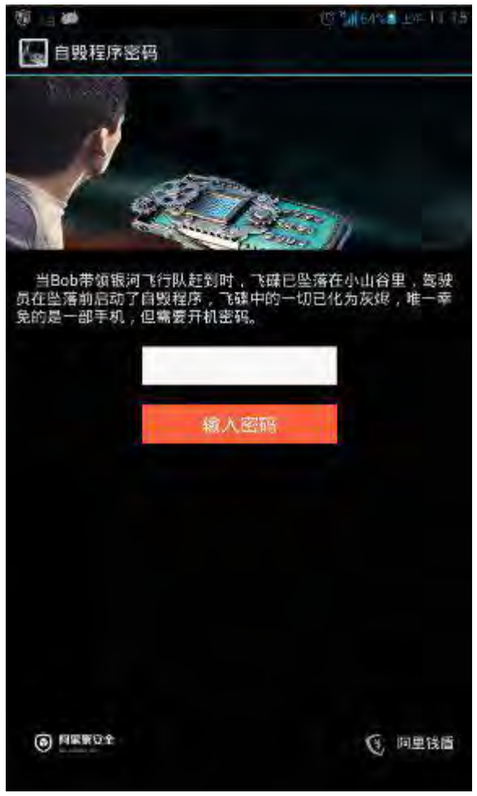
2 设置 IDA pro

2.1 进行端口转发

在 1.4 完成的情况下，成功启动 IDA pro 的 Android 调试服务器后，**重新打开一个 cmd**，输入端口转发命令：

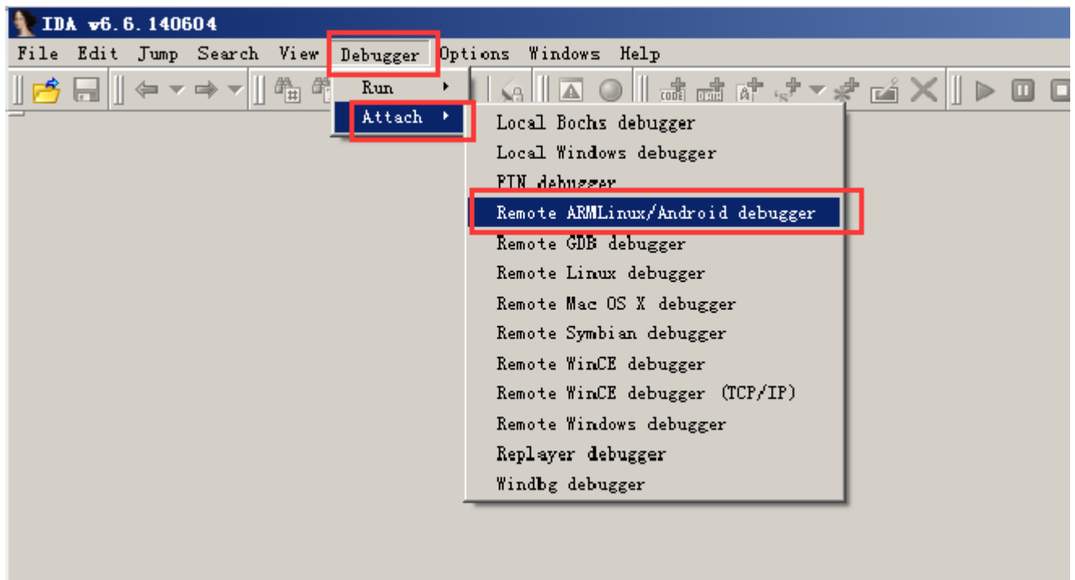
adb forward tcp:23946 tcp:23946

2.2 运行想要调试的应用



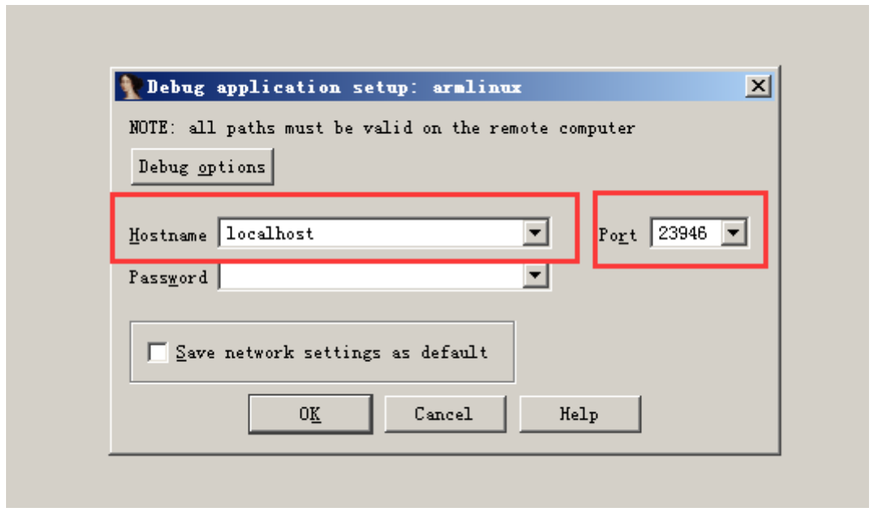
2.3 IDA pro 调试设置

启动 IDA pro，点击菜单栏中 Debugger->Attach->Remote ARMLinux/Android debugger：



在 Debug application setup 对话框中的 Hostname 中，选择或者输入 localhost，同时确保 Port 也是 23946:

点击 ok。

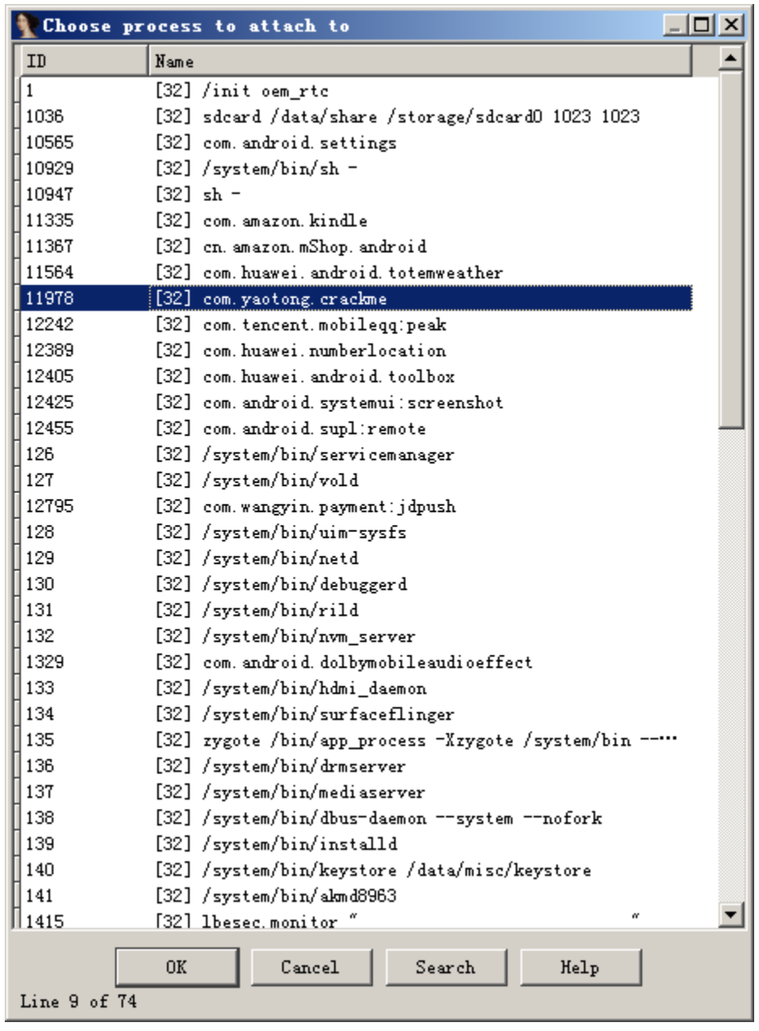


2.4 选择要 attach 的进程

点击 com.yaoton.crackme，再点击 OK 即可，

第一次设置可能 windows 的防火墙会拦截，设置为允许即可。

attach 成功。



2.5 为动态链接库中的函数设置断点

解压出要调试的程序的动态链接库 so 文件，再开启一个 IDA pro，找到从 Java 到 native 的 Jni 函数的起始地址，再回到 IDA pro 调试窗口，按下 ctrl + s 打开段选择对话框，查找动态链接库的基地址。

内存地址 = 基地址 + 偏移地址

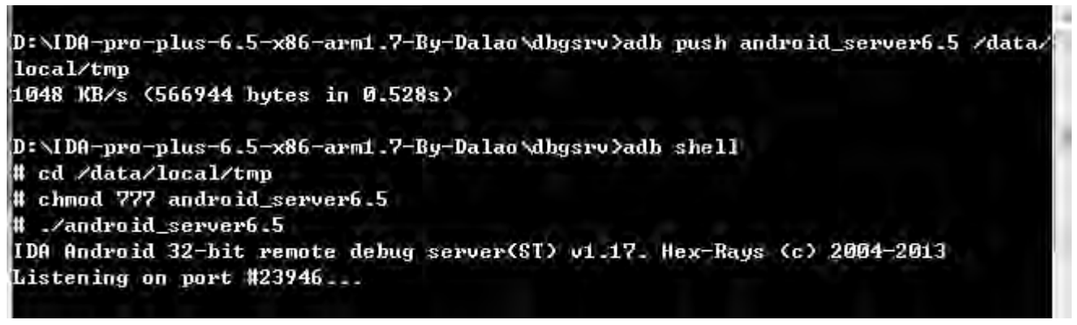
可以得出 Jni 函数在内存中的地址，然后按下快捷键 G，打开跳转对话框，输入内存地址。

接下来拿着《IDA pro 权威指南》和 ARM 指令手册调试吧，～～～～（＞__＜）～～～～ 。

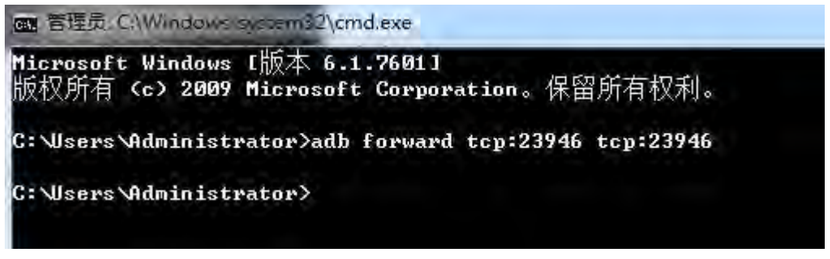
http://www.kechuandai.net/ida-pro%E8%B0%83%E8%AF%95android-apk%E7%9A%84%E5%8A%A8%E6%80%81%E9%93%BE%E6%8E%A5%E5%BA%93/

IDA 远程调试 Android so

1.把 ida 目录下 android_server 传到 android 目录中
如：
adb push android_server /data/local/tmp/
adb shell 进入模拟器
cd /data/local/tmp/
chmod 755 android_server
./android_server
看到监听端口 23946



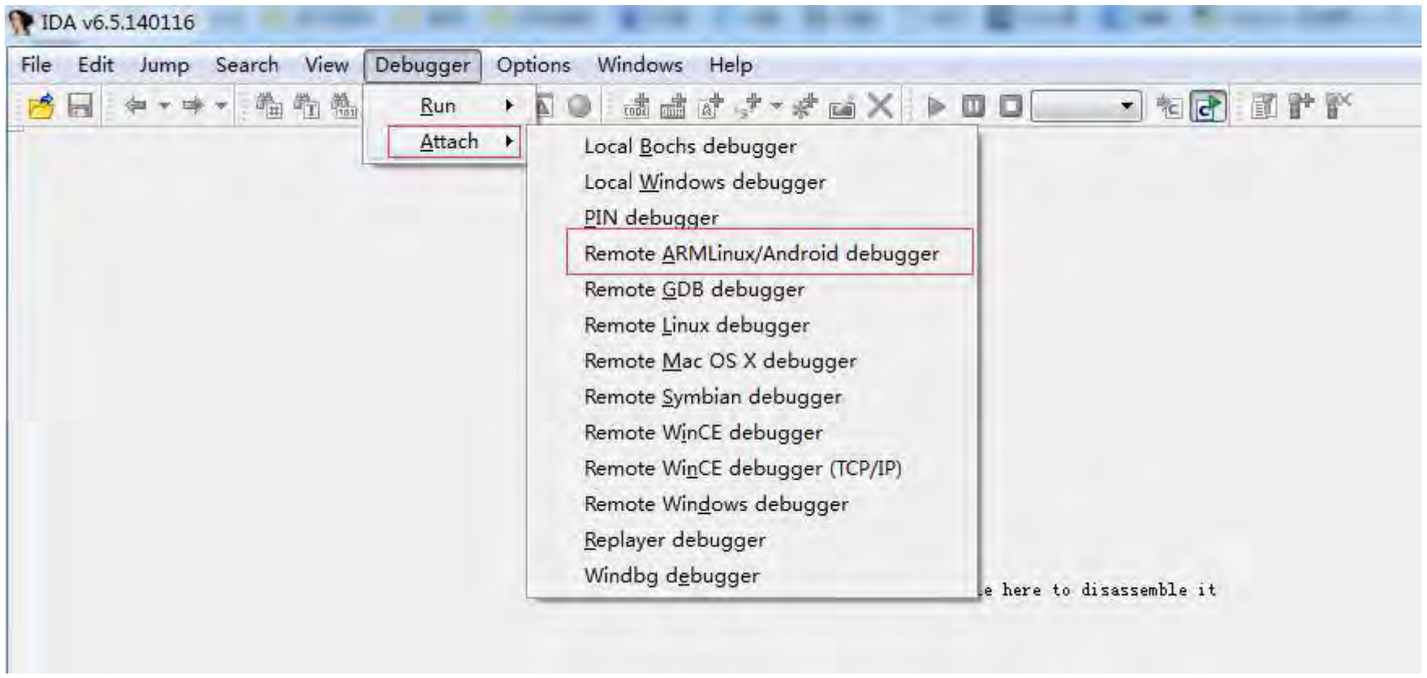
2.在 windows 控制台下转发 window 到模拟器或者手机的端口
adb forward tcp:23946 tcp:23946

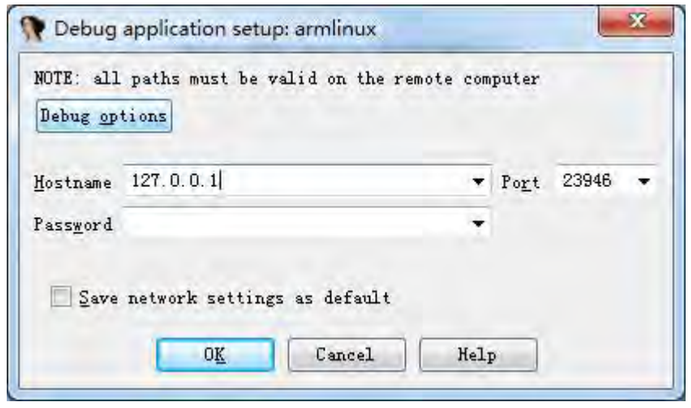


3.ida 中选择 android 调试, app 可以运行起来了
在 Debuggger 中的 process options 的

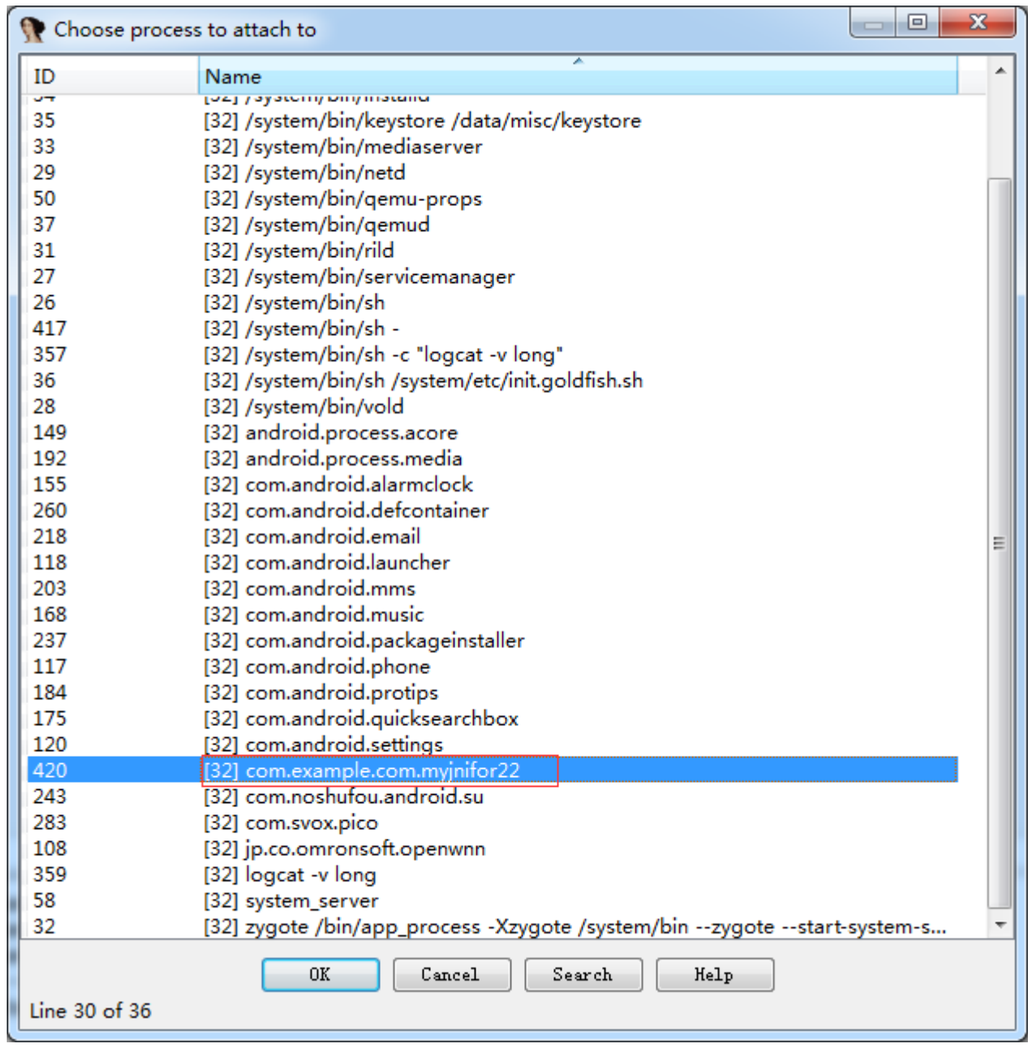
hostname 填上 localhost
port: 23946


5554:Android2.2
1:32 PM
MyJnifor2.2
NEW BUTTON
Basic Con
Hardware
DPAD not
Hardware
Use on sc





4. 在 Debuggger 中的 attach 上 android 所对应的程序就行了



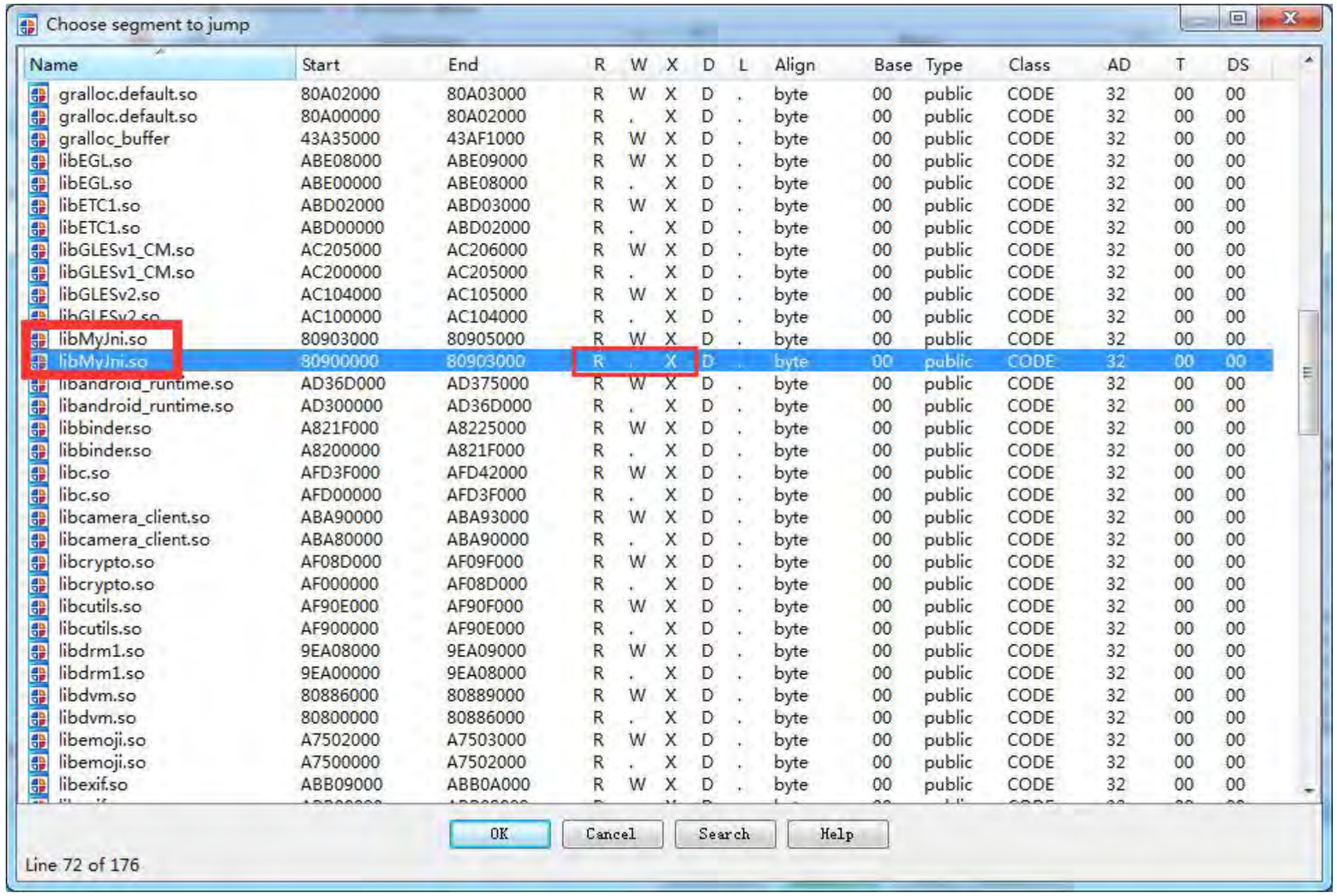
5. 找到需要下断点的位置 设置断点，进行调试即可

现在本地模块中找到需要下断点的函数偏移

然后获取远程模块的基地址，基地址+偏移即是当前需要下断点的地址

```
.text:00000C88
.text:00000C88                 EXPORT Java_com_example_com_myjnifor22_MainActivity_Mydecrypt
.text:00000C88 Java_com_example_com_myjnifor22_MainActivity_Mydecrypt
.text:00000C88
.text:00000C88 var_30          = -0x30
.text:00000C88 var_2C          = -0x2C
.text:00000C88 var_28          = -0x28
.text:00000C88 var_24          = -0x24
.text:00000C88 var_20          = -0x20
.text:00000C88 var_1C          = -0x1C
.text:00000C88
.text:00000C88                 PUSH    {R4-R7,LR}
.text:00000C8A                 LDR     R1, =(aJavaSecuritySe - 0xC96)
.text:00000C8C                 MOVS    R5, R3
.text:00000C8E                 LDR     R3, [R0]
.text:00000C90                 SUB     SP, SP, #0x1C
.text:00000C92                 ADD     R1, PC          ; "java/security/SecureRandom"
.text:00000C94                 LDR     R3, [R3,#0x18]
```

Ctrl + s 显示当前的所有加载的模块信息，找可读可执行的，一般代码段都是不可写的。



80900000 + C88 = 需要下断点的地址，然后设置断点，点击按钮就断下来了。

```
libMyJni.so:80900C88
libMyJni.so:80900C88 Java_com_example_com_myjnifor22_MainActivity_Mydecrypt
libMyJni.so:80900C88
libMyJni.so:80900C88 var_30= -0x30
libMyJni.so:80900C88 var_2C= -0x2C
libMyJni.so:80900C88 var_28= -0x28
libMyJni.so:80900C88 var_24= -0x24
libMyJni.so:80900C88 var_20= -0x20
libMyJni.so:80900C88 var_1C= -0x1C
libMyJni.so:80900C88
libMyJni.so:80900C88 PUSH            {R4-R7,LR}
libMyJni.so:80900C8A LDR             R1, =(aJavaSecuritySe - 0x80900C96)
libMyJni.so:80900C8C MOVS            R5, R3
libMyJni.so:80900C8E LDR             R3, [R0]
libMyJni.so:80900C90 SUB             SP, SP, #0x1C
libMyJni.so:80900C92 ADD             R1, PC                  ; "java/security/SecureRandom"
libMyJni.so:80900C94 LDR             R3, [R3,#0x18]
libMyJni.so:80900C96 MOVS            R4, R0
```

```
libMyJni.so:80900C88
libMyJni.so:80900C88 ; =============== S U B R O U T I N E =======================================
libMyJni.so:80900C88
libMyJni.so:80900C88
libMyJni.so:80900C88 Java_com_example_com_myjnifor22_MainActivity_Mydecrypt
libMyJni.so:80900C88
libMyJni.so:80900C88 var_30= -0x30
libMyJni.so:80900C88 var_2C= -0x2C
libMyJni.so:80900C88 var_28= -0x28
libMyJni.so:80900C88 var_24= -0x24
libMyJni.so:80900C88 var_20= -0x20
libMyJni.so:80900C88 var_1C= -0x1C
libMyJni.so:80900C88
libMyJni.so:80900C88 PUSH            {R4-R7,LR}
libMyJni.so:80900C8A LDR             R1, =(aJavaSecuritySe - 0x80900C96)
libMyJni.so:80900C8C MOVS            R5, R3
libMyJni.so:80900C8E LDR             R3, [R0]
libMyJni.so:80900C90 SUB             SP, SP, #0x1C
libMyJni.so:80900C92 ADD             R1, PC                  ; "java/security/SecureRandom"
libMyJni.so:80900C94 LDR             R3, [R3,#0x18]
libMyJni.so:80900C96 MOVS            R4, R0
libMyJni.so:80900C98 STR             R2, [SP,#0x30+var_1C]
```

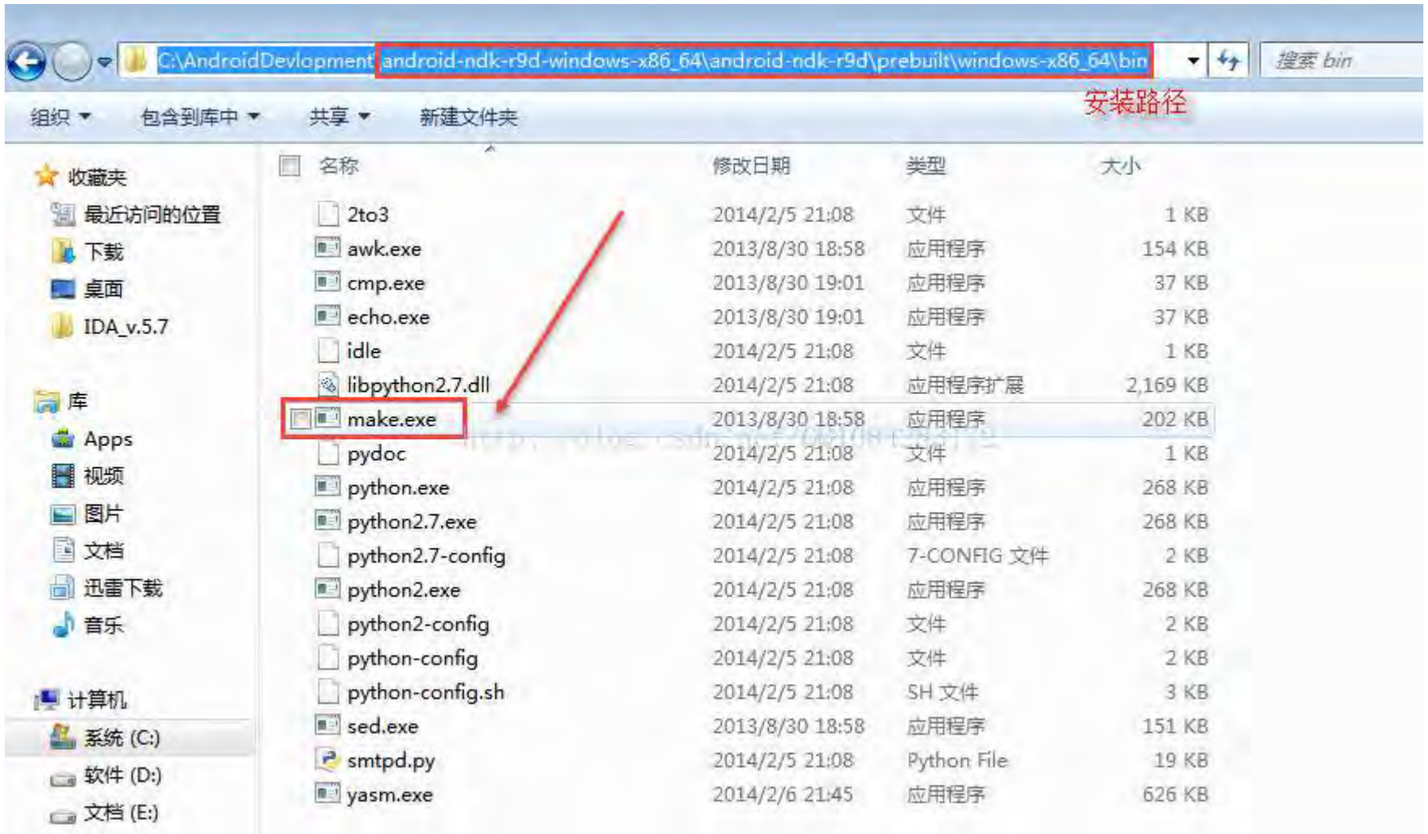
Android 逆向之动态调试 so 库 JNI_Onload 函数-基于 IDA 实现

Android 逆向之动态调试 so 库 JNI_Onload 函数-----基于 IDA 实现

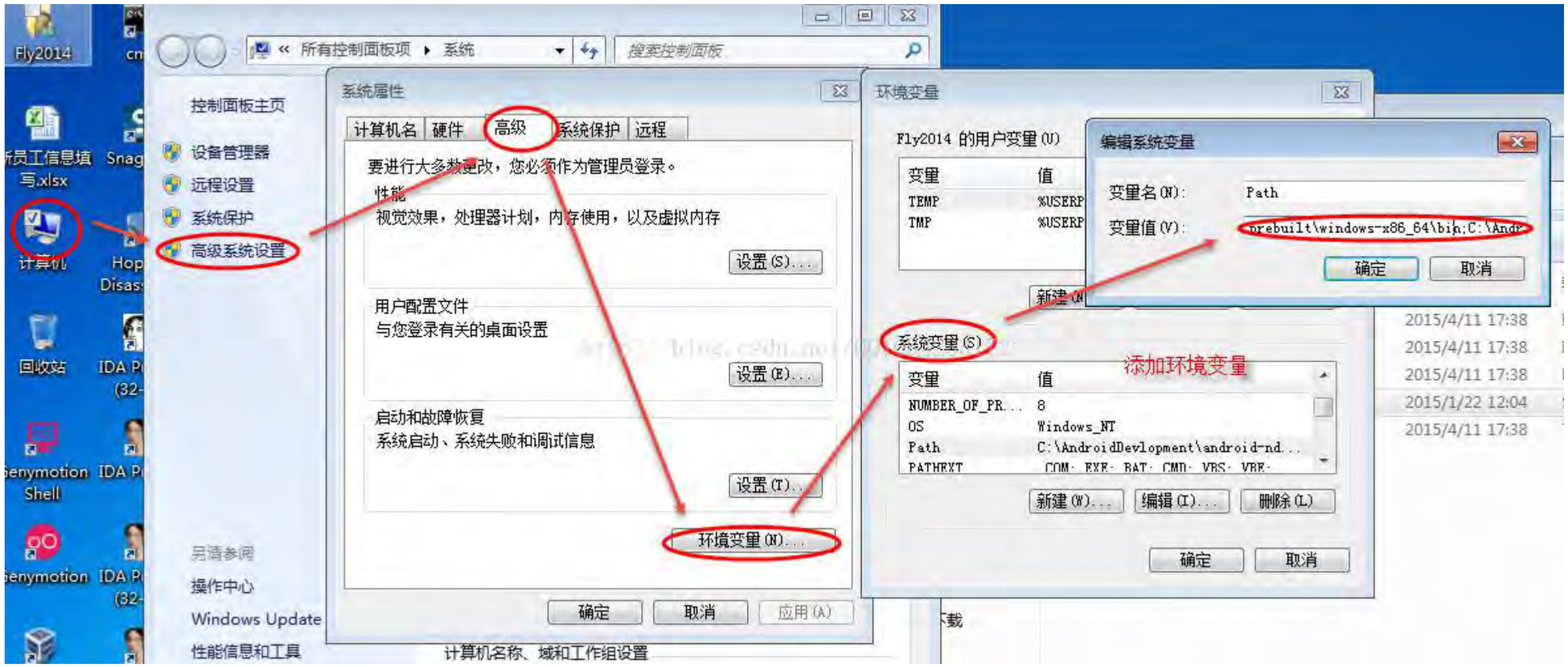
之 前看过吾爱破解论坛一个关于 Android'逆向动态调试的经验总结帖，那个帖子写的很好，对 Android 的脱壳和破解很有帮助，之前我们老师在上课的 时候也讲过集中调试的方法，但是现在不太实用。对吾爱破解论坛的该贴，我也是看了很多遍，自己也查了不少资料，但是自己动手的时候总觉比较繁琐，并且很多 细节的地方没有注意到，按照那个帖子尝试了几遍但是却出现了错误（后面会提到），今天周末重新拾起来试了试，终于把遇到的问题给解决了，顺便做个记录以免 忘记了，其中的一些细节我也不是太明白，忘知道的人给指出。

第一步、给 meke.exe 工具添加系统环境变量

对 Android 的动态调试，要在 cmd 控制台手动输入命令，比较繁琐，下面我就偷懒用个简单一点的方法。为了能偷懒，我们需要将 AndroidNDK 提供的 make.exe 工具位于安装目录 C:\AndroidDevlopment\android-ndk-r9d-windows-x86_64\android-ndk-r9d\prebuilt\windows-x86_64\bin 下（具体的路径由自己安装目录决定），在进行 Android 动态调试之前，需要将该路径添加到系统或临时的 PATH 环境变量中，具体的操作如下：



将该路径添加到系统环境 PATH 变量中：



第二步、准备 android_server 文件和编写 Android 动态调试需要的 mk 文件

android_server 文件是由 IDA 6.6 提供，具体的文件路径是在 IDA 的安装路径的 IDA 6.6\dbgsrv 目录下，这里直接拷贝过来使用（只有 IDA6.1 以上版本才支持 Android 的动态调试）。具体的 mk 文件我已经写好了，可以直接拷贝代码保存为.mk 后缀文件使用，至于 makefile 文件的编写我也不知道。

cmd 控制台 1 使用的 listen.mk 文件的编写：

```
#说明(控制台 1)
#使用 Android-NDK 提供的 make.exe 程序，需将该程序的路径 xx\android-ndk-r9d-windows-x86_64\android-ndk-r9d\prebuilt\windows-x86_64\bin 添加到环境变量
#记得要先开启 Android 模拟器或者将开发的手机连接到电脑
#这里 android_server 的版本是 IDA 6.6 的
#使用命令(控制台 1) make -f listen.mk


#文件名称
MODALE_NAME=crackme.apk
```

```
#安装程序到手机
listen:
	adb push $(MODALE_NAME) /data/local/tmp
	adb shell chmod 755 /data/local/tmp/$(MODALE_NAME)

	adb push android_server /data/local/tmp
	adb shell chmod 755 /data/local/tmp/android_server

#调试模式启动程序
#此时，手机界面会出现 Waiting For Debugger 页面
#格式 adb shell am start -D -n 包名/.类名
	adb shell am start -D -n com.yaotong.crackme/.MainActivity

#端口转发
	adb forward tcp:23946 tcp:23946

#启动 android_server
#adb shell su
	adb shell /data/local/tmp/android_server
```

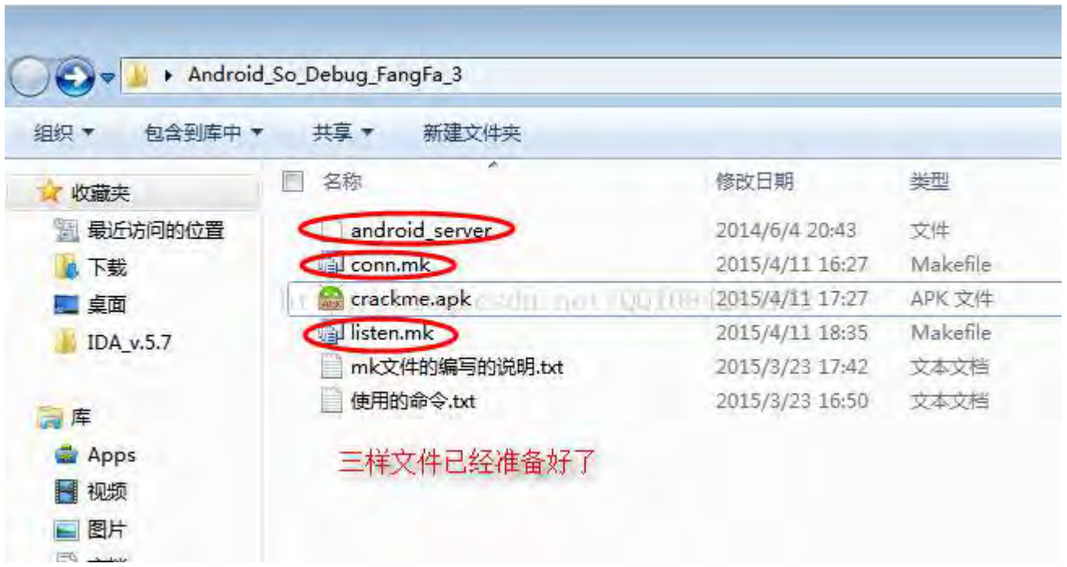
cmd 控制台 2 使用的 conn.mk 文件的编写：

```
#接下来 IDA 附加，设置调试的选项（控制台 2）
#静态找到目标函数对应所在模块的偏移地址，Ctrl+S 找到对应模块的基地址，两个地址相加得到最终地址
#G 跳转至地址，然后下断,F9 运行
#其中 port=8700 是从 ddms 中看到的
#IDA 中，F9 运行程序，此时是 runing 状态
conn:
	jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700
```

简要的说一下.mk 文件中的命令的作用：

```
#使用命令(控制台 1) make -f listen.mk
//注释  make -f listen.mk  说明是 listen.mk 文件在控制台使用的命令格式
#文件名称
MODALE_NAME=crackme.apk
//crackme.apk 是我们要动态调试的目标 apk 应用程序
adb push $(MODALE_NAME) /data/local/tmp
//使用 adb 程序将要调试的目标 apk 应用程序拷贝 Android 系统的文件/data/local/tmp 目录下
adb shell chmod 755 /data/local/tmp/$(MODALE_NAME)
//adb shell 命令的意思是进入到 Android 系统中
```

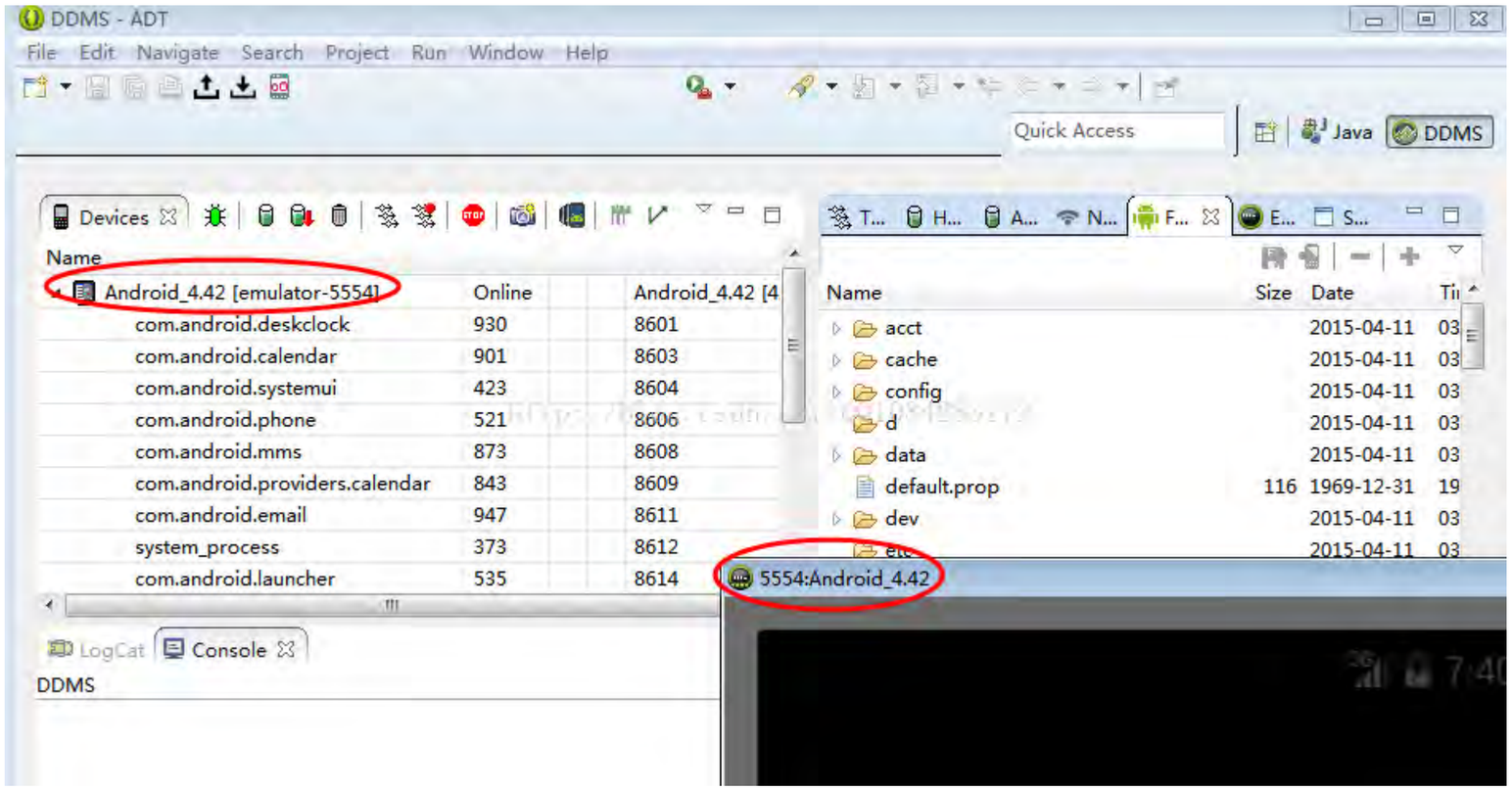
//上面这条命令的作用是修改/data/local/tmp 目录下的目标 apk 应用程序的文件权限为 755
adb push android_server /data/local/tmp
//使用 adb 程序将 android_server 程序文件拷贝 Android 系统的文件/data/local/tmp 目录下
adb shell chmod 755 /data/local/tmp/android_server
//修改 Android 系统目录/data/local/tmp 下的 android_server 程序文件的权限为 755
#格式 adb shell am start -D -n 包名/.类名 或者 adb shell am start -D -n 包名/包名.类名
adb shell am start -D -n com.yaotong.crackme/.MainActivity
//使用 adb shell am start -D -n 包名/.类名 命令以-D 调试模式启动 apk 应用程序，apk 应用程序调试模式启动以后，会停止在 Waiting For Debugger 界面上。am start 命令的具体的使用参考网址：
http://developer.android.com/tools/help /adb.html#IntentSpec
adb forward tcp:23946 tcp:23946
//adb 端口转发
adb shell /data/local/tmp/android_server
//启动拷贝到 Android 系统中 android_server 程序，等待 IDA6.6 程序的连接
#使用命令(控制台 2) make -f conn.mk
//注释 make -f conn.mk 说明是 conn.mk 文件在控制台使用的命令格式
jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700
//jdb 调试器的使用，具体命令行含义不知道，貌似 Linux 下该命令还不是这样的



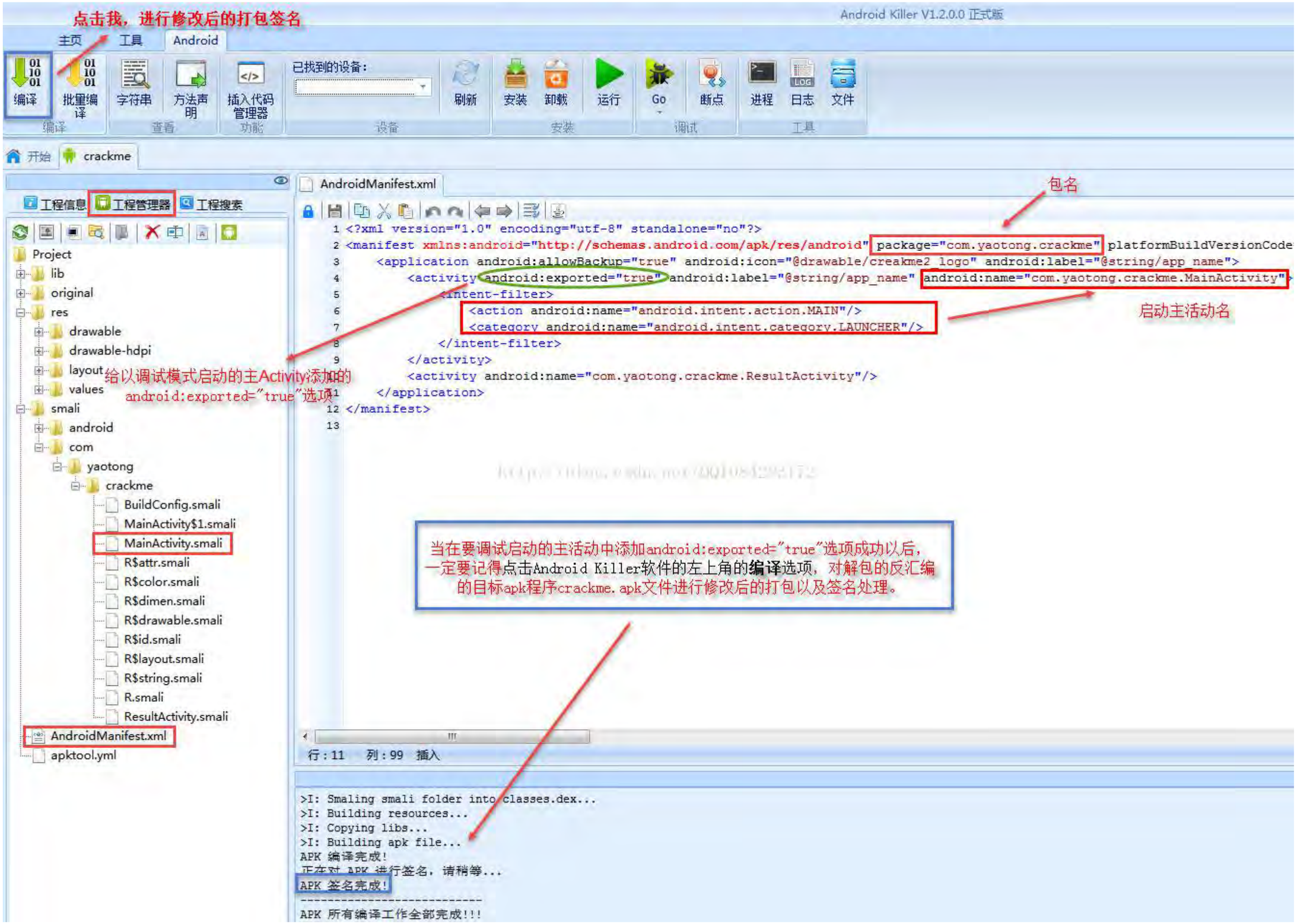
OK，需要的文件已经准备齐全了，Android 逆向动态调试的脚步又就进了一步。

第三步、打开 Eclipse 应用程序和为要调试的目标 apk 应用程序添加 android:exported="true"选项，生成符合调试的 apk 程序

在 Android 逆向动态调试的时候，必须要打开 Eclipse 程序，并且还要运行 Android 的模拟器 DDMS 以方便使用 adb 程序和 jdb 程序，有钱的土豪可以使用真机调试。



在进行 Android 动态调试的动手之前，我们还需要做一件事情，为要调试的目标 apk 程序 android:exported="true"以及修改 listen.mk 文件。前面看别人的写的帖子和博客也可能是人品问题！在 Android 动态调试的时候 am start -D 命令不能以调试模式启动要调试的目标 apk 应用程序，也可能是人品爆发，今天百度查资料解决了，解决的方案就是为启动 Activity 添加 android:exported="true"选项，加了这个选项之后执行 am start -D 命令，被调试的目标 Apk 程序能够出现 Waiting For Debugger 界面并停在那儿。那么现在的问题就是如何添加 android:exported="true"选项，很简单。我们使用 Android 逆向工程利器 AndroidKiller 工具，这个工具的使用很久简单。我这里要动态调试的目标程序为 crackme.apk 应用程序，接下来打开 AndroidKiller 程序，直接拖拽 crackeme.apk 程序到 AndroidKiller 程序的界面内进行 crackeme.apk 应用程序的逆向反汇编，在解压的工程选项中找到 crackme.apk 的配置文件 AndroidManifest.xml 打开，如下图依次找到 crackme.apk 文件的包名、主活动的 Activity 以及正确的在活动中添加 android:exported="true"选项:

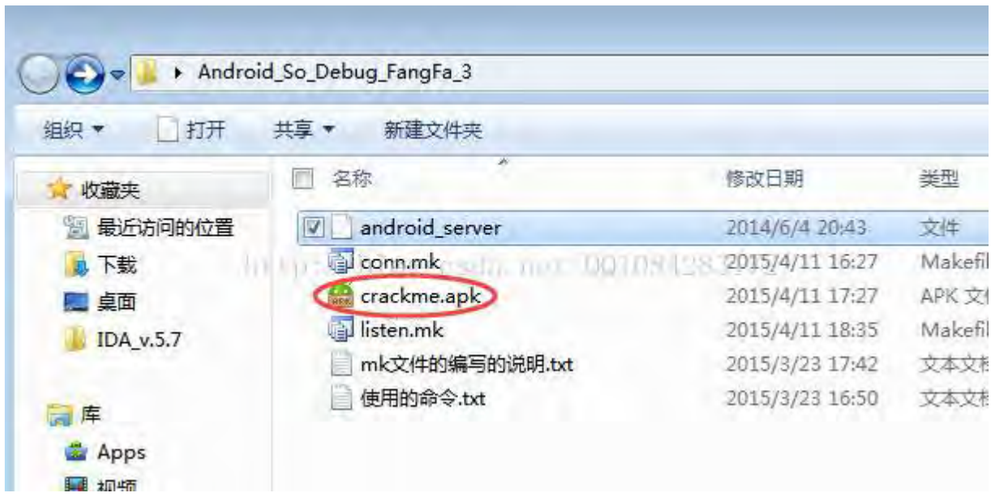

点击我，进行修改后的打包签名
Android Killer V1.2.0.0 正式版
主页
工具
Android
编译
批量编译
字符串
方法声明
插入代码管理器
已找到的设备：
刷新
安装
卸载
运行
Go
断点
进程
日志
文件
开始
crackme
工程信息
工程管理器
工程搜索
Project
lib
original
res
drawable
drawable-hdpi
layout
values
smali
android
com
yaotong
crackme
BuildConfig.smali
MainActivity$1.smali
MainActivity.smali
R$attr.smali
R$color.smali
R$dimen.smali
R$drawable.smali
R$id.smali
R$layout.smali
R$string.smali
R.smali
ResultActivity.smali
AndroidManifest.xml
apktool.yml
AndroidManifest.xml
包名
1 <?xml version="1.0" encoding="utf-8" standalone="no"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.yaotong.crackme" platformBuildVersionCode
3 <application android:allowBackup="true" android:icon="@drawable/creakme2_logo" android:label="@string/app_name">
4 <activity android:exported="true" android:label="@string/app_name" android:name="com.yaotong.crackme.MainActivity">
5 <intent-filter>
6 <action android:name="android.intent.action.MAIN"/>
7 <category android:name="android.intent.category.LAUNCHER"/>
8 </intent-filter>
9 </activity>
11 <activity android:name="com.yaotong.crackme.ResultActivity"/>
11 </application>
12 </manifest>
13
启动主活动名
给以调试模式启动的主Activity添加的android:exported="true"选项
当在要调试启动的主活动中添加android:exported="true"选项成功以后，一定要记得点击Android Killer软件的左上角的编译选项，对解包的反汇编的目标apk程序crackme.apk文件进行修改后的打包以及签名处理。
行：11 列：99 插入
>I: Smaling smali folder into classes.dex...
>I: Building resources...
>I: Copying libs...
>I: Building apk file...
APK 编译完成!
正在对 APK 进行签名，请稍等...
APK 签名完成!
APK 所有编译工作全部完成!!!

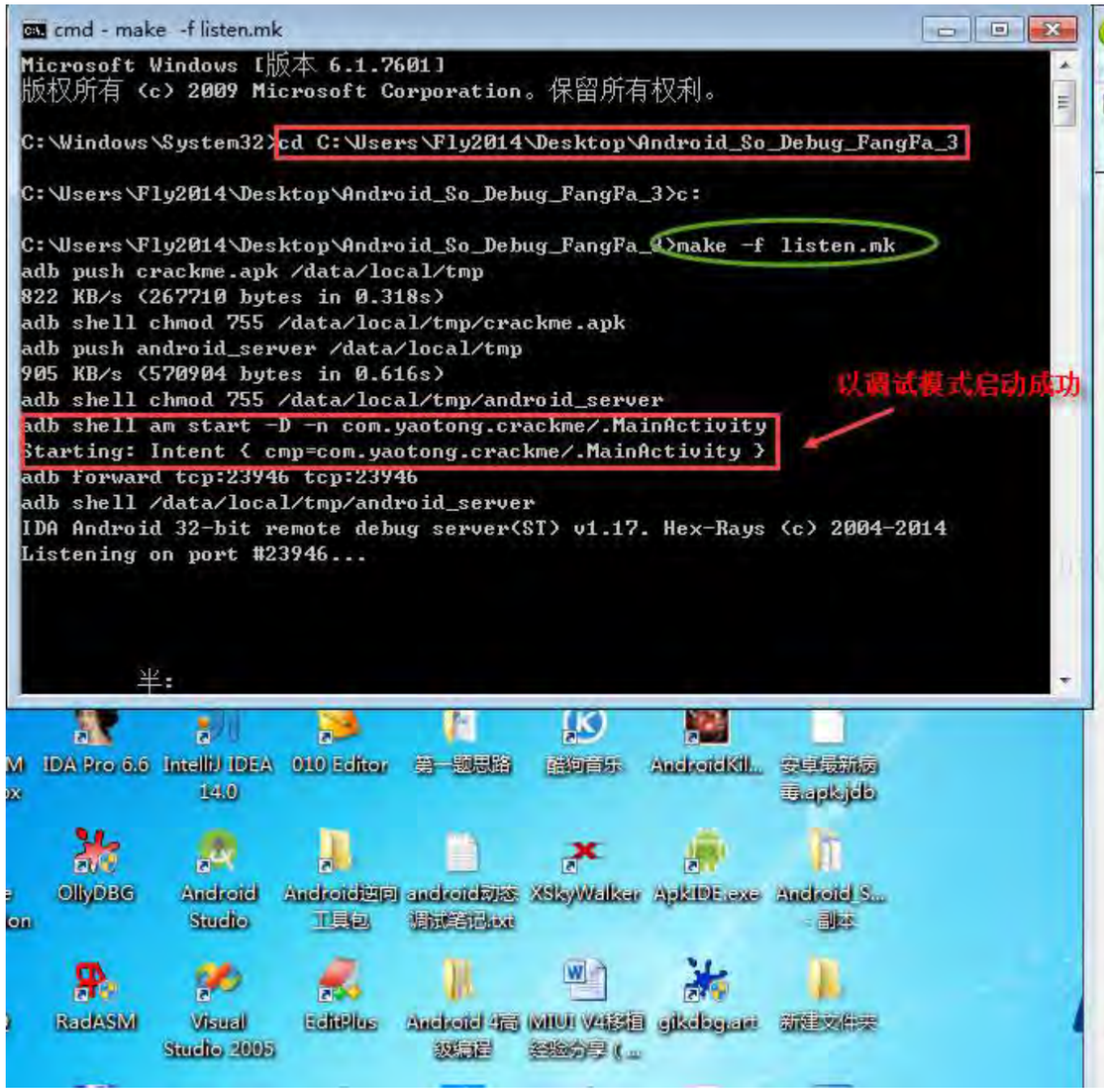
在 对要动态调试的目标 apk 程序 crackeme.apk 文件进行解包逆向修改添加 android:exported="true"选项以后，千万不能忘记 对修改后的目标程序的工程进行再次打包、签名处理生成新的 crakeme.apk 程序以备后面调试的时候使用，新生成的 crakeme.apk 程序在 Android Killer 的安装目录的 C:\AndroidDevlopment\AndroidKiller\projects\crackme\Bin 路径下（具体的路径由 Android Killer 的安装目录决定）。现在轻松多了，要动态调试的目标应用程序 crackme.apk 文件有了。

对了，对应着要调试的目标 apk 应用程序的名称以及它的主启动 Activity 相应的修改 listen.mk 文件中的变量 **MODALE_NAME=**crackme.apk 以及 命令 **adb shell am start -D -n**com.yaotong.crackme/.MainActivity。

第四步、开启一个 cmd 控制台以调试模式启动目标 apk 程序

将第三步中修改后重新打包生成的 C:\AndroidDevlopment\AndroidKiller\projects\crackme\Bin 路径下的 crackme.apk 文件以及 android_server、listen.mk 文件、conn.mk 文件拷贝到同一目录下，然后**开启一个 cmd 控制台** cd 命令进入到该目录，执行命令 make -f listen.mk 如下图：

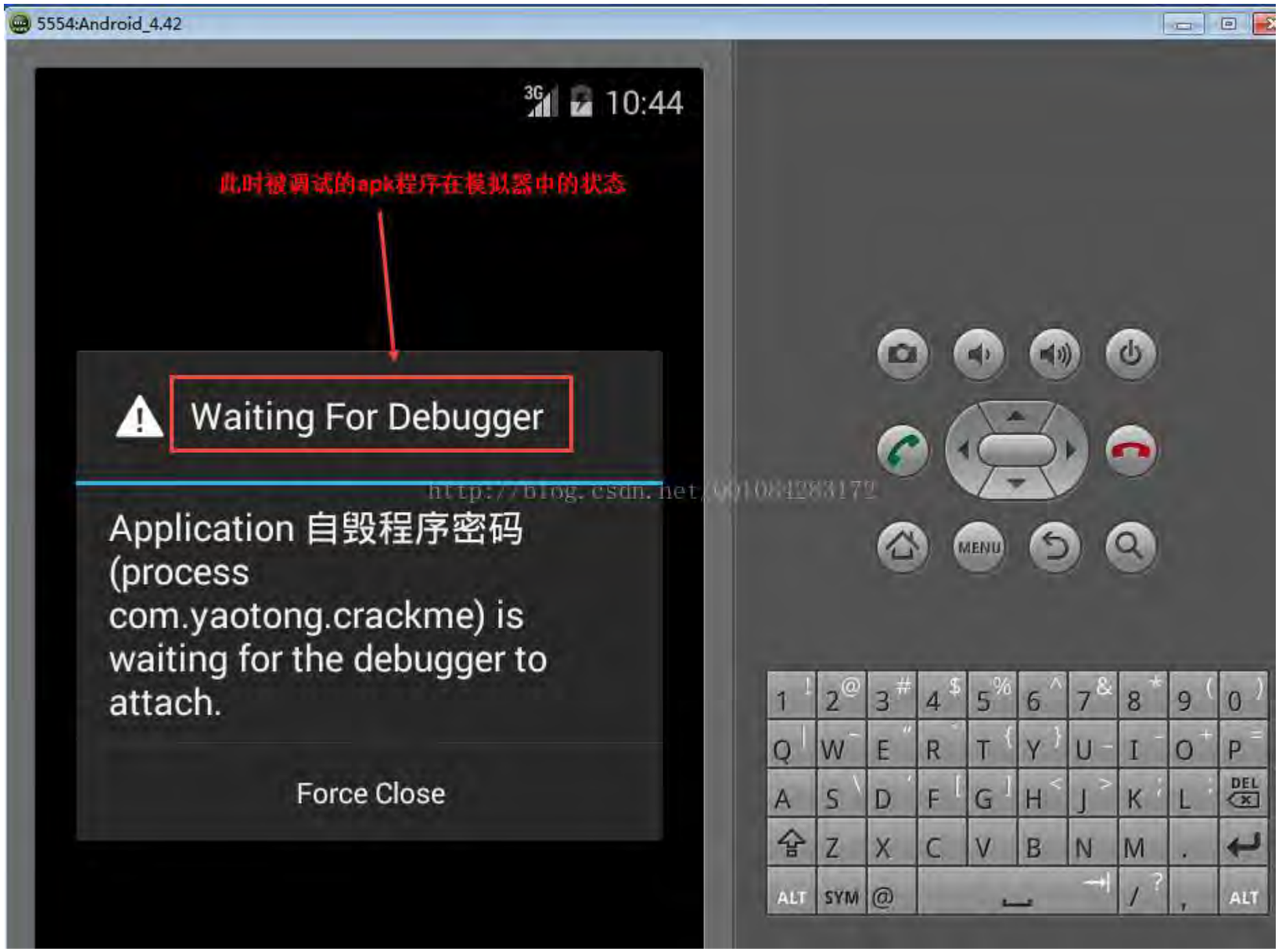




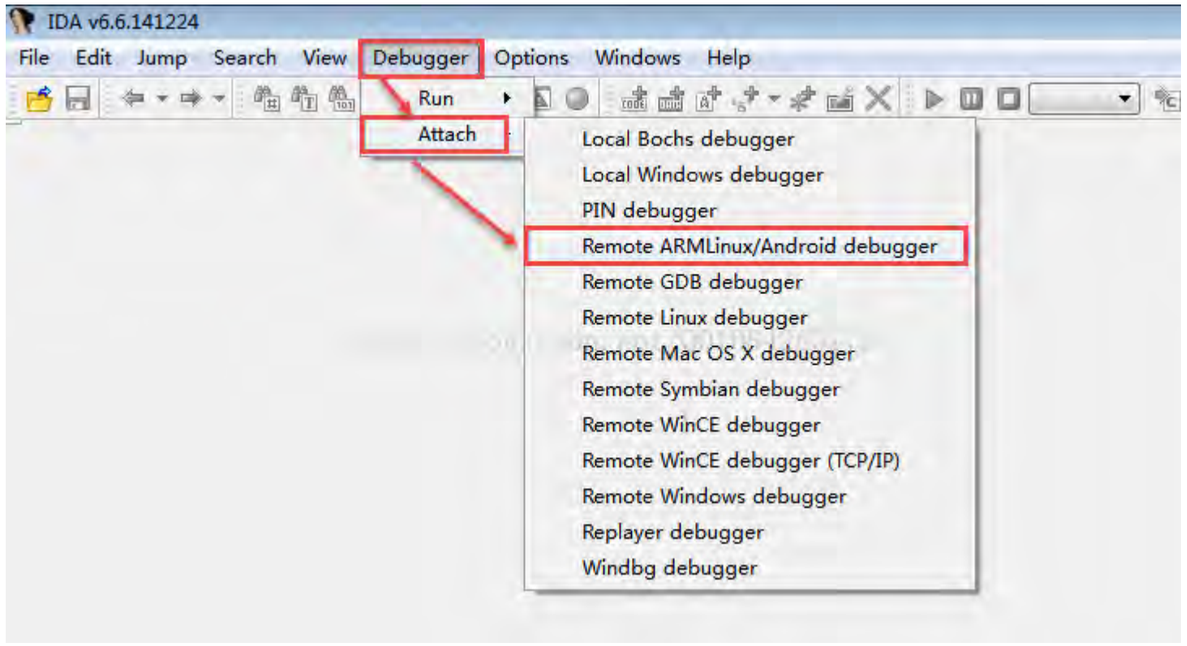
其实在这里的时候要注意一下，如果在第三步中**没有**为要调试的目标 apk 应用程序添加 android:exported="true" 选项，在执行 adb shell am start -D -n com.yaotong.crackme/.MainActivity 命令的时候会出现如下的错误，但是如果为要调试的目标 apk 应用程序添加 android:exported="true" 选 项以后，执行该命令不会出现下面的错误提示，具体的原因我也不知道，之前在尝试 Android 动态调试的时候也是卡在了这里，可能是 Android'应用 程序的权限问题吧，在注册 Android 内容提供者的时候也需要导出，可能道理一样，也可以参考一下这篇文章 http://chenxuebinbj.blog.163.com/blog/static/4286915120130223521583 2/。对于 此种错误的情况，我在看雪论坛上也遇到过了。

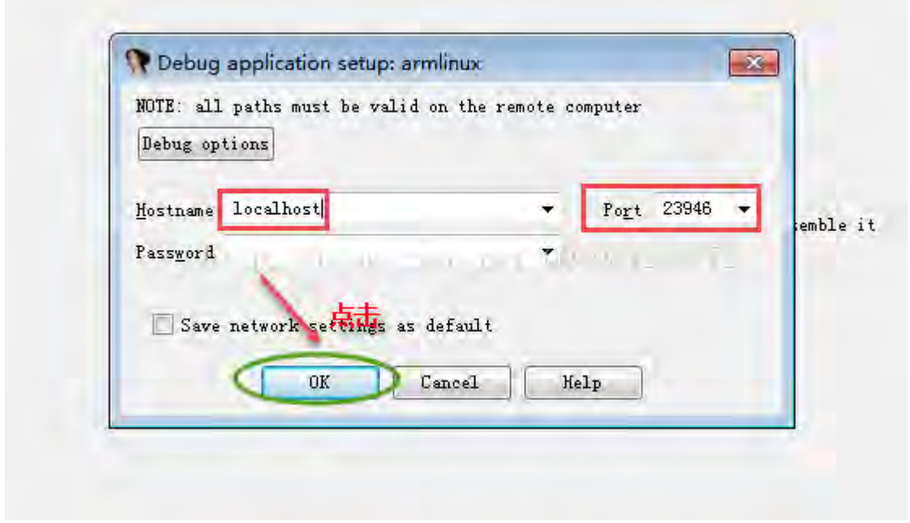
注：我真的很无语，当我写完这篇笔记想将该错误重现的时候，无语的事情出现了，我没有 android:exported="true" 选项再次执行 adb shell am start -D -n com.yaotong.crackme/.MainActivity 命令的时候，竟然没有报找不到 com.yaotong.crackme/.MainActivity 类的错误。我还连续试了几次，之前遇到的这个 am start -D 命令的错误提示都没有出现，真无语。更正一下，看来添加 android:exported="true" 选项不是必须的。

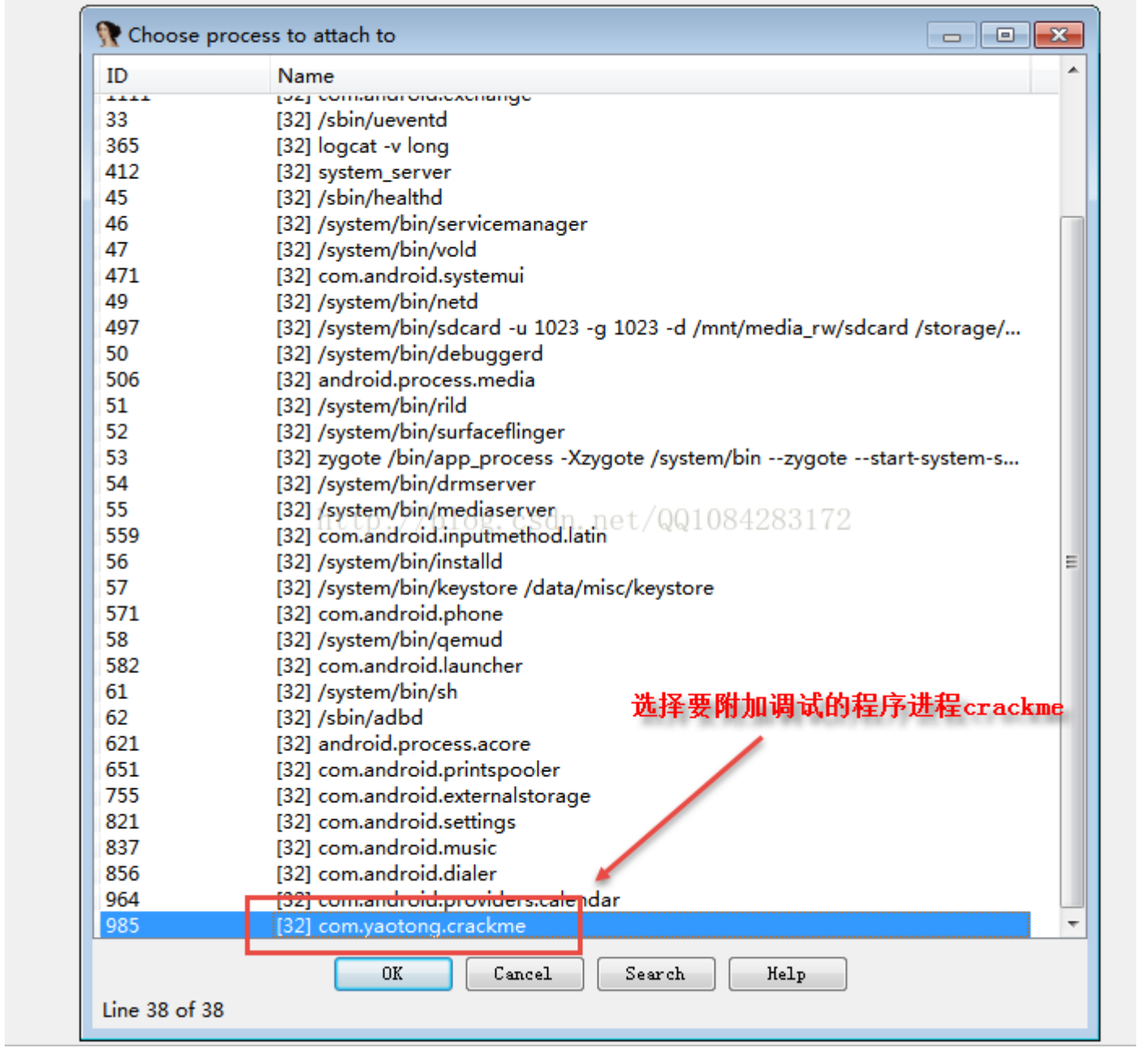
此时 Android'模拟中被动态调试的目标 crackme.apk 程序的运行状态如下，停止在了 Waiting For Debugger 界面：



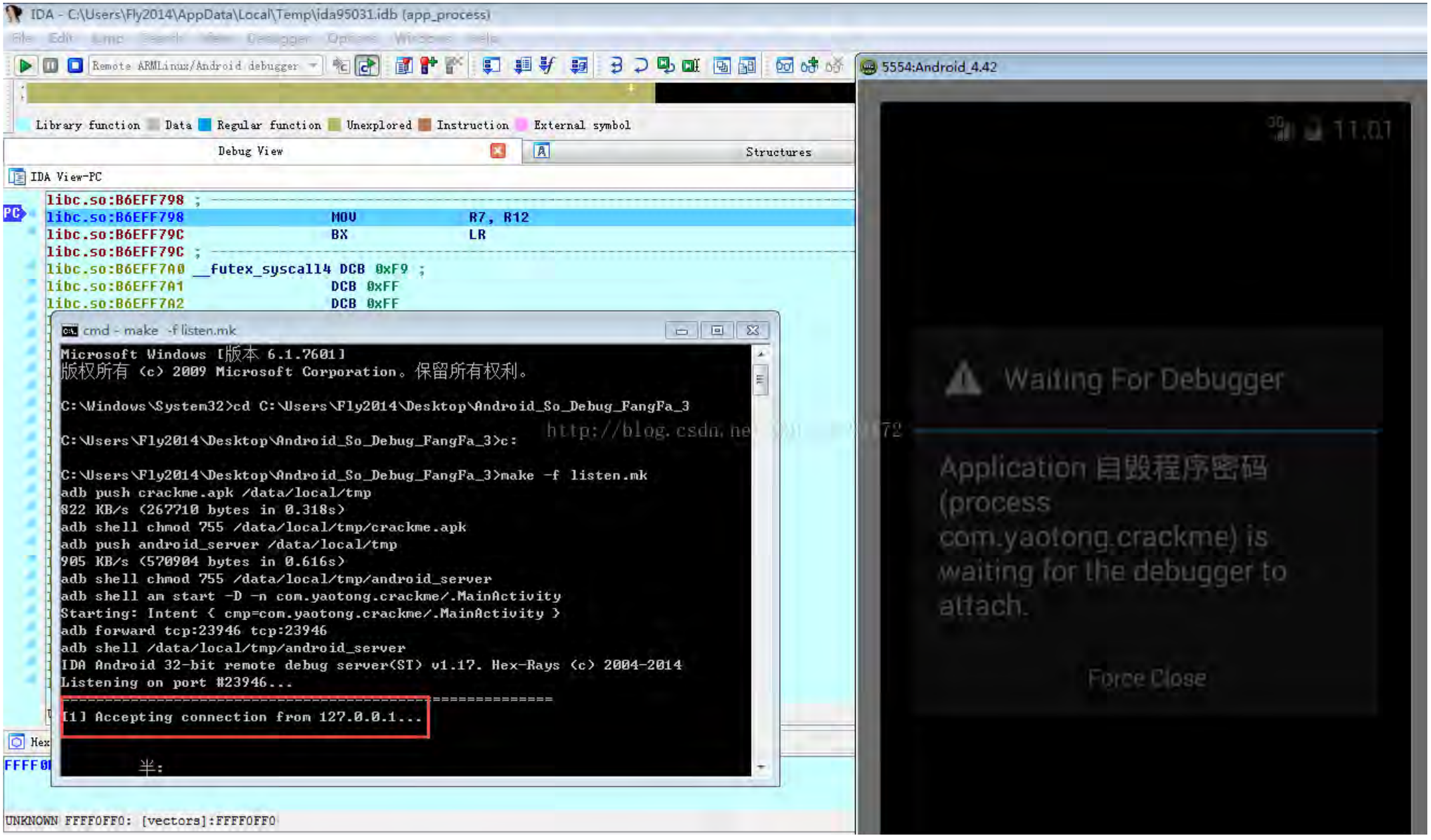
启动一个 IDA 主程序，点击菜单 Debugger->Attach->Remote ArmLinux/Android debugger，打开调试程序对话框，在 hostname 一栏输入 localhost,点击 ok，然后在 IDA 弹出的窗口中，选择自己要附加的进程 com.yaotong.crackme 后点击 OK 即可，如下图：


IDA v6.6.141224
File Edit Jump Search View Debugger Options Windows Help
Run
Attach
Local Bochs debugger
Local Windows debugger
PIN debugger
Remote ARMLinux/Android debugger
Remote GDB debugger
Remote Linux debugger
Remote Mac OS X debugger
Remote Symbian debugger
Remote WinCE debugger
Remote WinCE debugger (TCP/IP)
Remote Windows debugger
Replayer debugger
Windbg debugger

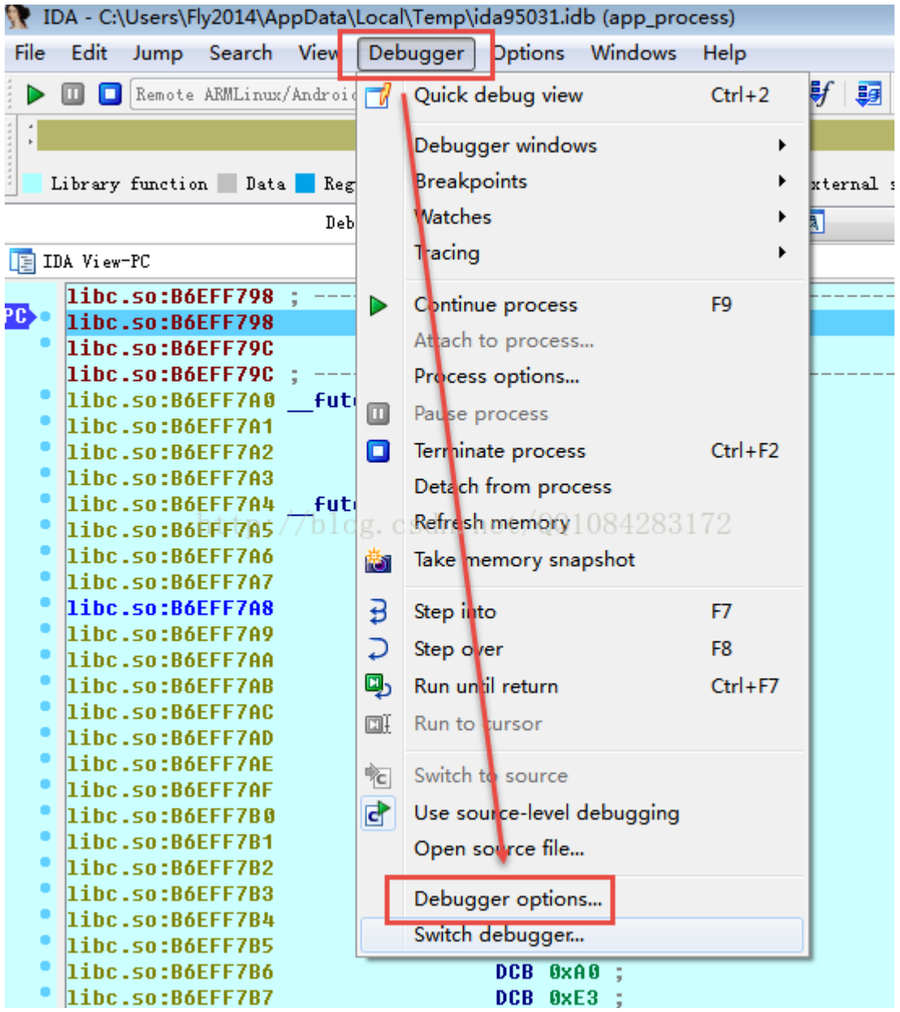

Debug application setup: armlinux
NOTE: all paths must be valid on the remote computer
Debug options
Hostname localhost
Port 23946
Password
点击
Save network settings as default
OK
Cancel
Help

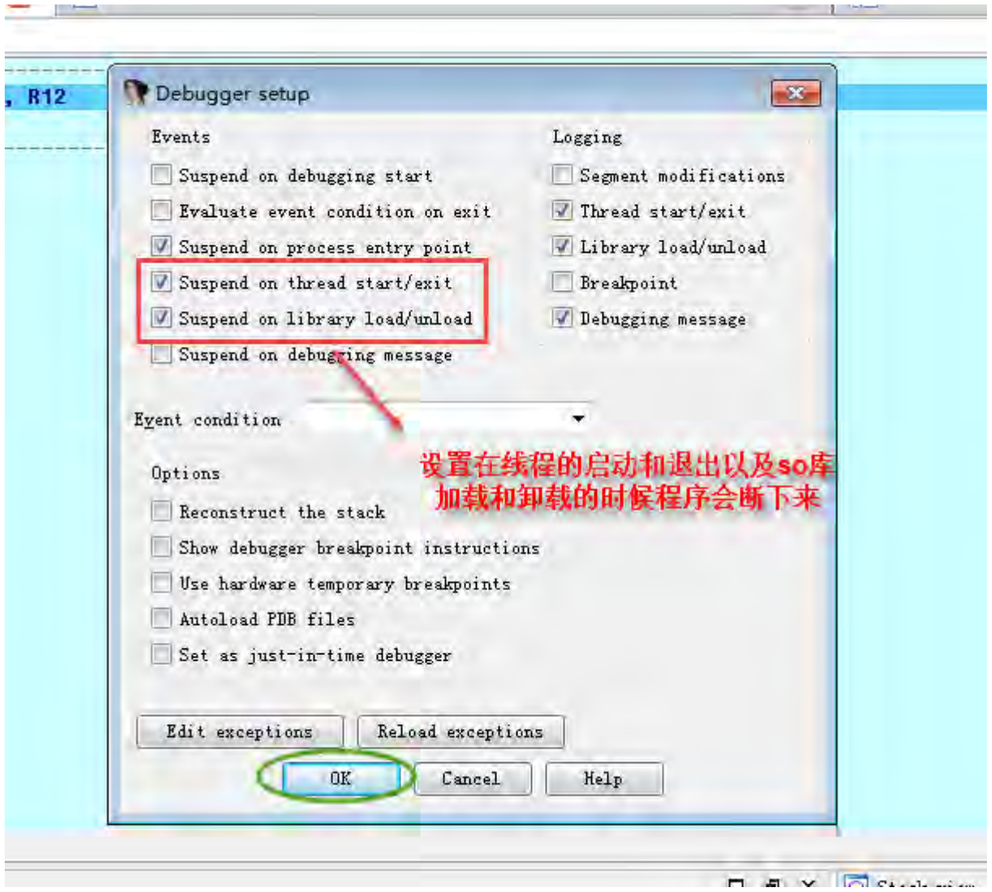


此时，要动态调试的目标 apk 应用程序的进程被 IDA6.6 程序附加成功以后的状态如下图：



下 面我们为目标 apk 应用程序的动态调试进行 IDA6.6 的一些调试设置，点击该 IDA 菜单项 Debugger->Debugger Opitions 在弹出的 Debugger setup 窗口的 Events 中选择 suspend on thread start/exit 以及 suspend on library load/unload，再点击 OK 退出。通过此操作可以设置程序在创建新线程和加载 so 时自动中断。具体操作见下图：

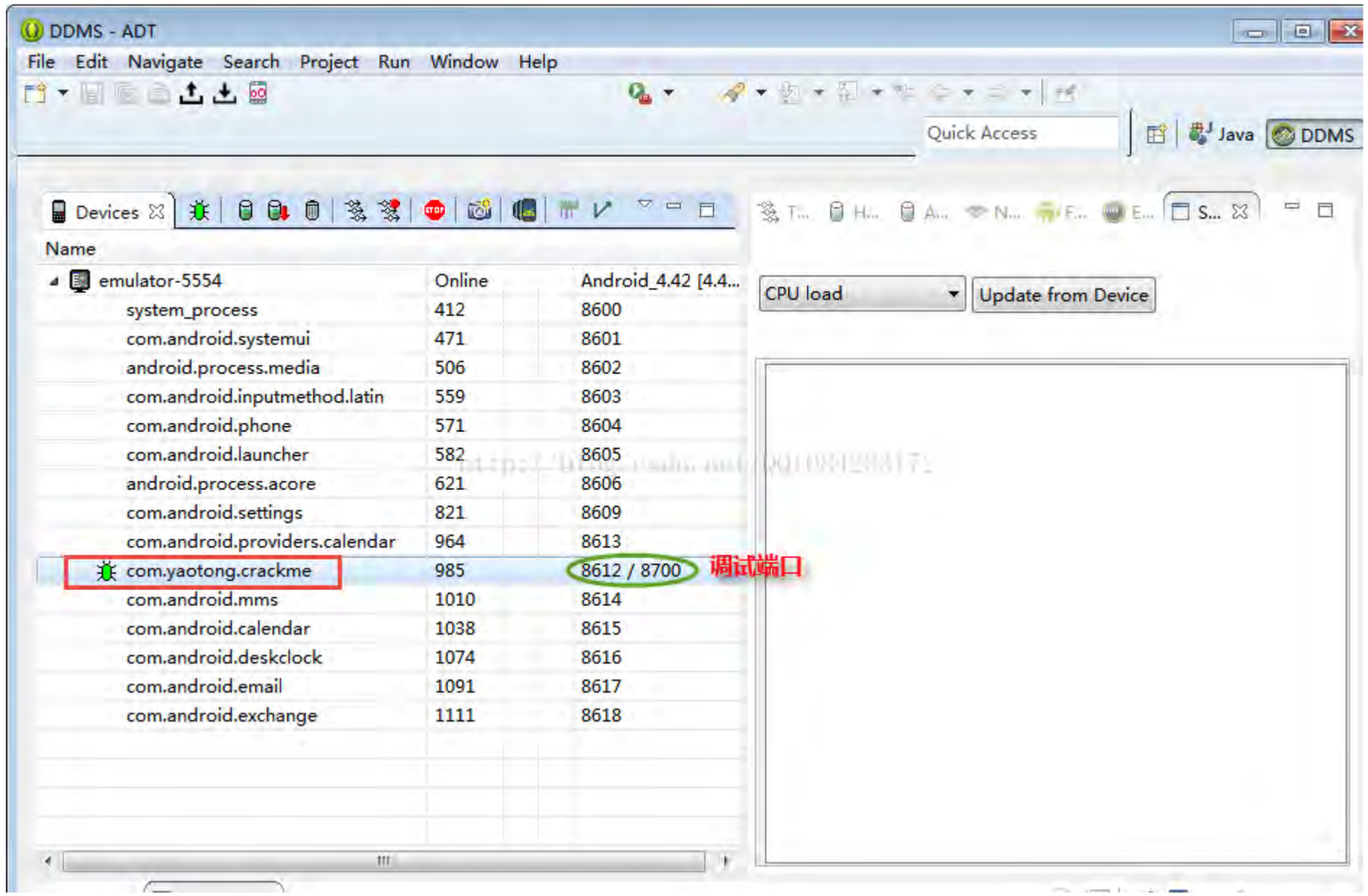

IDA - C:\Users\Fly2014\AppData\Local\Temp\ida95031.idb (app_process)
File Edit Jump Search View Debugger Options Windows Help
Remote ARMLinux/Androi
Library function Data Reg
IDA View-PC
libc.so:B6EFF798 ; ----
libc.so:B6EFF798
libc.so:B6EFF79C
libc.so:B6EFF79C ; ----
libc.so:B6EFF7A0 __fut
libc.so:B6EFF7A1
libc.so:B6EFF7A2
libc.so:B6EFF7A3
libc.so:B6EFF7A4 __fut
libc.so:B6EFF7A5
libc.so:B6EFF7A6
libc.so:B6EFF7A7
libc.so:B6EFF7A8
libc.so:B6EFF7A9
libc.so:B6EFF7AA
libc.so:B6EFF7AB
libc.so:B6EFF7AC
libc.so:B6EFF7AD
libc.so:B6EFF7AE
libc.so:B6EFF7AF
libc.so:B6EFF7B0
libc.so:B6EFF7B1
libc.so:B6EFF7B2
libc.so:B6EFF7B3
libc.so:B6EFF7B4
libc.so:B6EFF7B5
libc.so:B6EFF7B6
libc.so:B6EFF7B7
DCB 0xA0 ;
DCB 0xE3 ;
Quick debug view Ctrl+2
Debugger windows
Breakpoints
Watches
Tracing
Continue process F9
Attach to process...
Process options...
Pause process
Terminate process Ctrl+F2
Detach from process
Refresh memory
Take memory snapshot
Step into F7
Step over F8
Run until return Ctrl+F7
Run to cursor
Switch to source
Use source-level debugging
Open source file...
Debugger options...
Switch debugger...



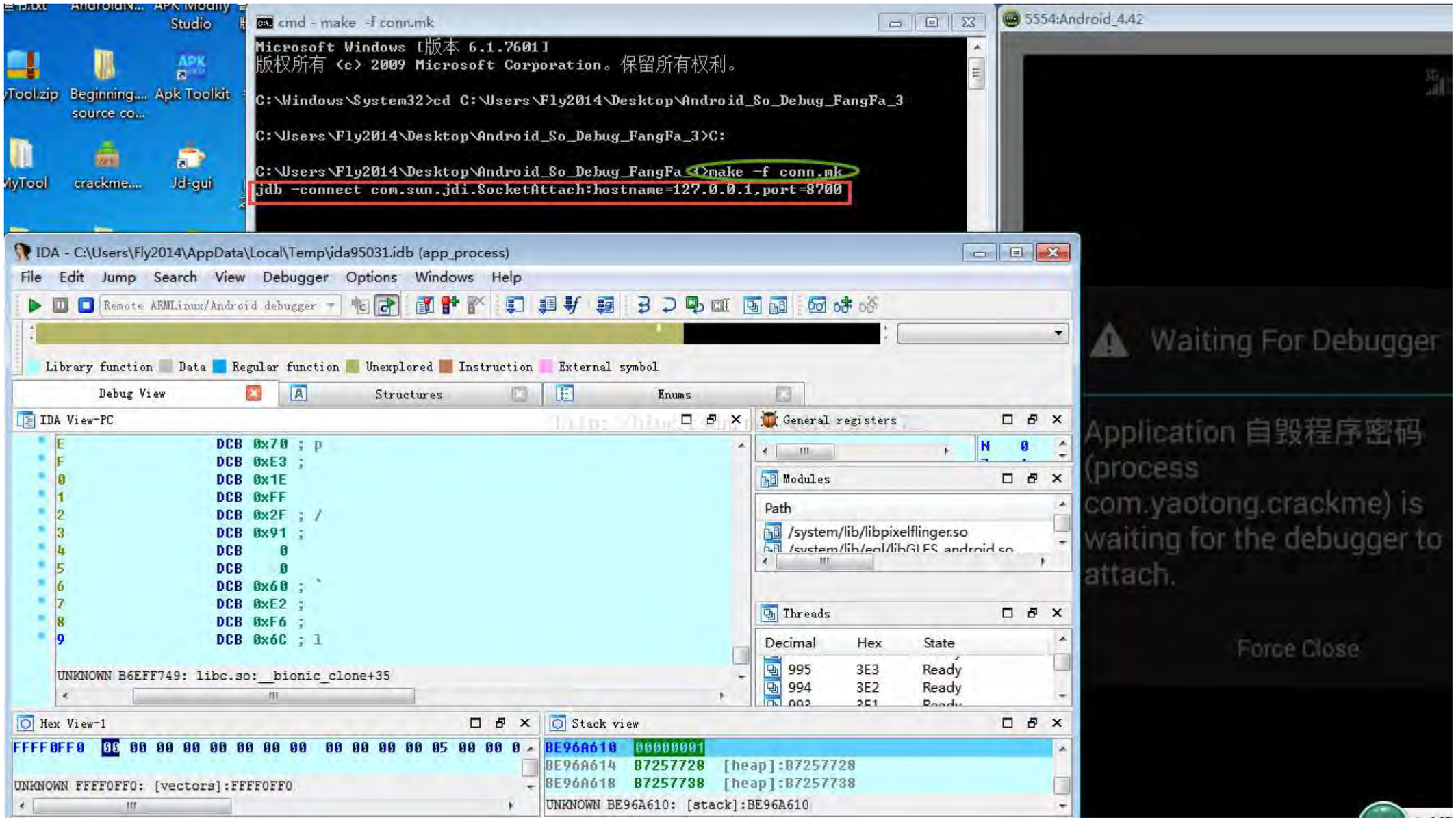
OK，IDA6.6 的设置完成，cmd 控制台不要关闭保持状态，IDA6.6 程序不要关闭保持状态，Eclipse 程序和 Android 模拟器也不能关闭继续保持状态，在后面的动态调试中还会用到。

第五步、再开启一个 cmd 控制台，在目标 apk 应用程序的 so 库的 JNI_OnLoad 函数处下断点

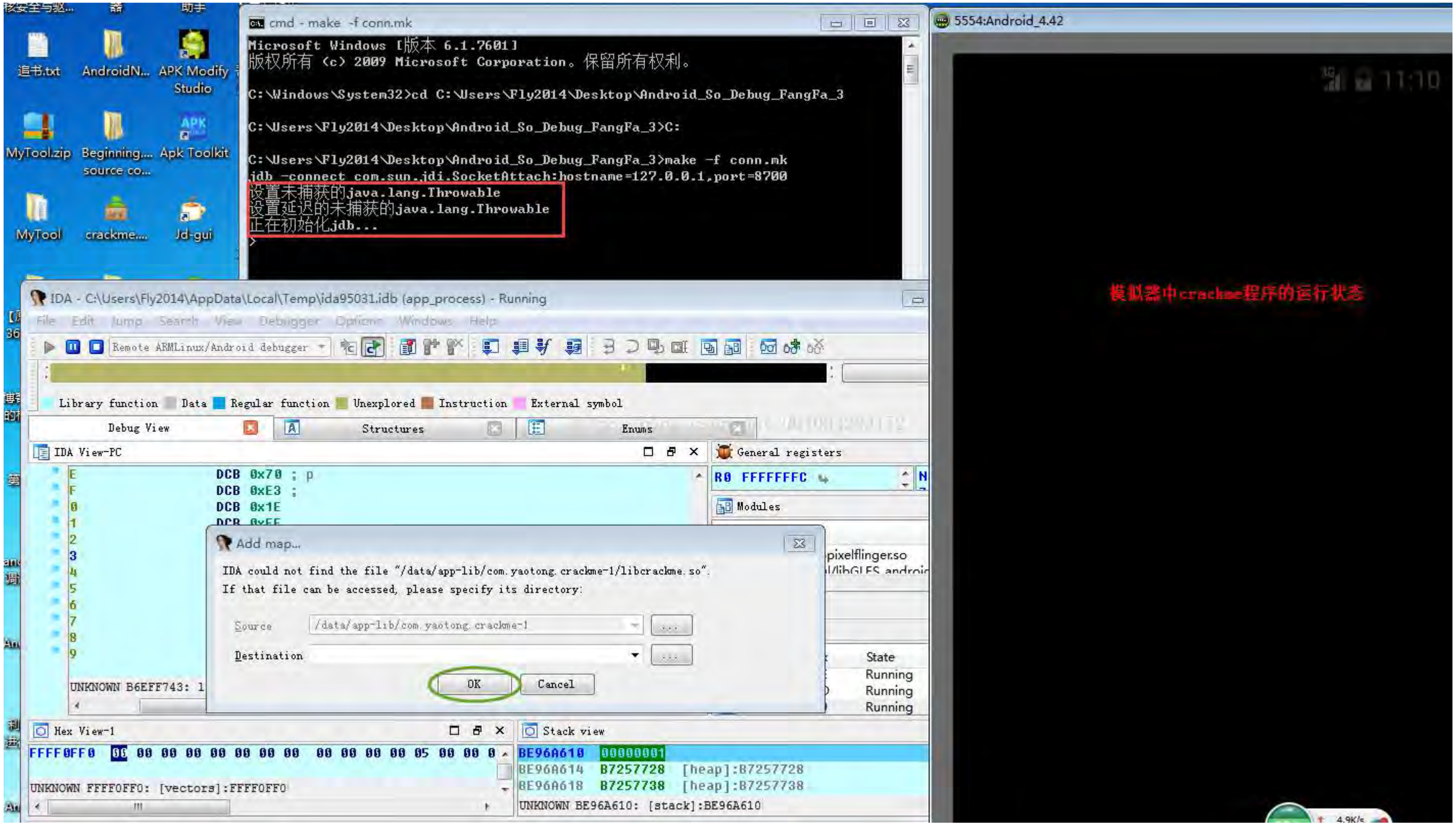
根据第四步中 com.yaotong.crackme **进程调试端口** 8612/8700 修改 conn.mk 文件中的 jdb 命令为 jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700，如下图：



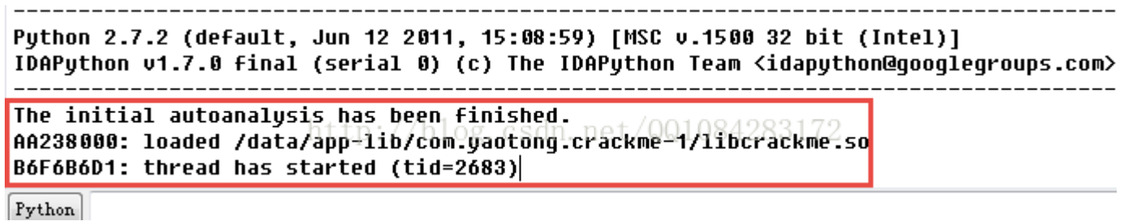
再开启一个 cmd 控制台，cd 命令进入到 conn.mk 所在的目录，执行 make -f conn.mk 命令，如下图：



连接成功后，在第三步的 IDA6.6 程序中按 F9 后 Android 模拟上的“waiting for debugger"提示会自动消失，这个时候应该已经断在新线程，或者加载 so 处，具体的状态如下图。不过有一点不太明白，在我的调试运行过程中 IDA6.6 出现了如下的提示，不知道是否出错，请指导的大神帮忙指点一下。

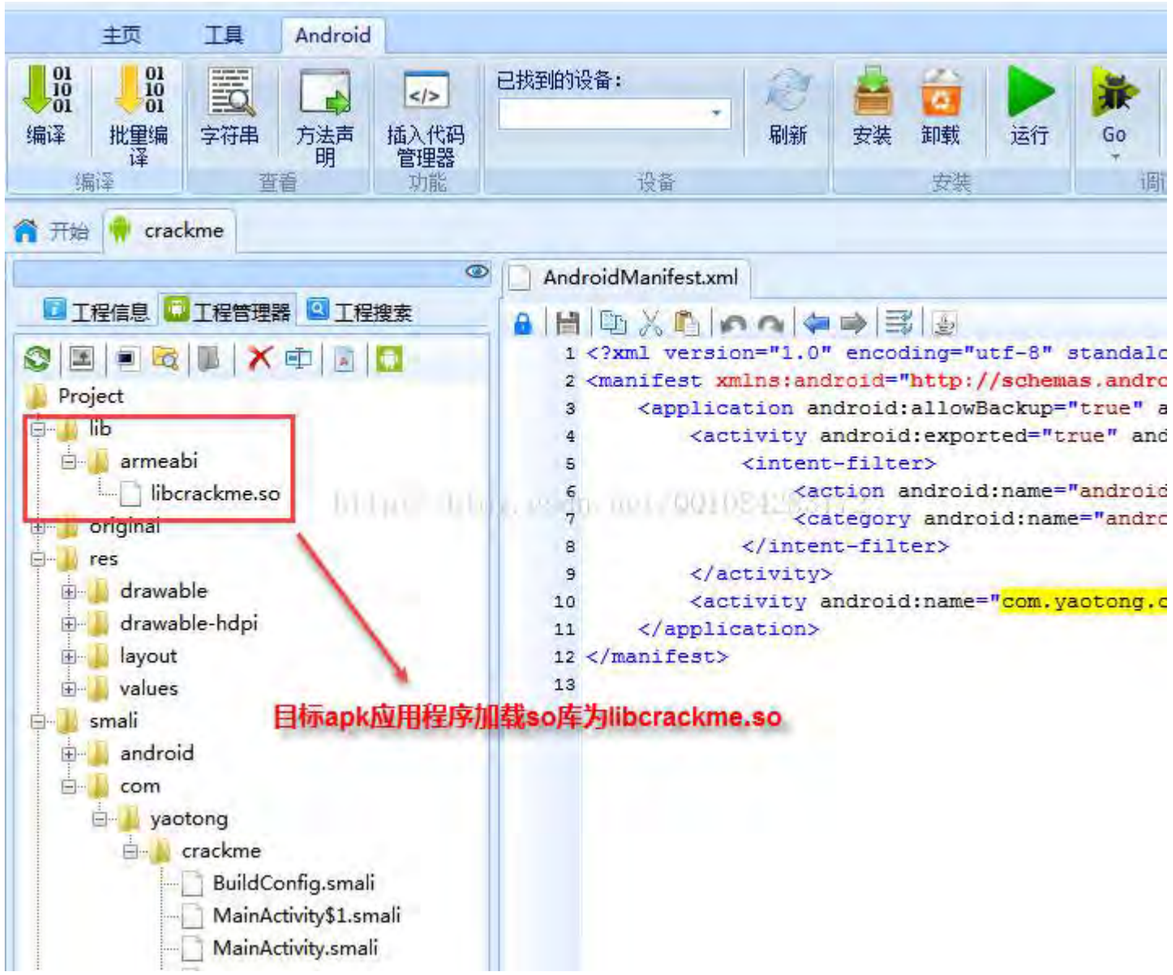


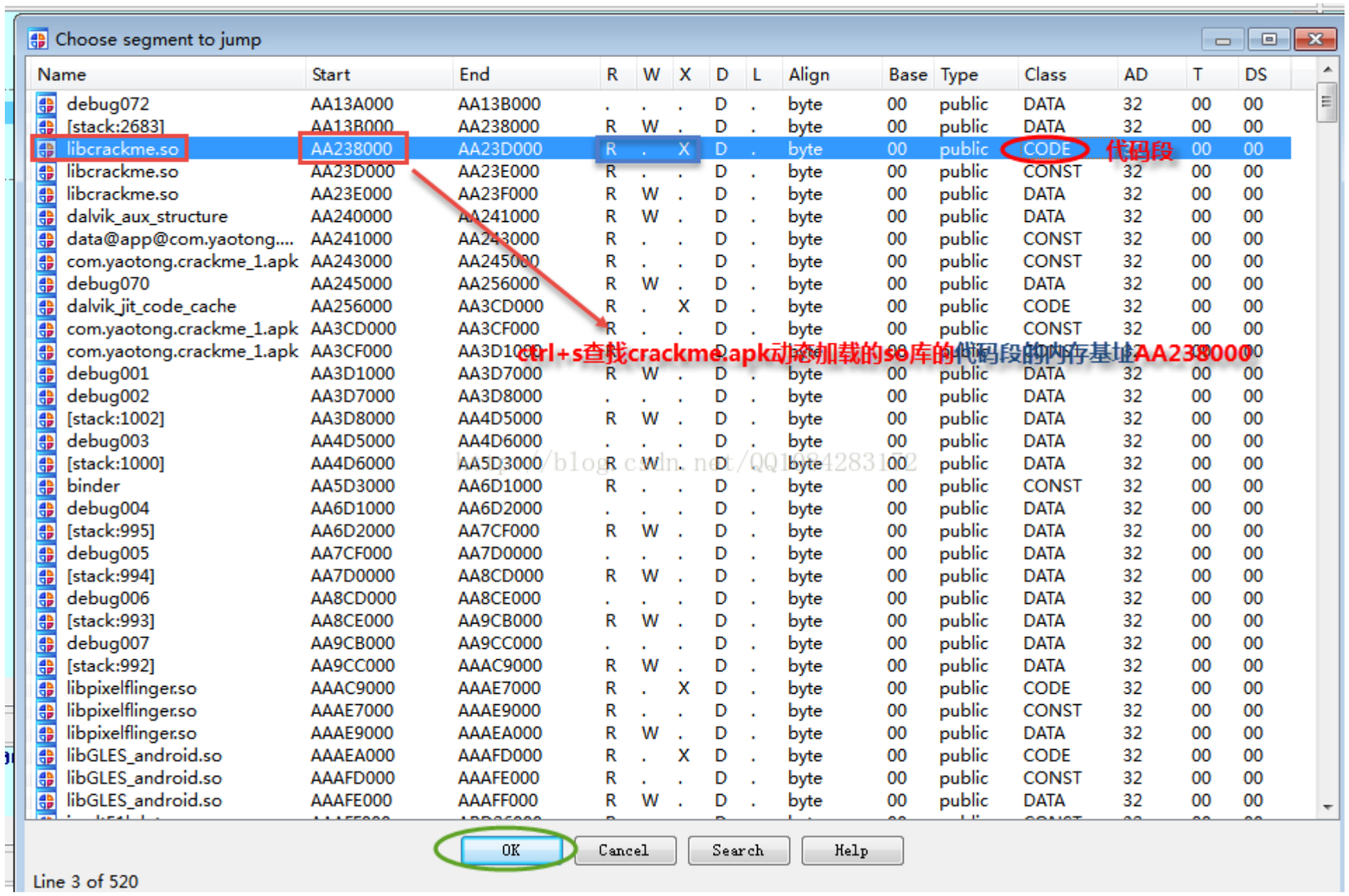
IDA6.6 下方的提示如下图，对于上图中的这个对话框我没有理会，直接一路**确定**迷迷糊糊的这个 Add map 对话框就被我给无视了。



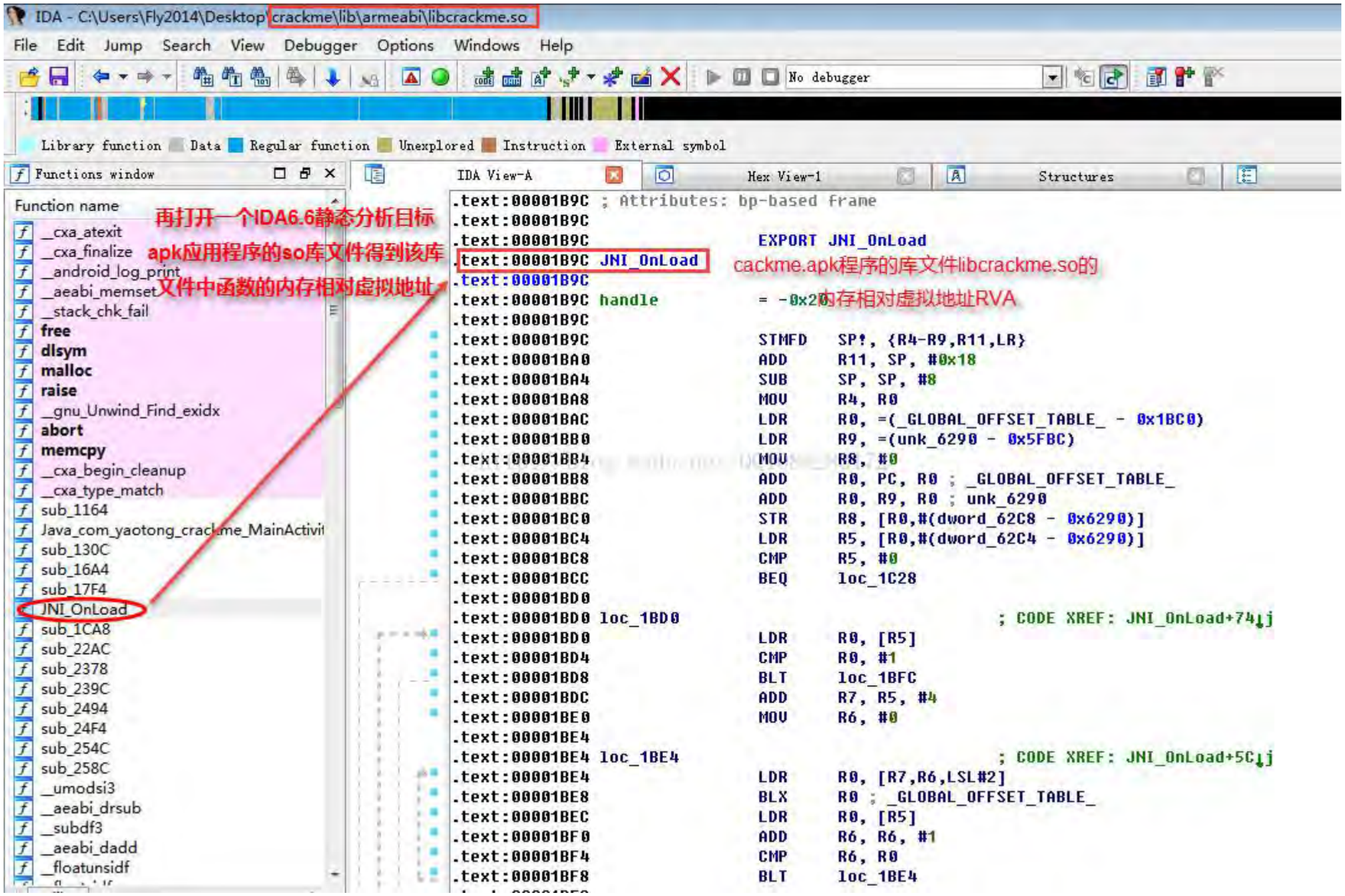
现在就可以在 IDA6.6 中按下快捷键 CTRL+S 来查看要调试的 so 是否已经加载了，如果**没有就 F9**，直到加载了为止；如果已经有了，就记下该 so 的 start 内存基地址，然后直接用**压缩软件**解压 crackme.apk 文件，得到 crackme.apk 文件要加载的 so 库，并且再**另开**一个 IDA6.6 静态分析该.so 库，找到 JNI_Onload 函数的内存相对虚拟地址（RVA）JNI_Onload_Offset，那么该 JNI_OnLoad 函数在内存中的真实地址为 so.start+

JNI_OnLoad_Offset（so 库的内存加载基址+JNI_Onload 函数的内存相对虚拟地址）。


主页
工具
Android
编译
批量编译
字符串
方法声明
插入代码管理器
已找到的设备:
刷新
安装
卸载
运行
Go
编译
查看
功能
设备
安装
开始
crackme
AndroidManifest.xml
工程信息
工程管理器
工程搜索
Project
lib
armeabi
libcrackme.so
original
res
drawable
drawable-hdpi
layout
values
smali
android
com
yaotong
crackme
BuildConfig.smali
MainActivity$1.smali
MainActivity.smali
1 <?xml version="1.0" encoding="utf-8" standalc
2 <manifest xmlns:android="http://schemas.andro
3 <application android:allowBackup="true" a
4 <activity android:exported="true" and
5 <intent-filter>
6 <action android:name="android
7 <category android:name="andro
8 </intent-filter>
9 </activity>
10 <activity android:name="com.yaotong.c
11 </application>
12 </manifest>
13
目标apk应用程序加载so库为libcrackme.so

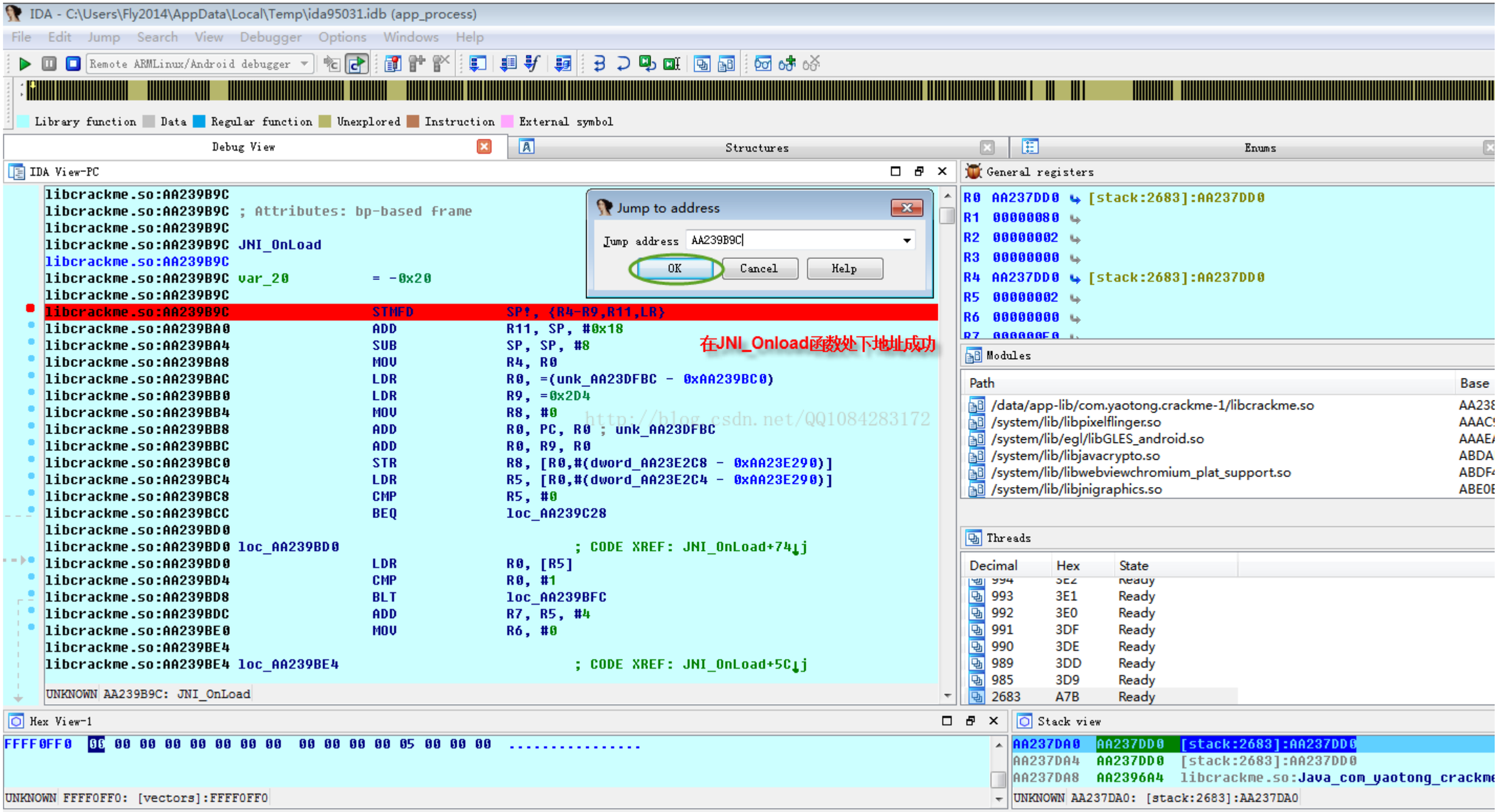


注意：在快捷键 CTRL+S 跳出的窗口中有几个个同名的 so 库，我们应当选择 class 类型为 CODE 即代码段的内存加载基址 **so.Start** 也就是权限为 RX 的这个，RX 一般是代码段，RW 一般是数据段。这里的 **so.Start=AA238000**.



通过另开的 IDA6.6 静态分析目标 apk 应用程序的 so 库文件得到 libcrackme.so 文件中 JNI_Onload 函数的内存相对虚拟地址为 00001B9C 即 **JNI_OnLoad_Offset=00001B9C**，因此 JNI_Onload 函数在内存中真实地址为 so.start+

JNI_OnLoad_Offset=AA238000+00001B9C=AA239B9C.得到真实地址后，在**附加目标进程的 IDA6.6** 中按下快捷键 **G** 跳转到地址 **AA239B9C**，然后按下快捷键 **F2** 就完成在 JNI_OnLoad 函数入口处下断点了。



OK，在 JNI_Onload 函数处下断点成功，下面就可以进行 SO 文件的动态调试 F7、F8 单步或者直接 F9 运行 JNI_Onload 函数断点处。

在 apk 应用程序的 JNI_Onload 函数进行下断点，对于从 arm 汇编的角度来产看 apk 程序的本地方法做了什么操作还是很有效果的，但是这种方法也不是经常有效的。在 Android 的 JNI 编程也可以不使用 JNI_Onload 函数进行 Android 的 JNI 编程。

这篇调试笔记也花费了几个小时的时间，但是结果很遗憾的是 F9 运行跑飞了，估计是我在进行 IDA6.6 的操作的那个 Add map 对话框出错导致的吧，希望大神能告诉其中的原因，也请知道的大神告知 Android'比较好的动态调试的方法以及 adb shell am start -D -n com.yaotong.crackme/.MainActivity 命令不添加导出选项，启动出错的原因。

参考网址：

http://www.cnblogs.com/wanyuanchun/p/3760825.html?utm_source=tuicool

http://www.52pojie.cn/forum.php?mod=viewthread&tid=293648

2015/4/12 2:08

IDA 远程调试 so 库 JNI_Onload 函数

JNI_OnLoad 函数大概功能就是在程序加载 so 的时候，会执行 JNI_OnLoad 函数，做一系列的准备工作。
很多时候，程序猿们会将一些重要信息放在此函数中，而不是通过某种事件来重复触发。包括说将反调试函数放置在此函数中。因此，调试手段发生了改变，上述调试方法基本上被淘汰。

1. 静态分析，找到 JNI_OnLoad 函数的偏移

```
.text:00001B9C
.text:00001B9C                  EXPORT JNI_OnLoad
.text:00001B9C JNI_OnLoad
.text:00001B9C
.text:00001B9C handle           = -0x20
.text:00001B9C
.text:00001B9C                  STMFD   SP!, {R4-R9,R11,LR}
.text:00001BA0                  ADD     R11, SP, #0x18
.text:00001BA4                  SUB     SP, SP, #8
.text:00001BA8                  MOV     R4, R0
.text:00001BAC                  LDR     R0, =(_GLOBAL_OFFSET_TABLE_ - 0x1BC0)
.text:00001BB0                  LDR     R9, =(unk_6290 - 0x5FBC)
.text:00001BB4                  MOV     R8, #0
.text:00001BB8                  ADD     R0, PC, R0 ; _GLOBAL_OFFSET_TABLE_
.text:00001BBC                  ADD     R0, R9, R0 ; unk_6290
.text:00001BC0                  STR     R8, [R0,#(dword_62C8 - 0x6290)]
```

2. 执行 android_server

```
D:\IDA-pro-plus-6.5-x86-arm1.7-By-Dalao\dbgsrv>adb push android_server6.5 /data/
local/tmp
1048 KB/s (566944 bytes in 0.528s)

D:\IDA-pro-plus-6.5-x86-arm1.7-By-Dalao\dbgsrv>adb shell
# cd /data/local/tmp
# chmod 777 android_server6.5
# ./android_server6.5
IDA Android 32-bit remote debug server(ST) v1.17. Hex-Rays (c) 2004-2013
Listening on port #23946...
```

3.端口转发

```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

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

C:\Users\Administrator>_
```

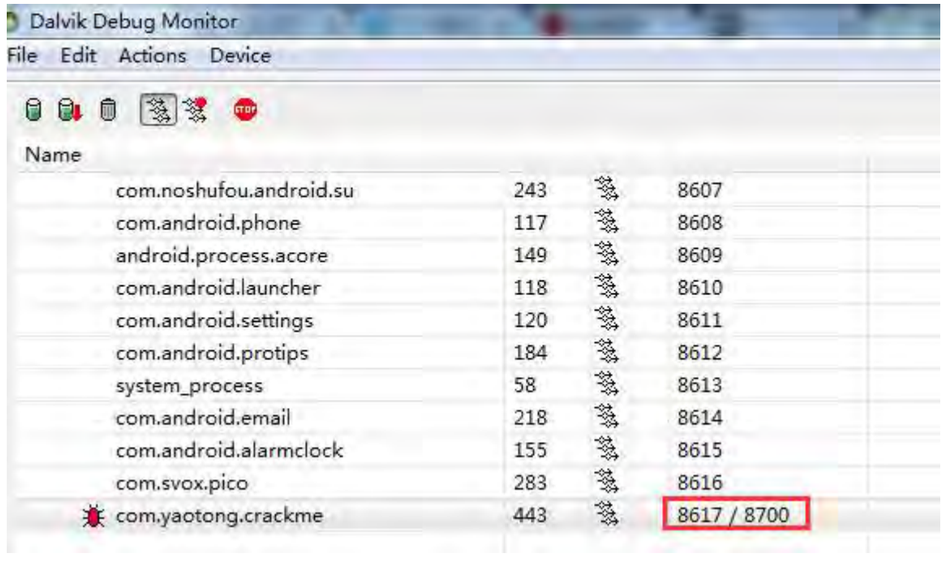
4. 以调试模式启动程序

adb shell am start -D -n com.example.mytestcm/.MainActivity

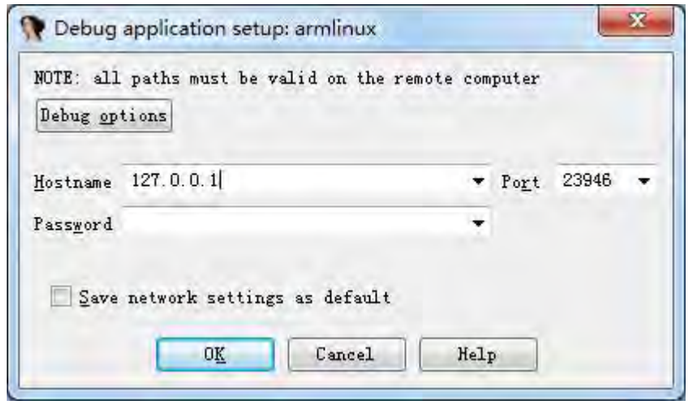
此时，手机界面会出现 Waiting For Debugger 页面

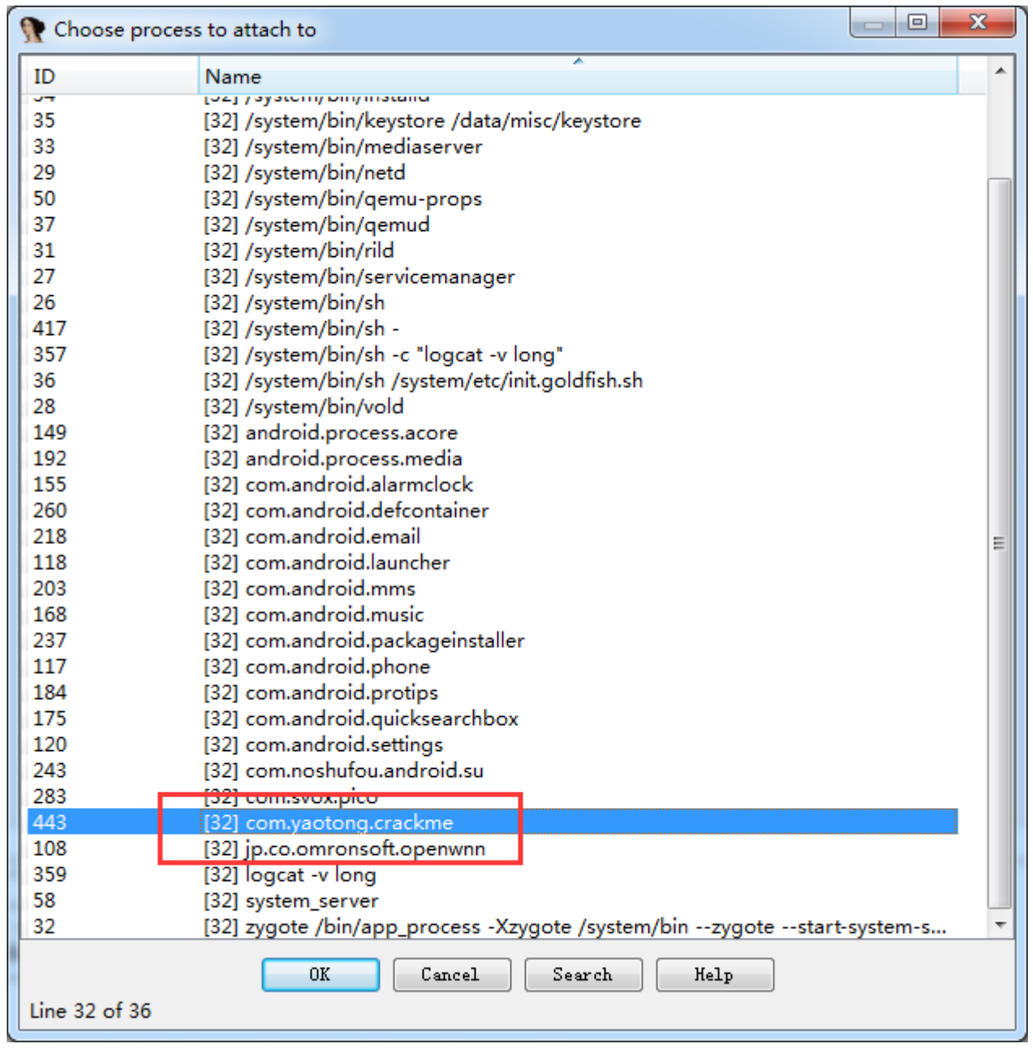
```
C:\Users\Administrator>adb shell am start -D -n com.yaotong.crackme/.MainActiv
y
Starting: Intent { cmp=com.yaotong.crackme/.MainActivity }

C:\Users\Administrator>
```

5.打开 ddms 或者 Eclipse （必要，为了使用 jdb 命令），获取应用程序的端口号

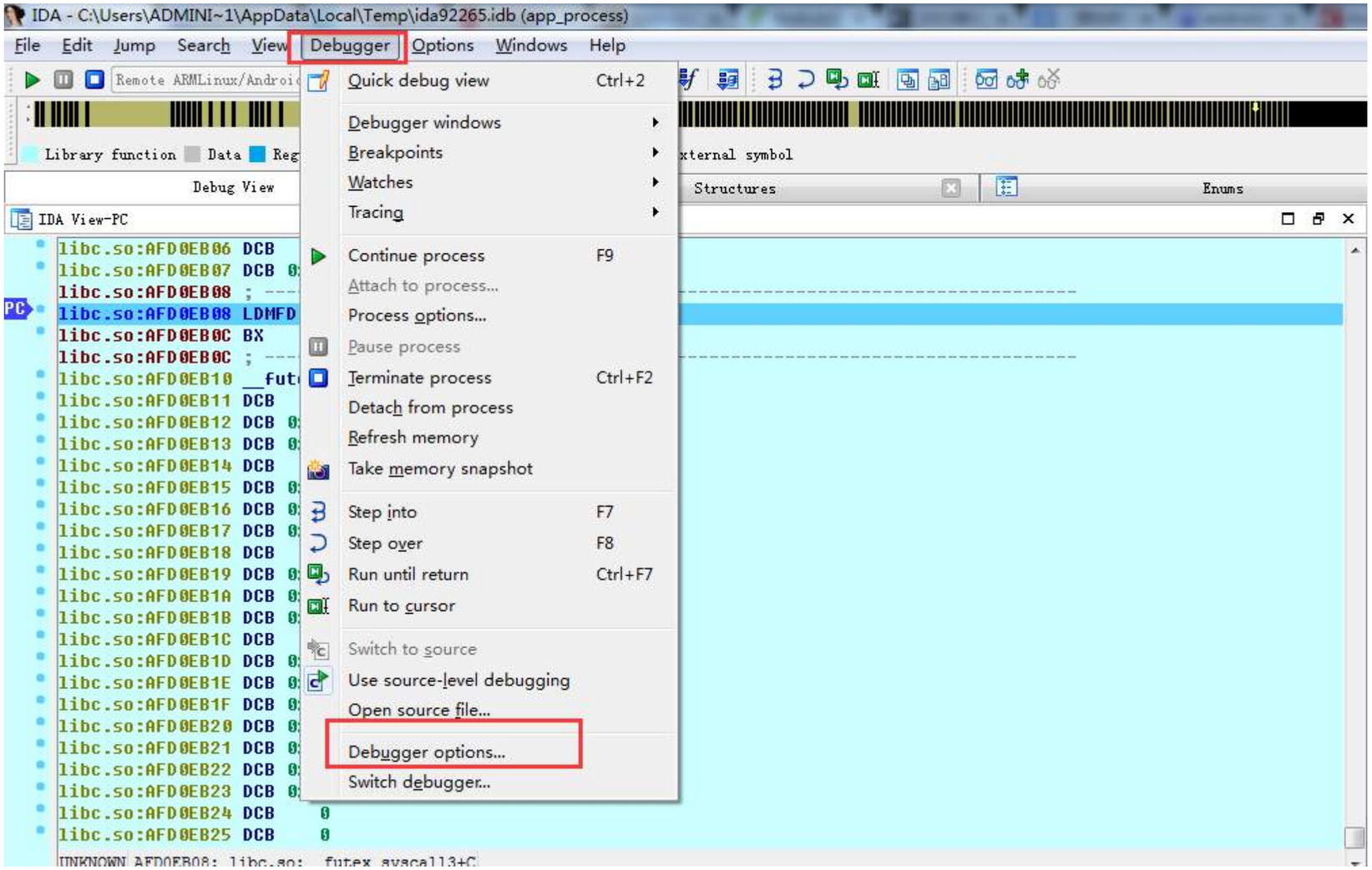

Dalvik Debug Monitor
File Edit Actions Device
Name
com.noshufou.android.su 243 8607
com.android.phone 117 8608
android.process.acore 149 8609
com.android.launcher 118 8610
com.android.settings 120 8611
com.android.protips 184 8612
system_process 58 8613
com.android.email 218 8614
com.android.alarmclock 155 8615
com.svox.pico 283 8616
com.yaotong.crackme 443 8617 / 8700

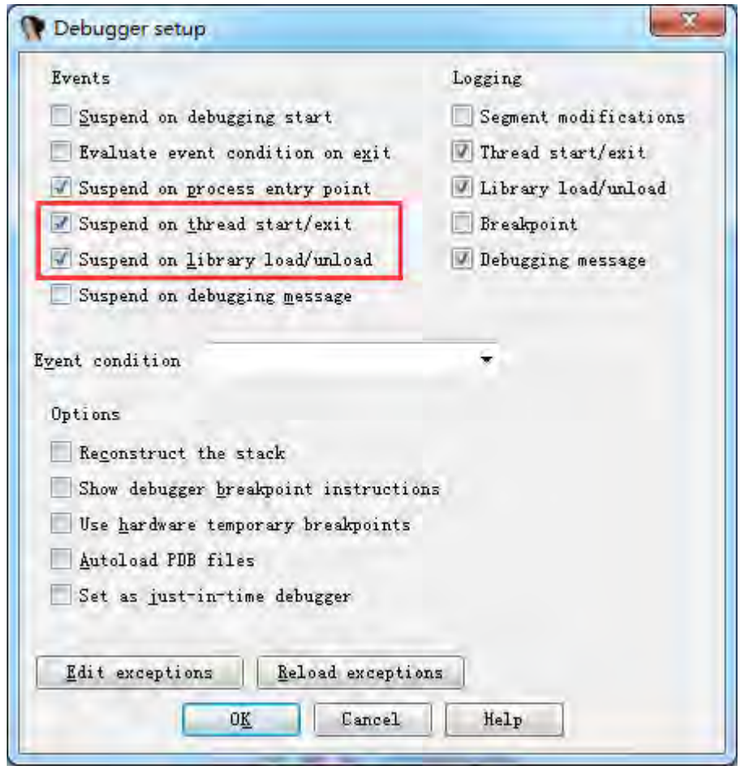
6. IDA 附加


Debug application setup: armlinux
NOTE: all paths must be valid on the remote computer
Debug options
Hostname 127.0.0.1
Port 23946
Password
Save network settings as default
OK
Cancel
Help

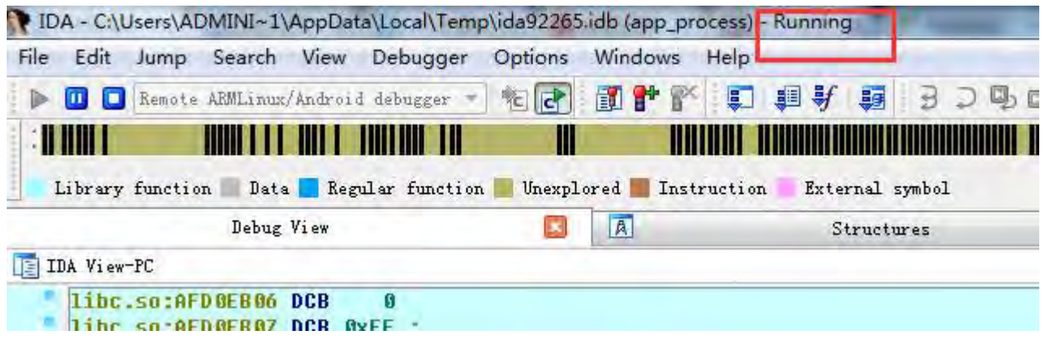

Choose process to attach to
ID
Name
35 [32] /system/bin/keystore /data/misc/keystore
33 [32] /system/bin/mediaserver
29 [32] /system/bin/netd
50 [32] /system/bin/qemu-props
37 [32] /system/bin/qemud
31 [32] /system/bin/rild
27 [32] /system/bin/servicemanager
26 [32] /system/bin/sh
417 [32] /system/bin/sh -
357 [32] /system/bin/sh -c "logcat -v long"
36 [32] /system/bin/sh /system/etc/init.goldfish.sh
28 [32] /system/bin/vold
149 [32] android.process.acore
192 [32] android.process.media
155 [32] com.android.alarmclock
260 [32] com.android.defcontainer
218 [32] com.android.email
118 [32] com.android.launcher
203 [32] com.android.mms
168 [32] com.android.music
237 [32] com.android.packageinstaller
117 [32] com.android.phone
184 [32] com.android.protips
175 [32] com.android.quicksearchbox
120 [32] com.android.settings
243 [32] com.noshufou.android.su
443 [32] com.yaotong.crackme
108 [32] jp.co.omronsoft.openwnn
359 [32] logcat -v long
58 [32] system_server
32 [32] zygote /bin/app_process -Xzygote /system/bin --zygote --start-system-s...
OK
Cancel
Search
Help
Line 32 of 36

7. 设置调试选项


IDA - C:\Users\ADMINI~1\AppData\Local\Temp\ida92265.idb (app_process)
File Edit Jump Search View Debugger Options Windows Help
Quick debug view Ctrl+2
Debugger windows
Breakpoints
Watches
Tracing
Continue process F9
Attach to process...
Process options...
Pause process
Terminate process Ctrl+F2
Detach from process
Refresh memory
Take memory snapshot
Step into F7
Step over F8
Run until return Ctrl+F7
Run to cursor
Switch to source
Use source-level debugging
Open source file...
Debugger options...
Switch debugger...
Library function Data
Debug View
Structures
Enums
IDA View-PC
libc.so:AFD0EB06 DCB
libc.so:AFD0EB07 DCB
libc.so:AFD0EB08 ;
libc.so:AFD0EB08 LDMFD
libc.so:AFD0EB0C BX
libc.so:AFD0EB0C ;
libc.so:AFD0EB10 __fut
libc.so:AFD0EB11 DCB
libc.so:AFD0EB12 DCB
libc.so:AFD0EB13 DCB
libc.so:AFD0EB14 DCB
libc.so:AFD0EB15 DCB
libc.so:AFD0EB16 DCB
libc.so:AFD0EB17 DCB
libc.so:AFD0EB18 DCB
libc.so:AFD0EB19 DCB
libc.so:AFD0EB1A DCB
libc.so:AFD0EB1B DCB
libc.so:AFD0EB1C DCB
libc.so:AFD0EB1D DCB
libc.so:AFD0EB1E DCB
libc.so:AFD0EB1F DCB
libc.so:AFD0EB20 DCB
libc.so:AFD0EB21 DCB
libc.so:AFD0EB22 DCB
libc.so:AFD0EB23 DCB
libc.so:AFD0EB24 DCB 0
libc.so:AFD0EB25 DCB 0
UNKNOWN AFD0EB08: libc.so: __futex_syscall3+C

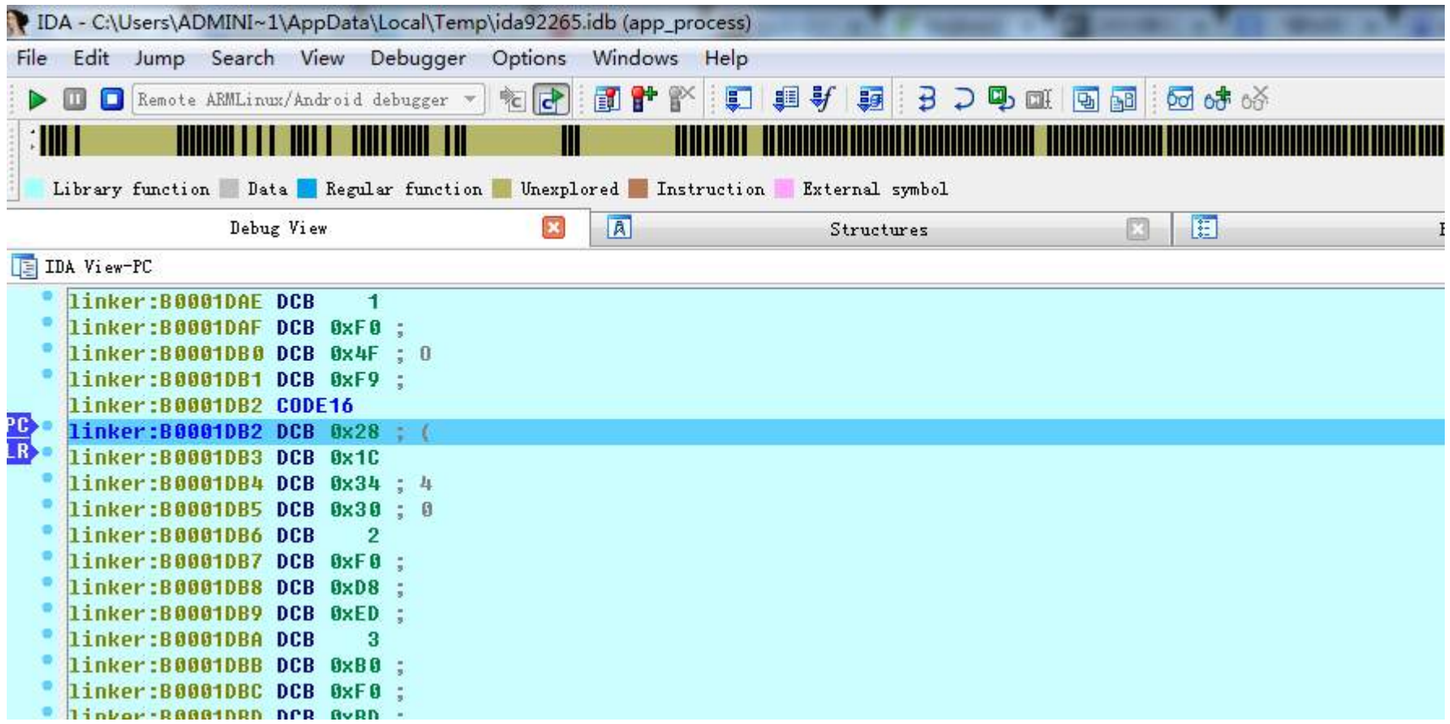


8. F9 运行程序



在命令行中执行：jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700 其中 port=8700 是从 ddms 中看到的。

此时程序会断下来



9. 下断点，首地址加偏移，跟上一篇远程调试的方法一样

Choose segment to jump

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	T	DS
libGLESv2.so	AC100000	AC104000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libandroid_runtime.so	AD36D000	AD375000	R	W	X	D	.	byte	00	public	CODE	32	00	00
libandroid_runtime.so	AD300000	AD36D000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libbinder.so	A821F000	A8225000	R	W	X	D	.	byte	00	public	CODE	32	00	00
libbinder.so	A8200000	A821F000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libc.so	AFD3F000	AFD42000	R	W	X	D	.	byte	00	public	CODE	32	00	00
libc.so	AFD00000	AFD3F000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libcamera_client.so	ABA90000	ABA93000	R	W	X	D	.	byte	00	public	CODE	32	00	00
libcamera_client.so	ABA80000	ABA90000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libcrackme.so	80905000	80907000	R	W	X	D	.	byte	00	public	CODE	32	00	00
libcrackme.so	80900000	80905000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libcrypto.so	AF08D000	AF09F000	R	W	X	D	.	byte	00	public	CODE	32	00	00
libcrypto.so	AF000000	AF08D000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libcutils.so	AF90E000	AF90F000	R	W	X	D	.	byte	00	public	CODE	32	00	00
libcutils.so	AF900000	AF90E000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libdrm1.so	9EA08000	9EA09000	R	W	X	D	.	byte	00	public	CODE	32	00	00
libdrm1.so	9EA00000	9EA08000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libdvm.so	80886000	80889000	R	W	X	D	.	byte	00	public	CODE	32	00	00
libdvm.so	80800000	80886000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libemoji.so	A7502000	A7503000	R	W	X	D	.	byte	00	public	CODE	32	00	00
libemoji.so	A7500000	A7502000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libexif.so	ABB09000	ABB0A000	R	W	X	D	.	byte	00	public	CODE	32	00	00
libexif.so	ABB00000	ABB09000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libexpat.so	A8A14000	A8A16000	R	W	X	D	.	byte	00	public	CODE	32	00	00
libexpat.so	A8A00000	A8A14000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libhardware.so	A7F01000	A7F02000	R	W	X	D	.	byte	00	public	CODE	32	00	00
libhardware.so	A7F00000	A7F01000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libhardware_legacy.so	A7E05000	A7E06000	R	W	X	D	.	byte	00	public	CODE	32	00	00
libhardware_legacy.so	A7E00000	A7E05000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libicudata.so	804FB000	804FC000	R	W	X	D	.	byte	00	public	CODE	32	00	00
libicudata.so	80000000	804FB000	R	.	X	D	.	byte	00	public	CODE	32	00	00
libicui18n.so	AD9E0000	AD9E4000	R	W	X	D	.	byte	00	public	CODE	32	00	00

OK Cancel Search Help

Line 76 of 172

```
libcrackme.so:80901B9C
libcrackme.so:80901B9C ; Attributes: bp-based frame
libcrackme.so:80901B9C
libcrackme.so:80901B9C JNI_OnLoad
libcrackme.so:80901B9C
libcrackme.so:80901B9C var_20= -0x20
libcrackme.so:80901B9C
libcrackme.so:80901B9C STMFD           SP!, {R4-R9,R11,LR}
libcrackme.so:80901BA0 ADD             R11, SP, #0x18
libcrackme.so:80901BA4 SUB             SP, SP, #8
libcrackme.so:80901BA8 MOV             R4, R0
libcrackme.so:80901BAC LDR             R0, =(unk_80905FBC - 0x80901BC0)
libcrackme.so:80901BB0 LDR             R9, =0x2D4
libcrackme.so:80901BB4 MOV             R8, #0
libcrackme.so:80901BB8 ADD             R0, PC, R0 ; unk_80905FBC
libcrackme.so:80901BBC ADD             R0, R9, R0
libcrackme.so:80901BC0 STR             R8, [R0,#(dword_809062C8 - 0x80906290)]
libcrackme.so:80901BC4 LDR             R5, [R0,#(dword_809062C4 - 0x80906290)]
libcrackme.so:80901BC8 CMP             R5, #0
libcrackme.so:80901BCC BEQ             loc_80901C28
libcrackme.so:80901BD0
```

下好断点之后，直接 F9 运行吧，就能断在 JNI_OnLoad 函数处~

```
libcrackme.so:80901B9C ; Attributes: bp-based frame
libcrackme.so:80901B9C
libcrackme.so:80901B9C JNI_OnLoad
libcrackme.so:80901B9C
libcrackme.so:80901B9C var_20= -0x20
libcrackme.so:80901B9C
R6
PC
libcrackme.so:80901B9C STMFD           SP!, {R4-R9,R11,LR}
libcrackme.so:80901BA0 ADD             R11, SP, #0x18
libcrackme.so:80901BA4 SUB             SP, SP, #8
libcrackme.so:80901BA8 MOV             R4, R0
libcrackme.so:80901BAC LDR             R0, =(unk_80905FBC - 0x80901BC0)
libcrackme.so:80901BB0 LDR             R9, =0x2D4
libcrackme.so:80901BB4 MOV             R8, #0
libcrackme.so:80901BB8 ADD             R0, PC, R0 ; unk_80905FBC
libcrackme.so:80901BBC ADD             R0, R9, R0
libcrackme.so:80901BC0 STR             R8, [R0,#(dword_809062C8 - 0x80906290)]
```

当这种调试手法出现之后，将特殊函数，或者反调试函数放在 JNI_OnLoad 中也不是那么的安全了。此时，程序猿们通过分析系统对 SO 文件的加载 链接过程发现，JNI_OnLoad 函数并不是最开始执行的。在 JNI_OnLoad 函数执行之前，还会执行 init 段和 init_array 中的一系列 函数。

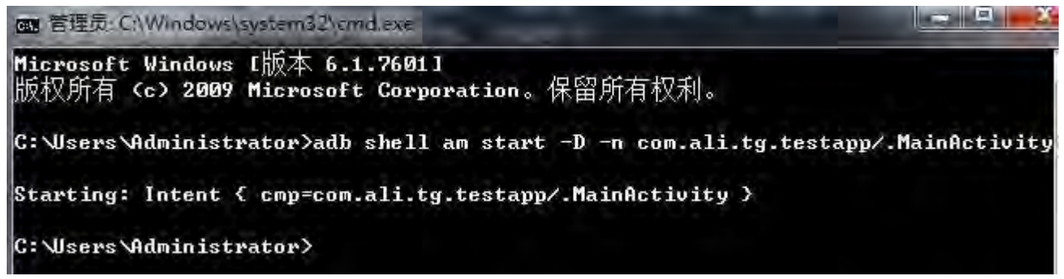
因此，现在的调试方法，都是将断点下在 init_array 中~
至于下断点的方法，可以类比于在 JNI_OnLoad 中下断点的方法，在 init_array 的函数中下断点。还有一种方法便是通过在 linker 模块中，通过对其中函数下断点，然后也能单步到 init_array 中

http://www.cnblogs.com/dacainiao/p/5101902.html

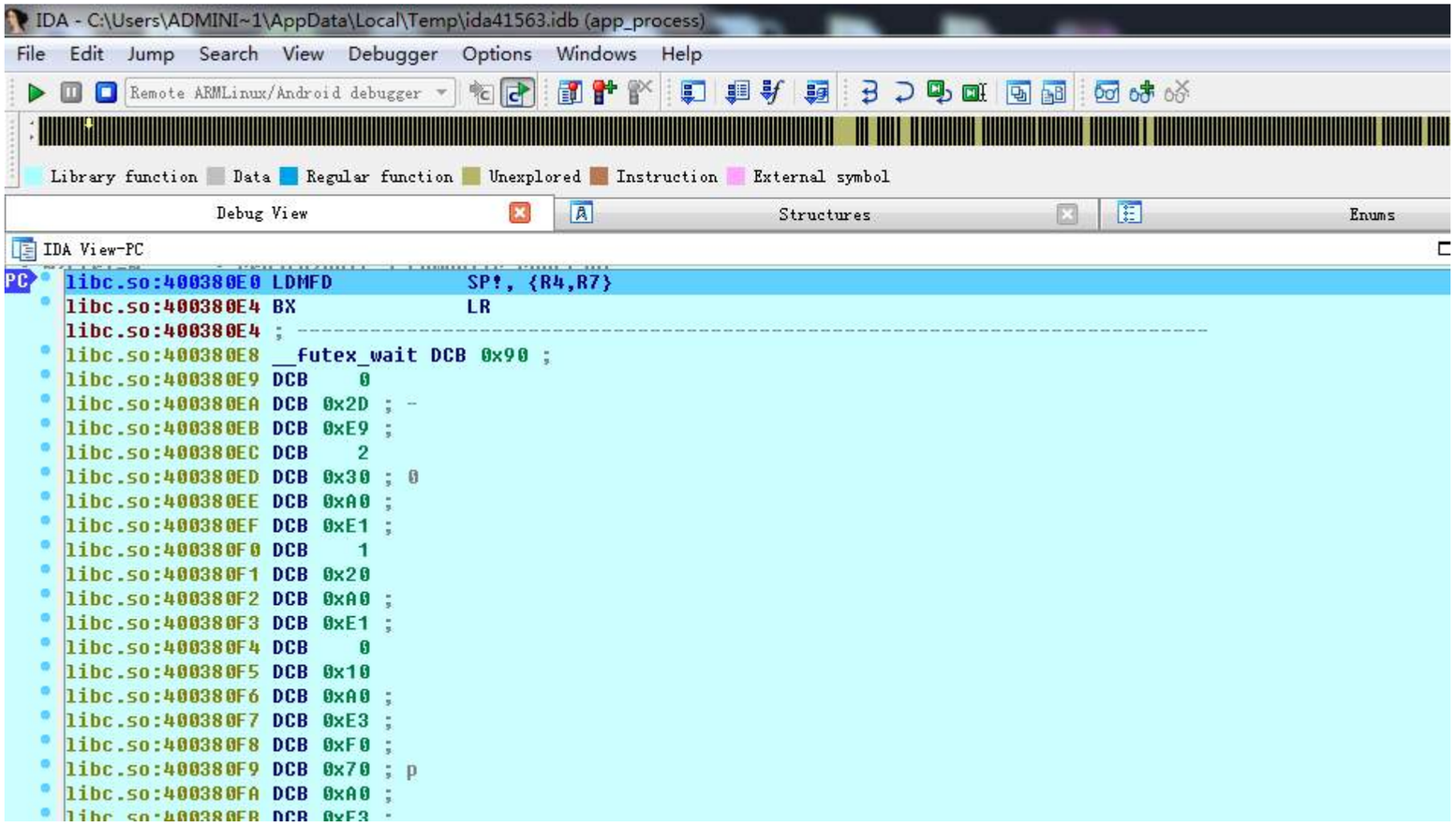
IDA 远程调试 在内存中 dump Dex 文件

http://www.cnblogs.com/dacainiao/p/5101902.html

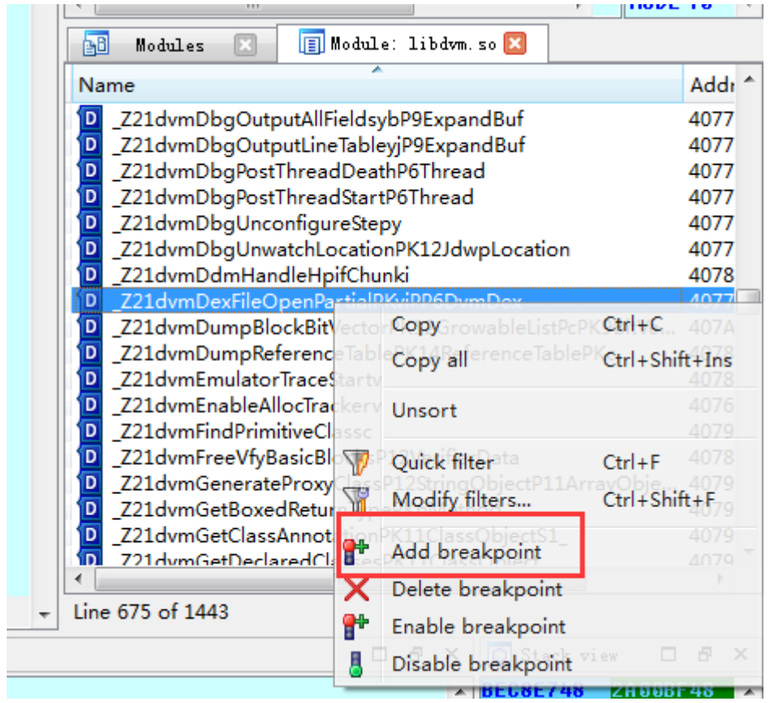
1. 首先使用调试 JNI_OnLoad 函数的方法，先将 apk 以调试状态挂起，使用 IDA 附加上去。




5554:Android4.2.2
3G
7:56
jscrack
Waiting For Debugger
Application jscrack (process com.ali.tg.testapp) is waiting for the debugger to attach.
Force Close



2. 然后在 libdvm.so 中的 dvmDexFileOpenPartial 函数上下一个断点



3. 然后我们点击继续运行，程序就会在 dvmDexFileOpenPartial()这个函数处暂停，R0 寄存器指向的地址就是 dex 文件在内存中的地址，R1 寄存器就是 dex 文件的大小

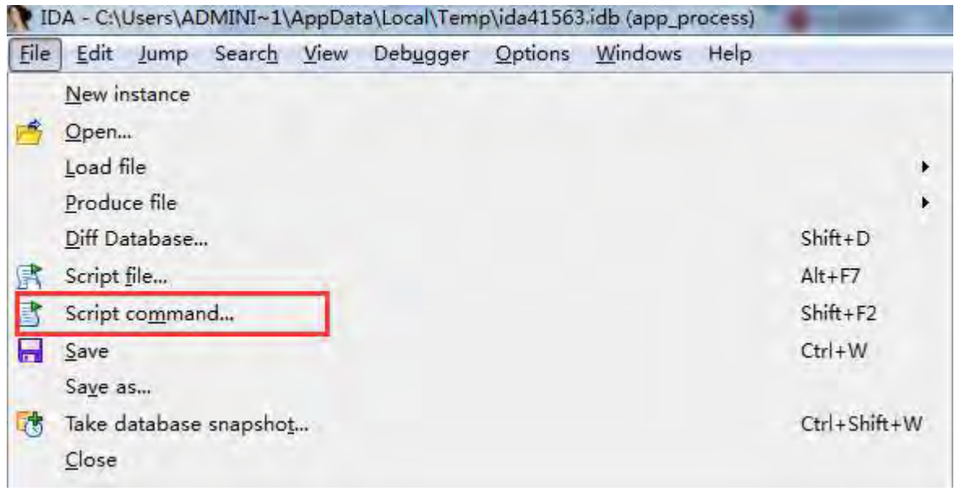
General registers

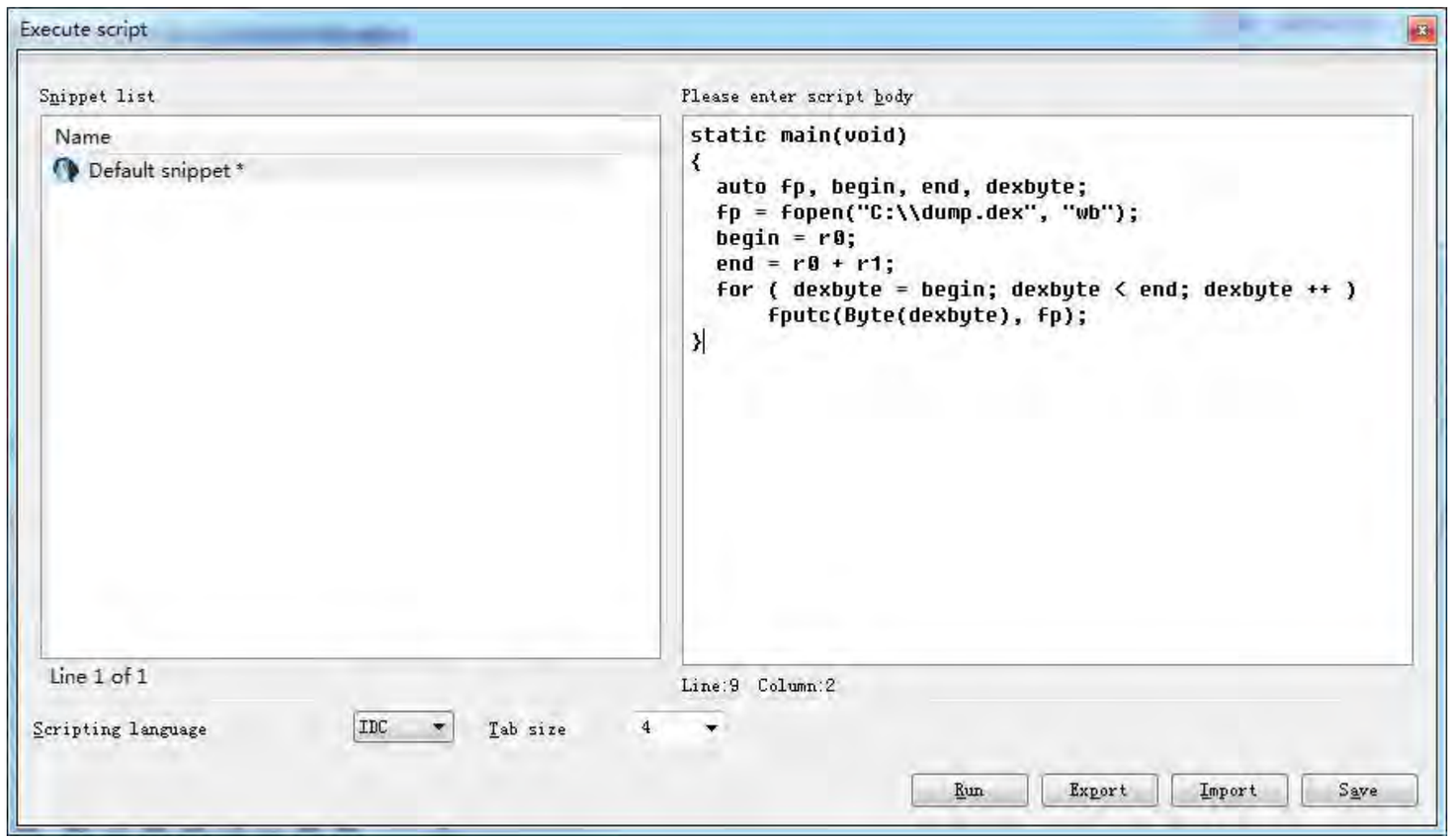
```
R0  4936B008 debug066:4936B008
R1  000941FC
R2  BEC8E2CC [stack]:BEC8E2CC
R3  5AA8F10E
R4  000941FC
R5  4936B008 debug066:4936B008
R6  00000000
R7  00000000
R8  BEC8E5AC [stack]:BEC8E5AC
R9  BEC8E370 [stack]:BEC8E370
R10 000941FC
R11 492A30D4 libmobisec.so:_ZN3ali10g_fil
R12 00000000
SP  BEC8E290 [stack]:BEC8E290
LR  4078A219 libdvm.so:_Z17dvmVerifyCodeF
PC  40773C04 _Z21dvmDexFileOpenPartialPKv
PSR 60000030
```

Hex View-1

```
4936AFC8  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
4936AFD8  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
4936AFE8  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
4936AFF8  00 00 00 00 00 00 00 00  00 00 00 00 F0 4F 00 00  ............ð...
4936B008  64 65 78 0A 30 33 35 00  6C 50 34 D8 82 04 15 79  dex.035.lP4....y
4936B018  EB 32 41 72 B7 D9 CE A3  6D 51 E6 41 06 56 BC 9D  .2Ar....mQ.A.V..
4936B028  FC 41 09 00 70 00 00 00  78 56 34 12 00 00 00 00  .A..p...xV4.....
4936B038  00 00 00 00 20 41 09 00  F0 18 00 00 70 00 00 00  .... A......p...
4936B048  02 04 00 00 30 64 00 00  2A 05 00 00 38 74 00 00  ....0d..*...8t..
4936B058  96 06 00 00 30 B2 00 00  08 19 00 00 E0 E6 00 00  ....0.......à...
4936B068  86 02 00 00 20 AF 01 00  1C 42 07 00 E0 FF 01 00  .... ....B....à..
4936B078  E0 FF 01 00 E2 FF 01 00  E5 FF 01 00 E9 FF 01 00  à...â...å...é...
4936B088  EF FF 01 00 F4 FF 01 00  0B 00 02 00 33 00 02 00  ï...ô.......3...
4936B098  43 00 02 00 66 00 02 00  80 00 02 00 9D 00 02 00  C...f...€.......
4936B0A8  BE 00 02 00 D6 00 02 00  FD 00 02 00 25 01 02 00  ....Ö...ý...%...
```

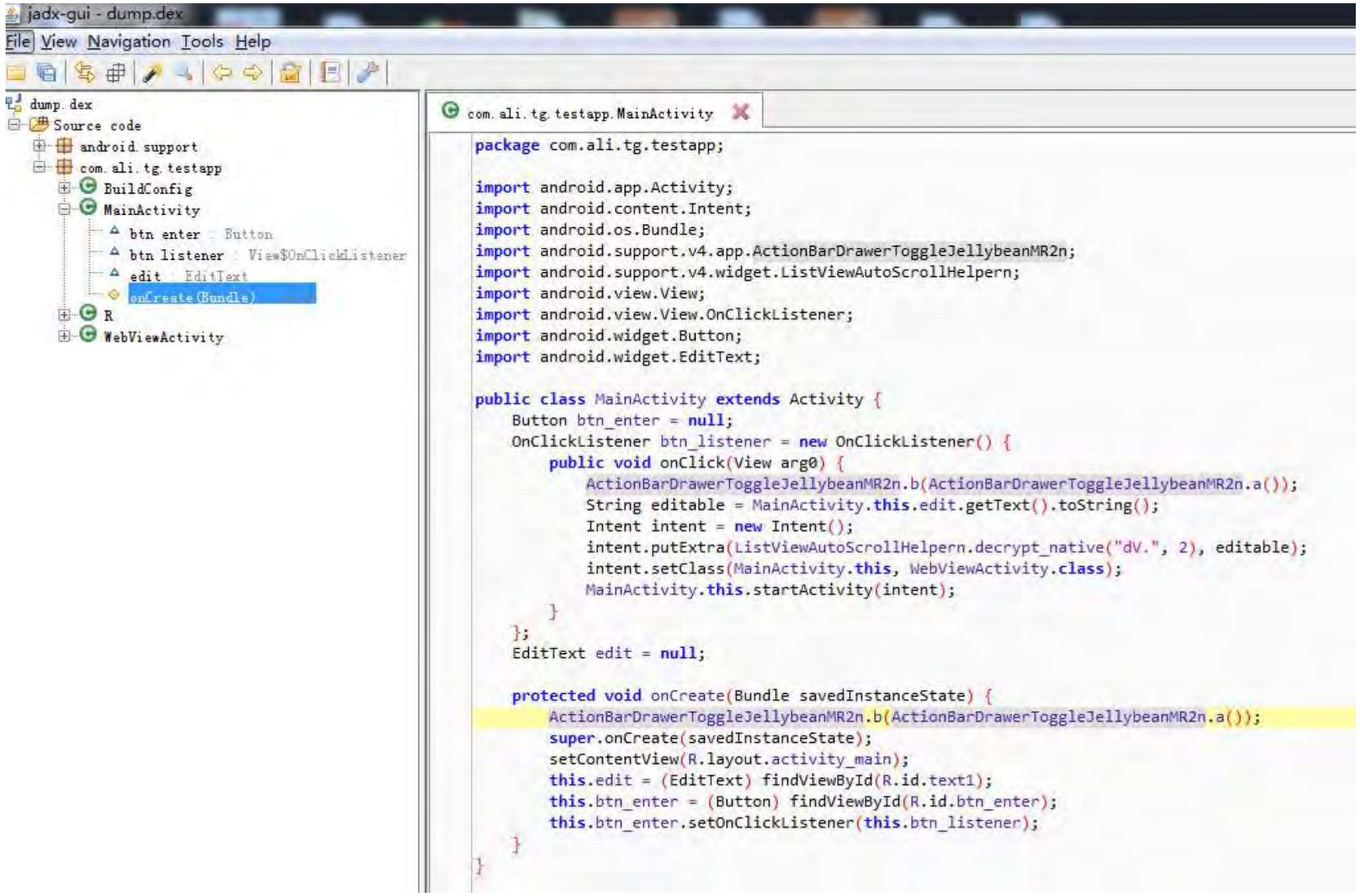
4. 然后我们就可以使用 ida 的 script command 去 dump 内存中的 dex 文件了。


IDA - C:\Users\ADMINI~1\AppData\Local\Temp\ida41563.idb (app_process)
File Edit Jump Search View Debugger Options Windows Help
New instance
Open...
Load file
Produce file
Diff Database... Shift+D
Script file... Alt+F7
Script command... Shift+F2
Save Ctrl+W
Save as...
Take database snapshot... Ctrl+Shift+W
Close


Execute script
Snippet list
Name
Default snippet *
Line 1 of 1
Please enter script body
static main(void)
{
auto fp, begin, end, dexbyte;
fp = fopen("C:\\dump.dex", "wb");
begin = r0;
end = r0 + r1;
for (dexbyte = begin; dexbyte < end; dexbyte ++)
fputc(Byte(dexbyte), fp);
}
Line:9 Column:2
Scripting language
IDC
Tab size
4
Run
Export
Import
Save

```
static main(void)
{
    auto fp, begin, end, dexbyte;
    fp = fopen("C:\\dump.dex", "wb");
    begin = r0;
    end = r0 + r1;
    for ( dexbyte = begin; dexbyte < end; dexbyte ++ )
            fputc(Byte(dexbyte), fp);
}
```

Dump 完 dex 文件后，我们就可以用工具来查看了。



当然这只是最简单脱壳方法，很多高级壳会动态修改 dex 的结构体，比如将 codeoffset 指向内存中的其他地址，这样的话你 dump 出来的 dex 文件其实是不完整的，因为代码段保存在了内存中的其他位置。

android 中 SO 文件动态调试

0X00 前言

为了增加 APK 文件的破解难度，很多比较重要的功能都是通过 native 实现，与反编译不同的是，java 的伪代码可以很清楚的得到程序的逻辑关 系，但是 so 文件反编译得到的是汇编代码，使用 ida 这样的神器可以得到 c 的伪代码。使用 ida 实现 so 的动态调试，可以在关键地方下断点，对于一些变 换，比如有的时候输入的内容与经过一些列的变换后的结果比较，这个时候我们不需要关心中间的变换过程，在动态调试的时候，在比较的地方下个断点，然后在内 存中查看此时比较的常量。正好最近做了 CTF 的一个 re。顺便熟悉下 ida 的动态调试步骤。

0x01 APK 逻辑的简单分析

在 apk 的 manifest 文件里，我们得到 apk 的入口 avtivity。包括包名和主类。

AndroidManifest.xml | strings.xml | Seclreg.smali

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.ssctf.seclreg">
    <application android:allowBackup="true" android:icon="@drawable/ic_launcher" android:label="@string/app_name" android:theme="@style/AppTheme">
        <activity android:label="@string/app_name" android:name="com.ssctf.seclreg.Seclreg">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

通过一步步分析，得知：用户输入 用户名和 密码， 其中用户名必须是 〈string name=”username”>secl-007</string〉 ，而密码的判断逻辑是调用了 native 函数（Java_com_ssctf_seclreg_Seclo0o_getpl ）去判断，这个时候用 ida 静态分析，打开 apk 加载的 so 文件，通过分析，我们发现对于输入的密码没有进行任何处理，只是判断输入密码的长度，如果长度 是 39，然后和一个常量做比较，是否正确。而这个常量在运行过程经过一些列的变化，因此我们不需要关心变化的过程。有两个方法：

1. 输入任意长度为 39 的内容，然后 dump 整个程序的内存，这个时候在 dump 内容中寻找，（因为我们知道这个常量特定方式 SSCTF{*****）
2. 使用 IDA 动态调试，在比较的地方下断点，得到此时常量的内容。这篇文章我们使用第二种方法实现

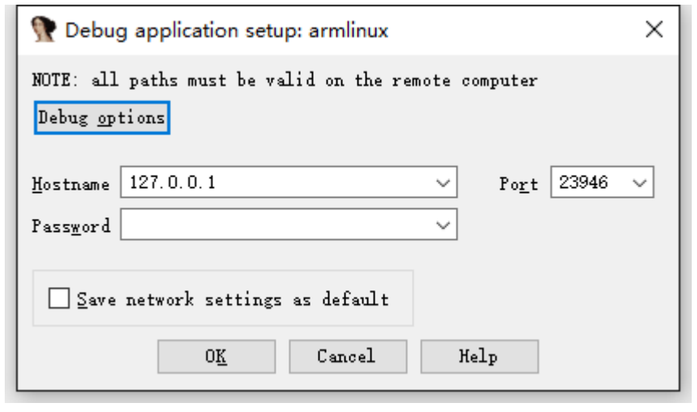
```
if ( userPass )
{
  v11 = j_j_strlen(userPass);
  v12 = 0;
  if ( v11 == 39 )
    v12 = (unsigned int)j_j_strncmp(pl, userPass, 0x27u) <= 0;
}
```

0x02 IDA 动态调试 SO

关于 IDA 动态调试步骤，首先你的手机需要 ROOT。

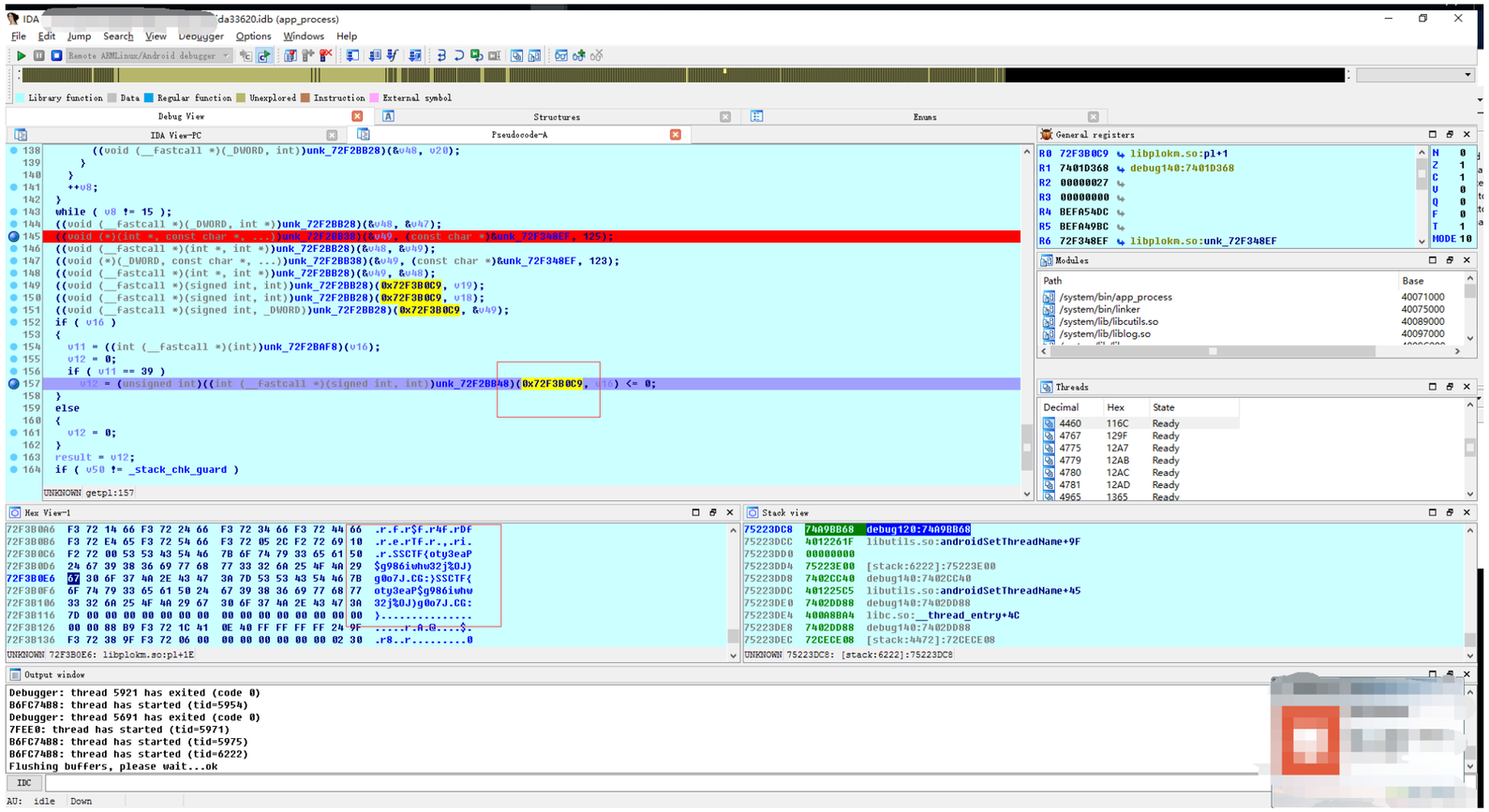
1. 把 ida 目录下的 android_server 文件放到手机的目录，adb pull android_server /data/local/tmp
2. 给与 android_server 777 权限， chmod 777 android_server

3. 端口的转发 adb forward tcp:23946 tcp:23946
4. 打开 ddms
5. 运行 android_server ./android_server
6. 将要调试的 apk 文件以调试的方式运行　adb shell am start -D -n com.ssctf.seclreg/.Seclreg　　包名/.类名
7. 这个时候手机显示等到调试连接模式，打开 ida，debugger-attach-remote android server



8. 选择 ok，并且 F9 运行，这个时候，jdb 附加程序，jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700
9. 这样程序就可以动态调试了，我们根据 SO 基址和函数的偏址找到调试时候函数的地址，然后 F2 下断点，

这个时候在 APK 输入，然后程序运行到我们设置断点的地方，



然后我们在数据窗口里查看数据，得到 FLAG：）

入口中断

init_array

动态调试 SO 之在.init_array 段下断点

前言

由前面的分析可以知道，so 被加载之后最开始执行的是.init_array 段的代码。然后才会去执行 jni_onload

那么，在.init_array 处断下来便是很有必要的

前期准备：

android 系统中位于/system/bin/的 linker
ida 6.4
android_server
mytestcm.apk

准备工作做好之后，下面正式开始调试

0×1

push android_server 到 tmp 目录下，给权限，然后以 root 身份执行。

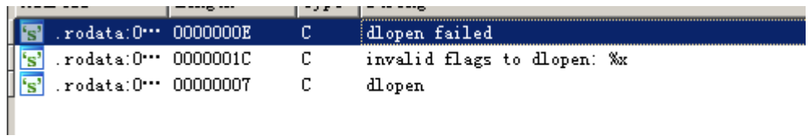
adb forward 端口转发

IDA attach 对应进程

0×2

新开一个 IDA，载入 linker

Shift+F12 打开字符串窗口，搜索字符串：dlopen



定位到：

```
text:00000F30                           sub_F30                                 ; DATA XREF: .data:00010034↓o
text:00000F30 38 B5                                     PUSH            {R3-R5,LR}
text:00000F32 04 46                                     MOV             R4, R0
text:00000F34 0B 48                                     LDR             R0, =(unk_100A4 - 0xF3C)
text:00000F36 0D 46                                     MOV             R5, R1
text:00000F38 78 44                                     ADD             R0, PC ; unk_100A4
text:00000F3A 07 F0 98 EB                               BLX             sub_866C
text:00000F3E 20 46                                     MOV             R0, R4
text:00000F40 29 46                                     MOV             R1, R5
text:00000F42 01 F0 31 FD                               BL              sub_29A8
text:00000F46 04 46                                     MOV             R4, R0
text:00000F48 30 B9                                     CBNZ            R0, loc_F58
text:00000F4A 01 F0 17 FB                               BL              sub_257C
text:00000F4E 01 46                                     MOV             R1, R0
text:00000F50 05 48                                     LDR             R0, =(aDlopenFailed - 0xF56)
text:00000F52 78 44                                     ADD             R0, PC  ; "dlopen failed"
text:00000F54 FF F7 30 FF                               BL              sub_DB8
```

找到 dlopen 函数的偏移 0xF30

动态调试的 IDA 中，G 跳转到：400D3000+F30=400D3F30 处，下好断点

```
linker:400D3F30                         CODE16
linker:400D3F30 38 B5                   PUSH    {R3-R5,LR}
linker:400D3F32 04 46                   MOV     R4, R0
linker:400D3F34 0B 48                   LDR     R0, =0xF168
linker:400D3F36 0D 46                   MOV     R5, R1
linker:400D3F38 78 44                   ADD     R0, PC ; unk_400E30A4
linker:400D3F3A 07 F0 98 EB             BLX     sub_400DB66C
linker:400D3F3E 20 46                   MOV     R0, R4
linker:400D3F40 29 46                   MOV     R1, R5
linker:400D3F42 01 F0 31 FD             BL      unk_400D59A8
linker:400D3F46 04 46                   MOV     R4, R0
linker:400D3F48 30 B9                   CBNZ    R0, off_400D3F58
linker:400D3F4A 01 F0 17 FB             BL      sub_400D557C
linker:400D3F4E 01 46                   MOV     R1, R0
linker:400D3F50                         CODE32
linker:400D3F50 05 48 78 44             LDRMIBT R4, [R8],#-0x805
```

搜索字符串：calling

```
's' .rodata:0··· 0000001D  C  [ Calling %s @ %p for '%s' ]
's' .rodata:0··· 00000022  C  [ Done calling %s @ %p for '%s' ]
's' .rodata:0··· 00000027  C  [ Calling %s (size %d) @ %p for '%s' ]
's' .rodata:0··· 0000001D  C  [ Done calling %s for '%s' ]
```

定位到：

```
.text:00002720
.text:00002720                          sub_2720                                ; CODE XREF: sub_2794+8E↓p
.text:00002720                                                                  ; sub_28A4+BA↓p ...
.text:00002720 53 1E                                    SUBS            R3, R2, #1
.text:00002722 03 33                                    ADDS            R3, #3
.text:00002724 73 B5                                    PUSH            {R0,R1,R4-R6,LR}
.text:00002726 06 46                                    MOV             R6, R0
.text:00002728 0D 46                                    MOV             R5, R1
.text:0000272A 14 46                                    MOV             R4, R2
.text:0000272C 24 D8                                    BHI             loc_2778
.text:0000272E 13 48                                    LDR             R0, =(dword_10678 - 0x2734)
.text:00002730 78 44                                    ADD             R0, PC ; dword_10678
.text:00002732 01 68                                    LDR             R1, [R0]
.text:00002734 01 29                                    CMP             R1, #1
.text:00002736 09 DD                                    BLE             loc_274C
.text:00002738 11 49                                    LDR             R1, =(aLinker - 0x2748)
.text:0000273A 04 20                                    MOVS            R0, #4
.text:0000273C 11 4A                                    LDR             R2, =(aCallingSPForS - 0x274A)
.text:0000273E 2B 46                                    MOV             R3, R5
.text:00002740 8D E8 50 00                              STMEA.W         SP, {R4,R6}
.text:00002744 79 44                                    ADD             R1, PC  ; "linker"
.text:00002746 7A 44                                    ADD             R2, PC  ; "[ Calling %s @ %p for '%s' ]"
.text:00002748 01 F0 54 FF                              BL              sub 45F4
```

得到偏移：0×2720

来到动态调试的 ida，G 跳转到：400D3000+2720=400D5720 处，下断点

```
linker:400D5720 53 1E                   SUBS    R3, R2, #1
linker:400D5722 03 33                   ADDS    R3, #3
linker:400D5724 73 B5                   PUSH    {R0,R1,R4-R6,LR}
linker:400D5726 06 46                   MOV     R6, R0
linker:400D5728 0D 46                   MOV     R5, R1
linker:400D572A 14 46                   MOV     R4, R2
linker:400D572C 24 D8                   BHI     loc_400D5778
linker:400D572E 13 48                   LDR     R0, =(dword_400E3678 - 0x400D5734)
linker:400D5730 78 44                   ADD     R0, PC ; dword_400E3678
linker:400D5732 01 68                   LDR     R1, [R0]
linker:400D5734 01 29                   CMP     R1, #1
linker:400D5736 09 DD                   BLE     loc_400D574C
linker:400D5738 11 49                   LDR     R1, =0x9879
linker:400D573A 04 20                   MOVS    R0, #4
linker:400D573C 11 4A                   LDR     R2, =0xA2F4
linker:400D573E 2B 46                   MOV     R3, R5
```

往下：

```
linker:400D5738 11 49                   LDR     R1, =0x9879
linker:400D573A 04 20                   MOVS    R0, #4
linker:400D573C 11 4A                   LDR     R2, =0xA2F4
linker:400D573E 2B 46                   MOV     R3, R5
linker:400D573E                         ; ---------------------------------------------------------------------------
linker:400D5740 8D                      DCB 0x8D ;
linker:400D5741 E8                      DCB 0xE8 ;
linker:400D5742                         ; ---------------------------------------------------------------------------
linker:400D5742 50 00                   LSLS    R0, R2, #1
linker:400D5744 79 44                   ADD     R1, PC
linker:400D5746 7A 44                   ADD     R2, PC
linker:400D5748 01 F0 54 FF             BL      unk_400D75F4
linker:400D574C
linker:400D574C                         loc_400D574C                            ; CODE XREF: linker:calloc+1EA↑j
linker:400D574C A0 47                   BLX     R4
linker:400D574E 0E 4A                   LDR     R2, =(dword_400E3678 - 0x400D5754)
linker:400D5750 7A 44                   ADD     R2, PC ; dword_400E3678
linker:400D5752 13 68                   LDR     R3, [R2]
linker:400D5754 01 2B                   CMP     R3, #1
linker:400D5756 09 DD                   BLE     loc_400D576C
linker:400D5758 0C 49                   LDR     R1, =0x9859
linker:400D575A 04 20                   MOVS    R0, #4
linker:400D575C 0C 4A                   LDR     R2, =0xA2F1
linker:400D575E 2B 46                   MOV     R3, R5
linker:400D575E                         ; ---------------------------------------------------------------------------
```

400D574C　　BLX　　R4 处就是调用 init 段代码的地方！直接跟进就能看到

0×3

按 F9 运行
然后打开 Eclipse 或者 ddms
执行 jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700
程序就会断在第一个断点处，F9 几次就段在 blx R4 处
F7 跟进就来到 init 段代码处：

```
libSyclover.so:758E433C                 ; ---------------------------------------------------------------------------
libSyclover.so:758E433C                 CODE16
libSyclover.so:758E433C 08 B5           PUSH    {R3,LR}
libSyclover.so:758E433E 04 49           LDR     R1, =(aWanniba - 0x758E4348)
libSyclover.so:758E4340 04 4A           LDR     R2, =(aJni_onloadWann - 0x758E434A)
libSyclover.so:758E4342 02 20           MOVS    R0, #2
libSyclover.so:758E4344 79 44           ADD     R1, PC                          ; "wanniba"
libSyclover.so:758E4346 7A 44           ADD     R2, PC                          ; "JNI_OnLoad wanniba: my init Func!"
libSyclover.so:758E4348 FF F7 BA EE     BLX     unk_758E40C0
libSyclover.so:758E434C 08 BD           POP     {R3,PC}
```

OnlyEnd 、

根据自己手机的系统版本，找到对应的安卓源码。然后再去找对应的 linker 部分源代码。根据你的源代码里的信息，去定位 dlopen。

调试 init.array 段中函数方法

环境：rom 包 4.4.3
手机：Nexus 4

源码文件路径：/dalvik/vm/Native.cpp， bionic/linker/linker.cpp

此文的作用，我想大家应该都知道，话不多说就是干。定位到/dalvik/vm/Native.cpp 文件中的 dvmLoadNativeCode 函数，此函数完成对 so 加载，初始化工作。

```
bool dvmLoadNativeCode(const char* pathName, Object* classLoader,
        char** detail)
 {
    SharedLib* pEntry;
    void* handle;
    bool verbose;

    /* reduce noise by not chattering about system libraries */
    verbose = !!strncmp(pathName, "/system", sizeof("/system")-;
    verbose = verbose && !!strncmp(pathName, "/vendor", sizeof("/vendor")-;

    if (verbose)
        LOGD("Trying to load lib %s %p", pathName, classLoader);

    *detail = NULL;

    /*
     * See if we've already loaded it If we have, and the class loader
     * matches, return successfully without doing anything.
     */
    pEntry = findSharedLibEntry(pathName); //检查参数 pathName 所指定的 so 文件是否已经加载过，若加载过返回 pEntry 对象
    if (pEntry != NULL) {
        if (pEntry->classLoader != classLoader) {
            LOGW("Shared lib '%s' already opened by CL %p; can't open in %p",
                pathName, pEntry->classLoader, classLoader);
            return false;
```

```
        }
        if (verbose) {
            LOGD("Shared lib '%s' already loaded in same CL %p",
                pathName, classLoader);
        }
        if (!checkOnLoadResult(pEntry))
            return false;
        return true;
    }

////////////////////////////////////////////////////////////////////////////////////////////////////
    Thread* self = dvmThreadSelf();
    ThreadStatus oldStatus = dvmChangeStatus(self, THREAD_VMWAIT);
    handle = dlopen(pathName, RTLD_LAZY);//此 so 没有被加载过，调用 dlopen 加载此 so
    dvmChangeStatus(self, oldStatus);

    if (handle == NULL) {
        *detail = strdup(dlerror());
        return false;
    }

/* create a new entry */
SharedLib* pNewEntry;               //建立 SharedLib 对象 pNewEntry 来描述此 so 加载信息
pNewEntry = (SharedLib*) calloc( sizeof(SharedLib));
pNewEntry->pathName = strdup(pathName);
pNewEntry->handle = handle;
pNewEntry->classLoader = classLoader;
dvmInitMutex(&pNewEntry->onLoadLock);
pthread_cond_init(&pNewEntry->onLoadCond, NULL);
pNewEntry->onLoadThreadId = self->threadId;

/* try to add it to the list */
SharedLib* pActualEntry = addSharedLibEntry(pNewEntry); //保存起来

if (pNewEntry != pActualEntry) {
    LOGI("WOW: we lost a race to add a shared lib (%s CL=%p)",
        pathName, classLoader);
    freeSharedLibEntry(pNewEntry);
    return checkOnLoadResult(pActualEntry);
} else {
    if (verbose)
        LOGD("Added shared lib %s %p", pathName, classLoader);
```

```
bool result = true;
void* vonLoad;
int version;

vonLoad = dlsym(handle, "JNI_OnLoad");
if (vonLoad == NULL) {
    LOGD("No JNI_OnLoad found in %s %p, skipping init",
        pathName, classLoader);
} else {
    /*
     * Call JNI_OnLoad We have to override the current class
     * loader, which will always be "null" since the stuff at the
     * top of the stack is around Runtime.loadLibrary() (See
     * the comments in the JNI FindClass function.)
     */
    OnLoadFunc func = (OnLoadFunc)vonLoad;
    Object* prevOverride = self->classLoaderOverride;

    self->classLoaderOverride = classLoader;
    oldStatus = dvmChangeStatus(self, THREAD_NATIVE);
if (gDvm.verboseJni) {
    LOGI("[Calling JNI_OnLoad for \"%s\"]", pathName);
}
version = (*func)(gDvmJni.jniVm, NULL);
dvmChangeStatus(self, oldStatus);
self->classLoaderOverride = prevOverride;

if (version != JNI_VERSION_&& version != JNI_VERSION_&&
    version != JNI_VERSION_
{
   LOGW("JNI_OnLoad returned bad version (%d) in %s %p",
       version, pathName, classLoader);
   /*
    * It's unwise to call dlclose() here, but we can mark it
    * as bad and ensure that future load attempts will fail.
    *
    * We don't know how far JNI_OnLoad got, so there could
    * be some partially-initialized stuff accessible through
    * newly-registered native method calls We could try to
    * unregister them, but that doesn't seem worthwhile.
    */
```

```
        result = false;
    } else {
        if (gDvm.verboseJni) {
            LOGI("[Returned from JNI_OnLoad for \"%s\"]", pathName);
        }
    }
}

if (result)
    pNewEntry->onLoadResult = kOnLoadOkay;
else
    pNewEntry->onLoadResult = kOnLoadFailed;

pNewEntry->onLoadThreadId =

/*
 * Broadcast a wakeup to anybody sleeping on the condition variable.
 */
dvmLockMutex(&pNewEntry->onLoadLock);
pthread_cond_broadcast(&pNewEntry->onLoadCond);
dvmUnlockMutex(&pNewEntry->onLoadLock);
        return result;
    }
}
```

1、调用 `findSharedLibEntry` 函数检查指定的 so 是否已经被加载过了。
2、若没有被加载过，此时调用 dlopen 加载，并执行该动态链接库的初始化代码（init 段里的函数就是在这里执行的），并获取 so 句柄。下图红色框中为 dlopen 调用
3、建立 SharedLib 对象 pNewEntry 来描述它的加载信息
4、之后通过调用 dlsym 获取 so 中的 JNI_OnLoad 函数的地址，保存在 func。若此函数地址为空，则说明此 so 中没有 JNI_Onload 函数—-隐喻使用 javah 的编码方式（静态）.
5、最后调用 `version = (*func)(gDvmJni.jniVm, NULL);` 执行 Jni_onLoad 函数。

```
libdvm.so:414B9ED4 BL              _Z13dvmThreadSelfv
libdvm.so:414B9ED8 MOVS            R1, #8
libdvm.so:414B9EDA MOV             R7, R0
libdvm.so:414B9EDC BL              _Z15dvmChangeStatusP6Thread12ThreadStatus
libdvm.so:414B9EE0 MOVS            R1, #1
libdvm.so:414B9EE2 MOV             R5, R0
libdvm.so:414B9EE4 MOV             R0, R4
libdvm.so:414B9EE6 BLX             sub_41487610            ; handle = dlopen(pathName, RTLD_LAZY);
libdvm.so:414B9EEA MOV             R1, R5
libdvm.so:414B9EEC MOV             R11, R0
libdvm.so:414B9EEE MOV             R0, R7
libdvm.so:414B9EF0 BL              _Z15dvmChangeStatusP6Thread12ThreadStatus
libdvm.so:414B9EF4 CMP.W           R11, #0
libdvm.so:414B9EF8 BNE             loc_414B9F1C
libdvm.so:414B9EFA BLX             unk_4148761C
libdvm.so:414B9EFE BLX             unk_41487430
```

dlopen 函数源码：

```
void* dlopen(const char* filename, int flags) {
```

```
  ScopedPthreadMutexLocker locker(&gDlMutex);
  soinfo* result = do_dlopen(filename, flags);
  if (result == NULL) {
    __bionic_format_dlerror("dlopen failed", linker_get_error_buffer());
    return NULL;
  }
  return result;
}
```

1. dlopen 函数受限获取一个互斥量。这个互斥量用于保护动态链接库的链表。
2. 调用 do_dlopen 函数完成 do_dlopen 函数完成正真 so 加载，初始化调用工作

```
linker:4001AF30 PUSH            {R3-R5,LR}
linker:4001AF32 MOV             R4, R0                  ; pathName
linker:4001AF34 LDR             R0, =(unk_4002A0A4 - 0x4001AF3C)
linker:4001AF36 MOV             R5, R1                  ; arg2
linker:4001AF38 ADD             R0, PC ; unk_4002A0A4
linker:4001AF3A BLX             unk_4002266C
linker:4001AF3E MOV             R0, R4
linker:4001AF40 MOV             R1, R5
linker:4001AF42 BL              loc_4001C9A8            ; do_dlopen(const char* name, int flags)
linker:4001AF46 MOV             R4, R0
linker:4001AF48 CBNZ            R0, loc_4001AF58
linker:4001AF4A BL              unk_4001C57C
linker:4001AF4E MOV             R1, R0
linker:4001AF50 LDR             R0, =(unk_40025FB3 - 0x4001AF56)
linker:4001AF52 ADD             R0, PC
linker:4001AF54 BL              unk_4001ADB8
linker:4001AF58
linker:4001AF58 loc_4001AF58                            ; CODE XREF: sub_4001AF30+18↑j
```

do_dlopen 函数源码：

```
soinfo* do_dlopen(const char* name, int flags) {
  if ((flags & ~(RTLD_NOW|RTLD_LAZY|RTLD_LOCAL|RTLD_GLOBAL)) !=  {
    DL_ERR("invalid flags to dlopen: %x", flags);
    return NULL;
  }
  set_soinfo_pool_protection(PROT_READ | PROT_WRITE);
  soinfo* si = find_library(name); //加载 so, 返回 soinfo 对象
  if (si != NULL) {
    si->CallConstructors(); //调用构造函数（init）
  }
  set_soinfo_pool_protection(PROT_READ);
  return si;
}
```

```
linker:400449E2 MOVS            R0, #3
linker:400449E4 BL              unk_40043868            ; set_soinfo_pool_protection(PROT_READ | PROT_WRITE);
linker:400449E4                                         ; 设置soinfo内存可度可写
linker:400449E8 MOV             R0, R4
linker:400449EA BL              unk_400442B4            ; soinfo* si = find_library(name); //加载so，返回soinfo对象
linker:400449EE MOV             R4, R0
linker:400449F0 CBZ             R0, loc_400449F6        ; if(si != null)
linker:400449F2 BL              sub_400448A4            ; si->CallConstructors(); //调用构造函数（init）
linker:400449F6
linker:400449F6 loc_400449F6                            ; CODE XREF: linker:_start+1F90↑j
linker:400449F6 MOVS            R0, #1
linker:400449F8 BL              unk_40043868            ; set_soinfo_pool_protection(PROT_READ); 修改soinfo内存池只读
linker:400449FC B               loc_40044A00
linker:400449FE ; ---------------------------------------------------------------------------
linker:400449FE
linker:400449FE loc_400449FE                            ; CODE XREF: linker:_start+1F6E↑j
linker:400449FE                                         ; linker:_start+1F80↑j
linker:400449FE MOVS            R4, #0
```

地 址 0x4001C9EA 地址处，就是调用 find_library(soName)函数，加载 so,然后获取 soinfo 对象,接下来调用对象的 CallConstructors()成员函数，去执行动态链接库的初始化代码，init 段的函数调用就是在这里执行的，接下来跟进 0x4001C9F2 的函数调用。

```
void soinfo::CallConstructors() {
  if (constructors_called) {
    return;
  }
  ////////////////////////////////
  TRACE("\"%s\": calling constructors", name);

  // DT_INIT should be called before DT_INIT_ARRAY if both are present.
  CallFunction("DT_INIT", init_func);
  CallArray("DT_INIT_ARRAY", init_array, init_array_count, false);
}
```

```
linker:40044944 LDR             R1, =(aLinker - 0x40044950)
linker:40044946 MOVS            R0, #4
linker:40044948 LDR             R2, =(aSCallingCons_0 - 0x40044952)
linker:4004494A MOV             R3, R4
linker:4004494C ADD             R1, PC                  ; "linker"
linker:4004494E ADD             R2, PC                  ; "\"%s\": calling constructors"
linker:40044950 BL              unk_400465F4            ; TRACE("\"%s\":calling constructors", name);
linker:40044954
linker:40044954 loc_40044954                            ; CODE XREF: sub_400448A4+52↑j
linker:40044954 LDR             R1, =(aDt_init - 0x40044960)
linker:40044956 MOV             R0, R4
linker:40044958 LDR.W           R2, [R4,#0xF0]
linker:4004495C ADD             R1, PC                  ; "DT_INIT"
linker:4004495E BL              unk_40044720            ; CallFunction(this, "DT_INIT", init_func);
linker:40044962 LDR             R1, =(aDt_init_array - 0x4004496C)
linker:40044964 MOVS            R0, #0
linker:40044966 STR             R0, [SP,#0x28+var_28]
linker:40044968 ADD             R1, PC                  ; "DT_INIT_ARRAY"
linker:4004496A MOV             R0, R4
linker:4004496C LDR.W           R2, [R4,#0xE0]
```

CallFunction 函数则是关键点，先来看看源码：

```
void soinfo::CallFunction(const char* function_name UNUSED, linker_function_t function) {
  if (function == NULL || reinterpret_cast<uintptr_t>(function) == static_cast<uintptr_t>(-) {
    return;
  }

  TRACE("[ Calling %s @ %p for '%s' ]", function_name, function, name);
  function(); //这里就是调用 init 中的函数
  TRACE("[ Done calling %s @ %p for '%s' ]", function_name, function, name);

  // The function may have called dlopen( or dlclose(, so we need to ensure our data structures
  // are still writable. This happens with our debug malloc (see http://b/.
  set_soinfo_pool_protection(PROT_READ | PROT_WRITE);
}
```

阅读上面的代码，可以了解到，在函数 CallFunction 内完成了对 init 段中函数的调用。对应的汇编：

```
linker:40044720 loc_40044720                            ; CODE XREF: sub_400448A4+BA↓p
linker:40044720 SUBS            R3, R2, #1
linker:40044722 ADDS            R3, #3
linker:40044724 PUSH            {R0,R1,R4-R6,LR}
linker:40044726 MOV             R6, R0
linker:40044728 MOV             R5, R1
linker:4004472A MOV             R4, R2
linker:4004472C BHI             loc_40044778
linker:4004472E LDR             R0, =(dword_40052678 - 0x40044734)
linker:40044730 ADD             R0, PC ; dword_40052678
linker:40044732 LDR             R1, [R0]
linker:40044734 CMP             R1, #1
linker:40044736 BLE             loc_4004474C            ; function()
linker:40044738 LDR             R1, =(aLinker - 0x40044748)
linker:4004473A MOVS            R0, #4
linker:4004473C LDR             R2, =(aCallingSPForS - 0x4004474A)
linker:4004473E MOV             R3, R5
linker:40044740 STMEA.W         SP, {R4,R6}
linker:40044744 ADD             R1, PC                  ; "linker"
linker:40044746 ADD             R2, PC                  ; "[ Calling %s @ %p for '%s' ]"
linker:40044748 BL              unk_400465F4
linker:4004474C
linker:4004474C loc_4004474C                            ; CODE XREF: linker:_start+1CD6↑j
linker:4004474C BLX             R4                      ; function()
linker:4004474E LDR             R2, =(dword_40052678 - 0x40044754)
linker:40044750 ADD             R2, PC ; dword_40052678
linker:40044752 LDR             R3, [R2]
```

上图中 BLX R4 函数即对应 function()函数，下断，跟进。成功来到 init 段函数

```
libinitsyc.so:76118C64 my_init
libinitsyc.so:76118C64 PUSH            {R3,LR}
libinitsyc.so:76118C66 LDR             R0, =(aThisInitSectio - 0x76118C6C)
libinitsyc.so:76118C68 ADD             R0, PC                  ; "this init section"
libinitsyc.so:76118C6A BLX             unk_76118BE0
libinitsyc.so:76118C6E POP             {R3,PC}
libinitsyc.so:76118C6E ; ---------------------------------------------------------------------------
libinitsyc.so:76118C70 off_76118C70 DCD aThisInitSectio - 0x76118C6C
```

定位到入口 init 中的函数过程：

1. 查看 dvm.so 模块，搜索 dvmLoadNativeCode 函数
2. 分析 DvmLoadNativeCode 函数，找到对 dlopen 函数的调用，并跟进
3. 由于 rom 包版本的不同这一点可能不一样，在 4.4.2rom 包上，dlopen 函数是对 do_dlopen 函数的封装，对 do_dlopen(soName, falgs)跟中;
4. do_dlopen 函数中调用 find_library(soName).完成对 so 的加载，并会返回一个 soinfo 对象
5. 调用 soinfo 对象的成员函数 constructors();完成调用动态链接库初始化代码
6. 在 constructors()函数中，调用 CallFunction(“DT_INIT”, init_func); 回调函数 init_func 就是 init 段中的函数
7. 进入 CallFunction 找到 BLX R4 既是对 init_func 函数的调用

当然上面这种方法过于复杂，其实可以刷 debug 版的 rom 包，通过定位 do_dlopen，dlopen，或 CallFunction 函数中 的特征字符串快速定位到关键点（BLX R4）。但有些手机并不好刷原生的 debug 版 rom 包，比如我的测试机大华为，我只有对他呵呵了，所以掌握以上的方法很有必要的.

http://1.honebl.sinaapp.com/?p=213

JNI_Onload

动态调试 Android so 库函数的方法

1、am start -D -n 包名/类名，以等待调试的模式启动 APK 应用；

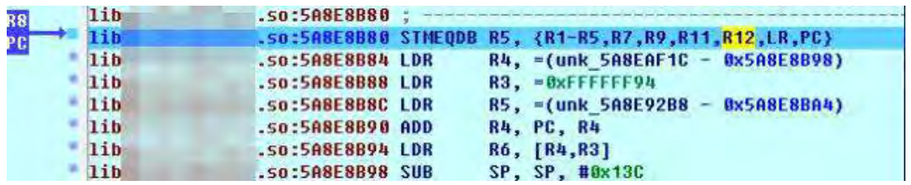
2、以 su 权限开启 android_server，然后 adb forward 转发端口，并用 IDA 附加相应进程；

3、通过 DDMS 获取相应进程的端口号，然后使用 jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700（DDMS 查询到的端口号）；

4、连接成功后，按 F9 后手机上的“waiting for debugger"提示会自动消失；

5、此时在 so 库的 JNI_Onload 函数上下断点，可通过 Ctrl+S 找到对应 so 库的基址，然后加上 JNI_Onload 的文件偏移量，即可找到 JNI_Onload 函数的内存地址，然后按 F9 运行后即可断下。

下图是使用上述方法，断在某 so 库 JNI_Onload 函数的截图：



android 在 JNI_OnLoad 入口函数下断点动态调试 so 库

一般来说，很多 APK 的校验代码，都会在程序运行的时候自动加载一些动态 so 库，然后执行这些库中的校验代码。所以为了能够通过程序的校验，我们必须在执行这些函数之前下断点——理想的方法就是在 JNI_OnLoad 入口函数下断点。

在 2.3.3 模拟器中详细步骤如下：

①在控制台输入 adb shell 进入手机，然后使用 am start -D -n 包名/类名，以等待调试的模式启动 APK 应用；

这里需要说明的是 “包名/类名”的书写方法：

am start -D -n {包(package)名}/{包名}.{活动(activity)名称}

程序的入口类可以从每个应用的 AndroidManifest.xml 的文件中得到，以计算器（calculator）为例，它的

<manifest xmlns:android="http://schemas.android.com/apk/res/android" …

package="com.android.calculator2" …>…

由此计算器（calculator）的启动方法为：# am start -D -n com.android.calculator2/com.android.calculator2.Calculator

对于 HelloActivity 这个示例工程，AndroidManifest.xml 如下所示：

<manifest …

package="com.example.android.helloactivity" …>

由此它的启动方法为：

am start -D -n com.example.android.helloactivity/com.example.android.helloactivity.HelloActivity

②使用 adb 将 IDA(我的是 6.1)中的 android_server 传入手机的/data/local/tmp 目录中：

adb push android_server /data/local/tmp

```
#    再赋予权限
  adb shell
  cd /data/local/tmp
  chmod 777 android_server
#运行 android_server
  ./android_server
```

③另起一个 cmd 窗口，使用命令 adb forward tcp:23946 tcp:23946 进行窗口转发

④启动 IDA 主程序，点击菜单 Debugger->Attach->Remote ArmLinux/Android debugger ，打开调试程序对话框，在 hostname 一栏输入 localhost,

点击 ok，然后在 IDA 弹出的窗口中，选择自己要附加的进程后点击 OK 即可。

⑤点击菜单 Debugger->Debugger Opitions 在弹出的 Debugger setup 窗口的 Events 中选择 stop on thread start/exit 以及 stop on library load/unload，再点击 OK 退出。通过此操作可以设置程序在创建新线程和加载 so 时自动中断。

⑥通过 DDMS 获取相应进程的端口号，然后使用 jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=XXXX（DDMS 查询到的端口号）；

⑦连接成功后，按 F9 后手机上的“waiting for debugger"提示会自动消失，这个时候应该已经断在新线程，或者加载 so 处了。

⑧现在就可以在 IDA 中按下快捷键 CTRL + S 来查看要调试的 so 是否已经加载了，如果没有就 F9，直到加载了为止；如果已经有了，就记下该 so 的 start 位置，然后另开一个 ida 分析.so 库，找到 JNI_ONLOAD 的偏移地址，那么该 JNI_OnLoad 函数在进程中的真实地址就是 so.start + JNI_OnLoad_Offset。

这里需要说明的是：有可能在快捷键 CTRL + S 跳出的窗口中有两个同名的 so，我们应当选择权限为 RX 的这个，RX 一般是代码段，RW 一般是数据段。

得到真实地址后，在 IDA 中按下快捷键 G 跳转到这个地址，然后按下快捷键 F2 就完成在 JNI_OnLoad 函数入口处下断点了。

http://www.cnblogs.com/wanyuanchun/p/3760825.html

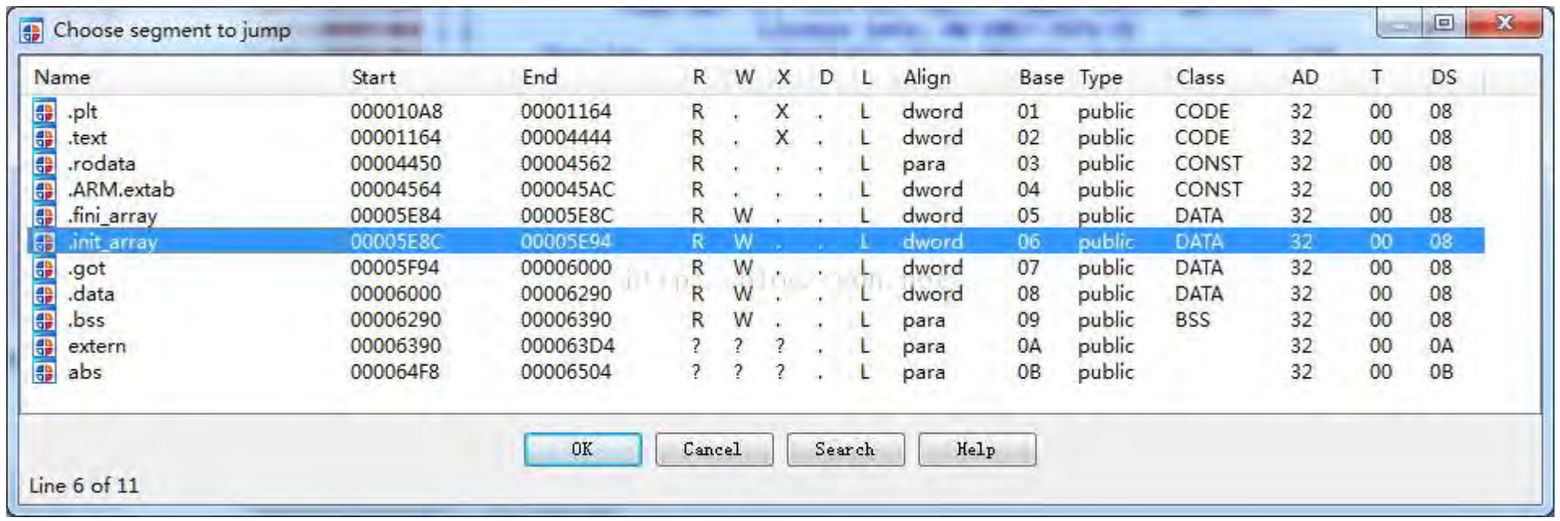
ida 动态调试 so，在 init_array 和 JNI_ONLOAD 处下断点

本文涉及到的 apk，请在 github 下载 https://github.com/jltxgcy/AliCrack/AliCrackme_2.apk。

0x00 如何在 JNI_ONLOAD 下断点

参考安卓逆向学习笔记(5) - 在 JNI_Onload 函数处下断点避开针对 IDA Pro 的反调试。最好使用模拟器调试，确保 Attach to process 后，对应进程在 DDMS 中出现小红蜘蛛。

下面将如何在 init_array 下断点，首先要找到 so 的 init_array 端，把 so 拖入 ida，然后按 Crtl+s，会出现该 so 的所有段。如下：



进入.init_array，如下：

```
.init_array:00005E8C ; Segment type: Pure data
.init_array:00005E8C                 AREA .init_array, DATA
.init_array:00005E8C                 ; ORG 0x5E8C
.init_array:00005E8C                 DCD sub_2378
.init_array:00005E90                 DCB    0
.init_array:00005E91                 DCB    0
.init_array:00005E92                 DCB    0
.init_array:00005E93                 DCB    0
.init_array:00005E93 ; .init_array   ends
.init_array:00005E93
```

其中 sub_2378 就是 init_array 的代码。

```
.text:00002378 sub_2378                                ; DATA XREF: .init_array:00005E8C↓o
.text:00002378                 STMFD   SP!, {R11,LR}
.text:0000237C                 LDR     R0, =(_GLOBAL_OFFSET_TABLE_ - 0x238C)
.text:00002380                 LDR     R1, =(sub_1CA8 - 0x5FBC)
.text:00002384                 ADD     R0, PC, R0 ; _GLOBAL_OFFSET_TABLE_
.text:00002388                 ADD     R0, R1, R0 ; sub_1CA8
.text:0000238C                 BL      sub_22AC
.text:00002390                 LDMFD   SP!, {R11,PC}
.text:00002390 ; End of function sub_2378
```

我们在这里下断点，具体调试的步骤和在 JNI_ONLOAD 下断点调试是一样的，参考安卓逆向学习笔记(5) - 在 JNI Onload 函数处下断点避开针对 IDA Pro 的反调试。网上有很多其他方法在 init_array 下断点，例如 Android 安全 - linker 加载 so 流程，在.init 下断点。我还是觉得上面的方法比较方便。

调试时使用 jdb -connect com.sun.jdi.SocketAttach:port=8700,hostname=localhost，有时会报如下错误：

```
C:\Users\Youku>jdb -connect com.sun.jdi.SocketAttach:port=8700,hostname=localhost
java.net.ConnectException: Connection refused: connect
        at java.net.DualStackPlainSocketImpl.connect0(Native Method)
        at java.net.DualStackPlainSocketImpl.socketConnect(DualStackPlainSocketImpl.java:79)
        at java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:345)
        at java.net.AbstractPlainSocketImpl.connectToAddress(AbstractPlainSocketImpl.java:206)
        at java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:188)
        at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:172)
        at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392)
        at java.net.Socket.connect(Socket.java:589)
        at com.sun.tools.jdi.SocketTransportService.attach(SocketTransportService.java:222)
        at com.sun.tools.jdi.GenericAttachingConnector.attach(GenericAttachingConnector.java:116)
        at com.sun.tools.jdi.SocketAttachingConnector.attach(SocketAttachingConnector.java:90)
        at com.sun.tools.example.debug.tty.VMConnection.attachTarget(VMConnection.java:519)
        at com.sun.tools.example.debug.tty.VMConnection.open(VMConnection.java:328)
        at com.sun.tools.example.debug.tty.Env.init(Env.java:63)
        at com.sun.tools.example.debug.tty.TTY.main(TTY.java:1066)
```


此时，我们先观察 DDMS。

com.yaotong.crackme	1284		8622 / 8700

使用 jdb -connect com.sun.jdi.SocketAttach:port=8622,hostname=localhost 即可。

0x01 也讲一下 ida 静态分析 so

首先列出 C++源码：

```
#include "com_example_jnidemo_JniGg.h"

int switch1(int a, int b, int i, int j, int k,int q){
    char *w = "i am winner in this year";
    switch (i){
    case 1:
        return a + b + j + k;
        break;
    case 2:
        return a - b;
        break;
    case 3:
```

```
        return a * b;
        break;
    case 4:
        return a / b;
        break;
    default:
        return a + b;
        break;
    }
}

JNIEXPORT jstring JNICALL Java_com_example_jnidemo_JniGg_ggPrintHello
  (JNIEnv * env, jobject this)
{

     return (*env)->NewStringUTF(env, "Current Coin is --  ");

}

JNIEXPORT jint JNICALL Java_com_example_jnidemo_JniGg_getCoin
  (JNIEnv * env, jobject this)
{

   return switch1(1,2,1,4,5,6);

}
```

编译成 so，然后拖进 ida 进行分析。

```
.text:00000E38              EXPORT Java_com_example_jnidemo_JniGg_getCoin
.text:00000E38 Java_com_example_jnidemo_JniGg_getCoin
.text:00000E38
.text:00000E38 var_10        = -0x10
.text:00000E38 var_C         = -0xC
.text:00000E38
.text:00000E38              PUSH    {LR}
.text:00000E3A              SUB     SP, SP, #0xC
.text:00000E3C              MOVS    R3, #5
.text:00000E3E              STR     R3, [SP,#0x10+var_10]
.text:00000E40              MOVS    R3, #6
.text:00000E42              STR     R3, [SP,#0x10+var_C]
.text:00000E44              MOVS    R0, #1
.text:00000E46              MOVS    R1, #2
.text:00000E48              MOVS    R2, #1
```

```
.text:00000E4A              MOVS    R3, #4
.text:00000E4C              BL      switch1
.text:00000E50              ADD     SP, SP, #0xC
.text:00000E52              POP     {PC}
.text:00000E52 ; End of function Java_com_example_jnidemo_JniGg_getCoin
```

arm 的参数传递规范，和函数中的局部变量定义规范，请参考 ARM 子函数定义中的参数放入寄存器的规则。这里使用 R0，R1，R2，R3 来传递前 4 个参数，使用堆栈来传递后两个参数。

```
.text:00000DEC
.text:00000DEC              EXPORT switch1
.text:00000DEC switch1                          ; CODE XREF: Java_com_example_jnidemo_JniGg_getCoin+14p
.text:00000DEC
.text:00000DEC arg_0        = 0
.text:00000DEC
.text:00000DEC              PUSH    {R4,LR}
.text:00000DEE ; 6:   v5 = a3 - 1;
.text:00000DEE              SUBS    R2, #1
.text:00000DF0 ; 7:   v6 = a1;
.text:00000DF0              MOVS    R4, R0
.text:00000DF2 ; 8:   result = a1 + a2;
.text:00000DF2              ADDS    R0, R0, R1
.text:00000DF4 ; 9:   if ( (unsigned int)v5 <= 3 )
.text:00000DF4              CMP     R2, #3          ; switch 4 cases
.text:00000DF6              BHI     def_DFA         ; jumptable 00000DFA default case
.text:00000DF8 ; 11:    result = v5;
.text:00000DF8              MOVS    R0, R2
.text:00000DFA ; 12:    switch ( v5 )
.text:00000DFA              BL      __gnu_thumb1_case_uqi ; switch jump
.text:00000DFA ; ---------------------------------------------------------------------------
.text:00000DFE jpt_DFA      DCB 2               ; jump table for switch statement
.text:00000DFF              DCB 0xA
.text:00000E00              DCB 7
.text:00000E01              DCB 0xC
.text:00000E02 ; ---------------------------------------------------------------------------
.text:00000E02 ; 15:        result = v6 + a2 + a4 + a5;
.text:00000E02
.text:00000E02 loc_E02                          ; CODE XREF: switch1+Ej
.text:00000E02              LDR     R2, [SP,#8+arg_0] ; jumptable 00000DFA case 0
.text:00000E04              ADDS    R4, R4, R1
.text:00000E06              ADDS    R3, R4, R3
.text:00000E08 ; 16:        break;
.text:00000E08              ADDS    R0, R3, R2
.text:00000E0A
.text:00000E0A def_DFA                          ; CODE XREF: switch1+Aj
```

```
.text:00000E0A                          ; switch1+24j ...
.text:00000E0A                POP     {R4,PC}         ; jumptable 00000DFA default case
.text:00000E0C ; ---------------------------------------------------------------------------
.text:00000E0C ; 18:        result = a2 * v6;
.text:00000E0C
.text:00000E0C loc_E0C                          ; CODE XREF: switch1+Ej
.text:00000E0C                MOVS    R0, R1          ; jumptable 00000DFA case 2
.text:00000E0E                MULS    R0, R4
.text:00000E10 ; 19:        break;
.text:00000E10                B       def_DFA         ; jumptable 00000DFA default case
.text:00000E12 ; ---------------------------------------------------------------------------
.text:00000E12 ; 21:        result = v6 - a2;
.text:00000E12
.text:00000E12 loc_E12                          ; CODE XREF: switch1+Ej
.text:00000E12                SUBS    R0, R4, R1      ; jumptable 00000DFA case 1
.text:00000E14 ; 22:        break;
.text:00000E14                B       def_DFA         ; jumptable 00000DFA default case
.text:00000E16 ; ---------------------------------------------------------------------------
.text:00000E16 ; 24:        result = v6 / a2;
.text:00000E16
.text:00000E16 loc_E16                          ; CODE XREF: switch1+Ej
.text:00000E16                MOVS    R0, R4          ; jumptable 00000DFA case 3
.text:00000E18                BLX     __divsi3
.text:00000E1C ; 25:        break;
.text:00000E1C ; 26:      default:
.text:00000E1C                B       def_DFA         ; jumptable 00000DFA default case
.text:00000E1C ; End of function switch1
```

在这个函数中由于使用了 R4 作为局部变量，所以在开始时要把 R4 放入堆栈，为了返回后程序可以继续运行，所以把 LR 也压入了堆栈。这样堆栈地址就减去了 8 个字节(R4,LR 都被压入堆栈)。所以取第一个参数要使用指令 LDR R2, [SP,#8+arg_0]。

http://blog.csdn.net/jltxgcy/article/details/50598430

apk 文件分析原则

1. 如果在 dex 生成的 jar 文件里没有发现关键内容的话，就要注意 jar 里面的 native 函数以及 loadlibrary 操作，从而可以判断出加载了哪些 so，调用了什么函数。就不会出现判断不出是不是加载了某 so 的情况了。
2. IDA 调试的时候，只要勾选加载 so 时 pause，通过 so 路径，就能很容易判断出来是不是 apk 自带的 so 文件了，attach 是一种方法，但是很多情况下，关键点是在加载过程中的，很容易漏掉。

http://www.cnblogs.com/xunbu7/p/4314118.html

ida pro 6.1 调试步骤

如何使用 ida 在 apk 执行前动态调试其 so 中的函数

1 am start -D -n 包名/类名

2 IDA pro attach 进程， 设置新线程，加载 so 时断点，continue

3 打开 ddms， 查看调试端口， jdb attach port

4 这个时候应该已经断在新线程，或者加载 so 处了，在你感兴趣的 so 处停下来

5 函数偏移+so 基地址

另外用 ida 打开 so，查看你感兴趣的函数偏移， 加上你感兴趣的 so 的基地址
打上断点，continue， 就大功告成了。

6 so 取地址方法

IDA Pro 工具

Ctrl + S 然后搜索到 so 文件名
记录下基地址(base)

把 so 文件解压到本地，新开一个 ida 载入 so 文件
逆向出关键函数的偏移地址(FunOffset)

Android 自带的工具

Adb shell ps|grep 加载 apk 取到 pid

Adb shell cat /proc/pid/maps

如：
#cat /proc/11547/maps
输出：
.....
0076a000-0076c000 rwxp 0076a000 00:00 0
00bbe000-00bbf000 r-xp 00bbe000 00:00 0

00fcc000-00fcd000 r-xp 00000000 03:01 1238761 /root/test/gdbservertest/libtest.so
00fcd000-00fce000 rwxp 00000000 03:01 1238761 /root/test/gdbservertest/libtest.so
08048000-08049000 r-xp 00000000 03:01 1238765 /root/test/gdbservertest/test.exe
08049000-0804a000 rw-p 00000000 03:01 1238765 /root/test/gdbservertest/test.exe
.....

\toolchains\arm-linux-androideabi-4.4.3\prebuilt\windows\bin

由此可以知道：libtest.so 的代码在 00fcc000-00fcd000 之间。
arm-linux-androideabi-objdump.exe –d libtest.so > arm.txt
同上一样找到此函数偏移

计算出地址

Address=base+FunOffset

开始调试选项那个好像得针对 eng Build 的机型，置为 waitdebugger 啥的，我只是看了下理论但是没有实践过。

不过我最近也遇到这种问题，迫于时间压力没时间研究那个 waitdebugger 了，想了个相对好上手的办法：

1、解包对方 APK，插入一个：
对应 SMALI： android.os.SystemClock.sleep(20000);
const-wide/16 v0, 0x2710 #20 秒
invoke-static {v0, v(X-1)}, Landroid/os/SystemClock;->sleep(J)V
这里(X-1)对应.local X，我想你懂该怎么写的

2、另外有的包在你要调试的那个 SO 里面有签名保护，反正你重新打了包之后会导致程序运行崩溃，这个相比 JAVA 修改困难些，建议你用那个签名漏洞来打包。事实上我调试那个 SO 也遇到过这样，然后打了个签名漏洞的包嵌入的延时函数就大丈夫了

你是 windows 吧，
jdb -connect com.sun.jdi.SocketAttach:port=xxxx

IDA6.1 调试 Android JNI－STEP By STEP

步骤 1：通过”adb push $(IDA-DIR)\android_server /data/local/”安装服务程序；
步骤 2：通过”chmod 755 /data/local/android_server”更改服务程序执行权限；
步骤 3：以 root 方式运行/data/local/android_server(不以 root 的方式将获取不到 App 的进程列表);
步骤 4：在 PC 命令行执行”adb forward tcp:23946 tcp:23946”命令,让 IDA 可以连接本地端口进行远程调试；
步骤 5：启动一个 IDA-ANA，加载并分析需要调试的 JNI 动态库，比如 libgikir_demo.so；
步骤 6：在手机端启动加载上述 JNI 动态库的 App，比如 com.gikir.demo；

步骤 7：启动另一个 IDA-DBG，执行 Debugger/Attach/Remote ARMlinux/android debugger 菜单设置 Hostname 为 localhost，Port 为 23946 确定后在进程列表中找到 com.gikir.demo 进程，attach 并记下 pid，此时 会断在 libc.so 的模块空间中；
步骤 8：在 PC 命令行依次执行”adb shell”，”su”，”cat /proc/pid/maps”，找到 libgikir_demo.so 的基址，并记下 base;
步骤 9：在 IDA-ANA 中定位需要调试的函数偏移 offset；
步骤 10：在 IDA-DBG 中利用 g 命令跳转到需要调试的函数起始地址(base + offset)，并执行 c 命令得到汇编代码，下断点并 F9 运行；
步骤 11：在 App 中操作直到上述断点被击中；
步骤 12：Have Fun！

请教下，怎么断 so 入口函数呢
最简单的办法：将入口函数打上死循环的补丁，Attach 上去后改回原来的代码进行正常调试。

调试实战

菜鸟总结 so 分析，arm 汇编，IDA 静态分析

Wruih

ARM 简单介绍

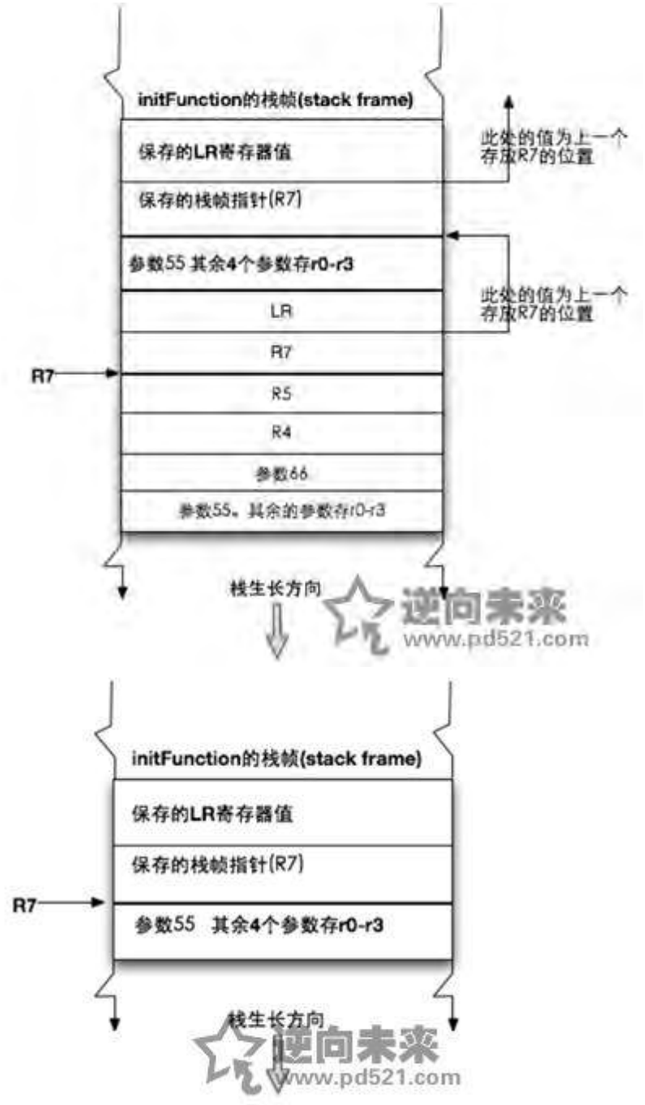
So 静态(arm 汇编，IDA 静态分析等）

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

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



这条指令(bl)对函数进行调用。请记住被调用函数需要的参数已经存储到相关的寄存器中了(r0 和 r1)。这条指令的执行一般被当做一个分支(branch)。可以理解为执行带链接的分支，也就是说，在跳转到分支之前，会将 lr(link register)的值设置为当前函数中将要执行的下一条指令，当从分支(被调函数)中返回时，通过 lr 中的值可以知道当前函数执行到哪里了。blx 中的 x 标示交换“exchange”，意思是如果有必要，处理器将对指令集模式进行切换。

返回值(存储在 r0 中）

mov r0, r1 => r0 = r1

mov r0, #10 => r0 = 10

ldr r0, [sp] => r0 = *sp

str r0, [sp] => *sp = r0

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

add r0, r1, r2 => r0 = r1 + r2

add r0, r1 => r0 = r0 + r1

push {r0, r1, r2} => 将 r0, r1 和 r2push 到栈中.

pop {r0, r1, r2} => 将 3 个值从栈中 pop 出来，并存放到 r0, r1 和 r2 中.

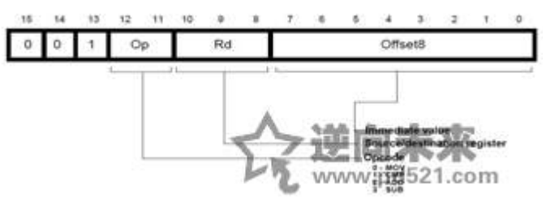
b_label => pc = _label

bl _label => lr = pc + 4; pc = _label

self 和_cmd 占用了 r0 和 r1 寄存器。它存储着当前执行方法的 selector

每次调用 Objective-C 方法时，都由 objc_msgSend 方法处理消息的派送。该方法根据传递的消息类型在类的方法列表中查找被调用方法的实现。objc_msgSend 方法：

id objc_msgSend(id self, SEL _cmd, ...)



MOV R1, #0

这里将 5 位 opcode 分成了两部分——前 3 位 001 是固定的，后 2 位用于标识 4 中不同的操作： mov, cmp, add, sub。所以 mov 指令的 opcode 二进制表示为 00100；这里

Rd 为 R1，所以 8~10 位为 001；同理， 0~7 为就 0000 0000。所以 MOVS R1, #0 的 2 进制表示为： 0010 0001 0000 0000 = 0x 21 00。

http://blog.csdn.net/zolovegd/article/details/1826192

http://blog.csdn.net/goooglеman/article/details/3758555（opcode 学习帖子）

so 库调试关键

.so 文件（shared object） linux 的动态链接库，

显示调用则是在主程序里使用 dlopen、dlsym、dlerror、dlclose 等系统函数。

1.IDA 计算出了成员变量的偏移地址并把 symbol 直接显示出来

IDA： __text:000026C4

mov ebx, ds:(_OBJC_IVAR_$_TestButton_m_model - 26C3h)[esi]

2.函数参数在 IDA 中被赋予名称

ebp+8 为 arg_0，ebp+12 为 arg_1。 arg 即为 argument 的缩写，第 n 个参数在+号后面的偏移量不是绝对的

。在函数开头和代码中，名称都会直接替换掉实际偏移量。基本上 arg_0 都是 self。

3.常数值型偏移地址被赋予名称

以 loc_为前缀。

IDA：jmp short loc_2732

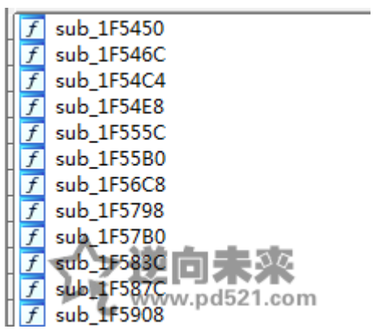
4.局部变量

即 ebp-xxx 会被命名为 var_xxx

搜索特征字符串。具体操作为：

①快捷键 Ctrl+S，打开搜索类型选择对话框-->双击 Strings，跳到字符串段-->菜单项“Search-->Text”；

②快捷键 Alt+T，打开文本搜索对话框，在 String 文本框中输入要搜索的字符串点击 OK 即可；



（C 的函数 ,抹掉了符号表）

So 动态调试

so 被加载之后最开始执行的是.init_array 段的代码。然后才会去执行 jni_onload 那么，在.init_array 处断下来便是很有必要的

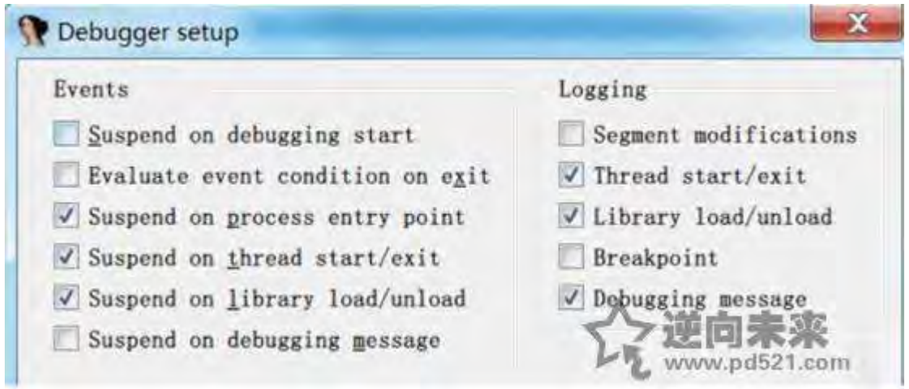
1.启动 android_server

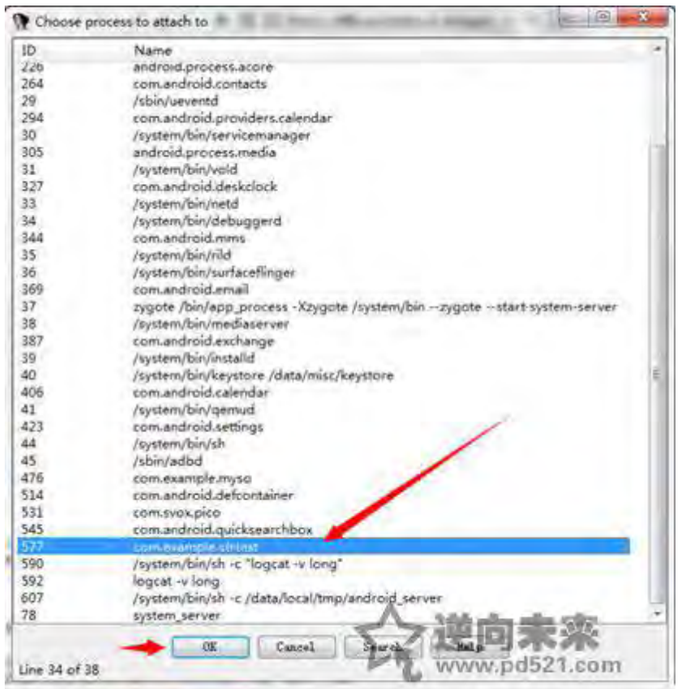
2.端口转发

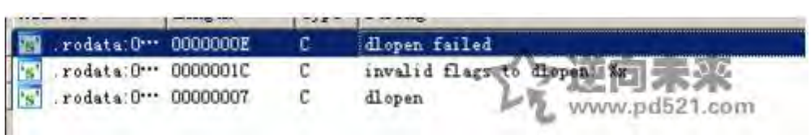
adb forward tcp:23946 tcp:23946

3.调试启动 adb shell am start-D-n com.scottgames.fnaf4/com.putaolab.ptsdk.activity.PTMainActivity

4.链接，下断点







Shift+F12 打开字符串窗口，搜索字符串： dlopen，找到 dlopen 函数的偏移 0xF30

动态调试的 IDA 中， G 跳转到： 400D3000+F30=400D3F30 处，下好断点

搜索字符串： calling

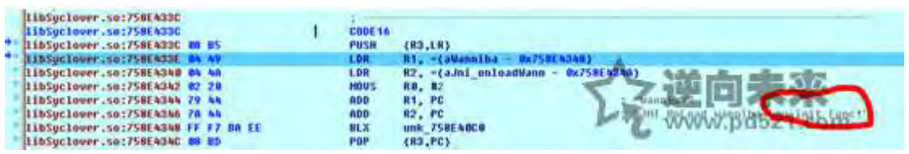
按 F9 运行

然后打开 Eclipse 或者 ddms

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

程序就会断在第一个断点处， F9 几次就段在 blx R4 处

F7 跟进就来到 init 段代码处：



1.IDA 用 32 位，

2 ./android_server 要 su

3 重启平板

4. <application android:allowBackup="true" android:debuggable="false" android:icon="@drawable/app_icon"

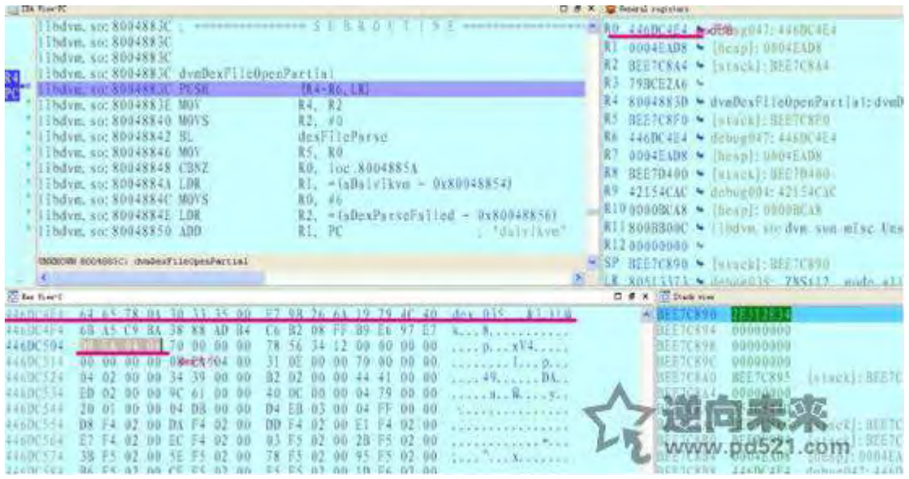
Dump

dvmLoadNativeCode 函数是加载和初始化 so 的函数， dvmDexFileOpenPartial 函数是对缓存 Dex 文件，该函数第一个参数就是解密后 dex 文件头内存地址，而第二个参数是该 dex 大小。

跳到 dvmDexFileOpenPartial 函数或 inflate 函数去下断,

int dvmDexFileOpenPartial(const void* addr, int len, DvmDex** ppDvmDex);

第一个参数就是 dex 内存起始地址，第二个参数就是 dex 大小。所以在这个函数下断点可以直接 dump 出明文 dex



static main(void)

{

auto fp, dex_addr, end_addr;

fp = fopen("D:\\dump.dex", "wb");

end_addr = r0 + r1;

for (dex_addr = r0; dex_addr < end_addr; dex_addr ++)

fputc(Byte(dex_addr), fp);

}


Execute script
Snippet list
Name
Default snippet *
Please enter script body
auto i;
fp = fopen("e:\\dump.dex", "wb");
for(i = 0xAA3C9008; i < 0xAA3C9008+0x19CF4; i++)
fputc(Byte(i),fp);
Line 1 of 1
Line 4 Column 23
Scripting language IDC
Tab size 4
Load...
Export...
Save
逆向未来
www.nd521.com

遇到反调试

```
__pid_t v2; // r0@1
FILE *v3; // r5@1
char *v4; // r0@3
signed int result; // r0@9
int v6; // [sp+4h] [bp-7Ch]@4
char s; // [sp+8h] [bp-78h]@1
int v8; // [sp+6Ch] [bp-14h]@1

v0 = off_4F60;
v1 = 1;
v8 = *(_DWORD *)off_4F60;
memset(&s, 0, 0x64u);
v2 = getpid();
sprintf(&s, "/proc/%d/status", v2);
v3 = fopen(&s, "r");
if ( v3 )
{
  while ( fgets(&s, 99, v3) )
  {
    v4 = strstr(&s, "TracerPid:");
    if ( v4 )
    {
      sscanf(v4 + 10, "%d", &v6);
      v1 = v6;
      if ( !v6 )
        goto LABEL_8;
      break;
    }
  }
  v1 = 1;
LABEL_8:
  fclose(v3);
```


先查 Pid　　ps -aux

cat /proc/xxxxx/status

就可以看 tracerPid 如果不为零 及被调试过

当程序打开进程成功后使用 fgets 获得信息 当获得如下信息进我们将其修改为 0，原因：底层会调用 libc 库中的 fopen 函数打开 so 文件句柄，然后通过 fgets 函数读取进程状态值，就可以通过修改读取到的状态值绕过调试进程检测。

```
C:\Users\asus>adb shell
root@C8815:/ # ps |grep com.dr
ps |grep com.dr
u0_a83    18593 145   310288 29892 ffffffff 403d3628 S com.droider.crackme0201
root@C8815:/ # cat /proc/18593/status
cat /proc/18593/status
Name:   der.crackme0201
State:  S (sleeping)
Tgid:   18593
Pid:    18593
PPid:   145
TracerPid:      0
Uid:    10083   10083   10083   10083
Gid:    10083   10083   10083   10083
FDSize: 256
Groups: 50083
```

```
libc.so:AFD18520
libc.so:AFD18520 fgets                                  ; CODE XREF: fgets
libc.so:AFD18520                                        ; DATA XREF: libex
libc.so:AFD18520 F0B5           PUSH            {R4-R7,LR}
libc.so:AFD18522 2DE9 F04F      PUSH.W          {R4-R10,LR}
libc.so:AFD18526 4680           MOV             R8, R0
libc.so:AFD18528 4614           MOV             R4, R2
libc.so:AFD1852A 1EC0           ITT GT
libc.so:AFD1852C F1B9 0900      ADDGT.W         R9, R1, #0xFFFFFFFF
libc.so:AFD18530 460D           MOVGT           R5, R0
UNKNOWN AFD18520: fgets
Hex View-1
BEE7CE84  54 72 61 63 65 72 50 69  64 3A 09 32 36 34 32 36  TracerPid:.26426
BEE7CE94  0A 00 0A 00 31 0A 00 00  00 00 00 00 00 00 00 00  ....1...........
BEE7CEA4  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
BEE7CEB4  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```


修改成 0（Thumb 00 20 ， 70 47

即 Mov R0,#0）

1.fopen—/proc/self/cmdline.debug.atrace.app_cmdlines

2.fgets—-包名

3.LoadNativeCode–加载 libexec.so

4.LoadNativeCode–加载 libexecmain.so

5.建立反调试线程（通过检查是否存在调试进程）

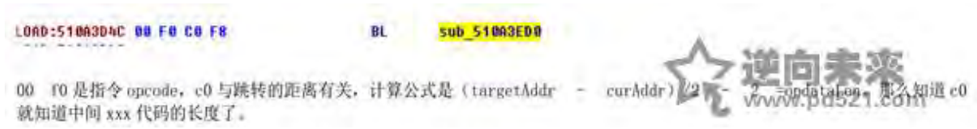
6.调用 fopen—-打开/proc/pid/status

7.调用 fgets—读取调试进程 pid

在 fgets 内部会调用 memchr 函数，和 memcpy 函数， memchr 函数完成以换行为分隔符， memcpy 将此次读取位置拷贝到目的缓冲区

脱壳

nop: 0xc046



INIT_ARRAY,JNI_OnLoad

壳入口 --> INIT_ARRAY--->解密第二层壳(JNI_OnLoad)---->解密原始 so 文件--->解压缩原始 so 的代码节。

http://www.52pojie.cn/thread-356096-1-1.html

__gnu_armfini_26 此函数是 ELF 的入口函数，此函数就完成了 jni_onload 和 verify 等的解密。

http://www.pd521.com/thread-790-1-1.html

对于 13 组 apk 的相关分析

谢邀。

最近工作挺忙，没咋顾上这些个比赛，上午看了下论坛，发现终于有一个移动方面的分析题了，下载下来中午看了下，apk 本身是有防护的，下午几个朋友告诉我，Hmily 竟然圈我了，好吧，跟他联系了下，说还没人破解呢，现在吃饭的时候我先写下分析文章吧，待会儿继续加班忙了。

0x1:拿到样本

拿到样本 apk，首先是对其反编译，查看下大致软件流程。

如下图：

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

public MainActivity() {
    super();
}

static TextView access$0(MainActivity arg1) {
    return arg1.edt1;
}

static TextView access$1(MainActivity arg1) {
    return arg1.edt2;
}

protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    this.setContentView(2130903064);
    this.edt1 = this.findViewById(2131034172);
    this.edt2 = this.findViewById(2131034173);
    this.btn = this.findViewById(2131034174);
    this.btn.setOnClickListener(new View$OnClickListener() {
        public void onClick(View arg0) {
            MainActivity.this.verify(MainActivity.this.edt1.getText().toString(), MainActivity.this
                    .edt2.getTe().toString());
        }
    });
}

public native void verify(String arg1, String arg2) {
}
}
```

所用的lib

关键函数

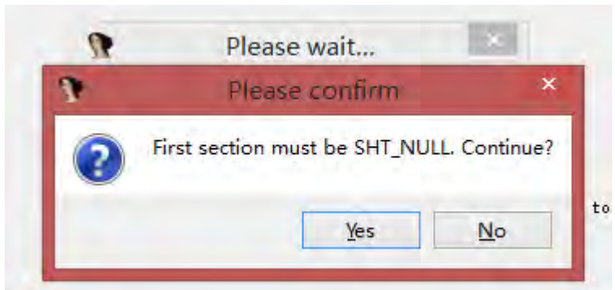
通过上图简单的标记可以明白，此 apk 是首先加载一个 libverify.so，然后在输入用户名和密码后，通过此 so 里面的函数 verify 函数来进行对比和处理。

0x2:开始分析

通过第一个步骤，我们可以有思路的来分析这个软件。

关键方法在 so 里面，那么我们挂起 IDA：

在加载 so 的时候提示:



我们先忽略，继续下去，看是否影响分析。

进去之后找 verify 函数:

```
340
340                 EXPORT verify
340 verify                                  ; DATA XREF: LOAD:0000500C↓o
340
340 var_1C          = -0x1C
340
340                 PUSH        |   {R0-R2,R4-R7,LR}
342                 MOVS            R5, R0
344                 MOVS            R6, R1
346                 MOVS            R7, R2
348                 STR             R3, [SP,#0x20+var_1C]
34A                 BL              sub_12A0
34E                 CMP             R0, #0
350                 BNE             locret_1364
352                 LDR             R3, =(off_4F64 - 0x135C)
354                 MOVS            R0, R5
356                 MOVS            R1, R6
358                 ADD             R3, PC ; off_4F64
35A                 LDR             R3, [R3]
35C                 MOVS            R2, R7
35E                 LDR             R4, [R3]
360                 LDR             R3, [SP,#0x20+var_1C]
362                 BLX             R4
364
364 locret_1364                             ; CODE XREF: verify+10↑j
364                 POP             {R0-R2,R4-R7,PC}
364 ; End of function verify
```

好，可以正常显示函数，那么我们就开始分析这个算法。

有 F5 插件，我们就这样很轻松的来看他的结构了：

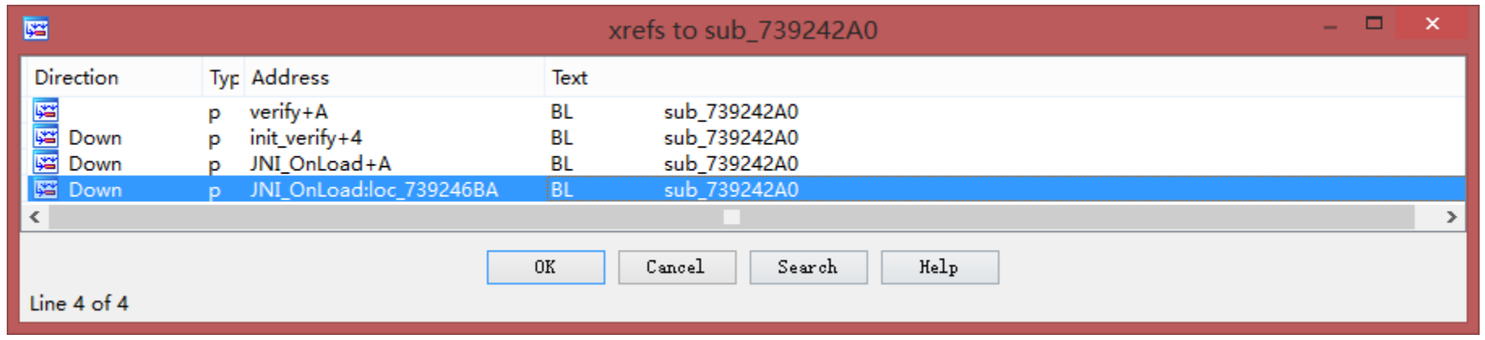
```
unsigned __int64 __fastcall verify(unsigned int a1, int a2, int a3, unsigned int a4)
{
  unsigned int v4; // r5@1
  int v5; // r6@1
  int v6; // r7@1
  unsigned __int64 v8; // [sp+0h] [bp-20h]@1

  v8 = __PAIR__(a4, a1);
  v4 = a1;
  v5 = a2;
  v6 = a3;
  if ( !sub_12A0() )
    ((void (__fastcall *)(unsigned int, int, int, _DWORD))*off_4F64)(v4, v5, v6, HIDWORD(v8));
  return v8;
}
```

可以看到，此方法里需要注意的，就是这个判断，然后下面那行执行的代码，所以，那个 if 判断我们是必须要注意的，点开那个 sub_12A0 方法看一下：

```
signed int __fastcall sub_12A0()
{
  void *v0; // r6@1
  signed int v1; // r4@1
  __pid_t v2; // r0@1
  FILE *v3; // r5@1
  char *v4; // r0@3
  signed int result; // r0@9
  int v6; // [sp+4h] [bp-7Ch]@4
  char s; // [sp+8h] [bp-78h]@1
  int v8; // [sp+6Ch] [bp-14h]@1

  v0 = off_4F60;
  v1 = 1;
  v8 = *(_DWORD *)off_4F60;
  memset(&s, 0, 0x64u);
  v2 = getpid();
  sprintf(&s, "/proc/%d/status", v2);
  v3 = fopen(&s, "r");
  if ( v3 )
  {
    while ( fgets(&s, 99, v3) )
    {
      v4 = strstr(&s, "TracerPid:");
      if ( v4 )
      {
        sscanf(v4 + 10, "%d", &v6);
        v1 = v6;
        if ( !v6 )
          goto LABEL_8;
        break;
      }
    }
    v1 = 1;
LABEL_8:
    fclose(v3);
```

通过读取/proc/进程/status 这个文件，通过 fgets 来获取里面的数值，如果 TracerPid:有数据，则说明其当前进程被调试，然后执行关闭方法，这样以后再执行相关方法断开调试链接。这样是来防止程序被 IDA 或者 GDB 之类的工具调试的。

查看此方法的被调用列表：



发现在 JNI_Onload 里面就有，所以如果调试，肯定是会出问题的，断开连接。。

所以通过这个 sub_12A0 的方法内容，我们可以控制其返回值为 0，则证明当前不在调试状态，所以此时，IDA 之类工具可以保持正常连接。

（TracerPid 的方式使用，在爱加密上一个版本有使用，有兴趣的小朋友，可以看论坛我发的那个安卓学习手册 dex 脱壳视频，有详细步骤）

0x3:动态调试

经过步骤 2，在 verify 方法中没明显算法，那我们进行动态调试看看。分析了上面那个 TracePid 所在的方法，我们在 16 进制下将方法便宜找到 0x12A0，然后在 16 进制编辑器里面修改，方法在 IDA 的 HexView 看了下，现在是 Thumb 指令，所以我们直接写　00 20　，　70 47

即 Mov R0,#0　　; Bx Lr .

至于为何写这个，自己想，百度搜搜 ARM 和 Thumb 的指令，多练习几个 so 就明白了。

然后我们替换掉我们修改的 so，签名 apk，安装到手机上。

简单截图下调试准备：

1.启动 android_server

2.端口转发

adb forward tcp:23946 tcp:23946

3.调试启动 apk

adb shell am start -D -n com.example.crackme52/com.example.crackme52.MainActivity

4.链接，下断点

```
libc.so:4013BF6C
libc.so:4013BF6C
libc.so:4013BF6C fopen
libc.so:4013BF6C
libc.so:4013BF6C var_1C= -0x1C
libc.so:4013BF6C
libc.so:4013BF6C STMFD           SP!, {R4-R7,LR}
libc.so:4013BF70 SUB             SP, SP, #0xC
libc.so:4013BF74 MOV             R7, R0
libc.so:4013BF78 LDR             R5, =(unk_40179FDC - 0x4013BF94)
libc.so:4013BF7C MOV             R0, R1
libc.so:4013BF80 ADD             R1, SP, #0x20+var_1C
libc.so:4013BF84 BL              __sflags
libc.so:4013BF88 SUBS            R6, R0, #0
libc.so:4013BF8C ADD             R5, PC, R5 ; unk_40179FDC
```

步骤 2 中看到是以 fopen 来打开 status 文件，那我们断在这里看是否还会执行。

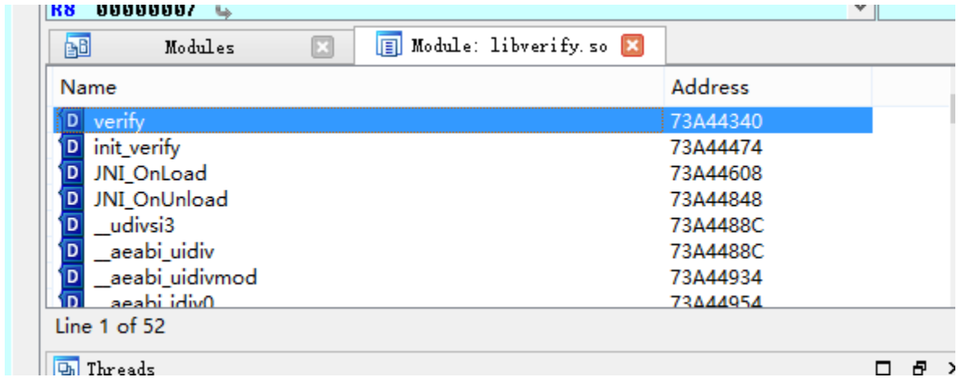
4.继续执行

```
C:\Users\muzahiwan520>adb forward tcp:8899 jdwp:13891

C:\Users\muzahiwan520>jdb -connect com.sun.jdi.SocketAttach:port=8899,hostname=l
ocalhost
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...
> _
```

好了，然后我们 F9，在 IDA 窗口下继续分析。

一路 F9 下去，观察寄存器 R0 的数值（所打开的文件），然后到最后发现 IDA 没有断开链接退出。

那么，我们的目的就达到了，然后在 Debugger-->DebuggerView-->ModelList 里面找到 libverfiry.so，即程序的 so，双击此 so, ok，很直接，可以一眼看到 verfy 方法，我们双击打开



打开后发现一堆乱码：

```
libverify.so:73A4433F DCB    0
libverify.so:73A44340 verify DCB 0xF7 ;
libverify.so:73A44341 DCB 0xB5 ;
libverify.so:73A44342 DCB    5
libverify.so:73A44343 DCB 0x1C
libverify.so:73A44344 DCB  0xE
libverify.so:73A44345 DCB 0x1C
libverify.so:73A44346 DCB 0x17
libverify.so:73A44347 DCB 0x1C
libverify.so:73A44348 DCB    1
libverify.so:73A44349 DCB 0x93 ;
libverify.so:73A4434A DCB 0xFF
libverify.so:73A4434B DCB 0xF7 ;
libverify.so:73A4434C DCB 0xA9 ;
libverify.so:73A4434D DCB 0xFF
libverify.so:73A4434E DCB    0
libverify.so:73A4434F DCB 0x28 ; (
libverify.so:73A44350 DCB    8
libverify.so:73A44351 DCB 0xD1 ;
libverify.so:73A44352 DCB    5
libverify.so:73A44353 DCB 0x4B ; K
libverify.so:73A44354 DCB 0x28 ; (
libverify.so:73A44355 DCB 0x1C
libverify.so:73A44356 DCB 0x31 ; 1
libverify.so:73A44357 DCB 0x1C
```

不要着急，我们选中一大块按下键盘 C：

```
libverify.so:73A44340 ; ---------------------------------------------------------------------------
libverify.so:73A44340
libverify.so:73A44340 verify
libverify.so:73A44340 PUSH            {R0-R2,R4-R7,LR}
libverify.so:73A44342 MOVS            R5, R0
libverify.so:73A44344 MOVS            R6, R1
libverify.so:73A44346 MOVS            R7, R2
libverify.so:73A44348 STR             R3, [SP,#4]
libverify.so:73A4434A BL              sub_73A442A0
libverify.so:73A4434E CMP             R0, #0
libverify.so:73A44350 BNE             locret_73A44364
libverify.so:73A44352 LDR             R3, =(dword_73A47F64 - 0x73A4435C)
libverify.so:73A44354 MOVS            R0, R5
libverify.so:73A44356 MOVS            R1, R6
libverify.so:73A44358 ADD             R3, PC ; dword_73A47F64
libverify.so:73A4435A LDR             R3, [R3]
libverify.so:73A4435C MOVS            R2, R7
libverify.so:73A4435E LDR             R4, [R3]
libverify.so:73A44360 LDR             R3, [SP,#4]
libverify.so:73A44362 BLX             R4
libverify.so:73A44364
libverify.so:73A44364 locret_73A44364                         ; CODE XREF: libverify.so:verify+10↑j
libverify.so:73A44364 POP             {R0-R2,R4-R7,PC}
libverify.so:73A44364 ; ---------------------------------------------------------------------------
libverify.so:73A44366 DCB 0xC0 ;
libverify.so:73A44367 DCB 0x46 ; F
libverify.so:73A44368 off_73A44368 DCD dword_73A47F64 - 0x73A4435C
libverify.so:73A44368                                         ; DATA XREF: libverify.so:verify+12↑r
```

发现一切都好了，原因是当时ＩＤＡ只加载，而没有分析ｓｏ的内容，所以我们手动执行命令即可。

然后断点下。

```
.so:73A44340 verify
.so:73A44340
.so:73A44340 var_1C= -0x1C
.so:73A44340
.so:73A44340 PUSH            {R0-R2,R4-R7,LR}
.so:73A44342 MOVS            R5, R0
.so:73A44344 MOVS            R6, R1
.so:73A44346 MOVS            R7, R2
.so:73A44348 STR             R3, [SP,#0x20+var_1C]
.so:73A4434A BL              sub_73A442A0
.so:73A4434E CMP             R0, #0
.so:73A44350 BNE             locret_73A44364
.so:73A44352 LDR             R3, =(dword_73A47F64 - 0x73A4435C)
.so:73A44354 MOVS            R0, R5
.so:73A44356 MOVS            R1, R6
.so:73A44358 ADD             R3, PC ; dword_73A47F64
.so:73A4435A LDR             R3, [R3]
.so:73A4435C MOVS            R2, R7
.so:73A4435E LDR             R4, [R3]
.so:73A44360 LDR             R3, [SP,#0x20+var_1C]
.so:73A44362 BLX             R4
.so:73A44364
.so:73A44364 locret_73A44364                         ; CODE XREF: verify+10↑j
.so:73A44364 POP             {R0-R2,R4-R7,PC}
.so:73A44364 ; End of function verify
```

可以根据上面我们静态加载 so 的分析，这里关键点在于 R4 的内容，那么我们 F8 单步到 BLX R4　上面那一行，然后 F7 步入 R4 进去看一下。

0x4:简单分析算法

经历步骤 3，我们下好断点后，继续手机上操作，输入用户名和密码，

然后断点停住，我们走进 R4.

```
libverify.so:73A4435A LDR             R3, [R3]
libverify.so:73A4435C MOVS            R2, R7
libverify.so:73A4435E LDR             R4, [R3]
PC libverify.so:73A44360 LDR          R3, [SP,#0x20+var_1C]
libverify.so:73A44362 BLX             R4
libverify.so:73A44364
```

```
debug083:73A4CCD3 DCB     0
debug083:73A4CCD4 ; ---------------------------------------------------------------------------
debug083:73A4CCD4 CODE16
debug083:73A4CCD4 PUSH            {R4-R7,LR}              ; DATA XREF: libverify.so:__eabi_Unwind_cpp_ptr384↑o
debug083:73A4CCD6 LDR             R4, =(dword_73A4FFB0 - 0x73A4CCE0)
debug083:73A4CCD8 SUB             SP, SP, #0x74
debug083:73A4CCDA STR             R1, [SP,#0x10]
debug083:73A4CCDC ADD             R4, PC ; dword_73A4FFB0
debug083:73A4CCDE LDR             R4, [R4]
debug083:73A4CCE0 LDR             R1, =(a0123456789qwer - 0x73A4CCEE)
debug083:73A4CCE2 STR             R3, [SP,#0xC]
debug083:73A4CCE4 LDR             R3, [R4]
debug083:73A4CCE6 MOVS            R5, R0
debug083:73A4CCE8 STR             R2, [SP,#8]
debug083:73A4CCEA ADD             R1, PC                  ; "0123456789QWERTYUIOPASDFGHJKLZXCVBNMqwe"...
debug083:73A4CCEC MOVS            R2, #0x3F
debug083:73A4CCEE ADD             R0, SP, #0x2C
debug083:73A4CCF0 STR             R3, [SP,#0x6C]
debug083:73A4CCF2 BLX             unk_73A4CC5C
debug083:73A4CCF6 LDR             R3, =0x213F2E2C
debug083:73A4CCF8 MOVS            R6, #0
debug083:73A4CCFA MOVS            R7, #0xA8
debug083:73A4CCFC STR             R3, [SP,#0x18]
debug083:73A4CCFE ADD             R3, SP, #0x18
debug083:73A4CD00 STRB            R6, [R3,#4]
debug083:73A4CD02 LDR             R3, [R5]
debug083:73A4CD04 LSLS            R7, R7, #2
debug083:73A4CD06 MOVS            R0, R5
debug083:73A4CD08 LDR             R3, [R3,R7]
debug083:73A4CD0A LDR             R1, [SP,#0xC]
debug083:73A4CD0C BLX             R3
debug083:73A4CD0E STR             R4, [SP,#0x14]
```

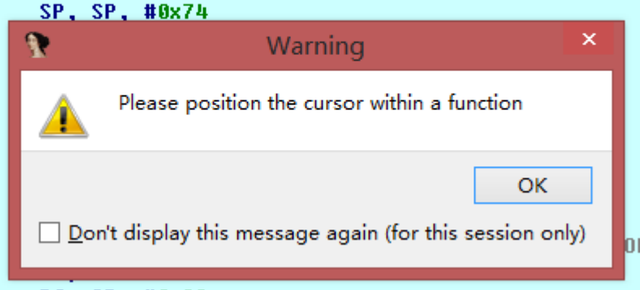
这个时候，我们简单看一下上下文：

```
ug083:73A4CDA4 LDR             R3, [R5]
ug083:73A4CDA6 LDR             R1, =(aAndroidWidgetT - 0x73A4CDB0)
ug083:73A4CDA8 MOVS            R0, R5
ug083:73A4CDAA LDR             R3, [R3,#0x18]
ug083:73A4CDAC ADD             R1, PC                  ; "android/widget/Toast"
ug083:73A4CDAE BLX             R3
ug083:73A4CDB0 SUBS            R6, R0, #0
ug083:73A4CDB2 BEQ             loc_73A4CE1A
ug083:73A4CDB4 LDR             R0, [R5]
ug083:73A4CDB6 LDR             R2, =(aMaketext - 0x73A4CDC4)
ug083:73A4CDB8 LDR             R3, =(aLandroidConten - 0x73A4CDC6)
ug083:73A4CDBA MOVS            R1, #0x1C4
ug083:73A4CDBE LDR             R4, [R0,R1]
ug083:73A4CDC0 ADD             R2, PC                  ; "makeText"
ug083:73A4CDC2 ADD             R3, PC                  ; "(Landroid/content/Context;Ljava/lang/Ch"...
ug083:73A4CDC4 MOVS            R0, R5
ug083:73A4CDC6 MOVS            R1, R6
ug083:73A4CDC8 BLX             R4
ug083:73A4CDCA SUBS            R4, R0, #0
ug083:73A4CDCC BEQ             loc_73A4CE1A
ug083:73A4CDCE LDR             R3, [R5]
ug083:73A4CDD0 MOVS            R2, #0x1C8
ug083:73A4CDD4 LDR             R1, =(aCongratulation - 0x73A4CDE2)
ug083:73A4CDD6 LDR             R7, [R3,R2]
ug083:73A4CDD8 MOVS            R2, #0x29C
ug083:73A4CDDC LDR             R3, [R3,R2]
ug083:73A4CDDE ADD             R1, PC                  ; "Congratulation! You crack it!"
ug083:73A4CDE0 MOVS            R0, R5
ug083:73A4CDE2 BLX             R3
ug083:73A4CDE4 MOVS            R3, #0
ug083:73A4CDE6 STR             R0, [SP]
```

有弹出的 Toast，这样，就很简单明白了，注册成功后，是会弹出一个这样的字符的，所以，核心对比代码，就在这里。

然后我们使用 F5 功能，提示:

```
73A4CCD4 ; ---------------------------------------------------------------------------
73A4CCD4 CODE16
73A4CCD4 PUSH            {R4-R7,LR}              ; DATA XREF: libverify.so:__eabi_Unwi
73A4CCD6 LDR             R4, =(dword_73A4FFB0 - 0x73A4CCE0)
73A4CCD8 SUB             SP, SP, #0x74
73A4CCDA STR
73A4CCDC ADD
73A4CCDE LDR
73A4CCE0 LDR
73A4CCE2 STR
73A4CCE4 LDR
73A4CCE6 MOVS
73A4CCE8 STR
73A4CCEA ADD                                                    OPASDFGHJKLZXCVB
73A4CCEC MOVS
73A4CCEE ADD             R0, SP, #0x2C
```



因为传入参数没解析，IDA 不认为这是一个连贯的算法，所以我们在第一行，也就是 push 那一行，按下字母 P 键，

```
:73A4CCD4
:73A4CCD4 sub_73A4CCD4                            ; DATA X
:73A4CCD4
:73A4CCD4 var_88= -0x88
:73A4CCD4 var_84= -0x84
:73A4CCD4 var_80= -0x80
:73A4CCD4 var_7C= -0x7C
:73A4CCD4 var_78= -0x78
:73A4CCD4 var_74= -0x74
:73A4CCD4 var_70= -0x70
:73A4CCD4 var_68= -0x68
:73A4CCD4 var_5C= -0x5C
:73A4CCD4 var_1C= -0x1C
:73A4CCD4
:73A4CCD4 PUSH            {R4-R7,LR}
:73A4CCD6 LDR             R4, =(dword_73A4FFB0 - 0x73A4CCE
```

此时，我们再按 F5,就是很明显的算法了。

剩下的，根据我们输入的账户和密码，继续跟下去就行了。。

```
v28 = _stack_chk_guard;
((void (__fastcall *)(_DWORD, _DWORD, signed int))unk_73A4CC5C)(
  v27,
  "0123456789QWERTYUIOPASDFGHJKLZXCVBNMqwertyuiopasdfghjklzxcvbnm",
  63);
v5 = 0;
v24 = 557788716;
v25 = 0;
result = (*(int (__fastcall **)(int, int))(*(_DWORD *)v4 + 672))(v4, v22);// 输入字符的长度
if ( result == 12 )
{
  result = (*(int (__fastcall **)(int, int))(*(_DWORD *)v4 + 672))(v4, v21) - 8;// 这里输入12长度的，result=12-8=4
  if ( (unsigned int)result <= 0xC )
  {
    v7 = (*(int (__fastcall **)(int, int))(*(_DWORD *)v4 + 672))(v4, v21);// v7=12，我们输入的长度
    v8 = (*(int (__fastcall **)(int, int, _DWORD))(*(_DWORD *)v4 + 676))(v4, v21, 0);// 我输入的账号，12个数字1
    v9 = 0;
    if ( v7 <= 12 )                         // v7上面计算为12，进入此判断流程
    {
      while ( v9 < v7 )                     // v9为0，与v7比较，进入循环
      {
        v26[v9] = *(_BYTE *)(v8 + v9);      // v26[v9]为我们输入的账号，这里我输入的为12个1
        ++v9;
      }
      v10 = (char *)&v24 - v7;              // v10为感叹号,英文 !
      while ( v7 != 12 )
      {
        v26[v7] = v10[v7];
        ++v7;
      }
    }
    else
    {
      do
```

后面还有计算密码的一个算法，F7 跟进去，慢慢搞吧

```
debug083:73A4CE58
debug083:73A4CE58 loc_73A4CE58                            ; CODE XREF: debug083:73A4
debug083:73A4CE58 EOR             R12, R0, R1
debug083:73A4CE5C RSBMI           R1, R1, #0
debug083:73A4CE60 SUBS            R2, R1, #1
debug083:73A4CE64 BEQ             loc_73A4CEE8
debug083:73A4CE68 MOVS            R3, R0
debug083:73A4CE6C RSBMI           R3, R0, #0
debug083:73A4CE70 CMP             R3, R1
debug083:73A4CE74 BLS             loc_73A4CEF4
debug083:73A4CE78 TST             R1, R2
debug083:73A4CE7C BEQ             loc_73A4CF04
debug083:73A4CE80 CLZ             R2, R1
debug083:73A4CE84 CLZ             R0, R3
debug083:73A4CE88 SUB             R0, R2, R0
debug083:73A4CE8C MOV             R2, #1
debug083:73A4CE90 MOV             R1, R1,LSL R0
debug083:73A4CE94 MOV             R2, R2,LSL R0
debug083:73A4CE98 MOV             R0, #0
debug083:73A4CE9C
debug083:73A4CE9C loc_73A4CE9C                            ; CODE XREF: debug083:73A4
debug083:73A4CE9C CMP             R3, R1
debug083:73A4CEA0 SUBCS           R3, R3, R1
debug083:73A4CEA4 ORRCS           R0, R0, R2
debug083:73A4CEA8 CMP             R3, R1,LSR#1
debug083:73A4CEAC SUBCS           R3, R3, R1,LSR#1
debug083:73A4CEB0 ORRCS           R0, R0, R2,LSR#1
debug083:73A4CEB4 CMP             R3, R1,LSR#2
debug083:73A4CEB8 SUBCS           R3, R3, R1,LSR#2
debug083:73A4CEBC ORRCS           R0, R0, R2,LSR#2
debug083:73A4CEC0 CMP             R3, R1,LSR#3
debug083:73A4CEC4 SUBCS           R3, R3, R1,LSR#3
debug083:73A4CEC8 ORRCS           R0, R0, R2,LSR#3
```

记得在跳转指令和方法的时候，尽量在汇编下，不要在 F5 下

```
73      v11 = 0;
74      v12 = (*(int (__fastcall **)(int, int, _DWORD))(*(_DWORD *)v4 + 676))(v4, v22, 0);// v12为输入的密码
75      v13 = 11;
76      while ( 1 )
77      {
78        v11 += (unsigned __int8)v26[v13];       // 0x31
79        result = ((int (__fastcall *)(int, signed int))loc_73A4CF2C)(v11, 62);// 这里是将密码进行对比了
80        if ( (unsigned __int8)v27[v14] != *(_BYTE *)(v12 + v13) )// 输入的密码v12+固定值v13,把这个分析了，就完了，有兴趣的，自己加油啊，我加班呢
81          break;
82        v15 = (unsigned int)v13-- >= 1;
83        if ( !v15 )
84        {
85          result = (*(int (__fastcall **)(int, _DWORD))(*(_DWORD *)v4 + 24))(v4, "android/widget/Toast");
86          v16 = result;
87          if ( result )
88          {
89            result = (*(int (__fastcall **)(int, int, _DWORD, _DWORD))(*(_DWORD *)v4 + 452))(
```

关键的比较之处在这里，在下面 16 进制下修改内存，将会直接弹出注册成功的提示，也算爆破吧。

```
debug078:7392CD92 BLX             loc_7392CF2C
debug078:7392CD96 ADD             R3, SP, #0x88+var_5C
debug078:7392CD98 LDRB            R2, [R3,R1]
debug078:7392CD9A LDRB            R3, [R7,R4]
debug078:7392CD9C CMP             R2, R3
debug078:7392CD9E MOVS            R0, R0
debug078:7392CDA0 SUBS            R4, #1
debug078:7392CDA2 BCS             loc_7392CD88
debug078:7392CDA4 LDR             R3, [R5]
debug078:7392CDA6 LDR             R1, =(aAndroidWidgetT - 0x7392CDB0)
debug078:7392CDA8 MOVS            R0, R5
debug078:7392CDAA LDR             R3, [R3,#0x18]
debug078:7392CDAC ADD             R1, PC                  ; "android/widget/Toast"
debug078:7392CDAE BLX             R3
debug078:7392CDB0 SUBS            R6, R0, #0
debug078:7392CDB2 BEQ             loc_7392CE1A
debug078:7392CDB4 LDR             R0, [R5]
debug078:7392CDB6 LDR             R2, =(aMaketext - 0x7392CDC4)
debug078:7392CDB8 LDR             R3, =(aLandroidConten - 0x7392CDC6)

UNKNOWN 7392CD9E: sub_7392CCD4+CA
```

Hex View-R3

```
7392CD5E  01 33 A3 42 F9 DB 06 AB  1B 1B 03 E0 19 5D 08 AA  .3.B.........]..
7392CD6E  A1 54 01 34 0C 2C F9 D1  2B 68 28 1C 03 99 DB 59  .T.4.,..+h(....Y
7392CD7E  00 22 98 47 00 26 07 1C  0B 24 08 AB E3 5C 3E 21  ."·G.&...$...\>!
7392CD8E  F8 48 30 1C 00 F0 CC E8  0B AB 5A 5C 3B 5D 9A 42  ..0.......Z\;].B
7392CD9E  00 00 01 3C F1 D2 2B 68  24 49 28 1C 9B 69 79 44  ...<..+h$I(..iyD
7392CDAE  98 47 06 1E 32 D0 28 68  21 4A 21 4B E2 21 49 00  .G..2.(h!J!K.!I.
7392CDBE  44 58 7A 44 7B 44 28 1C  31 1C A0 47 04 1E 25 D0  DXzD{D(.1..G..%.
7392CDCE  2B 68 E4 22 52 00 1B 49  9F 58 A7 22 92 00 9B 58  +h."R..I.X."...X
7392CDDE  79 44 28 1C 98 47 00 23  00 90 01 93 28 1C 31 1C  yD(..G.#....(.1.
7392CDEE  22 1C 04 9B B8 47 07 1E  10 D0 29 68 13 4A 13 4B  "....G....)h.J.K
7392CDFE  08 31 7A 44 7B 44 CC 6F  28 1C 31 1C A0 47 2B 68  .1zD{D.o(.1..G+h
7392CE0E  02 1C 39 1C F4 33 1B 68  28 1C 98 47 05 99 1B 9A  ..9..3.h(..G....
7392CE1E  0B 68 9A 42 01 D0 FF F7  20 EF 1D B0 F0 BD D0 32  .h.B.... ·......2
7392CE2E  00 00 52 18 00 00 2C 2E  3F 21 00 17 00 00 01 17  ..R...,.?!......
7392CE3E  00 00 08 17 00 00 37 17  00 00 33 17 00 00 36 17  ......7...3...6.
```

。。。。

附上调试中的那个用户名：

111111111111

Xix7ABabQFqf

剩下的，就不用多说了吧。。。。

大家继续。。。

我继续加班去了。。。

跟以往惯例一样，不多提供代码，有兴趣的，继续分析吧。。。

此文仅供分析参考吧，请不必计分，不算比赛内容。。。

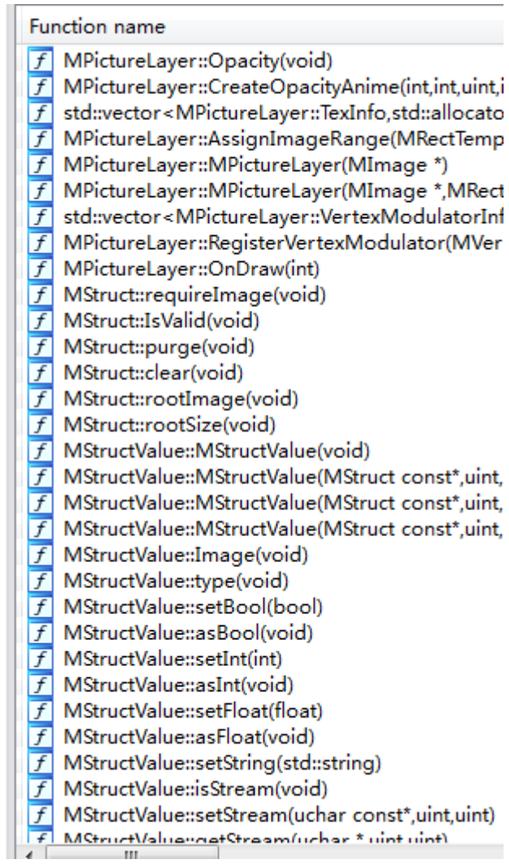
仅谢 Hmily 的邀请。。。

听鬼哥说故事

Android 逆向 so 文件，调试加解读

标 题: 【原创】Android 逆向 so 文件，调试加解读
作 者: luyangliu
时 间: 2014-10-16,19:10:19
链 接: http://bbs.pediy.com/showthread.php?t=193353

本科刚毕业，实在无聊想学学破解，还是个弱菜，接触 ida 还不到 2 个月，破解程序都不到 5 个，但是还是很努力的在玩破解，无聊时看到这个 apk，开始这个文件的破解。
apk 下载地址 http://pan.baidu.com/s/1eQ29ypk
这是个 android 程序，程序本事很大，主要是里面的 .psb.m 文件（请教大神这是啥引擎），这个文件是被压缩后解密的，非常恶心，大概源文件用 psb 开头 压缩后用(psb 开头 加密后用 mfl。
程序拿到手，弱菜（就是我）第一反应是 dex2jar-0.0.9.15 后 jd-gui.exe 然后配合 apktool 修改，可是看到 jar 里面没啥值钱的东西~~~.psb.m 这字段都木有，也不在 string 里面，怀疑是 so 文件内实现的~~~~（非常不敢怀疑 so 文件比 smali 难多了），打开 lib-main.so（非常大）



里面很像加密解密，所有文件也在数据段里面有~~~只能硬着头皮看了~~~里面疑似加密解密的文件太多~~看需要花好长时间~~~看了半天一个函数（具体不好意思说），发现不是相关代码~~~决定开始动态调试~~~

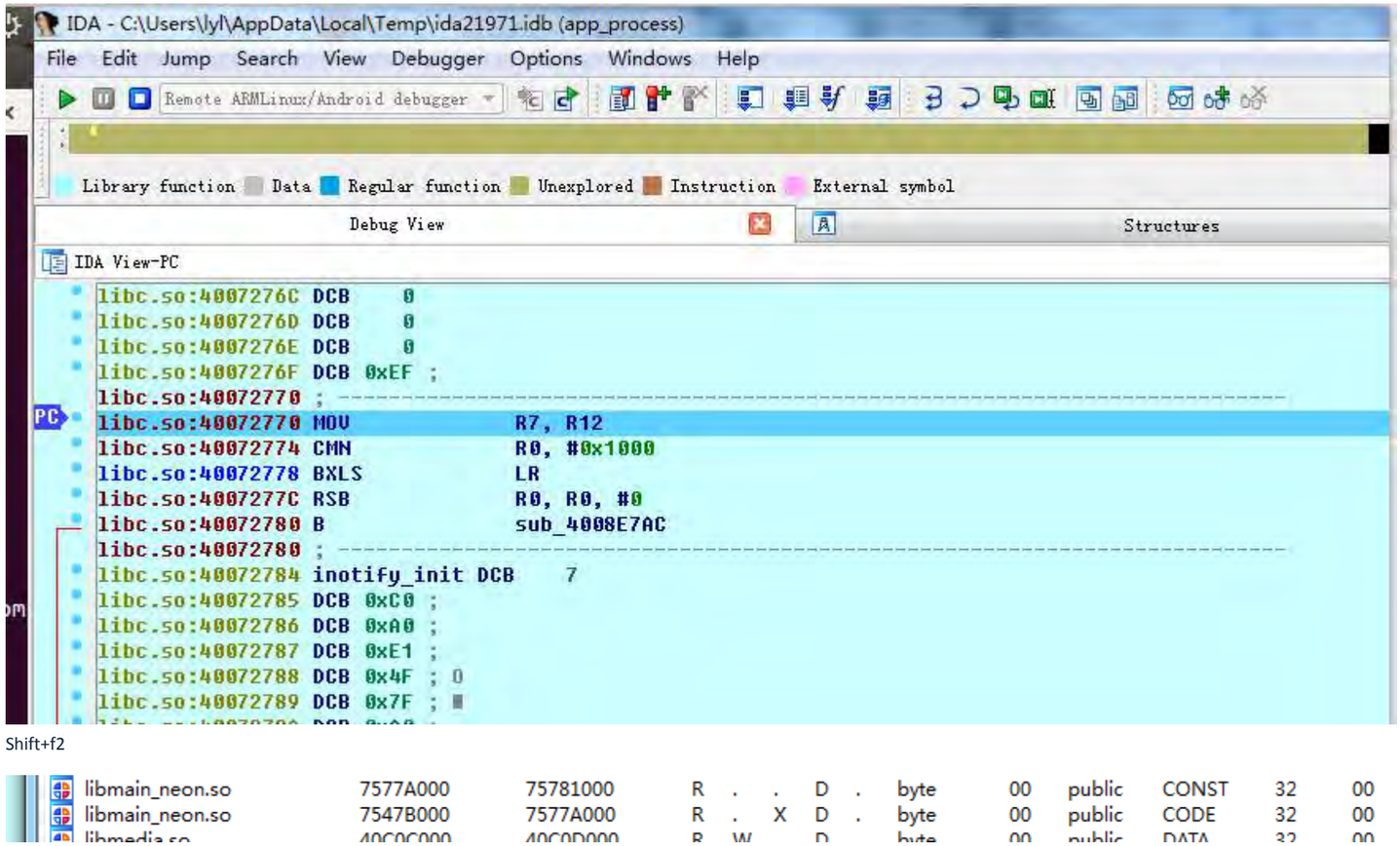
一开始准备用下面这个方法调试 http://bbs.pediy.com/showthread.php?t=178659&highlight=android+%E5%90%AF%E5%8A%A8+%E5%8A%A8%E5%89%8D。发现进不去，根本就没有 lib-main.so，是不是没有加载，这个还请大神解答

我赌他不是一开始就把所有文件处理完，所以用《已解答: 关于用 ida 远程调试 android native c 的 so 文件的问题》 这个帖子的方法

```
C:\check>adb remount
remount succeeded

C:\check>adb push android_server /system/bin/
4359 KB/s (566944 bytes in 0.127s)

C:\check>adb shell chmod 777 /system/bin/android_server

C:\check>adb forward tcp:23946 tcp:23946

C:\check>adb shell android_server
IDA Android 32-bit remote debug server(ST) v1.17. Hex-Rays (c) 2004-2013
Listening on port #23946...
```

手机端打开程序（若打不开请装谷歌配件）

然后进 23946 端口 ~~其中有个解析太慢直接 canel



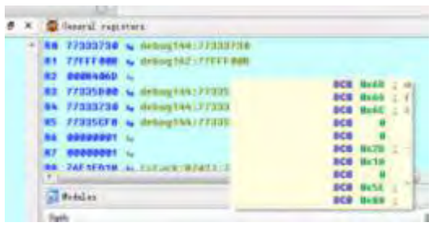
Shift+f2

找 Code 段~~~（也只有 code 段这么大）

然后在所有你怀疑的地方加偏移 7547b000（有些人不是这偏移）下断点放 f2，f9 后按按手机屏幕，然后 f9 配合 f7，f2 调试

在 13a258 这个函数发现目标

```
; MFileReadTask::Decrypt(void *, unsigned int, std::string const&)
EXPORT _ZN13MFileReadTask7DecryptEPvjRKSs
_ZN13MFileReadTask7DecryptEPvjRKSs
```



mfl 字段

怎么找到 13a258~~这只能说看着函数样子和名字找的~~其实我一开始是查找，psb.m 字段~~然后发现那函数不对~~花了很大代价~~~这个得请教大神

在这个函数的末尾 13a298 也设下断点，按一下 f9 mlf 对应的 hex view 立即变成(psb，确定是解密函数。

```
780FD008  6D 66 6C 00 00 2D 1A 00  5E 88 E6 49 B4 D8 64 39  mfl..-..^..I..d9
780FD018  47 64 A9 D5 9C 51 71 AC  48 38 E5 31 55 D4 16 42  Gd...Qq.H8.1U..B
780FD028  8C E9 13 2E 3B 06 36 7A  25 5C 5D 0B 52 09 BC EE  ....;.6z%\].R...
780FD038  64 C2 B2 E3 EF 1A 27 8A  CA 41 7C 6E 67 87 DA 05  d.....'..A|ng...
780FD048  02 A6 19 54 1E A3 25 EE  CB F5 5C CD E0 A6 C0 A7  ...T..%...\.....
780FD058  9C 68 97 25 79 B6 5C 90  5F 8D 86 F8 19 BF 77 4B  .h.%y.\._.....wK
780FD068  6A 7D 3B 76 37 88 DB 26  E4 76 C3 B5 17 B4 C7 64  j};v7..&.v.....d
780FD078  27 47 53 E9 D4 A5 E0 7D  4D 64 BF DA 12 55 2F 36  'GS....}Md...U/6
780FD088  1E 8C 75 33 AD 30 87 1B  F9 25 51 6C 8F 52 82 9D  ..u3.0...%Q1.R..
```

```
780FD008  6D 66 6C 00 00 2D 1A 00  28 50 53 42 00 02 00 00  mfl..-..(PSB....
780FD018  00 28 A0 03 04 B1 2C 01  00 B8 20 03 00 DD 20 03  .(....,... ... .
780FD028  00 E1 20 03 0B 00 2D 01  00 8E 31 00 00 0E 21 0A  .. ...-...1...!.
780FD038  0E 20 0D 01 00 01 20 2D  1F 03 00 04 00 05 00 06  . .... -........
780FD048  00 07 00 08 00 09 00 0A  00 0B 00 0C 00 0D 00 0E  ................
780FD058  00 0F 00 10 00 11 00 12  00 18 13 00 14 00 15 00  ................
780FD068  16 00 17 00 18 00 19 00  1A 76 C3 B5 17 B4 C7 64  .........v.....d
780FD078  27 47 53 E9 D4 A5 E0 7D  4D 64 BF DA 12 55 2F 36  'GS....}Md...U/6
780FD088  1E 8C 75 33 AD 30 87 1B  F9 25 51 6C 8F 52 82 9D  ..u3.0...%Q1.R..
780FD098  62 6A 65 BF 6A EF 92 07  2D D5 C9 7C 75 EB 82 57  bje.j...-..|u..W
780FD0A8  03 8C A1 96 5C 8E AA B4  E4 59 FE CF C1 74 AB 55  ....\....Y...t.U
780FD0B8  A9 0A 67 00 35 E1 A7 C5  82 7E 95 0F F8 12 24 62  ..g.5....~....$b
780FD0C8  D7 7C E0 2C E8 2F 2A C2  85 FE D2 D8 10 0B 12 DA  .|.,./*.........
```

这个函数也太大了~~~~无奈二分查看~~~神马时候 hex view 改变

```
MOV             R2, R3,LSR#4
VLD1.8          {D16-D17}, [R4]
SUB             R10, R2, #1
MOV             R8, #1
AND             R5, R10, #7
```

发现是这些 vld 造成的

锁定 vld

```
755B5680 MOV             R0, R9
755B5684 MOV             R2, R3,LSR#4
755B5688 VLD1.8          {D16-D17}, [R4]
```

不停的调试和看代码

还像就是从第 8 位源文件和一个解密段互相异或，中间多数是数组下标判断越界的改变~~~

Dump 出解密段 shift+f2

```
780FD010  5E 88 E6 49 B4 D8 64 39  47 64 A9 D5 9C 51 71 AC  ^..I..d9Gd...Qq.
780FD020  48 38 E5 31 55 D4 16 42  8C E9 13 2E 3B 06 36 7A  H8.1U..B....;.6z
780FD030  25 5C 5D 0B 52 09 BC EE  64 C2 B2 E3 EF 1A 27 8A  %\].R...d.....'.
780FD040  CA 41 7C 6E 67 87 DA 05  02 A6 19 54 1E A3 25 EE  .A|ng......T..%.
780FD050  CB F5 5C CD E0 A6 C0 A7  9C 68 97 25 79 B6 5C 90  ..\......h.%y.\.
780FD060  5F 8D 86 F8 19 BF 77 4B  6A 7D 3B 76 37 88 DB 26  _.....wKj};v7..&
780FD070  E4 76 C3 B5 17 B4 C7 64  27 47 53 E9 D4 A5 E0 7D  .v.....d'GS....}
780FD080  4D 64 BF DA 12 55 2F 36  1E 8C 75 33 AD 30 87 1B  Md...U/6..u3.0..
780FD090  F9 25 51 6C 8F 52 82 9D  62 6A 65 BF 6A EF 92 07  .%Q1.R..bje.j...
780FD0A0  2D D5 C9 7C 75 EB 82 57  03 8C A1 96 5C 8E AA B4  -..|u..W....\...
```

```
Please enter script body
auto i;
fp=fopen("c:\\script.y","wb");
for (i=0x780fd010;i<0x780fd010+600;i++)
   fputc(Byte(i),fp);
```

搜索 d9Gd 发现源文件是 font_ahoge_main.psb.m（也可以 dump）

尝试写了异或程序发现成功（这句话说起来简单，尝试+调试了好几次呢）

```
ioff=8;
soff=0;
int t=0;
while (ioff<len){
    uchar tmp=a[ioff]^script[soff];
    fwrite(&tmp,1,1,fw);
    ioff++;soff++;
    if (soff==97) soff=0;
}
```

现在关键是解码段是怎么来的

发现虽然前面的代码很长，但是很快跳到了（这个我没花很长时间~现在想起来真是侥幸）

```
755B54DC ADD             R0, R6, #8
755B54E0 STR             R0, [SP,#0xF0+var_E0]
755B54E4 MOV             R0, R9
755B54E8 BL              md5_init
755B54EC LDR             R1, [SP,#0xF0+var_88]
755B54F0 MOV             R0, R9
755B54F4 LDR             R2, [R1,#-0xC]
755B54F8 BL              md5_append
755B54FC MOV             R0, R9
755B5500 ADD             R1, SP, #0xF0+var_98
755B5504 BL              md5_finish
755B5508 MOV             R1, R5
755B550C LDRB            R5, [SP,#0xF0+var_8E]
```

这一行，在这一行前面下断点

发现 r1 是对应这一段

```
52 6A 39 50 65 67 6F 68  34 66 6F 6E 74 5F 61 68  Rj9Pegoh4font_ah
6F 67 65 5F 6D 61 69 6E  2E 70 73 62 2E 6D 00 2E  oge_main.psb.m..
```

跳到 755b54ec 这段上时，这是 r0 指向的栈

```
76E1FC7C  00000000
76E1FC80  00000000
76E1FC84  67452301  dalvi
76E1FC88  EFCDAB89
76E1FC8C  98BADCFE
76E1FC90  10325476
```

我以前手动编过 md5 的加密~~很熟悉这就是标准的 md5 加密~~不熟悉的也可以自己看

```
76E1FC54  772EEF84  debug143:
76E1FC58  810269AF
76E1FC5C  A9E551BE
76E1FC60  EB1031A4
76E1FC64  7CCCE5E0
```

到了 755b5508 就成了 Rj9Pegoh4font_ahoge_main.psb.m（不断尝试可以知道都是 Rj9Pegoh4+小写文件名）的 md5 值~~yeah

然后就到了

```
755B55A0 STR             R6, [SP,#0xF0+var_9C]
755B55A4 BL              _ZN7MRandomC1EPKjj
755B55A8 LDR             R0, [R10,#0x8C]
```

这个函数~~~f9 后发现很重要（大多数函数我都是跳过后才发现很重要再回头看）

```
755BEF18 MOV             R8, R1
755BEF1C MOV             R1, #0x12BD6AA
755BEF24 MOV             R9, R2
755BEF28 BL              _ZN7MRandom8GenerateEj
755BEF2C LDR             R2, [R4]
```

这中间函数也很重要

F5 发现是

```
int __fastcall MRandom::Generate(int a1, int a2)
{
  int v2; // r4@1
  int v3; // r5@1
  int v4; // r0@1
  signed int v5; // r3@1
  unsigned int v6; // lr@2
  int result; // r0@2

  v2 = a1;
  v3 = a2;
  v4 = operator new[](2496);
  v5 = 1;
  *(_DWORD *)v2 = v4;
  *(_DWORD *)v4 = v3;
  *(_WORD *)(v2 + 4) = 1;
  do
  {
    v6 = *(_DWORD *)(*(_DWORD *)v2 + 4 * v5 - 4);
    *(_DWORD *)(*(_DWORD *)v2 + 4 * v5) = v5 + 1812433253 * (v6 ^ (v6 >> 30));
    result = *(_WORD *)(v2 + 4) + 1;
    v5 = (unsigned __int16)result;
    *(_WORD *)(v2 + 4) = result;
  }
  while ( (unsigned __int16)result < 0x270u );
  *(_WORD *)(v2 + 4) = 0;
  return result;
}
```

调试时发现传入的参量为 0x12bd6aa 和存储位置

```
AA D6 2B 01 13 AB D2 82  B7 96 20 34 36 65 84 54  ..+....... 46e.T
B7 5D E7 2A 38 E8 6F 82  E8 A8 03 2B 8F CB 2D 6C  .].*8.o....+..-l
0E 4D E2 A8 C5 D1 D7 E7  28 B9 D2 3B D3 74 51 4F  .M......(..;.tQO
E6 78 65 AC 01 B6 F6 CF  D8 E0 CA E5 76 E9 5C C5  .xe.........v.\.
39 B8 CA 54 29 A6 18 9A  09 92 D3 AB 6A 81 ED A5  9..T).......j...
1C B6 CC FA 50 71 13 8F  70 97 8E 8B 11 C2 6B 9A  ....Pq..p.....k.
97 BC E4 E9 7D 9A 35 48  06 4F 36 B6 AF 50 D2 CC  ....}.5H.O6..P..
F8 DF D9 20 F5 14 99 4E  62 D8 47 71 2E 5A DA D2  ... ...Nb.Gq.Z..
E1 A8 A2 7D 81 80 AC BB  D1 CE 69 23 98 71 23 6C  ...}.■....i#.q#l
81 B2 F5 48 A5 EC F5 26  3F AA 2D 0F 02 E2 D8 A8  ...H...&?.-.....
```

Dump 下写了个相同的模拟出来

```
void gen(){
    ge[0]=0x12bd6aa;
    int v5=1;
    do{
       uint v6=ge[v5-1];
       ge[v5]=v5+1812433253 * (v6 ^ (v6 >> 30));
       v5++;
    }while(v5<0x270);
    for (int i=0;i<20;i++) printf("ge %d  %x\n",i,ge[i]);
}
```

后面也都是体力活 f5 后再调试~~模拟出_ZN7MRandomC1EPKjj

```
void changegen(){
    uint v19=1,v20=0;
    for (int ci=0;ci<0x270;ci++){
        uint v33=ge[v19-1],v34=ge[v19];
        ge[v19]= v20 + ((1664525* (v33 ^ (v33 >> 30))) ^ v34)+A[v2
        v19++;v20++;
        if (v19>=0x270){ge[0]=ge[0x270-1];v19=1;}
        if (v20>=4) v20=0;
    }
     for (int ci=0;ci<0x270-1;ci++){
         uint v33=ge[v19-1],v34=ge[v19];
         ge[v19]=((1566083941*(v33^ (v33>> 30))) ^ v34)-v19;
         v19++;
         if (v19>=0x270){ge[0]=ge[0x270-1];v19=1;}
     }
     ge[0]=0x80000000;
     for (int i=0;i<20;i++) printf("ge %d  %x\n",i,ge[i]);
}
```

再模拟同样模拟出这个

```
755B55F4 MOV             R0, R8
755B55F8 BL              _ZN7MRandom6RandomEv
755B55FC B               loc_755B55D0
```

```
void changegen2(){
    int ch=0;
    for (int v1=0;v1<30;v1++){
        uint *v3=(ge+v1);
        uint v5=*(v3+1);
        uint v6=*(v3+(1588/4));
        uint v7=ge[v1]& 0x80000000;
        uint tmp2=(v5 & 0x7FFFFFFF );
        uint tmp=(tmp2| v7);
       // if (v1==2) printf("%x  %x   %x   %x %x\n",v5,v6,v7,tmp2,tmp);
        ge[v1]=v6 ^ (tmp>>1) ^ tv4[tmp & 1];
        uint v11=ge[v1];
        uint v12 = v11 ^ (v11 >> 11);
        uint vr=(((v12 << 7) & 0x9D2C5680 ^ v12) << 15) & 0xEFC60000 ^ (v12 << 7) & 0x9D2C5680 ^ v12 ^ (((((v12 << 7) & 0x9D2C568(
        printf("vr  %d %x\n",v1,vr);
        {
            script[ch++]=vr&0xff;
            vr/=0x100;
            script[ch++]=vr&0xff;
            vr/=0x100;
            script[ch++]=vr&0xff;
            vr/=0x100;
            script[ch++]=vr&0xff;
        }
    }
    for (int i=0;i<20;i++) printf("ge %d  %x\n",i,ge[i]);
}
```

最后解密段终于好了~~~~yeah

最后解密就好了~~~在加密回去时 mfl 头文件还有 4 个字节是解压缩后的大小~~~关于怎么解压缩我就不说了~~反正也是这种方法~~~~还请各位破解的愉快~~~

不错，其实很多事情，只要肯投入去做，总会找到解决方法的，即使之前从来没遇到过。
其实呢~整篇文章，没看懂。
楼主写文章，完全沉浸在自己解决的过程中了。对于大多数人而言，并不关心你怎么破解的，关心的是你通过什么手段，收集到了什么信息，然后如果根据这些信息做判断，从而进行下一步操作。
所以如果文章增加一些对环境的介绍，以及自己的思考过程的话，会更好。

说点实际的，如果在 jar 里面没有发现关键内容的话，就要注意 jar 里面的 native 函数以及 Loadlibrary 操作，从而可以判断出加载了哪些 so，调用了什么函数，就不会出现判断不出是不是加载了 lib-main.so 的情况了。
IDA 调试的时候，只要勾选 加载 so 时 pause ，通过 so 路径，就能很容易判断出来是不是 apk 自带的 so 文件了，attach 是一种方法，但是很多情况下关键点是在加载过程中的，很容易漏掉。

之后的内容就太模糊了，建议花点篇幅介绍一下 vld 指令，反正我是第一次见，看完还是没明白这是什么意思。

基本就这样，已经做得很不错了，楼主继续加油吧。

嗯（总算等到大神）~~~第一次写文章~~~事后发现真的没写书天赋~~~~还没怎么学破解~~~那个 ida 调试还是自己刚学会的~~~至于 vld 指令~~可以查 arm 的文档的~~应该在这里 http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204ic/Chdhcfbc.html
非常感谢大神指导~~~~~

http://bbs.pediy.com/showthread.php?t=193353

Android native 反调试方式及使用 IDA 绕过反调试

0x00

为了避免我们的 so 文件被动态分析，我们通常在 so 中加入一些反调试代码，常见的 Android native 反调试方法有以下几种。

1、直接调用 ptrace(PTRACE_TRACEME, 0, 0, 0)，参考 Android Native 反调试。

2、根据上面说的/proc/$pid/status 中 TracerPid 行显示调试程序的 pid 的原理，可以写一个方法检查下这个值，如果!=0 就退出程序。参考 Android Native 反调试，用 JNI 实现 APK 的反调试。

3、检查代码执行的间隔时间，参考 Android 应用方法隐藏及反调试技术浅析的 0×03 反调试初探。

4、检测手机上的一些硬件信息，判断是否在调试器中，参考 Android 应用方法隐藏及反调试技术浅析的 0×03 反调试初探。

0x01

那么我们如何过掉这些反调试呢？

我们以阿里比赛第二题为例，参考安卓动态调试七种武器之孔雀翎 - Ida Pro。

我们讲解两种方式：

1、Ida Patch so

2、Ida 动态修改内存数据和寄存器数值

我们首先讲解 Ida Patch so，有几处都可以 patch。我们从易到难依次讲解。

第一处：

我们在 JNI_ONLOAD 下断点，如下图：

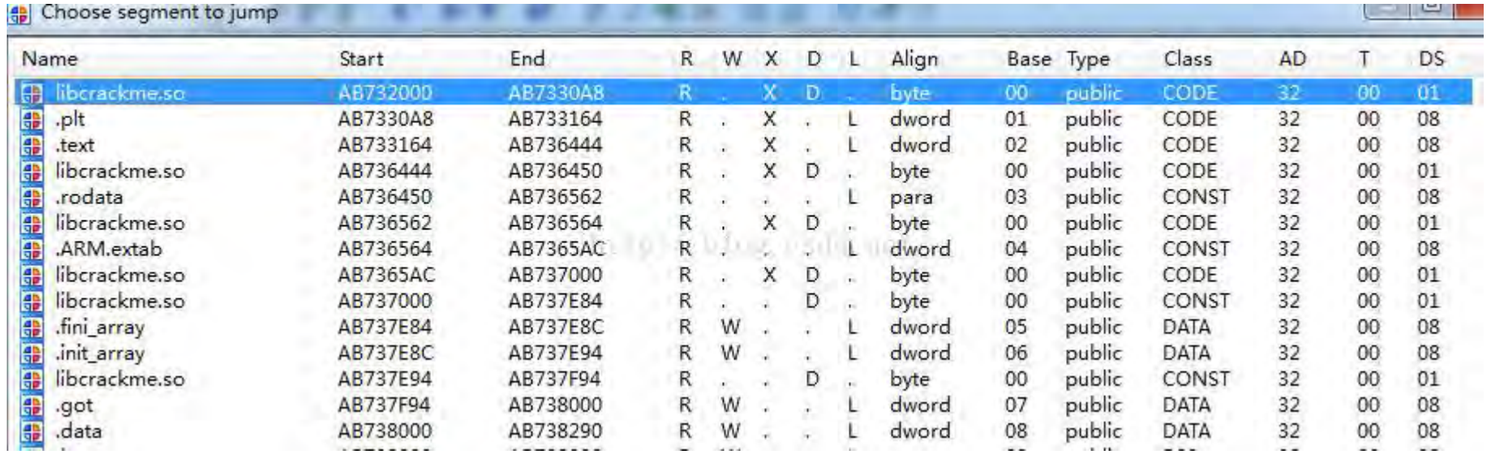
```
.text:AB733B9C handle= -0x20
.text:AB733B9C
.text:AB733B9C STMFD   SP!, {R4-R9,R11,LR}
.text:AB733BA0 ADD     R11, SP, #0x18
.text:AB733BA4 SUB     SP, SP, #8
.text:AB733BA8 MOV     R4, R0
.text:AB733BAC LDR     R0, =(_GLOBAL_OFFSET_TABLE_ - 0xAB733BC0)
.text:AB733BB0 LDR     R9, =(unk_AB738290 - 0xAB737FBC)
.text:AB733BB4 MOV     R8, #0
.text:AB733BB8 ADD     R0, PC, R0 ; _GLOBAL_OFFSET_TABLE_
.text:AB733BBC ADD     R0, R9, R0 ; unk_AB738290
.text:AB733BC0 STR     R8, [R0,#(dword_AB7382C8 - 0xAB738290)]
.text:AB733BC4 LDR     R5, [R0,#(dword_AB7382C4 - 0xAB738290)]
```

依次单步执行到 BLX R7

```
.text:AB733C50 LDR     R7, [R0,#0x24]
.text:AB733C54 SUB     R0, R11, #-handle               ; handle
PC
.text:AB733C58 BLX     R7 ; __imp_dlsym
.text:AB733C5C BL      sub_AB7337F4
.text:AB733C60 LDR     R0, [R4]
.text:AB733C64 MOV     R6, #4
.text:AB733C68 MOV     R1, R5
.text:AB733C6C ORR     R6, R6, #0x10000
.text:AB733C70 MOV     R2, R6
.text:AB733C74 LDR     R3, [R0,#0x18]
.text:AB733C78 MOV     R0, R4
.text:AB733C7C BLX     R3 ; __imp___aeabi_memset
.text:AB733C80 CMP     R0, #0
.text:AB733C84 MOVNE   R6, #0xFFFFFFFF
.text:AB733C88 MOV     R0, R6
.text:AB733C8C SUB     SP, R11, #0x18
.text:AB733C90 LDMFD   SP!, {R4-R9,R11,PC}
.text:AB733C90 ; End of function JNI_OnLoad
```


我们发现当执行完这步后，我们的 ida 就退出了，说明反调试代码是从这个入口进入执行的。那么我们只要把这个入口给 NOP 掉，就可以绕过反调试了。

Patch so 就是修改 so 中的二进制代码，然后再重新签名生成新的 apk。Patch so，需要修改的本地 so 中的代码，而不是内存中的，所以我们需要通过上图内存中指令地址减去 so 在内存中的基地址来获取这条指令在本地 so 文件中的偏 移。那么 so 在内存中的基地址怎么获取呢？按 Crtl+s。

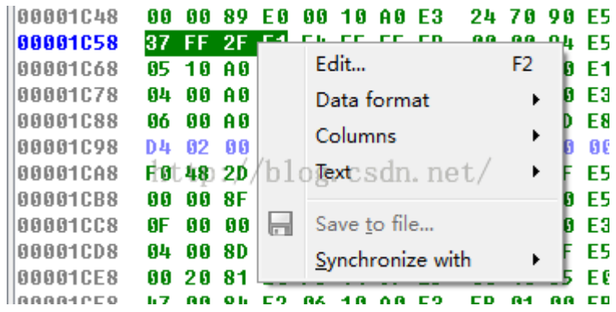


我们看到 libcrackme.so 的基地址是 AB732000，用 BLX R7 的地址 AB733C58 减去 AB732000，等于 1C58。

然后我们双开 ida，在另一个 ida 中打开 libcrackme.so，按 G，然后输入 1C58，果然我们调到了 BLX R7 的位置，如下图：

```
.text:00001C48                ADD     R0, R9, R0
.text:00001C4C                MOV     R1, #0          ; name
.text:00001C50                LDR     R7, [R0,#0x24]
.text:00001C54                SUB     R0, R11, #-handle ; handle
.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
.text:00001C70                MOV     R2, R6
```

下面就要把这行代码 NOP，可以修改为 00 00 00 00，也可以修改为 00 00 A0 E1。



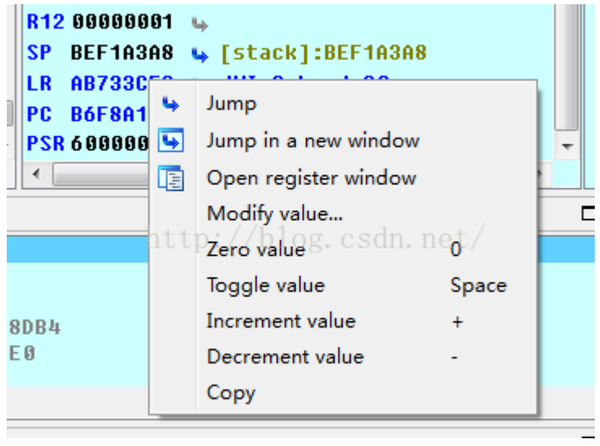
修改后点击右键，applay change。然后重新签名生成 apk，再次运行 apk，ida 调试时就没有反调试的干扰了。

第二处：

我们按 F7 进入 BLX R7 的内部执行，如下图：

```
R7
PC libc.so:B6F8A108 pthread_create
   libc.so:B6F8A108 STMFD   SP!, {R4-R11,LR}
   libc.so:B6F8A10C SUB     SP, SP, #0xC
   libc.so:B6F8A110 MOV     R9, R1
   libc.so:B6F8A114 MOV     R8, R2
   libc.so:B6F8A118 MOV     R7, R3
   libc.so:B6F8A11C MOV     R10, R0
   libc.so:B6F8A120 BLX     __errno
   libc.so:B6F8A124 LDR     R4, =(unk_B6FC5FD8 - 0xB6F8A138)
   libc.so:B6F8A128 LDR     R2, =0xFFFFFF50
   libc.so:B6F8A12C MOV     R5, #1
   libc.so:B6F8A130 ADD     R4, PC, R4 ; unk_B6FC5FD8
   libc.so:B6F8A134 MOV     R1, R5
```


是创建了一个线程去执行反调试，这里有个小技巧，如果我们想回到刚才的函数 BLX R7，怎么办呢？选择寄存器 LR，然后点击右键选择 Jump 即可，同理选择 PC 是跳到当前的位置。



这个线程执行的函数体是 sub_AB7336A4，如下图：

```
.text:AB733C3C STR     R8, [R5]
.text:AB733C40 ADD     R0, PC, R0 ; _GLOBAL_OFFSET_TABLE_
.text:AB733C44 ADD     R2, R1, R0 ; sub_AB7336A4
.text:AB733C48 ADD     R0, R9, R0
.text:AB733C4C MOV     R1, #0                  ; name
.text:AB733C50 LDR     R7, [R0,#0x24]
.text:AB733C54 SUB     R0, R11, #-handle       ; handle
.text:AB733C58 BLX     R7 ; __imp_dlsym
```

sub_AB7336A4 函数体如下：

```
.text:AB7336A4 sub_AB7336A4                            ; DATA XREF: JNI_OnLoad+A8↓o
.text:AB7336A4                                         ; .text:off_AB733CA4↓o
.text:AB7336A4 STMFD   SP!, {R4,LR}
.text:AB7336A8 LDR     R0, =(_GLOBAL_OFFSET_TABLE_ - 0xAB7336B8)
.text:AB7336AC LDR     R1, =(unk_AB738290 - 0xAB737FBC)
.text:AB7336B0 ADD     R0, PC, R0 ; _GLOBAL_OFFSET_TABLE_
.text:AB7336B4 ADD     R4, R1, R0 ; unk_AB738290
.text:AB7336B8                     http://blog.csdn.net/
.text:AB7336B8 loc_AB7336B8                            ; CODE XREF: sub_AB7336A4+24↓j
.text:AB7336B8 BL      sub_AB73330C
.text:AB7336BC LDR     R1, [R4,#(off_AB7382B0 - 0xAB738290)]
.text:AB7336C0 MOV     R0, #3
.text:AB7336C4 BLX     R1
.text:AB7336C8 B       loc_AB7336B8
```

这个方法循环执行 sub_AB73330C。我们进入 sub_AB73330C，怀疑这里就是真正检查是否处于调试状态的地方，但是代码经过了严重的混淆，所以找不到反调试的代码。

那怎么办呢？在 0x00 中我们谈到了常见的反调试代码，最常见的是第二种方式，第二种方式检查的过程中会调用 fopen，所以我们在 libc 的 fopen 方法下断点，来是哪个函数调用的 fopen，基本上就可以断定这个函数是反调试代码。

首先我们需要找到 fopen 的位置，按 Alt+T，然后输入 fopen 关键字，如下图：

找到 fopen 后，代码是这样的：

```
libc.so:B6F917EF DCB    0
libc.so:B6F917F0 fopen DCB 0xF7 ;
libc.so:B6F917F1 DCB 0xB5 ;
libc.so:B6F917F2 DCB    7
libc.so:B6F917F3 DCB 0x46 ; F
libc.so:B6F917F4 DCB    8
libc.so:B6F917F5 DCB 0x46 ; F
libc.so:B6F917F6 DCB    1
libc.so:B6F917F7 DCB 0xA9 ;
libc.so:B6F917F8 DCB 0xFF
libc.so:B6F917F9 DCB 0xF7 ;
libc.so:B6F917FA DCB 0xBF ;
```


此时按 P，就可以变成代码形式。如下图，在 fopen 处下断点。

```
libc.so:B6F917F0
libc.so:B6F917F0 fopen
libc.so:B6F917F0 PUSH    {R0-R2,R4-R7,LR}
libc.so:B6F917F2 MOV     R7, R0
libc.so:B6F917F4 MOV     R0, R1
libc.so:B6F917F6 ADD     R1, SP, #4
libc.so:B6F917F8 BL      __sflags
libc.so:B6F917FC LDR     R5, =(unk_B6FC5FD8 - 0xB6F91802)
libc.so:B6F917FE ADD     R5, PC ; unk_B6FC5FD8
libc.so:B6F91800 MOV     R6, R0
libc.so:B6F91802 CBNZ    R0, loc_B6F91808
```


点击 F9，继续运行，我们看到程序停在 fopen 处，此时 LR 就是刚刚我们谈到的 sub_AB73330C，如下图：

```
SP  AB731A68 ↳ [stack:1307]:AB731A68
LR  AB733420 ↳ sub_AB73330C+114
PC  B6F917F0 ↳ libc.so:fopen
PSR 60000030
```


所以我们可以确定 sub_AB73330C 就是进行反调试的代码。我们可以看到这个函数是被 sub_AB7336A4 调用的。

```
.text:AB73330C ; =============== S U B R O U T I N E =======================================
.text:AB73330C
.text:AB73330C
.text:AB73330C sub_AB73330C                            ; CODE XREF: sub_AB7336A4:loc_AB7336B8↓p
.text:AB73330C
.text:AB73330C var_348= -0x348
.text:AB73330C var_344= -0x344
.text:AB73330C var_338= -0x338
.text:AB73330C var_334= -0x334
```


点击右侧的 CODE XREF:sub_AB7336A4 就能进入到调用 sub_AB73330C 的地方。在 sub_AB7336A4 函数上按 F5，就能看到对应的 C 语言代码。如下：

```
1 void sub_AB7336A4()
2 {
3   unsigned int v0; // r1@1
4   int *i; // r0@1
5
6   v0 = 0x2D4u;
7   for ( i = &GLOBAL_OFFSET_TABLE_; ; i = (int *)off_AB7382B0(3) )
8     sub_AB73330C(i, v0);
9 }
```


可见程序是在 sub_AB73330C 循环检测是否被反调试的。

此时我们可以用和第一处一样的方式，找到本地 so 中对应的方法，然后 Patch so，Nop 掉对应的方法，然后重新签名，重新运行。

第三处:

其实第三处和第二处原理是一样的，只不过这里不使用 Nop 了，sub_AB73330C 开始和结束的汇编代码如下：

开始时：

```
.text:AB73330C
.text:AB73330C STMFD   SP!, {R4-R11,LR}
.text:AB733310 SUB     SP, SP, #0x324
.text:AB733314 LDR     R0, =(_GLOBAL_OFFSET_TABLE_ - 0xAB733328)
.text:AB733318 MOV     R1, #0x64
```

结束时：

```
.text:AB733638 ADDEQ   SP, SP, #0x324
.text:AB73363C LDMEQFD SP!, {R4-R11,PC}
.text:AB733640 BL      __stack_chk_fail
```

所以我们可以把 AB733310 的代码修改为 AB73363C 处的代码，不执行任何操作，直接返回。

0x02

Ida 动态修改内存数据和寄存器数值

我们看到反调试方法第二点，代码如下：

```
void be_attached_check()
{
	try
	{
		const int bufsize = 1024;
		char filename[bufsize];
		char line[bufsize];
		int pid = getpid();
		sprintf(filename, "/proc/%d/status", pid);
		FILE* fd = fopen(filename, "r");
		if (fd != nullptr)
		{
			while (fgets(line, bufsize, fd))
			{
				if (strncmp(line, "TracerPid", 9) == 0)
				{
					int statue = atoi(&line[10]);
					LOGD("%s", line);
					if (statue != 0)
					{
						LOGD("be attached !! kill %d", pid);
						fclose(fd);
						int ret = kill(pid, SIGKILL);
					}
					break;
				}
			}
			fclose(fd);
		} else
		{
			LOGD("open %s fail...", filename);
		}
	} catch (...)
	{

	}

}
```

我们发现该程序会用 fopen ()打开/proc/[pid]/status 这个文件，随后会用 fgets()和 strcmp()来比较，于是我们在 strcmp() 处下个断点，然后让 hex view 的数据与 R0 同步。每次点击继续，我们都会看到 strstr 传入的参数。当传入的参数变为 TracerPid:XXXX 的时候我们停一下。因为在 正常情况下，TracerPid 的值应该是 0。但是当被调试的时候就会变成调试器的 pid。

我们在 strcmp 下断点：

```
libc.so:B6FB6388 strncmp
libc.so:B6FB6388 PUSH    {R4,R5,LR}
libc.so:B6FB638A CBZ     R2, loc_B6FB63A6
libc.so:B6FB638C MOVS    R3, #0
libc.so:B6FB638E
libc.so:B6FB638E loc_B6FB638E                    ; CODE XREF: strncmp+18↓j
libc.so:B6FB638E LDRB    R4, [R0,R3]
```

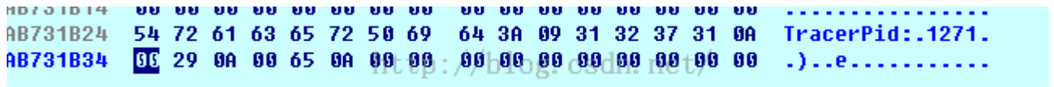

程序会在此处断下，当我们发现 R0 地址中的内容为 TracerPid:XXXX 时，我们停一下，如下图：

```
R0  AB731B25 ↳ [stack:1307]:AB731B25
R1  AB738326 ↳ .bss:AB738326
R2  00000008 ↳
R3  00000054 ↳
R4  AB731B24 ↳ [stack:1307]:AB731B24
R5  AB731B25 ↳ [stack:1307]:AB731B25
R6  AB738326 ↳ .bss:AB738326
R7  00000054 ↳
R8  00000008 ↳
R9  AB731B24 ↳ [stack:1307]:AB731B24
R10 AB738290 ↳ .bss:unk_AB738290
R11 AB731AA4 ↳ [stack:1307]:AB731AA4
R12 051B4001 ↳
SP  AB731A50 ↳ [stack:1307]:AB731A50
LR  B6FA7895 ↳ libc.so:strstr+2D
```

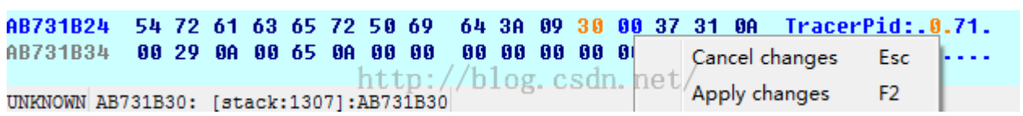

R0 的里面存的地址是 AB731B25，里面的内容为，如下图：

```
[stack:1307]:AB731B24 DCB 0x54 ; T
[stack:1307]:AB731B25 DCB 0x72 ; r
[stack:1307]:AB731B26 DCB 0x61 ; a
[stack:1307]:AB731B27 DCB 0x63 ; c
[stack:1307]:AB731B28 DCB 0x65 ; e
[stack:1307]:AB731B29 DCB 0x72 ; r
[stack:1307]:AB731B2A DCB 0x50 ; P
[stack:1307]:AB731B2B DCB 0x69 ; i
[stack:1307]:AB731B2C DCB 0x64 ; d
[stack:1307]:AB731B2D DCB 0x3A ; :
[stack:1307]:AB731B2E DCB    9
[stack:1307]:AB731B2F DCB 0x31 ; 1
[stack:1307]:AB731B30 DCB 0x32 ; 2
[stack:1307]:AB731B31 DCB 0x37 ; 7
[stack:1307]:AB731B32 DCB 0x31 ; 1
[stack:1307]:AB731B33 DCB  0xA
```

我们可以通过修改内存值的方式来过掉这一次反调试。



把 TracerPid 改为 0，如下图：



然后点击 Apply changes。这样就可以过掉这次反调试。我们在前面也看到了，程序是在一个循环中进行反调试检查，所以这样的方试只是过了其中一次反调试。

不推荐使用这种方式，最好使用 Patch So 的方式。

http://blog.csdn.net/jltxgcy/article/details/50598670

调试伪码

IDA Pro 6.xx 看图教程之用伪 c 代码调试某社交软件的 So 库

作者：大老
博客：blog.sina.com.cn/dalaoqd
qq:79234668

先说一下前期的准备工作

android native c 的 so 调试
基本上跟 linux 一样
1.把 ida 目录下 android_server 传到 android 目录中
如：
adb push android_server /data/local/tmp/
adb shell 进入模拟器
su --------->这个必须要打主要是提权，不然你后面看到的进程就寥寥无几啦。
cd /data/local/tmp/
chmod 755 android_server ========》给你上传上来的文件加运行权限
./android_server ------》运行你传上来的服务端给 ida 提供调试服务
看到监听端口 23946

2.在 windows 控制台下转发 window 到模拟器或者手机的端口

adb forward tcp:23946 tcp:23946 =======》这个是端口转发

3.ida 中选择 android 调试
在 Debuggger 中的 process options 的
hostname 填上 localhost
port: 23946

4.在 Debuggger 中的 attach 上 android 所对应的安卓程序就行了
下面来张 ida pro 调试的布局说明。

IDA - C:\Users\ADMINI~1\AppData\Local\Temp\ida09930.idb (app_process)

File Edit Jump Search View Debugger Options Windows Help

Remote ARMLinux/Android debugger

Library function Data Regular function Unexplored Instruction External symbol

Debug View | Structures | Enums

IDA View-PC

```
libsign.so:571794DC                                     ; DATA XREF: JstringToPchar+4E↑r
libsign.so:571794E0 ; ---------------------------------------------------------------------------
libsign.so:571794E0
libsign.so:571794E0 Java_com_immomo_momo_util_jni_Codec_xxoo
libsign.so:571794E0 PUSH            {R4-R7,LR}
libsign.so:571794E2 SUB             SP, SP, #0x13C
libsign.so:571794E4 LDR             R6, [SP,#0x154]
libsign.so:571794E6 STR             R3, [SP,#0x2C]
libsign.so:571794E8 LDR             R4, [SP,#0x150]
libsign.so:571794EA STR             R6, [SP,#0x20]
libsign.so:571794EC LDR             R6, =(dword_5717DF98 - 0x571794F6)
libsign.so:571794EE LDR             R1, =(aJavaLangString - 0x571794FC)
libsign.so:571794F0 STR             R2, [SP,#0x28]
libsign.so:571794F2 ADD             R6, PC ; dword_5717DF98
libsign.so:571794F4 LDR             R6, [R6]
libsign.so:571794F6 STR             R4, [SP,#0x24]
libsign.so:571794F8 ADD             R1, PC                  ; "java/lang/String"
libsign.so:571794FA LDR             R3, [R6]
libsign.so:571794FC MOVS            R5, R0
libsign.so:571794FE LDR             R7, =(dword_5717E004 - 0x57179508)
libsign.so:57179500 STR             R3, [SP,#0x134]
libsign.so:57179502 LDR             R3, [R0]
libsign.so:57179504 ADD             R7, PC ; dword_5717E004
libsign.so:57179506 LDR             R3, [R3,#0x18]
libsign.so:57179508 BLX             R3
libsign.so:5717950A MOVS            R1, R0
libsign.so:5717950C STR             R0, [R7,#(dword_5717E008 - 0x5717E004)]
libsign.so:5717950E LDR             R0, [R5]

UNKNOWN 571794E0: libsign.so:Java_com_immomo_momo_util_jni_Codec_xxoo
```

R12
PC

现在状态已调试中断
到所关心的函数

这里是寄存器，入口
参数的地址和值也在
这要密切关注

这里是SO库文件的模
块列表你要分析的模
块从这找

这里是内存数据窗口
看数据这里比较直观

General registers

```
R0  00C9CC60  [heap]:00C9CC60
R1  D3300001
R2  CF3E07EA
R3  0004CB1D
R4  5745DFB8  dalvik_LinearAlloc:5745DFB8
R5  00C9F628  [heap]:00C9F628
R6  00000004
R7  5CE08DE0  debug076:5CE08DE0
```

Modules | Module: libs

```
Path
/data/data/com.immomo.momo/lib/libmjni.so
/data/data/com.immomo.momo/lib/libsign.so
/system/bin/app_process
/system/bin/linker
/system/lib/egl/libGLES_android.so
/system/lib/libEGL.so
```

Line 2 of 110

Threads

Decimal	Hex	State
2430	97E	Ready
2428	97C	Ready
2426	97A	Ready
2424	978	Ready
2422	976	Ready
2494	9BE	Ready
2495	9BF	Ready

Hex View-1

```
FFFF0FF0  00 00 00 00 00 00 00 00  00 00 00 00 03 00 00 00  ................

UNKNOWN FFFF0FF0: [vectors]:FFFF0FF0
```

Stack view

```
602CAC08  15300005
602CAC0C  D3D00009
602CAC10  5CE08DD0  debug076:5CE08DD0
UNKNOWN 602CAC08: debug080:602CAC08
```

Output window

```
  Please check the Edit/Plugins menu for more informaton.
--------------------------------------------------------------------------------
Python 2.7.2 (default, Jun 12 2011, 15:08:59) [MSC v.1500 32 bit (Intel)]
IDAPython v1.7.0 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>
```

备注：这张是静态分析到关键点设断点后的抓图。
下面重点来了在这按 F5，前提你要有 arm 插件哦

IDA - C:\Users\ADMINI~1\AppData\Local\Temp\ida09930.idb (app_process)

File Edit Jump Search View Debugger Options Windows Help

Remote ARMLinux/Android debugger

Library function Data Regular function Unexplored Instruction External symbol

Debug View | Structures | Enums

IDA View-PC | Pseudocode-A

```
37  char v38[24]; // [sp+78h] [bp-D8h]@7
38  char v39; // [sp+88h] [bp-98h]@6
39  int v40; // [sp+134h] [bp-1Ch]@1
40
41  v4 = a4;
42  v5 = a3;
43  v6 = dword_5717DF98;
44  v7 = a1;
45  v40 = *(_DWORD *)dword_5717DF98;
46  dword_5717E008 = (*(int (**)(void))(*(_DWORD *)a1 + 24))();
47  dword_5717E004 = (*(int (__fastcall **)(int, int, _DWORD, _DWORD))(*(_DWORD *)v7 + 132))(
48                     v7,
49                     dword_5717E008,
50                     "getBytes",
51                     "(Ljava/lang/String;)[B");
52  v30 = 0;
53  v28 = JstringToPchar(v7);
54  v8 = (*(int (__fastcall **)(int))(*(_DWORD *)v7 + 124))(v7);
55  (*(void (__fastcall **)(int, int, _DWORD, _DWORD))(*(_DWORD *)v7 + 132))(
56    v7,
57    v8,
58    "getPackageName",
59    "()Ljava/lang/String;");
60  v9 = _JNIEnv::CallObjectMethod(v7);
61  v10 = v5;
62  (*(void (__fastcall **)(int, int, _DWORD))(*(_DWORD *)v7 + 676))(v7, v9, 0);
63  v11 = (char *)&v31;
64  v27 = v4;
65  v29 = v6;
```

UNKNOWN Java_com_immomo_momo_util_jni_Codec_xxoo:41

Dalao

qq:79234668

http://blog.sina.co.cn/dalaoqd

这个是按F5后得到的
结果可以直接调试想
看那个变量把鼠标放
到变量上即可得到结
果或地址

General registers | IDA View

```
R0  00C9CC60  [heap]:00C9CC60
R1  D3300001
R2  CF3E07EA
R3  0004CB1D
R4  5745DFB8  dalvik_LinearAlloc:5745DFB8
R5  00C9F628  [heap]:00C9F628
R6  00000004
R7  5CE08DE0  debug076:5CE08DE0
```

Modules | Module: libs

Path

```
/data/data/com.immomo.momo/lib/libmjni.so
/data/data/com.immomo.momo/lib/libsign.so
/system/bin/app_process
/system/bin/linker
/system/lib/egl/libGLES_android.so
```

Line 2 of 110

Threads

Decimal	Hex	State
2434	982	Ready
2433	981	Ready
2432	980	Ready
2431	97F	Ready
2430	97E	Ready
2428	97C	Ready
2426	97A	Ready

Hex View-1

```
00C9CC60  B0 CC AE 40 00 00 00 00  11 00 00 00 28 F6 C9 00  ...@........(...
00C9CC70  00 00 00 00 A0 54 BE 00  A8 C0 C9 00 D3 11 00 00  .....T..........
00C9CC80  D0 CC C9 00 2E 6C 61 6E  67 2E 44 61 65 6D 6F 6E  .....lang.Daemon
```

UNKNOWN 00C9CC60: [heap]:00C9CC60

Stack view

```
602CAAB8  00B19FB0  [heap]:00B19FB0
602CAABC  00B1A000  [heap]:00B1A000
602CAAC0  00B1EEC0  [heap]:00B1EEC0
```

UNKNOWN 602CAAB8: debug080:602CAAB8

Output window

```
57179CFC: using guessed type _DWORD Sha256Digest::getHash(Sha256Digest *__hidden this, unsigned __int8 *);
57179D18: using guessed type int __fastcall sub_57179D18(_DWORD, _DWORD, _DWORD, _DWORD);
5717BD97: using guessed type char byte_5717BD97;
5717BDA4: using guessed type int dword_5717BDA4[3];
```

Dalao

qq:79234668

http://blog.sina.co.cn/dalaoqd

现在的结果是不是就清晰明朗了很多。根据这个结果很快定位到关键的 call 来到了下面这张图

IDA - C:\Users\ADMINI~1\AppData\Local\Temp\ida09930.idb (app_process)

File Edit Jump Search View Debugger Options Windows Help

Remote ARMLinux/Android debugger

Library function Data Regular function Unexplored Instruction External symbol

Debug View | Structures | Enums

IDA View-PC | Pseudocode-A

```
76  memset_0(v14);
77  if ( v30 )
78  {
79    v15 = dword_5717DCFC[10 * v32 + v31];
80    v19 = dword_5717DCFC[v35 + 10 * v34];
81    v20 = dword_5717DCFC[10 * v33 + v36];
82    v17 = (char *)dword_5717BDA4;
83  }
84  else
85  {
86    v15 = dword_5717DCFC[10 * v32 + v31];
87    v16 = dword_5717DCFC[10 * v34 + v35];
88    v17 = &byte_5717BD97;
89    v18 = dword_5717DCFC[10 * v33 + v36];
90  }
91  ((void (__fastcall *)(unsigned __int8 *, char *, int))sub_5717939C)(v14, v17, v15);
92  Sha256Digest::reset((Sha256Digest *)&v39);
93  v21 = strlen_0(v14);
94  Sha256Digest::update((Sha256Digest *)&v39, v14, v21);
95  Sha256Digest::finalise((Sha256Digest *)&v39);
96  Sha256Digest::getHash((Sha256Digest *)&v39, (unsigned __int8 *)v37);
97  v22 = 0;
98  do
99  {
100   v23 = &v38[2 * v22];
101   v24 = (unsigned __int8)v37[v22++];
102   sub_5717939C(v23, &byte_5717BDB3, v24);
103 }
104 while ( v22 != 32 );
```

UNKNOWN Java_com_immomo_momo_util_jni_Codec_xxoo:91

Dalao

QQ:79234668

http://blog.sina.com.cn/dalao

找到关键点下断点，
中断后按F8单步调试
这里可以很清晰的看
到5717939C子程序
的入口参数V14 V17
V15

General registers | IDA View

```
R0  00C9CC60  [heap]:00C9CC60
R1  D3300001
R2  CF3E07EA
R3  0004CB1D
R4  5745DFB8  dalvik_LinearAlloc:5745DFB8
R5  00C9F628  [heap]:00C9F628
R6  00000004
R7  5CE08DE0  debug076:5CE08DE0
R8  602C0C10  debug080:602C0C10
```

Modules | Module: libs

Path

/data/data/com.immomo.momo/lib/libmjni.so
/data/data/com.immomo.momo/lib/libsign.so
/system/bin/app_process
/system/bin/linker
/system/lib/egl/libGLES_android.so

Line 2 of 110

Threads

Decimal	Hex	State
2434	982	Ready
2433	981	Ready
2432	980	Ready
2431	97F	Ready
2430	97E	Ready
2428	97C	Ready
2426	97A	Ready

Hex View-1

```
00C9CC60  B0 CC AE 40 00 00 00 00  11 00 00 00 28 F6 C9 00  ...@........(...
00C9CC70  00 00 00 00 A0 54 BE 00  A8 C0 C9 00 D3 11 00 00  .....T..........
00C9CC80  D0 CC C9 00 2E 6C 61 6E  67 2E 44 61 65 6D 6F 6E  .....lang.Daemon
```

UNKNOWN 00C9CC60: [heap]:00C9CC60

Stack view

```
602CAAB8  00B19FB0  [heap]:00B19FB0
602CAABC  00B1A000  [heap]:00B1A000
602CAAC0  00B1EEC0  [heap]:00B1EEC0
```

UNKNOWN 602CAAB8: debug080:602CAAB8

Output window

```
57179CFC: using guessed type _DWORD Sha256Digest::getHash(Sha256Digest *__hidden this, unsigned __int8 *);
57179D18: using guessed type int __fastcall sub_57179D18(_DWORD, _DWORD, _DWORD, _DWORD);
5717BD97: using guessed type char byte_5717BD97;
5717BDA4: using guessed type int dword_5717BDA4[31];
```

Dalao

QQ:79234668

http://blog.sina.com.cn/dalao

blog.sina.c

以下是对应的汇编代码

```
libsign.so:5717963C LDR             R3, [R0,R3]
libsign.so:5717963E LDR             R6, [SP,#0x24]
libsign.so:57179640 ADD             R1, PC ; dword_5717BDA4
libsign.so:57179642 STR             R3, [SP,#0x14]
libsign.so:57179644 STR             R6, [SP,#0xC]
libsign.so:57179646
libsign.so:57179646 loc_57179646                            ; CODE XREF: libsign.so:57179600j
libsign.so:57179646 MOVS            R0, R7
libsign.so:57179648 BLX             sub_5717939C      ==========》关键的子程序
libsign.so:5717964C ADD             R0, SP, #0xB8
libsign.so:5717964E BL              _ZN12Sha256Digest5resetEv ; Sha256Digest::reset(void)
libsign.so:57179652 MOVS            R0, R7
libsign.so:57179654 BLX             strlen_0
libsign.so:57179658 MOVS            R1, R7
libsign.so:5717965A MOVS            R2, R0
libsign.so:5717965C ADD             R0, SP, #0xB8
```

对比一下 2 部分的代码，汇编代码看参数很难看清楚，但是伪 c 代码的参数还有程序执行流程就非常的清晰了。

不难看出 F5 功能的强大。

再这个关键点上设断点看参数入口。按 F8 后来到了下面这张图

IDA - C:\Users\ADMINI~1\AppData\Local\Temp\ida09930.idb (app_process)

File Edit Jump Search View Debugger Options Windows Help

Remote ARMLinux/Android debugger

Library function Data Regular function Unexplored Instruction External symbol

Debug View | Structures | Enums

IDA View-PC | Pseudocode-A

```
76  memset_0(v14);
77  if ( v30 )
78  {
79    v15 = dword_5717DCFC[10 * v32 + v31];
80    v19 = dword_5717DCFC[v35 + 10 * v34];
81    v20 = dword_5717DCFC[10 * v33 + v36];
82    v17 = (char *)dword_5717BDA4;
83  }
84  else
85  {
86    v15 = dword_5717DCFC[10 * v32 + v31];
87    v16 = dword_5717DCFC[10 * v34 + v35];
88    v17 = &byte_5717BD97;
89    v18 = dword_5717DCFC[10 * v33 + v36];
90  }
91  ((void (__fastcall *)(unsigned __int8 *, char *, int))sub_5717D9D8)(v14, v17, v15);
92  Sha256Digest::reset((Sha256Digest *)&v39);
93  v21 = strlen_0(v14);
94  Sha256Digest::update((Sha256Digest *)&v39, v14, v21);
95  Sha256Digest::finalise((Sha256Digest *)&v39);
96  Sha256Digest::getHash((Sha256Digest *)&v39, (unsigned __int8 *)v37);
97  v22 = 0;
98  do
99  {
100   v23 = &v38[2 * v22];
101   v24 = (unsigned __int8)v37[v22++];
```

UNKNOWN Java_com_immomo_momo_util_jni_Codec_xxoo:91

按F8单步调试后点V14变量得到变量地址

General registers | IDA View

```
R0  000001C1
R1  4012B69C  debug006:__stack_chk_guard
R2  5900738A  icudt46l.dat:5900738A
R3  00C442A1  [heap]:00C442A1
R4  00000002
R5  00C9CC60  [heap]:00C9CC60
R6  00BDBC78  [heap]:00BDBC78
```

Modules | Module: libs

Path

/data/data/com.immomo.momo/lib/libmjni.so
/data/data/com.immomo.momo/lib/libsign.so
/system/bin/app_process
/system/bin/linker

Line 2 of 110

Threads

Decimal	Hex	State
2428	97C	Ready
2426	97A	Ready
2424	978	Ready
2422	976	Ready
2494	9BE	Ready
2495	9BF	Ready

Hex View-1

```
00C440E0  6F 75 31 33 34 39 32 32  38 37 39 38 32 38 31 37  ou13492287982817
00C440F0  30 36 41 30 63 6F 6D 2E  69 6D 6D 6F 6D 6F 2E 6D  06A0com.immomo.m
00C44100  6F 6D 6F 7B 61 63 63 6F  75 6E 74 3A 31 33 30 30  omo{account:1300
00C44110  36 35 30 36 36 36 36 2C  61 70 6B 73 69 67 6E 3A  6506666,apksign:
00C44120  34 66 33 61 35 33 31 63  61 66 66 33 65 33 37 63  4f3a531caff3e37c
00C44130  32 37 38 36 35 39 63 63  37 38 62 66 61 65 63 63  278659cc78bfaecc
```

UNKNOWN 00C44100: [heap]:00C44100

添入V14参数地址后这里得到想要找的结果

Stack view

```
602CAAB8  CF3E07EA
602CAABC  0004CB1D
602CAAC0  5717BDEE  libsign.so:5717BDEE
602CAAC4  00BDBC78  [heap]:00BDBC78
602CAAC8  00BE0708  [heap]:00BE0708
602CAACC  5717BE8D  libsign.so:5717BE8D
```

UNKNOWN 602CAAB8: debug080:602CAAB8

Output window

```
57179CFC: using guessed type _DWORD Sha256Digest::getHash(Sha256Digest *__hidden this, unsigned __int8 *);
57179D18: using guessed type int __fastcall sub_57179D18(_DWORD, _DWORD, _DWORD, _DWORD);
5717BD97: using guessed type char byte_5717BD97;
5717BDA4: using guessed type int dword_5717BDA4[3];
5717BDB3: using guessed type char byte_5717BDB3;
```

这里已经可以得到想要的数据结果了。我的教程也告一段落。希望能帮助到大家
大老
QQ:79234668
http://blog.sina.com.cn/dalaoqd
20140618

IDA 调试原生程序

1. IDA 调试原生程序
软件：
IDA6.5 plus（android_server 使用看雪 IDA6.1 的)
eclipse

待调试的文件
1. debugnative
支持文件
2. android_server(IDA6.1 目录下

1. 启动模拟器
emulator -avd android233
2. 上传文件
adb push debugnative /data/local/tmp
adb push android_server /data/local/tmp

3. 可执行权限
adb shell chmod 755 /data/local/tmp/debugnative
adb shell chmod 755 /data/local/tmp/android_server

4. 打开 cmd
adb shell /data/local/tmp/android_server
5. 打开 cmd2
adb forward tcp:23946 tcp:23946

6. IDA 远程调试
Debugger->Run->Remote ArmLinux/Android debugger
Application 栏, /data/local/tmp/debugnative
Directory 栏, /data/local/tmp
Hostname 栏,localhost
其他默认
7. 确定
http://blog.chinaunix.net/uid-24709751-id-4623970.html

Ida Pro 调试 FAQ

IDA 调试遇到的问题

```
C:\Users\Administrator>jdb -connect com.sun.jdi.SocketAttach:port=8899,hostname=
localhost
java.net.SocketException: Connection reset
        at java.net.SocketInputStream.read(SocketInputStream.java:189)
        at java.net.SocketInputStream.read(SocketInputStream.java:121)
        at com.sun.tools.jdi.SocketTransportService.handshake(SocketTransportSer
vice.java:130)
        at com.sun.tools.jdi.SocketTransportService.attach(SocketTransportServic
e.java:232)
        at com.sun.tools.jdi.GenericAttachingConnector.attach(GenericAttachingCo
nnector.java:116)
        at com.sun.tools.jdi.SocketAttachingConnector.attach(SocketAttachingConn
ector.java:90)
        at com.sun.tools.example.debug.tty.VMConnection.attachTarget(VMConnectio
n.java:519)
        at com.sun.tools.example.debug.tty.VMConnection.open(VMConnection.java:3
28)
        at com.sun.tools.example.debug.tty.Env.init(Env.java:63)
        at com.sun.tools.example.debug.tty.TTY.main(TTY.java:1066)

致命错误:
无法附加到目标 VM。
```

jdb 不能附加到目标是因为

有的手机默认没开 ro.debuggable，没有 adb jdwp 那个的

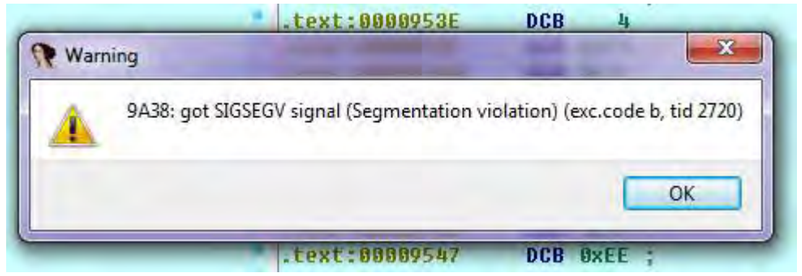
```
C:\Users\Administrator>adb shell getprop ro.debuggable
0
```

这样看到是 0,本以为 adb shell setprop ro.debuggable 1 能修改 但是还是不行

后面鬼哥发了工具修改 结果提示手机不知道 三星的烂手机

http://www.cnblogs.com/Reserved/p/4044287.html

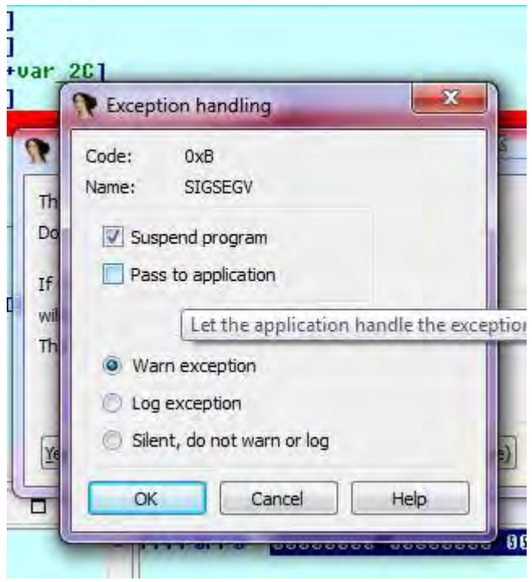
ida pro exception 处理



遇到如图情况，点 ok,

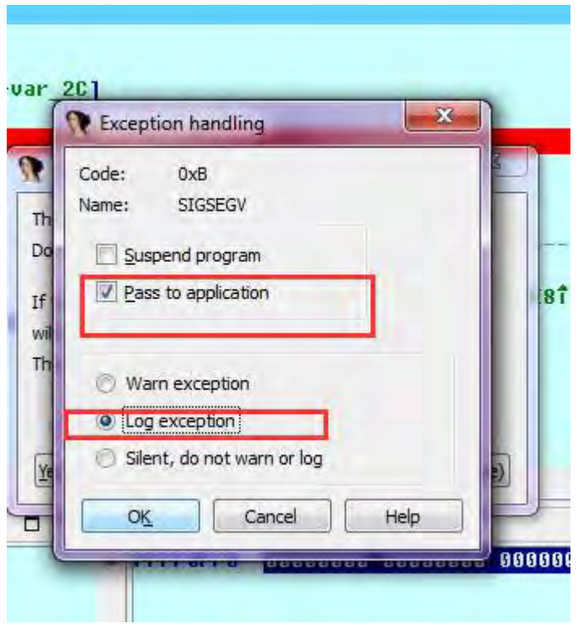
再次下断电，单步后，又会出现这个：

点击 change exception definition



出现这个界面，

将其设置为：



后点击 Ok

然后就可以愉悦的调试了。以后就不会有这种 exception 跳出来干扰你了。

IDA 动态调试时注意

调试 SO 的时候按那个键来着，换成 arm 汇编码看

人生无 NG 也就是淡然出尘

```
ibc.so:40070AB0 ; ------------------
ibc.so:40070AB4 inotify_init DCB 0x
ibc.so:40070AB5 DCB    0
ibc.so:40070AB6 DCB 0x2D ; -
ibc.so:40070AB7 DCB 0xE9 ;
ibc.so:40070AB8 DCB 0x4F ; O
ibc.so:40070AB9 DCB 0x7F ;
ibc.so:40070ABA DCB 0xA0 ;
ibc.so:40070ABB DCB 0xE3 ;
ibc.so:40070ABC DCB    0
ibc.so:40070ABD DCB    0
ibc.so:40070ABE DCB    0
ibc.so:40070ABF DCB 0xEF ;
ibc.so:40070AC0 DCB 0x90 ;
ibc.so:40070AC1 DCB    0
ibc.so:40070AC2 DCB 0xBD ;
ibc.so:40070AC3 DCB 0xE8 ;
```

选中一部分然后 按 P

提示 please position the cursor within a function

鬼谷子 c 也就是听鬼哥说故事

如果是在方法刚开始的 Push 太多没法识别，按一下键盘 P，把那里解析下就可以 F5 的了

IOSX

用 IDA Pro 调试 iPhone 应用程序

因为自 5.6 版之后，IDA Pro 就不开发自己的 iphone debugger 了，统一使用 gdb server.

参考

跨平台调试

IDA gdb debugger 调试 QEMU 模拟器中的 ARM 程序

调试远程程序

ssh 登录到 iphone

gdbserver :port programm arguments
或者
gdbserver :port -attach pid

需要安装 adv-cmds 才能有 ps 这个命令，才能看到已经启动的程序的 pid, 才能 atttach

在 Mac OS X 的
/Developer/Platforms/iPhoneOS.platform/Developer/usr/libexec/gdb/ 目录下有一个
gdb-arm-apple-darwin 文件

可以用来做跨平台调试

用 XCode 作物理机器调试时，会将
/Developer/Platforms/iPhoneOS.platform/DeviceSupport/3.1.3/DeveloperDiskImage.dmg
安装到 iPhone 上

/dev/disk1 on /Developer (hfs, local, read-only) /* DeveloperDiskImage.dmg */

是 /usr/libexec/mobile_image_mounter 将 dmg 挂载到 /dev/disk1

http://zhiwei.li/text/2010/12/%E7%94%A8ida-pro%E8%B0%83%E8%AF%95iphone%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F/?replytocom=745

IDA + GDBServer 实现 iPhone 程序远程调试

在早期的 IDA 中包含了一个 iphoneserver 的程序，这个程序就是配合 IDA 实现远程调试的。但是在最新版的 IDA 中这个东西已经不复存在了，因 而下载的破解版的 IDA 中没有那个文件并不是被删除掉了，而是本来就没有，*^_^*。所以一直以来调试 iPhone 上的二进制程序只能悲催的使用 ssh+gdb 进行调试，虽然调试器的功能还算可以，但是每次调试都需要设置显示，只能使用命令进行控制，因而用起来还是不是十分爽。
其实网上关于 IDA 实现 ios 设备远程调试的文章从网上也是可以找到的，但是说的都不是十分具体。本文主要是介绍下 IDA 实现远程 iPhone 程序调试的方法，当然这样调试还存在一些问题，如果大家有什么好的解决方案还请不吝赐教。

需要注意的是要想调试 ios 设备上的程序并不是简单的吧 gdbserver 拷贝到 ios 设备上行就可以了，此时如果使用 gdbserver 启动进程将得到类似如图 1 所示的提示信息：

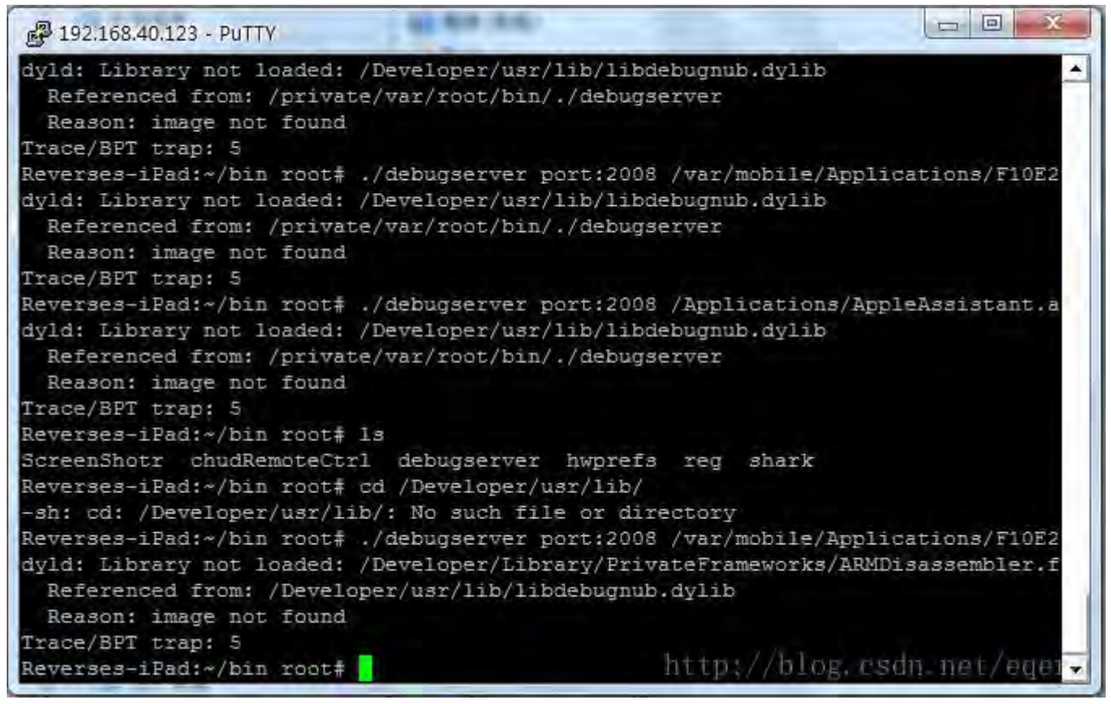


图 1

提示文件没有找到，但是如果设备曾经链接过 xcode 进行过程序调试则不会出现下面的情况。这是因为 xcode 在进行程序调试的时候会将所需要的文件拷贝 到 Develop 目录下，这样的话不论 gdbserver 在哪个目录下程序都是可以正常执行的。同样在上面的命令中看到的 debugserver 是苹果自 己编译的一个 gdbserver 调试服务器，而实际使用则基本是没有任何的差异和影响的。

首先来配置下设备上的调试环境，从 xcode 的/Developer/Platforms/iPhoneOS.platform /DeviceSupport/3.2 目录下找到 DeveloperDiskImage.dmg 文件，而 3.2 则是对应的 ios 设备上的系统版本，在 DeviceSupport 目录下通常会包含如如 2 所示的版本列表：

图 02
在实际的调试过程中只需要选择相应的版本就可以了。在每个目录下都存在一个 DeveloperDiskImage.dmg 文件，用 ultraiso 打开这个文件可以看到如图 3 的文件目录列表：

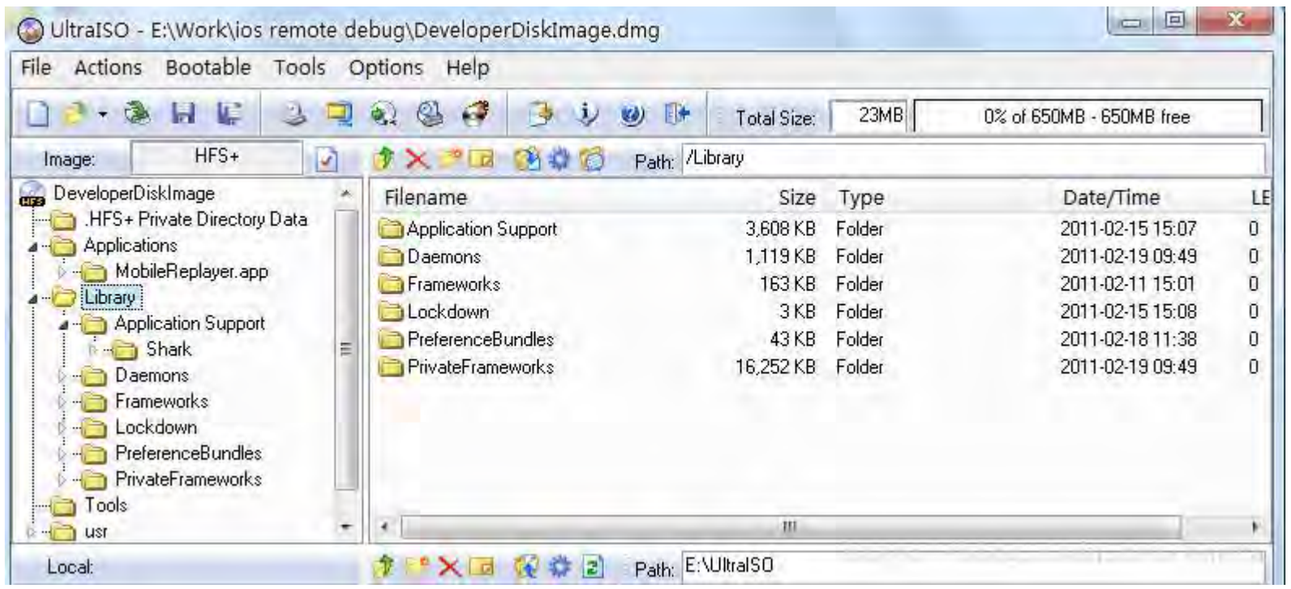


图 3
在调试的过程中需要将整个 dmg 文件下的所有目录和文件拷贝到设备上的/Developer 目录下，如图 4 所示。这一步可以使用 ios 的文件管理工具，或 者使用 winscp 上传文件，不建议使用后者，太卡了～，如果使用过 xcode 开发调试过程序那么这个目录下的所有东西应该都是存在的，可以无需手工复 制。

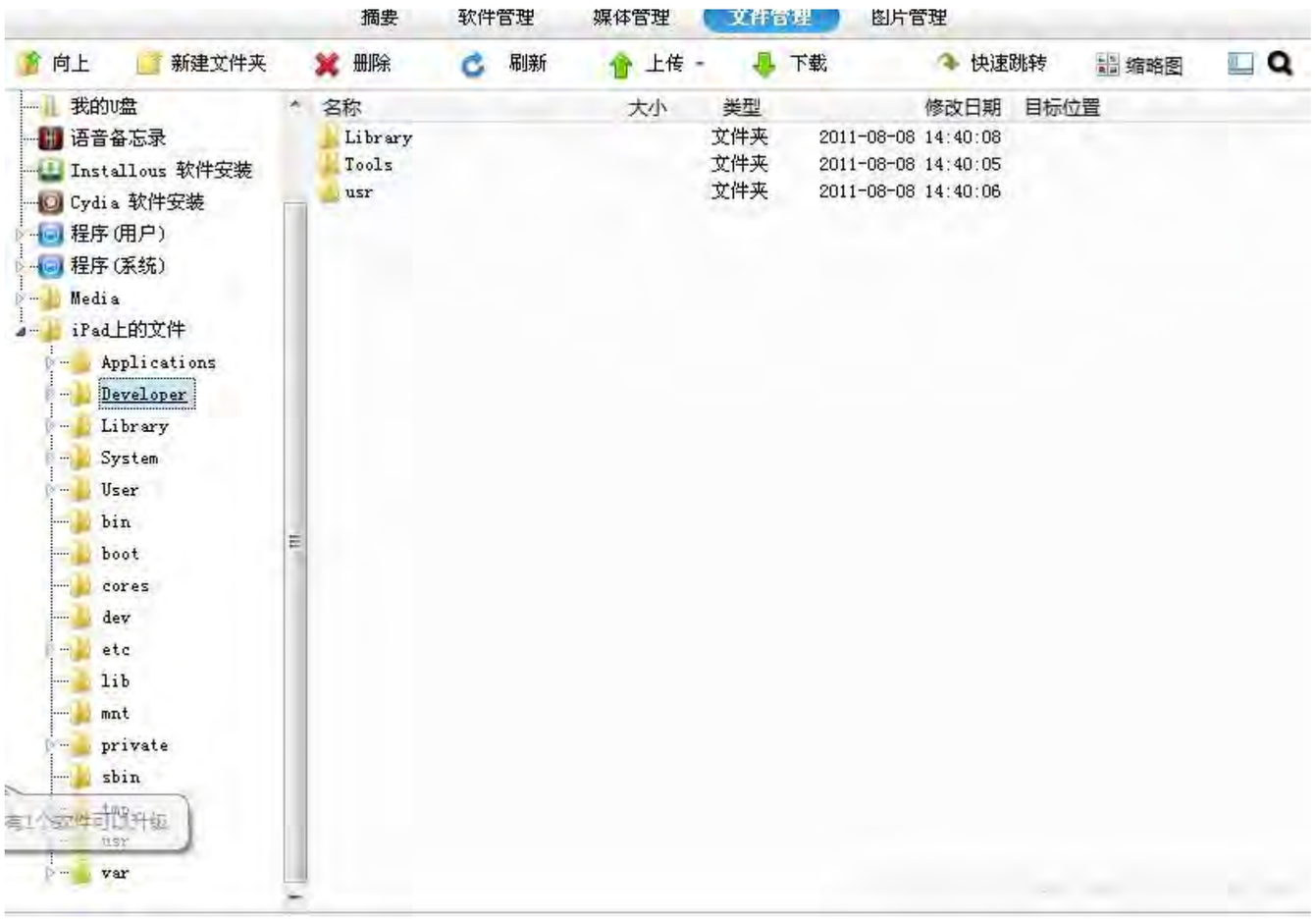
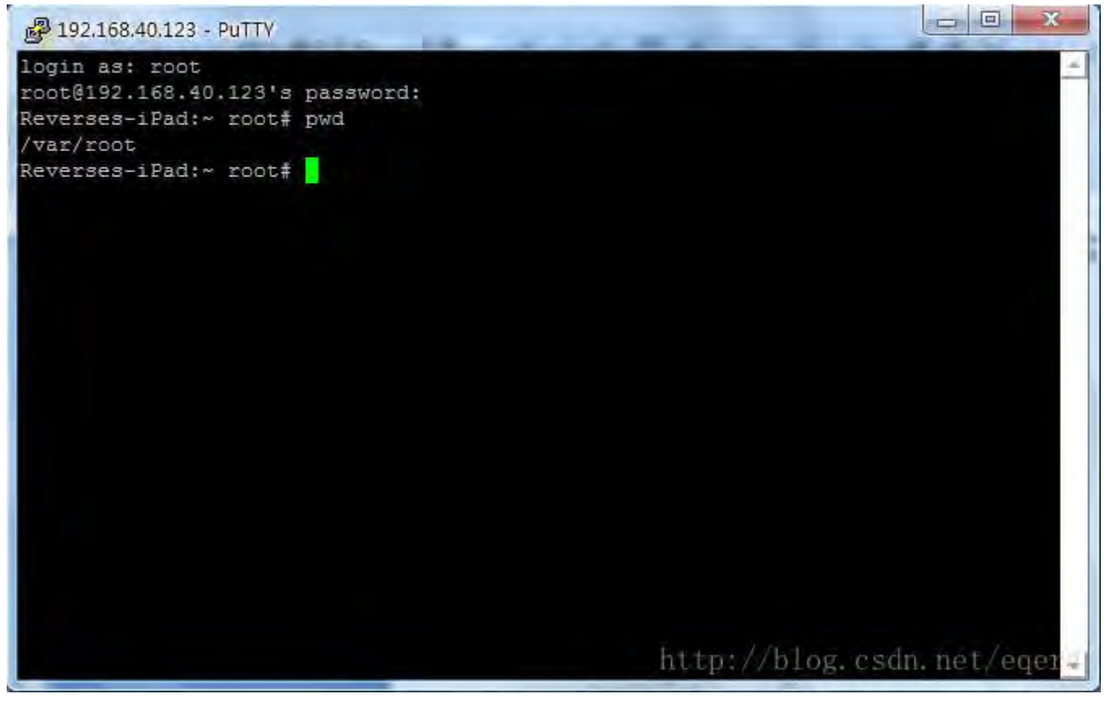


图 4

另外还需要在设备上安装 ssh 服务端，这个直接从 cydia 源中安装即可。最后就是程序的调试了。

在这里的调试可以使用预先分析文件和直接 attach 的方式进行调试。直接 attach 的方式比较简单一些，使用 ssh 客户端连接到设备（需要无线网络支持，如果木有，那我也有办法哦～），在 win 下可以使用 putty。

登录后默认会在 root 目录下，如图 5 所示



切换到/Develop/usr/bin/目录下，给 debugserver 增加执行权限，如图 6 所示

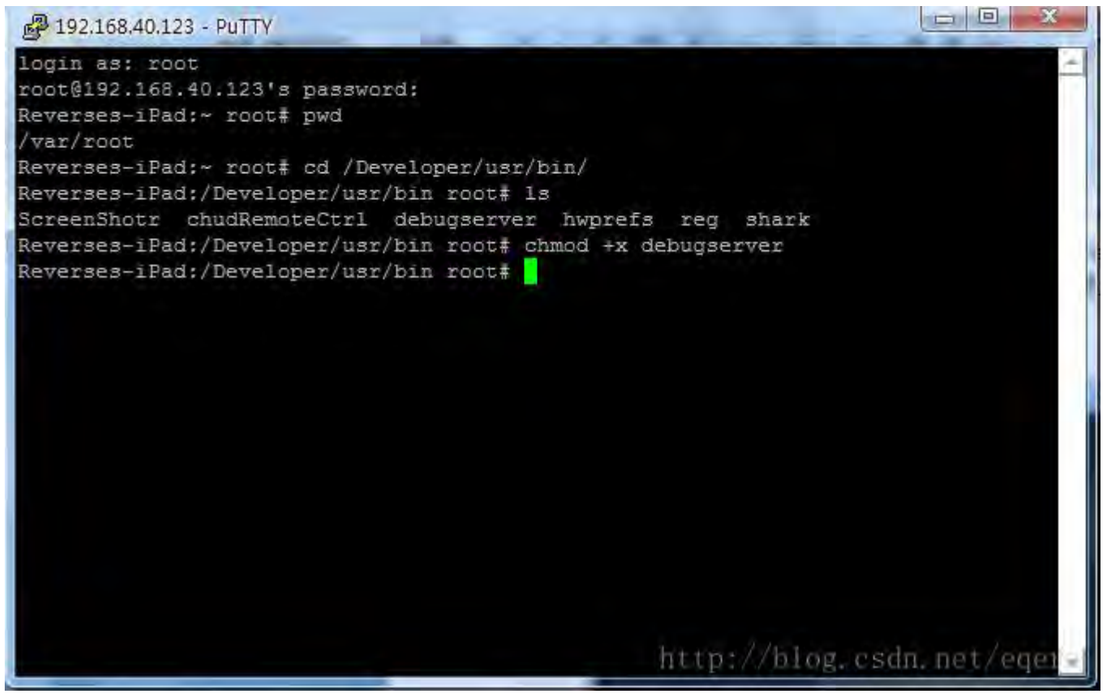


图 6

现在就可以执行一个程序了，以 AppleAssistantHD 为例，执行./debugserver port:2008 /var/mobile /Applications/ABDEE3BA-02BE-4251-A57B-ECC363813133/AppleAssistantHD.app /AppleAssistantHD，此时 debugserver 会执行目标程序并且同时打开一个端口监听远程连接，如图 7 所示

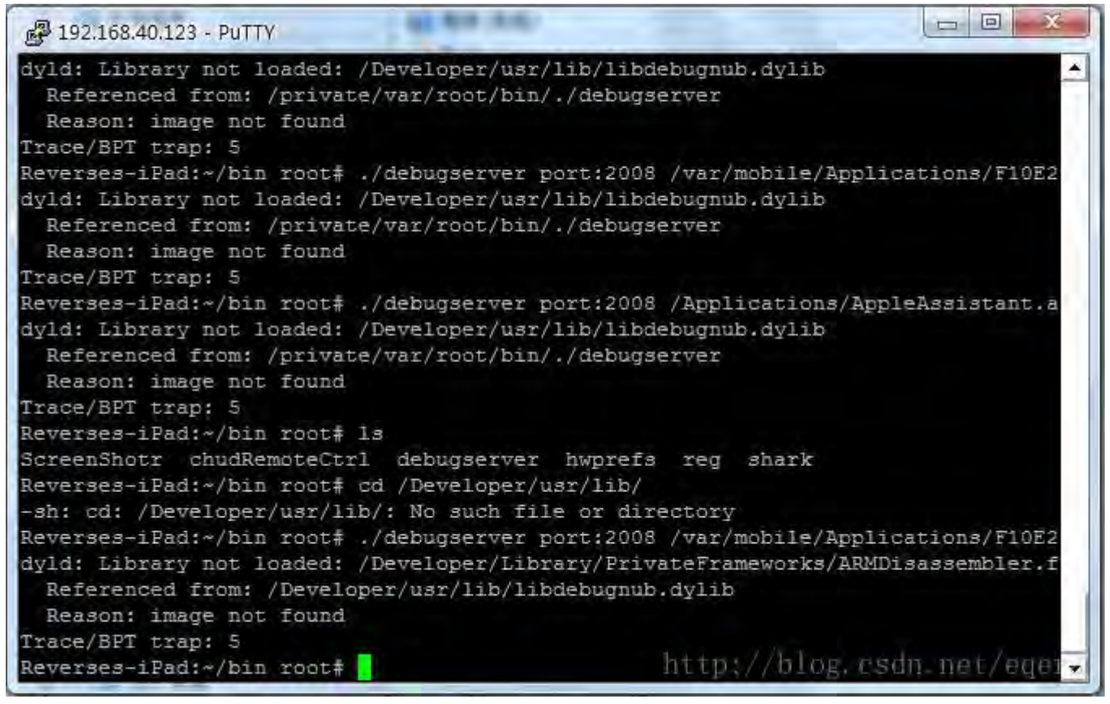


图 7

现在就可以运行 ida 链接远程调试器了。执行 ida 直接点 go 进入到程序界面即可，如图 8 所示。

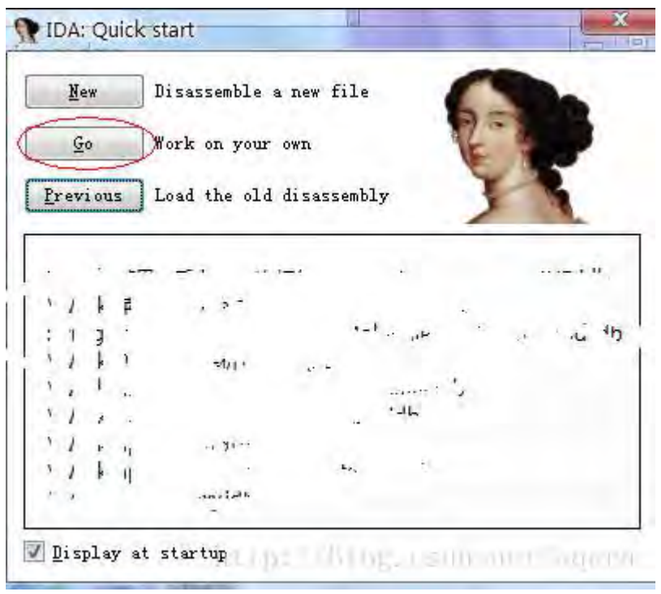
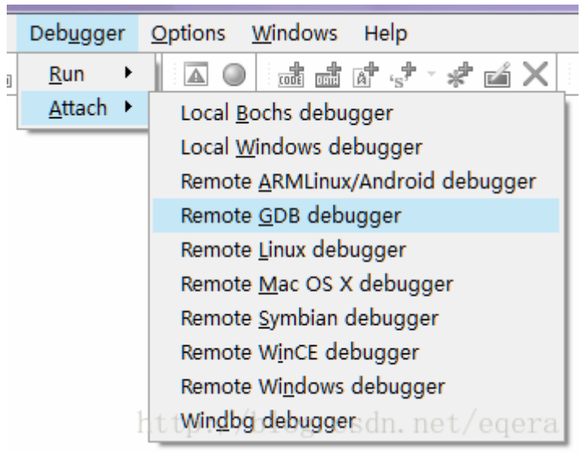


图 8
进入之后点击菜单的 Debugger->Attach->Remote GdbDebuger，如图 9 所示



执行之后将会打开如图 10 所示的调试器附加窗口。

图 10
在 Hostname 中输入设备的 ip，port 中输入 debugserver 的监听端口。点击 Debug options 打开调试选项，如图 11 所示。

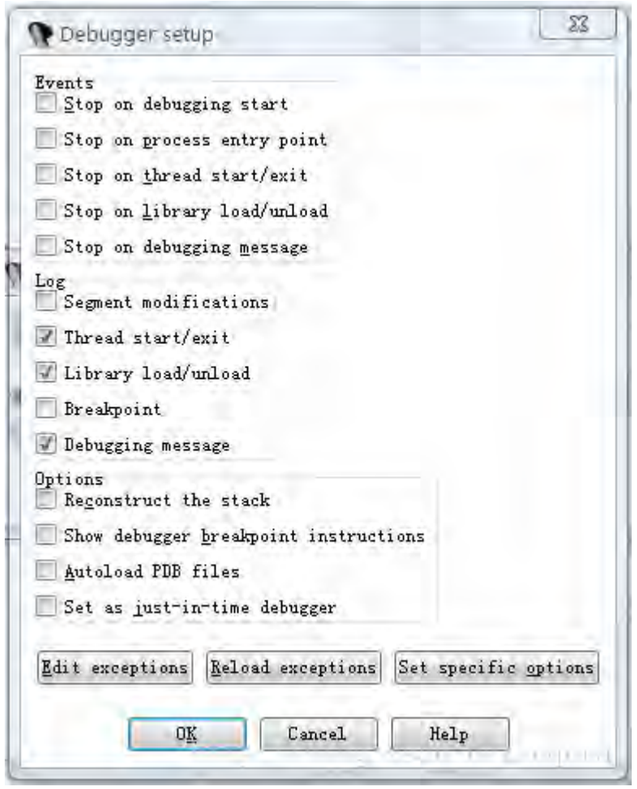


图 11

执行 Edit exceptions 打开异常处理选项窗口，如图 12 所示

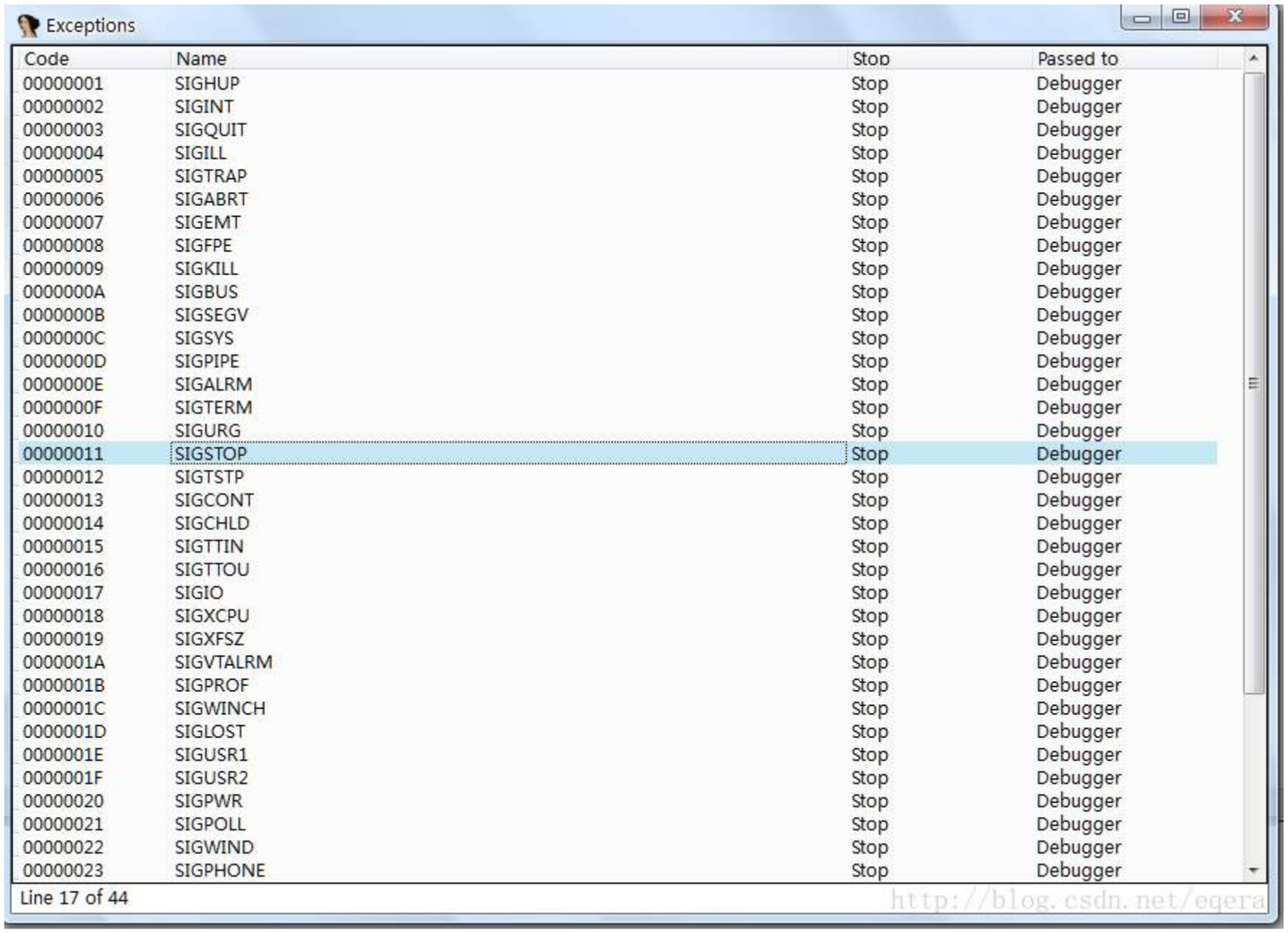


图 12

编辑 11 号异常，去掉 stop program 的勾，如图 13，否则在调试的过程中会非常的痛苦。

图 13
关闭所有的设置窗口，现在就可以进行附加了，点击 ok 之后将会弹出如图 14 所示的进程列表。

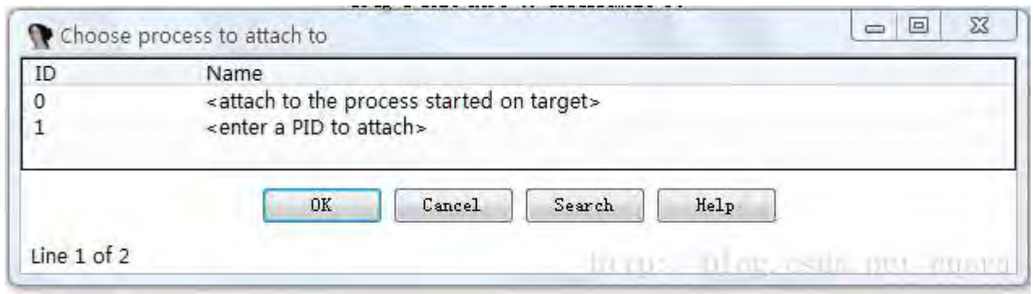


图 13
关闭所有的设置窗口，现在就可以进行附加了，点击 ok 之后将会弹出如图 14 所示的进程列表。

图 15
现在就可以进行调试了，如图 16 所示。

IDA - D:\DOCUME~1\obaby\LOCALS~1\Temp\ida20277.idb (<GDB remote process>)

File Edit Jump Search View Debugger Options Windows Help

Remote GDB debugger

IDA View-EIP, General registers, Modules, Threads, Hex View-1, Stack view | Structures | Enums

IDA View-EIP

```
MEMORY:00000000 ; +-------------------------------------------------------------------------+
MEMORY:00000000 ;
MEMORY:00000000
MEMORY:00000000 .686p
MEMORY:00000000 .mmx
MEMORY:00000000 .model flat
MEMORY:00000000
MEMORY:00000000 ; ===========================================================================
MEMORY:00000000
MEMORY:00000000 ; Segment type: Regular
MEMORY:00000000 ; Segment permissions: Read/Execute
MEMORY:00000000 MEMORY segment byte public 'UNK' use32
MEMORY:00000000 assume cs:MEMORY
MEMORY:00000000 assume es:MEMORY, ss:MEMORY, ds:MEMORY, fs:MEMORY, gs:MEMORY
MEMORY:00000000 db    ? ;
MEMORY:00000001 db    ? ;
MEMORY:00000002 db    ? ;
MEMORY:00000003 db    ? ;
MEMORY:00000004 db    ? ;
MEMORY:00000005 db    ? ;
MEMORY:00000006 db    ? ;
MEMORY:00000007 db    ? ;
MEMORY:00000008 db    ? ;
MEMORY:00000009 db    ? ;
MEMORY:0000000A db    ? ;
MEMORY:0000000B db    ? ;
MEMORY:0000000C db    ? ;
MEMORY:0000000D db    ? ;
MEMORY:0000000E db    ? ;
MEMORY:0000000F db    ? ;
MEMORY:00000010 db    ? ;
MEMORY:00000011 db    ? ;
UNKNOWN 00000000: MEMORY:00000000
```

EDX
ESI
EIP
CS

General registers

```
EAX 10004005  MEMORY:10004005
ECX 07000006  MEMORY:07000006
EDX 00000000  MEMORY:00000000
EBX 00000C00  MEMORY:00000C00
ESP 00001203  MEMORY:00001203
EBP FFFFFFFF
ESI 00000000  MEMORY:00000000
EDI 2FDFEC30  MEMORY:2FDFEC30
```

Modules

Path
<GDB remote process>

Threads

Decimal	Hex	State
11011	2B03	Ready
12291	3003	Ready
12035	2F03	Ready
11779	2E03	Ready
11523	2D03	Ready
11267	2C03	Ready

Hex View-1

```
FFEFFFEF  ?? ?? ?? ?? ?? ?? ?? ??  ?? ?? ?? ?? ?? ?? ?? ??  ????????????????

UNKNOWN FFEFFFEF: MEMORY:FFEFFFEF
```

Stack view

```
00001203  0004D400  MEMORY:0004D400
00001207  0AFCA000  MEMORY:0AFCA000

UNKNOWN 00001203: MEMORY:00001203
```

Output window

```
You may start to explore the input file right now.
Hex-Rays Decompiler plugin has been loaded (v1.5.0.110408)
  License: 57-BB7B-79C4-FF Tu Qiao Ying, Xiamen Tongbu Networks Ltd (1 user)
  The hotkeys are F5: decompile, Ctrl-F5: decompile all.
  Please check the Edit/Plugins menu for more informaton.

LoadMap: Plugin init.

------------------------------------------------------------------------------
Python 2.6.5 (r265:79096, Mar 19 2010, 21:48:26) [MSC v.1500 32 bit (Intel)]
IDAPython v1.5.0 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>
------------------------------------------------------------------------------
The initial autoanalysis has been finished.
```

GDB

图 16
上面的是直接 attach 的方法，这种方法能看到的只有内存数据，所以在实际的调试中并不是最好的方法。另外一种方式是直接用 ida 载入文件之后采用链接 远程调试器的方式。首先将调试器设置为 Remote gdb debuger 然后编辑 ida 的 Debug application setup:gdb 设 置，如图 17 所示：

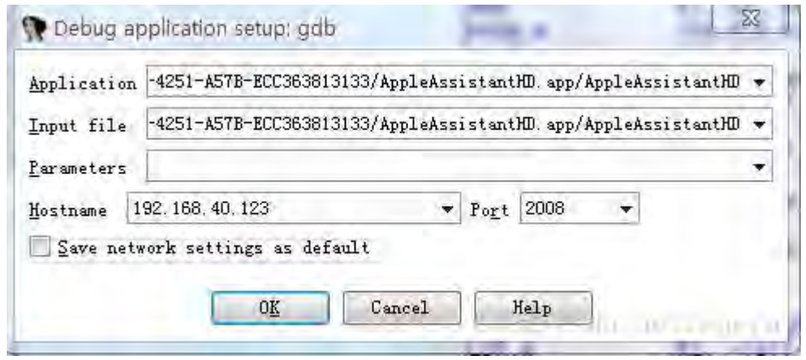


图 17
在 application 和 input file 中输入设备上的文件的绝对路径，其余的设置与 attach 的方式是完全一致的。设置完成之后就可以启动调试器了，此时会提示已经有远程进程在被调试，是否附加到，如图 18.

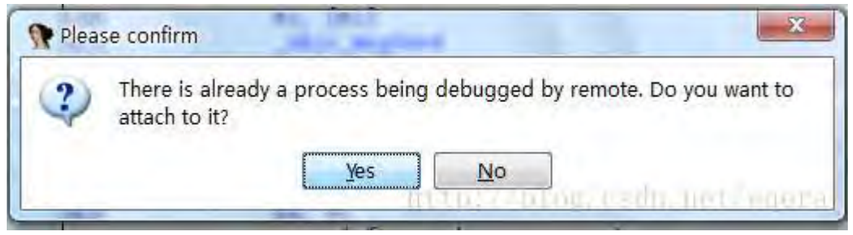


图 18
点击 yes 之后就可以进行调试了，如果附加成功将会弹出提示信息。这样调试的时候代码看起来还是比较清晰的，如图 19

IDA - [E:\Work\苹果助手 ipa 二次打包\ipad ipa\AppleAssistantHD.idb (AppleAssistantHD) - Running

File Edit Jump Search View Debugger Options Windows Help

Remote GDB debugger

IDA View-PC, Breakpoints, General registers, Modules, Threads, Hex View-1, Stack view

Structures

Enums

IDA View-PC

Breakpoints

```
UNDEF:00182F94                                  ; DATA XREF: __cfstring:cfstr_ErrorRemovingSTo
UNDEF:00182F94                                  ; __cfstring:cfstr_PeerNameParameTo ...
UNDEF:00182F98 IMPORT _kCFAllocatorDefault      ; DATA XREF: __nl_symbol_ptr:_kCFAllocatorDefault_ptrTo
UNDEF:00182F9C IMPORT _kCFBooleanFalse          ; DATA XREF: __nl_symbol_ptr:_kCFBooleanFalse_ptrTo
UNDEF:00182FA0 IMPORT _kCFBooleanTrue           ; DATA XREF: __nl_symbol_ptr:_kCFBooleanTrue_ptrTo
UNDEF:00182FA4 IMPORT _kCFNull                  ; DATA XREF: __nl_symbol_ptr:_kCFNull_ptrTo
UNDEF:00182FA8 IMPORT _kCFRunLoopCommonModes    ; DATA XREF: __nl_symbol_ptr:_kCFRunLoopCommonModes_ptrTo
UNDEF:00182FAC IMPORT _kCFRunLoopDefaultMode    ; DATA XREF: __nl_symbol_ptr:_kCFRunLoopDefaultMode_ptrTo
UNDEF:00182FB0 IMPORT _kCFStreamErrorDomainSOCKS ; DATA XREF: __nl_symbol_ptr:_kCFStreamErrorDomainSOCKS_ptrTo
UNDEF:00182FB4 IMPORT _kCFStreamErrorDomainSSL  ; DATA XREF: __nl_symbol_ptr:_kCFStreamErrorDomainSSL_ptrTo
UNDEF:00182FB8 IMPORT _kCFStreamPropertySOCKSProxy ; DATA XREF: __nl_symbol_ptr:_kCFStreamPropertySOCKSProxy_ptrTo
UNDEF:00182FBC IMPORT _kCFStreamPropertySOCKSProxyHost ; DATA XREF: __nl_symbol_ptr:_kCFStreamPropertySOCKSProxyHost_ptrTo
UNDEF:00182FC0 IMPORT _kCFStreamPropertySOCKSProxyPort ; DATA XREF: __nl_symbol_ptr:_kCFStreamPropertySOCKSProxyPort_ptrTo
UNDEF:00182FC4 IMPORT _kCFStreamPropertyShouldCloseNativeSocket
UNDEF:00182FC4                                  ; DATA XREF: __nl_symbol_ptr:_kCFStreamPropertyShouldCloseNativeSocket_ptrTo
UNDEF:00182FC8 IMPORT _kCFStreamPropertySocketNativeHandle
UNDEF:00182FC8                                  ; DATA XREF: __nl_symbol_ptr:_kCFStreamPropertySocketNativeHandle_ptrTo
UNDEF:00182FCC IMPORT _kCFTypeDictionaryValueCallBacks ; DATA XREF: __nl_symbol_ptr:_kCFTypeDictionaryValueCallBacks_ptrTo
UNDEF:00182FCC
UNDEF:00182FCC END start
```

UNKNOWN 00182FCC: UNDEF: kCFTypeDictionaryValueCallBacks

General registers

R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10

Modules

Path

Threads

Decimal Hex State

Hex View-1

```
0005C4B0  58 41 07 00 B8 25 07 00  3E 39 07 00 98 26 07 00  XA....%..>9....
0005C4C0  CC 27 07 00 60 25 07 00  E2 40 07 00 EE 2A 07 00  ..'..`%..@...*..
```

0005B4F0 0005C4F0: InstalledApplicationCell rebuildPackage

Stack view

```
0005C4F0  AF005F0
0005C4F4  0D00E92D
```

0005B4F0 0005C4F0: InstalledApplicationCell rebuildPackage

Output window

```
FFFFFFFF: process  has started (pid=4294967294)
Debugger: attached to process <GDB remote process> (pid=4294967294)
```

GDB

图 19

虽然现在可以调试了，但是还有几个问题。也是我现在没有处理掉的：

1. 在静态分析的时候设置的断点会变为无效，如图 20 所示；

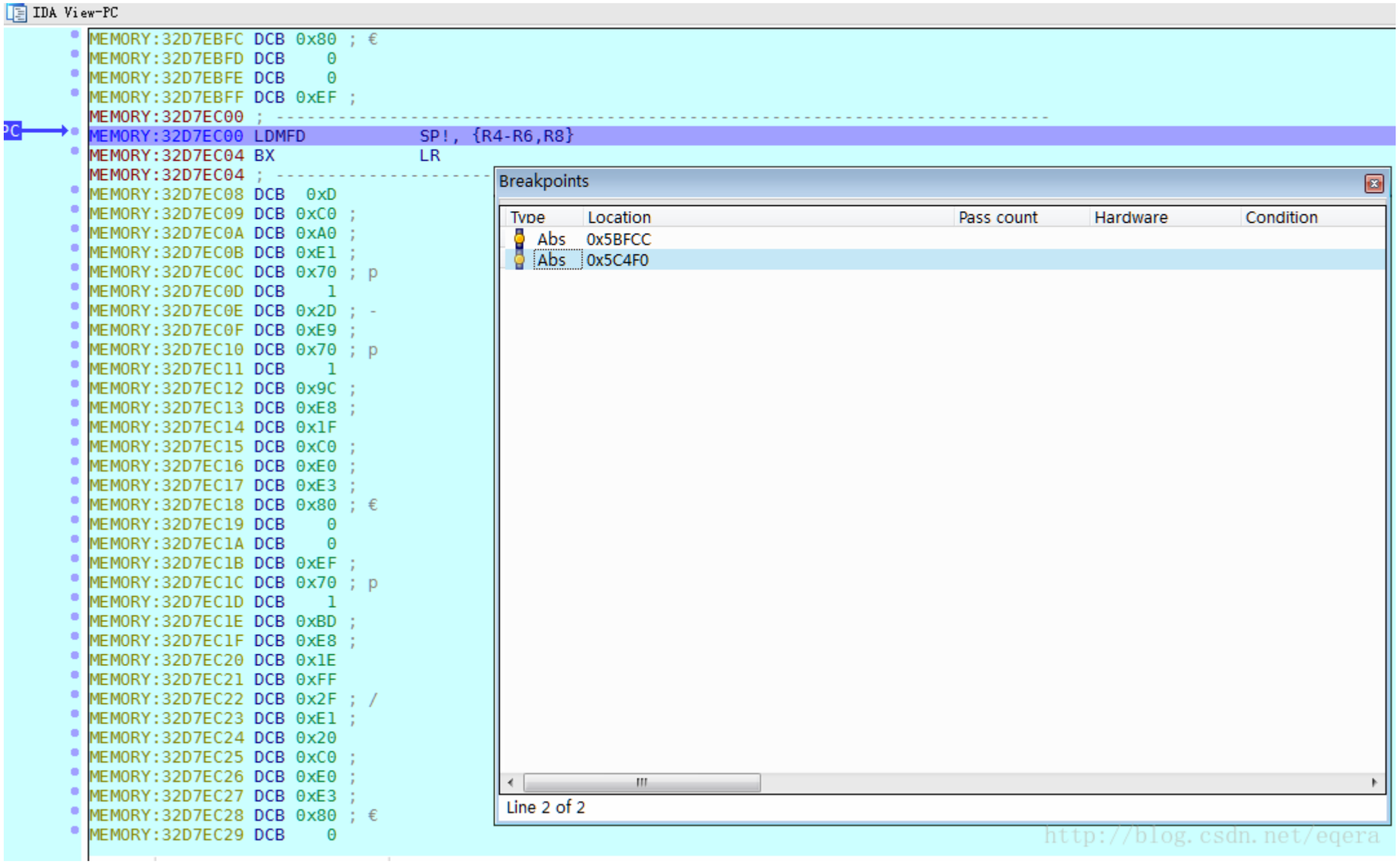


图 20

2. 一旦将进程挂起之后重启启动进程已经卡死了，没有任何的响应。

由于上面的两个问题导致现在 ida 虽然可以附加到进程或者启动独立的进程进行调试，但是实际的效果并不理想。明显是一锤子买卖，一旦暂停就完全死翘翘了，这也就是在文章开始处提到的那个问题。所以对于上面的两个问题哪位大大有好的解决办法还望不吝赐教

http://blog.csdn.net/eqera/article/details/18927547

IDA6.1 远程调试 Mac OS X 程序

参考资料
IDA Mac OS X Debugging

IDA Debugger: Mac Debugger Preview

在 IDA 6.1 的目录下有一个 mac_server，是用来在远程端（Mac 机器）上运行的小程序
如果你的 Mac 是 64 位的，需要使用 mac_serverx64

把它上传到 Mac 机器的适当目录， 比如 /Users/zhiwei/bin, 并设置适当权限
```
$ sudo chgrp procmod mac_server
$ sudo chmod g+s mac_server
```
查看是否设置成功
```
$ls -l mac_server
-rwxr-sr-x 1 zhiwei procmod 545996 Dec 31 04:11 mac_server
```

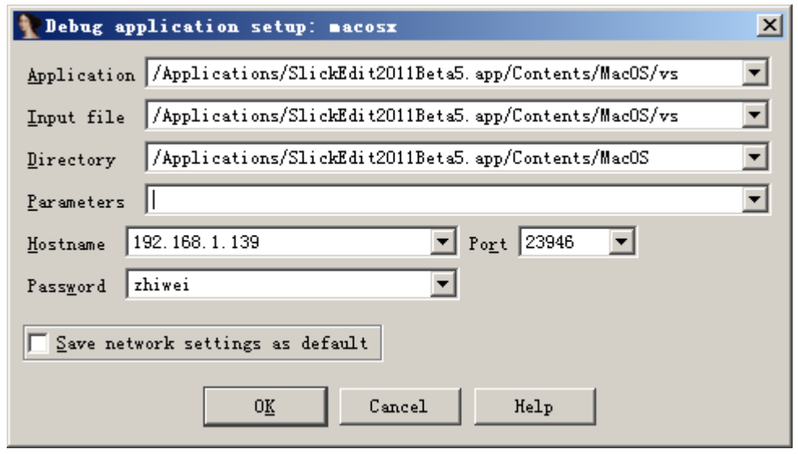
如果没有 x 权限，先 chmod +x mac_server, 然后再执行 chmod g+s mac_server 就可以了
请注意它 属于 procmod 组， 并且该组有 set root 权限

启动 mac_server
```
./mac_server -Pzhiwei
IDA Mac OS X 32-bit remote debug server(MT) v1.14. Hex-Rays (c) 2004-2011
Listening on port #23946…
```

其中 zhiwei 就是密码（口令）， 从 IDA 连接上来时要用到
默认监听在 23946 端口

现在就可以通过 IDA 来创建新的进程或者附加到一个已经存在的进程上去

在 Debugger -> Process Options 里设置下马的参数




```
2011-12-31 04:39:42.298 vs[753:903] License directory permissions need correction
2011-12-31 04:39:42.395 vs[753:903] permitWithRights returned -60007
```

http://zhiwei.li/text/2011/12/ida6-1%E8%BF%9C%E7%A8%8B%E8%B0%83%E8%AF%95mac-os-x%E7%A8%8B%E5%BA%8F/

Linux

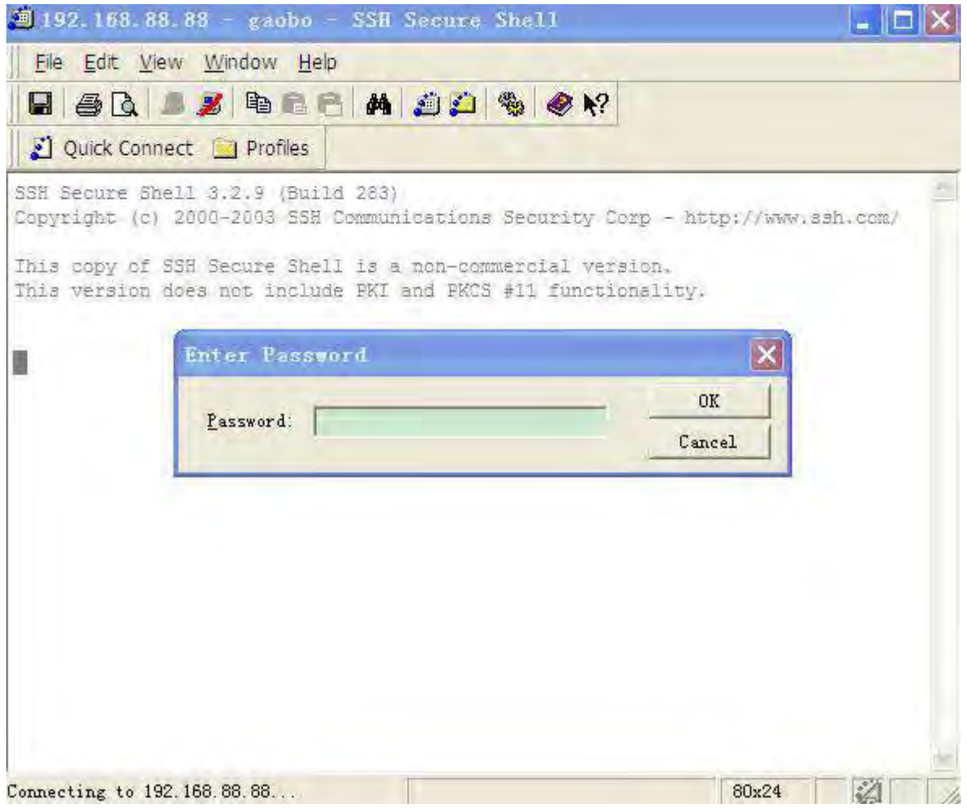
我用 IDA 远程调试 LINUX 下程序的笔记（附图）

首先说明下这是自己的实际学习经验的小小记录，顺便贴出来给需要的同样作为 LINUX 新人的同学。虽然遇到的问题很简单，但有时可能会浪费点时间想一下。

我使用的参考资料是 IDA 的帮助文档以及 arthurchen 翻译的《007 使用 IDA 专业版进行远程调试》，感谢下。大家可以在坛子里搜。

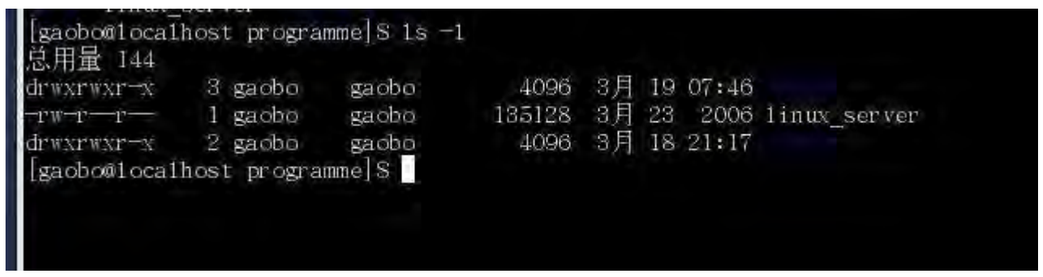
建议新同学可以用 VMWARE 虚拟机来装 LINUX，我用的是红帽子的 FEDORA 版本。注意在最新的 FEDORA10 中库文件需要自己更新。所以我还是用老版本的。

为了让 VM 中的 LINUX 和我的 WINXP 通信，可以使用 SSH secure Shell,很好用，具体怎么装大家去 GOOGLE 啦。



1 步：在 IDA 目录下找到 linux_server 程序，把她通过 SSH 传到 LINUX 的某个文件目录下。

2 步：修改文件访问权限。我在这里卡了一下，开始以为是程序本身问题，后来想到无法执行应该是权限问题。如下图



使用 ls - l 命令查看，会发现 linux_server 的访问权限无法执行，所以需要修改。

要改访问权限需要在 ROOT 用户状态下，所以首先用 su - root 这种比较方便的方式进入 ROOT 状态，然后使用 chmod a+x 命令将文件改成所有用户可执行。如图：

```
文件(F)  编辑(E)  查看(V)  终端(T)  转到(G)  帮助(H)
[gaobo@localhost gaobo]$ ls
                                    下载任务.txt
[gaobo@localhost gaobo]$ cd programme/
[gaobo@localhost programme]$ ls
    linux_server
[gaobo@localhost programme]$ ls -l
总用量 144
drwxrwxr-x    3 gaobo    gaobo        4096  3月 19 07:46
-rw-r--r--    1 gaobo    gaobo      135128  3月 23  2006 linux_server
drwxrwxr-x    2 gaobo    gaobo        4096  3月 18 21:17
[gaobo@localhost programme]$ su - root
Password:
[root@localhost root]# cd /home/gaobo/programme/
[root@localhost programme]# ls -l
总用量 144
drwxrwxr-x    3 gaobo    gaobo        4096  3月 19 07:46
-rw-r--r--    1 gaobo    gaobo      135128  3月 23  2006 linux_server
drwxrwxr-x    2 gaobo    gaobo        4096  3月 18 21:17
[root@localhost programme]# chmod a+x linux_server
[root@localhost programme]# ls -l
总用量 144
drwxrwxr-x    3 gaobo    gaobo        4096  3月 19 07:46
-rwxr-xr-x    1 gaobo    gaobo      135128  3月 23  2006
drwxrwxr-x    2 gaobo    gaobo        4096  3月 18 21:17
[root@localhost programme]#
```

3 步：回到自己的用户状态，启动./linux_server

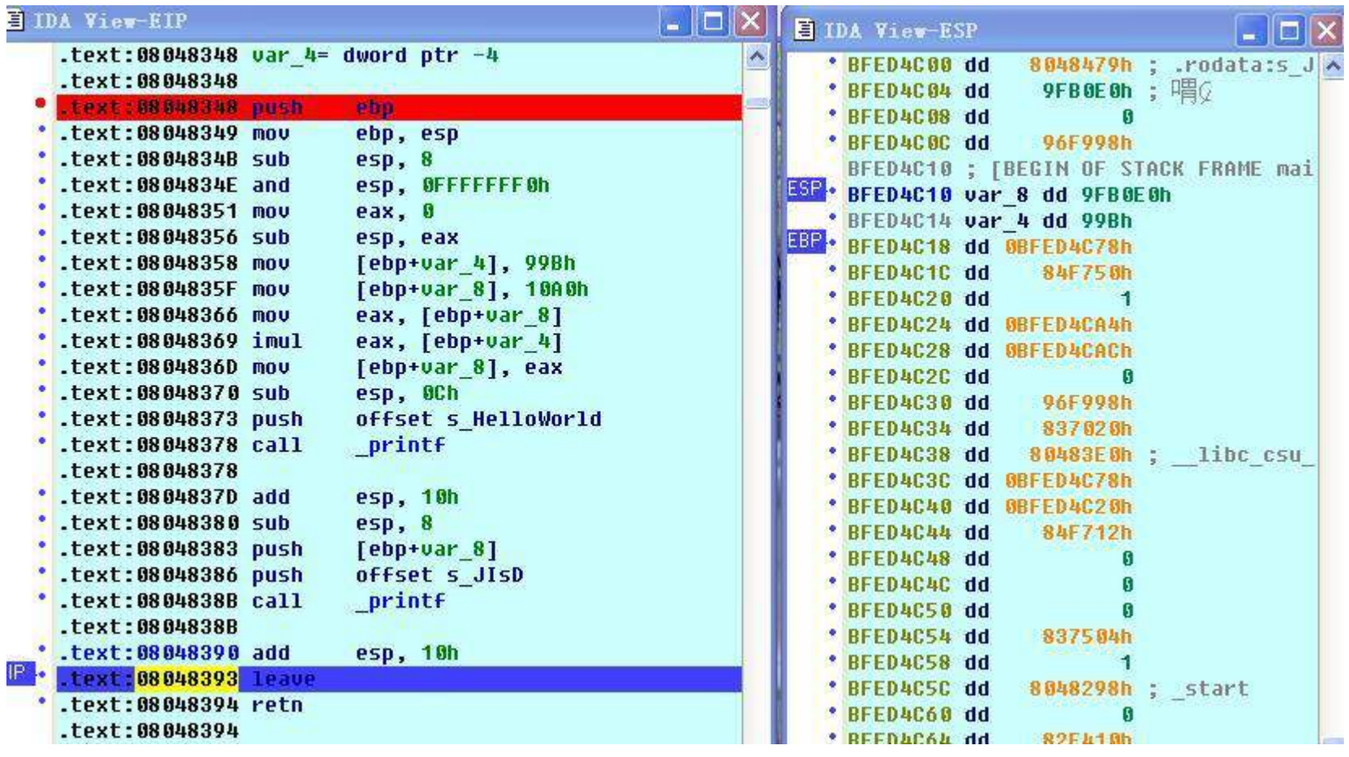
```
[gaobo@localhost programme]$ ls -l
总用量 144
drwxrwxr-x    3 gaobo    gaobo        4096  3月 19 07:46
-rwxr-xr-x    1 gaobo    gaobo      135128  3月 23  2006
drwxrwxr-x    2 gaobo    gaobo        4096  3月 18 21:17
[gaobo@localhost programme]$ ./linux_server
IDA Linux remote debug server. Version 1.9. Copyright Datarescue 2004-2006
Listening on port #23946...
```

4 步 使用 PWD 命令查看自己的 HELLO 程序路径，这个是后面需要的。注意因为 LINUX 的可执行程序没有后缀，所以很容易和文件目录混淆。

```
[gaobo@localhost hello]$ ls
hello  hello.c  hellodump  hello.s
[gaobo@localhost hello]$ pwd
/home/gaobo/programme/Cprj/hello
[gaobo@localhost hello]$
```

5 步 将自己的 HELLO 程序拷贝到 WIN 下面随便一个地方，用 IDA 正常启动，设置进程选项，里面就要用到在 LINUX 里程序的路径，一定要对应，然后 LINUX 的 IP 或用户名写上。如图

6 步 好了，可以启动调试了，这就是我的 HELLO WORLD 的反汇编，执行它



然后在 LINUX 终端中可以看到执行结果如下，hello world !!!

好了，第一次写点小心得，希望大家多多支持，共同进步!!!
http://bbs.pediy.com/showthread.php?p=604379#poststop

Windows

使用 IDA 的调试器

一个小 bug 程序

这个小程序只是简单的计算了一下一组数据（1,2,3,4,5）的平均值。这组数据被保存在两个数组里，一个是 8bit 的数值，一个是 32bit 数值表示。

```
#include <stdio.h>

charchar_average(char array[],int count)

{

int i;

char average;

average = 0;

for (i = 0; i < count; i++)

average += array[i];
```

```
average /= count;
return average;
}

int int_average(int array[], int count)
{
int i,average;
average = 0;
for (i = 0; i < count; i++)
average += array[i];
average /= count;
return average;
}

voidmain(void)
{
char chars[]= { 1, 2, 3, 4, 5 };
int integers[]= { 1, 2, 3, 4, 5 };
printf("chars[] - average = %d\n",
char_average(chars, sizeof(chars)));
printf("integers[] - average = %d\n",
int_average(integers, sizeof(integers)));
```

```
}
```

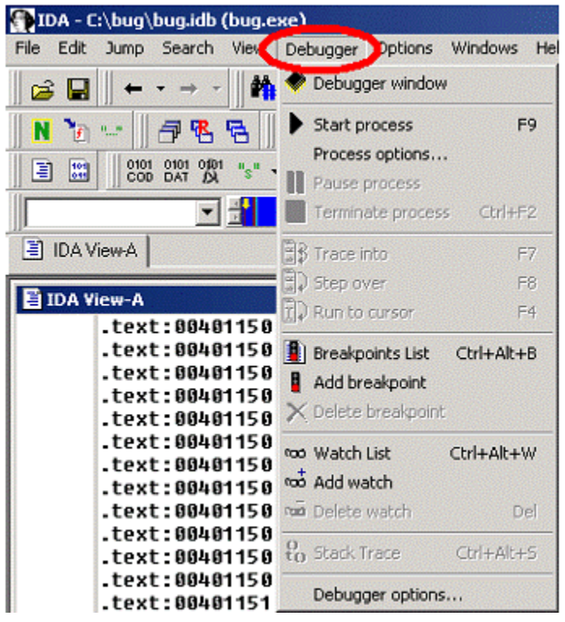
运行它，我们得到如下结果：

```
chars[]- average = 3

integers[]- average = 1054228
```

可以看出在整数方式下计算的结果是错误的，我们用 IDA 的调试器来分析一下吧。

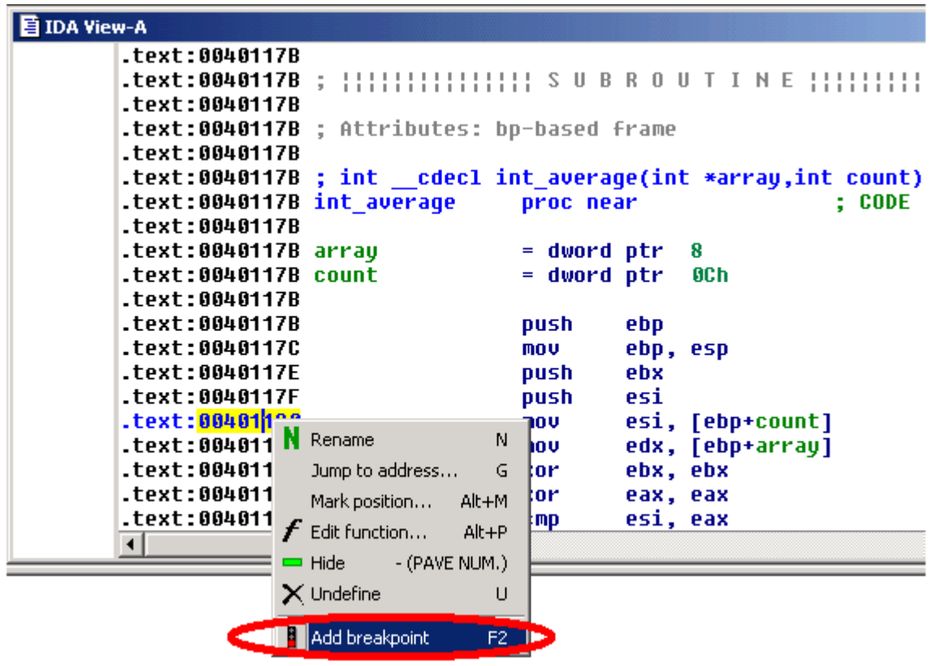
装入文件

调试器被完美的集成在 IDA 中，首先我们使用 IDA 装入文件，来生成数据库，用户可以使用反汇编功能，所有反汇编的信息，均可以在调试器中使用。如果反汇编的文件被调试器正确识别，调试器的菜单将自动出现在 IDA 的主窗口中。



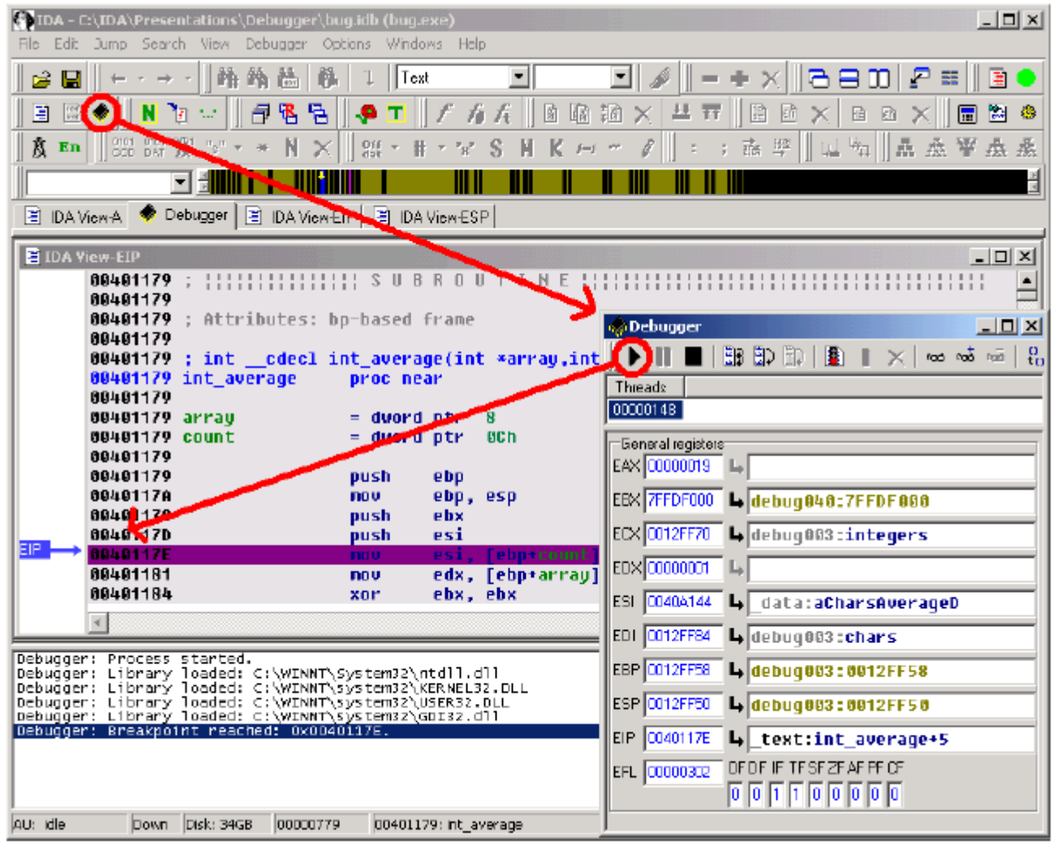
指令断点

首先我们在反汇编窗口中找到 *int_average()*这个函数，我们在程序前导部分（指程序开始的一些初始化等）的后面下断点，可以使用右键菜单中的 *AddBreakpoint* 的命令，或使用快捷键 F2。



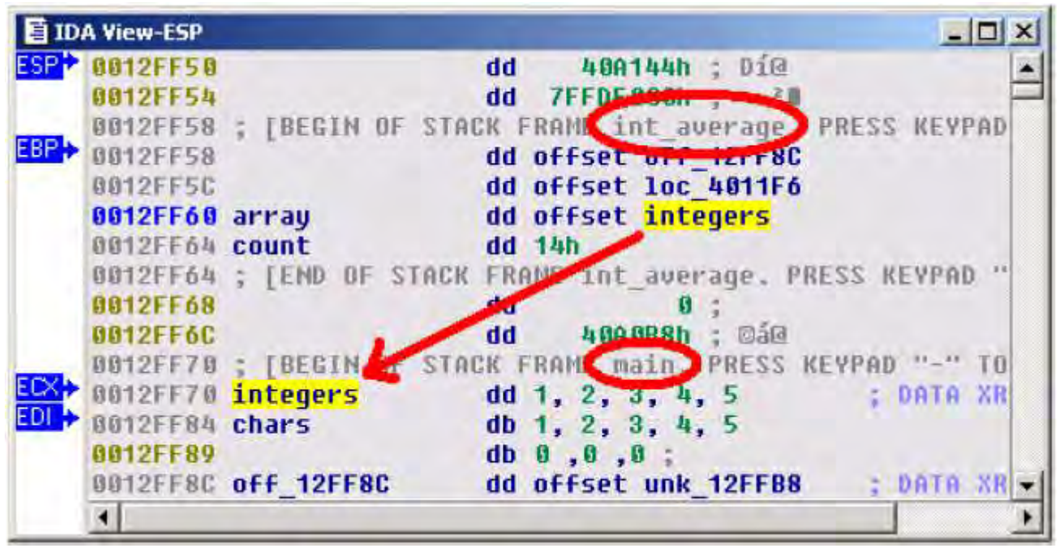
程序运行

现在我们开始执行这个程序。首先使用相应的图标打开调试器窗口，然后运行这个文件直到它到达断点，我们可以使用快捷键 F9，或使用调试工具条上的 *Start* 按钮。



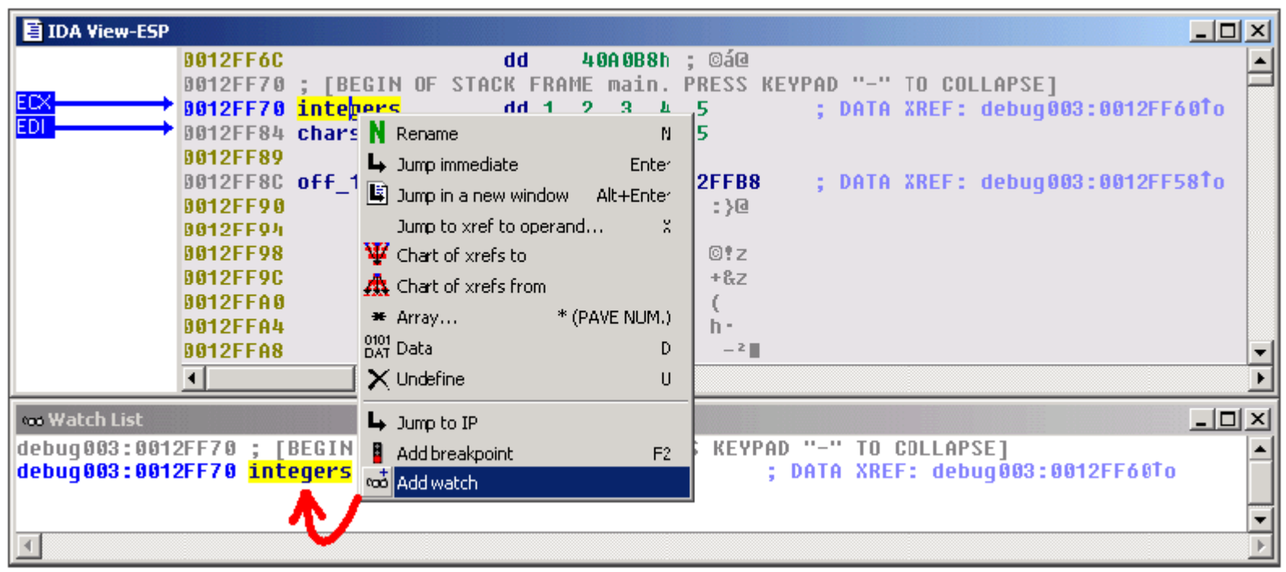
堆栈

IDA-ESP 窗口现在列出了我们感兴趣的函数的堆栈结构，我们很容易就找到了函数 *int_average（）* 的整数数组参数，它指向了在 *main()*的整数数组 *integers[]*。



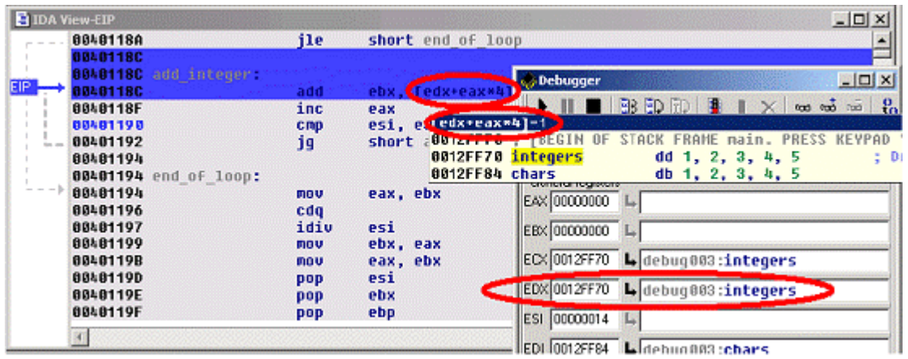
监视

我们可以添加监视，通过监视观察在程序执行过程中数据的变化



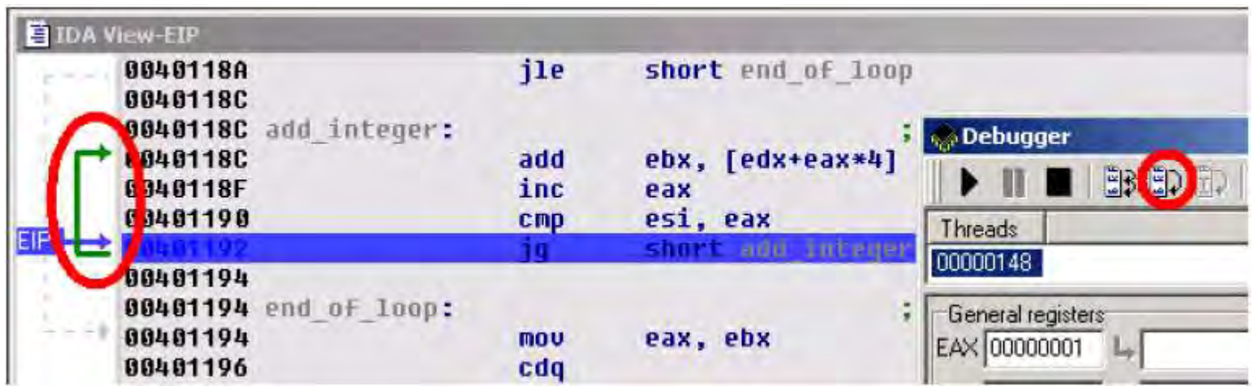
地址分析

通过分析汇编代码，我们可以知道计算累加和的循环所在，它将计算结果保存在 EBX 中，[EBX+EAX*4]明显的指明 EBX 是数组的开始地址，EAX 是数组的索引，这样它就可以定位到数组中的每一项。



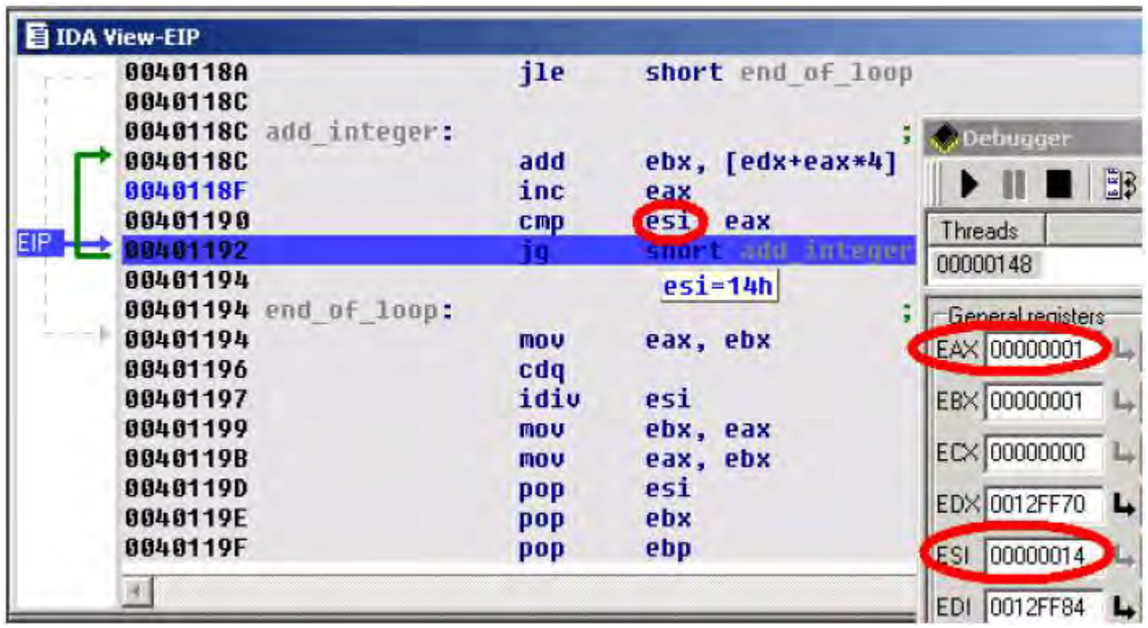
单步跟踪

让我们继续一步一步的跟踪这个循环。通过使用调试工具条上的按钮或用快捷键 F8，有时，IDA 会用绿箭头指示调转命令的调转方向和位置



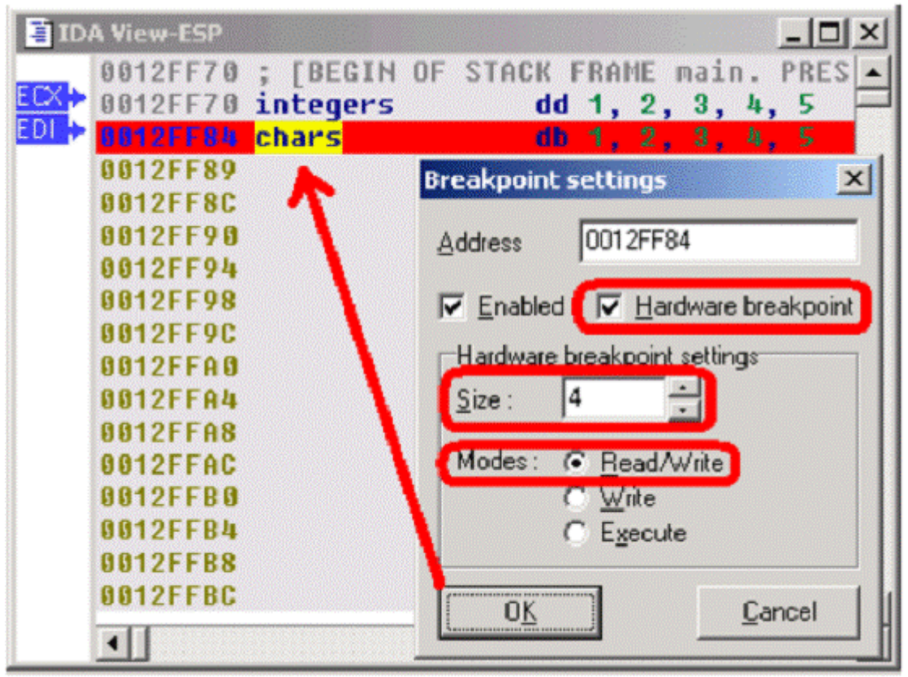
找到 Bug

现在让我们看一下 ESI 值，每轮循环时，EAX 都要与 ESI 比较，我们可以判别 ESI 是循环总轮数，但我们也注意到 ESI 中有一个奇怪的数据 14h(=20)，而我们的数组只有 5 个成员。我们已经找到问题所在了……

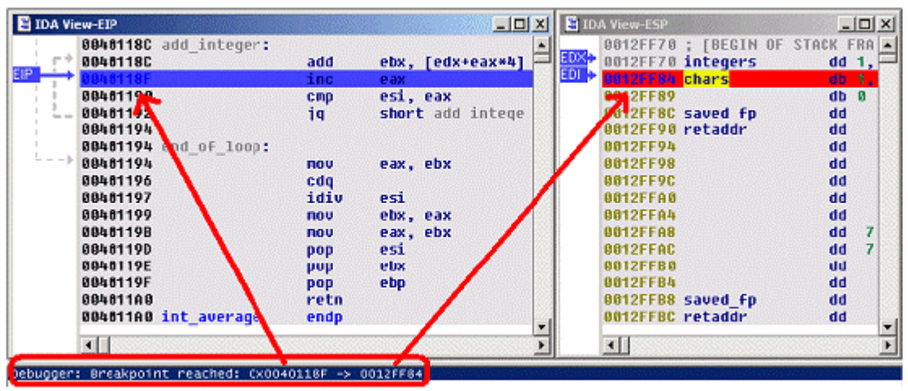


硬件断点

为了确认这个问题，我们使用硬件断点，就设在我们 *intergers* 这个数组最后一个成员的后面（事实上，就是在 *chars* 这个数组的第一个成员上）。如果我们在实行 *intergers* 的循环断到了这里，说明我们的循环读取了 *intergers* 这个数组之外的数据。我们在这里下硬件的 4 字节（一个整数的大小）访问断点。

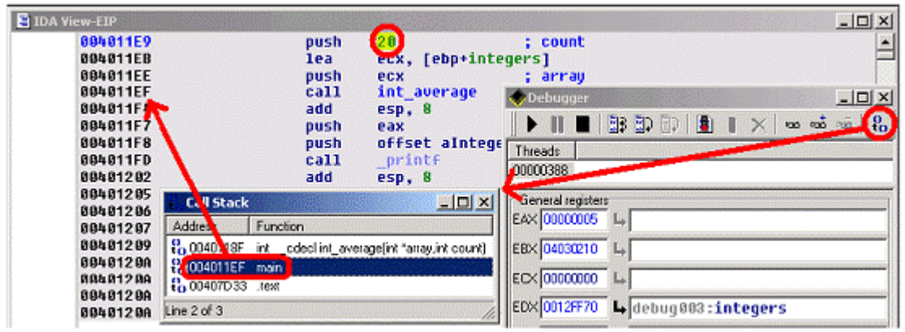


正如我们猜测的，当我们继续运行，它确实执行了对 *chars* 数组的访问操作。注意，此时 EIP 执行导致硬件中断的后一条指令上。



堆栈跟踪

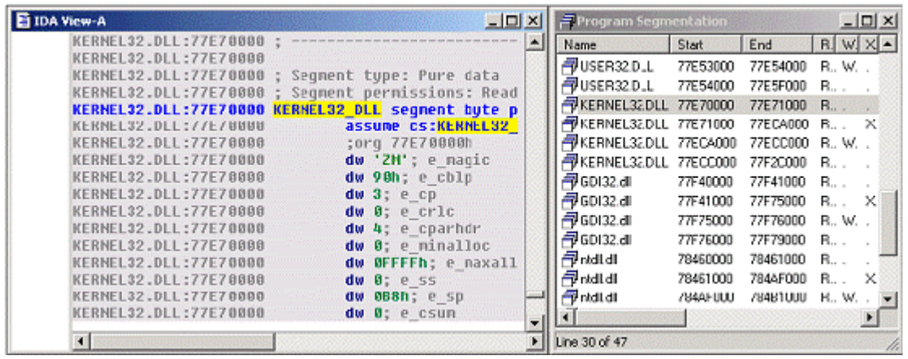
让我们看一下 ESI 来自有调用 *int_average()*时传递来的参数，为什么调用的程序会传递过来这么奇怪的一个参数？我们看一下，所有调用函数的堆栈。我们单击一下 *main()*，来到调用程序这里。通过 IDA 的 *PIT*(参数识别技术)，我们看到 push 20 这条指令，是它将这个错误的参数传递到 *int_average()*。



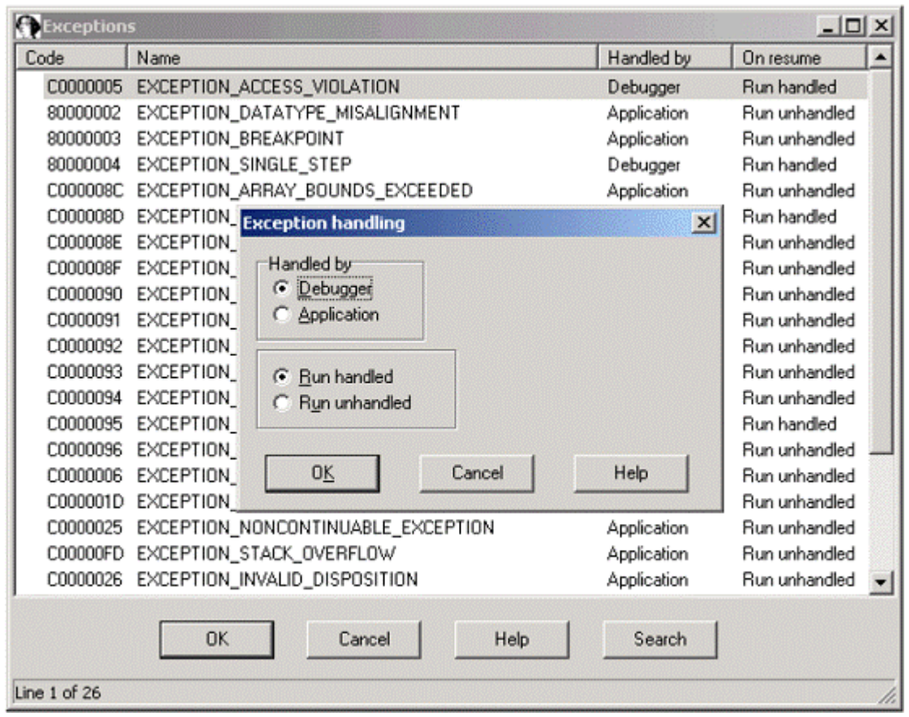
我们回过来看看我们的 C 代码，我们就明白了问题的所在，我们使用了 *sizeof* 这个函数，它返回了数组的字节数，而不是数组的成员数！

其它功能

IDA 可以帮助你进入你所调试的进程内存空间的所有区段，在这些段内，你可以使用调试器的所有功能：可以在 DLL 中设置断点，可以绘制流程图，可以应用结构的数据类型等等。



对异常的处理也完全可以由使用者来决定。



有了调试器功能的 IDA 更加完美，你可以使用它的反汇编和调试器功能随时随地的进行静态分析和动态调试。

http://blog.csdn.net/eqera/article/details/8235843

使用 IDA 的跟踪功能

一个小 bug 程序

这个小程序只是简单的计算了一下一组数据（1,2,3,4,5）的平均值。这组数据被保存在两个数组里，一个是 8bit 的数值，一个是 32bit 数值表示。

```
#include <stdio.h>

charchar_average(char array[],int count)

{

int i;
```

```
char average;

average = 0;

for (i = 0; i < count; i++)

average += array[i];

average /= count;

return average;

}


int int_average(int array[], int count)

{

int i,average;

average = 0;

for (i = 0; i < count; i++)

average += array[i];

average /= count;

return average;

}


voidmain(void)

{

char chars[]= { 1, 2, 3, 4, 5 };

int integers[]= { 1, 2, 3, 4, 5 };
```

```
printf("chars[] - average = %d\n",
char_average(chars, sizeof(chars)));
printf("integers[] - average = %d\n",
int_average(integers, sizeof(integers)));
}
```

运行它，我们得到如下结果：

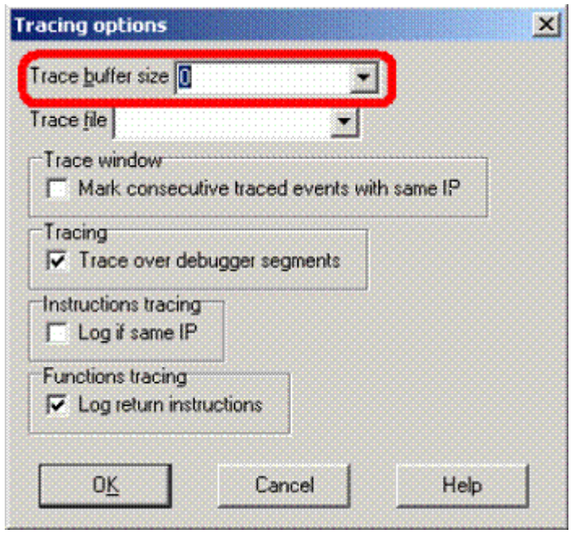
```
chars[]- average = 3
integers[]- average = 1054228
```

可以看出在整数方式下计算的结果是错误的，我们用 IDA 的调试器来分析一下吧。

什么是跟踪

跟踪允许你记录应用程序运行时的变化信息。我们把跟踪信息称为“跟踪事件”。

IDA 记录把跟踪事件记录在一个跟踪缓存区中。跟踪缓存区的大小可以设定为无限大（此时你需要很多内存），或者固定的大小（这种情况下，新的跟踪事件会覆盖老的跟踪事件）。在我们这个例子中，由于我们调试的程序非常小，我们把它设定为无穷大：选择在 *Debugger* 主菜单下的 *Trace* 子菜单中的 *Tracing options*，设定 *Tracing Buffer Size* 为 0。



IDA 提供几种不同的跟踪机制：

1、 指令跟踪：IDA 将会记录每一条指令的执行，并保存寄存器数值，通过使用这些信息，你可以找出应用程序的执行过程，并可找出哪条指令修改了哪个寄存器。

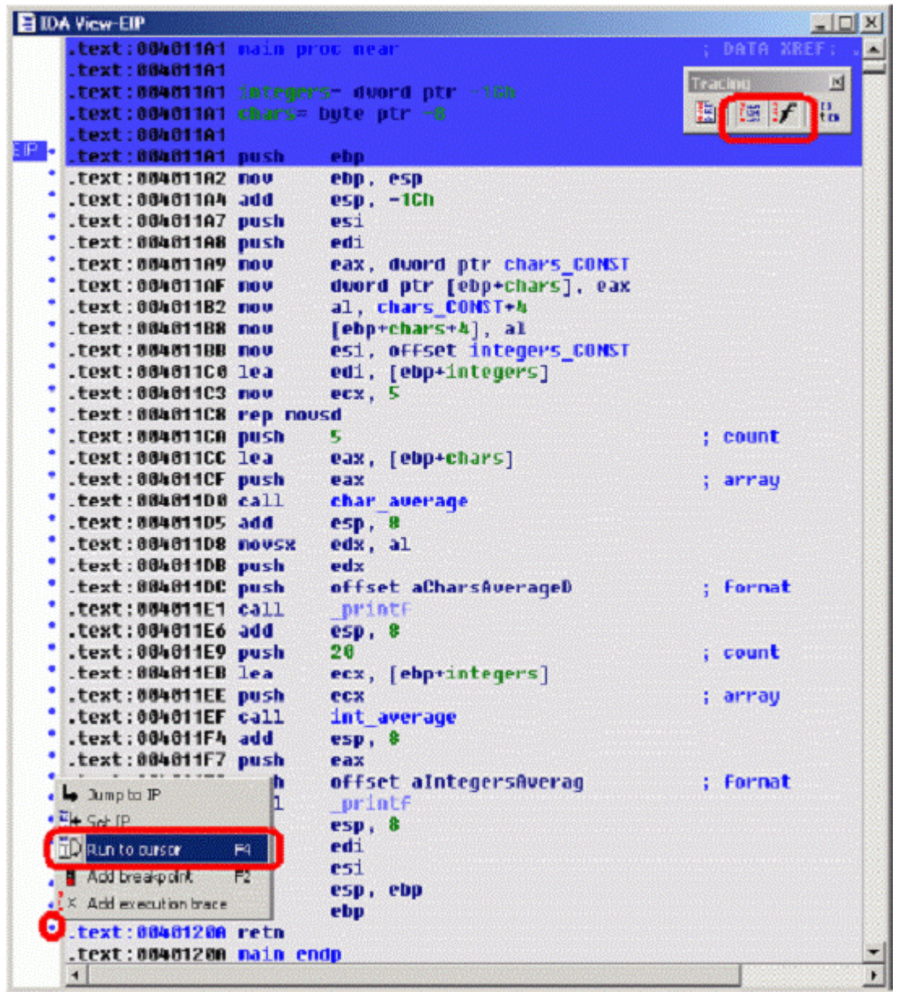
2、 函数跟踪：IDA 将会记录所有的函数调用和函数返回。

3、 读写-写-执行跟踪：IDA 将会记录一个对指定地址的所有访问。这种机制相当于是不停止的断点。

对每种跟踪机制，都会记录相应的跟踪事件到跟踪缓存区，也可以保存到一个 txt 的文件中，同样可以通过 *Tracingoptions* 里的选项来设定。

指令和函数跟踪

为了定位在程序中的 bug 所在，我们要记录所有程序的执行指令，函数调用和函数返回。我们不想记录在 *main()*函数之前的指令。因此我们把光标放置在 *main()*的开始位置（0x4011A1），使用快捷键 F4，开始运行程序并执行到光标位置。我们再通过点按跟踪工具条上的相应图标打开指令和函数跟踪功能，然后我们继续运行程序直到到达 *main()*函数的结尾（0x40120A）。注意 *Runto*（执行到…）命令在调试菜单和鼠标右键菜单中都可选择。



跟踪回溯

IDA 现在已记录了程序的运行。我们点击 *Tracing* 跟踪工具条上 *Tracing Window* 跟踪窗口的按钮，来查看跟踪事件。

如果我们在 *TracingWindow* 跟踪窗口中点中了一条跟踪事件，IDA 更新了屏幕上相应信息，以显示跟踪事件发生时，程序的运行状态。下面是几条我们通常关心的信息：

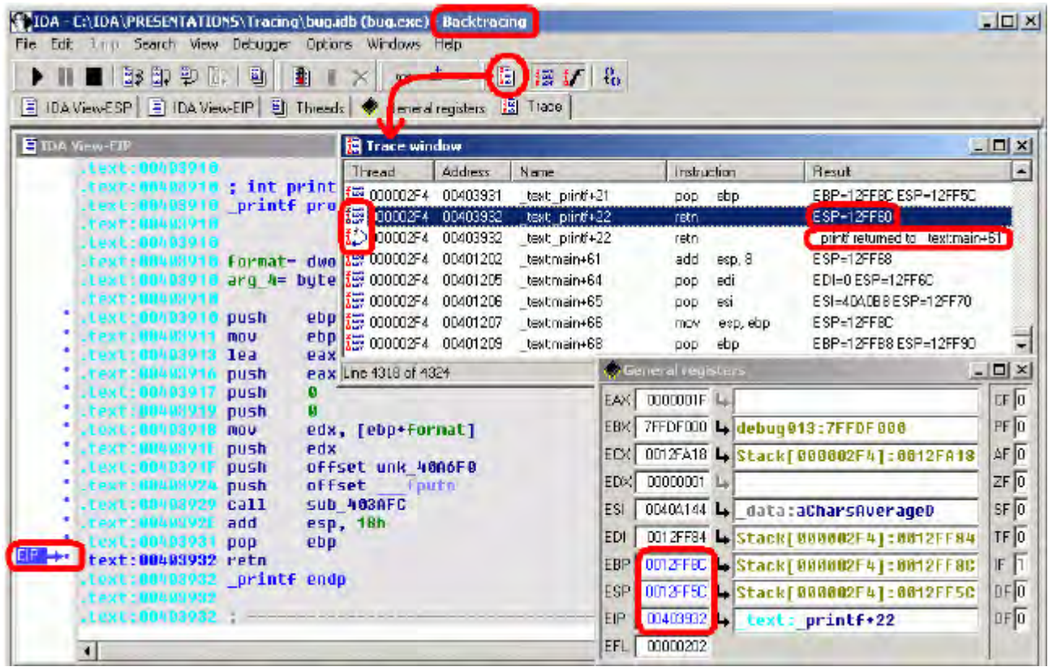
IDA's titlebar(IDA 工具条)：*Backtracing* 代表前面跟踪事件的屏幕信息。

Trace event icons（跟踪事件图标，在跟踪窗口的第一列）：代表记录的跟踪事件类型：指令执行，函数调用，函数返回...例如,我们在*_printf()*函数的结尾，我们看两个不同的跟踪事件，一个代表指令，一个代表函数返回。

Result column（结果列）：将包含跟踪事件指定的信息，当是指令事件的时候，它显示被修改的寄存器，注意 IP 寄存器，由于会被所有指令修改，因此不会显示 *IP* 寄存器。

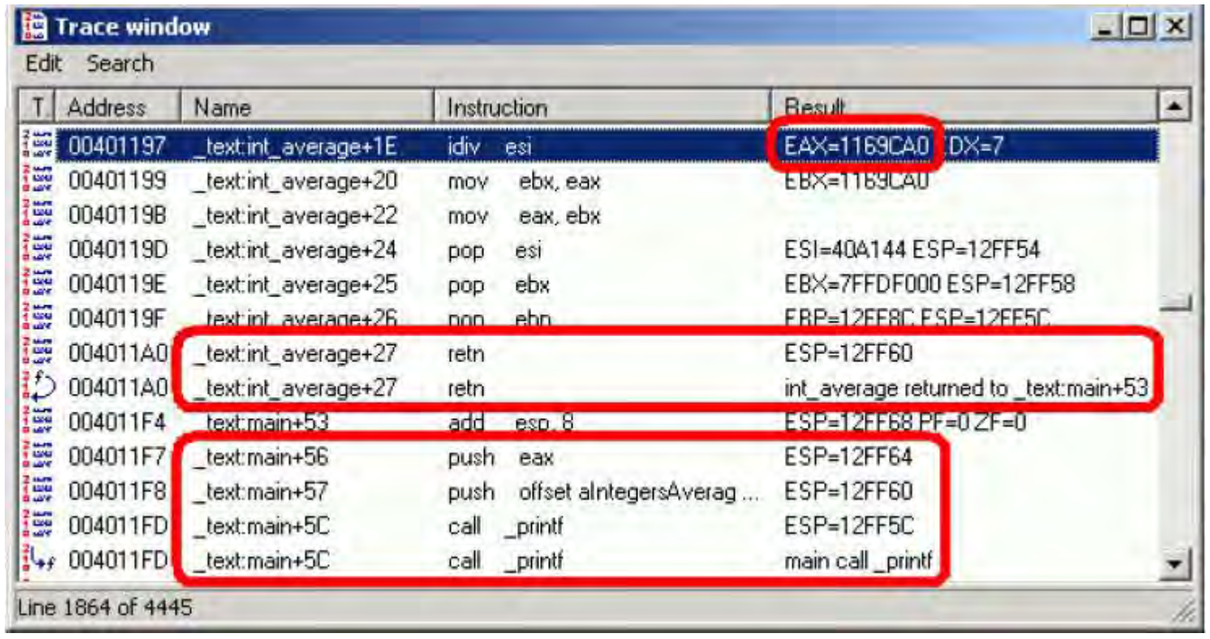
Register arrows（寄存器箭头指示，在反汇编窗口中）：反应在指令执行之前的寄存器数值。

Registers windows（寄存器窗口）：对指令跟踪事件，每个寄存器窗口会显示指令执行前的寄存器数值，刚被更改的寄存器也会用颜色标记出来。



跟踪缓存区内搜索

还记得那个奇怪的平均值结果吗？它是 18259104 (0x1169CA0)，在 Search 搜索菜单中的 *Trace window'sSearch*（跟踪窗口搜索）命令，会至少找到一条与这个数值相关的跟踪事件。事实上，如果我们启动搜索，会发现这个数值在 *int_average+1E* 这个地址：



通过观察跟踪，我们试着看一下，程序是如何使用这个值的。

在 *int_average+1E*：我们发现一条 *"idivesi"* 指令，它的 EAX 中包含这个数值。

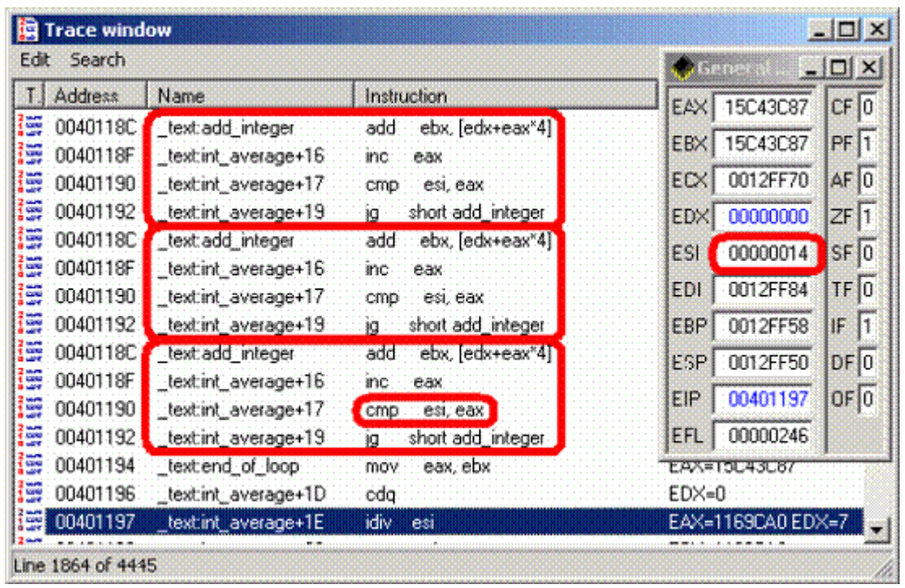
在 *int_average+27*：我们发现 *int_average()* 返回到它的调用程序 *main()*，这个数值是 *int_average()* 的返回值。

在 *main+5c*：这个数值被 *printf()* 打印到屏幕上。

这是一个错误的计算结果！

错误的循环

现在我们在看一下在 *idivesi* 指令之前的指令跟踪情况。

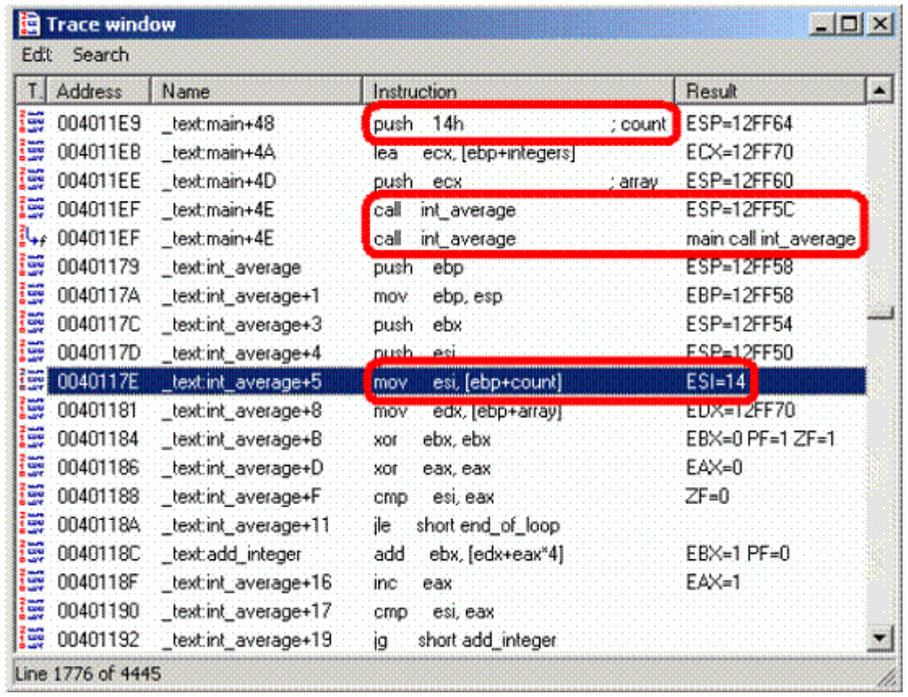


我们可以观察到一个小的循环来累加我们要计算的数据，跟踪窗口内显示了 3 个最新的循环。我们看一下循环中止的条件，它是比较 ESI 和 EAX 中的数值，那么在 ESI 中是什么呢？

寄存器窗口给出了这个数值，它显示 ESI=0X14(20)，我们不是希望它执行 5 次吗？怎么是 20 次？

Bug 修复

我们再浏览一下这个问题循环之前的指令，以查找 ESI 中的这个奇怪的数值从哪里来。



在第一个循环之前，*int_average+F*，我们看到 ESI 从 *int_average()*的 *count* 参数中获得。我们通过 IDA 的参数识别跟踪技术 *（PIT）*，很容易定位到 *PUSH 14h* 这条指令，它将错误的参数传递了进来。现在我们在仔细看一下我们的 C 源程序，就能找到这个错误了：我们使用了 *sizeof()*这个函数，它返回的是数组中的字节数，而不是数组的成员数。

http://blog.csdn.net/eqera/article/details/8237916

使用 IDA 进行远程调试

从 4.8 版开始，IDA PRO 支持通过 TCP/IP 网络对 x86/AMD64 Windows PE 应用程序和 Linux ELF 应用程序进行远程调试。所谓“远程调试”是指通过网络调试在另一个网络上的计算机运行的代码的过程：

1 运行 IDA PRO 界面的计算机被称为“调试器客户端”。

1 运行被调试的应用程序的计算机被成为“调试器服务器 ”。

远程调试主要用于下面一些特殊应用：

1 用于调试病毒/木马/恶意软件：通过这种方法，调试器客户端可以与可能受到这些软件攻击的计算机隔离。

1 调试那些在一台计算机上运行遇到问题，而且没有被拷贝安装在其他计算机上的应用程序。

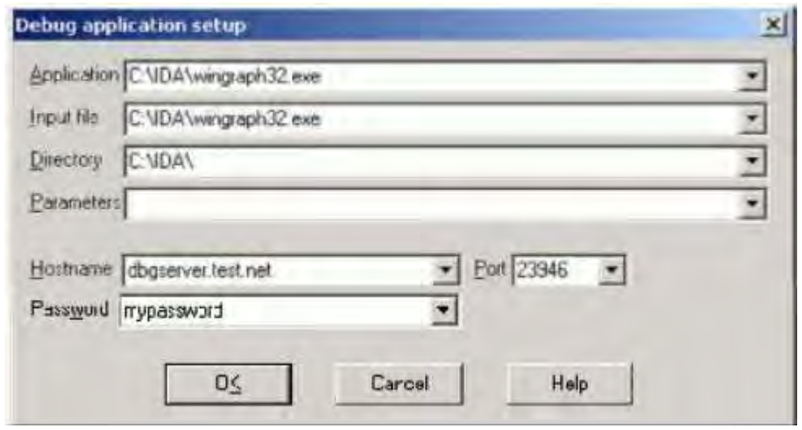
l 调式分布式应用程序
l 始终在你的主工作站上运行，因此你无需将 IDA 配置、文件和不同的调试相关资源
拷贝到其它机器上。
l 以后，将可以在更多的操作系统和结构下调试应用程序。
这个小教程将会讲述如何在实际应用中配置和使用远程调试。

远程 IDA 调试器服务端
为了让 IDA 客户端和调试器服务端可以通过网络进行通信，我们必须首先启动一个小的服务端，它会处理所有低级操作和调试器操作。IDA 的软件包中包括一个 Windows 调试器服务端（win32_remote.exe 文件）和一个 Linux 调试器服务端（linux_server.exe 文件），通过这两个文件，我们可以：
l 在 IDA 窗口中本地调试 x86/AMD64 的 Windows 应用程序和 DLL 文件。
l 在 IDA 窗口中远程调试 x86/AMD64 的 Windows 应用程序和 DLL 文件。
l 在 IDA 窗口中本地调试 x86 的 Linux 应用程序和共享库文件。
l 在 IDA 窗口中远程调试 x86 的 Linux 应用程序和共享库文件。

我们先拷贝这个小的 windows 调试器服务端文件到我们的调试器服务器上。
服务端可以接收下面几种命令行参数：
```
C:\> win32_remote -?
IDA Windows32 remote debugger server. Version 1.0. Copyright Datarescue 2004
Error: usage: ida_remote [switches]
-p... port number
-P... password
-v         verbose
```

我们可以设定一个密码，以阻止那些未授权的链接：
```
C:\>win32_remote -Pmypassword

IDA Windows32 remote debugger server. Version 1.0. Copyright Datarescue 2004
Listening to port #23946...
```
注意远程调试器服务器同时只能处理一个调试会话。如果你需要在一个相同的主机上同时调试几个应用程序，则需要使用 -p 开关在不同的端口启动多个服务端。

设置调试器客户端
首先，把我们想要在调试器服务端（Windows 或 Linux）调试的可执行文件拷贝到调试器客户端（Windows 或 Linux）。我们可以像通常那样把这个文件装入 IDA。通过点击在 Debugger 菜单中的“Process options…”菜单条，来设置远程调试



在上面的窗口中我们设定 Application，Directory，和 Input file 路径。注意这些文件路径应该在远程调试器服务端上有效，同时不要忘记输入调试器服务端的主机名字和 IP 地址：远程调试只有当这些设置完成后才有效！最后，输入我们在 IDA 的调试器服务端设定的密码。

启动远程调试

现在，调试器服务端和调试器客户端均已设置完成，可以开始远程调试了。事实上，你现在可以使用本地调试可以用到的所有调试命令。例如，我们可以跳到程序入口点，然后使用 F4 快捷键，运行进程到程序的入口点。

如果我们现在直接终止进程（通过 CTRL-F2）并查看 win32_remote’s 的输出(在调试器服务端)，我们看到它确实接受了这个命令并关闭了我们的网络链接：

```
C:\> win32_remote -Pmypassword
IDA Windows32 remote debugger server. Version 1.0. Copyright Datarescue 2004
Listening to port #23946...
```

```
Accepting incoming connection...
Closing incoming connection...
```

附加到一个正在运行的进程

另一有趣的功能的是，可以附加调试器到一个在远程计算机上已经存在并运行着的进程上。如果你在 Debugger 菜单中点击“Attach to process…”命令，IDA 将会显示所有与你的反汇编数据库中的文件对应的远程运行进程的列表：

在列表中双击一个进程，将会自动挂起并附加调试器到这个进程上，这可以让你无需手动启动这个进程就可以调试它。附加进程的功能可以工作在 Windows 到 Linux，Windows 到 Windows， Linux 到 Linux 和从 Linux 到 Windows 多种情况下。

从调试进程中解除附加

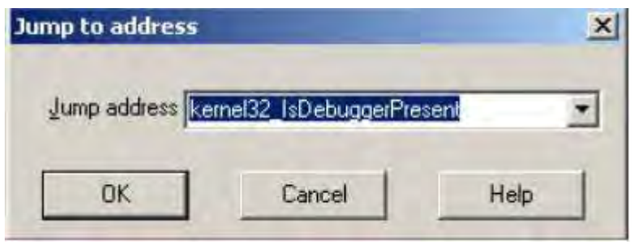
最后，如果如果调试器服务端运行在 Windows XP,Windows Server 2003 或 Linux，你也可以从你当前调试的进程中解除附加，只需要使用 Debugger 菜单中的“Detach from process”命令即可。

IDA 支持调试 Windows 下的 DLLs 和 Linux 下的共享库文件。在 Windows 下， IDA 也可以附加到 Windows 服务上，无论该服务是运行在本地或远端。尤其是当你附加到 Windows 服务上后，“Detach from process”命令非常有用：这可以使你无需终止调试器服务端的关键 Windows 服务就可以停止调试器！

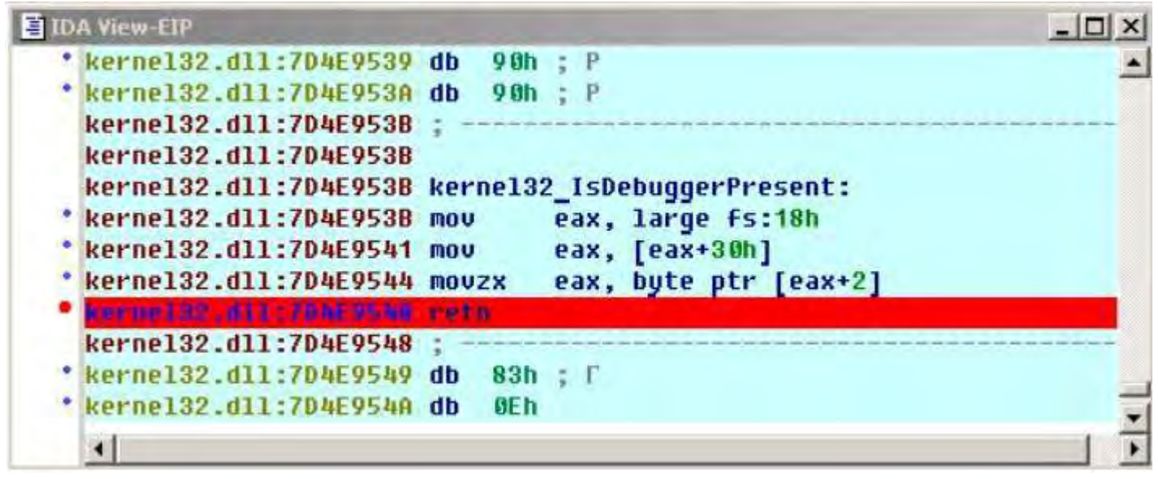
http://blog.csdn.net/eqera/article/details/8239622

IDA 教程-隐藏 IDA 调试器

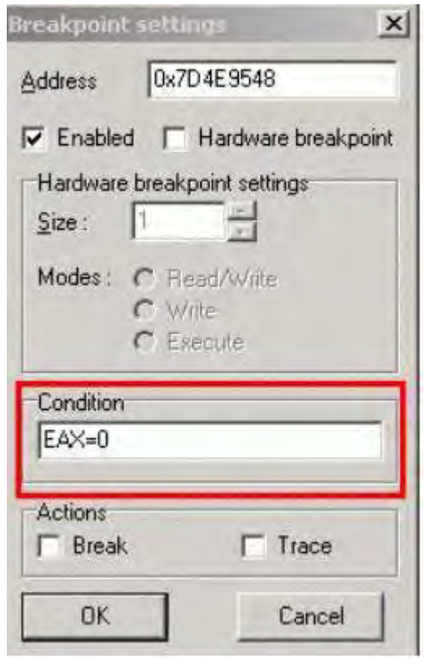
一个隐藏 IDA 调试器的小技巧
很多 IDA 使用者都向我索要一个可以隐藏调试器的插件或功能。事实上，有很多反调试的手段，而针对每一种反调试的手段都需要对应一种处理方法。我们先从一些简单的开始，我们的目的是让 IsDebuggerPresent 函数始终返回 0 值。
当一个调试器处于运行状态时，我们可以在反汇编窗口使用”go to the specified address”命令进入 IsDebuggerPresent 函数。不幸的是，当前版本的 IDA（译者注：作者文章写于 2005 年）不能在名称列表中显示输入函数名字，因此我们需要在输入框中手动输入函数名字：



请注意我是如何来构造这个地址的：在 Dll 名字后面有一个下划线，之后再加上函数名。
我们在这个函数的结尾放置一个断点。这样我们就可以拦截它并修改它的返回值：



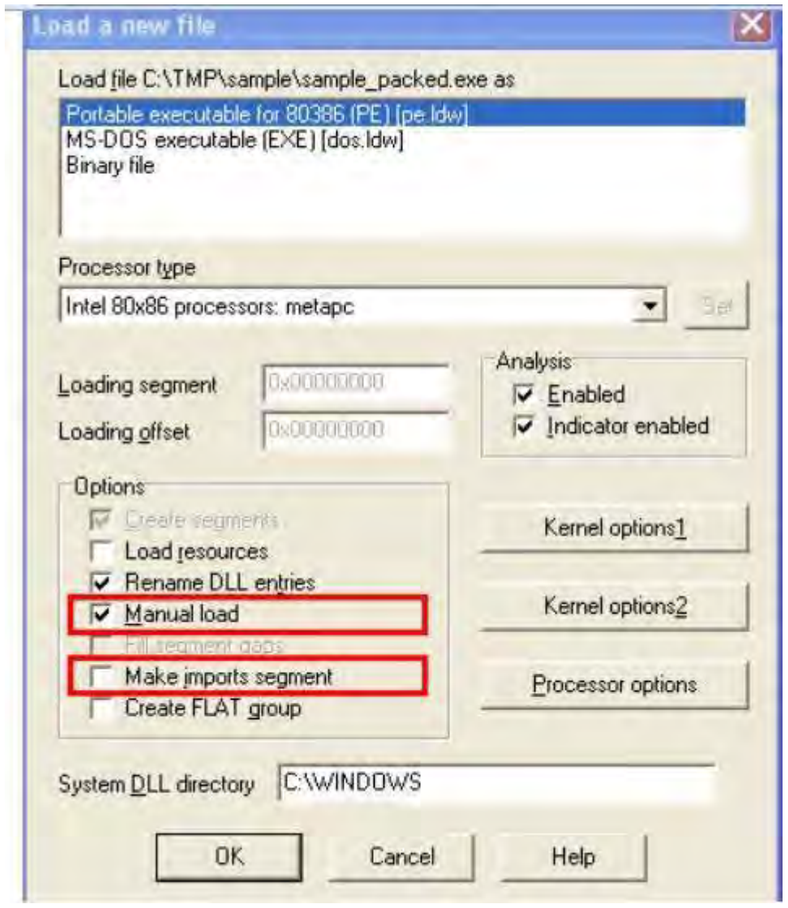
我们并不希望每次这个 IsDebuggerPresent 函数被调用时挂起程序，然后每次手动修改这个数值，这很麻烦，我们希望它能自动完成这个操作。方法是：我可以使用条件断点，“断点条件”通常用来确定一个断点应该被触发或抛弃，其语法为 IDC 表达式，如果表达式等效为 0，断点就不会触发。IDA 需要执行这个表达式，以判断其结果，因此我们可以利用它的执行结果，修改寄存器或内存为任何你需要的数值。我们编辑断点的属性如下（右击鼠标，编辑断点）：



我们设定条件为” EAX=0”。注意这里不是一个比较，而是一个赋值。当 IDA 执行它的时候，EAX 会被附带设置为 0，这正是我们需要的。因为我们不想挂起程序，我们也清除了这个断点的属性。以这种方式断点，就可以让我们的调试器对付 IsDebuggerPresent 的调用了。这听起来很简单，你或许会问“如果壳作者使用的不是这种幼稚的反调试技巧该如何应付呢？”，请不要着急，我们以后会继续这个话题。

一个隐藏调试器的插件

上次我提到了一个使用条件断点的小技巧，今天我们会给大家提供一个可以自动处理这些断点的插件，以便可以使用 IDA 解压一些象“Zotob 蠕虫”这样的恶意软件。
如果测试一个“活”的恶意软件很危险，因此我们将在演示中使用一个示例程序。Zotob 使用 Yoda’s Protector 的一个变种来加壳（可以通过这个连接来认识它）。我们将会使用这个示例程序，并使用 Yoda’s Protector 来保护它。首先我们尝试对它脱壳。
IDA 给出了很多关于压缩了的可执行文件的提示并尝试装载它。它找到了这个压缩器的入口，但通常来说，当我们处理一个恶意软件的时候，推荐打开 manual load 选项，并关闭 make imports section 选项。Manual load 选项可以将输入文件的所有区段都装载到数据库中（恶意软件通常将他们的代码隐藏在文件的任何地方）。取消 Creating the imports seciton，则可以是 IDA 以原始方式显示整个输入表目录的内容，谁说恶意软件不能在输入表目录中隐藏呢？



接着我们遇到的问题是，当我们在调试器中试图跟踪壳时，会有非常多的异常。你必须非常小心的处理这些伪调用和异常。一个不留神我们就会发现已不知身在何处，程序跑飞，崩溃甚至关闭调试器。它是壳有意这样做的-壳作者喜欢将一些事情复杂化以使其“几乎”无法被分析。如果一个程序可以在不需要特殊的 key 或额外数据的情况下运行，那么我们可以让它运行在一个虚拟环境下，并完整的分析它。今天我们不会虚拟化整个环境，只是其中一小部分-我们将会处理 Windows API 以让壳无法使用他们。

好，回到我们的程序。壳在解压进程中使用 SEH(结构化异常处理)，IDA 不停的报告着每一次异常。这可能是成百上千次。默认情况下，IDA 和多数其它调试器有着类似的设置：一旦发生异常，它就会挂起程序。通常在这种设置适用于调试“正常”程序，但对我们跟踪恶意软件却没有任何帮助。

我们修改一下这些异常处理的设置，以让 IDA 不在每次异常时都停下来。你可以通过用户界面（Debugger,Debugger options,Edit exceptions）或通过编辑 cfg/exceptions.cfg 文件来进行设置。第二种方法更好些，因为设置将会应用到所有以后的调试中，而第一种方法，只

针对当前的程序有效。我们要告诉 IDA 所有的异常都应交给应用程序来处理。下面是这个配置文件中的一行：

view plaincopy

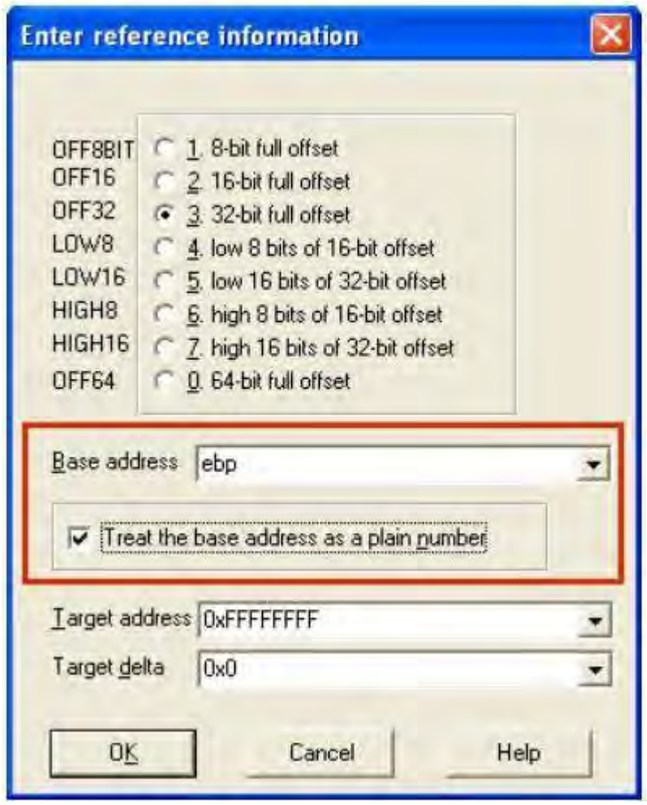
```
1. 0xC0000005  nostop app EXCEPTION_ACCESS_VIOLATION    The instruction at
2. 0x%a referenced memory at 0x%a. The memory could not be %s
```

这行设置表示当该异常发生时，调试器不会停止，应用程序会处理这个异常。如果你有过一些 IDA 的使用经验，你会发现注意到，尽管有这些设置有时 IDA 仍会停止在一个异常处。如果这是一个“Second Changce”的异常，IDA 将会停止在这个异常：一个未被处理的“Second Changce”异常将会中止应用程序，IDA 会给你一个处理它的机会。
你可以使用这个文件替换你的配置文件，方法是打开 Debugger,debugger options 对话框中的 Load 按钮来装载这个配置文件。
使用这个新的配置文件，你可以很容易的单步跟踪它了。你甚至可以在 4766a4 处设置一个断点来查看它的一个小把戏-在 4766a8 处的自修改代码（就在下面的几条指令）。一旦开始执行在 4766a5 处的’STOSB’，你就可以在 4766a6 处按 F8，接着代码就在屏幕上出现了。
我不会细说每一个壳中的小把戏，你可以参看其它现在做的非常好的站点上的介绍（译者注：比如 pediy.com，hehe）。现在，让我们在 476854 处使用一个硬件断点，并返回到程序中。
你会看到壳愉快而又卖力的工作着，并没有发现调试器正在使用 SEH 技巧。
为什么我们要用一个硬件断点，而不是软件断点呢？原因是因为应用程序可以很容易检测到软件断点。例如，在 4775CE 处有一个计算校验和的函数。如果你使用软件断点，这里就会得到一个错误的校验和，随之壳就会在其后的某处崩溃。通常说来，推荐使用硬件断点，但不幸的是，IDA 并没有自动使用它们的选项。我手动在恶意软件中使用硬件断点以让它从开始运行到一个指定位置。错误是不可避免的，但硬件断点可以让我在整个调试过程中重复到达上一个已知地址中。好处是我可以在几天后，甚至重启机器仍能继续调试进程

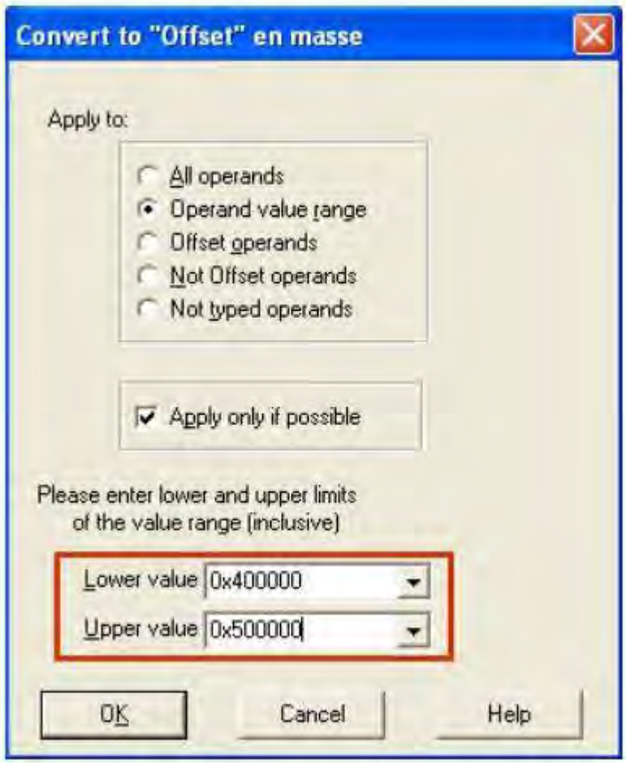
如果你在 476854 处放置一个硬件断点，你会看到下面代码：

```
.yP:0047680B cmp     ebx, 5A4Dh
.yP:00476811 jnz     short loc_4767F2
.yP:00476813 mov     [ebp+42A034h], edx
.yP:00476819
.yP:00476819 loc_476819:
.yP:00476819 mov     eax, [ebp+42A034h]
.yP:0047681F add     eax, [eax+3Ch]
.yP:00476822 add     eax, 80h
.yP:00476827 mov     ecx, [eax]
.yP:00476829 add     ecx, [ebp+42A034h]
.yP:0047682F add     ecx, 10h
.yP:00476832 mov     eax, [ecx]
.yP:00476834 add     eax, [ebp+42A034h]
.yP:0047683A mov     ebx, [eax]
.yP:0047683C mov     [ebp+42722Bh], ebx
.yP:00476842 add     eax, 4
.yP:00476845 mov     ebx, [eax]
.yP:00476847 mov     [ebp+42722Fh], ebx
.yP:0047684D lea     eax, [ebp+426E46h]
.yP:00476853 push    eax
.yP:00476854 mov     ebx, 42722Bh
.yP:00476859 call    dword ptr [ebp+ebx+0]
.yP:0047685D call    sub_476865
.yP:00476862 jmp     short sub_476865
.yP:00476862 sub_4767DC endp
```

我们面对的是一个间接调用。当前我们处于调试器中，因此我们可以很容易找到这个被调用函数的地址，但是显示出来的却很丑陋。让我们使用 IDA 命令来修改它，壳使用许多基于 EBP 寄存器的参考。很明显 EBP 寄存器并不变化。我们选择整个屏幕，使用“user-defined offset”命令：



偏移是 EBP，它是个简单的数字（并没有保存一个地址）。因为我们选择了一个区域，因此它还有一个对话框需要设置：



我们让它转换在 400000..500000 范围内所有地址为偏移。结果比开始的要好多了：

```
.yP:0047680B cmp     ebx, 5A4Dh
.yP:00476811 jnz     short loc_4767F2
.yP:00476813 mov     ss:(ImageBase - 4F21Ah)[ebp], edx
.yP:00476819
.yP:00476819 loc_476819:                             ; CODE
.yP:00476819 mov     eax, ss:(ImageBase - 4F21Ah)[ebp]
.yP:0047681F add     eax, [eax+3Ch]
.yP:00476822 add     eax, 80h
.yP:00476827 mov     ecx, [eax]
.yP:00476829 add     ecx, ss:(ImageBase - 4F21Ah)[ebp]
.yP:0047682F add     ecx, 10h
.yP:00476832 mov     eax, [ecx]
.yP:00476834 add     eax, ss:(ImageBase - 4F21Ah)[ebp]
.yP:0047683A mov     ebx, [eax]
.yP:0047683C mov     ss:(LoadLibraryA_0 - 4F21Ah)[ebp], ebx
.yP:00476842 add     eax, 4
.yP:00476845 mov     ebx, [eax]
.yP:00476847 mov     ss:(GetProcAddress_0 - 4F21Ah)[ebp], el
.yP:0047684D lea     eax, (dword_476060 - 4F21Ah)[ebp]
.yP:00476853 push    eax
.yP:00476854 mov     ebx, (offset LoadLibraryA_0 - 4F21Ah)
.yP:00476859 call    dword ptr [ebp+ebx+0]
.yP:0047685D call    sub_476865
.yP:00476862 jmp     short sub_476865
.yP:00476862 sub_4767DC endp
```

我们看到壳用到 LoadLibrary 这个函数来访问 Windows API 函数。它会检索许多函数的地址并创建导入表。如果你让程序运行到 476e77(你可以使用硬件断点)，你将会在 476451 处看到导入表：

```
.yP:00476451 dd offset kernel32_GetModuleHandleA
.yP:00476455 dd offset kernel32_VirtualProtect
.yP:00476459 dd 0CCCCCCCCh
.yP:0047645D dd offset kernel32_GetModuleFileNameA
.yP:00476461 dd offset kernel32_CreateFileA
.yP:00476465 dd offset kernel32_GlobalAlloc
.yP:00476469 dd offset kernel32_GlobalFree
.yP:0047646D dd offset kernel32_ReadFile
.yP:00476471 dd offset kernel32_GetFileSize
.yP:00476475 dd offset kernel32_CloseHandle
.yP:00476479 dd offset kernel32_IsDebuggerPresent
.yP:0047647D dd offset kernel32_CreateToolhelp32Snapshot
.yP:00476481 dd offset kernel32_GetCurrentProcess
.yP:00476485 dd offset kernel32_GetCurrentProcessId
.yP:00476489 dd offset kernel32_Process32First
.yP:0047648D dd offset kernel32_Process32Next
.yP:00476491 dd offset kernel32_Module32First
.yP:00476495 dd offset kernel32_Module32Next
.yP:00476499 dd offset kernel32_Thread32First
.yP:0047649D dd offset kernel32_Thread32Next
.yP:004764A1 dd offset kernel32_OpenThread
.yP:004764A5 dd offset kernel32_OpenProcess
.yP:004764A9 dd offset kernel32_TerminateProcess
.yP:004764AD dd offset kernel32_SetPriorityClass
.yP:004764B1 dd offset kernel32_GetPriorityClass
.yP:004764B5 dd offset kernel32_ExitThread
```

（默认情况下，是看不到这张表的；你需要将鼠标放置在起始位置，创建一个双字，再创建双字数组）。这张表还不是很好，因为它包括一些名字的参考，但这些条目却没有被命名。在导入表的每一条将会一个接着一个的被使用，但一个张没有名字的列表没有什么意义。下面这段脚本，将会修正这张表。使用 F2 进入脚本对话框：

view plaincopy

```
1.  auto ea, name;
2.  for ( ea=here; ea < 0x476545; ea=ea+4 )
3.  {
4.    name = Name(Dword(ea));
5.    name = substr(name, strstr(name, "_")+1, -1);
6.    MakeName(ea, name);
7.  }
```

下面是脚本运行结果：

```
; HMODULE __stdcall LoadLibraryA_0(LPCSTR lpLibFileName)
LoadLibraryA_0 dd offset kernel32_LoadLibraryA ; DATA XREF: sub_47
                                        ; sub_4767DC+78↓o
; FARPROC __stdcall GetProcAddress_0(HMODULE hModule,LPCSTR lpProc
GetProcAddress_0 dd offset kernel32_GetProcAddress ; DATA XREF: su
                                        ; GetProcAddress_1+2↓o
dword_47644D dd 7D4C0000h               ; DATA XREF: sub_476865+2↓
; HMODULE __stdcall GetModuleHandleA(LPCSTR lpModuleName)
GetModuleHandleA dd offset kernel32_GetModuleHandleA
; BOOL __stdcall VirtualProtect(LPVOID lpAddress,DWORD dwSize,DWOR
VirtualProtect dd offset kernel32_VirtualProtect
dd 0CCCCCCCCh
; DWORD __stdcall GetModuleFileNameA(HMODULE hModule,LPSTR lpFilen
GetModuleFileNameA dd offset kernel32_GetModuleFileNameA
; HANDLE __stdcall CreateFileA(LPCSTR lpFileName,DWORD dwDesiredAc
CreateFileA dd offset kernel32_CreateFileA
; HGLOBAL __stdcall GlobalAlloc(UINT uFlags,DWORD dwBytes)
GlobalAlloc dd offset kernel32_GlobalAlloc
; HGLOBAL __stdcall GlobalFree(HGLOBAL hMem)
GlobalFree dd offset kernel32_GlobalFree
; BOOL __stdcall ReadFile(HANDLE hFile,LPVOID lpBuffer,DWORD nNumb
ReadFile dd offset kernel32_ReadFile
; DWORD __stdcall GetFileSize(HANDLE hFile,LPDWORD lpFileSizeHigh)
GetFileSize dd offset kernel32_GetFileSize
; BOOL __stdcall CloseHandle(HANDLE hObject)
CloseHandle dd offset kernel32_CloseHandle
; BOOL IsDebuggerPresent(void)
IsDebuggerPresent dd offset kernel32_IsDebuggerPresent
```

查看这张表，我们会发现很多邪恶函数。著名的 IsDebuggerPresent 也在其中，还有一些如 SuspendThread, TerminateProcess, BlockInput，看上去也不像是好东西。
下面是我们的对策：我们在每一个危险函数上设置条件断点如下：
(EIP=address_of_ret_instruction) && (EAX=return_value)
例如，在 BlockInput 中：

```
user32.dll:7D9A0586 user32_BlockInput proc near
user32.dll:7D9A0586
user32.dll:7D9A0586 arg_0= dword ptr  4
user32.dll:7D9A0586
user32.dll:7D9A0586 mov     eax, 1224h
user32.dll:7D9A058B lea     edx, [esp+arg_0]
user32.dll:7D9A058F mov     ecx, 0
user32.dll:7D9A0594 call    large dword ptr fs:0C0h
user32.dll:7D9A059B retn    4
user32.dll:7D9A059B user32_BlockInput endp
```

条件断点应是：
(EIP=0x7D9A059B) && (EAX=0x1)
断点会跳过函数的执行并返回了一个我们预设的结果。它不会挂起运行的程序。壳将没有机会阻止用户的输入，也无法探测调试器和终止它。即使它试图这样，也会失败。
如果每次在运行应用程序时都设置这些断点是很乏味的事情，因此我写了个插件。它非常简单，我提供了源代码。通过这个插件，可以很容易让应用程序运行且无法检测到调试器：只需激活插件，然后运行调试器即可。应用程序将会自加压，然后没有任何疑虑的运行。

http://blog.csdn.net/eqera/article/details/8239569

IDA 教程-脚本化的调试器

从 2003 年开始，IDA PRO 开始支持调试器功能，这很好的弥补了静态分析的不足。但在很多情况下，它的功能仍然弱于动态分析。IDA PRO 的调试器目前支持 32 位和 64 位下 MS Windows 可执行程序，它支持在 Windows,Linux,Mac OS x 下进行本地和远程调试。然而，因为调试器的 API 需要熟悉我们的 SDK 和使用基于事件的模型，这对我们有的用户来说有些困难和并不容易操作。

- 由于 API 使用了基于事件的模型，这使得它很难用我们常用的方法来设计一个顺序执行（线性模型）的程序。使用者被强制设计一个事件句柄，来实现一个有限状态机（插件的核心逻辑）。很多时候这是个强大的方式，但可能对某些简单的任务来说又过于复杂了。
- 因为 API 只能在插件层使用，一个简单的调试器操作需要写一个插件，这比写一个简单的 IDC 脚本需要花费更多的时间和精力。

IDA 5.2 针对这两个问题做了改进。以往的基于事件的模型依然可用，同时也可通过使用 get_debugger_event()这个函数来支持简单的线性模型。这个函数暂停插件（或脚本）的执行直到一个新的调试器时间发生。使用者可以指定他是只对进程挂起事件感兴趣或是对所有事件。也可以进行定时设置，如果没有其它事件发生，超时后程序可以继续执行。

新功能允许我们抛弃以往的事件模型（除了在那些事件逻辑优先线性逻辑的那些情况），通过编写 IDC 脚本来控制调试器。例如，启动调试器，运行到指定位置，输出一些数据和两次单步运行，如下所示意：

```
AppBpt(some_address);
StartDebugger("","","");          // start debugger with default params
GetDebuggerEvent(WFNE_SUSP, -1); // ... and wait for bpt
Message ("Stopped at %a, event code is %x\n", GetEventEA(), GetEventId());
StepInto();                       // request a single step
GetDebuggerEvent(WFNE_SUSP, -1); // ... and wait for app to execute
StepInto();                        // request a single step
```

```
    GetDebuggerEvent(WFNE_SUSP, -1); // ... and wait for app to execute
```

在 IDA 5.1 中这需要使用一个事件句柄来处理一个小的有限状态机自动执行，一共需要超过 200 行的代码。请注意，在上面的例子中，为了清晰，错误处理代码被忽略。在实际生活中，你应该需要检查一些不希望的情况例如单步 StepInto()之后的发生异常的情况。

为了演示使用新的方式来编写脚本是如何简单，我们重写了 UUNP 解压器插件的核心功能。原程序需要大概 600 行代码，并有一个复杂的逻辑。新的脚本只需要 100 行的代码（其中几乎有一半是注释和空行）。更重要的是，脚本易懂并可根据你的需要随意修改。它展示了 IDA5.2 的新调试器功能的使用。

```
#include <idc.idc>

//--------------------------------------------------------------------------

static main()

{

  auto ea, bptea, tea1, tea2, code, minea, maxea;

  auto r_esp, r_eip, caller, funcname;


  // Calculate the target IP range. It is the first segment.

  // As soon as the EIP register points to this range, we assume that

  // the unpacker has finished its work.

  tea1 = FirstSeg();
```

```
tea2 = SegEnd(tea1);

// Calculate the current module boundaries. Any calls to GetProcAddress
// outside of these boundaries will be ignored.
minea = MinEA();
maxea = MaxEA();

// Launch the debugger and run until the entry point
if ( !RunTo(BeginEA()) )
  return Failed(-1);

// Wait for the process to stop at the entry point
code = GetDebuggerEvent(WFNE_SUSP, -1);
if ( code <= 0 )
  return Failed(code);

// Set a breakpoint at GetProcAddress
bptea = LocByName("kernel32_GetProcAddress");
if ( bptea == BADADDR )
  return Warning("Could not locate GetProcAddress");
AddBpt(bptea);
```

```
while ( 1 )
{
  // resume the execution and wait until the unpacker calls GetProcAddress
  code = GetDebuggerEvent(WFNE_SUSP|WFNE_CONT, -1);
  if ( code <= 0 )
    return Failed(code);

  // check the caller, it must be from our module
   r_esp = GetRegValue("ESP");
  caller = Dword(r_esp);

if ( caller < minea || caller >= maxea )
      continue;

    // if the function name passed to GetProcAddress is not in the ignore-list,
    // then switch to the trace mode
    funcname = GetString(Dword(r_esp+8), -1, ASCSTR_C);
    // ignore some api calls because they might be used by the unpacker
    if ( funcname == "VirtualAlloc" )
      continue;
    if ( funcname == "VirtualFree" )
```

```
    continue;

  // A call to GetProcAddress() probably means that the program has been
  // unpacked in the memory and now is setting up its import table
  break;
}

// trace the program in the single step mode until we jump to
// the area with the original entry point.
DelBpt(bptea);
EnableTracing(TRACE_STEP, 1);
for ( code = GetDebuggerEvent(WFNE_ANY|WFNE_CONT, -1); // resume
      code > 0;
      code = GetDebuggerEvent(WFNE_ANY, -1) )
{
  r_eip = GetEventEa();
  if ( r_eip >= tea1 && r_eip < tea2 )
    break;
}
if ( code <= 0 )
  return Failed(code);
```

```
// as soon as the current ip belongs OEP area, suspend the execution and
// inform the user
PauseProcess();
code = GetDebuggerEvent(WFNE_SUSP, -1);
if ( code <= 0 )
  return Failed(code);

EnableTracing(TRACE_STEP, 0);

// Clean up the disassembly so it looks nicer
MakeUnknown(tea1, tea2-tea1, DOUNK_EXPAND|DOUNK_DELNAMES);
  MakeCode(r_eip);
AutoMark2(tea1, tea2, AU_USED);
  AutoMark2(tea1, tea2, AU_FINAL);
  TakeMemorySnapshot(1);
  MakeName(r_eip, "real_start");
  Warning("Successfully traced to the completion of the unpacker code\n"
          "Please rebuild the import table using renimp.idc\n"
          "before stopping the debugger");
}
//-------------------------------------------------------------------------
// Print an failure message
```

```
static Failed(code)
{
  Warning("Failed to unpack the file, sorry (code %d)", code);
  return 0;
}
```

http://blog.csdn.net/eqera/article/details/8239505

IDA 6.1 调试驱动

今天在测试的时候发现 IDA 5.5 可以启动 windbg 调试器，而 IDA 6.0 却无法启动 windbg 调试器。大体看了一下可能是由于搜索路径造成的，重新将 windbg 安装到 program files 下之后问题就结局了。

网上也有关于用 IDA 调试驱动的文章，这里只是再整理一下，用 IDA 载入驱动分析完成之后选择调试器为 Windbg debugger，如图 1 所示：

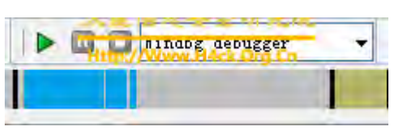


图 1

然后执行菜单中的 Debugger->Debugger options 打开如图 2 所示的设置窗口。

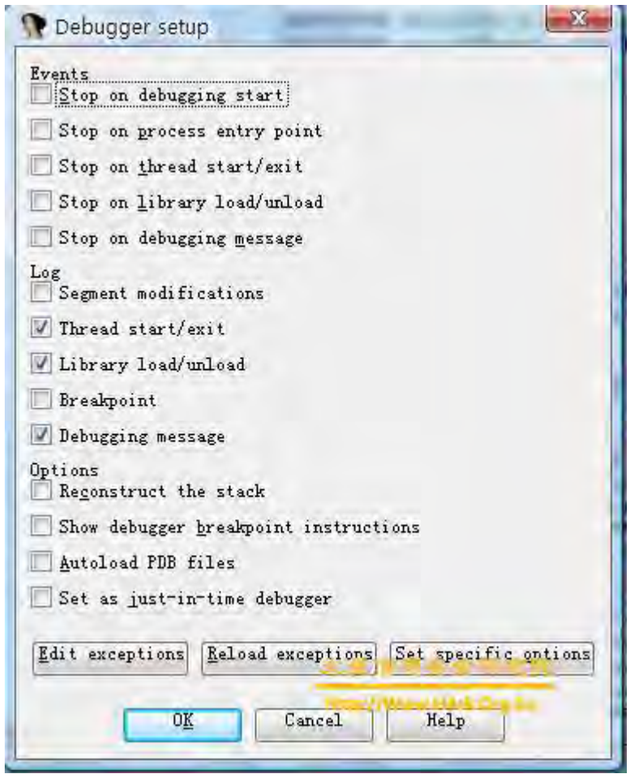


图 2

点击 Set specific options 打开特殊选项窗口，如图 3 所示：

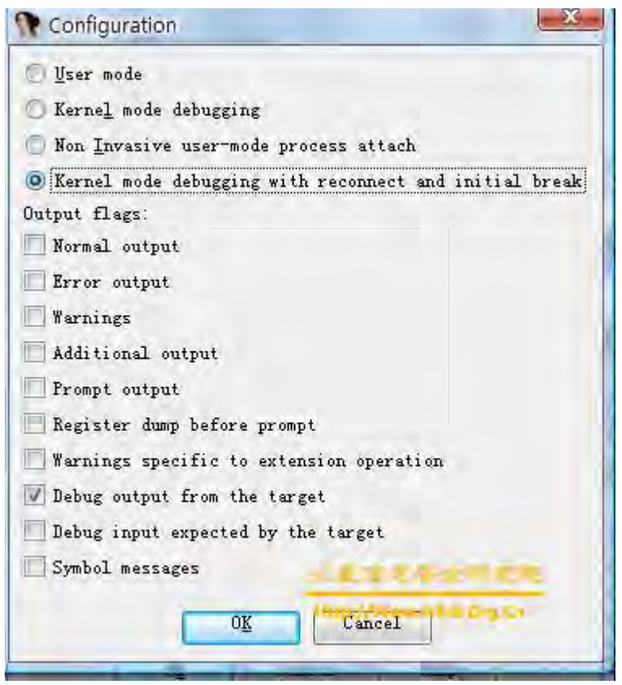


图 3

将最上方的默认 user mode 修改为 Kernel mode debugging 或者 kernel mode debugging with reconnect and initial break，至于两个选项的区别读者可以自行测试一下，这里就不说废话了，按照字面意思理解即可。设置完成后关闭设置窗口，然后执行菜单中的 Debugger->Process options 打开进程选项设置窗口，在 Connet string 中输入要连接的字符串，也就是 com 接口的名称，这里是 com:port=\\.\pipe \com_1,baud=115200,pipe，如图 4 所示。

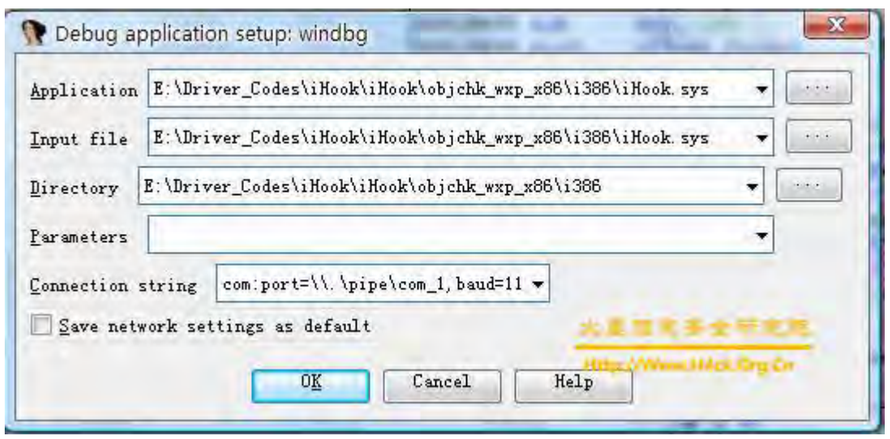


图 4

设置完成后关闭设置窗口，执行菜单中的 Debugger->Attach Process，打开进程附加窗口，如图 05 所示。

图 5

附加之后等待符号库加载完就可以进行调试了。调试器挂在之后如果没有意外会中断在第一个 int3 断点，如图 6 所示。

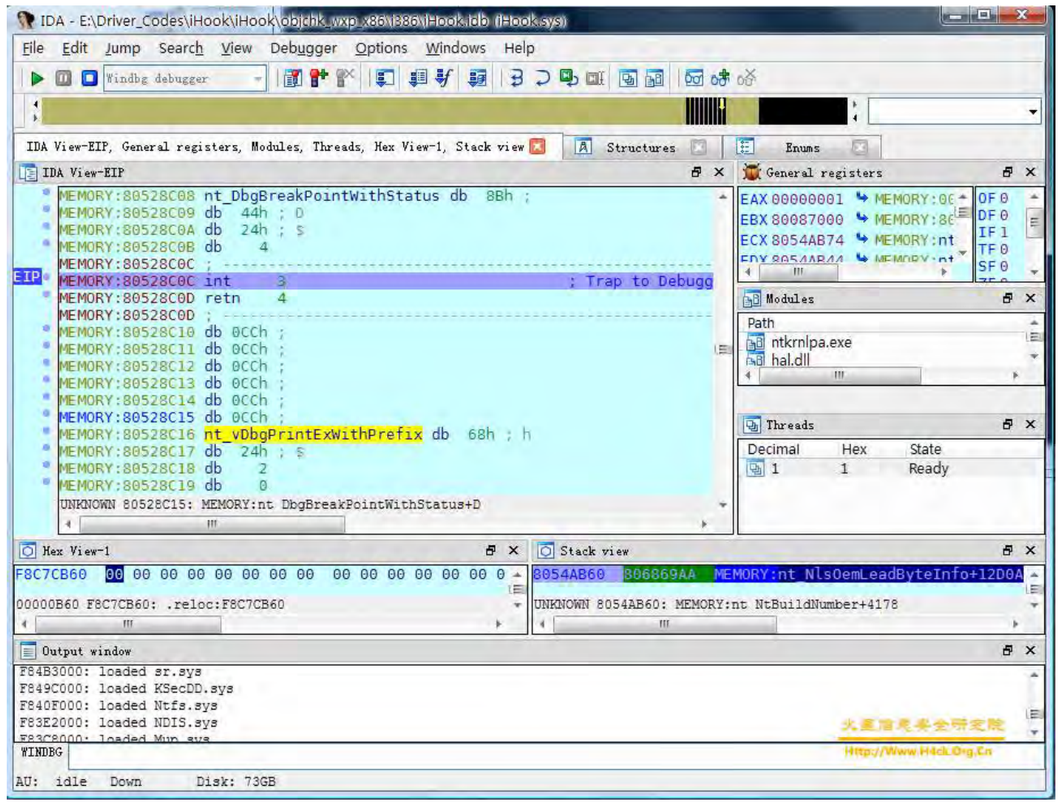


图 6

现在就可以对驱动进行设置断点和调试了，效果如图 7 所示：

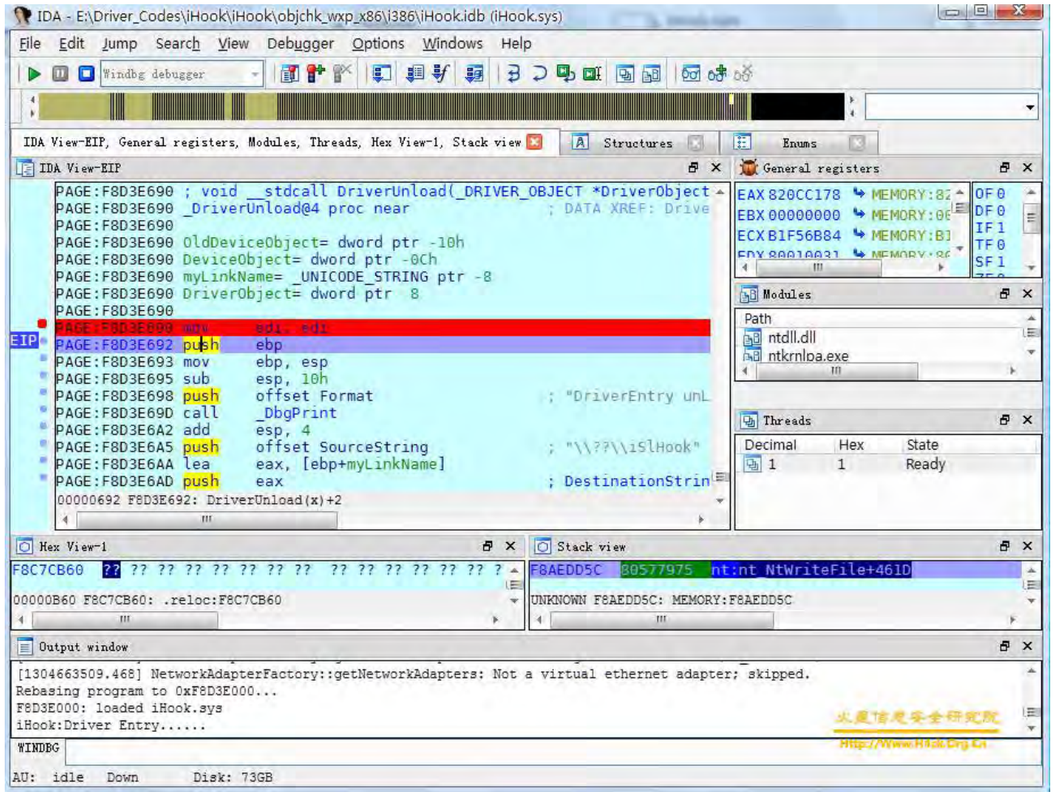


图 7

在调试之前为了使程序的断点能够中断需要修正 Process options 选项中的部分参数，如图 8 所示。

图 8

需要将 Application 修改为程序在远程目标机器上的路径，另外 Directory 同样需要修改为远程系统的目录。否则在调试的过程中如果设置断点将会询问本地文件与远程文件是否一样，并且设置的断点将无法触发。

如果使用 IDA 5.5 设置方法是类似的，与上一篇文章《IDA 调试内核》http://www.h4ck.org.cn/2011/05/kernel-debugging-with-ida-pro/ 不同的是本文的调试没有使用第三方的工具，并且实现方法也比较简单。如果调试没有源码的驱动用 IDA 应该会更直观一些吧，如果有源码的话还是使用 Windbg 更好一些。

注意：

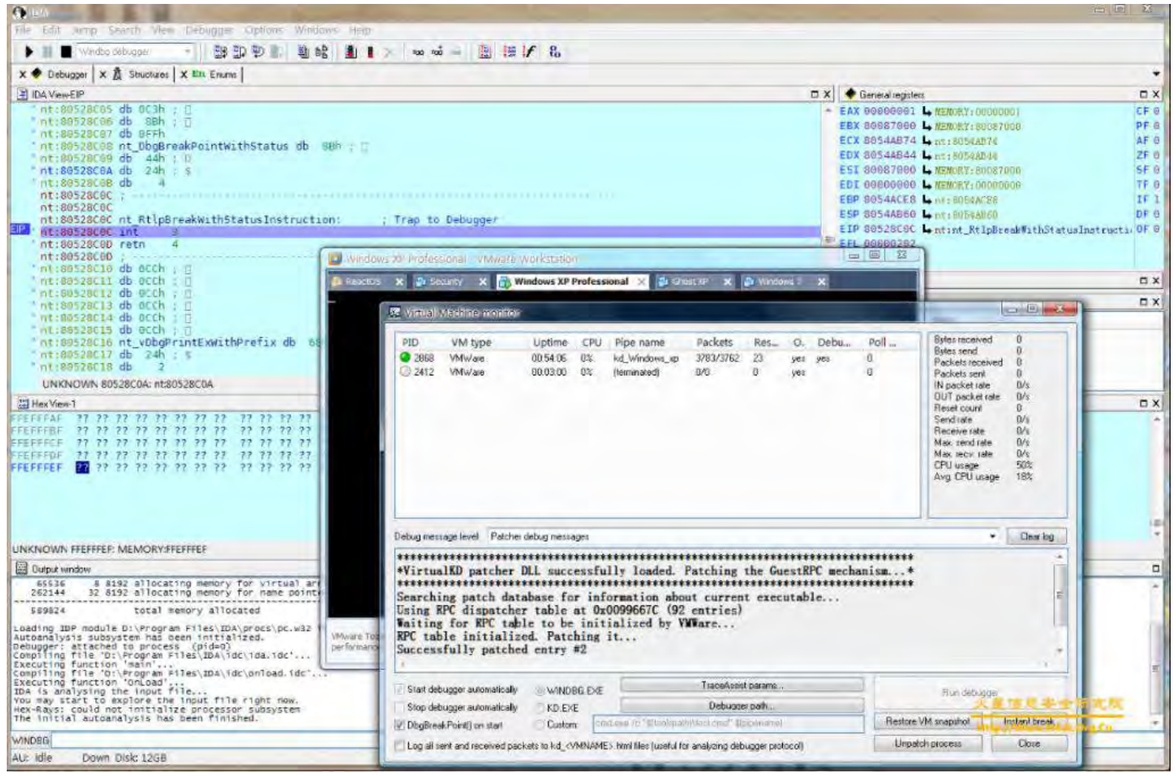
需要注意的是前提已经配置好了 Windbg 的远程调试，否则使用 IDA 是无法连接虚拟机进行调试的，设置方法可以参考下面的连接：

Windows 7：http://www.h4ck.org.cn/2010/11/win7-remote-debug-via-windbg/

Windows XP：http://www.h4ck.org.cn/2009/09/driverdevelop1/

IDA 调试 Windows 内核

由 obaby



以前总想知道 IDA 是否能够实现内核调试，后来找了一段时间没什么结果就暂时放弃了。今天在国外的一个博客上偶然看到了用 IDA 实现内核调试的方法。

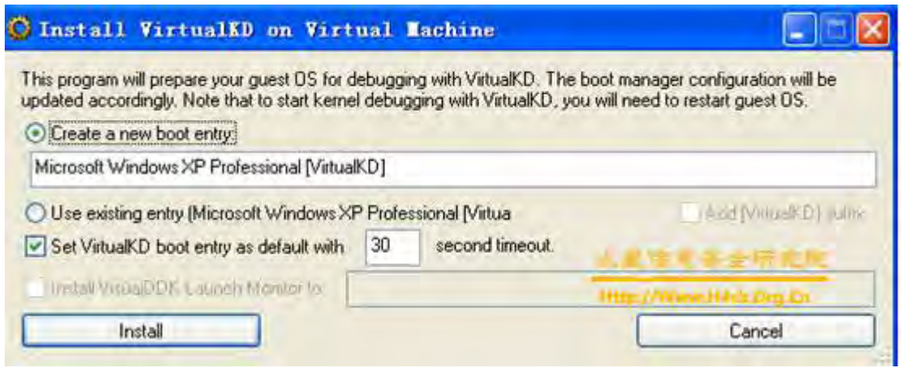
其实现在国内也有很多文章介绍了 IDA 通过串口进行调试的文章，如果大家想看的话可以搜索下。

这里只是参考远远吧实现的方法大体的用中文表述了一下。在调试之前需要安装如下的软件：

1. IDA PRO 这个我想大家都应该有了；
2. Windbg 如果调试过驱动或者系统内核的话这个东西也应该有了；
3. VirtualKd 这个东西我想大家如果没有做过使用 IDA 调试内核的话这个东西应该是还没有。

安装 VirtualKD

首先从官方网站上下载 VirtualKd。将程序解压到任意目录下，将程序目录下的 Target 文件夹拷贝到虚拟机系统中运行（如果是 VirtualBox 则安装比较麻烦），运行之后将会出现如下的界面：



点击 Install 之后将会在系统的启动菜单中创建一个新的启动项，如下图所示：

如果使用设置 Windbg 调试器的方法来设置 pipe 在使用 IDA 调试的时候是无法正常连接调试器的（话说这个东西我测试了好久，囧）。

另外如果不使用上面的工具进行安装设置启动项的话可以手工设置，不过过程比较繁琐：

1） 拷贝 kdvm.dll 到你的客户机系统的 system32 目录下，在这个目录下应该可以找到 KDCOM.DLL 和 KD1394.DLL 文件；

2） 打开并且编辑 boot.ini 文件添加一项新的启动项如下：

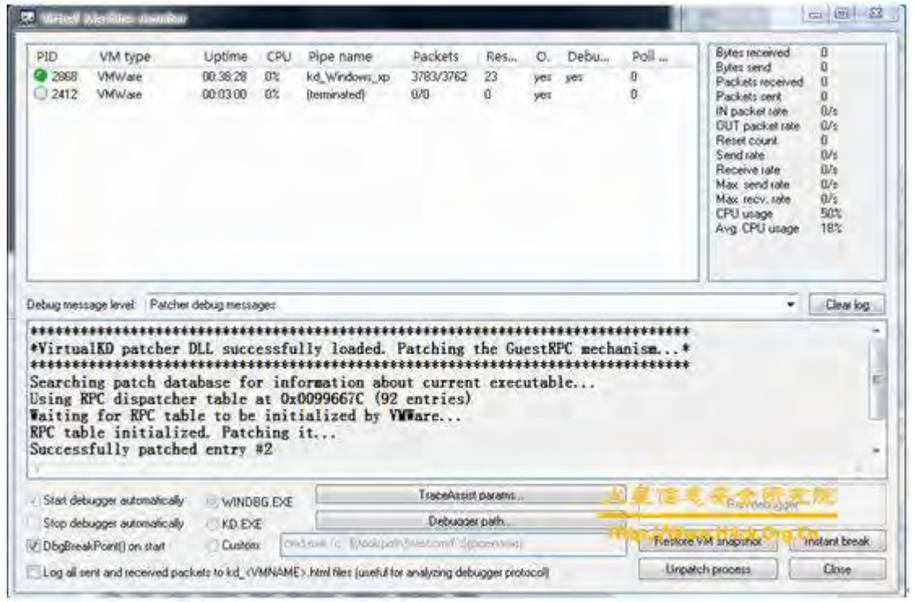
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS=”Microsoft Windows XP Professional” /noexecute=optin /fastdetect /DEBUG /DEBUGPORT=VM

如果是 windows　vista 或者 win7 则需要手工执行 bcdedit 命令来激活 kdvm.dll

bcdedit /set dbgtransport kdvm.dl

3）　重新启动虚拟机并且运行 vmmon.exe 进行监视。（这一步与自动安装是相同的）

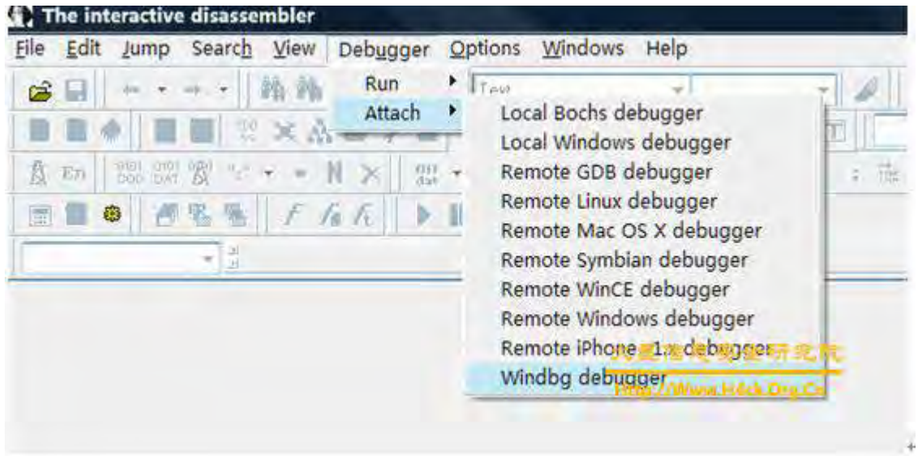
Vmmon 运行界面如下所示：



程序列出了当前运行中（其实有的是已经结束了）的虚拟机的状态，这里需要记住需要调试的虚拟机的 pipe name.,在这里是 kd_Windows_xp。

设置 IDA/Windbg

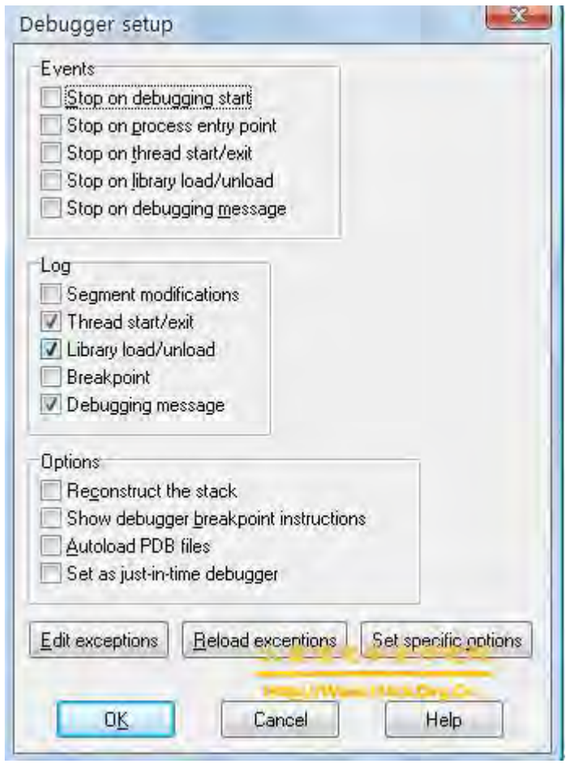
运行 IDA 不要选择任何输入数据库，执行菜单中的 Debugger/Attach/Windbg debugger，如下图所示：



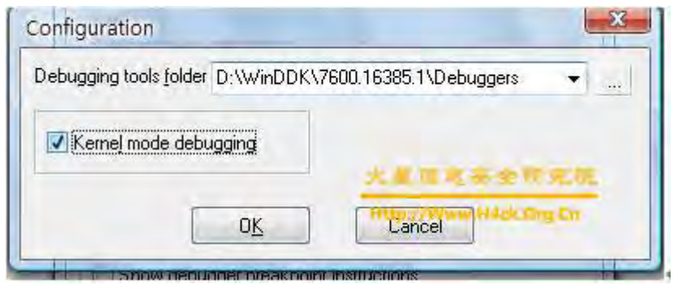
打开如下图所示的设置窗口：

在 Connection string 中输入管道的名称 com:port=\\.\pipe\kd_Windows_xp,pipe，这里需要将 kd_Windows_xp 修改为你的虚拟机对应的名称。

设置完成之后点击 Debug Options 打开选项窗口，如下图所示：



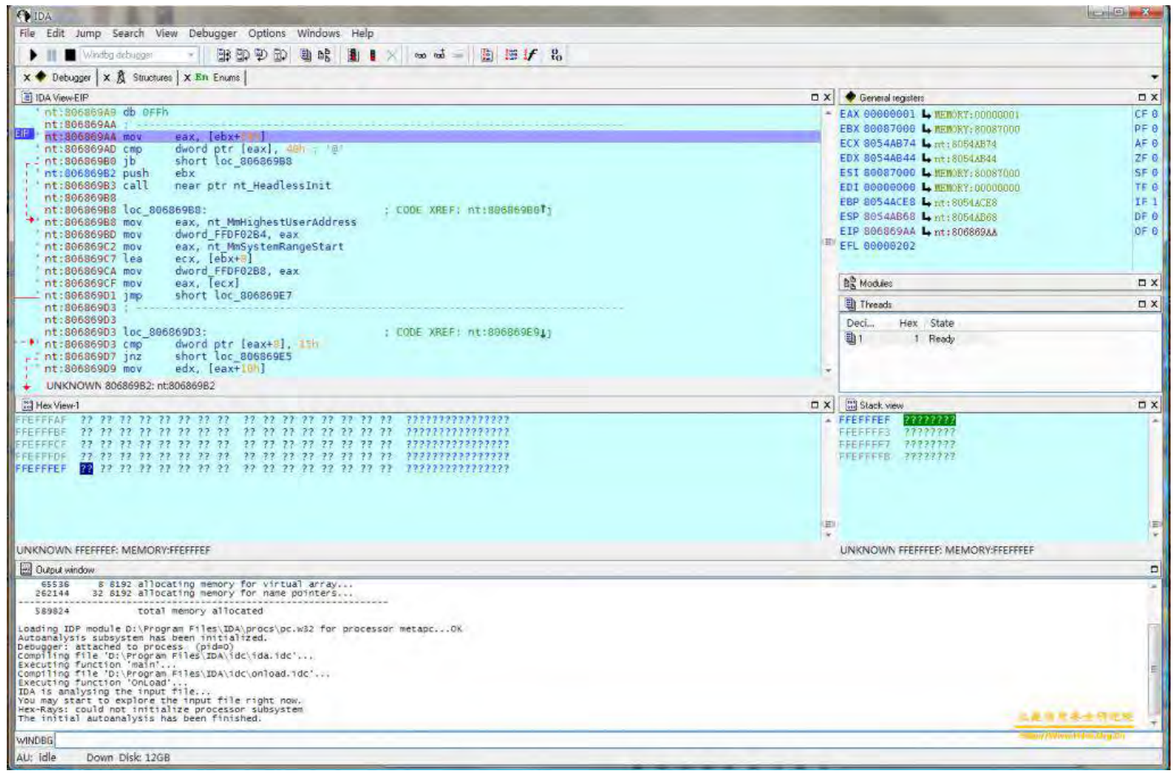
点击 Set specific options，打开特殊选项窗口，如下所示：



输入 Debugging tools folder（调试器路径），并且勾选下面的 Kernel mode debugging（内核调试模式）然后确定即可。

关闭所有的设置窗口之后将会打开如下图所示的附加进程列表：

此时只有一个进程 id 为 0 的进程，选择这个进程 ok 之后就可以进行内核的调试了。不过这个进程的附加会异常的痛苦，尤其是下载符号库的时候，并且将 进程挂起的时候可能会让 ida 假死掉，因而可以多等待一会儿，直到所有的符号库下载并且识别之后就可以真正的中断在系统的 int3 断点上了（这个过程简直 是一种折磨啊）。



挂载之后就是上面的效果，看起来还是不错的。

[猛击此处下载此文的 pdf 版本！]

原创文章，转载请注明： *转载自 火星信息安全研究院*

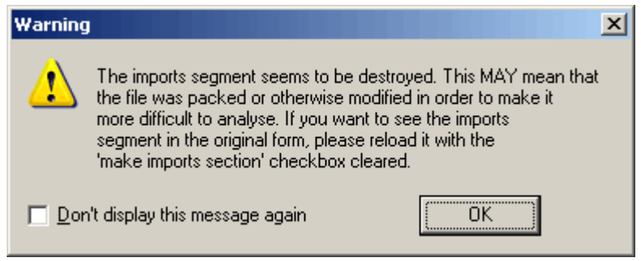
本文标题： *《IDA 调试 Windows 内核》*

本文链接地址： *http://www.h4ck.org.cn/2011/05/kernel-debugging-with-ida-pro/*

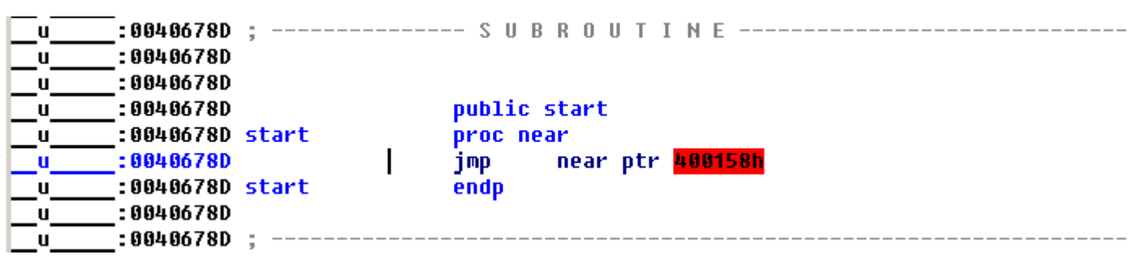
用 IDA 的调试器来手工脱壳（PE 格式文件）

[作者]未知（感谢 linhanshi 提供资料）
[译者]4nil
[date]12-Jan-05

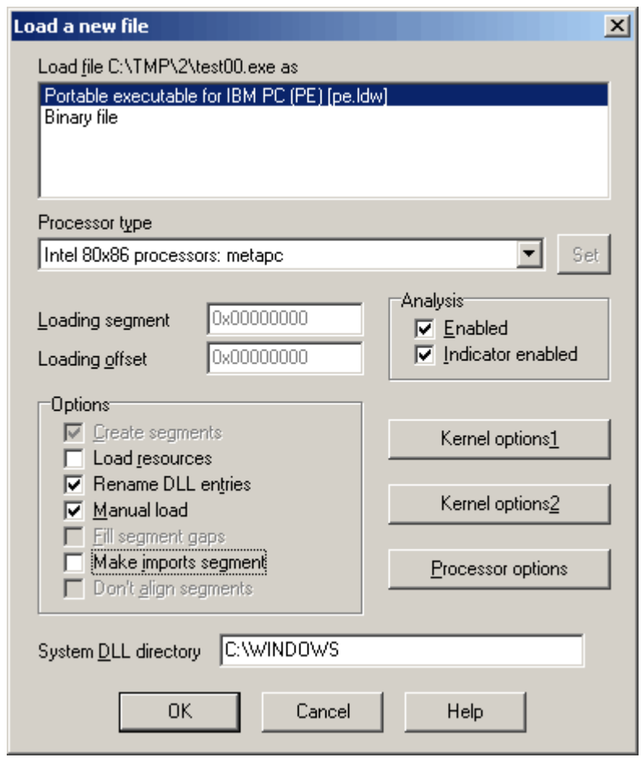
几天前，一个 IDA Pro 使用者发给我们一个小程序(test00.exe)（非破坏性），无法用 IDA 调试，断点没法停下来，程序依旧自由执行，就好 像调试器太慢跟不上它。当我们第一次载入的时候，提示告诉我们无法找到引入段。这种情况我们平时在破解受 保护的或者加过壳后的程序时经常碰到的。



还有一件值得注意的事就是入口点的跳转不存在。。。红色标记的地址表示 IDA 无法解决的。



这个文件的代码企图阻止反编译，因此，我们缺省的加载参数就不合适了。这种形式的迷惑性代码，问题的关键在于单击的过程当中。
但是假如我们深入研究，比如用人工模式载入文件？这个模式下，用户可以指定哪个特定的段需要载入。基于安全考虑，我们加载所有的段。如下，去掉'make imports section'选框防止"missing imports"的提示.



一旦我们解决了关于段的载入问题：问题好多了。

现在我们去除了无法定位的地址的问题，可以开始分析这个程序了。程序第一个指令是一个 jump，跳到程序开头： loc_400158。一般来说程序开 头不建议有代码跳转，但是这个程序用了。这个导致了程序头是只读的。这样就回答了为什么我们无法不能在那里下断的问题。
让我们继续来看程序是怎么运行的。我们发现程序给 ESI 加载了一个指针，然后又马上拷到 EBX：

```
HEADER:00400158                         mov         esi,  offset  off_40601C
HEADER:0040015D                         mov         ebx,  esi
(Ctrl-O  converted  the  hexadecimal  number  in  the  first  instruction  to  a  label  expression)
```

（用 Ctrl-O 把第一个指令里的 16 进制数字转为标签形式）
后来 EBX 的值又被用来 call 一个子程序：

```
HEADER:00400169                         call        dword  ptr  [ebx]
```

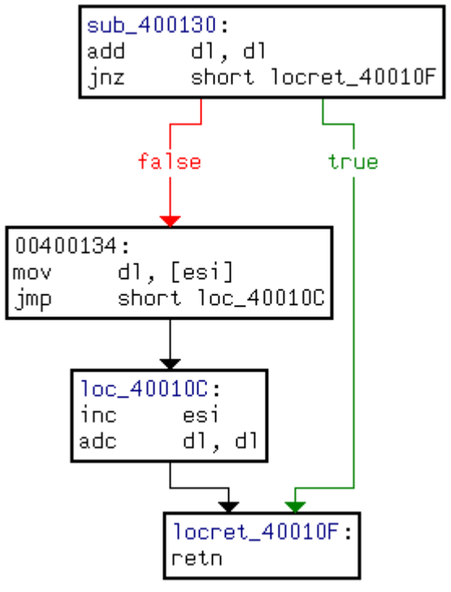
像这样的 Call 在列表里很多，所以我们要找出它是什么功能的。显然指向改功能的指针如下：

```
__u_____:0040601C  off_40601C           dd  offset  __ImageBase+130h
```

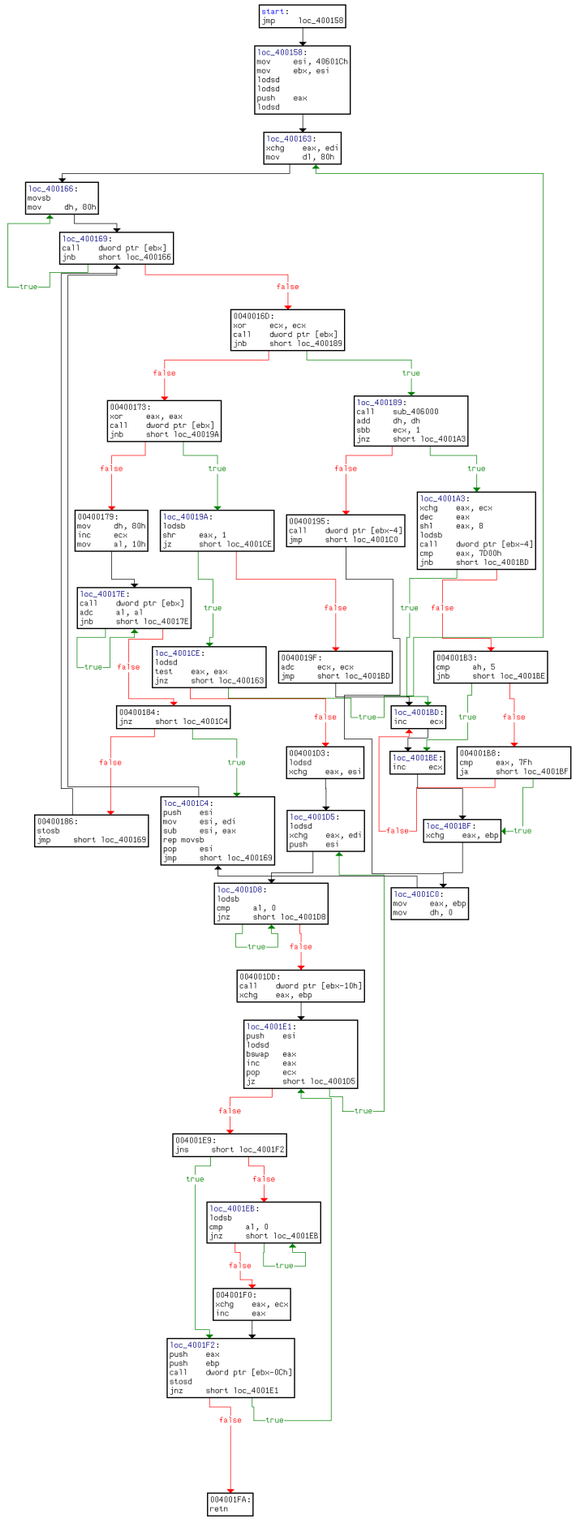
假如我们点击 ImageBase，我们看到一列的 dwords。IDA 展现给我们的程序头看起来不是我们想要的形式。我们 undefine 那段代码（热键 U），回到指针那里（热键 Esc）然后再次跟踪。这次我们会停在 0x400130 处，这里本应该包含一个函数。我们确信这一点是因为在 0x400169 的 指令间接 calls 0x400130。我们按下 P（创建一个函数或者子程）告诉 IDA 当前地址应该有一个函数。现在函数出来了，但是我们只有一半！看起 来写程序的人为了迷惑我们，将函数分割为几部分。现在 IDA 知道怎样处理这些碎片函数了，它将函数其他部分也显示在屏幕上。

```
HEADER:0040012E                 db    0
HEADER:0040012F                 db 0C0h ; L
HEADER:00400130
HEADER:00400130 ; --------------- S U B R O U T I N E ------------------------------------
HEADER:00400130
HEADER:00400130
HEADER:00400130 sub_400130      proc near               ; DATA XREF: __u____:off_40601
HEADER:00400130
HEADER:00400130 ; FUNCTION CHUNK AT HEADER:0040010C SIZE 00000004 BYTES
HEADER:00400130
HEADER:00400130                 add     dl, dl
HEADER:00400132                 jnz     short locret_40010F
HEADER:00400134                 mov     dl, [esi]
HEADER:00400136                 jmp     short loc_40010C
HEADER:00400136 sub_400130      endp
HEADER:00400136
HEADER:00400136 ; ---------------------------------------------------------------------------
HEADER:00400138                 db    0
```

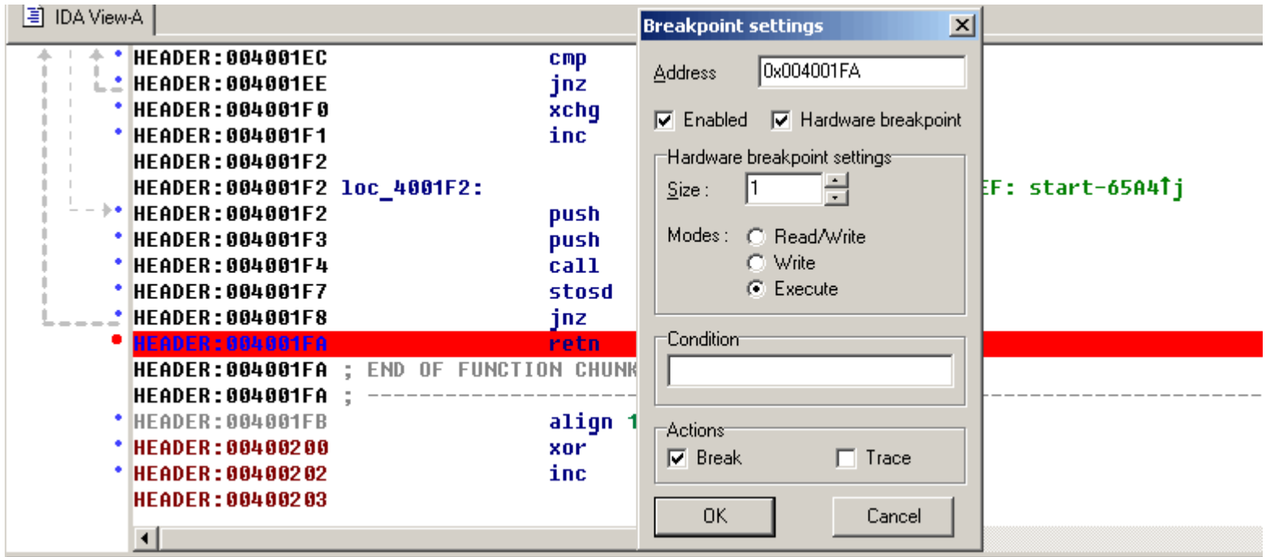
但是它只是给我们一些参考地址，我们希望的的是整个的函数在一起。有一个特别的命令可以帮我们：IDA 里可以制作函数流程图的命令，热键为 F12。这个命令在对于碎片的函数就象我们这种特别有用，它将整个函数显示在一起：



显示主程序的流程会很有趣（很长，继续用你的滚轴吧）：


start:
jmp loc_400158
loc_400158:
mov esi, 40601Ch
mov ebx, esi
lodsd
lodsd
push eax
lodsd
loc_400163:
xchg eax, edi
mov dl, 80h
loc_400166:
movsb
mov dh, 80h
loc_400169:
call dword ptr [ebx]
jnb short loc_400166
true
false
0040016D:
xor ecx, ecx
call dword ptr [ebx]
jnb short loc_400189
false
true
00400173:
xor eax, eax
call dword ptr [ebx]
jnb short loc_40019A
loc_400189:
call sub_406000
add dh, dh
sbb ecx, 1
jnz short loc_4001A3
00400179:
mov dh, 80h
inc ecx
mov al, 10h
loc_40019A:
lodsb
shr eax, 1
jz short loc_4001CE
0040019F:
call dword ptr [ebx-4]
jmp short loc_4001C0
loc_4001A3:
xchg eax, ecx
dec eax
shl eax, 8
lodsb
call dword ptr [ebx-4]
cmp eax, 7D00h
jnb short loc_4001BD
loc_40017E:
call dword ptr [ebx]
adc al, al
jnb short loc_40017E
loc_4001CE:
lodsd
test eax, eax
jnz short loc_400163
0040019F:
adc ecx, ecx
jmp short loc_4001BD
004001B3:
cmp ah, 5
jnb short loc_4001BE
00400184:
jnz short loc_4001C4
loc_4001BD:
inc ecx
004001D3:
lodsd
xchg eax, esi
loc_4001BE:
inc ecx
00400188:
cmp eax, 7Fh
ja short loc_4001BF
00400186:
stosb
jmp short loc_400169
loc_4001C4:
push esi
mov esi, edi
sub esi, eax
rep movsb
pop esi
jmp short loc_400169
loc_4001D5:
lodsd
xchg eax, edi
push esi
loc_4001BF:
xchg eax, ebp
loc_4001D8:
lodsb
cmp al, 0
jnz short loc_4001D8
loc_4001C0:
mov eax, ebp
mov dh, 0
004001DD:
call dword ptr [ebx-10h]
xchg eax, ebp
loc_4001E1:
push esi
lodsd
bswap eax
inc eax
pop ecx
jz short loc_4001D6
004001E9:
jns short loc_4001F2
loc_4001EB:
lodsb
cmp al, 0
jnz short loc_4001EB
004001F0:
xchg eax, ecx
inc eax
loc_4001F2:
push eax
push ebp
call dword ptr [ebx-0Ch]
stosd
jnz short loc_4001E1
004001FA:
retn

快速浏览发现流程图告诉我们函数只有一个出口"ret"指令(0x4001FA)。我们可以在那里下断，然后运行程序。我们在这里要重申的是，不要在电脑 上运行不可信的代码。假如有一个"sandbox"机来做测试会更好，比如使用 remote debugging facilities IDA Pro offers。因此 IDA 会显示警告当一个新文件将被调试：忽略，一切自己负 责。
因为断点位于程序头，而且该程序头是系统写保护的，我们不能在这里使用平常的软件断点。我们必须使用硬件断点：首先用 F2 设断点，右击，选择"edit breakpoint"，把它变成一个硬件断点，我们用"execution"模式。



设好断点后，我们按 F9 开始调试。当我们在那里断下后，程序会被解压到'MEW'段。我们跳到那里，然后将所有的都转为代码（最快的方式是在断点的地方按 F7）。
现在我们得到一个很不错的程序段，但是唯一不足的是它会在结束调试后消失。

```
MEW:00401000 MEW segment para public 'BSS' use32
MEW:00401000 assume cs:MEW
MEW:00401000 ;org 401000h
MEW:00401000 assume es:nothing, ss:nothing, ds:nothing, fs:nothing,
MEW:00401000
MEW:00401000 loc_401000:                             ; DATA XREF: __
MEW:00401000                                         ; __u____:0040
MEW:00401000 push    0
MEW:00401002 call    sub_4012DC
MEW:00401007 mov     ds:dword_403010, eax
MEW:0040100C call    sub_4012D6
MEW:00401011 mov     ds:dword_403008, eax
MEW:00401016 push    0Ah
MEW:00401018 push    ds:dword_403008
MEW:0040101E push    0
MEW:00401020 push    ds:dword_403010
MEW:00401026 call    sub_401031
MEW:0040102B push    eax
MEW:0040102C call    sub_4012D0
MEW:00401031
```

原因很简单，这个程序段是一些内存内容，这样在程序终止的时候它也会消失。我们假如能够把它存成数据会更好，可以不在调试模式下分析。我们会考虑在将来的 IDA 版本中添加这个功能，但是现在我们必须手工做这个事。“手工”不代表我们要一个一个字节拷贝，当然，我们可以用 IDC 内置语言来做这个。
我们要保存两样东西：内存内容还有输入函数名。内存内容可以用下面的 4 行语句来保存：

```
auto  fp,  ea;
fp  =  fopen("bin",  "wb");
for  (  ea=0x401000;  ea  <  0x406000;  ea++  )
    fputc(Byte(ea),  fp);
```

当脚本执行后，我们得到一个名为"bin"的文件。它包含了"MEW"段的所有字节。你可以发现，我存的是 16 进制的地址。这个脚本可以任意使用。
我们还要保存输入函数名，看一下 0x401002 处的 Call：

```
MEW:00401002  call        sub_4012DC
```

假如我们要得到名字，我们只要按 Enter 键数次，最后就可以得到了：

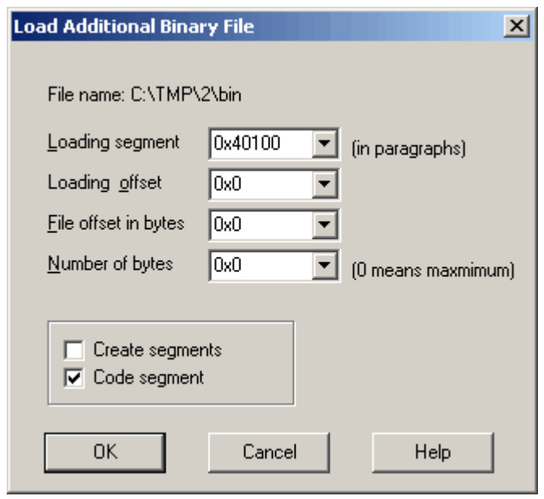
```
kernel32.dll:77E7AD86
kernel32.dll:77E7AD86  kernel32_GetModuleHandleA:                    ;  CODE  XREF:  sub_4012DCj
kernel32.dll:77E7AD86                                                ;  DATA  XREF:  MEW:off_402000o
kernel32.dll:77E7AD86  cmp          dword  ptr  [esp+4],  0
```

当我们退出调试后，kernel32.dll 的段也会消失，还有它所有的名称，指令，函数，所有的。我们必须在它消失前将它拷贝过来：

```
auto  ea,  name;
for  (ea  =  0x401270;  ea<0x4012e2;  ea  =ea+6  )
{
    name  =  Name(Dword(Dfirst(ea)));                    /*  get  name  */
    name  =  substr(name,  strstr(name,  "_")+1,  -1);  /*  drop  the  prefix  */
    MakeName(ea,  name);
```

}

现在已经运行了这些脚本，停止调试（Ctrl+F12）。拷回数据到内存里去，我们可以这样做：



注意，我们没必要创建一个段，它已经存在（清除"create segments"标志），而且，地址同样被指定了，比如：右偏移 4。

载入文件，0x401000 处按下 P，可以看到如下信息：

```
MEW:004010FA                push    [ebp+X]         ; X
MEW:004010FD                push    0CF0000h        ; dwStyle
MEW:00401102                push    offset WindowName ; lpWindowName
MEW:00401107                push    offset ClassName ; lpClassName
MEW:0040110C                push    300h            ; dwExStyle
MEW:00401111                call    CreateWindowExA
MEW:00401116                mov     ds:hWnd, eax
MEW:0040111B                push    258h            ; lpMenuName
MEW:00401120                push    [ebp+hInstance] ; hInstance
MEW:00401123                call    LoadMenuA
MEW:00401128                push    eax             ; hMenu
MEW:00401129                push    ds:hWnd         ; hWnd
MEW:0040112F                call    SetMenu
MEW:00401134                push    1               ; nCmdShow
MEW:00401136                push    ds:hWnd         ; hWnd
MEW:0040113C                call    ShowWindow
MEW:00401141                push    ds:hWnd         ; hWnd
MEW:00401147                call    UpdateWindow
```

接下去的分析就很平坦了。。。你自己想怎么做吧。。

http://www.pediy.com/kssd/pediy07/pediy7-714.htm

IDA + Bochs 调试器插件进行 PE+ 格式 DLL 脱壳

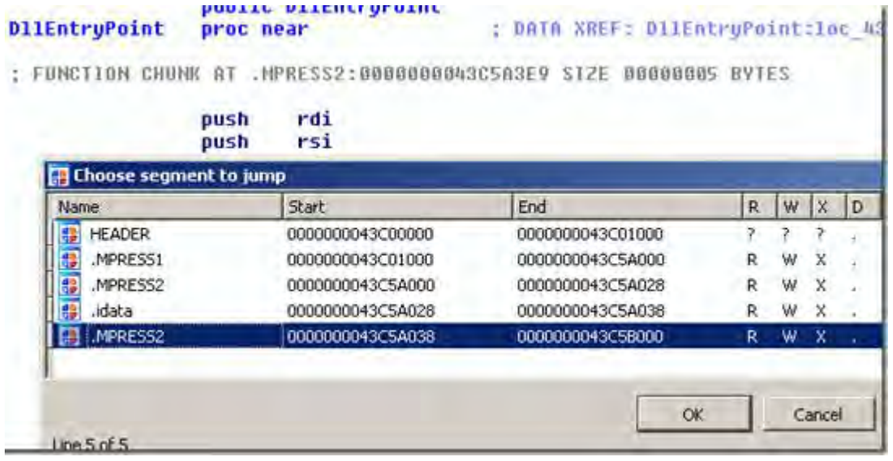
By :obaby

在 IDA Pro6.1 中我们扩展了 Bochs 调试器插件，现在已经可以进行 64 位代码段的调试。在 IDA Pro 6.2 版本中将有可能实现 PE+ 可执行程序的动态调试。由于程序将会在 Bochs 系统中执行，因而在调试的过程中我们并不需要实际的 64 位操作系统，因而在实际的调试过程中可以从任何的 32 位或者 64 位的 Linux，Mac OS 或者 Windows 操作系统中使用 IDA Pro 进行 64 位可执行文件的调试。

为了确认这一项新的功能，我们将进行 PE+格式的一个木马程序进行脱壳并且进行一个大体的分析，这个文件是由 MATCODE Software 公司的 mpress 进行压缩的。我们将会对讲解 DLL 文件脱壳，修复输入表并且最终修复数据库来进行分析。

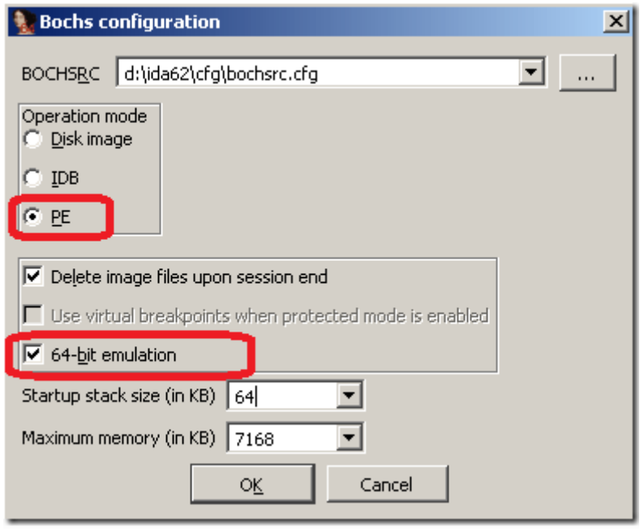
Unpacking the DLL

我们的目标文件是一个木马的 DLL 文件，该文件被杀软识别为“Win32/Giku”。我们从使用 idaq64 载入 DLL 文件开始进行分析，载入之后按 Ctrl+S 键打开区段窗口：

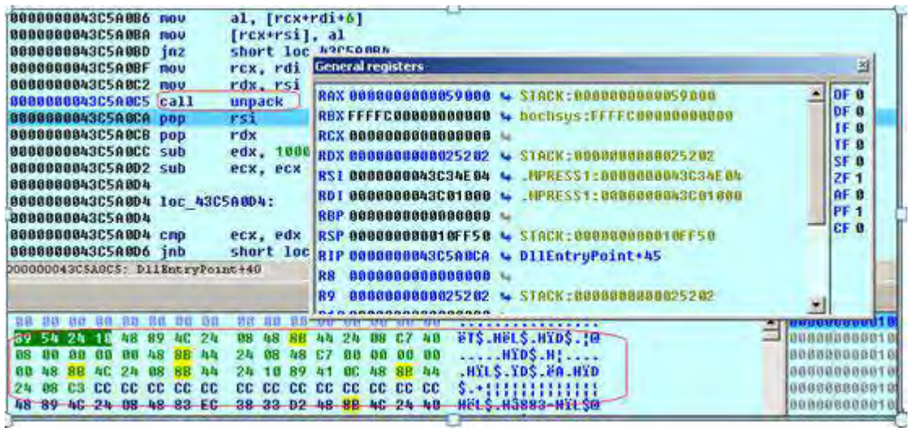


打开区段窗口之后注意观察区段的名称和 mpress 压缩壳设置的区段的属性。

为了进行 DLL 文件调试需要确保在启动之前已经设置调试器的选项设置（“Bochs debugger plugin”）为 PE 和 64bit emulation 模式。



在启动调试器之后，注意观察下面的代码段，在这段代码中调用了 unpack()函数：



如果我们继续单步执行到更远的地方我们将会到达修复输入表的代码处，为了实现输入表的修复程序将会循环调用 LoadLibrary()/GetModuleHandle()函数并且在这个循环中会包含另外的一个子循环调用 GetProcAddress()。 Mpress 外壳通过这两层循环来实现 IAT 修复：

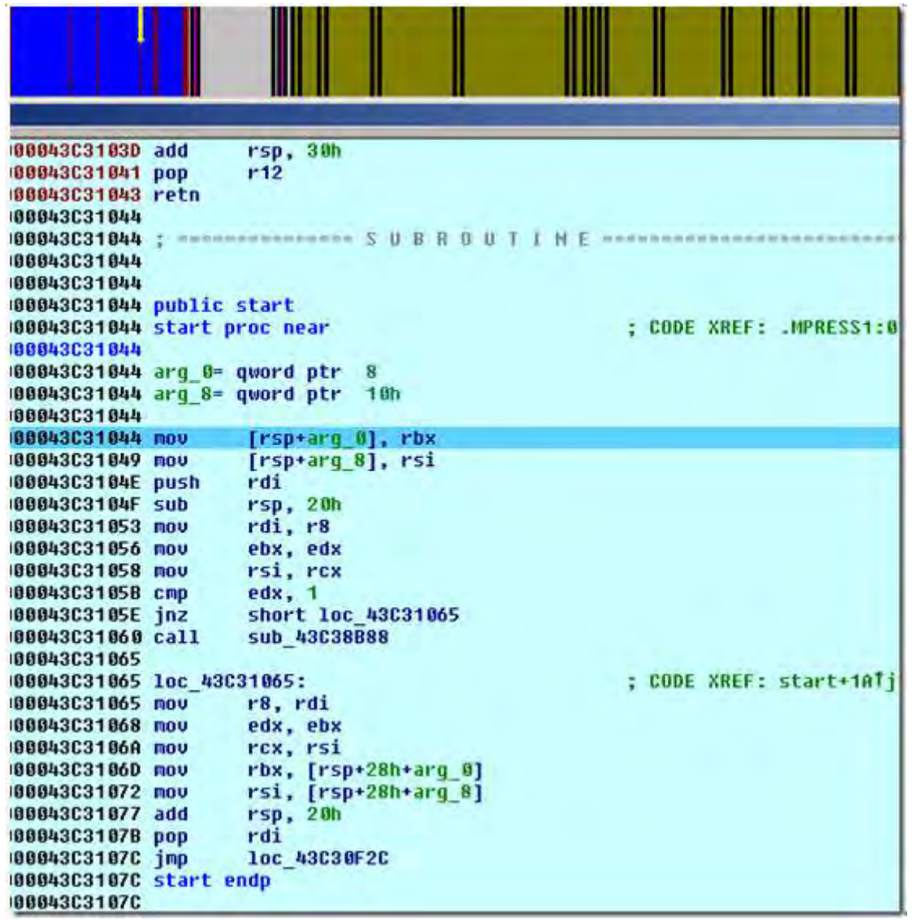
```
1043C4FB1B ; get module handle
1043C4FB1B add     rdi, rax
1043C4FB1E mov     rcx, rsi                    ; lpModuleName
1043C4FB21 call    GetModuleHandleA
1043C4FB26 mov     rbx, rax
1043C4FB29 @@clear_module_name:                ; CODE XREF: sub_43C4FA70+C1↓j
1043C4FB29 lodsb
1043C4FB2A or      al, al
1043C4FB2C mov     al, 0
1043C4FB2E loc_43C4FB2E:
1043C4FB2E mov     [rsi-1], al
1043C4FB31 jnz     short @@clear_module_name
1043C4FB33 @@next_proc:                        ; CODE XREF: sub_43C4FA70+F4↓j
1043C4FB33 lodsb
1043C4FB34 or      al, al
1043C4FB36 loc_43C4FB36:
1043C4FB36 jz      short loc_43C4FB14
1043C4FB38 cmp     al, 20h
1043C4FB3A jbe     short loc_43C4FB44
1043C4FB3C mov     rdx, rsi
1043C4FB3F dec     rdx
1043C4FB42 jmp     short loc_43C4FB50
1043C4FB44 loc_43C4FB44:                       ; CODE XREF: sub_43C4FA70+CA↑j
1043C4FB44 lodsw
1043C4FB46 dec     rsi
1043C4FB49 mov     byte ptr [rsi], 0
1043C4FB4C movzx   rdx, ax                     ; lpProcName
1043C4FB50 loc_43C4FB50:                       ; CODE XREF: sub_43C4FA70+D2↑j
1043C4FB50 mov     rcx, rbx                    ; hModule
1043C4FB53 call    GetProcAddress
1043C4FB58 stosq                               ; store procaddr in IAT
1043C4FB5A @@clear_import_name:                ; CODE XREF: sub_43C4FA70+F2↓j
1043C4FB5A xor     al, al
1043C4FB5C mov     [rsi-1], al
1043C4FB5F lodsb
1043C4FB60 or      al, al
1043C4FB62 jnz     short @@clear_import_name
1043C4FB64 jmp     short @@next_proc
1043C4FB66 loc_43C4FB66:                       ; CODE XREF: sub_43C4FA70+A9↑j
```

在 stosq 执行之后我们将可以从 rdi 寄存器中得到 IAT 结构的起始地址，同样在两层循环全部结束之后我们可以从 rdi 寄存器中得到 IAT 结构的结束地址。

在 IAT 修复之后不远的地方我们可以找到一个跳转到原始入口点的 jmp 代码：

```
000000043C4FB6D mov     al, 0E9h
000000043C4FB6F stosb
000000043C4FB70 mov     eax, 10Ch
000000043C4FB75 stosd
000000043C4FB76 add     rsp, 28h
000000043C4FB7A pop     r8
000000043C4FB7C pop     rdx
000000043C4FB7D pop     rcx
000000043C4FB7E pop     rbx
000000043C4FB7F pop     rsi
000000043C4FB80 pop     rdi
000000043C4FB81 jmp     loc_43C31044                ; jump to OEP
```

程序的入口点代码如下所示：



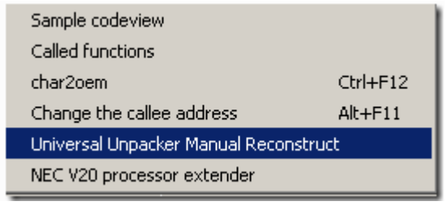
这里就是脱壳之后的程序的真实的 DllEntryPoint()函数了。现在有了程序的 OEP 和 IAT 结构的起始/结束地址，我们就可以清空数据库并且重现脱壳之后的程序了。

Reconstructing and cleaning the database

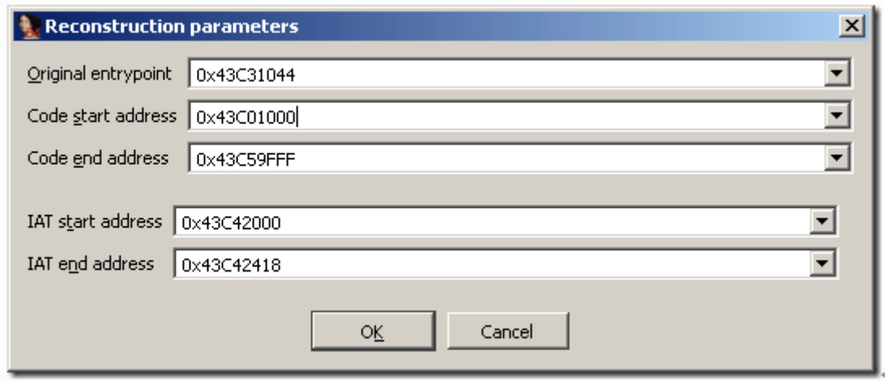
到这里有许多的办法在程序脱壳之后进行清理数据库清理.通常会包含如下几步：

1. 定位 IAT 并且创建一个额外的区段来重现程序的输入表；
2. 删除外壳代码的入口点，并且添加脱壳之后程序的原始入口点 OEP；
3. 重新分析代码；
4. 重新加载 FLIRT 特征库
5. 删除无用的外壳区段（可选）

其中第一步到第三步可以通过 IDA 的 uunp 插件来自动完成，执行菜单中的“Edit/Plugins/Universal unpacker manual reconstruct”即运行该插件：

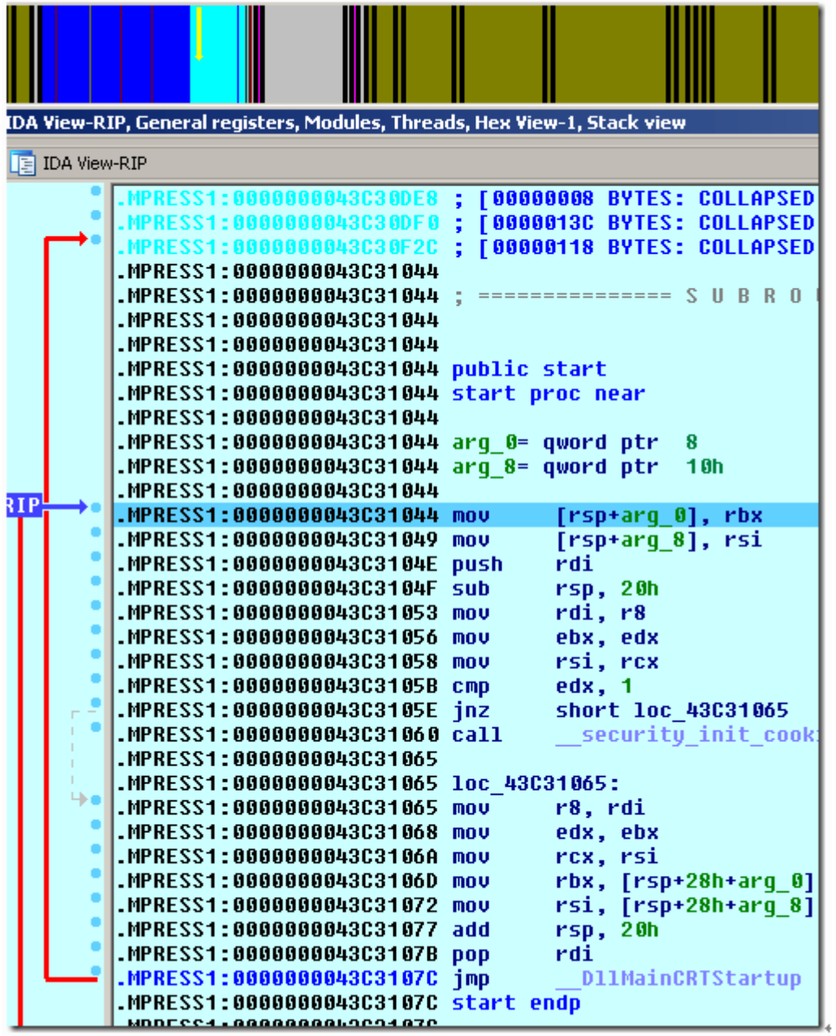


在插件中填入通过上面的操作得到的数据即可：



点击确定之后一个新的区段将会被创建，并且代码段将会被重新分析，在分析完成之后脱壳之后的程序的一个内存快照将会被呈现出来。

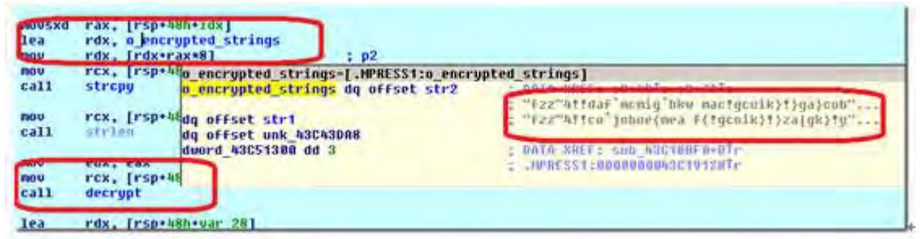
现在我们就可以重新引用 FLIRT 特征库了：



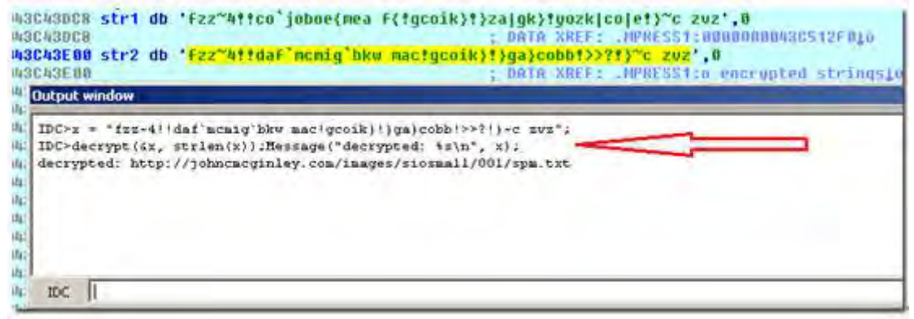
在选择“vc64rtf”签名（shift +F5）之后我们可以看到 IDA 已经成功的识别出了库函数并且对这些库函数使用浅蓝色进行了标记，这样可以使得后续的分析工作变得更加简单。

Analyzing the unpacked code

在代码解压之后我们可以通过 String Window 窗口进行一个快速分析，String Window 窗口呈现了一些加密的字符串：



通过交叉引用，我们可以定位到解密函数。在给函数适当的参数之后我们可以直接通过 Appcall 来解密这些字符串：



我们得到了一个指向加密的文本文件的 URL。在深入挖掘之后我们定位到了解密文件的函数：

```
43C10D20 ; int __cdecl decrypt_file(LPCSTR lpFileName, void *p_buf)
43C10D20 decrypt_file    proc near               ; CODE XREF: sub_43C13A50+28↓p
43C10D20
43C10D20 sz              = qword ptr -18h
43C10D20 buf             = dword ptr -10h
43C10D20 lpFileName      = qword ptr  8
43C10D20 p_buf           = qword ptr  10h
43C10D20
43C10D20                 mov     [rsp+p_buf], rdx
43C10D25                 mov     [rsp+lpFileName], rcx
43C10D2A                 sub     rsp, 38h
43C10D2E                 mov     [rsp+38h+sz], 0
43C10D37                 lea     rdx, [rsp+38h+buf] ; sz
43C10D3C                 mov     rcx, [rsp+38h+lpFileName] ; lpFileName
43C10D41                 call    read_whole_file
43C10D41
43C10D46                 mov     [rsp+38h+sz], rax
43C10D4B                 cmp     [rsp+38h+sz], 0
43C10D51                 jz      short loc_43C10D74
43C10D51
43C10D53                 mov     edx, [rsp+38h+buf] ; sz
43C10D57                 mov     rcx, [rsp+38h+sz] ; buf
43C10D5C                 call    decrypt2
43C10D5C
43C10D61                 cmp     [rsp+38h+p_buf], 0
43C10D67                 jz      short loc_43C10D74
43C10D67
43C10D69                 mov     rcx, [rsp+38h+p_buf]
43C10D6E                 mov     eax, [rsp+38h+buf]
43C10D72                 mov     [rcx], eax
43C10D72
```

下面是 Appcall 版本的 decrypt_file()函数：

```python
import idaapi

def decfile(fn):
    try:
        f = open(fn, "rb")
    except:
        return None

    s = f.read()
    sz = f.tell()
    f.close()

    # Create a reference to the read buffer
    buf = Appcall.byref(s)
    Appcall.decrypt2(buf, sz)

    s = buf.value
    f = open(fn+".out", "wb")
    f.write(s)
    f.close()
    return s
```

我们使用这个函数就可以解密 spm.txt 文件了，解密之后的内容如下所示：

X32.jpg 是一个 upx 压缩的 DLL 文件，x64.jpg 是一个 mpress 压缩的 PE+格式的 Dll 文件。

PDF 版下载：http://dl.dbank.com/c0edp2nnb8

好了文章到此结束，欢迎评论！

原创文章，转载请注明： *转载自 火星信息安全研究院*

本文标题： *《实战 IDA PE+ DLL 脱壳》*

本文链接地址： *http://www.h4ck.org.cn/2011/07/ida-pe6-dll-unpack/*

IDA + Debug 插件 实现 64Bit Exe 脱壳

By ： obaby

对于 64 位的可执行程序已经搞了好长一段时间了，但是却一直没有写点什么东西。前面的两篇文章仅仅是单纯的翻译，个人认为不管是 32 位还是 64 位的 程序脱壳只要能到达程序的 OEP 就可以了。现在支持 64 位加壳的程序貌似也不多，这里以 mpress 压缩的 64 位系统下的 64 位 notepad 为例进行简 单的演示。在《IDA + Bochs 调试器插件进行 PE+ 格式 DLL 脱壳 》一问中提到了可以使用 bochs 调试器进行 DLL 文件脱壳。但是却没有办法进行 64 位 EXE 文件调试，启动调试之后由于代码完全识别错误，因为会出现异 常导致无法调试。要想调试 64 位可执行程序目前只有通过远程调试的方式，使用 Windbg 插件同样是无法进行调试的。但是用 windbg 调试时将会提示如 图 1 所示的信息：

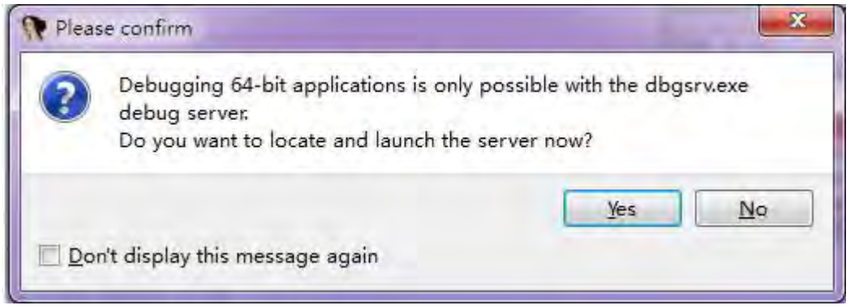


图 1

而直接运行 dbgsrv 启动的其实是和 ida 内置的远程调试插件差不多的一个服务。命令行参数如图 02 所示。

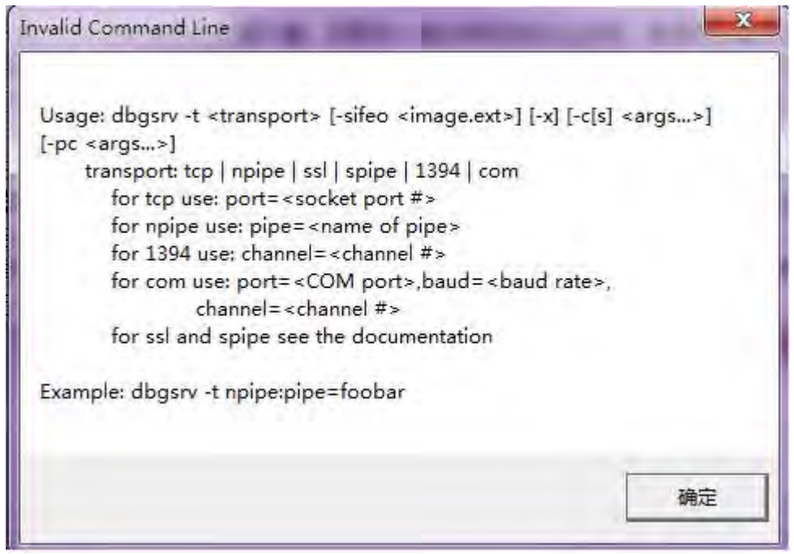


图 2

因而在实际的调试过程中还是一个远程调试模式。至于具体的用法感兴趣的可以看下 windbg 的帮助文件，里面有详细的介绍。

这里就直接以 IDA 的远程调试程序进行吧，由于是本机调试，并且本地系统是 64bit win7 所示直接运行 win64_remotex64.exe 启动调试器即可，启动之后界面如图 3 所示。

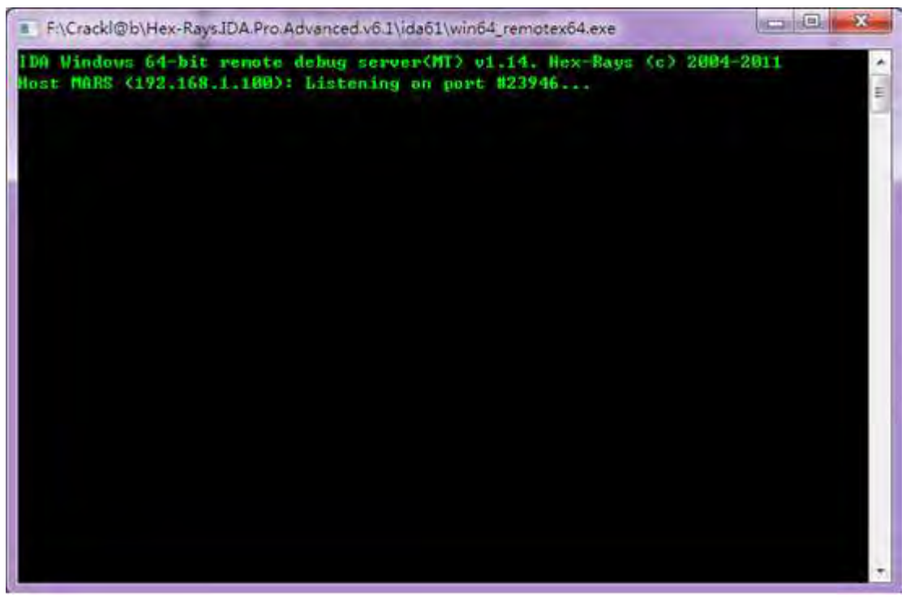


图 3

在 IDA 中将调试器设置为 Remote Win32 Debugger，入股 4 所示。

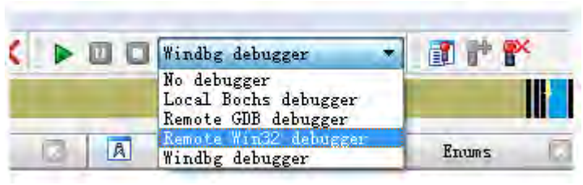


图 4

此时就可以 F9 运行程序了，由于这里没有配置调试器选项因而在启动的时候会出现一些提示信息，全部点掉之后将会打开如图 5 所示的进程调试设置窗口， 在 Hostname 中输入本机 IP 地址，端口中输入服务器监听的调试端口。如果服务器不再本地则直接输入服务器的 IP，通过这种方式可以在 32 位系统上进 行 64 位程序的调试，这不是本文的重点就不再介绍了。

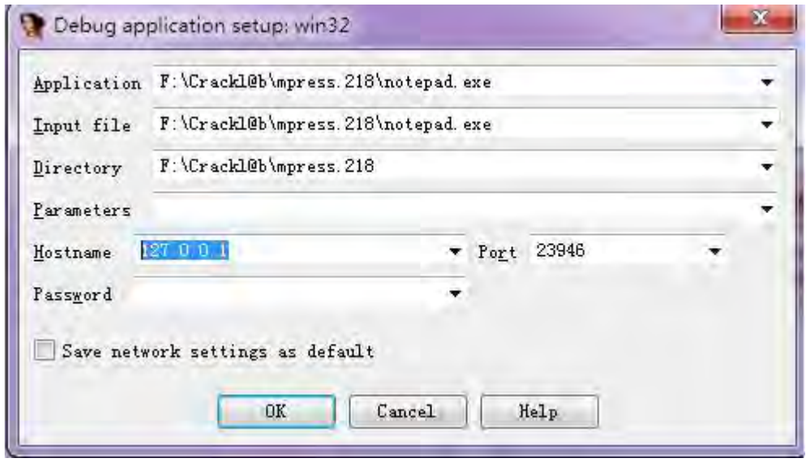


图 5

在启动之前最好在程序的入口点设置一个断点，否则程序就会直接启动了。启动之后将会停留在如下的代码处（已经设置断点）

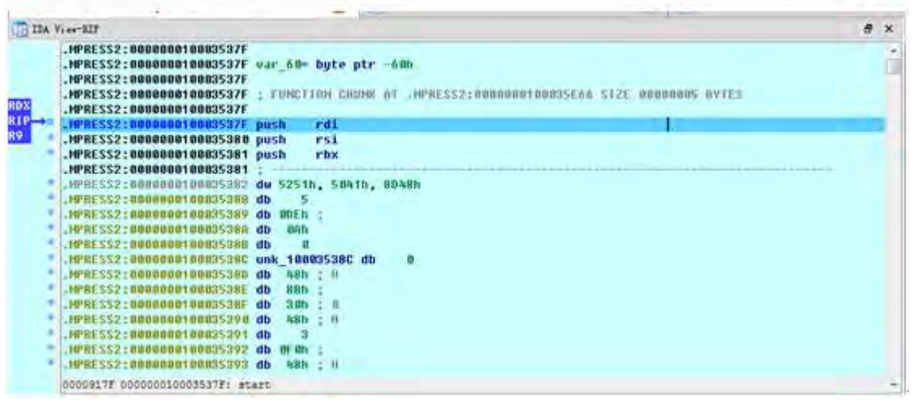


图 6

F8 单步执行到如下代码处之后一切就比较明朗了。

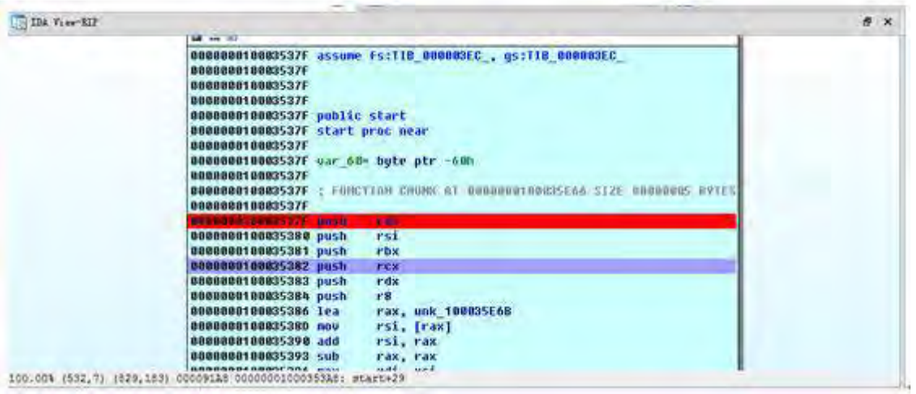


图 7

往下看可以看到一个跳转到不知道什么地方的 jmp 跳转，对于压缩壳比较熟悉的应该知道这个就是跳转到入口点的 Jmp 了。

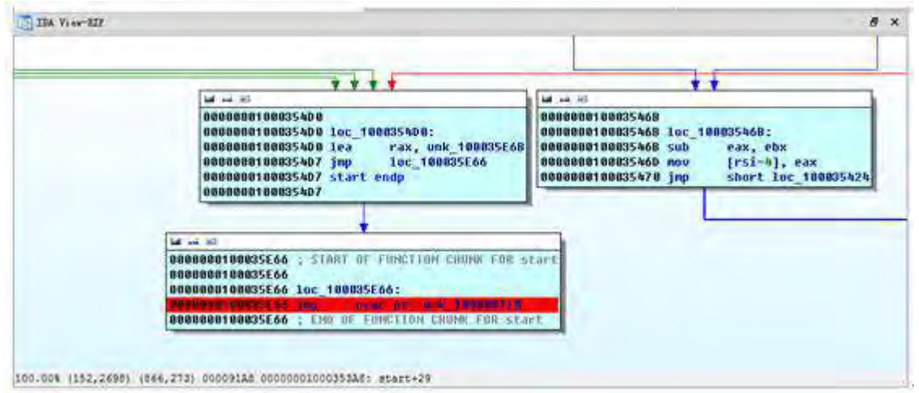


图 8

其实使用 ida 进行调试的最大的好处就是看到的代码会比较直观，程序的流程一目了然。而这些代码如果在 od 中查看的话可能直接单步调试的话会更好一些。

直接在 0000000100035E66 jmp near ptr unk_10000B770，一行设置断点然后 F9 运行即可。其实通过后面的地址也可以知道跳转到的地址 IDA 分析失败了，因而会显示 unk 前缀，如果跳转到这个地址看到的应该是一片乱码，如图 9 所示。

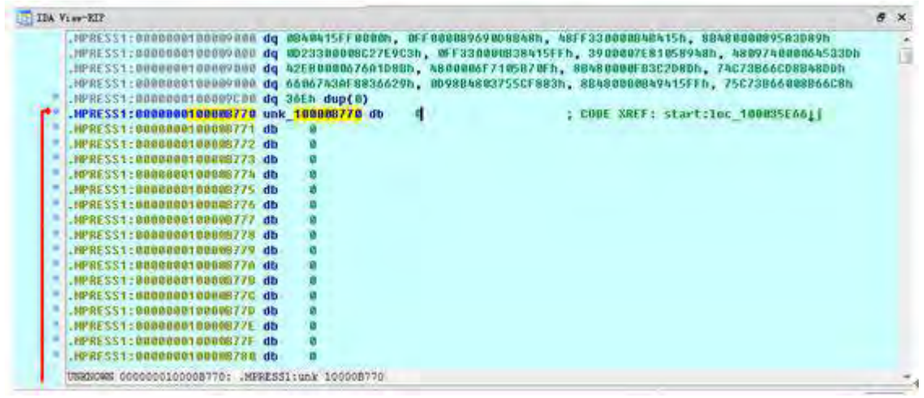


图 9

这是因为程序自身的代码还没有解压，所以看到的是一片空白的区域。直接 F9 执行到 0000000100035E66 jmp near ptr unk_10000B770 一行之后再进行 F8 单步一次就到达了程序的原始 OEP 了，如图 10

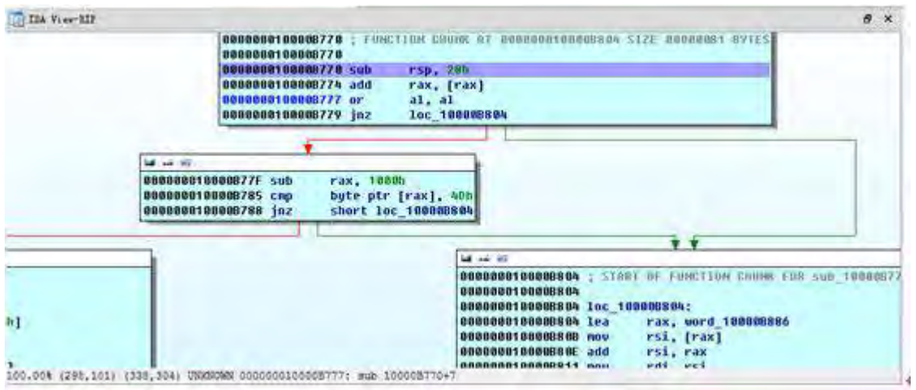


图 10

到这里就可以进行脱壳了，需要注意的是这里原来已经熟悉的 32 位的脱壳工具都不再适用了，需要使用 64 位的工具。运行 CHimpREC 64，程序界面如图 11 所示。

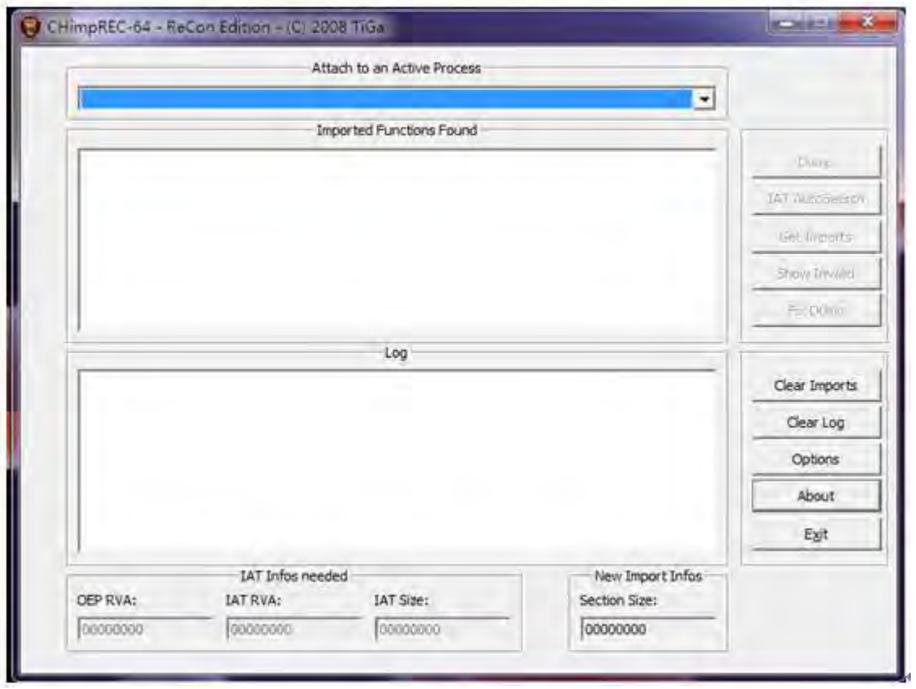


图 11

在附加进程中选择 Notepad.exe，选择之后首先要执行 dump 进行转存，这个比较简单一些。将内存镜像保存为 exe 即可。最后就是 IAT 的修复了，这里与 32 位系统的操作基本一致。

可以先点击 IAT auto search，如果查找成功将会出现如图 12 所示的提示信息。

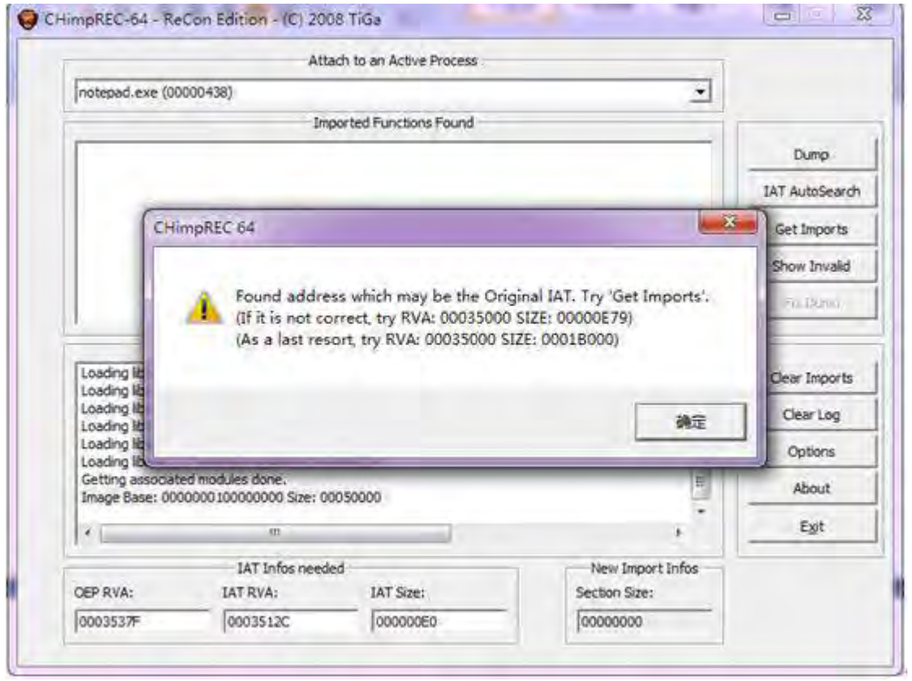


图 12

然后点击 Get Imports 获取输入表。

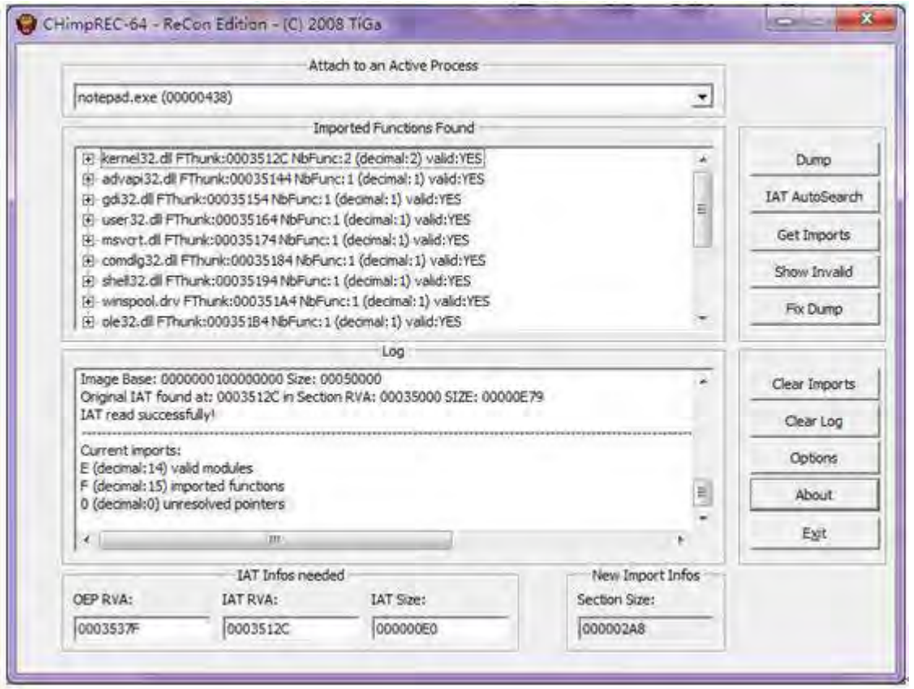
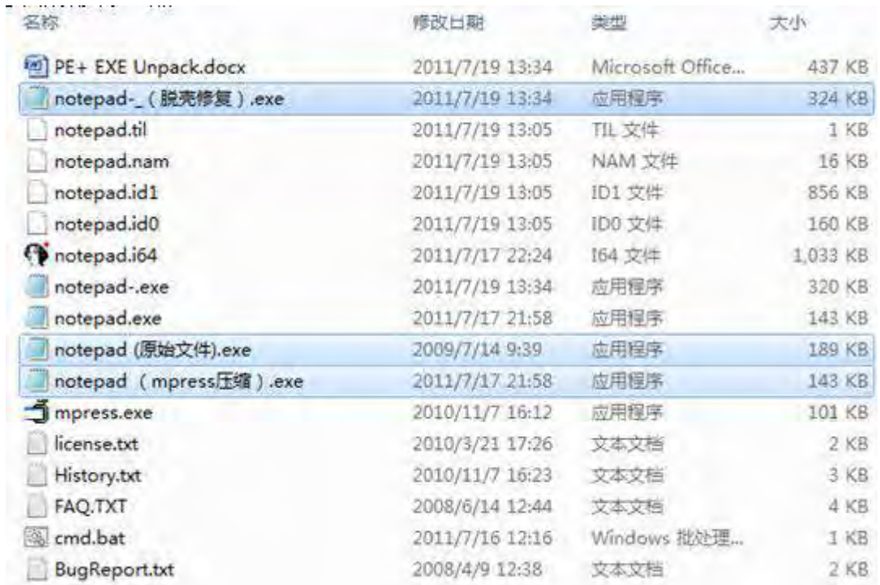


图 13

点击 show invalid 去掉或者修复无效的函数，最后 fix dump 即可。脱壳之后的程序比原始程序稍微大了一点。



名称	修改日期	类型	大小
PE+ EXE Unpack.docx	2011/7/19 13:34	Microsoft Office...	437 KB
notepad-_（脱壳修复）.exe	2011/7/19 13:34	应用程序	324 KB
notepad.til	2011/7/19 13:05	TIL 文件	1 KB
notepad.nam	2011/7/19 13:05	NAM 文件	16 KB
notepad.id1	2011/7/19 13:05	ID1 文件	856 KB
notepad.id0	2011/7/19 13:05	ID0 文件	160 KB
notepad.i64	2011/7/17 22:24	I64 文件	1,033 KB
notepad-.exe	2011/7/19 13:34	应用程序	320 KB
notepad.exe	2011/7/17 21:58	应用程序	143 KB
notepad (原始文件).exe	2009/7/14 9:39	应用程序	189 KB
notepad （mpress压缩）.exe	2011/7/17 21:58	应用程序	143 KB
mpress.exe	2010/11/7 16:12	应用程序	101 KB
license.txt	2010/3/21 17:26	文本文档	2 KB
History.txt	2010/11/7 16:23	文本文档	3 KB
FAQ.TXT	2008/6/14 12:44	文本文档	4 KB
cmd.bat	2011/7/16 12:16	Windows 批处理...	1 KB
BugReport.txt	2008/4/9 12:38	文本文档	2 KB

图 14

但是程序正常运行还是没有问题的的。呵呵。说实话本文没什么技术含量，让大家见笑了。

原创文章，转载请注明： *转载自 火星信息安全研究院*

本文标题： *《IDA + Debug 插件 实现 64Bit Exe 脱壳》*

本文链接地址： *http://www.h4ck.org.cn/2011/07/unpack-64bit-exe-via-ida-debug-plugin/*

Applied Reverse Engineering with IDA Pro

This editorial is committed to subverting the essential security restriction mechanisms of a native binary executable by employing the IDA Pro Dissembler. This paper is basically elaborating a very complex mechanism of reverse engineering among the previously demonstrated papers, yet because it is a very exhaustive and long process, it claims epic proportions of patience and proficiencies in Machine code instructions — but it is very interesting and challenging. We have mostly focused on .NET reverse engineering so far, which is relatively an easy task, instead of native binary reversing, because the source code is straightforwardly manipulated by dissemblers. This article would surely assist the aspirants who are repeatedly seeking a step by step tutorial on IDA Pro reverse engineering, because no such paper is perfectly crafted in laborious fashion so far.

This article showcases the particulars of these contents:

Live Binary Sample Target

Target Analysis with IDA Pro

Cracking the Target

Alternative way of Tracing

Final Note

There are ‘n’ numbers of approaches for reverse engineering, and picking the appropriate one depends on the target program, the platform on which it runs and on which it was

developed, and what kind of information you're seeking to extract. Generally speaking, there is one fundamental reversing methodology: *offline analysis,* which is all about taking a binary executable and using a disassembler to convert the machine code into a human-readable form. Reversing is then performed by manually reading and analyzing parts of that output. Offline code analysis is a powerful approach because it provides a good outline of the program and makes it easy to search for specific functions that are of interest.
The disassembler is one of the most significant reverse engineering apparatuses. Essentially, a disassembler decodes binary machine code into a readable assembly language code. The disassembler merely decodes each instruction and creates a textual representation for the code. It is trivial to say, the specific instruction encoding format and the resulting textual representation are entirely platform-specific. Each platform provides a different set of instructions and registers. Therefore a disassembler is also platform-specific (even though there are a couple of disassemblers that contain specific support for multiple platforms).

Essential

The target file
IDA Pro Interactive Dissembler
PE signature verifier
Assembly Language skills

Live Binary Sample Target

This time, I have chosen a target binary which is being applied over reverse engineering, and its origin is in fact totally unknown to us. This executable basically first validates the user identity by asking the password. If user enters the correct information, then he would be able to proceed; otherwise it echoes the wrong password message over the screen. But we have obtained this binary from some unauthentic sources. Thus we are not provided with the user password token key which would probably have part of a license key. Luckily, we have only the executable, not the source code, so that we can figure out the mechanism implemented behind the scenes.

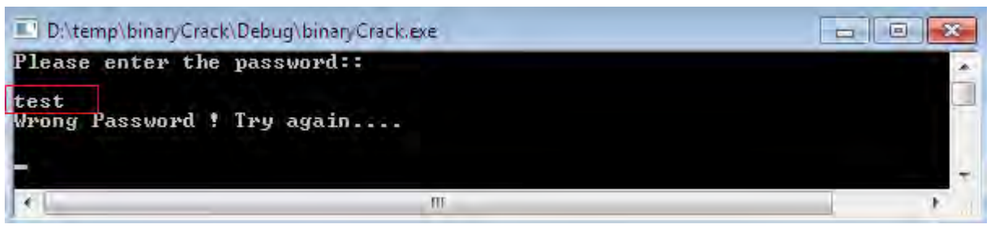


Figure 1.1
So the only possibility left to carry on without buying the license code is to reverse engineer this binary file using IDA Pro dissembler. It is indispensable to confirm either this target binary is a standard Windows PE file or belongs to some other platforms, because it is mandatory for a binary file to have the PE file signature; otherwise, IDA Pro won't dissemble it. So to achieve this, we can assist *PE Explorer* to obtain the signature information of this binary as follows in the file Header section.

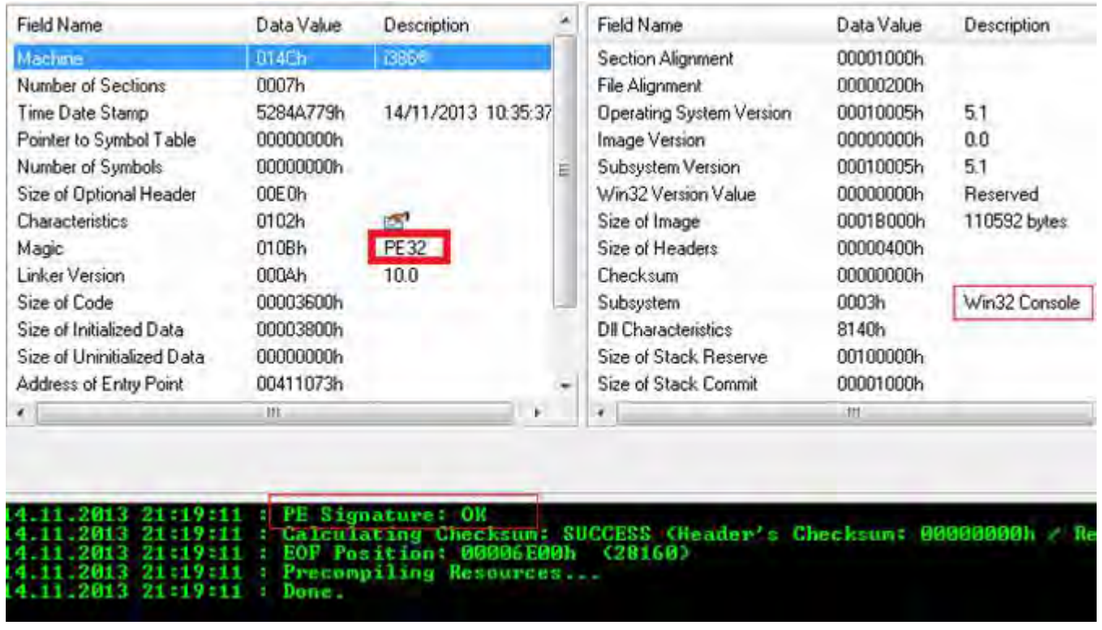


Figure 1.2

The aforesaid figure clearly expresses that this binary has a Windows PE signature and hence could be dissembled by IDA Pro, which we are going to elaborate in the next section.

Target Analysis with IDA Pro

Well, we have obtained the origin information of the target file so far using PE explorer. As we have stated earlier, reversing with IDA Pro is truly a laborious task, because we have to encounter trivial machine code. We don't have the source code, rather only the binary executable. However, we first decompile or dissemble the binary using IDA Pro in order to comprehend what mechanics are implemented implicitly. Thus, launch the IDA Pro software, and it will ask to choose the prototype of a new disassemble file:

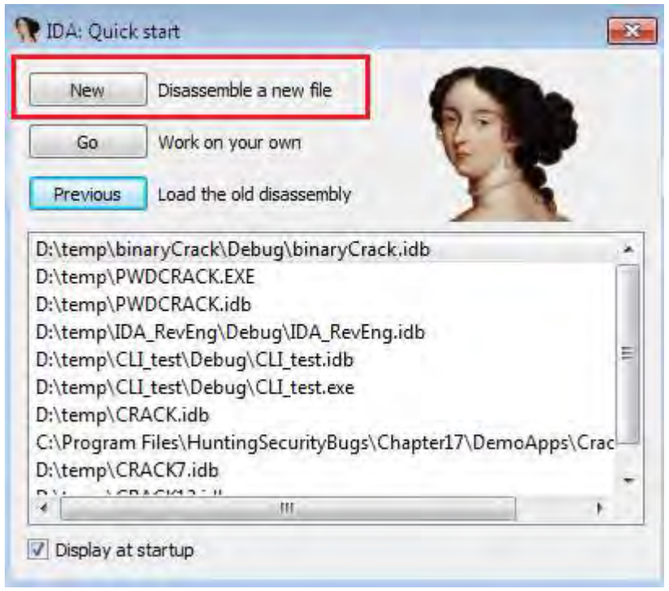


Figure 1.3

After configuring the disassemble prototype as a new project, the IDA prompts to open the target binary as in the following figure.

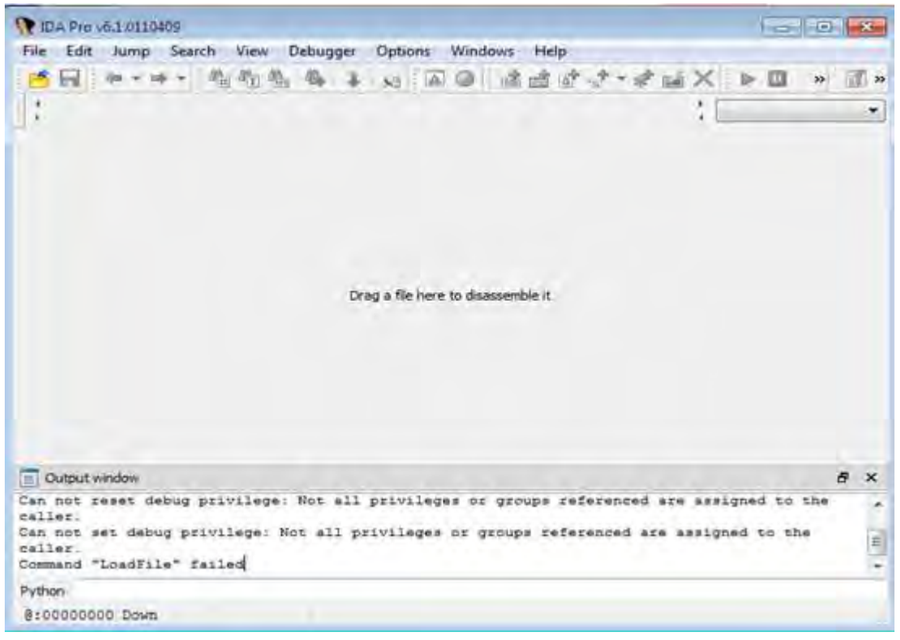


Figure 1.4

Right after choosing the target file, IDA Pro displays a screen dialog which stated three type of a file to be reversing as PE File, DoS Executable File, and Binary file. These file types basically point out the platform on which they were developed. In our scenario, the PE file is best fit-in because we have chosen a Windows 32 console application

as per the figure 1.2; Apart from that, we can organize other option such as processor type, kernel and DLL renaming as follows:

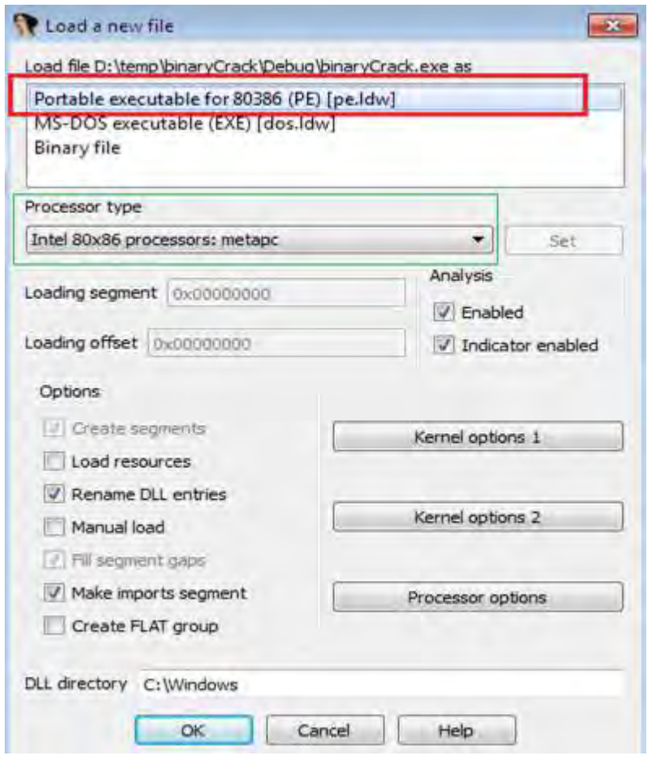


Figure 1.5

Finally, click OK. Don't bother ahead because IDA Pro prompts a couple of alert messages, open in two different dialog windows. On the whole, much internal processing is done before opening a target file.

As you can grasp the RED BOX in the following figure 1.6, we have a function window which enumerates all the methods and routines used in this executable. The graph window creates the control execution in flow chart format stated in the RED BOX. It provides a drag-able and moveable dashed rectangle box which can let us reach anywhere in the code execution.

In the BLUE BOX, it shows the decompile code in assembly code format and most importantly, we can access any segment of code such as entry point, containing text string, binary pattern and marked position just by dragging the pointer in the first RED BOX.

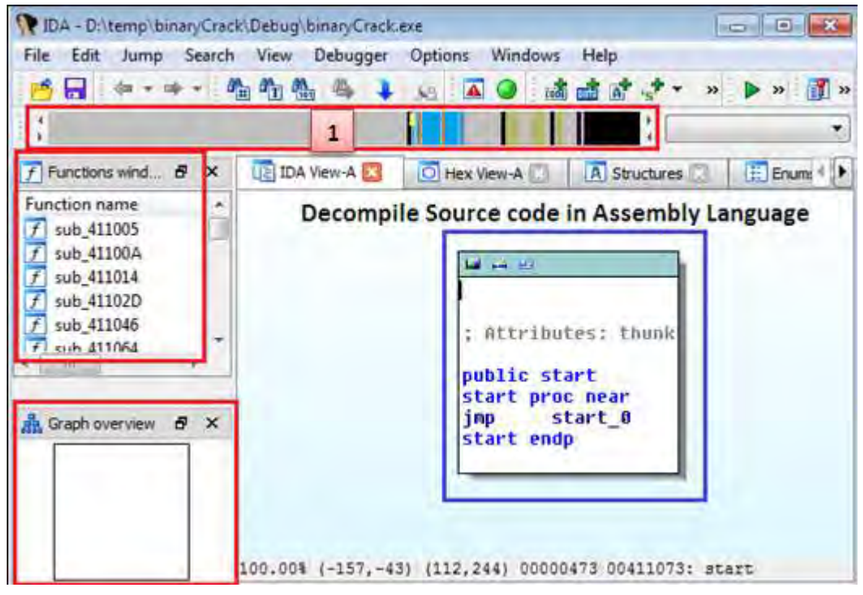


Figure 1.6

The important point to note here is that the *Debugger* menu would only be visible if the target file has the correct PE signature; otherwise it remains invisible. That's why is better to identify first the signature of the target file using PE explore. We shall accomplish the task of logic tracing by debugging the decompiled file. We however choose the appropriate debugger. In our case, we pick *Local Win32 debugger* as follows:

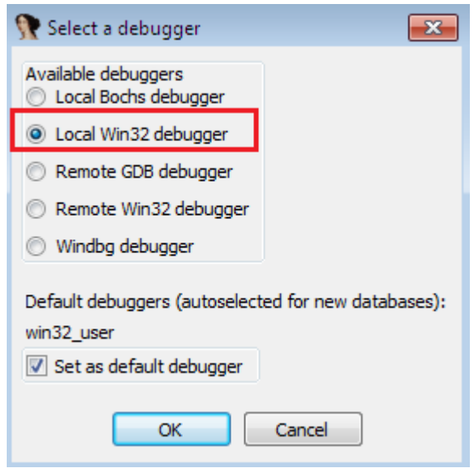


Figure 1.7

After ensuring the mandatory configuration, it is time to correlate with the actual mechanism by using the arrow mentioned in the following figure 1.8; it basically requires some hit and trail to find out the actual execution code path by dragging the pointer. The alert string message mentioned in figure 1.1 assist us to figure out the execution path. In fact, this target file shows numerous execution paths, and some of them are useful in the context of reversing, and the remaining ones are useless. So, the code flow that shows the alert message "*Enter the password*" is very significant to the reversing point of view, because this is the key value or entry point by which we can trace the essential code.

Figure 1.8

After moving the pointer to a specific location, we can find the actual mechanism logic flow as following. It typically shows the control flow when we enter the wrong password value.

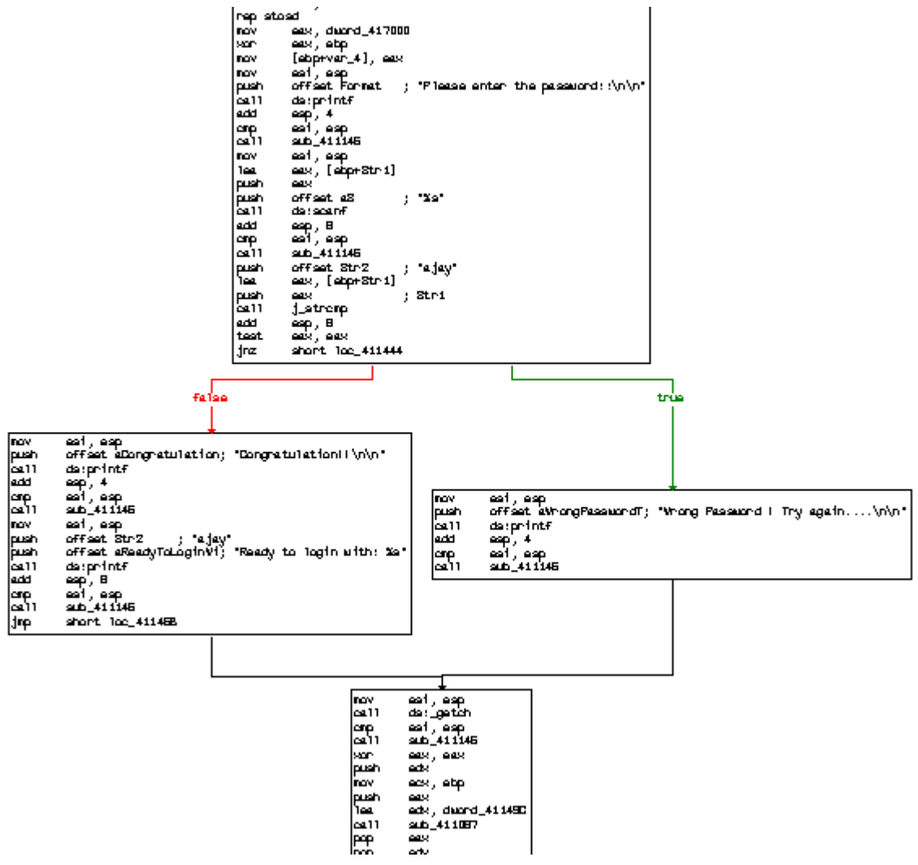


Figure 1.9

The logic path flow mentioned in the aforesaid figure usually does not fit in the work area window. For this purpose, we can move the dashed rectangle in the graph overview by dragging it to reach a specific segment as follows:

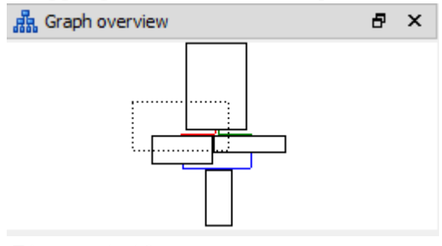


Figure 1.10

After moving the pointer to the appropriate location, we have found the decompiled code in assembly language format. Here, we can easily assume that this program prompts the user to enter the password by the *scanf* method mentioned in the RED BOX. Then this value is compared to a predefined string value (which is password) using the *strcmp* method. The test *eax* register is holding the value 0 or 1 which would come based on the string comparison. Finally, the *jnz* instructs the compiler to directly jump to the false segment branching, which is location 411444.

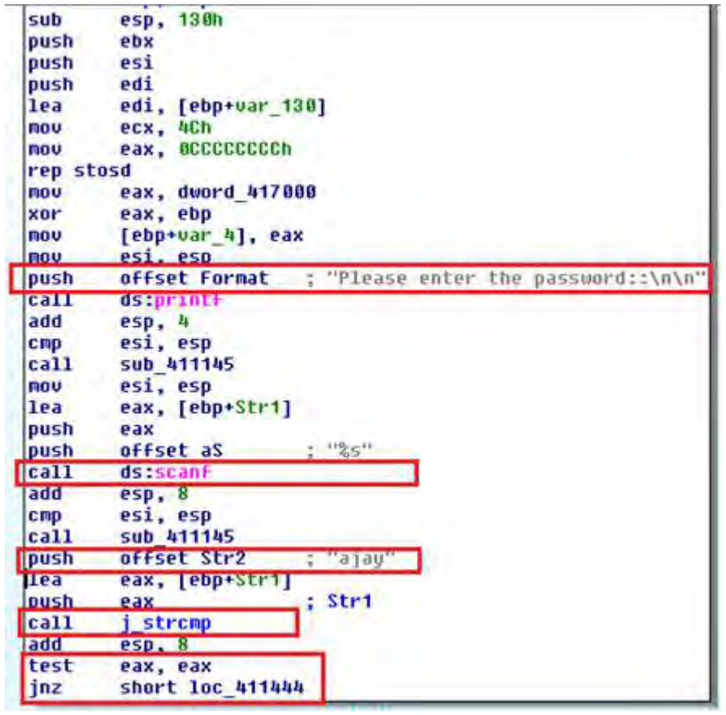


Figure 1.11

If the eax register contains the value 0, then the condition would be true and the code execution directed to the box highlighted by cyan color. If it has a value of 1, then the control flow diverts toward the false condition block as follows:

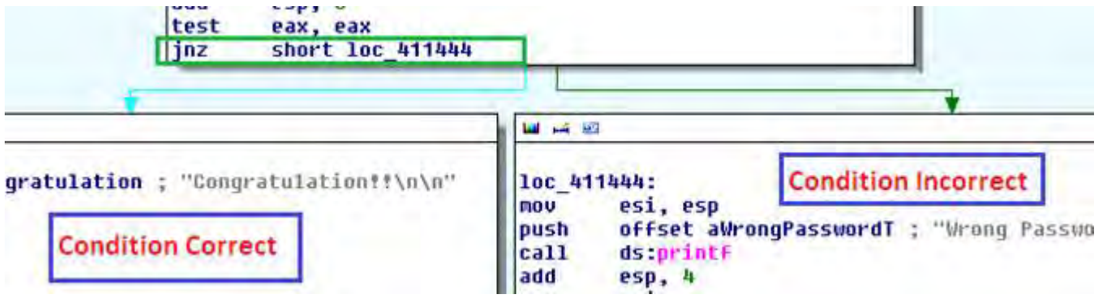


Figure 1.12

If the user enters the correct password, the following assembly code segment would be executed, in which first the "*congratulations*" message would be displayed along with the actual password information as follows:

```
mov     esi, esp
push    offset aCongratulation ; "Congratulation!!\n\n"
call    ds:printf
add     esp, 4
cmp     esi, esp
call    sub_411145
mov     esi, esp
push    offset Str2     ; "ajay"
push    offset aReadyToLoginWi ; "Ready to login with: %s"
call    ds:printf
add     esp, 8
cmp     esi, esp
call    sub_411145
jmp     short loc_41145B
```

Figure 1.13

As we have stated earlier, the code flow instructions are huge in quantity, so we have to move the dashed rectangle from time to time in order to reach the specific code block. This time we move to a false condition block as follows:

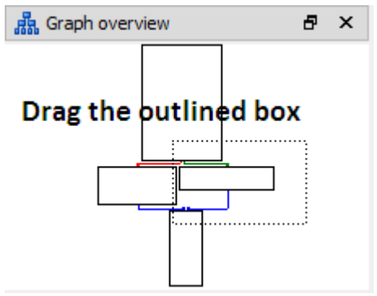


Figure 1.14

If the *eax* register doesn't contain the value 0, then the execution is diverted toward the following figure block where the "*Wrong Password! Try again*" message will be print on the screen as shown below.

```
loc_411444:
mov     esi, esp
push    offset aWrongPasswordT ; "Wrong Password ! Try again....\n\n"
call    ds:printf
add     esp, 4
cmp     esi, esp
call    sub_411145
```

Figure 1.15

Finally, no matter what value the *eax* register holds, the compiler always executes the following assembly instruction, where the *getch* method is encountered every time as follows:

```
loc_41145B:
mov     esi, esp
call    ds:_getch
cmp     esi, esp
call    sub_411145
xor     eax, eax
push    edx
mov     ecx, ebp
push    eax
lea     edx, dword_41149C
call    sub_411087
pop     eax
pop     edx
pop     edi
pop     esi
pop     ebx
mov     ecx, [ebp+var_4]
xor     ecx, ebp
call    sub_411014
add     esp, 130h
cmp     ebp, esp
call    sub_411145
mov     esp, ebp
pop     ebp
retn
sub_4113A0 endp
```

Figure 1.16

So, we have successfully disassembled the target assembly code to correlate to the actual mechanism running behind the scenes. We have come to a conclusion that the *eax* register value is the key hack. If its value is 0, then we entered into the true condition code block; otherwise we entered into the false condition block.

Cracking (Reversing) the Target

So, the *eax* register value would be the key interest for the reverser to subvert the password mechanism. If we change that value manually during debugging, then we can reach the true condition block even if we enter the wrong password information.
In this encounter, our leading objective is to collapse the jump statement (*jnz*) execution so that the calling of method loc_411444 won't happen. To do so, run this application in debugging mode. However, place a breackpoint at *eax* instruction by using F2. The instruction would be submerged in red box as follows:

```
call    j_strcmp
add     esp, 8
test    eax, eax
jnz     short loc_411444
```

Figure 1.17

Then, run this executable by Start Process (F9) from Debugger. Again, a couple of windows appeared, then disappeared as usual.

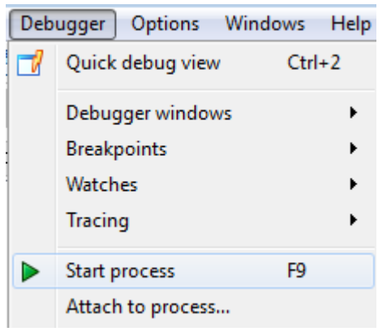


Figure 1.18

After that, the target starts to execute in DoS mode, because it is a console application. Here, it asks the user to enter the password as per its functionality, but unfortunately we don't have the password. So, just enter any value as a password and press enter.

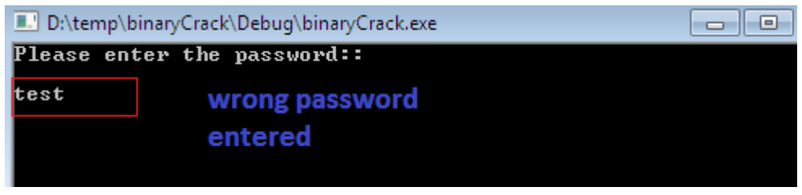


Figure 1.19

The moment we press enter after entering the bogus password value as *test*, the execution is transferred to the IDA View-A, and we reach the instruction where we placed the breakpoint earlier. Then we have to move ahead manually by pressing the step into (F7) and we step forward to the jump instruction. Here, we notice that the green arrow (in the RED BOX) started blinking, which points out that the execution is about to transfer to the B31444 method block. If we didn't do anything, the *wrong password* message display as usual.

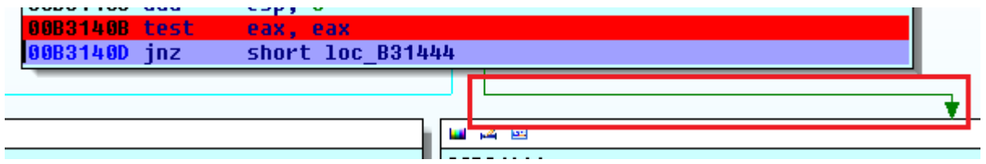


Figure 1.20

The execution is transferred to B31444 (false condition) block because *eax* register has a value as 1. The transmission to the false condition block happens due to the ZF value as 0. If this value would have 1, then the true condition block would execute.

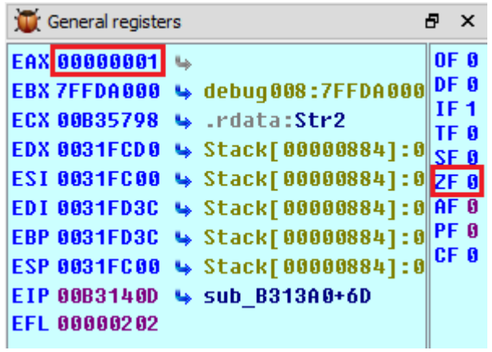


Figure 1.21

However, if we modify the value of ZF to 1, then we could control the code path execution to a true condition block. In order to change this value, right click over value and select *Modify value* as:

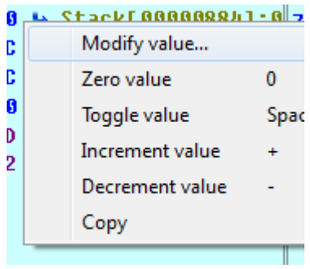


Figure 1.22

Here, just replace the ZF register value from 1 to 0 and click OK.

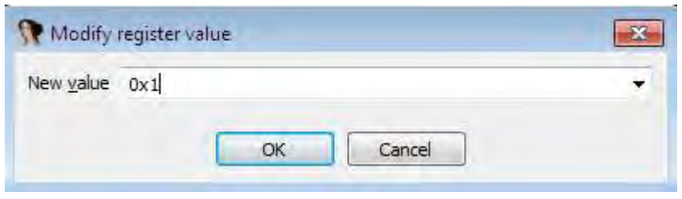


Figure 1.23

The moment the ZF register value is changed to 1, we notice that the execution flow is instantly changed to the true condition block by repeatedly blinking the red arrow in the BOX as follows:

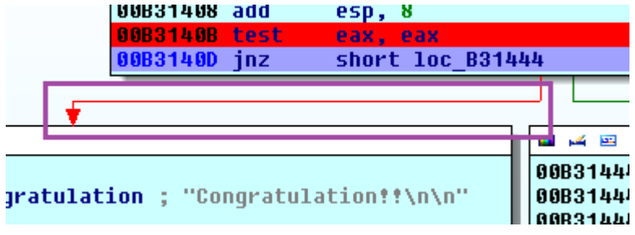


Figure 1.24

Now, go to the Debugger menu and press the *Run until return* option, which ends the execution after reaching the end of the code. Bingo!!! The target binary is showing the

congratulation message even if entering the wrong password value. In addition to that, this program shows the original password value as *ajay.*

Figure 1.25

So, we successfully subvert the password security mechanism just by diverting the binary code flow execution. But to get rid of a common misconception, we have to reverse engineer this target file, which is actually not patched. The password restriction is still in place, because we have not modified the corresponding bytes so far.

Alternate way of Tracing

Identification of the program entry point is always complex in IDA Pro because it shows raw assembly code. We have applied such a process by moving the arrow as in Figure 1.8. But this is a very cumbersome task. There is another way which eases the task of identifying such entry points.

A program displays some string to carry on the execution ahead or to assist the user to control the execution. Just consider the following figure: It asks the user to enter the password by prompting the “Please enter the password” string.

Figure 1.26

This string could be beneficial in terms of finding the entry points. So go to *Text search* and enter this string in the box, which finds such entries in the assembly code as follows:

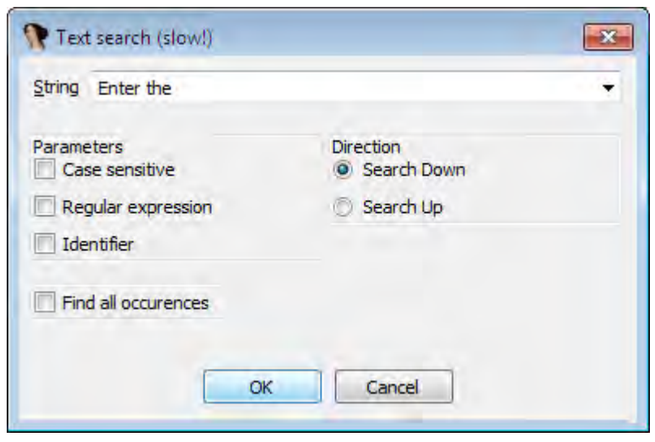


Figure 1.27

Alternatively, the string window displays all the in-built strings value of the target program. Just double click over the “Please enter the password” string, which redirects us to its assembly code.

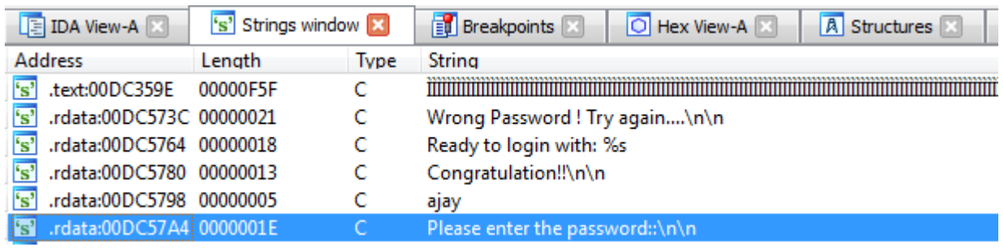


Figure 1.28

Here, we didn't find the code execution in flow chart format, but in actual assembly code. So we reached the location where the string is located. Place the breakpoint here by F2 as earlier. Therefore, follow the operation or steps as performed earlier from Figure 1.18:

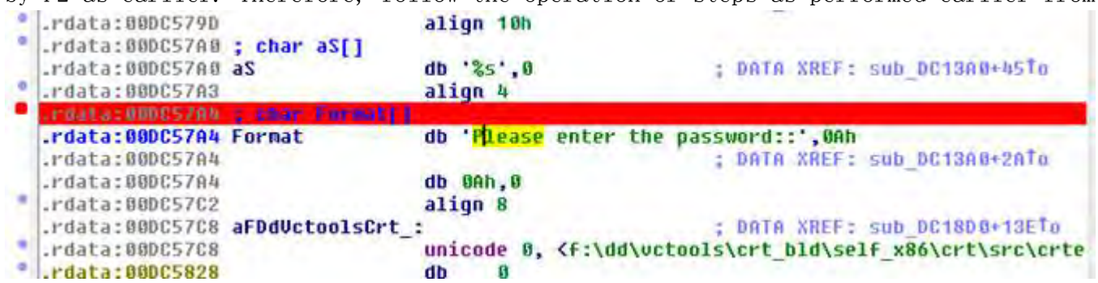


Figure 1.29

Final Note

This rare piece of information illustrated the process of disassembling as well as reverse engineering tactics over a native binary by using IDA Pro disassembler. We have seen the importance of the register values of binary code to correlate with actual program implementation and what role they can play in the reversing process. We have subverted the password mechanism just by modifying the value of the *ZF* register, which is connected to the *eax* register. The tutorial that applies to the binary target is dedicated to reversing the logic flows only, not patching the byte code associated with mechanism. There is no permanent change in memory related bytes; if we run this target again without IDA Pro, the password mechanism is still there, and it is not bypassed yet. We will discuss byte patching in detail in the next article.

http://resources.infosecinstitute.com/applied-reverse-engineering-ida-pro/#article

How to debug the DLL of an EXE using IDA Pro?

Very easy, if I got you right:

1. Make an Ida project from the DLL, i.e. drag and drop the dll into the blank Ida page.
2. In Menu Debugger, Process Options, put the path to your exe into the textbox "Application", Into "input file" put the path to your DLL. Confirm with OK.
3. Start with menu Debugger, Start Process or F9.

Your breakpoint should be hit.

Exactly what I was looking for. Is it possible for me to debug the DLL when the application is already running as a process?

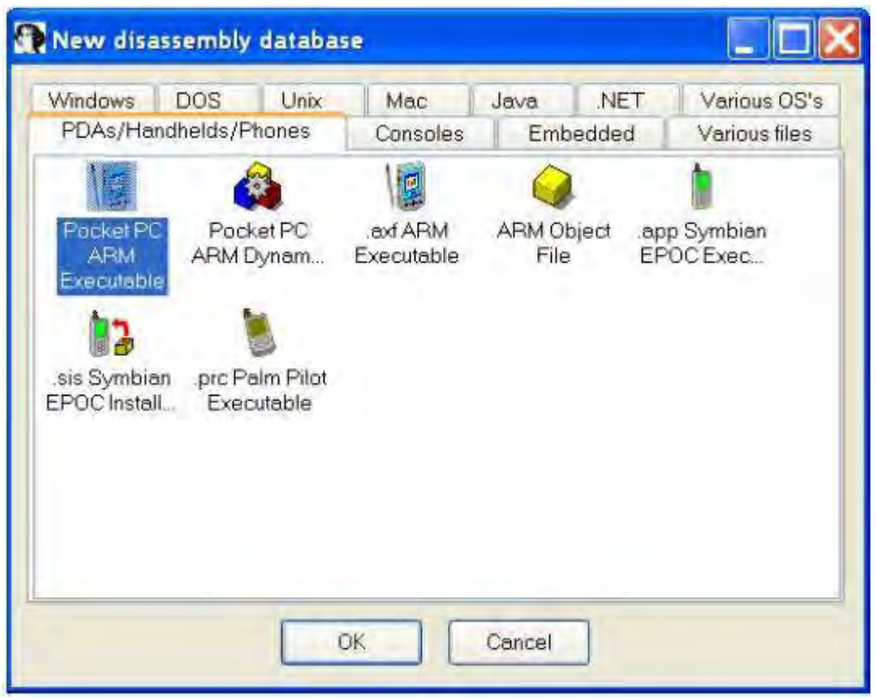
Menu Debugger, Attach to Process...

http://reverseengineering.stackexchange.com/questions/9328/how-to-debug-the-dll-of-an-exe-using-ida-pro

WPhone

IDA 教程-WINCE ARM 调试器入门教程

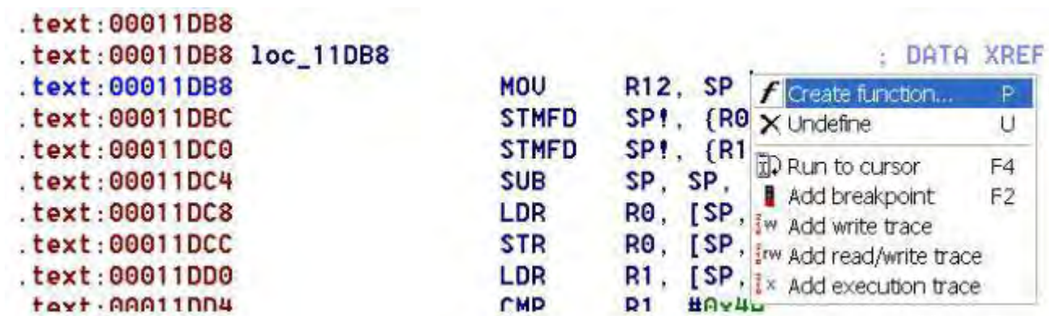
本教程将会展示我们如何对 Windows CE 设备的应用程序进行调试。示例文件可以从这里下载，这是一个 Windows CE 应用程序的示例，原型由 ljw004 创建（原始文件可在这里找到）。为了获得最好的反汇编效果，我们将输入文件拷贝到我们的 Windows 机器，并开始一个新的 PocketPC ARM Executable（File,New）：



经过向导页面后，我们接受所有默认设置，IDA 将会创建一个数据库来分析输入文件，
我们第一步工作是启动程序来查看它主窗口的 windows 进程.为了找到 windows 进程，我们
只需滚动鼠标找到下面的代码

```
.text:000125B4                 LDR     R3, =loc_11DB8
.text:000125B8                 STR     R3, [SP,#0xB0+WndClass.lpfnWndProc]
.text:000125BC                 LDR     R0, =unk_14424
.text:000125C0                 LDR     R1, [R0]
.text:000125C4                 STR     R1, [SP,#0xB0+WndClass.hInstance]
.text:000125C8                 MOV     R0, #0x40000005 ; int
.text:000125CC                 BL      GetSysColorBrush
.text:000125D0                 STR     R0, [SP,#0xB0+var_1C]
.text:000125D4                 LDR     R1, [SP,#0xB0+var_1C]
.text:000125D8                 STR     R1, [SP,#0xB0+WndClass.hbrBackground]
.text:000125DC                 LDR     R0, =aLumainclass_0
.text:000125E0                 STR     R0, [SP,#0xB0+WndClass.lpszClassName]
.text:000125E4                 ADD     R0, SP, #0xB0+WndClass ; lpWndClass
.text:000125E8                 BL      RegisterClassW
```

RegisterClassW()注册了一个新窗口类并指定了它的过程。IDA 显示为’loc_11DB8’，我
们在名称上双击跳到这里。为了更好的显示，我们在这里创建一个函数（鼠标右键的弹出菜
单中选择 Create function）：



我们将其重新命名如下(按 N 键):

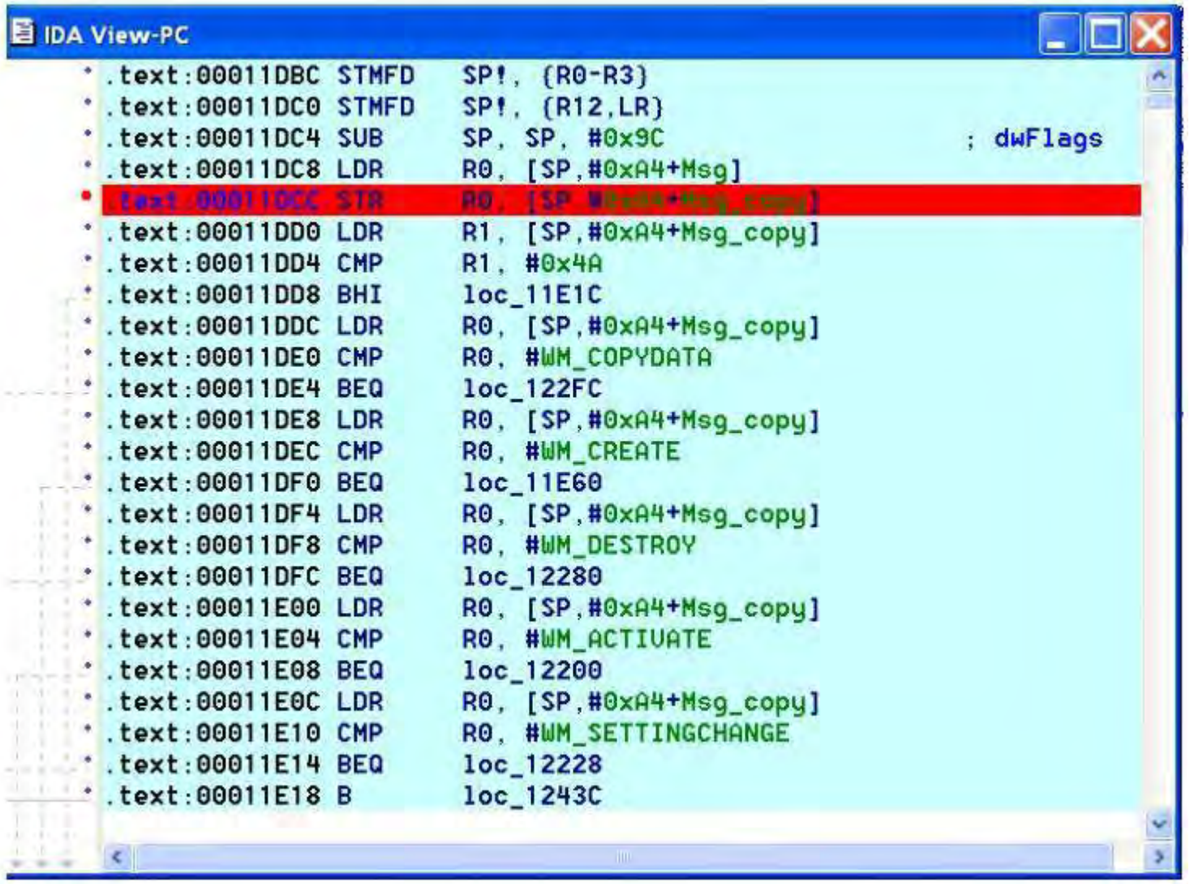
接着，我们在开始处按 F2 设置一个软件断点

```
.text:00011DB8
.text:00011DB8                  MOV     R12, SP
.text:00011DBC                  STMFD   SP!, {R0-R3}
.text:00011DC0                  STMFD   SP!, {R12,LR}
.text:00011DC4                  SUB     SP, SP, #0x9C   ; dwFlags
.text:00011DC8                  LDR     R0, [SP,#0xA4+Msg]
.text:00011DCC                  STR     R0, [SP,#0xA4+var_20]
.text:00011DD0                  LDR     R1, [SP,#0xA4+var_20]
.text:00011DD4                  CMP     R1, #0x4A
.text:00011DD8                  BHI     loc_11E1C
.text:00011DDC                  LDR     R0, [SP,#0xA4+var_20]
.text:00011DE0                  CMP     R0, #0x4A
.text:00011DE4                  BEQ     loc_122FC
.text:00011DE8                  LDR     R0, [SP,#0xA4+var_20]
.text:00011DEC                  CMP     R0, #1
```

现在我们准备启动调试器了。非常重要的一点是在启动调试器之前一定要在相应位置设置断点。为什么？因为，IDA 很难挂起（暂停）一个在 Windows CE 下运行的进程。如果一个进程在用户模式下执行，那么调试器可以通过在当前执行指令中设置一个断点来挂起它。

然而多数时候进程在执行系统代码，此时调试器无法挂起它，因为系统区域无法被修改。修改系统代码将会导致频繁崩溃或导致 PDA 挂掉，这就是为什么 IDA 不能让用户在系统区域设置断点的原因。当前所有在 0x80000000 地址以上的区域和 Core.dll 的代码都被保护。使我们按 F9 来启动调试器，或者通过菜单（Debugger,Start Process）。IDA 将会连接到 PDA，如果必要下载调试器服务端到 PDA 并查找输入文件。通常在 PDA 上的输入文件会被放置在一个与笔记本电脑不同的路径下，因此首次启动调试器时 IDA 无法找到它并会请求下载

按 Yes 后，IDA 将会下载并在 PDA 上执行这个程序。我们设置的断点会被马上触发



为使显示更清晰，我重命名了局部变量为”Msg_copy”，并使用 Windows 消息号替换了 16 进制数（使用 Edit,Operand types,Enum member 命令来处理）。现在如果你查看 PDA 的屏幕，仍然没有任何变化，我们的程序仍在挂起状态。我们按 F9 继续，断点会被立即重新触发-WndProc()被频繁调用，这很正常。我们希望通过 WndProc()了解 window 消息如何被处理，但我们不想每次都按 F9，因此我们考虑将我们断点做个简单的记录。我们右击鼠标并选择”Edit breakpoint”，在对话框中我们输入断点条件并清除’Break’选项框。

现在我们按 F9,程序会顺利执行不会中断，同时在 PDA 屏幕上显示它的主窗口。在 IDA 的信息窗口将会显示我们需要的 Windows 信息号：

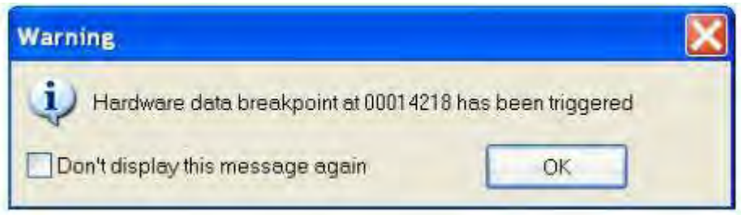
这就是创建和编辑软件断点的方法。你可以在 PDA 中随意操作示例程序，来看一下我们的记录断点如何工作的。你也可以创建更多的断点并浏览进程内存。整个进程的内存可以在调试器中获得。你可以通过 Ctro-S 查看内存的段信息。

现在我们再来看一下硬件断点。有时我们想知道何时一个特定的数据被修改。假定我们想知道在我们的示例程序中哪里访问了我们的文件名字：

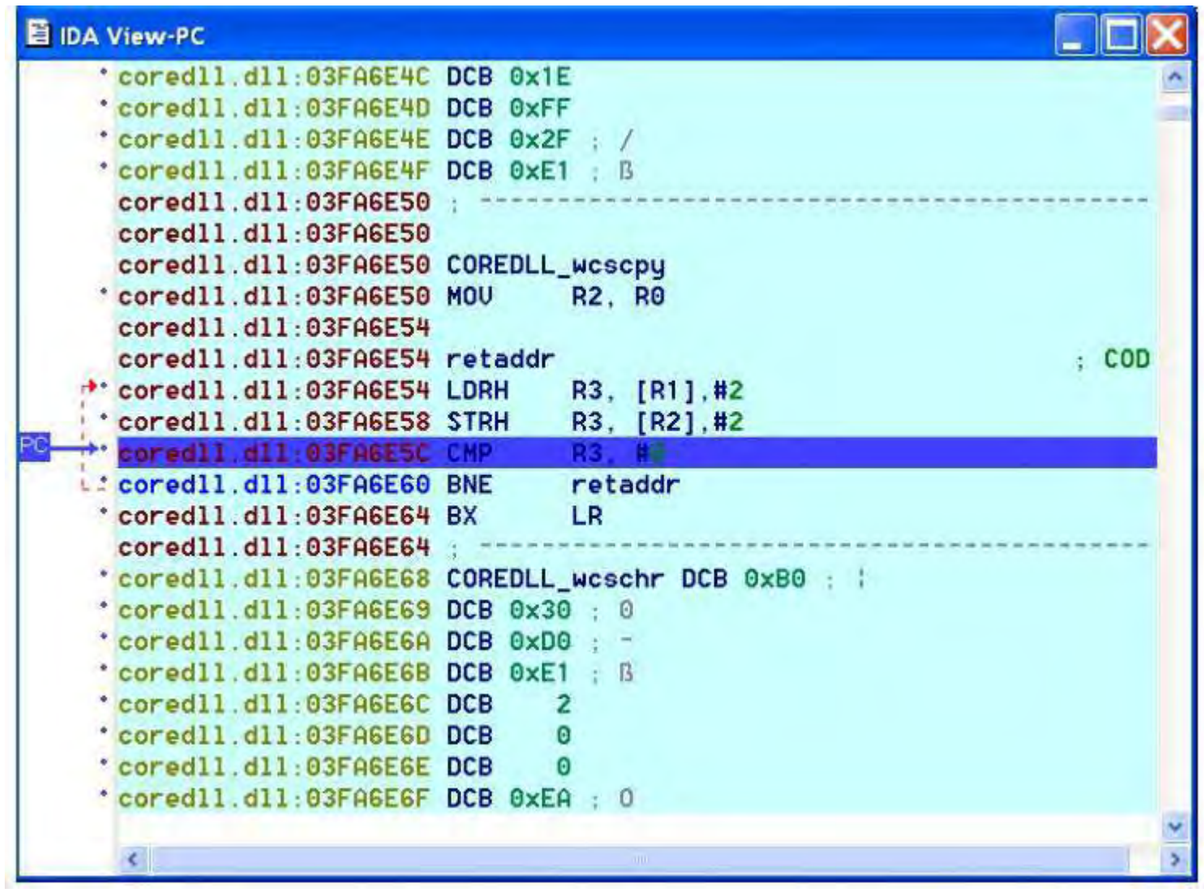
```
.data:00014218 aMyDocumentsMac unicode 0, <\My Document
.data:00014218
.data:00014218
.data:0001424A DCB     0
.data:0001424B DCB     0
.data:0001424C DCB     0
.data:0001424D DCB     0
```

我们通过按 F2 键（正如我们创建一个软件断点一样）来创建一个硬件断点。因为光标在数据区，不是代码区，IDA 会弹出下面的对话框：

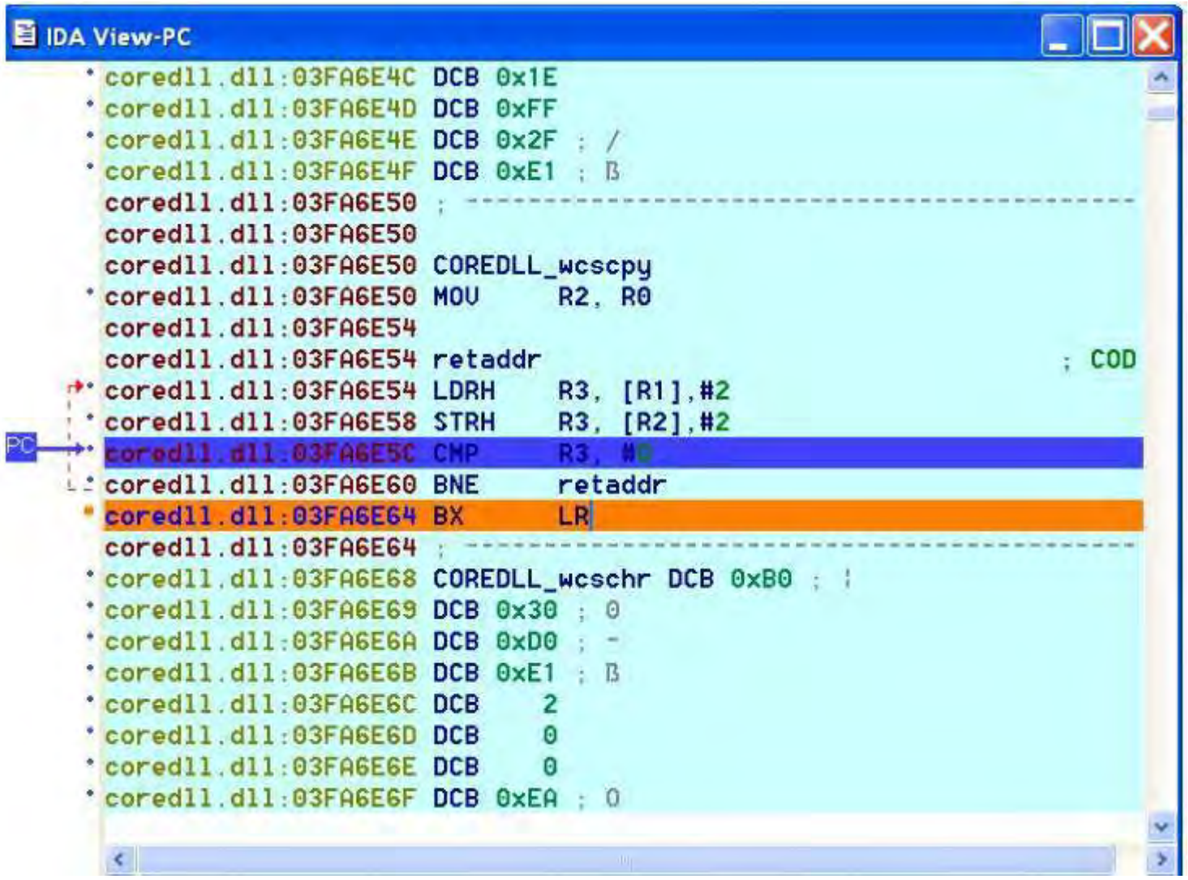
我们选择'Hardware breakpoint'选项框并选择的断点模式。我们对读写访问感兴趣。现在我们通过按 F9 回到应用程序(如果它处于挂起中)，并在 PDA 中操作应用程序。敲入文件名将会触发我们设置的断点。



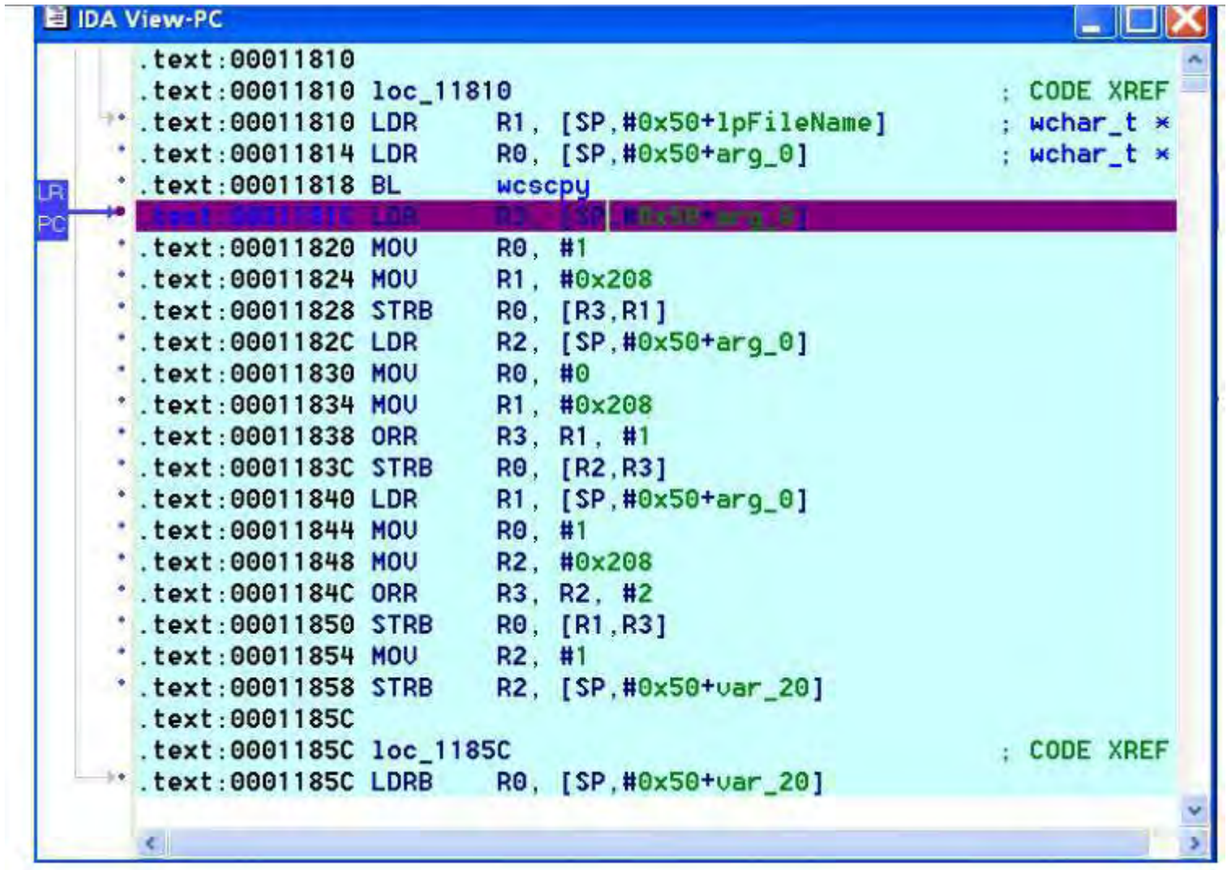
下面是我们激活断点时的代码：



我们看到是 coredll.dll 库中的 wcscpy()函数。因为它是个系统区域，我们不能在这里创建断点。即使我们尝试，我们所能得到如下：



请注意断点颜色，它不是红色而是橙色，这意味这这个断点不能被设置（可以通过 Options,Clolors,Debugger 命令设置断点颜色）。因为我们不能在这里设置断点，我们双击 LR 寄存器，它包含了返回地址：



幸运的是，返回地址在用户领空，我们可以在这里创建断点，并按 F9 以获得对程序的控制。这就是我们创建和使用硬件断点的方法。

http://blog.csdn.net/eqera/article/details/8239127

IDA Pro 5.0 动态调试 Smartphone 程序方法

花了差不多一天的时间（呵呵，大部分时间用在重装 VS 2005 上了），终于成功了，现在动态调试方法还在摸索中，以前看到很多 XD 说 IDA 动态调试 Microsoft Smartphone 程序不成功，花了点时间把方法整理出来，大家一起研究动态调试方法，相信论坛很多 XX 高手动态调试不在话下!!

本贴原本发表于 Smartphone 论坛，本论坛的大侠就不要笑偶了!!

下面附上方法：
原贴地址，更新可以关注此贴：http://www.ioicn.com.cn/bbs/thread-182856-1-2.html

一、准备工作：

1、系统要求：Windows XP SP2，SP1 系统没有测试；
2、安装同步软件 ActiveSync 3.8 或更高，我的是 4.2，软件自己去微软网站下或用 google 搜；
3、安装模拟器 Microsoft Device Emulator Version 1.0 或 Microsoft Visual Studio 2005，我用的是 Microsoft Visual Studio 2005 中带 的 Microsoft Device Emulator Version 1.0 中文版；

如果仅仅是用 IDA 动态调试，没有必要安装 Microsoft Visual Studio 2005，毕竟这东东太大，当时为了测试，我两者都安装了， 结果导致我原来安装的 Microsoft Visual Studio 2005 中的模拟器不能用了，执行修复时，找不到模拟器修复这一项，没有时间去研 究修复方法，只好全部重新安装了一遍，模拟器才能正常使用；

单独 Microsoft Device Emulator Version 1.0 模拟器下载地址，以前我下载的一个是 27M，现在微软件网站这个有 50 多 M，不知是不是加了 WM5 的 ROM 镜像：

微软件的，现在有 50 多 M
http://www.microsoft.com/downloads/d...displaylang=en

或
下面这个只有 27M
http://www.ioicn.com.cn/bbs/viewthread.php?tid=165946

4、下载 IDA Pro 5.0，下载地址本论坛，已经包含 CE 动态调试插件，菜单已经汉化：
电信 http://bbs.pediy.com/temp/IDA5.0.rar （57M）

网通 http://bbs1.pediy.com:8081/temp/IDA5.0.rar

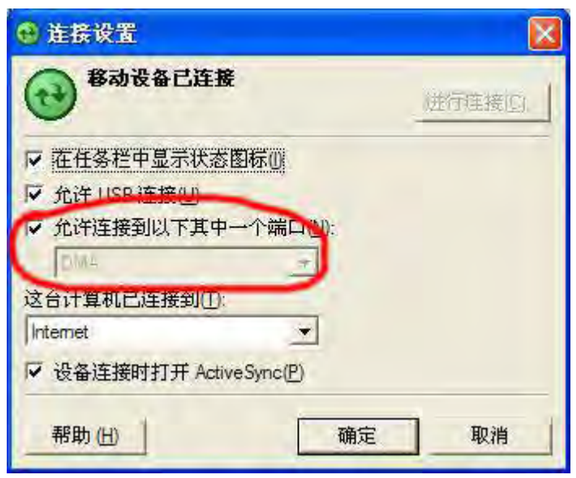
二、启动模拟器

1、VS 2005 模拟器启动方法：

1）先运行 VS 2005，再单击“工具”菜单——“连接到设备”，选择 Smartphone 2003，中的相关菜单；

2）启动模拟器管理器，单击“工具”菜单——“设备访真器管理器”；

3）设置同步软件 ActiveSync，右击任务栏图标，选择“连接设置”，在弹出的窗口中的“允许连接到以下其中一个端口”选择“DMA”，这一步很重要，一定要选择 DMA；

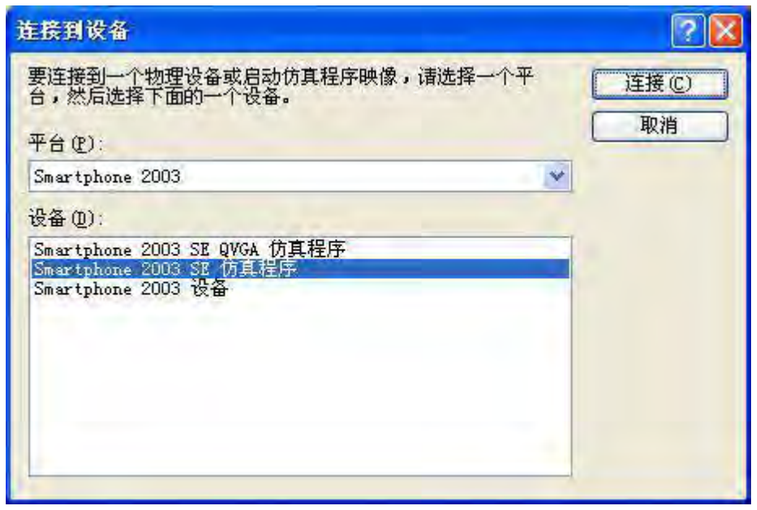


4）打开设备访真器管理器窗口，按刷新按钮，可以看到“Smartphone 2003”的“Smartphone 2003 SE 仿真程 序”变成一个绿色的播放按钮图标，右击该仿真器，选择“插入底座”，此时应该可以看到任务栏的同步图标开始变绿，一

会就弹出同步模式窗口，可以选择“标准 同步”或“来宾方式”，我选择“来宾方式”也可以动态调试成功；

5）对手机模拟器进行解锁，这也是很多网友动态调试不成功的根本原因，开始我没有做这一步，打开 IDA，装载 SP 程序后，点运行时，似乎进入了 一个死循环，不停的提示正在更新 CE 调试器服务，于是想到了是不是要解锁呢，死马当活马医，反正是模拟器，没什么影响，谁知道执行解锁程序（其实就是 SDA 的解锁工具，下载附件解压后，有一个“SDA_ApplicationUnlock.exe”文件）后，模拟器还没有重启，IDA 那个死循环窗口马 上消失了，提示我程序没有复制到手机，点确定，竟然成功了!! 呵呵!

6）动态调试方法，先用 IDA 装载程序后，进行静态分析，找到关键点设置断点，具体方法就不描述了，因为我也在摸索中。



2、Microsoft Device Emulator Version 1.0 模拟器中的方法：

声明，因为安装它，使我的 VS 2005 模拟器运行不起来，所以将它删除了，不过，应该在这个模拟器中也可以通过，只有让没有装 VS 2005 的兄弟测试了。

1）安装完 Microsoft Device Emulator Version 1.0 后，首先注册模拟器，单击“开始->所有程 序->Microsoft Device Emulator Preview”中的 Register Device Emulator 进行注册；

2）单击“开始->所有程序->Microsoft Device Emulator Preview”中的 Emulate Smartphone-WM 2003 SE(Cold Boot)模拟器，启动后，再关闭，系统会提示是否保存，保存即可。

3）单击“开始->所有程序->Microsoft Device Emulator Preview”中的 Emulate Smartphone-WM 2003 SE(Restore)模拟器；

4）单击“开始->所有程序->Microsoft Device Emulator Preview”中的 Device Emulator Manager，运行模拟器管理程序；

5）模拟器同步，方法同 VS 2005 模拟器的第 3、4 步；

6）模拟器中文支持，安装中文字体，否则中文软件在模拟器中显示全部是“口”，同步成功后，将中文字体文件（下载附件中的文件解压后有一个名为 “Chinese.mpx200.2K3.cab”的文件）复制到模拟器中安装，安装后需要重启，重启的方法要注意，具体我记不清了，应该是软重启，方法 有两个，一个是在模拟的菜单中选择软件重启，另一个是在模拟器管理中右击启动的模拟器，选择重启，但确切是哪一个，我不记得了，很久以前用这个模拟器时用 过，请大家自己测试了，重启后如果没有安装的字体文件，说明重启方法不对，请重复此步骤；

7）解锁，同 VS 2005 模拟器的第 5 步；

8）动态调试方法，同 VS 2005 模拟器的第 6 步。

注：

单独的 Microsoft Device Emulator Version 1.0 模拟器是英文版的，自带的 SP 和 PPC 模拟镜像文 件也是英文的，在微软网站我没有找到中文版下载，VS 2005 中的也没有找到单独安装的文件，各位找到的说一声。因为是英文模拟器，所以不能显示程序中 的中文，必须安装中文字体，试验时我用 VS 2005 的 Smartphone 2003 SE 的中文镜像文件替换英文模拟器的镜像文件，可以正常启动，但 按键不正常，最主要的两个按键，左软键和右软键，按的时候变成了“1”和“2”，“*”号键和“#”变成了“0”和“9”，我对比看了一下两个模拟器的 XML 配置文件，按键定义完全一样，但格式稍有不同，VS 2005 中的模拟器配置文件“Smartphone_2003_Skin.xml”换行比较 少，而单独模拟器中的，基本上每一个语句都换了行，而且两个文件不能互换，我将 VS 2005 中的配置文件替换单独模拟器中的配置文件后，模拟器运行不起 来，有兴趣的朋友可以研究一下按键不正常的原因，我不想再重装 VS 2005 了!!

因论坛限制，附件无法上传，需要到我的网盘下：
http://kwzlj.ys168.com，在 XX 目录下，文件名："Chinese.rar"
http://bbs.pediy.com/showthread.php?t=37454

IDA 远程调试 WINCE 程序的环境搭建

标 题: 【原创】IDA 远程调试 WINCE 程序的环境搭建
作 者: csjwaman
时 间: 2012-08-28,22:02:20
链 接: http://bbs.pediy.com/showthread.php?t=155236

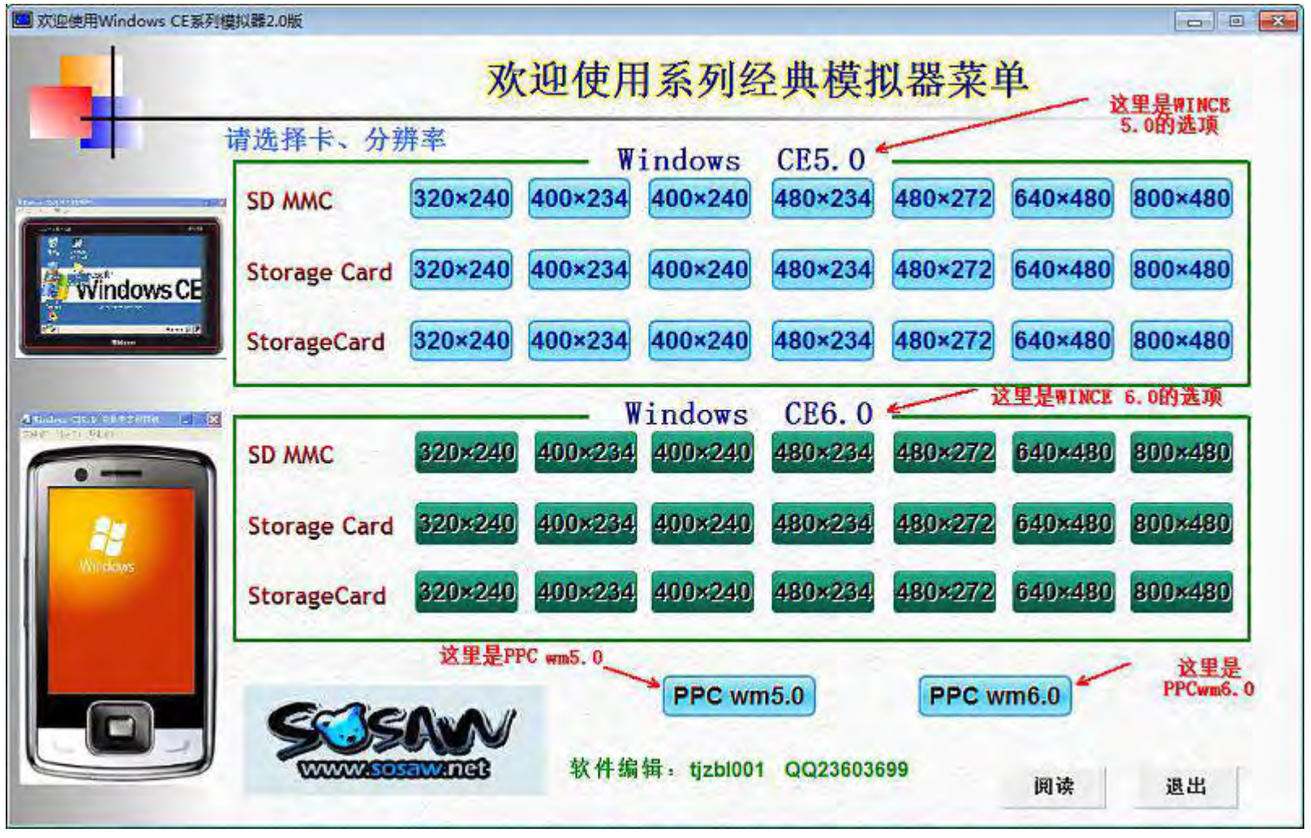
IDA 远程调试 WINCE 程序的环境搭建

网上有很多有关 IDA 远程调试 WINCE 程序的方法介绍。但在提到搭建相应的环境时，均说需要完整安装 VS2005。而 VS2005 体积太大，安装既不方便也不经济。本人经过摸索，找到一个非常简便的方法，供各位参考。

一、调试环境的搭建

(搭建平台时最好选择 WINCE5.0,因 WINCE6.0 有内存保护，无法进入 DLL 地址空间。)

1.首先下载 Windows CE 系列模拟器（百度一下，有很多下载链接。），运行“连接加载.bat”进行模块注册，然后运行 wince.exe。



2.选择一项打开。如果选择 WINCE5.0，还要在 WINCE 里作连接设置。如下图：


CMMB 数字电视
GT-4322
回收站
我的设备
Internet Explorer
Media Player
Microsoft
Windows CE
www.WinceHome.com
程序(P)
收藏(A)
文档(D)
设置(S)
帮助(H)
运行(R)...
关机(U)
控制面板(C)
网络和拨号连接(N)
任务栏和开始菜单(T)...
选择“网络和拨号连接”
开始
12:00
Shinco


CMMB 数字电视
GT-4322
文件(F) 编辑(E) 查看(V) 高级(N)
新建连接
1. 双击这里
新建连接
键入连接名称(T):
我的连接
选择连接类型(S):
拨号连接(D)
直接连接(I)
虚拟专用网络(PPTP)(V)
虚拟专用网络(L2TP)(R)
以太网上的 PPP [PPPoE](P)
2. 选择这里
< 上一步(B)
下一步(N) >
3. 点击这里
开始
网络连接
12:03
Shinco


CMMB 数字电视
GT-4322
文件(F) 编辑(E) 查看(V) 高级(N)
新建连接
设备
我的连接
选择设备(S):
Serial over DMA
Serial Cable on COM1:
Serial over DMA
选择DMA设备
TCP/IP 设置(T)...
安全设置(E)...
< 上一步(B)
完成
开始
网络连接
12:07
Shinco


CMMB 数字电视
GT-4322
文件(F) 编辑(E) 查看(V) 高级(N)
新建连接
我的连接
选择这里
程序(P)
收藏(A)
文档(D)
设置(S)
帮助(H)
运行(R)...
关机(U)
控制面板(C)
网络和拨号连接(N)
任务栏和开始菜单(T)...
开始
网络连接
12:09
Shinco


CMMB 数字电视
GT-4322
文件(F) 查看(V)
1. 双击这里
Internet 选项
PC 连接
笔针
电源
键盘
密码
区域设置
日期/时间
删除程序
输入面板
鼠标
所有者
网络和拨号连接
系统
显示
音量和声音
正在拨号
PC 连接属性
PC 连接
这些设置控制 Windows CE 设备 与台式计算机之间的连接。警告:
调整这些设置可能会导致无法与台式计算机连接。
设备在线时允许与台式计算机建立连接(A)
2. 点击这里
连接方式:
``桌面 @ 19200``
更改(C)...
开始
网络连接
控制面板
PC 连接属性
12:11
Shinco


CMMB 数字电视
GT-4322
文件(F) 查看(V)
Internet 选项
PC 连接
笔针
电源
键盘
密码
区域设置
日期/时间
删除程序
输入面板
鼠标
所有者
网络和拨号
系统
显示
音量和声音
正在拨号
更改连接
连接到台式计算机时使用:
`桌面 @ 19200`
`桌面 @ 19200`
我的连接
警告: 更改此设置可能会导致无法与台式计算机连接。
选择“我的连接”
连接方式:
``桌面 @ 19200``
更改(C)...
开始
网络连接
控制面板
PC 连接属性
12:12
Shinco

(WINCE 6.0 下由系统直接设置成“我的连接”，不需要手工设置。但调试时无法跟踪进 DLL 空间，建议选择 WINCE5.0 调试。)

3.最后安装 Microsoft ActiveSync 程序，WIN7 需要安装 6.1 版本的，WIN7 以下安装 4.5 版本。

二、设置调试环境

1.启动 Microsoft ActiveSync 等待仿真器连接。

2.运行“连接启动.exe”，弹出“设备仿真器管理器”窗口。

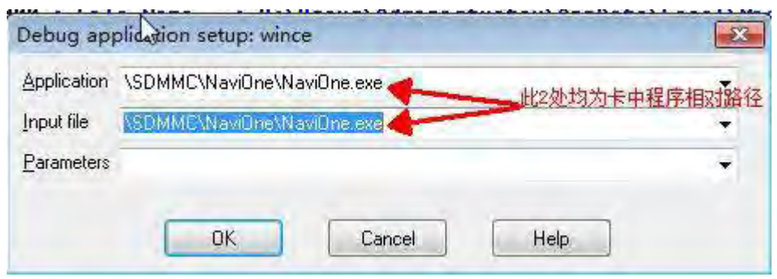
4.“插入底座”后 Microsoft ActiveSync 设置成 DMA 连接，将自动连接。

三、启动 IDA 进行调试

1.启动 IDA 后，File->Open

2.Debugger->Select Debugger….

3.Debugger->Process Options….



4.设置好相应的断点后就可以开始调试了。

用 IDA 调试 wince 灵图 13GPS 程序

标 题: 【原创】用 IDA 调试 wince 灵图 13GPS 程序
作 者: cecial
时 间: 2015-03-05,16:17:21
链 接: http://bbs.pediy.com/showthread.php?t=198414

初学 IDA 调试 WINCE 程序。分享下我的经验。
这是一个 WINCE 下的导航程序。用模拟器运行时会提示获取卡 ID 错误。
用字符串搜索也搜不到。
百度了一下，DeviceIocontrol 子程序一般拿来获取 SD 卡信息的，就从这里入手。

看图。

IDA View-A | Hex View-A | Structures | Enums | Imports | Exports

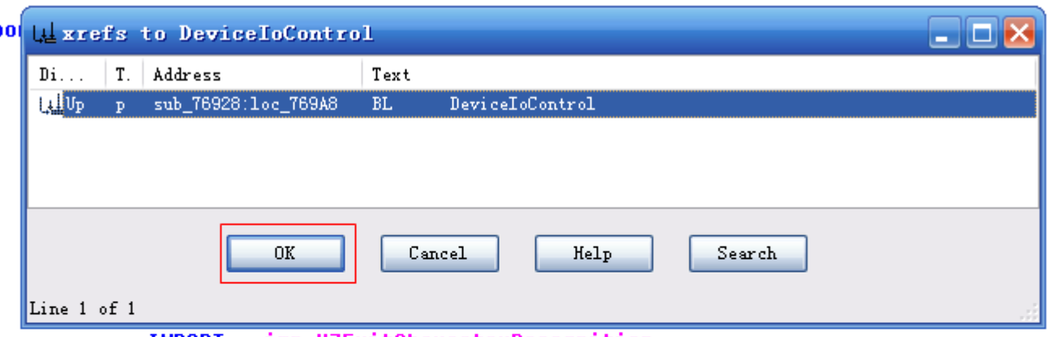
Address	Ordinal	Name	Library
001F00B8	1095	??2@YAPAXI@Z	COREDLL
001F00B0	1094	??3@YAXPAX@Z	COREDLL
001F005C	260	BeginPaint	COREDLL
001F0018	275	BringWindowToTop	COREDLL
001F02B0	1611	ChangeDisplaySettingsEx	COREDLL
001F01D0	553	CloseHandle	COREDLL
001F0214	164	CopyFileW	COREDLL
001F0030	910	CreateCompatibleDC	COREDLL
001F0028	90	CreateDIBSection	COREDLL
001F0244	160	CreateDirectoryW	COREDLL
001F017C	168	CreateFileW	COREDLL
001F01CC	555	CreateMutexW	COREDLL
001F01E4	1238	CreateSemaphoreW	COREDLL
001F0290	492	CreateThread	COREDLL
001F0098	246	CreateWindowExW	COREDLL
001F004C	264	DefWindowProcW	COREDLL
001F027C	3	DeleteCriticalSection	COREDLL
001F0024	911	DeleteDC	COREDLL
001F0210	165	DeleteFileW	COREDLL
001F0038	912	DeleteObject	COREDLL
001F0068	265	DestroyWindow	COREDLL
001F02B4	179	DeviceIoControl	COREDLL
001F0084	859	DispatchMessageW	COREDLL
001F0054	261	EndPaint	COREDLL
001F01C4	4	EnterCriticalSection	COREDLL
001F0138	180	FindClose	COREDLL

双击

```
.idata:001F02B0                                         ; .text:off_17CE40↑o
.idata:001F02B4 ; BOOL __stdcall DeviceIoControl(HANDLE hDevice, DWORD dwIoControlCode, LPV
.idata:001F02B4                 IMPORT __imp_D                          loControl↑o
.idata:001F02B4
.idata:001F02B8 ; HLOCAL __stdcall LocalAlloc(
.idata:001F02B8                 IMPORT __imp_L                          ↑o
.idata:001F02B8
.idata:001F02BC ; UINT __stdcall LocalSize(HLO
.idata:001F02BC                 IMPORT __imp_L
.idata:001F02BC
.idata:001F02C0
.idata:001F02C4 ;
.idata:001F02C4 ; Imports from HZRecog.dll
.idata:001F02C4 ;
.idata:001F02C4                 IMPORT __imp_H                          ↑o
.idata:001F02C4
.idata:001F02C8                 IMPORT __imp_H                          ↑o
.idata:001F02C8
.idata:001F02CC                 IMPORT __imp_H
.idata:001F02CC                                                         cterRecogni↑
.idata:001F02CC
.idata:001F02D0                 IMPORT __imp_H
.idata:001F02D0                                         ; DATA XREF: HZCharacterRecognize↑o
```

- 重命名 (N) — N
- 跳到操作数 (J) — Enter
- 跳到新窗口 (W) — Alt+Enter
- 在一个新的十六进制窗口内跳转 (O)
- 跳转到参考选定操作数的交叉参考 (X)... — X
- 图表 - 交叉参考到 (P)...
- 图表 - 交叉参考来自 (Q)...
- 转换为数组 (Y)... — Num *
- 转换为数据 (D) — D
- 撤消转换 (U) — U
- Synchronize with ▸
- 添加断点 (A) — F2
- 添加写入跟踪 (T)
- 添加读取/写入跟踪 (R)
- 添加监视 (V)
- 复制地址到命令行 (Z)

```
.idata:001F02C4 ;
.idata:001F02C4 ; Impo
.idata:001F02C4 ;
.idata:001F02C4
.idata:001F02C4
.idata:001F02C8
.idata:001F02C8
.idata:001F02CC
.idata:001F02CC
.idata:001F02CC
.idata:001F02D0
.idata:001F02D0
.idata:001F02D0
.idata:001F02D4                 IMPORT __imp_HZExitCharacterRecognition
.idata:001F02D4                                         ; DATA XREF: HZExitCharacterRecognition↑o
.idata:001F02D4                                         ; .text:off_14524C↑o
.idata:001F02D8
.idata:001F02DC ;
.idata:001F02DC ; Imports from WININET.dll
.idata:001F02DC ;
.idata:001F02DC ; HINTERNET __stdcall InternetConnectW(HINTERNET hInternet, LPCWSTR lpszServerName, INTERNET_PI
.idata:001F02DC                 IMPORT __imp_InternetConnectW ; DATA XREF: InternetConnectW↑o
```



```
STR     R7, [SP,#0x34+lpOutBuf] ; lpOutBuf
STR     R10, [SP,#0x34+lpOverlapped] ; lpOverlapped
STR     LR, [SP,#0x34+lpBytesReturned] ; lpBytesReturned
STR     R4, [SP,#0x34+nOutBufSize] ; nOutBufSize

loc_769A8
BL      DeviceIoCon
CMP     R0, #0
BEQ     loc_76A64
```

```
LDR     R3, [SP,#0x34+BytesReturned]
CMP     R3, #0
BEQ     loc_76A48
```

```
LDR     R2, [R7,#0xC]
MOV     R4, #0
MOV     R5, #0
CMP     R2, R3
BCS     loc_76A40
```

```
LDR     R0, =aSerialnoS ; "SerialNo = %s\r\n"
ADD     R6, R2, R7
```

查询储存卡ID失败，错误号%lu！\r\n"

- 组节点 (L)
- 重命名 (N) — N
- 跳到操作数 (J) — Enter
- 跳到新窗口 (W) — Alt+Enter
- 在一个新的十六进制窗口内跳转 (M)
- 跳转到参考选定操作数的交叉参考 (X)... — X
- 图表 - 交叉参考到 (O)...
- 图表 - 交叉参考来自 (P)...
- 手工输入 (U)... — Alt+F1
- 编辑函数 (E)... — Alt+P
- 隐藏 (H) — Num -
- 反汇编代码视图 (Q)
- 撤消转换 (T) — U
- Synchronize with ▸
- 执行到光标处 (C) — F4
- 添加断点 (A) — F2
- 添加写入跟踪 (V)
- 添加读取/写入跟踪 (R)
- 添加执行跟踪 (Y)
- 复制地址到命令行 (Z)

在000769A8断下来了。
按F8一步一步执行

```
00076980 SUB     R1, R3, #0x3DC  ; dwIoControlCode
00076984 MOV     R7, R0
00076988 ADD     LR, SP, #0x34+BytesReturned
0007698C MOV     R3, #0          ; nInBufSize
00076990 MOV     R0, R9          ; hDevice
00076994 MOV     R2, #0          ; lpInBuf
00076998 STR     R7, [SP,#0x34+lpOutBuf] ; lpOutBuf
0007699C STR     R10, [SP,#0x34+lpOverlapped] ; lpOverlapped
000769A0 STR     LR, [SP,#0x34+lpBytesReturned] ; lpBytesReturned
000769A4 STR     R4, [SP,#0x34+nOutBufSize] ; nOutBufSize
000769A8
000769A8 loc_769A8
000769A8 BL      DeviceIoControl
000769AC CMP     R0, #0
000769B0 BEQ     loc_76A64
```

```
00076AA0 BL      GetLastErr
00076AA4 ADD     SP, SP, #0
00076AA8 LDMFD   SP!, {R4-R
00076AA8 ; End of function
00076AA8
```

```
000769B4 LDR     R3, [SP,#0x34+BytesReturned]
000769B8 CMP     R3, #0
000769BC BEQ     loc_76A48
```

```
00076A64
00076A64 loc_76A64
00076A64 BL      GetLastError
00076A68 MOV     R1, R0
00076A6C LDR     R0, =aSPVCIid2GmALug ; "查询储存卡ID失败，错误号%lu
00076A70 BL      sub_74A34
00076A74 MOV     R0, R7          ; hMem
00076A78 BL      LocalFree
00076A7C MOV     R0, R9          ; hObject
```

1

执行到000769B0时，有一个跳转箭头会闪动，提示执行顺序

```
00076974 MOV     R0, #0x40       ; fuFlags
00076978 BL      LocalAlloc
0007697C MOV     R3, #loc_72000
00076980 SUB     R1, R3, #0x3DC  ; dwIoControlCode
00076984 MOV     R7, R0
00076988 ADD     LR, SP, #0x34+BytesReturned
0007698C MOV     R3, #0          ; nInBufSize
00076990 MOV     R0, R9          ; hDevice
00076994 MOV     R2, #0          ; lpInBuf
00076998 STR     R7, [SP,#0x34+lpOutBuf] ; lpOutBuf
0007699C STR     R10, [SP,#0x34+lpOverlapped] ; lpOverlapped
000769A0 STR     LR, [SP,#0x34+lpBytesReturned] ; lpBytesReturned
000769A4 STR     R4, [SP,#0x34+nOutBufSize] ; nOutBufSize
000769A8
000769A8 loc_769A8
000769A8 BL      DeviceIoControl
000769AC CMP     R0, #0
000769B0 BEQ     loc_76A64
```

```
000769B4 LDR     R3, [SP,#0x34+BytesReturned]
000769B8 CMP     R3, #0
000769BC BEQ     loc_76A48
```

```
00076A64
00076A64 loc_76A64
00076A64 BL      GetLastError
00076A68 MOV     R1, R0
00076A6C LDR     R0, =aSPVCIidZGmALug ; "查询储存卡ID失败，错误号%lu
00076A70 BL      sub_74A34
00076A74 MOV     R0, R7          ; hMem
00076A78 BL      LocalFree
00076A7C MOV     R0, R9          ; hObject
```

```
00076A94 MOV     R1, R0
00076A98 LDR     R0, =aCrea
00076A9C BL      sub_74A34
00076AA0 BL      GetLastErr
00076AA4 ADD     SP, SP, #0
00076AA8 LDMFD   SP!, {R4-R
00076AA8 ; End of function
00076AA8
```

```
R0 00000000
R1 9F790A38 debug069:9F790A38
R2 00000000
R3 00000000
R4 00000BB8
R5 002AA930 chsime03.dll:002AA930
R6 00284E20 chsime03.dll:00284E20
R7 00515F90 chsime03.dll:00515F90
R8 1A2BF87C debug085:1A2BF87C
R9 FF478CBE
R10 00000000
R11 00000000
R12 00000000
SP 1A2BF7C8 debug085:1A2BF7C8
LR 000769AC sub_76928+84
PC 000769B0 sub_76928+88
PSR 6000001F
```

```
MODE 1F
T 0
F 0
Q 0
V 0
C 1
Z 1
N 0
```

2 可以修改Z值改变执行顺序

把光标放到1前面，然后输入0就改变值了，程序就向另一个分支运行

Modules

Path

Threads

Decimal	Hex	State
-10...	BF...	Ready
527...	1F...	Ready

这里我修改了，
程序就走左边了
可以看到最下面的
窗口有输出

```
000769A0 STR     LR, [SP,#0x34+lpBytesReturned] ; lpBytesReturned
000769A4 STR     R4, [SP,#0x34+nOutBufSize] ; nOutBufSize
000769A8
000769A8 loc_769A8
000769A8 BL      DeviceIoControl
000769AC CMP     R0, #0
000769B0 BEQ     loc_76A64
```

```
000769B4 LDR     R3, [SP,#0x34+BytesReturned]
000769B8 CMP     R3, #0
000769BC BEQ     loc_76A48
```

```
00076A64
00076A64 loc_76A64
00076A64 BL      GetLastError
00076A68 MOV     R1, R0
00076A6C LDR     R0, =aSPUCIidZGmALug ; "查询储存卡ID失败，错误号%lu
00076A70 BL      sub_74A34
00076A74 MOV     R0, R7          ; hMem
00076A78 BL      LocalFree
00076A7C MOV     R0, R9          ; hObject
00076A80 BL      CloseHandle
00076A84 BL      GetLastError
00076A88 ADD     SP, SP, #0x14
00076A8C LDMFD   SP!, {R4-R10,PC}
```

```
000769C0 LDR     R2, [R7,#0xC]
000769C4 MOV     R4, #0
000769C8 MOV     R5, #0
000769CC CMP     R2, R3
```

```
00076A30
00076A30 loc_76A30
00076A30 ADD     R4, R4, #1
00076A34 LDRB    R3, [R6,R4]
00076A38 CMP     R3, #0
00076A3C BNE     loc_769F4
```

```
00076A40
00076A40 loc_76A40
00076A40 SUB     R3, R4, R5
00076A44 STRB    R10, [R3,R8]
```

```
00076A48
00076A48 loc_76A48                ; hMem
00076A48 MOV     R0, R7
00076A4C BL      LocalFree
00076A50 MOV     R0, R9           ; hObject
00076A54 BL      CloseHandle
00076A58 MOV     R0, #0
00076A5C ADD     SP, SP, #0x14
00076A60 LDMFD   SP!, {R4-R10,PC}
```

一直执行到子程序返回。
窗口显示SD卡序列号为空，因为是模拟器嘛。
如果走右边就会现在查询SD id失败

Output window

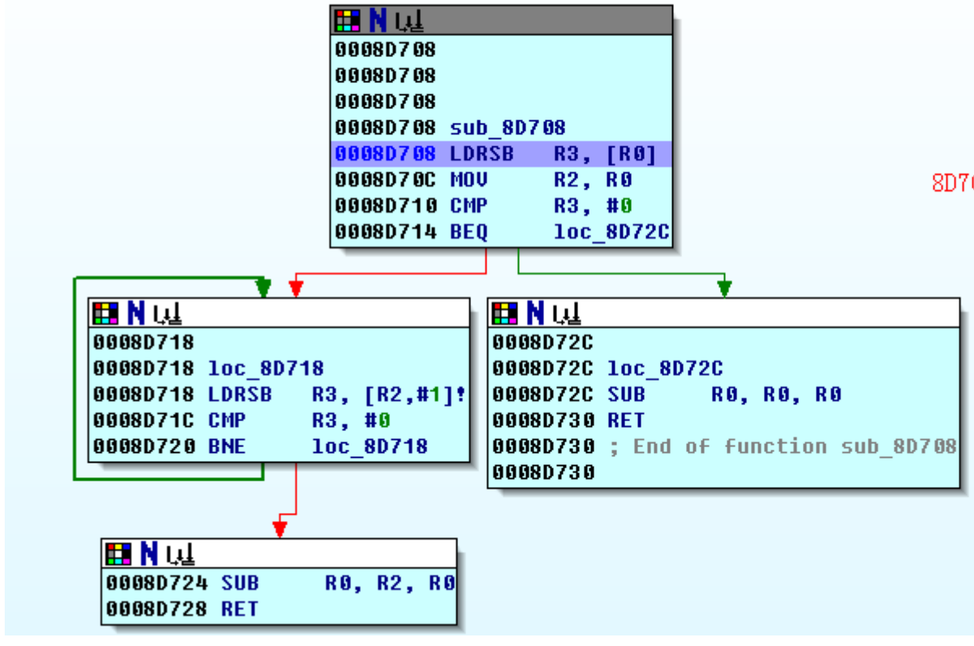
```
Debugger: loaded hzrecog.dll
Debugger: loaded \Windows\wininet.dll
Debugger: loaded \Windows\shlwapi.dll
Debugger: loaded \Windows\oleaut32.dll
Debugger: loaded \Windows\ws2.dll
Debugger: loaded \Windows\ole32.dll
Debugger: loaded \Windows\coredll.dll
Debugger: loaded \Windows\chsime03.dll
Debugger: loaded travelinfo.wince.dll
Debugger: unloaded travelinfo.wince.dll
75498: thread has started (tid=527196114)
75498: thread has started (tid=527192154)
75498: thread has started (tid=1600913330)
Debugged application message: ScInit completed
```

查询SD卡ID是在sub_76928中运行的。
一般查询完，要做的就是比较啦。我们猜一下，sub_8D708可能是比较的过程。
完了以后，在BEQ 000179C4中就是成功，失败的关键了。
我们先进8D708看看，光标在这行的时候，F7进入。（如果F8，是直接把这个子程序执行完了）

```
00017984 MOV     R1, #0          ; int
00017988 MOV     R8, R0
0001798C BL      j_memset
00017990 MOV     R2, R5
00017994 MOV     R1, R4
00017998 MOV     R0, R8
0001799C BL      sub_175498
000179A0 CMP     R4, R0
000179A4 BNE     loc_17D14
```

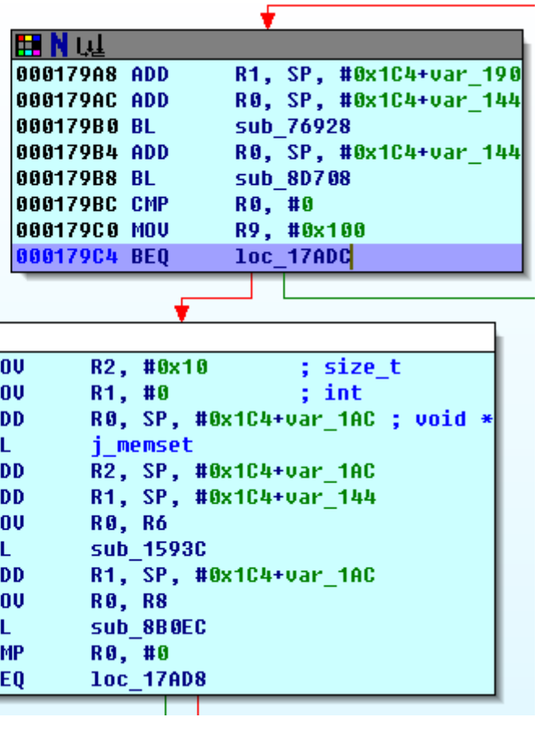
```
000179A8 ADD     R1, SP, #0x1C4+var_190
000179AC ADD     R0, SP, #0x1C4+var_144
000179B0 BL      sub_76928
000179B4 ADD     R0, SP, #0x1C4+var_144
000179B8 BL      sub_8D708
000179BC CMP     R0, #0
000179C0 MOV     R9, #0x100
000179C4 BEQ     loc_17ADC
```

```
000179C8 MOV     R2, #0x10       ; size_t
000179CC MOV     R1, #0          ; int
000179D0 ADD     R0, SP, #0x1C4+var_1AC ; void *
000179D4 BL      j_memset
000179D8 ADD     R2, SP, #0x1C4+var_1AC
000179DC ADD     R1, SP, #0x1C4+var_144
000179E0 MOV     R0, R6
```



8D708看不出什么，执行到返回吧

来到179C4，光标提示走右边，我们试着改Z值，
1改为0，走左边看看。
改完可以直接按F9，让程序跑起来，而不是一步一步



2

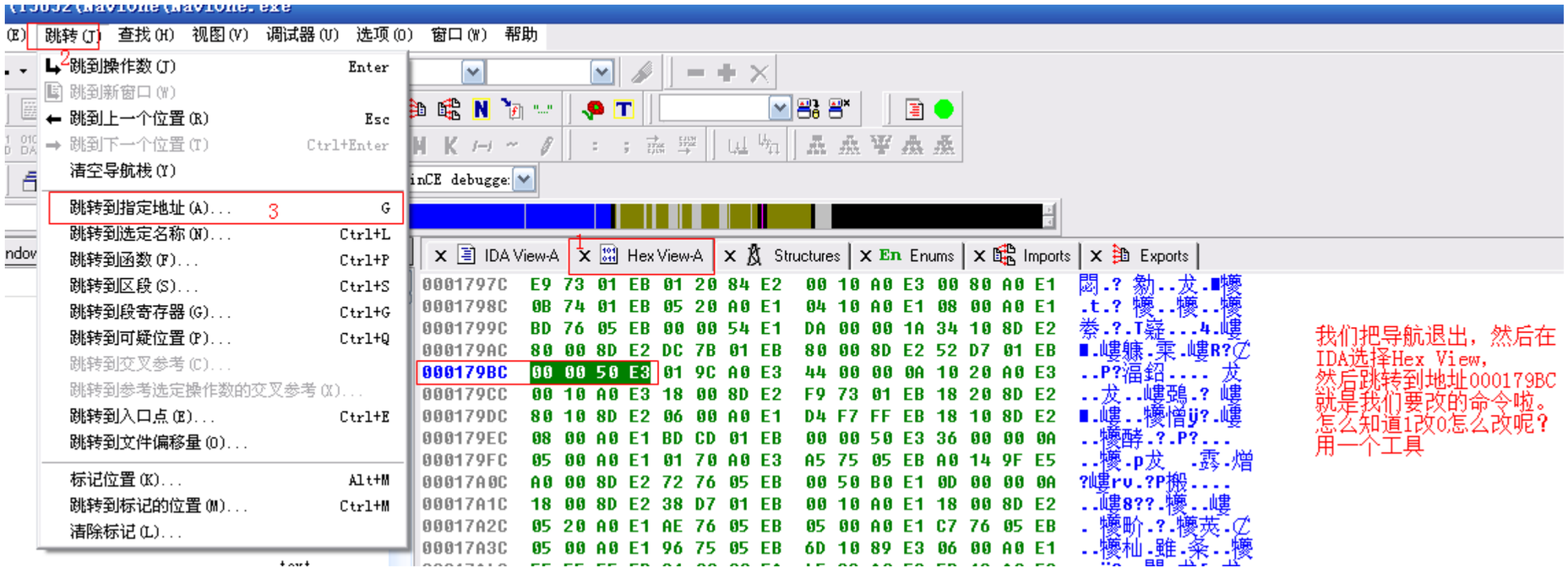
我们发现在179BC有个CMP R0, #0
这个比较结果会影响Z位，简单点
把#0，改成#1，试试

1
我们看见程序在模拟器里跑起来了
说明我们猜对了。
接下来就看如果修改了。

```
000179A8 ADD     R1, SP, #0x1C4+var_190
000179AC ADD     R0, SP, #0x1C4+var_144
000179B0 BL      sub_76928
000179B4 ADD     R0, SP, #0x1C4+var_144
000179B8 BL      sub_8D708
000179BC CMP     R0, #0
000179C0 MOV     R9, #0x100
000179C4 BEQ     loc_17ADC
```

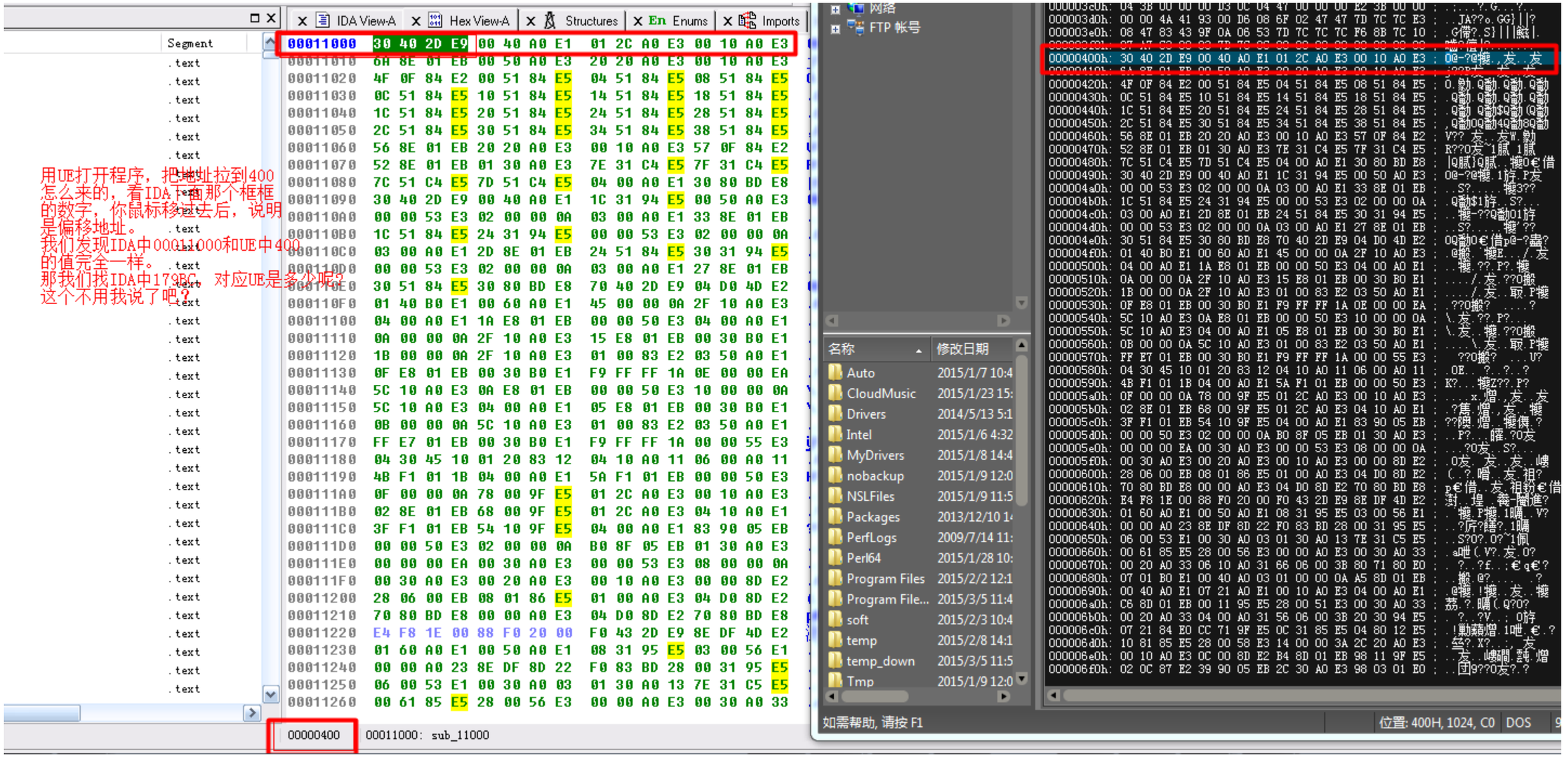
```
000179C8 MOV     R2, #0x10       ; size_t
000179CC MOV     R1, #0          ; int
000179D0 ADD     R0, SP, #0x1C4+var_1AC ; void *
000179D4 BL      j_memset
000179D8 ADD     R2, SP, #0x1C4+var_1AC
000179DC ADD     R1, SP, #0x1C4+var_144
000179E0 MOV     R0, R6
000179E4 BL      sub_1593C
000179E8 ADD     R1, SP, #0x1C4+var_1AC
000179EC MOV     R0, R8
000179F0 BL      sub_8B0EC
000179F4 CMP     R0, #0
000179F8 BEQ     loc_17AD8
```

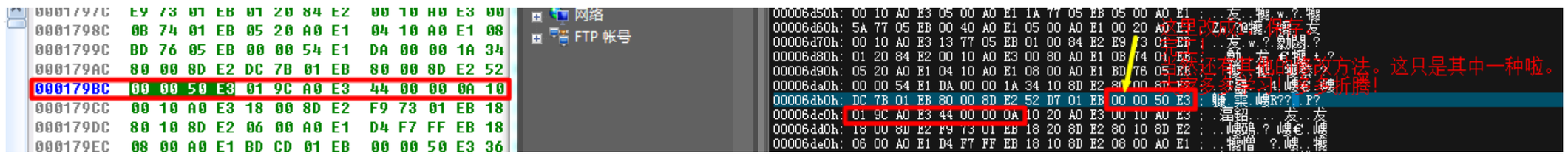
```
472)  00006DC4  000179C4: sub_17884+140

00 00 50 E3  2B 00 00 0A 10 30 9D E5  %..?.P?....0清
21 00 00 0A  0C 20 97 E5 00 40 A0 E3   ..S?.... 柄.@戈
03 00 52 E1  1A 00 00 2A D8 00 9F E5  .P戈..R?..*?熠
06 10 A0 E1  13 F8 FF EB 00 30 D6 E5  .`催..懊.??0皱
13 00 00 0A  10 10 9D E5 0C 30 97 E5  ..S?.....清.0柄
```


(E) 跳转(J) 查找(H) 视图(V) 调试器(U) 选项(O) 窗口(W) 帮助
2
跳到操作数(J) Enter
跳到新窗口(W)
跳到上一个位置(R) Esc
跳到下一个位置(T) Ctrl+Enter
清空导航栈(Y)
跳转到指定地址(A)... 3 G
跳转到选定名称(N)... Ctrl+L
跳转到函数(F)... Ctrl+P
跳转到区段(S)... Ctrl+S
跳转到段寄存器(G)... Ctrl+G
跳转到可疑位置(P)... Ctrl+Q
跳转到交叉参考(C)...
跳转到参考选定操作数的交叉参考(X)...
跳转到入口点(E)... Ctrl+E
跳转到文件偏移量(O)...
标记位置(K)... Alt+M
跳转到标记的位置(M)... Ctrl+M
清除标记(L)...
inCE debugge:
1
IDA View-A
Hex View-A
Structures
Enums
Imports
Exports
0001797C E9 73 01 EB 01 20 84 E2 00 10 A0 E3 00 80 A0 E1
0001798C 0B 74 01 EB 05 20 A0 E1 04 10 A0 E1 08 00 A0 E1
0001799C BD 76 05 EB 00 00 54 E1 DA 00 00 1A 34 10 8D E2
000179AC 80 00 8D E2 DC 7B 01 EB 80 00 8D E2 52 D7 01 EB
000179BC 00 00 50 E3 01 9C A0 E3 44 00 00 0A 10 20 A0 E3
000179CC 00 10 A0 E3 18 00 8D E2 F9 73 01 EB 18 20 8D E2
000179DC 80 10 8D E2 06 00 A0 E1 D4 F7 FF EB 18 10 8D E2
000179EC 08 00 A0 E1 BD CD 01 EB 00 00 50 E3 36 00 00 0A
000179FC 05 00 A0 E1 01 70 A0 E3 A5 75 05 EB A0 14 9F E5
00017A0C A0 00 8D E2 72 76 05 EB 00 50 B0 E1 0D 00 00 0A
00017A1C 18 00 8D E2 38 D7 01 EB 00 10 A0 E1 18 00 8D E2
00017A2C 05 20 A0 E1 AE 76 05 EB 05 00 A0 E1 C7 76 05 EB
00017A3C 05 00 A0 E1 96 75 05 EB 6D 10 89 E3 06 00 A0 E1
我们把导航退出，然后在
IDA选择Hex View,
然后跳转到地址000179BC
就是我们要改的命令啦。
怎么知道1改0怎么改呢？
用一个工具


Arm汇编转换工具-by:游人客栈-加盐白咖啡
Arm汇编
汇编语句行数[共 2 行]
论坛地址：http://bbs.urenkz.com
CMP R0,#0
CMP R0,#1
1
2
机器码
转换机器码
联动转换
arm-linux: Assembler messages:
arm-linux: Warning: end of file not at end of a line; newline inserted
ARM GAS arm-linux page 1
3
1 0000 000050E3 CMP R0,#0
2 0004 010050E3 CMP R0,#1
看到没？所以我们只要把000050E3
改成010050E3，其实只要改一位就可
以了。

用UE打开程序，把地址拉到400
怎么来的，看IDA下面那个框框
的数字，你鼠标移过去后，说明
是偏移地址。
我们发现IDA中00011000和UE中400
的值完全一样。
那我们找IDA中179BC，对应UE是多少呢？
这个不用我说了吧？





Symbian

IDA 动态调试病毒样本准备工作

准备:

1. IDA

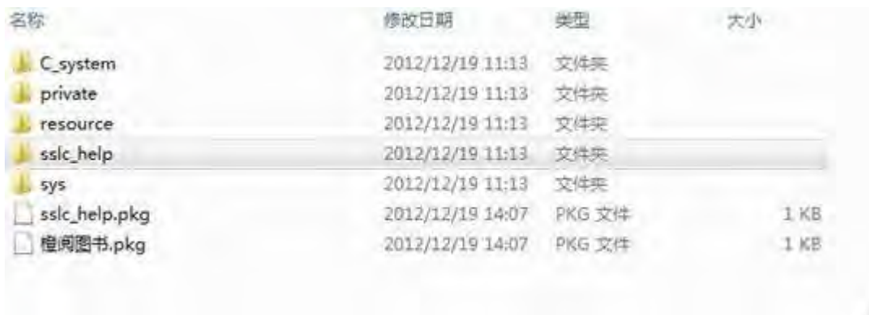
2. Nokia PC 套件 :下载地址:http://nds1.nokia.com/files/support/global/phones/software/Nokia_PC_Suite_7_1_18_0_chi_sc_web.exe

3. SISContents 工具

病毒样本压缩包截图

第一步: 使用 Nokia 提供的解压工具 SISContents 打开上图包如下图

第二步：单击解包文件, 解开压缩包, 同时查看文件信息, 注意文件安装路径.



通过上图获取信息:

程序名称: 橙阅图书

程序安装路径: c:/!:\sys\bin\MoLibrary.exe (!表示可能其他盘符).

可疑的文件:

1. \sys\bin\ MoLibrary.exe

2. \sys\bin\s.exe

3. \sys\bin\ssl.exe

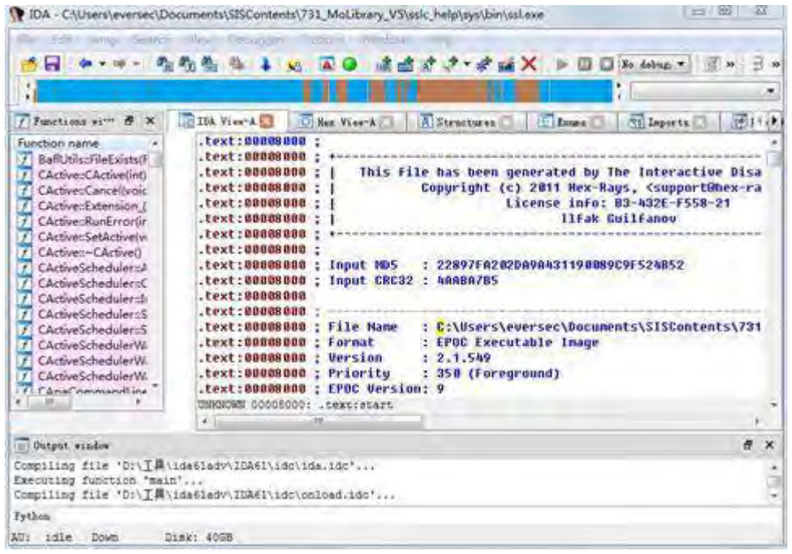
第三步: 拷贝病毒样本到手机客户端,然后安装,获取程序安装路径

C:\sys\bin\ssl.exe

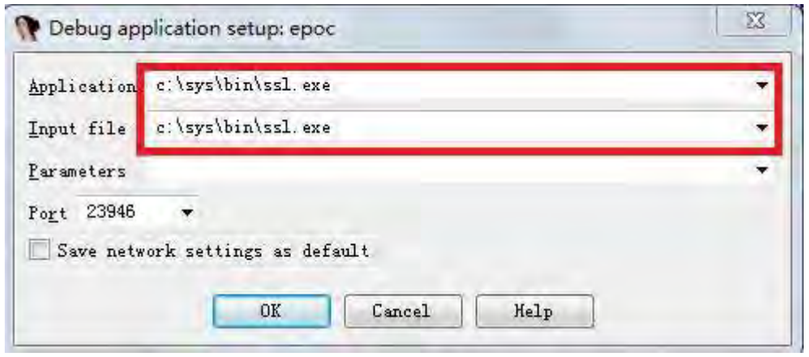
第四步: 在 PC 上安装诺基亚 PC 套件,启动它.

第五步：用 PC 连接手机,选择 PC 套件连接.

第六步：打开 IDA,选择 ARM 处理器加载样本文件,如下图

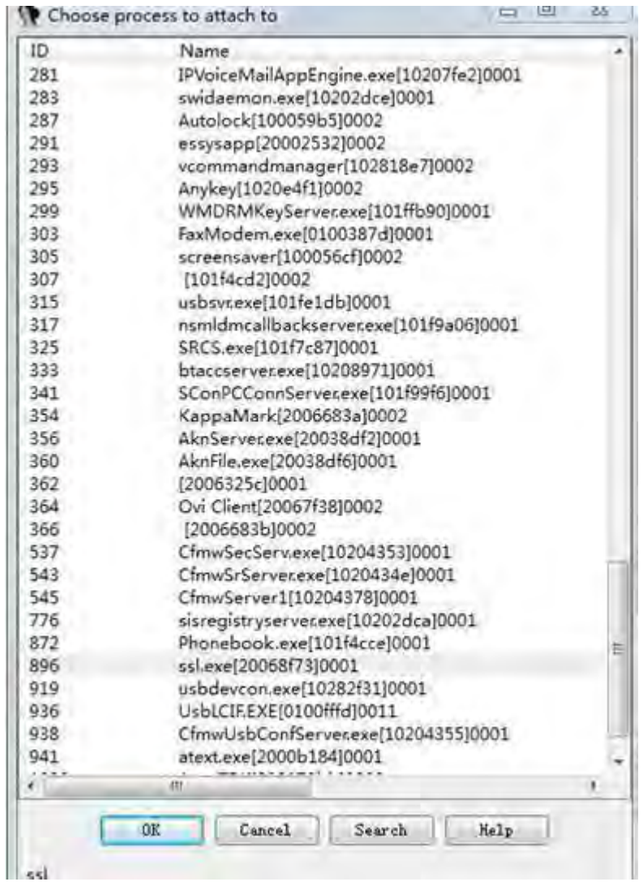


第七步：选择 Remote Symbian debugger,设置 IDA，如下图



第八步：设置好了后,打开手机 TRK 软件连接。

第九步：在 IDA 中 API 函数中下常用断点,后挂接进程如下图：



选择 ssl.exe 后,单击 OK.

开始调试分析…

http://www.cnblogs.com/xiaojinma/archive/2012/12/19/2825474.html

MIPS

通过 QEMU 和 IDA Pro 远程调试设备固件

cssembly · 2015/01/08 9:05

0x00 背景与简介

这篇文章主要讲了如何在模拟环境下调试设备固件。

作者：Zach Cutlip

原文链接：http://shadow-file.blogspot.gr/2015/01/dynamically-analyzing-wifi-routers-upnp.html。

在分析嵌入式设备的固件时，只采用静态分析方式通常是不够的。你需要实际执行你的分析目标来观察它的行为。在嵌入式 Linux 设备的世界里，很容易 把一个调试器放在目标硬件上进行调试。如果你能在自己的系统上运行二进制文件，而不是拖着硬件做分析，将会方便很多。这就需要用 QEMU 进行仿真。

接下来的一系列文章，我将专注于 Netgear 的一个比较流行的无线路由器，对其 UPnP 守护进程进行逆向分析。这篇文章将介绍如何在系统仿真环境下运行守护进程，以便可以通过调试器对其进行分析。

0x01 先决条件

首先，建议你读一下我的工作区以及所使用的工具的描述。这里是链接 http://shadow-file.blogspot.com/2013/12/emulating-and-debugging-workspace.html。

你需要一个 MIPS Linux 的模拟环境。对于这一点，建议读者查看我之前的搭建 QEMU 的帖子。http://shadow-file.blogspot.com/2013/05/running-debian-mips-linux-in-qemu.html

你还需要一个 MIPS Linux 的交叉编译器。我不会详细描述交叉编译器的建立过程，因为它们一团糟。有时候，你需要较旧的工具链，而其他时候，你需要较新的工具链。最好使用 uClibc buildroot project（http://buildroot.uclibc.org/）同时建立 big endian 和 little endian 的 MIPS Linux 工具链。除此之外，每当我发现其他交叉编译工具链时，我都会保存他们。类似 D-Link 和 Netgear 公司发布 GPL 版本 tarballs 是 旧工具链的好来源。

一旦你有了目标架构的交叉编译工具链，你需要为目标建立 GDB 调试器。你至少需要适合目标架构的被静态编译的 gdbserver。如果要使用 GDB 进行远程调试，你需要编译 GDB，以便在你的本地机器架构（如 X86-64）上运行，来对目标架构（如 MIPS 或 mipsel 体系结构）进行调试。同样， 我也不会去讨论这些工具的构建，如果你建立了你的工具链，这应该很容易了。

后面我将描述如何使用 IDA Pro 来进行远程调试。但是，如果你想使用 gdb，可以看看我的 MIPS gdbinit 文件：https://github.com/zcutlip/gdbinit-mips

0x02 模拟一个简单的二进制

假设你已经建立了上述工具，并能正常工作，你现在应该能够通过 SSH 进入您的模拟 MIPS 系统。正如我在 Debian MIPS QEMU 文章中所述，我喜欢连接 QEMU 的接口到 VMWare 的 NAT 接口，这样就能够用我的 Mac 通过 SSH 来接入，而不需要先登陆我的 Ubuntu 虚 拟机。这也可以让我通过 NFS 将我的 Mac 工作区挂载到 QEMU 系统。这样，无论我工作在主机环境中，或是 Ubuntu 中，还是在 QEMU 中，我都在用相 同的工作区。

```
zach@malastare:~ (130) $ ssh root@192.168.127.141
/* <![CDATA[ */!function(){try{var t="currentScript"in document?document.currentScript:function(){for(var t=document.getElementsByTagName("script"),e=t.length;e--;)if(t[e].getAttribute("cf-hash"))return t[e]}();if(t&&t.previousSibling){var e,r,n,i,c=t.previousSibling,a=c.getAttribute("data-cfemail");if(a){for(e="",r=parseInt(a.substr(0,2),16),n=2;a.length-n;n+=2)i=parseInt(a.substr(n,2),16)^r,e+=String.fromCharCode(i);e=document.createTextNode(e),c.parentNode.replaceChild(e,c)}}}catch(u){}}();/* ]]> */
root@192.168.127.141
/* <![CDATA[ */!function(){try{var t="currentScript"in document?document.currentScript:function(){for(var t=document.getElementsByTagName("script"),e=t.length;e--;)if(t[e].getAttribute("cf-hash"))return t[e]}();if(t&&t.previousSibling){var e,r,n,i,c=t.previousSibling,a=c.getAttribute("data-cfemail");if(a){for(e="",r=parseInt(a.substr(0,2),16),n=2;a.length-n;n+=2)i=parseInt(a.substr(n,2),16)^r,e+=String.fromCharCode(i);e=document.createTextNode(e),c.parentNode.replaceChild(e,c)}}}catch(u){}}();/* ]]> */'s password:
Linux debian-mipsel 2.6.32-5-4kc-malta #1 Wed Jan 12 06:13:27 UTC 2011 mips

root@debian-mipsel:~# mount
/dev/sda1 on / type ext3 (rw,errors=remount-ro)
malastare:/Users/share/code on /root/code type nfs (rw,addr=192.168.127.1)
root@debian-mipsel:~#cd code
root@debian-mipsel:~/code#
```

一旦登陆到模拟系统，cd 到从设备的固件中提取的文件系统中。你应该能够 chroot 到固件的根文件系统。需要使用 chroot，因为目标二进制是和固件的库链接的，很可能不能跟 Debian 的共享库一起工作。

```
root@debian-mipsel:~#cd code/wifi-reversing/netgear/r6200/extracted-1.0.0.28/rootfs/
root@debian-mipsel:~/code/wifi-reversing/netgear/r6200/extracted-1.0.0.28/rootfs#file ./bin/ls
./bin/ls: symbolic link to `busybox'
root@debian-mipsel:~/code/wifi-reversing/netgear/r6200/extracted-1.0.0.28/rootfs#file ./bin/busybox
./bin/busybox: ELF 32-bit LSB executable, MIPS, MIPS32 version 1 (SYSV), dynamically linked (uses shared libs), stripped
root@debian-mipsel:~/code/wifi-reversing/netgear/r6200/extracted-1.0.0.28/rootfs#chroot . /bin/ls -l /bin/busybox
-rwxr-xr-x    1 10001    80         276413 Sep 20  2012 /bin/busybox
root@debian-mipsel:~/code/wifi-reversing/netgear/r6200/extracted-1.0.0.28/rootfs#
```

在上面的例子中，我已切换到所提取的文件系统的根目录中。然后使用 file 命令，我了解到 busybox 是 little endian MIPS 可执行文件。然后，chroot 到提取根目录，并运行 bin/ls，它是 busybox 的符号链接。

如果试图简单的通过“chroot .”来地启动一个 shell，将无法正常工作。因为你的默认 shell 是 bash，而大多数嵌入式设备没有 bash。

root@debian-mipsel:~/code/wifi-reversing/netgear/r6200/extracted-1.0.0.28/rootfs#chroot . chroot: failed to run command `/bin/bash': No such file or directory root@debian-mipsel:~/code/wifi-reversing/netgear/r6200/extracted-1.0.0.28/rootfs#

相反，你可以 chroot 并执行 bin / sh：

root@debian-mipsel:~/code/wifi-reversing/netgear/r6200/extracted-1.0.0.28/rootfs#chroot . /bin/sh

BusyBox v1.7.2 (2012-09-20 10:26:08 CST) built-in shell (ash)
Enter 'help' for a list of built-in commands.

#
#
#exit
root@debian-mipsel:~/code/wifi-reversing/netgear/r6200/extracted-1.0.0.28/rootfs#

0x03 硬件解决方法

即使有了必要的工具，建立了仿真环境且工作正常，你仍然可能遇到障碍。虽然 QEMU 在模拟核心芯片组包括 CPU 上都做的很不错，但是 QEMU 往往不 能提供你想运行的二进制程序需要的硬件。如果你试图模仿一些简单的像/bin/ls 的程序，通常能够正常工作。但更复杂的东西，如肯定有特定的硬件依赖的 UPnP 守护程序，QEMU 就不能够满足。尤其对于管理嵌入式系统硬件的程序更是这样，例如打开或关闭无线适配器。

你将遇到的最常见问题是在运行系统服务，如 Web 服务器或 UPnP 守护进程时，缺乏 NVRAM。非易失性 RAM 通常是包含配置参数的设备快速存储器 的一个分区。当一个守护进程启动时，它通常会尝试查询 NVRAM，获取其运行时配置信息。有时一个守护进程会查询 NVRAM 的几十甚至上百个参数。

为了解决在模拟条件下缺乏 NVRAM 的问题，我写了一个叫 nvram-faker 的库。当你运行二进制程序时，应该使用 LD_PRELOAD 对 nvram-faker 库进行预加载。它会拦截通常由 libnvram.so 提供的 nvram_get（）调用。nvram-faker 会查询你提供的一 个 INI 风格的配置文件，而不是试图查询 NVRAM。

附带的 README 提供更完整的说明。这里有一个链接：https://github.com/zcutlip/nvram-faker

即使解决了 NVRAM 问题，该程序可能会假设某些硬件是存在的。如果硬件不存在，该程序可能无法运行，或者即便它运行了，行为可能也与在其目标硬件 上运行时有所不同。在这种情况下，你可能需要修补二进制文件。采用二进制补丁的情况各不相同。这取决于期望什么硬件，以及它不存在时的行为是什么。如果硬 件缺失，你可能需要修补一个被执行到的条件分支。也可能需要修补针对特殊设备的 ioctl（）调用，如果你想将其替代为针对常规文件的读取和写入时。这里 我将不谈论修补的细节，但在我的 BT HomeHub 文章以及 44CON 的相应讲座中进行了讨论。

这里是这些资源的链接：http://shadow-file.blogspot.com/2013/09/44con-resources.html

0x04 附加调试器

一旦在 QEMU 中运行了二进制程序，就可以附加调试器了。当然你需要 gdbserver。同样，这个工具应该适合目标架构并被静态编译，因为你会在 chroot 下运行它。你需要将它复制到提取的文件系统的根目录中。

```
# ./gdbserver
Usage: gdbserver [OPTIONS] COMM PROG [ARGS ...]
gdbserver [OPTIONS] --attach COMM PID
gdbserver [OPTIONS] --multi COMM

COMM may either be a tty device (for serial debugging), or
HOST:PORT to listen for a TCP connection.

Options:
  --debug               Enable general debugging output.
--remote-debug        Enable remote protocol debugging output.
--version             Display version information and exit.
  --wrapperWRAPPER --  Run WRAPPER to start new programs.
  --once                Exit after the first connection has closed.
#
```

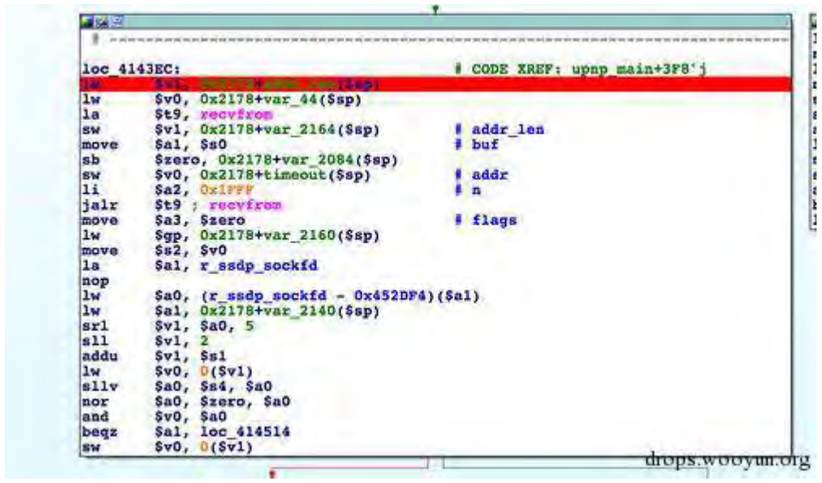
你可以将 gdbserver 附加到正在运行的进程，或者用它来直接执行二进制程序。如果你需要调试只发生一次的初始化程序，你可以选择后者。

另一方面，你可能要等到守护进程被创建。据我所知没有办法让 IDA 跟踪创建的进程（forked processes）。你需要单独的附加到它们。如果采用这种方式，你可以在 chroot 外，附加到已经运行的进程。

以下 shell 脚本将在 chroot 下执行 upnpd。如果 DEBUG 设置为 1，它会附加在 upnpd 上，并在 1234 端口上暂停等待远程调试会话。

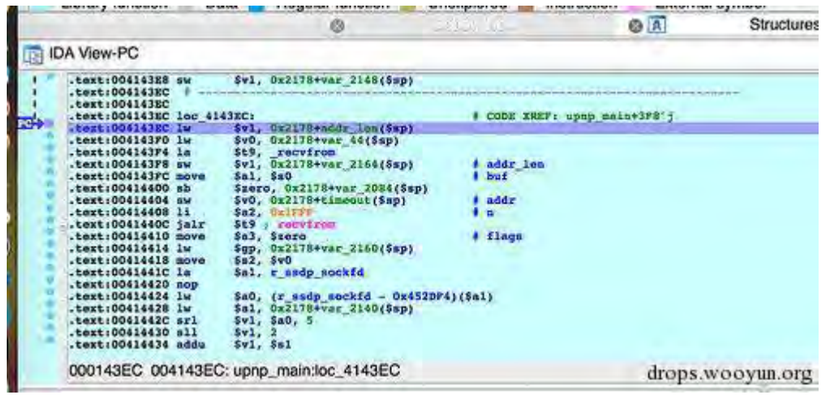
```
ROOTFS="/root/code/wifi-reversing/netgear/r6200/extracted-1.0.0.28/rootfs"
chroot$ROOTFS /bin/sh-c "LD_PRELOAD=/libnvram-faker.so /usr/sbin/upnpd"

#Give upnpd a bit to initialize and fork into the background.
sleep3;

if[ "x1"= "x$DEBUG"];
then

$ROOTFS/gdbserver--attach 0.0.0.0:1234 $(pgrepupnpd)
fi
```

你可以在 recvfrom（）调用之前创建一个断点，然后当你向 upnpd 发送 M-SEARCH 包时，验证调试器断点。



然后，在 IDA 的 Debugger 菜单中的 Process 选项中设置“主机名”为你的 QEMU 系统的 IP 地址，并设置端口为 gdbserver 正在监听的端口。我用的是 1234。

接受设置，然后通过 IDA 的 CTRL+8 热键连接到远程调试会话。再次按 Ctrl+8 继续执行。你应该能够发送一个 M-SEARCH 包 1，可以看到调试器命中断点。



显然还有很多需要探索，也会遇到有很多这里未提及的情况，但希望这可以让你开始。

我推荐 Craig Heffner 用于 UPnP 分析的 miranda 工具：https://code.google.com/p/miranda-upnp/

http://drops.wooyun.org/tips/4523

脚本编写

ida idc 函数列表全集

下面是函数描述信息中的约定：

'ea' 线性地址
'success' 0 表示函数失败；反之为 1
'void'表示函数返回的是没有意义的值（总是 0）

AddBptEx
AddBpt
AddCodeXref
AddConstEx
AddEntryPoint
AddEnum
AddHotkey
AddSourceFile
AddStrucEx
AddStrucMember
AltOp
Analysis
AnalyzeArea

AppendFchunk
ApplySig
AskAddr
AskFile
AskIdent
AskSeg
AskSelector
AskStr
AskYN
AttachProcess
AutoMark2
AutoMark
AutoShow
Batch
BeginEA
Byte
CanExceptionContinue
ChangeConfig
CheckBpt
ChooseFunction
CmtIndent
CommentEx
Comments
Compile
CompileEx
CreateArray
DelArrayElement
DelBpt
DelCodeXref
DelConstEx
DelEnum
DelExtLnA
DelExtLnB
DelFixup
DelFunction
DelHashElement
DelHiddenArea
DefineException
DelHotkey
DelLineNumber
DelSelector
DelSourceFile

DelStrucMember
DelStruc
DelXML
DeleteAll
DeleteArray
Demangle
DetachProcess
DfirstB
Dfirst
DnextB
Dnext
Dword
EnableBpt
EnableTracing
Eval
Exec
Exit
ExtLinA
ExtLinB
Fatal
FindBinary
FindCode
FindData
FindExplored
FindFuncEnd
FindImmediate
FindSelector
FindText
FindUnexplored
FindVoid
FirstFuncFchunk
FirstSeg
ForgetException
GenCallGdl
GenFuncGdl
GenerateFile
GetArrayElement
GetArrayId
GetBmaskCmt
GetBmaskName
GetBptAttr
GetBptEA

GetBptQty
GetCharPrm
GetColor
GetConstBmask
GetConstByName
GetConstCmt
GetConstEnum
GetConstEx
GetConstName
GetConstValue
GetCurrentLine
GetCurrentThreadId
GetDebuggerEvent
GetDisasm
GetEntryName
GetEntryOrdinal
GetEntryPointQty
GetEntryPoint
GetEnum
GetEnumCmt
GetEnumFlag
GetEnumIdx
GetEnumName
GetEnumQty
GetEnumSize
GetEnumWidth
GetEventBptHardwareEa
GetEventEa
GetEventExceptionCode
GetEventExceptionEa
GetEventExceptionInfo
GetEventExitCode
GetEventId
GetEventInfo
GetEventModuleBase
GetEventModuleName
GetEventModuleSize
GetEventPid
GetEventTid
GetExceptionCode
GetExceptionFlags
GetExceptionName

GetExceptionQty
GetFchunkAttr
GetFirstBmask
GetFirstConst
GetFirstHashKey
GetFirstIndex
GetFirstMember
GetFirstModule
GetFirstStrucIdx
GetFixupTgtDispl
GetFixupTgtOff
GetFixupTgtSel
GetFixupTgtType
GetFlags
GetFrameArgsSize
GetFrameLvarSize
GetFrameRegsSize
GetFrameSize
GetFrame
GetFuncOffset
GetFunctionAttr
GetFunctionCmt
GetFunctionFlags
GetFunctionName
GetHashLong
GetHashString
GetIdaDirectory
GetIdbPath
GetInputFilePath
GetInputFile
GetInputMD5
GetLastBmask
GetLastConst
GetLastHashKey
GetLastIndex
GetLastMember
GetLastStrucIdx
GetLineNumber
GetLocalType
GetLocalTypeName
GetLongPrm
GetManualInsn

GetMarkComment
GetMarkedPos
GetMaxLocalType
GetMemberComment
GetMemberFlag
GetMemberName
GetMemberOffset
GetMemberQty
GetMemberSize
GetMemberStrId
GetMnem
GetModuleName
GetModuleSize
GetNextBmask
GetNextConst
GetNextFixupEA
GetNextHashKey
GetNextIndex
GetNextModule
GetNextStrucIdx
GetOpType
GetOperandValue
GetOpnd
GetOriginalByte
GetPrevBmask
GetPrevConst
GetPrevFixupEA
GetPrevHashKey
GetPrevIndex
GetPrevStrucIdx
GetProcessName
GetProcessPid
GetProcessQty
GetProcessState
GetRegValue
GetRegValue
GetReg
GetSegmentAttr
GetShortPrm
GetSourceFile
GetSpDiff
GetSpd

GetStringType
GetString
GetStrucComment
GetStrucIdByName
GetStrucId
GetStrucIdx
GetStrucName
GetStrucNextOff
GetStrucPrevOff
GetStrucQty
GetStrucSize
GetThreadId
GetThreadQty
GetTrueName
GetType
GetXML
GetnEnum
GuessType
HideArea
HighVoids
IdbByte
Indent
IsBitfield
IsEventHandled
IsFloat
IsLong
IsString
IsUnion
ItemEnd
ItemSize
Jump
LineA
LineB
LoadDebugger
LoadTil
LocByName
LowVoids
MK_FP
MakeAlign
MakeArray
MakeByte
MakeCode

MakeComm
MakeData
MakeDouble
MakeDword
MakeFloat
MakeFrame
MakeFunction
MakeLocal
MakeName
MakeNameEx
MakeOword
MakePackReal
MakeQword
MakeRptCmt
MakeStr
MakeStructEx
MakeTbyte
MakeUnkn
MakeUnknown
MakeVar
MakeWord
MarkPosition
MaxEA
Message
MinEA
Name
NextAddr
NextFchunk
NextFuncFchunk
NextFunction
NextHead
NextNotTail
NextSeg
OpAlt
OpBinary
OpChr
OpDecimal
OpEnumEx
OpHex
OpHigh
OpNot
OpNumber

OpOctal
OpOffEx
OpOff
OpSeg
OpSign
OpStkvar
OpStroffEx
ParseTypes
PatchByte
PatchDbgByte
PatchDword
PatchWord
PauseProcess
PopXML
PrevAddr
PrevFchunk
PrevFunction
PrevHead
PrevNotTail
PushXML
Qword
RefreshDebuggerMemory
RefreshLists
Refresh
RemoveFchunk
RenameArray
RenameEntryPoint
ResumeThread
Rfirst0
RfirstB0
RfirstB
Rfirst
Rnext0
RnextB0
RnextB
Rnext
RunPlugin
RunTo
SaveBase
ScreenEA
SegAddrng
SegAlign

SegBounds
SegByBase
SegByName
SegClass
SegComb
SegCreate
SegDefReg
SegDelete
SegEnd
SegName
SegRename
SegStart
SelEnd
SelStart
SelectThread
SetArrayFormat
SetArrayLong
SetArrayString
SetBmaskCmt
SetBmaskName
SetBptAttr
SetBptCnd
SetCharPrm
SetColor
SetConstCmt
SetConstName
SetDebuggerOptions
SetEnumBf
SetEnumCmt
SetEnumFlag
SetEnumIdx
SetEnumName
SetEnumWidth
SetFchunkAttr
SetFchunkOwner
SetFixup
SetFlags
SetFunctionAttr
SetFunctionCmt
SetFunctionEnd
SetFunctionFlags
SetHashLong

SetHashString
SetHiddenArea
SetInputFilePath
SetLineNumber
SetLocalType
SetLongPrm
SetManualInsn
SetMemberComment
SetMemberName
SetMemberType
SetProcessorType
SetRegValue
SetRegValue
SetReg
SetRegEx
SetRemoteDebugger
SetSegmentAttr
SetSegmentType
SetSelector
SetShortPrm
SetSpDiff
SetStatus
SetStrucComment
SetStrucIdx
SetStrucName
SetType
SetXML
Sleep
StartDebugger
StepInto
StepOver
StepUntilRet
StopDebugger
StringStp
SuspendThread
Tabs
TailDepth
Til2Idb
Voids
Wait
Warning
Word

XrefShow
XrefType
add_dref
atoa
atol
byteValue
del_dref
fclose
fgetc
filelength
fopen
form
fprintf
fputc
fseek
ftell
hasName
hasValue
isBin0
isBin1
isChar0
isChar1
isCode
isData
isDec0
isDec1
isDefArg0
isDefArg1
isEnum0
isEnum1
isExtra
isFlow
isFop0
isFop1
isHead
isHex0
isHex1
isLoaded
isOct0
isOct1
isOff0
isOff1

isRef
isSeg0
isSeg1
isStkvar0
isStkvar1
isStroff0
isStroff1
isTail
isUnknown
isVar
loadfile
ltoa
ord
readlong
readshort
readstr
rotate_left
savefile
set_start_cs
set_start_ip
strlen
strstr
substr
writelong
writeshort
writestr
xtol
http://www.cnblogs.com/Y4ng/p/ida_fuction.html

关于 idapython 编程的资料

ida 官方提供了一个地址：https://www.hex-rays.com/products/ida/support/idapython_docs/

ida 在命令行下启动：http://www.2cto.com/shouce/ida/417.htm

一些小脚本的例子：

ida 6.8 测试

1.获得程序所有入口点地址（十机制）

```
entries = Entries()

for i in entries:

print i[1]
```

2.获得程序的基址

```
import idaapi

print idaapi.get_imagebase()
```

3.交叉引用：

```
coderefs = CodeRefsTo(addr,1)        #addr 为我们输入的地址,返回值是地址，这些地址在引用你输入的这个地址的指令
```

4.得到当前地址指令的操作符：

```
ins = GetMnem(addr)        #addr 是当前地址
```

########################

dll 插件编写中会用到的函数：

ua_mnem()　　#ua.hpp　　作用是获得当前地址的操作符

http://fccload.lofter.com/post/1d9051c4_94bf65b

Loading your own modules from your IDAPython scripts with idaapi.require()

TL;DR

If you were using import to import your own “currently-in-development” modules from your IDAPython scripts, you may want to use idaapi.require(), starting with IDA 6.5.

Rationale

When using IDAPython scripts, users were sometimes facing the following issue

Specifically:

- User loads script
- Script imports user's module mymodule
- Script ends
- User modifies code of mymodule (Note: the module is modified, not the script)
- User reloads script
- Modifications to mymodule aren't taken into consideration.

While that's perfectly understandable (the python runtime doesn't have to reload mymodule if it has been compiled & loaded already), this is somewhat of an annoyance for users that were importing modules that were often modified.

IDA <= 6.4: Ensuring a user-specified module gets reloaded, by destroying it.

Up until IDA 6.4, the IDAPython plugin would do some magic after you have run your user script.
(click "expand all" to reveal the diff)

The sequence becomes:

- User loads script
- Script imports user's module mymodule
- Script ends
- [module mymodule is deleted]
- User modifies code of mymodule
- User reloads script
- Modifications to mymodule are taken into consideration, since module was deleted.

Unfortunately we have to stop doing this because:

- That prevents us from using python-based hooks to be used after the script is finished (see below).
- That goes against the rest of the python philosophy (i.e., modifications to objects are not reverted), and is therefore unexpected.

Issues with hooks.

Imagine you have the following script, dbghooks.py:

```
from idaapi import *
import mydbghelpers

class MyHooks(DBG_Hooks):

  def __init__(self):
    ...

  def dbg_bpt(self, tid, ea):
    mydbghelpers.do_something()
    return 0

  def dbg_step_into(self):
    ...

hooks = MyHooks()
hooks.hook()
```

- User loads script
- Scripts `imports mydbghelpers`
- Script creates instance of `MyHooks`, and hooks it into IDA's debugger APIs
- Script ends
- [module `mydbghelpers` is deleted]
- User runs debugger, and a breakpoint is hit. Two things can happen:
  - The hook fails executing
  - IDA crashes (that can happen if the form `from mydbghelpers import *` was used)

IDA > 6.4: Introducing `idaapi.require()`

Everywhere else in python, when you modify a runtime object, those changes will remain visible.

We decided it would be better to not go against that standard behaviour anymore, and provide a helper to achieve the same results as what was achieved before with the deletion of user modules.

You can now import & re-import of a module with: idaapi.require(name)

Here is its definition:

```
def require(modulename):
    if modulename in sys.modules.keys():
        reload(sys.modules[modulename])
    else:
```

```
        import importlib
        import inspect
        m = importlib.import_module(modulename)
        frame_obj, filename, line_number, function_name, lines, index = inspect.stack()[1]
        importer_module = inspect.getmodule(frame_obj)
        if importer_module is None: # No importer module; called from command line
            importer_module = sys.modules['__main__']
        setattr(importer_module, modulename, m)
        sys.modules[modulename] = m
```

Example

The example debugger hooks script above becomes:

```
from idaapi import *
idaapi.require("mydbghelpers")

class MyHooks(DBG_Hooks):

  def __init__(self):
    ...

  def dbg_bpt(self, tid, ea):
    mydbghelpers.do_something()
    return 0

  def dbg_step_into(self):
    ...

hooks = MyHooks()
hooks.hook()
```

I.e., only the second line changes.

http://www.hexblog.com/?p=749

过掉百度加固的 java 层调试

在百度的加固中会使用这句话来判断本程序是否被调试

```
if (!Debug.isDebuggerConnected())
```

这样对于 ida 使用者而言,网上的公开调试方式 so 的方式就不可以用了,怎么办呢?

闲来没事分析了一下,调用的是下面这个函数

```
public static boolean isDebuggerConnected(){
            return VMDebug.isDebuggerConnected();
}
```

在 libcore\dalvik\src\main\java\dalvik\system\VMDebug.java 下：

```
public static native boolean isDebuggerConnected()
```

发现是一个 native 函数，那就 hook 它过调试检测。

下面给出脚本:

```
from idaapi import *
from idc import *

debug_addr = LocByName("_Z25dvmDbgIsDebuggerConnectedv")
end = FindFuncEnd(debug_addr)  - 0x02
count = 0;
class DumpHook(DBG_Hooks):
def dbg_bpt(self,tid,ea):
        global count
        r0 = GetRegValue('r0')
        if r0 == 1:
                count = count + 1
```

```
                if count == 2:
                        SetRegValue(0,"r0")
        ResumeProcess()
        return 0


AddBpt(end)
debug = DumpHook()
debug.hook()
print "hook"
```

http://fccload.lofter.com/post/1d9051c4_acc867b

idapython dump 内存脚本

第一种

```
auto fp, dexAddress;
    fp = fopen("D:\\test.dll", "wb");
    for ( dexAddress=0x7B95304C; dexAddress < 0x7BD0224C; dexAddress++ )
    fputc(Byte(dexAddress), fp);
```

第二种

```
static main(void)
{
    auto fp, begin, end, dexbyte;
    fp = fopen("C:\\dump.dex", "wb");
    begin = r0;
    end = r0 + r1;
    for ( dexbyte = begin; dexbyte < end; dexbyte ++ )
            fputc(Byte(dexbyte), fp);
}
```

第三种

```
from idaapi import *

from idc import *

import os


#get data begin and end

dump_data_begin= AskAddr(0,'dump data begin')

dump_data_end = AskAddr(0,'dump data end')

#get data len

len = dump_data_end - dump_data_begin

#get dex data

dump_data = DbgRead(dump_data_begin,len)

#write the data to file

fd = open("c:\\dump_data",'wb')

fd.write(dump_data)

fd.close()

print "end"
```

一个在调试状态下 dump 内存的 idapython 小脚本

IDAPython 的妙用

看过 Python 灰帽子的密友都知道 IDAPython 在逆向的场景中常常能够发挥出巨大的威力，笔者偶然一次在逆向的过程中使用了它，下面就来总结一下使用的方法和一些注意事项：

环境：IDA 6.1 ，Android 2.3 AVD。

我这个 IDAPython 的版本是：

Python 2.6.6 (r266:84297, Aug 24 2010, 18:46:32) [MSC v.1500 32 bit (Intel)]

IDAPython v1.5.2 final (serial 0) (c) The IDAPython Team idapython@googlegroups.com

---

下载这个包

安装方法：

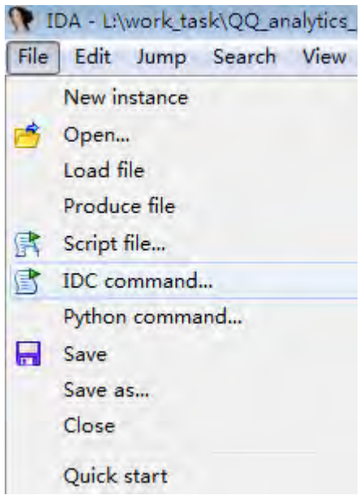
1）解压压缩包

2）将 python.plw，python.p64 复制到 IDA 的插件目录 plugins 下。

3）将 python 的目录拷贝到 IDA 的主目录中。

如果安装成功，在启动 IDA 后，输出面板会有如上的输出。

安装好了以后，我的只有一个 Python command... 命令项，并没有 Python file...项



应用：

```
from idaapi import *
from idc import *


count = 0


class DumpHook(DBG_Hooks):
    def dbg_bpt (self,tid,ea):
        global count
        count += 1;
        print "[*] Hit: 0x%08x the %d time\n" % (ea, count)
        data = "\xBE\x91\x0A\xF3\x9A\x26\xA4\xA9\x92\xC6\xFD\x01\xA1\x43\xED\x19"
        dbg_write_memory(GetRegValue("r7"), data)
        return 1


try:
    if debugger:
        print("Removing previous hook ...")
        debugger.unhook()
except:
    pass


AddBpt (0x8050a42e)
```

```
SetBptAttr(0x8050a42e, BPTATTR_FLAGS, BPT_ENABLED|BPT_TRACE)
print "[*] set hook OK...\n"


debugger = DumpHook()
debugger.hook()
```

关于里面的函数，可以去查看 IDA 主目录下 python 文件夹里的 idc.py 和 idaapi.py，里面有对各个函数的详细说明。

注意：python 脚本从 python command...里输入，而不是作为一个文件从 script file...里输入

http://blog.csdn.net/chence19871/article/details/20446119

用 IDAPython 获取程序的执行流程(函数调用流程)

下面的 IDAPython 脚本用于获取目标程序的所有函数，并且在每个函数的开始处都设置好断点。该脚本摘自<<Python 灰帽子>>:

```
from idaapi import *
class FuncCoverage(DBG_Hooks):

        # Our breakpoint handler
        def dbg_bpt(self, tid, ea):
                print "[*] Hit: 0x%08x" % ea
                return 1

# Add our function coverage debugger hook
debugger = FuncCoverage()
debugger.hook()

current_addr = ScreenEA()

# Find all functions and add breakpoints
for function in Functions(SegStart( current_addr ), SegEnd( current_addr )):
        AddBpt( function )
        SetBptAttr( function, BPTATTR_FLAGS, 0x0)


num_breakpoints = GetBptQty()
```

```
print "[*] Set %d breakpoints." % num_breakpoints
```

在 IDA 中运行上面的脚本，提示“[*] Set 0 breakpoints.” 即没有设置任何断点，为什么？

1、猜测 ScreenEA()不能获取当前地址，修改为：

```
current_addr = ScreenEA()
ss= str(hex(current_addr))

print "current_addr=%s" %ss
```

发现 ScreenEA()能正确地获取当前地址

2、猜测 Functions（）不能正确获取所有函数，修改为：

```
# Find all functions and add breakpoints
i=0
print "before loop, i=",i
for function in Functions(SegStart( current_addr ), SegEnd( current_addr )):
       AddBpt( function )
       i=i+1
       SetBptAttr( function, BPTATTR_FLAGS, 0x0)
print "after loop, i=",i
```

发现 Functions（）能正确获取所有函数，在被测程序里：

```
current_addr=0x4209c9
before loop, i= 0
after loop, i= 4070
```

3、剩下最后一个疑点，即 GetBptQty()不能正确获取当前断点数量：

于是决定手动添加一个断点。这时才发现，无法添加断点。立刻明白了：IDA 的调试器没有打开!打开调试器,选 local win32 debugger,再次运行脚本,成功!

```
current_addr=0x4209ce
before loop, i= 0
after loop, i= 4070
[*] Set 4070 breakpoints.
```

之后运行 IDA 调试器，debugger hook 会把每一次断点触发的情况通知我们。但本例中有 4070 个断点,不敢尝试,换个程序再试吧.

http://blog.163.com/lixiangdong2510@126/blog/static/3499482120119171180342/

使用 IDAPYTHON 跟踪程序执行路径-未加壳

IDA Pro 是一款静态反汇编的利器，具有良好的交互性、可编程性、可扩展性及对多处理器的支持。

对于 IDA Pro 的可扩展性表现在两个方面，一是采用内置的脚本语言 IDC 写脚本；二是采用 Hex-rays 提供的 SDK 写插件 Plug-in。简单的任务交由脚本完成，复杂的任务则需要定制满足需求的插件。

IDA 本身附带了一个内建的调试器，可采用该调试器来调试外挂程序，然后使用 IDAPython 脚本来获取程序执行路径。

技术说明

使用 IDAPyhon 脚本来获取程序执行路径的基本方法如下：

①　先用 IDA 反汇编可执行程序，生成 xx.pdb 文件。

② 编写 IDAPython 脚本，用来在稍后的调试中获取程序执行路径。脚本框架如下：

```
from idaapi import *

class FuncPath(DBG_Hooks):

	# Our breakpoint handler
	def dbg_bpt(self, tid, ea):
		print "[*] Hit: 0x%08x" % ea
		return 1


debugger = FuncPath () #建立一个对象
debugger.hook()		#将钩子装入 IDA 内建调试器

current_addr = ScreenEA()	#获取光标坐标

# 遍历所有函数，并添加断点，设置断点的属性为跟踪
for function in Functions(SegStart( current_addr ), SegEnd( current_addr )):
	AddBpt( function )
	SetBptAttr( function, BPTATTR_FLAGS, BPT_ENABLED|BPT_TRACE)

num_breakpoints = GetBptQty()
```

print "[*] Set %d breakpoints." % num_breakpoints

基本的思想是 建立一个 FuncPath 类，该类从 DBG_Hooks 类继承而来，包含了调试器的钩子和一些和调试相关的功能。你可以重写它的相关函数。

然后建立一个钩子类对象，获取光标所在位置，枚举每一个函数，然后在函数上下断点，并设置属性为跟踪。

将该脚本保存为文本，且命名为 get_path.py

③打开 PDB 文件，选择调试器为本地调试。然后从 文件->Script file... 载入 get_path.py 脚本。等几秒钟，断点就设置好了。

④ F9 运行调试器，就可以看到有断点命中了。并且在输出窗口里有 log 信息。

⑤ 你可以将在关键动作触发前的命中的函数的断点全部去除（可重断点窗口里手动删除）。然后触发你想跟踪的功能。

这样输出窗口里的命中函数就是你需要重点分析的函数了。

附脚本：

```
from idaapi import *


class FuncPath(DBG_Hooks):


    # Our breakpoint handler
    def dbg_bpt(self, tid, ea):
        print "[*] Hit: 0x%08x" % ea
        return 1
```

```
# Add our function coverage debugger hook

debugger = FuncPath ()

debugger.hook()


current_addr = ScreenEA()


# Find all functions and add breakpoints

for function in Functions(SegStart( current_addr ), SegEnd( current_addr )):

    AddBpt( function )

    SetBptAttr( function, BPTATTR_FLAGS, BPT_ENABLED|BPT_TRACE)


num_breakpoints = GetBptQty()


print "[*] Set %d breakpoints." % num_breakpoints
```

http://blog.csdn.net/chence19871/article/details/50727935

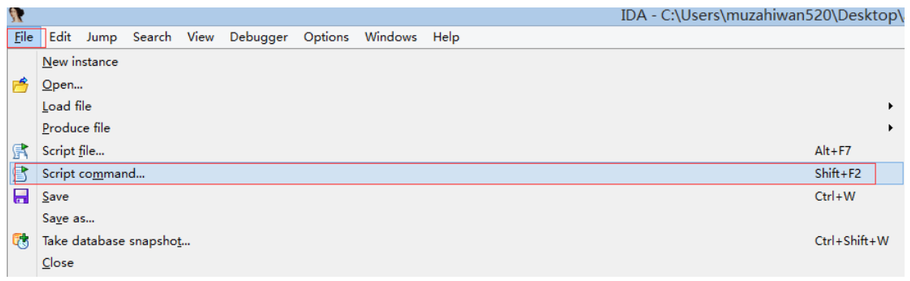
IDA 脚本编写基础

平时我们使用 IDA 进行动态调试的时候，有时候需要使用一些插件，有时候则选择更为方便的 idc 脚本，那么我们肯定是要学习了解下 idc 脚本的编写语法规范，所以以下是学习过程中的记录，便于以后翻阅。

0x1:简介

IDC 是 IDA 内置的脚本语言，语法与 C 类似，为解释性语言。

①.IDA 中可以直接执行 IDA 语句,点击 File->Script command 或者快捷键 shilt + F2 即可调出。



②.可以直接载入定义好的 IDC 脚本，如上图，选择 Script File 进行选择即可。

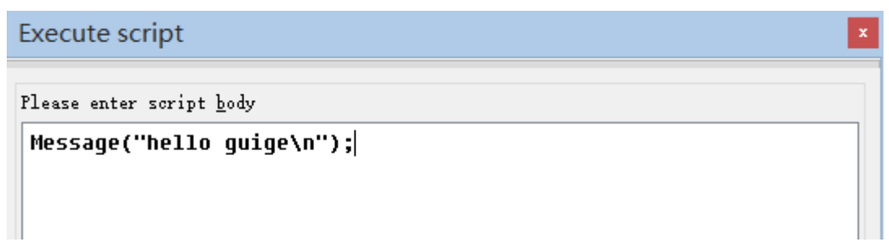
不过我建议还是用第三方的工具如 Notepad+打开脚本，然后复制，在 Script command 里面粘贴进去，因为这样我们可以修改一些数据，如在 dump 中的内存起始位置。

0x2:语法相关知识

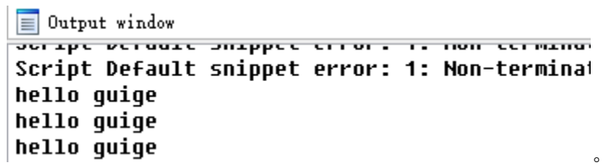
对于反汇编后的内容，编写脚本比编写插件简单很多，因为脚本是比较灵活的，我们可以针对单个分析任务来写对应的脚本，加快我们的分析速度，使得任务自动化。

①．输出函数

在 Shift +F2 后，在编辑框输入 Message 函数



然后观察输出窗口，可以看到有输出



。

Message 函数跟 C 里面的 printf 函数类似，可以使用格式化字符串，函数原型为:

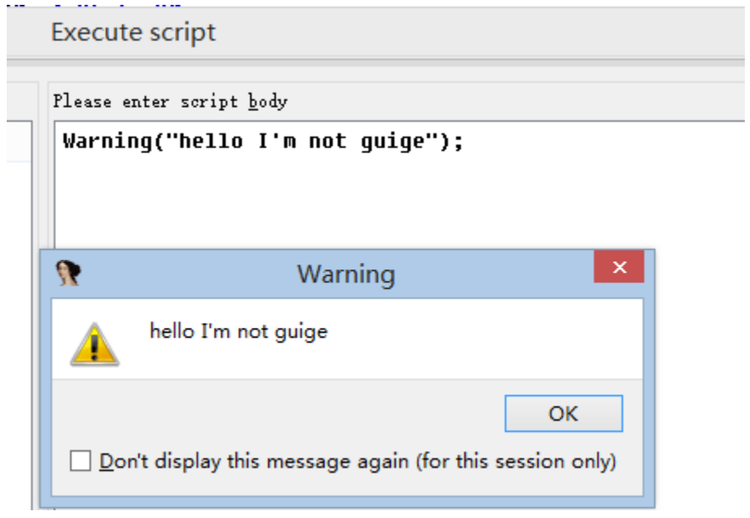
Void Message(string formart .....)

其他形式，如：

```
Message("%s\n","hello I'm not guige");
```

另外，还有其他常用函数，Warning 函数，Fatal 函数。

如:Warning 函数，只是弹出提示



其用法跟 Message 类似，不过这个是弹出框。

再如 Fatal 函数，弹出提示错误，点击 Ok 则会退出 IDA

②.变量

IDC 中所有变量被定义为 auto 类型，如定义一个计数器 counter 的变量。

auto counter

变量根据他们包含的数据的类型，有大小限制：

整数为 32 位的（IDA Pro64 中是 64 位）；

字符串最多包含 1023 个字符；

浮点变量最多 25 个十进制数位；

auto 变量可以表示不同类型的数据，所以需要用到转换规则，不过在编写脚本时候类型转换不是太常见，有些函数需要用于手动执行类型转换：

long(expr)

char(expr)

float(expr)

变量的声明和赋值必须在不同的语句中进行。

auto currentAddress

currentAddress=ScreenEA()

听鬼哥说故事

实战 IDA 脚本编程--用 idc 实现 JumpNotFunction

Mavermaver

用 IDA 反汇编可执行文件时，经常遇到未命名代码片段。对于少量这样的代码片段可以用 IDA 的 Search->not function 来定位（快 捷键通常为 Alt-U），再用 Edit->Functoins->Create function...(快捷键为 P)创建新函数。如果出现 大量这样的代码片段，反复用 Alt-U，P 操作就很繁琐。于是就想到利用用 idc 脚本来实现这个功能。查阅了 IDA 的 Help（https://www.hex-rays.com/products/ida/support/idadoc /index.shtml），Search->not function 这个功能的名称为 JumpNotFunction，但在 Help->Index of IDC functions 中找不到相对应的函数（快捷键 P 有对应的 MakeFunction）。

用 Google 搜 索 "How to implement JumpNotFunction using idc script in IDA environment" (如 何在 IDA 环境下用 idc 脚本实现 JumpNotFunction)，找不到答案。于是决定自己动手解决问题。经过一番学习，找到了主要的可以使用的相关 函数：FindCode、GetFunctionName、FindFuncEnd、isCode、GetFlags，摘录如下。

1.

```
// ea - address to start from
// flag is combination of the following bits:
#define SEARCH_DOWN     0x01          // search forward
#define SEARCH_NEXT     0x02          // search next occurence
#define SEARCH_CASE     0x04          // search case-sensitive
                                      // (only for bin&txt search)
#define SEARCH_REGEX    0x08          // enable regular expressions
#define SEARCH_NOBRK    0x10          // don't test ctrl-break
#define SEARCH_NOSHOW   0x20          // don't display the search progress
//      return BADADDR - not found
long FindCode(long ea,long flag);
```

2.

```
// ea - any address belonging to the function
// returns: null string - function doesn't exist otherwise returns function name
string GetFunctionName(long ea);
```

3.

```
// ea  - starting address of a new function
// returns:       if a function already exists, then return its end address.
//                if a function end cannot be determined, then return BADADDR
//                otherwise return the end address of the new function
long FindFuncEnd(long ea);
```

4.

```
#define MS_CLS  0x00000600L           // Mask for typing
#define FF_CODE 0x00000600L           // Code ?
#define FF_DATA 0x00000400L           // Data ?
```

```
#define FF_TAIL 0x00000200L            // Tail ?
#define FF_UNK  0x00000000L            // Unknown ?
#define isCode(F)       ((F & MS_CLS) == FF_CODE) // is code byte?
```

5.

```
// ea - linear address
// returns: 32-bit value of internal flags. See start of IDC.IDC file for explanations.
long GetFlags(long ea);         // get internal flags for ea
```

以上函数有关参数的简要说明如下（其中的 BADADDR 为无效地址）：

1. FindCode 参数：
输入：
 ea=有效起始地址
 flag=SEARCH_DOWN=向下（地址增加方向）搜索
    （SEARCH_NEXT=搜索下一个；
    SEARCH_CASE=搜索时区分大小写，仅对二进制和文本有效；
    SEARCH_REGEX=允许正则表达式；
    SEARCH_NOBRK=不检测 Ctrl-Break 按键；
    SEARCH_NOSHOW=不显示搜索进度）
输出：
 返回代码片段起始地址（如果没找到代码片段，则返回 BADADDR）。

2. GetFunctionName 参数：
输入：
 ea=属于某个函数代码范围内的任何有效地址。
输出：
 返回函数名字符串，如果函数不存在，则返回空字符串。

3. FindFuncEnd 参数说明：
输入：
 ea=某个新建函数的起始有效地址。
输出：
 如果函数已经存在，则返回其结束地址；如果函数的结束地址不确定，则返回 BADADDR；否则返回新建函数的结束地址。

4. isCode 参数说明（isCode 实际上为宏定义）：
输入：
 F=相对与某个有效地址的标志。
输出：
 逻辑值：1（True）=是代码；0（False）=非代码。

5. GetFlags 参数说明：
输入：
 ea=线性有效地址。
输出：
 对应于 ea 的标志。

有了这些资料就可以动手构建 JumpNotFunction 了。首先要获取一个有效地址，通常使用 ScreenEA 获取当前屏幕光标所在地址，然后判断该 地址是否为代码，关键在于判断代码是否属于某个函数。也可以用 ScreenEA 保存当前地址，再用 MinEA 获取代码段起始地址并对整个代码段进行扫描， 然后恢复当前地址。

我们可以用 File->IDC Command...(快捷键 Shift-F2)把以下代码粘贴到弹出文本窗口测试一下：

```
 auto ea,name;   // 声明自动类型变量
 ea = ScreenEA(); // 获取当前地址
 ea = FindCode(ea,SEARCH_DOWN); // 找代码片段
 name = GetFunctionName(ea);  // 获取函数名
 Message("ea=%lx   name=%s\n",ea,name); // 显示信息
```

先找到某个已经定义函数（sub_40xxxx 为某个函数的起始地址），运行上述代码，可以到到以下信息：
ea=40xxxx   name=sub_40xxxx

再用 Edit->Functions->Delete function 删除该函数后运行上述代码，可以到到以下信息：
ea=40xxxx   name=

这说明上述代码有效。

以下是该函数以及利用该函数解决问题的完整代码：

```
#include <idc.idc>

// 输入：
//   ea=起始有效地址
// 输出：
//   返回未命名代码片段有效地址；如果没找到，则返回 BADADDR。
static JumpNotFunction(ea)
{
  auto name;
  do {
    if (!isCode(GetFlags(ea)) || Byte(ea) == 0x90 || Byte(ea) == 0xCC) // 过滤无效代码
      ea = FindCode(ea,SEARCH_DOWN);
    name = GetFunctionName(ea);
    if (name == "" && ea != BADADDR) // 如果找到未定义函数，就跳转到该有效地址
    {
```

```
    Jump(ea);
   }
   else
    ea = FindFuncEnd(ea); // 如果是一定义函数，继续查找
  } while(name != "" && ea != BADADDR);
  return ea;
}

// 利用 JumpNotFunction 解决问题
static main(void)
{
  auto ea0,ea,ea_end,fok;
  ea0 = ScreenEA(); // 记住当前光标位置
  ea=MinEA(); // 从头开始
  ea_end = SegEnd(ea); // 防止地址越界
  Message("ea0=%lx\n",ea);
  do{
   ea = JumpNotFunction(ea); // 调用刚建立的函数
   if (ea != BADADDR)
    fok = MakeFunction(ea,BADADDR);
   if (!fok) {
    ea = FindFuncEnd(ea);
    Jump(ea);
   }
  } while (ea < ea_end && ea != BADADDR);
  Jump(ea0); // 恢复光标位置
}
```

将以上内容另存为 JumpNotFunction.idc 文件并复制到 idc 子文件夹下，就可以用 File->IDC file...打开运行了。
http://bbs.pediy.com/showthread.php?t=170272

浅谈 IDA 脚本在漏洞挖掘中的应用

博文作者：dragonltx[TSRC]

发布日期：2012-08-30

阅读次数：10922

博文内容：

[目录]

[1]- 前言

IDA 毫无疑问是逆向领域里的一大神器，无所不能。有人的地方就有江湖，有江湖的地方就有武器。那么，在逆向这个江湖中，IDC 和 IDAPython 就好比倚天剑和屠龙刀，威力无比。

在漏洞挖掘领域，IDA 同样能够大展身手。OpenRCE 上提供的 BugScam 脚本正是 IDC 应用的最好诠释，著名的 Paimei 也是应用了 IDApython。

接下来，笔者将以自己的经验来分享下两把利器“倚天剑”和“屠龙刀”的应用。

[2]- IDC 和 IDAPython 简介

事实上，没有哪一个应用程序能够满足每名用户的一切需求。应用开发者面临两种选择：要么满足用户提出的无止境的功能要求，要么提供一种方法，供用户解决问题。IDA 采用了后一种方法，它集成了一个脚本引擎，让用户从编程角度对 IDA 的操作进行全面控制。

IDA 脚本语言可看成是一种查询语言，它能够以编程方式访问 IDA 数据库的内容。IDA 的脚本语言叫做 IDC，之所以取这个名称，可能是因为它的语法与 C 语言的语法非常相似。

得 益于 IDA Pro 极为开放的构架，Gergely Erdelyi 和 Ero Carrera 在 2004 年发布了 IDAPython--一款 IDA Pro 的插件。通过这款插件，逆向工程师能够以 Python 脚本的形式访问 IDC 脚本引擎核心、完整的 IDA 插件 API，以及所有与 Python 捆绑在一 起的常见模块。IDAPython 无论是在商业产品中（例如 Zynamics 的 BinNavi），还是在一些开源项目中（例如 Paimei 和 PyEmu） 均有所应用。

[3]- 倚天剑：IDC 应用

如果各位读者对这篇文章感兴趣，应该都对 IDC 有了解。不过不了解也没关系，那就请先参考下相关资料[1]，里面有详细的 IDC 语言介绍，这里就不再进行介绍。

当我们想通过自动化运行 IDA 获取一些对漏洞挖掘有用的信息，而不是手工运行 IDA，该怎么做？

IDA 提供了如下两个函数，可以帮助我们实现自动化。

Wait

```
// Wait for the end of autoanalysis
// This function will suspend execution of IDC program
// till the autoanalysis queue is empty.

void    Wait            ();                     // Process all entries in the
                                                // autoanalysis queue
```

Exit

```
// Stop execution of IDC program, close the database and exit to OS
//      code - code to exit with.

void    Exit            (long code);            // Exit to OS
```

在 IDA 启动后，IDA 会执行一些自动分析操作。Wait 函数会等待，直到这些自动分析结束。该函数会挂起我们的 IDC 脚本，直到自动分析队列为空。当自动 分析队列为空时，就开始执行我们的 IDC 脚本。Exit 函数会结束 IDC 函数的执行，并将 idb 关闭，然后结束 IDA 主进程，相当 nice 的功能。

有了这两个函数后，还不够，革命尚未成功。IDA 还提供了丰富的命令行参数，帮助我们实现自动化。

```
-A      autonomous mode. IDA will not display dialog boxes.
        Designed to be used together with -S switch.

-S###   Execute a script file when the database is opened.
        The script file extension is used to determine which extlang
        will run the script.

        It is possible to pass command line arguments after the script name.

        For example: -S"myscript.idc argument1 \"argument 2\" argument3"

        The passed parameters are stored in the "ARGV" global IDC variable.
        Use "ARGV.count" to determine the number of arguments.
        The first argument "ARGV[0]" contains the script name.
```


“-A” 参数是自动模式，IDA 将不会显示对话框，是和“-S”参数一起使用。“-S”参数指定执行那个 IDC 脚本，后面可以跟 IDC 脚本的参数。参数会放在 ARGV 这个全局变量里，其中 ARGV[0]，存放的是 IDC 脚本名。IDA 还提供了-c 参数，用来反汇编一个文件[3]。

现在实现自动化的各个因素都凑齐了，“万事具备，只欠东风”，接下来是一个自动导出一个文件中的所有函数名、起始地址、结束地址的 IDC 脚本。

```
#include <idc.idc>

static main()
{

  auto addr, end, args, locals, frame, firstArg, name, ret ,handle, path, index, filename, outputfilename ,segaddr;

  addr = 0;
```

```
Wait();    //等待直到 IDA 自动分析完成

segaddr = MinEA();

Message("Base:%x\n",segaddr);

handle = fopen(ARGV[1],"w");

for( addr = NextFunction(addr); addr != BADADDR; addr = NextFunction(addr))

{
  name = Name(addr);

  end  = GetFunctionAttr(addr, FUNCATTR_END);
  if(substr(name,0,4) == "sub_")
  continue;

  Message("Function:%s, starts at %x,ends at %x\n", name, addr-segaddr, end-segaddr);

  fprintf(handle,"Function:%s, starts at %x,ends at %x\n", name, addr-segaddr, end-segaddr);
  }

  fclose(handle);

  Exit(0);
}
```

当我们以这样的命令行 idaq -c -A -S"dumpfunc.idc E:\func.txt" E:\test.dll 运行 IDA，结果就会自动保存在 E 盘的 func.txt 中，相当惬意吧！心动了吧，心动了就赶快行动吧！你可以尽情发挥自己的才 华，向 IDA 获取你想要的东西。

[4]- 屠龙刀：IDAPython 应用

在 《Python 灰帽子》[2]第十章中，Justin 提供了一种自动化获取驱动程序 IO 控制码的方法，不过该脚本是基于 Immunity Debugger 的库。笔者用 Immunity Debugger 加载驱动文件，发现加载失败。后来一想，既然是基于静态分析的方法，何必用 Immunity Debugger，IDA 才是静态分析领域的王者。下面探讨用 IDAPython 来实现自动获取驱动程序 IO 控制码的初级版程序。

[4.1] - 获取驱动程序设备名

通过 FindText 这个函数来查找包含“\\Device\\”这个函数的偏移地址，然后通过 GetString 来获取字符串，如果获取的字符串为空，继续查找。

```
def getDeviceName():
    """
    Get Device Name from a driver.
    @rtype:   void
    @returns: void
    """
ea = 0
while True:
 ea =  FindText(ea, SEARCH_NEXT | SEARCH_REGEX, 0, 0, "\\\\Device\\\\")
 string = GetString(ea, -1, ASCSTR_UNICODE)
 if string is None:
    continue
 else:
    #Message("Find in %x\n" % ea)
    Message("device is %s\n" % string)
   Break
```

[4.2] - 获取驱动分发函数地址

首先用 FindText 查找 mov dword ptr [edx+70h], offset sub_11010 类似这种形式的指令，通过正则匹配查找。找到后，用 GetOperandValue 函数获取第二个操作数的值，即是分发函数的地址。

```
def getDispatchAddress():
    """
    Get Device Dispatch Address from a driver.
    @rtype:   int
    @returns: Dispatch Address
    """
ea = 0
ea =  FindText(ea,  SEARCH_DOWN |SEARCH_NEXT | SEARCH_REGEX, 0, 0, "mov *dword *ptr *\\[[a-zA-Z]* *\\+ *70h\\],[a-zA-Z0-9_ ]*")
#ea =  FindText(ea, SEARCH_NEXT | SEARCH_REGEX, 0, 0, "test *[a-zA-Z]*, +[a-zA-Z]*")
#Message("Find in %x\n" % ea)
if ea == BADADDR:
    Message("Cann't find the Dispatch address")
    address = BADADDR
else:
    address = GetOperandValue(ea,1)
    Message("Dispatch address is %x\n" % address)
```

```
return address
```

[4.3] - 获取函数内所有指令或指令偏移

通过 GetFunctionAttr 获取函数的结束地址，再通过 ItemSize 来获取每条指令的大小，然后循环遍历即可获得这个函数的所有指令的偏移地址。这边先获取所有指令的偏移地址，而不是指令，下面获取 io 控制码会用到。

```
def getFunctionInstructions():
    """
    Get All Instructions from a function.
    Here,Just Get All Instructions Offset,and store them in list
    @rtype:   List
    @returns: List of All Instructions
    """
    Instructions = []
    DispatchBeginAddress = getDispatchAddress()
    if DispatchBeginAddress == BADADDR:
       Message("Cann't find the Function Instructions List")
       return None
    DispatchEndAddress = GetFunctionAttr(DispatchBeginAddress,FUNCATTR_END)
    i =  DispatchBeginAddress
```

```
while True:
  #Instructions.append(GetDisasm(i))
  Instructions.append(i)
  tmp = i + ItemSize(i)
  if tmp < DispatchEndAddress:
     i = i + ItemSize(i)
  else:
     break
address = i
return Instructions
```

[4.4] - 获取驱动程序的所有 IO 控制码

获 取分发函数的所有指令偏移后，倒序查找。如果碰到是 jz 或者是 je 的，且接下来是 cmp 的指令，并且比较操作的寄存器是否一样，一样的话，则把 io 控制码 存储。（这样还是不够准确的，如果碰到其他的 jz 且连着 jmp 的指令，但不是 io 控制码。纯自动分析有时候不能识别）。

```
def getIoctlCode():
    """
    Get All IoctlCodes from a driver.
    @rtype:   List
```

```
@returns: List of All IoctlCodes

"""

isConditionalJmp             = False

isFirst                      = True

BaseRegister                 = None

OperRegister                 = None

IoctlCode                    = []

DispatchFunctionInstructions = []

DispatchFunctionInstructions = getFunctionInstructions()[::-1]

if DispatchFunctionInstructions == None:

   Message("Cann't get the IoctlCodes")

   return

for i in DispatchFunctionInstructions:

  #Message("The instrucion of this function is %x\n" % i)

  mnem = GetMnem(i)

  if "jz" in mnem or "je" in mnem:

     isConditionalJmp = True

     continue



  if "cmp" in mnem and isConditionalJmp and isFirst:

     sisConditionalJmp = False
```

```
    BaseRegister  = GetOpnd(i,0)
    IoctlCode.append(GetOperandValue(i,1))
    isFirst = False
    continue

if "cmp" in mnem and isConditionalJmp and not isFirst:
    isConditionalJmp = False
    OperRegister  =  GetOpnd(i,0)
    if OperRegister == BaseRegister:
        IoctlCode.append(GetOperandValue(i,1))

for i in IoctlCode:
    Message("The ioctlcode of this driver is %x\n" % i)
```

[4.5]不足与缺陷

上面实现的自动获取 io 控制码的比较简单，有些情况没有考虑到，算是初级版。Switch 反汇编的形式有很多种，上面只是考虑了 cmp 的形式。有兴趣的读者可以继续深入挖掘。上面的 IDAPython 脚本可以在这里（http://bbs.pediy.com/showthread.php?t=153965）获取到，里面还有对函数的解释。

[5]- 结语

本文主要对 IDA 脚本在漏洞挖掘领域应用进行简单的探讨，主要起到抛砖引玉的效果。希望对给位读者有所帮助。如果你有更好的思路，可以跟我探讨。

“思想有多远，就能走多远”。尽情发挥你的奇思妙想，在漏洞挖掘的海洋里尽情畅游吧！

References

[1]IDA 权威指南

[2]Python 灰帽子--黑客与逆向工程师的 Python 编程之道

[3]IDA Pro Documentation

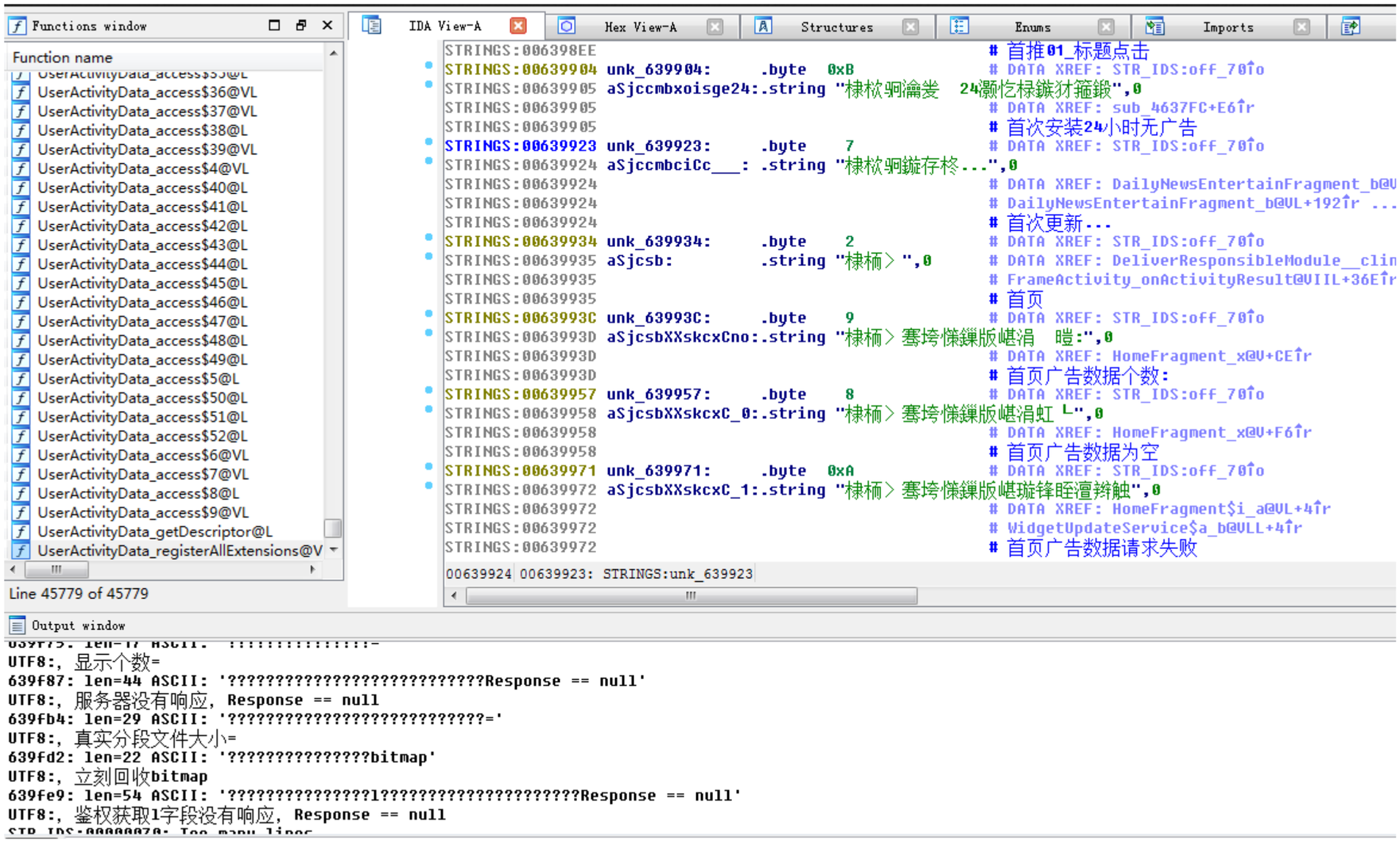
http://security.tencent.com/index.php/blog/msg/4

Android IDA 脚本解中文字串

前几天想捣鼓移动安全，看了版主的书受益匪浅
习惯了 IDA 逆向，但是在破解过程中中文字串不能显示感觉很纠结

```
import idautils
from idaapi import *
s=Strings()
for i in s:
  print "%x: len=%d ASCII: '%s'" % (i.ea, i.length, str(i))
  gbkstr=str(i).decode('utf-8','ignore').encode('gbk','ignore')
  MakeComm(i.ea,gbkstr)
  Message("UTF8:")
  Message(gbkstr)
  Message("\n")
```

IDA python 的 IDE 很纠结，用 pring 函数只能是 ASCII 模式，最后用了 IDA PYTHON 的 MESSAGE()函数，刚好



保存为.py 之后，每次都要选择文件有点麻烦。

Syphurith

這個是編碼問題，需要手動添加 GBK 編碼之類。
先找到 Options->ASCII String Style..。選擇 Change Encoding。
打開的窗口中有至少四個選項：<default>,<no conversion>,UTF8,UTF16LE。很顯然是沒有 GBK 什麼的。
在列表上右鍵，選擇 Insert。添加編碼的名字，如 GBK，OK。
編碼加入後需要選中他。ASCII String Style 的窗體上，Change Encoding 右側有 Set Default Encodings。
8 位寬的選擇對應的 OEM 集，如剛剛加入的 GBK，16 位寬的選擇 UTF16LE 之類或者保持 no Convertion 不變。
這個辦法剛纔個人在 IDA65 上測試通過。
http://bbs.pediy.com/showthread.php?t=99436
http://bbs.pediy.com/showthread.php?t=193195

菜鸟 IDA python 调试脚本

Wruih
IDA 动态调试下断点 还是比较费劲的，写个脚本可能更好一些

```
import idc
import idaapi
import struct

#idaapi.dbg_write_memory(ea,buf)
idaapi.enable_extlang_python(1)
md={}
add=0
size=0
index=0
cross_refs=0
findname=["MZ","PE"]
for func in findname:
        addr=LocByName(func)
        if addr!=BADADDR:
            cross_refs=CodeRefsTo(addr,0)
            print "got it"
        for ref in range(cross_refs):
            print "%08x"% ref
            SetColor(ref,CiC_ITEM,0x0000ff)
def createfile():
        global md
    #print "code %d  \ninput %d" % (GetRegValue("eax") ,GetRegValue("edi"))
    ea1=GetRegValue("esp")+4
        buf=""
```

```
    a=idc.Dword(ea1)

    #print "s%x\n" % a
        for i in range(50):
          k=idc.Byte(a+i)
          buf=buf+chr(k)
        print "filename  ___%s_____\n" % buf
    #SetRegValue(0x11113333,"edx")
        md[ea1]=buf

    return 1
print "start"
def readfile():
  global add, size,index
  index=index+1
  if GetRegValue("eip")==0x20095280:
    add=Dword(GetRegValue("esp")+8)
    size=Dword(GetRegValue("esp")+12)
  if GetRegValue("eip")==0x20095338:
    b=""
    for i in range(size) :
        b=b+chr(Byte(add+i))
    v="%d" %index
    file0=r"c:\hk"+v+"dat"
    file1=r"c:\hk"+v+"dat"
    f1=open(file0,"wb")
    f1.write(b)
    f1.close()
    f1=open(file1,"w")
    f1.write(b)
    f1.close()
  return 1
ea=0x7c00 #fopen
idc.AddBpt(ea)
idc.SetBptAttr(ea,  BPTATTR_FLAGS, GetBptAttr(ea, BPTATTR_FLAGS)&0xfffe)
idc.SetBptCnd(ea, 'createfile()')

ea=0x20001000 #fopen
idc.AddBpt(ea)
idc.SetBptAttr(ea,  BPTATTR_FLAGS, GetBptAttr(ea, BPTATTR_FLAGS)&0xfffe)
idc.SetBptCnd(ea, 'createfile()')
```

```
ea=0x20095960 #fopen
idc.AddBpt(ea)
idc.SetBptAttr(ea,  BPTATTR_FLAGS, GetBptAttr(ea, BPTATTR_FLAGS)&0xfffe)
idc.SetBptCnd(ea, 'createfile()')

ea=0x20095280 #fopen
idc.AddBpt(ea)
idc.SetBptAttr(ea,  BPTATTR_FLAGS, GetBptAttr(ea, BPTATTR_FLAGS)&0xfffe)
idc.SetBptCnd(ea, 'readfile()')


ea=0x20095338 #fopen
idc.AddBpt(ea)
idc.SetBptAttr(ea,  BPTATTR_FLAGS, GetBptAttr(ea, BPTATTR_FLAGS)&0xfffe)
idc.SetBptCnd(ea, 'readfile()')


print "end"
```

http://www.pd521.com/thread-820-1-1.html

菜鸟 Dump Memory python 脚本

Wruih
直接用就行，改改地址啥的

```
from idaapi import *
from idc import *
import struct
def dump_memory(filename,start_add,size):
    fd=open(filename,"wb")
    for i in range(0,size,4):
        ea=start_add+i
        buf=idaapi.dbg_read_memory(ea,4)
        fd.write(buf)
    fd.close()
def write_memory(memoryfile,linefile,start_add):
    fd=open(memoryfile,'rb')
    fd2=open(linefile,'r')
    lines=fd2.readlines()
    for i in lines:
        pos=int(i[0:len(i)-2],16)
```

```
            ea=start_add+pos
            fd.seek(pos)
            buf=fd.read(1)
            buf=idaapi.dbg_write_memory(ea,buf)
        fd2.close()
        fd.close()
def write_all_memory(memoryfile,start_add):
        fd=open(memoryfile,'rb')
        fd.seek(0,2)
        pos=fd.tell()
        fd.seek(0,0)
        for i in range(pos):
            ea=start_add+i
            buf=fd.read(1)
            buf=idaapi.dbg_write_memory(ea,buf)
        fd.close()
def write_pwd():
        ea=0x3f6eee
        idaapi.dbg_write_memory(ea,struct.pack('=H',0x0035))
        ea=ea+2
        idaapi.dbg_write_memory(ea,struct.pack('=H',0x0031))
        ea=ea+2
        idaapi.dbg_write_memory(ea,struct.pack('=H',0x0032))
        ea=ea+2
        idaapi.dbg_write_memory(ea,struct.pack('=H',0x0035))
        ea=ea+2
        idaapi.dbg_write_memory(ea,struct.pack('=H',0x0031))
        ea=ea+2
        idaapi.dbg_write_memory(ea,struct.pack('=H',0x0032))
        ea=ea+2
        idaapi.dbg_write_memory(ea,struct.pack('=H',0x0035))
        ea=ea+2
        idaapi.dbg_write_memory(ea,struct.pack('=H',0x0031))
        ea=ea+2
        idaapi.dbg_write_memory(ea,struct.pack('=H',0x0032))
filename='memory'
linefile='result_compare.txt'
start=0x7c00
size=0x300000
#dump_memory(filename,start,size)
write_memory(filename,linefile,start)
#write_all_memory(filename,start)
```

```
#print idaapi.dbg_get_registers()
#write_pwd()
print "run_ok"
```
http://www.pd521.com/thread-821-1-1.html

IDAPython 让你的生活更滋润 part1 and part2

蒸米 · 2016/01/06 9:11

0x00 序

今天在网上看到平底锅 blog 上 Josh Grunzweig 发表了一系列关于利用 IDAPython 分析 malware 的教程。感觉内容非常不错，于是翻译成中文与大家一起分享。原文地址：

Part1： http://researchcenter.paloaltonetworks.com/2015/12/using-idapython-to-make-your-life-easier-part-1/

Part2： http://researchcenter.paloaltonetworks.com/2015/12/using-idapython-to-make-your-life-easier-part-2/

0x01 背景

作为一名 malware 逆向工程师，我的日常活动就是使用 IDA Pro。这并不奇怪，因为 IDA Pro 可以说是行业标配（尽管它的替代品，如 radare2 和 hopper 也*越来越受欢迎*）。IDA 其中一个非常强大的功能就是可以使用 Python 脚本(又被称为 IDAPython)。用户可以通过 IDAPython 调用大量的 IDA API。当然，用户还可以通过使用 IDAPython 获取到脚本语言提供的各种功能。

不幸的是，只有少量的关于 IDAPython 的资料，仅有的一些资料如下：

- Chris Eagle 的 The IDA Pro Book
- Alex Hanel 的 The Beginner’s Guide to IDAPython
- Magic Lantern 的 IDAPython Wiki

0x02 利用 IDAPython 解决字符串加密问题

为了能提供更多的教程给分析师，我准备写一篇带例子的分析文章供大家学习。在本系列的第一部分，我将教大家编写一个脚本用来解决一个 malware 样本的多处字符串混淆问题。

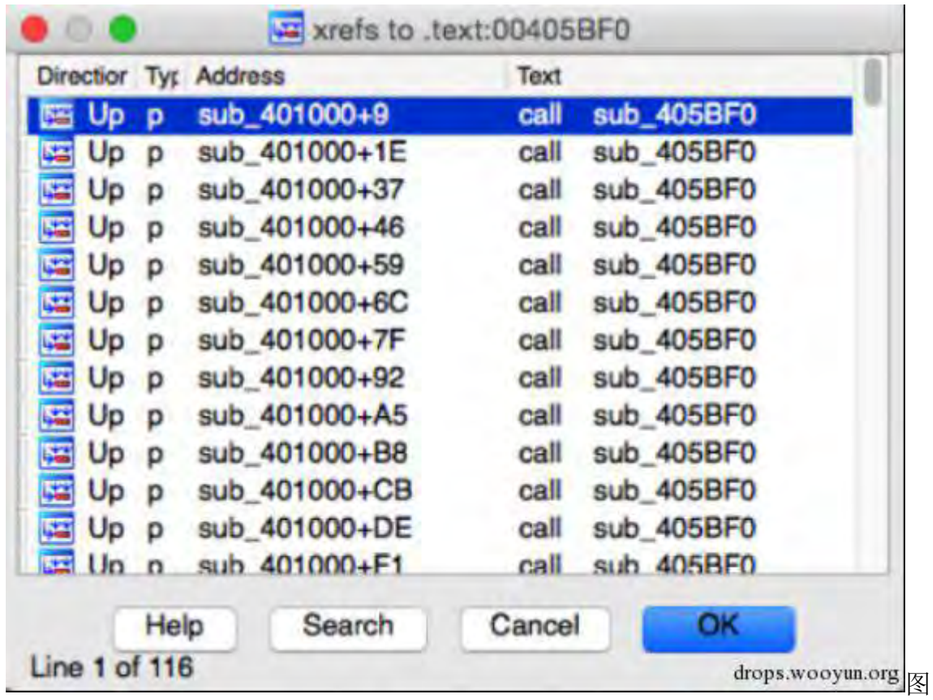
在逆向分析一个病毒样本的时候，我遇到了这样一个函数：

```
.text:00405C7D ; ---------------------------------------------------------------------------
.text:00405C7D
.text:00405C7D loc_405C7D:                             ; CODE XREF: sub_405BF0+6B'j
.text:00405C7D                 mov     eax, 1
.text:00405C82                 cmp     edi, eax
.text:00405C84                 jle     short loc_405CA3
.text:00405C86                 mov     ecx, dword_41C050
.text:00405C8C                 shl     ecx, 9
.text:00405C8F                 lea     ecx, unk_41E477[ecx]
.text:00405C95
.text:00405C95 loc_405C95:                             ; CODE XREF: sub_405BF0+B1'j
.text:00405C95                 mov     dl, [eax+esi]
.text:00405C98                 xor     dl, [esi]
.text:00405C9A                 inc     eax
.text:00405C9B                 cmp     eax, edi
.text:00405C9D                 mov     [ecx+eax-1], dl
.text:00405CA1                 jl      short loc_405C95
.text:00405CA3
.text:00405CA3 loc_405CA3:                             ; CODE XREF: sub_405BF0+8B'j
.text:00405CA3                                         ; sub_405BF0+94'j
.text:00405CA3                 push    offset CriticalSection ; lpCriticalSection
.text:00405CA8                 call    ds:LeaveCriticalSection
.text:00405CAE                 mov     eax, dword_41C050
.text:00405CB3                 shl     eax, 9
.text:00405CB6                 add     eax, offset unk_41E478
.text:00405CBB                 pop     edi
.text:00405CBC                 retn
.text:00405CBC sub_405BF0      endp
.text:00405CBC
.text:00405CBC ; ---------------------------------------------------------------------------
```

drops.wooyun.org

图片 1 字符串解密函数

根据以往的经验，我怀疑这个函数是用来进行解密的。关于这个函数大量的引用证实了我的猜想。



drops.wooyun.org

图 2 大量的对可疑函数的引用

在图 2 中，我们可以看到有 116 处对这个函数的引用。每当这个函数被调用时，都有一段数据作为参数通过 ESI 寄存器提供给这个函数。

```
.text:00401004          mov     esi, offset unk_418BE0
.text:00401009          call    sub_405BF0
.text:0040100E          mov     ebp, ds:LoadLibraryA
.text:00401014          push    eax             ; lpLibFileName
.text:00401015          call    ebp ; LoadLibraryA
.text:00401017          mov     esi, offset unk_418BF0
.text:0040101C          mov     edi, eax
.text:0040101E          call    sub_405BF0
.text:00401023          push    eax             ; lpProcName
.text:00401024          push    edi             ; hModule
.text:00401025          mov     edi, ds:GetProcAddress
.text:0040102B          call    edi ; GetProcAddress
.text:0040102D          mov     esi, offset unk_418C08
.text:00401032          mov     dword_41D020, eax
.text:00401037          call    sub_405BF0
.text:0040103C          push    eax             ; lpLibFileName
.text:0040103D          call    ebp ; LoadLibraryA
.text:0040103F          mov     esi, offset unk_418C18
.text:00401044          mov     ebx, eax
.text:00401046          call    sub_405BF0
.text:0040104B          push    eax             ; lpProcName
.text:0040104C          push    ebx             ; hModule
.text:0040104D          call    edi ; GetProcAddress
.text:0040104F          mov     esi, offset unk_418C2C
.text:00401054          mov     dword_41D01C, eax
.text:00401059          call    sub_405BF0
.text:0040105E          push    eax             ; lpProcName
.text:0040105F          push    ebx             ; hModule
.text:00401060          call    edi ; GetProcAddress
.text:00401062          mov     esi, offset unk_418C44
.text:00401067          mov     dword_41D018, eax
.text:0040106C          call    sub_405BF0
```


图 3 可疑的函数（405BF0）被调用的实例

在这个时候，我已经非常肯定这个函数是 malware 用来在运行时进行字符串解密的函数了。当我们面临这种情况时，我们一般有如下几种选择：

1. 我可以手动解密然后重命名这些字符串。
2. 我可以动态调试这个样本然后重命名遇到的字符串
3. 我可以写一个脚本用来解密并且重命名这些字符串

如果 malware 只解密了很少的几个字符串的话，我会选择第一种或者第二种方法。但是，根据之前确认的，这个函数被调用了 116 次，所以采用 IDAPython 脚本来解决问题会更靠谱一些。

解决字符串混淆问题的第一步是确认和重写解密函数。幸运的是，这个解密函数非常的简单。这个函数只是把数据的第一个字符当做 XOR 算法的 key 用来解密剩余的数据。

E4 91 96 88 89 8B 8A CA 80 88 88

在上面这个例子中，我们把 E4 作为 key 来异或剩余的数据。最后的结果是”urlmon.dll”。在 Python 中，我们可以把这个解密函数重写为：

```
def decrypt(data):
        length = len(data)
        c = 1
        o = ""
        while c < length:
                o += chr(ord(data[0]) ^ ord(data[c][/c]))
                c += 1
        return o
```

可以看到，我们的测试脚本可以得到我们所期望的结果：

```
1>>> from binascii import*
2>>> d = unhexlify("E4 91 96 88 89 8B 8A CA 80 88 88".replace(" ",''))
3>>> decrypt(d)
4'urlmon.dll'
```

我们要做的下一步工作就是确认哪些代码引用了这个解密函数，并且提取作为参数的数据。获取到函数的引用非常的简单，只需要使用 XrefsTo()这 个 API 函数就能达到我们的目的。在这个脚本中，我将会在脚本中硬编码这个地址。作为测试，我先将这些地址用 16 进制打印出来：

```
1 for addr in XrefsTo(0x00405BF0, flags=0):
2         print hex(addr.frm)
3
4 Result:
5 0x401009L
6 0x40101eL
7 0x401037L
8 0x401046L
9 0x401059L
100x40106cL
110x40107fL
12<truncated>
```

获取到这些交叉引用的参数并且提取这些原始数据需要一些技巧，但绝非很难。第一件我们想要做的事是获取”mov esi, offset unk_??”指令中的偏移地址，因为这个指令会把参数传递给解密函数。为了做到这点，我们需要找到调用解密函数指令的前一个指令。找到这个指令后，我们 可以使用 GetOperandValue() 这个指令得到这个偏移地址的值。如下面的代码所示：

```
1 def find_function_arg(addr):
2     while True:
3         addr = idc.PrevHead(addr)
4         if GetMnem(addr) == "mov" and "esi" in GetOpnd(addr, 0):
5             print “We found it at 0x%x” %GetOperandValue(addr, 1)
6             break
7
8 Example Results:
9 Python>find_function_arg(0x00401009)
10We found it at 0x418be0
```

现在我们只需要将字符串从那个偏移地址中提取出来即可。正常来说我们会使用 GetString()这个 API 函数，但是在这个问题中这些字符串都是 原始的二进制数据，因此使用这个 API 可能不太合适。解决方案是我们自己编写一个函数，然后一个字符一个字符的读取数据直到碰到空的终止符为止。代码如 下：

```
1def get_string(addr):
2     out =""
3     while True:
```

```
        if Byte(addr) != 0:
            out += chr(Byte(addr))
        else:
            break
        addr += 1
    return out
```

最后，我们将所有的代码放在一起：

```
def find_function_arg(addr):
    while True:
        addr = idc.PrevHead(addr)
        if GetMnem(addr) == "mov" and "esi" in GetOpnd(addr, 0):
            return GetOperandValue(addr, 1)
    return ""

def get_string(addr):
    out = ""
    while True:
        if Byte(addr) != 0:
            out += chr(Byte(addr))
        else:
            break
        addr += 1
    return out

def decrypt(data):
    length = len(data)
    c = 1
    o = ""
    while c < length:
        o += chr(ord(data[0]) ^ ord(data[c][/c]))
        c += 1
    return o

print "[*] Attempting to decrypt strings in malware"
for x in XrefsTo(0x00405BF0, flags=0):
    ref = find_function_arg(x.frm)
    string = get_string(ref)
    dec = decrypt(string)
    print "Ref Addr: 0x%x | Decrypted: %s" % (x.frm, dec)

```

```
Results:
[*] Attempting to decrypt strings in malware
Ref Addr: 0x401009 | Decrypted: urlmon.dll
Ref Addr: 0x40101e | Decrypted: URLDownloadToFileA
Ref Addr: 0x401037 | Decrypted: wininet.dll
Ref Addr: 0x401046 | Decrypted: InternetOpenA
Ref Addr: 0x401059 | Decrypted: InternetOpenUrlA
Ref Addr: 0x40106c | Decrypted: InternetReadFile
<truncated>
```

我们可以看到所有解密后的字符串。如果我们可以进一步给字符串的引用地址和加密数据提供解密后的字符串作为注释就更完美了。想要做到这一点，我们需要 MakeComm()这个 API 函数。增加这样两行代码就会给程序加入必要的注释：

```
MakeComm(x.frm, dec)
MakeComm(ref, dec)
```

增加了这一步后，我们能够非常清晰的看到交叉引用的数据。如下图所示，我们可以很轻松的分辨出哪些字符串被引用了：

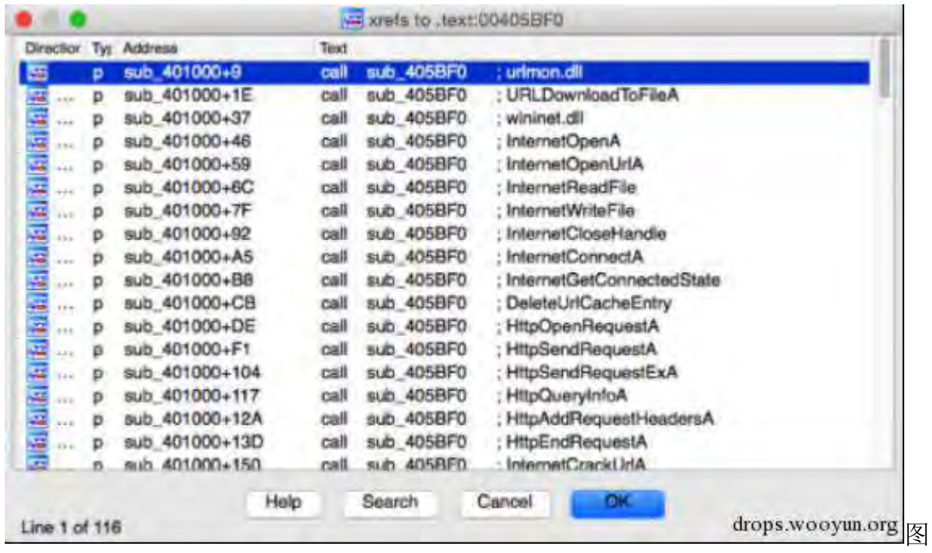


图 4 运行完脚本后的字符串交叉引用界面

除此之外，我们在反汇编代码中也能看到这些解密后的字符串作为注释：

```
.text:00401004                mov     esi, offset unk_418BE0
.text:00401009                call    sub_405BF0      ; urlmon.dll
.text:0040100E                mov     ebp, ds:LoadLibraryA
.text:00401014                push    eax             ; lpLibFileName
.text:00401015                call    ebp ; LoadLibraryA
.text:00401017                mov     esi, offset unk_418BF0
.text:0040101C                mov     edi, eax
.text:0040101E                call    sub_405BF0      ; URLDownloadToFileA
.text:00401023                push    eax             ; lpProcName
.text:00401024                push    edi             ; hModule
.text:00401025                mov     edi, ds:GetProcAddress
.text:0040102B                call    edi ; GetProcAddress
.text:0040102D                mov     esi, offset unk_418C08
.text:00401032                mov     dword_41D020, eax
.text:00401037                call    sub_405BF0      ; wininet.dll
.text:0040103C                push    eax             ; lpLibFileName
.text:0040103D                call    ebp ; LoadLibraryA
.text:0040103F                mov     esi, offset unk_418C18
.text:00401044                mov     ebx, eax
.text:00401046                call    sub_405BF0      ; InternetOpenA
.text:0040104B                push    eax             ; lpProcName
.text:0040104C                push    ebx             ; hModule
.text:0040104D                call    edi ; GetProcAddress
.text:0040104F                mov     esi, offset unk_418C2C
.text:00401054                mov     dword_41D01C, eax
.text:00401059                call    sub_405BF0      ; InternetOpenUrlA
.text:0040105E                push    eax             ; lpProcName
.text:0040105F                push    ebx             ; hModule
.text:00401060                call    edi ; GetProcAddress
.text:00401062                mov     esi, offset unk_418C44
.text:00401067                mov     dword_41D018, eax
.text:0040106C                call    sub_405BF0      ; InternetReadFile
```

图 5 运行完脚本后的反汇编代码

0x03 利用 IDAPython 解决函数/库调用的哈希混淆问题

在反编译中我们经常会见到 shellcode 和 malware 使用哈希算法来混淆加载的函数或者库。比如逆向工程师们经常会在 shellcode 中 看到混淆后的函数名。总的来说，整个过程是很简单的。代码在运行时会先加载 knerel32.dll。然后，它会用这个加载的镜像去识别并存储 LoadLibraryA 函数，这函数是用来加载更多的库和函数的。这种特定的技术一般采用某种哈希算法来识别函数的。最常用的哈希算法一般是 CRC32，当然，其他的一些变种算法，如 ROR13，也是非常常见的。

比如说，当我逆向一个 malware 的某一部分内容的时候，我看到了如下的代码：

```
.text:004125A0          push    1
.text:004125A2          push    7695D1CCh
.text:004125A7          push    edx
.text:004125A8          call    load_function
.text:004125AD          add     esp, 0Ch
.text:004125B0          mov     dword_41A59C, eax
.text:004125B5          cmp     eax, ebx
.text:004125B7          jz      loc_4124E6
.text:004125BD          push    1
.text:004125BF          push    0E62E824Dh
.text:004125C4          push    esi
.text:004125C5          call    load_function
.text:004125CA          add     esp, 0Ch
.text:004125CD          mov     dword_41A3D8, eax
.text:004125D2          cmp     eax, ebx
.text:004125D4          jz      loc_4124E6
.text:004125DA          push    1
.text:004125DC          push    9A80E589h
.text:004125E1          push    esi
.text:004125E2          call    load_function
.text:004125E7          add     esp, 0Ch
.text:004125EA          mov     dword_41A56C, eax
.text:004125EF          cmp     eax, ebx
.text:004125F1          jz      loc_4124E6
.text:004125F7          push    1
.text:004125F9          push    0F3B07FCCh
.text:004125FE          push    edi
.text:004125FF          call    load_function
.text:00412604          add     esp, 0Ch
.text:00412607          mov     dword_41A380, eax
.text:0041260C          cmp     eax, ebx
.text:0041260E          jz      loc_4124E6
.text:00412614          mov     edi, [ebp+var_18]
.text:00412617          push    1
.text:00412619          push    301BF0h
.text:0041261E          push    edi
.text:0041261F          call    load_function
.text:00412624          add     esp, 0Ch
.text:00412627          mov     dword_41A544, eax
.text:0041262C          cmp     eax, ebx
.text:0041262E          jz      loc_4124E6
.text:00412634          mov     eax, [ebp+var_4]
.text:00412637          push    1
.text:00412639          push    0A9290135h
.text:0041263E          push    eax
.text:0041263F          call    load_function
.text:00412644          add     esp, 0Ch
.text:00412647          mov     dword_41A38C, eax
```


图片 6 malware 使用 CRC32 哈希算法来动态的加载函数

因为 0xEDB88320 这个常数是 CRC32 算法的常用参数。所以我们可以判断出这个例子使用了 CRC32 哈希算法。

```
.text:00405590 crc32           proc near               ; CODE XREF: sub_405640+C p
.text:00405590                 cmp     byte_41AA9E, 0
.text:00405597                 jnz     locret_405631
.text:0040559D                 mov     byte_41AA9E, 1
.text:004055A4                 xor     ecx, ecx
.text:004055A6
.text:004055A6 loc_4055A6:                             ; CODE XREF: crc32+9B j
.text:004055A6                 mov     eax, ecx
.text:004055A8                 shr     eax, 1
.text:004055AA                 test    cl, 1
.text:004055AD                 jz      short loc_4055B4
.text:004055AF                 xor     eax, 0EDB88320h
.text:004055B4
.text:004055B4 loc_4055B4:                             ; CODE XREF: crc32+1D'j
.text:004055B4                 test    al, 1
.text:004055B6                 jz      short loc_4055C1
.text:004055B8                 shr     eax, 1
.text:004055BA                 xor     eax, 0EDB88320h
.text:004055BF                 jmp     short loc_4055C3
.text:004055C1 ; ---------------------------------------------------------------------------
.text:004055C1
.text:004055C1 loc_4055C1:                             ; CODE XREF: crc32+26'j
.text:004055C1                 shr     eax, 1
.text:004055C3
.text:004055C3 loc_4055C3:                             ; CODE XREF: crc32+2F'j
.text:004055C3                 test    al, 1
.text:004055C5                 jz      short loc_4055D0
.text:004055C7                 shr     eax, 1
.text:004055C9                 xor     eax, 0EDB88320h
.text:004055CE                 jmp     short loc_4055D2
.text:004055D0 ; ---------------------------------------------------------------------------
.text:004055D0
.text:004055D0 loc_4055D0:                             ; CODE XREF: crc32+35'j
.text:004055D0                 shr     eax, 1
.text:004055D2
.text:004055D2 loc_4055D2:                             ; CODE XREF: crc32+3E'j
.text:004055D2                 test    al, 1
.text:004055D4                 jz      short loc_4055DF
.text:004055D6                 shr     eax, 1
.text:004055D8                 xor     eax, 0EDB88320h
.text:004055DD                 jmp     short loc_4055E1
.text:004055DF ; ---------------------------------------------------------------------------
```

drops.wooyun.org

图片 7 确认 CRC32 算法

通过图 7 我们可以确定这个算法是 CRC32 算法。现在，算法和函数已经确定了。我们可以通过交叉引用（ida 中按 x）的数量来确定这个函数被被调用 了多少次。可以看到这个函数一共被调用了 190 次。显然，手动的解密并重命名这些哈希值并不是我们想要的。因此，我们可以使用 IDAPython 来帮我们 解决。

第一步实际上并不需要 IDAPython，但是它用到了 Python。为了验证哪个哈希值对应哪个函数，我们需要生成一个 windows 通用函数哈希列表。想要做到这点，我们只需要获取一个 windows 通用库的列表，然后遍历这些库的函数列表。代码如下：

```
1 def get_functions(dll_path):
2     pe = pefile.PE(dll_path)
3     if ((not hasattr(pe, 'DIRECTORY_ENTRY_EXPORT')) or (pe.DIRECTORY_ENTRY_EXPORT is None)):
4         print "[*] No exports for %s" % dll_path
5         return []
6     else:
7         expname = []
8         for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
9             if exp.name:
10                expname.append(exp.name)
11        return expname
```

我们随后可以得到函数名字的列表，然后计算他们的 CRC32 的哈希值。代码如下：

```
def calc_crc32(string):
    return int(binascii.crc32(string) & 0xFFFFFFFF)
```

最后我们将结果写入一个 JSON 格式的文件中，并且命名为"output.json"。这个 JSON 文件包含了一个非常大的字典，采用如下的格式：

```
HASH => NAME
```

完整版的代码如下：

https://github.com/pan-unit42/public_tools/blob/master/ida_scripts/gen_function_json.py

当这个文件生成之后，我们可以返回 IDA，并且继续编写我们的 IDAPython 脚本。我们脚本第一步要做的事情是读取我们之前创建的 'output.json'这个 JOSON 数据文件。不幸的是，JSON 对象并不支持整数作为 key，因此当数据被加载后，我们需要手动把 key 从字符串 转换为整数。代码如下：

```
for k,v in json_data.iteritems():
    json_data[int(k)] = json_data.pop(k)
```

当这些数据被加载后，我们将会创建一个枚举对象保存了哈希值与函数名的对应关系。（想要了解更多的关于枚举对象的信息，我推荐你阅读这篇教程：

http://www.cprogramming.com/tutorial/enum.html

使用枚举对象，我们可以找到一个整数对应的字符串，比如说 CRC32 哈希值对应的函数名。为了在 IDA 中创建新的枚举对象，我们可以使用 AddEnum()这个函数。为了让这个脚本更加健壮，我们先使用 GetEnum()函数来检测用来枚举的值是否已经存在。

```
enumeration = GetEnum("crc32_functions")
if enumeration == 0xFFFFFFFF:
        enumeration = AddEnum(0, "crc32_functions", idaapi.hexflag())
```

这个枚举的值随后将会被修改。下一步要干的事情是根据函数的哈希值来确定真实的函数地址。这一部分看起来很像第一部分的内容。我们通过观察这个函数的结构可以发现 CRC32 哈希值是这个加载函数的第二个参数。

```
.text:004124F3                push    1
.text:004124F5                push    4C8A5B22h
.text:004124FA                push    edx
.text:004124FB                call    load_function
```

图片 8 传递给 load_function()的参数

同样的，我们还是枚举之前的指令来寻找函数的第二个参数。当我们找到后，我们通过 output.json 中的 JSON 数据来进行检测，并且确保有一个函数名对应了这个哈希值。代码如下：

```
for x in XrefsTo(load_function_address, flags=0):
        current_address = x.frm
        addr_minus_20 = current_address-20
        push_count = 0
```

```
        while current_address >= addr_minus_20:
            current_address = PrevHead(current_address)
            if GetMnem(current_address) == "push":
                push_count += 1
                data = GetOperandValue(current_address, 0)
                if push_count == 2:
                    if data in json_data:
                        name = json_data[data]
```

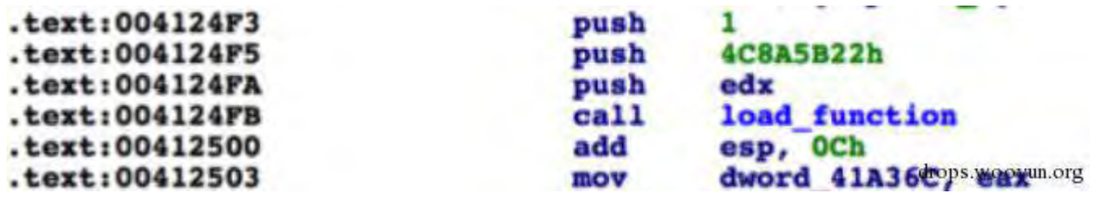
这个时候，我们使用 AddConstEx()这个函数将 CRC32 哈希和函数名加入我们之前创建的枚举对象中。

```
AddConstEx(enumeration, str(name), int(data), -1)
```

当这个数据加入到枚举对象中后，我们可以将 CRC32 的哈希值转换为对应的枚举名字了。下面的两个函数一个是用来将一个整数转换成对应的枚举数据，另一个是用来将某个地址的数据转换成对应的枚举数据。

```
def get_enum(constant):
    all_enums = GetEnumQty()
    for i in range(0, all_enums):
        enum_id = GetnEnum(i)
        enum_constant = GetFirstConst(enum_id, -1)
        name = GetConstName(GetConstEx(enum_id, enum_constant, 0, -1))
        if int(enum_constant) == constant: return [name, enum_id]
        while True:
            enum_constant = GetNextConst(enum_id, enum_constant, -1)
            name = GetConstName(GetConstEx(enum_id, enum_constant, 0, -1))
            if enum_constant == 0xFFFFFFFF:
                break
            if int(enum_constant) == constant: return [name, enum_id]
    return None

def convert_offset_to_enum(addr):
    constant = GetOperandValue(addr, 0)
    enum_data = get_enum(constant)
    if enum_data:
        name, enum_id = enum_data
        OpEnumEx(addr, 0, enum_id, 0)
        return True
    else:
        return False
```

当我们把这个枚举转换完成后，我们来研究一下如何修改 DWORD 处的值，因为 DWORD 处的值保存了加载后的函数地址。



图片 9 当加载完函数后，程序将函数地址存储到了 DWORD 地址

为了做到这一点，我们不光需要遍历之前的指令，还要查找之后的指令，也就是将 eax 存储到一个 DWORD 地址的指令。当我们发现这条指令之后，我们可以给这个 DWORD 地址重新命名成正确的函数名。为了防止冲突，我们在函数名前加上一个”d_”字符串。

```
address = current_address
while address <= address_plus_30:
    address = NextHead(address)
    if GetMnem(address) == "mov":
        if 'dword' in GetOpnd(address, 0) and 'eax' in GetOpnd(address, 1):
            operand_value = GetOperandValue(address, 0)
            MakeName(operand_value, str("d_"+name))
```

等这一切都做完后，我们会发现原来很难读懂的汇编代码变得很好理解了。如图所示：

图片 10 运行完脚本后的变化

现在，当我们看到 DOWRDS 列表的时候，就已经能得到真实的函数名字了。并且这些数据能够很好的帮助我们进行静态分析。

```
.data:0041A510 d_func_advapi32_cryptgethashparam dd ?  ; DATA XREF: sub_40ADB0+68↑r
.data:0041A510                                         ; sub_4122E0+12E4↑w
.data:0041A514 d_func_kernel32_resumethread dd ?       ; DATA XREF: sub_4122E0+928↑w
.data:0041A518 d_func_shlwapi_urlescapea dd ?          ; DATA XREF: sub_4121E0+78↑r
.data:0041A518                                         ; sub_4122E0+1532↑w
.data:0041A51C d_func_wininet_internetconnecta dd ?    ; DATA XREF: sub_40B7F0+73↑r
.data:0041A51C                                         ; sub_4122E0+CC1↑w
.data:0041A520 d_func_user32_flashwindow dd ?          ; DATA XREF: sub_404DB0+75↑r
.data:0041A520                                         ; sub_4122E0+7F1↑w
.data:0041A524 d_func_kernel32_getlasterror dd ?       ; DATA XREF: sub_4030E0+1FB↑r
.data:0041A524                                         ; sub_403F40+683↑r ...
.data:0041A528 d_func_msvcrt_malloc dd ?               ; DATA XREF: .text:00406A89↑r
.data:0041A528                                         ; sub_4068E0+1B7↑r ...
.data:0041A52C d_func_kernel32_getmodulefilenamea dd ? ; DATA XREF: sub_4030E0+62↑r
.data:0041A52C                                         ; sub_4122E0+8CE↑w ...
.data:0041A530 d_func_kernel32_thread32first dd ?      ; DATA XREF: sub_4122E0+CA1↑w
.data:0041A534 d_func_ntdll_memcmp dd ?                ; DATA XREF: sub_403F40+58B↑r
.data:0041A534                                         ; sub_409890+13D↑r ...
.data:0041A538 d_func_ntdll_ntquerysysteminformation dd ? ; DATA XREF: sub_404710+33↑r
.data:0041A538                                         ; sub_4122E0+FEE↑w
.data:0041A53C d_func_ntdll_memcpy dd ?                ; DATA XREF: sub_402A90+A5↑r
.data:0041A53C                                         ; sub_402A90+167↑r ...
.data:0041A540 d_func_ntdll_strncat dd ?               ; DATA XREF: sub_40BD80+9C↑r
.data:0041A540                                         ; sub_4122E0+512↑w
.data:0041A544 d_func_ntdll_rwqueueapcthread dd ?      ; DATA XREF: sub_40B220+CE↑r
.data:0041A544                                         ; sub_4122E0+347↑w
.data:0041A548 d_func_kernel32_process32first dd ?     ; DATA XREF: sub_40C160+42↑r
.data:0041A548                                         ; sub_4122E0+649↑w
.data:0041A54C d_func_kernel32_getcurrentprocessid dd ? ; DATA XREF: sub_402EF0+61↑r
.data:0041A54C                                         ; sub_406A20:loc_406AFB↑r ...
.data:0041A550 d_func_advapi32_regenumvaluew dd ?      ; DATA XREF: sub_4122E0+6C3↑w
.data:0041A554 d_func_ntdll_rtlgetversion dd ?         ; DATA XREF: sub_404A20+6D↑r
.data:0041A554                                         ; sub_4122E0+1643↑w
.data:0041A558 d_func_kernel32_createtoolhelp32snapshot dd ? ; DATA XREF: sub_40C160+E↑r
.data:0041A558                                         ; sub_4122E0+73D↑w
.data:0041A55C d_func_ole32_coinitializesecurity dd ?  ; DATA XREF: sub_405DD0+36↑r
.data:0041A55C                                         ; sub_4122E0+293↑w
.data:0041A560 d_func_oleaut32_sysallocstring dd ?     ; DATA XREF: sub_[illegible]
```


完整的代码如下：

```
1 import json
2
3 def get_enum(constant):
4     all_enums = GetEnumQty()
5     for i in range(0, all_enums):
6         enum_id = GetnEnum(i)
7         enum_constant = GetFirstConst(enum_id, -1)
8         name = GetConstName(GetConstEx(enum_id, enum_constant, 0, -1))
9         if int(enum_constant) == constant: return [name, enum_id]
10        while True:
11            enum_constant = GetNextConst(enum_id, enum_constant, -1)
12            name = GetConstName(GetConstEx(enum_id, enum_constant, 0, -1))
13            if enum_constant == 0xFFFFFFFF:
14                break
15            if int(enum_constant) == constant: return [name, enum_id]
16    return None
17
18
19def convert_offset_to_enum(addr):
```

```
    constant = GetOperandValue(addr, 0)
    enum_data = get_enum(constant)
    if enum_data:
        name, enum_id = enum_data
        OpEnumEx(addr, 0, enum_id, 0)
        return True
    else:
        return False


def enum_for_xrefs(load_function_address, json_data, enumeration):
    for x in XrefsTo(load_function_address, flags=0):
        current_address = x.frm
        addr_minus_20 = current_address-20

        push_count = 0
        while current_address >= addr_minus_20:
            current_address = PrevHead(current_address)
            if GetMnem(current_address) == "push":
                push_count += 1
                data = GetOperandValue(current_address, 0)
                if push_count == 2:
                    if data in json_data:
                        name = json_data[data]
                        AddConstEx(enumeration, str(name), int(data), -1)
                        if convert_offset_to_enum(current_address):
                            print "[+] Converted 0x%x to %s enumeration" % (current_address, name)
                            address_plus_30 = current_address+30
                            address = current_address
                            while address <= address_plus_30:
                                address = NextHead(address)
                                if GetMnem(address) == "mov":
                                    if 'dword' in GetOpnd(address, 0) and 'eax' in GetOpnd(address, 1):
                                        operand_value = GetOperandValue(address, 0)
                                        MakeName(operand_value, str("d_"+name))


fh = open("output.json", 'rb')
d = fh.read()
json_data = json.loads(d)
fh.close()

```

```
# JSON objects don't allow using integers as dict keys. Little workaround for
# this issue.
for k,v in json_data.iteritems():
    json_data[int(k)] = json_data.pop(k)

conversion_function = 0x00405680
enumeration = GetEnum("crc32_functions")
if enumeration == 0xFFFFFFFF:
    enumeration = AddEnum(0, "crc32_functions", idaapi.hexflag())
enum_for_xrefs(conversion_function, json_data, enumeration)
```

0x04 总结

在上一节中，我们利用 IDAPython 成功的解决了一个哈希混淆的问题，在这个问题中我们用到了枚举对象。枚举对象对我们分析这类问题会很有帮助，能够节省我们大量的时间。并且这个对象可以很容易的在 IDA 工程中提取或者加载，这对我们进行批量的逆向分析会很有帮助。

http://drops.wooyun.org/tips/11849

IDAPython 让你的生活更滋润 – Part 3 and Part 4

0x00 简介

今天在网上看到平底锅 blog 上 Josh Grunzweig 发表了一系列关于利用 IDAPython 分析 malware 的教程。感觉内容非常不错，于是翻译成中文与大家一起分享。

Part1 和 Part2 的译文地址：
http://drops.wooyun.org/papers/11849

Part3 和 Par4 原文地址：

http://researchcenter.paloaltonetworks.com/2016/01/using-idapython-to-make-your-life-easier-part-3/

http://researchcenter.paloaltonetworks.com/2016/01/using-idapython-to-make-your-life-easier-part-4/

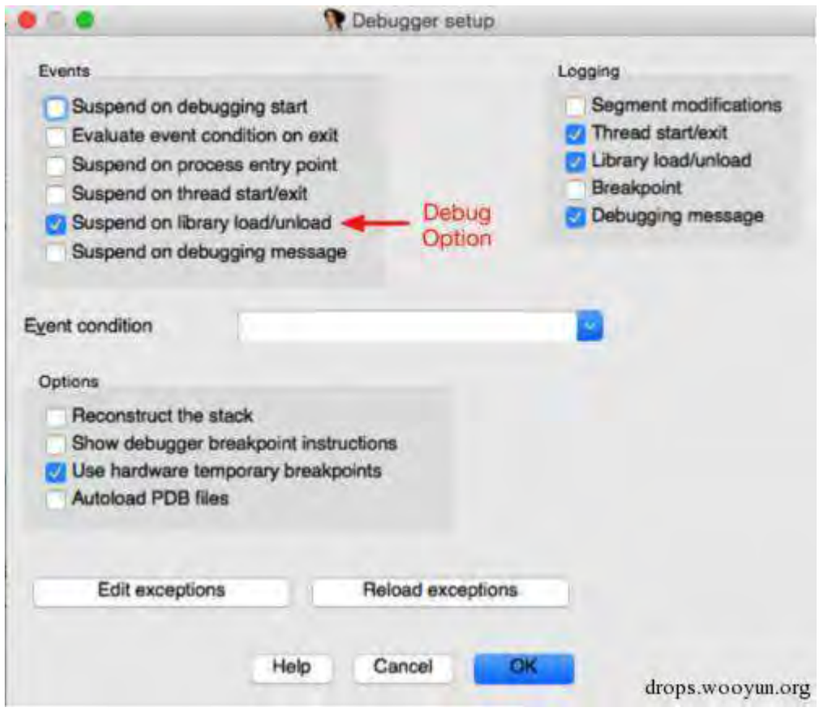
0x01 Part3- 序

上一篇译文中我们讨论了如何利用 Python 来方便我们进行逆向，这一篇中我们将会介绍如何利用 IDAPython 来解决条件断点问题。

当我们用 IDA Pro 进行调试的时候，我们总想断点在某个满足某些条件的特殊地址上。举两个例子，第一个是我们只想将断点下在某个函数被传递了指定的参数的情况下；第二 个是我们想把断点下在某个特定的库被加载到我的虚拟机的时候。今天我将会给大家介绍如何解决这些问题的方法。

0x02 Part3 – 背景介绍

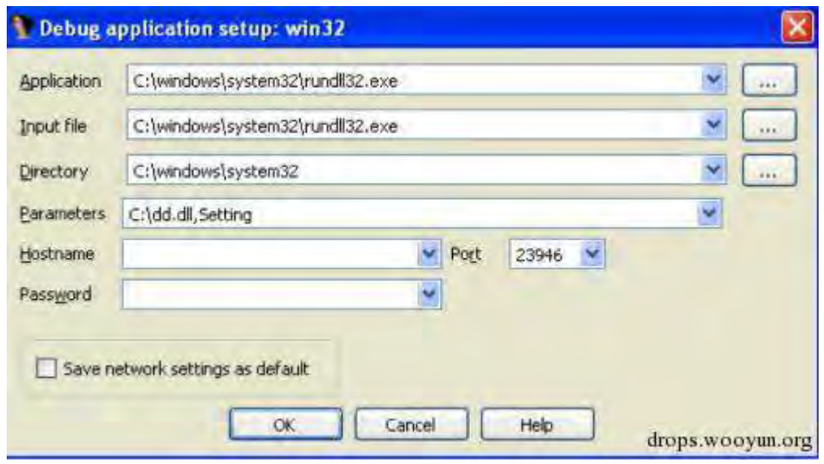
研究员经常会对 dll 文件进行分析。在很多情况下，这些 dll 会被其他的可执行文件加载。一种解决方案是用 IDA 在选项中设置，确保 IDA 在每个库被加载的时候都暂停下来。如图所示：



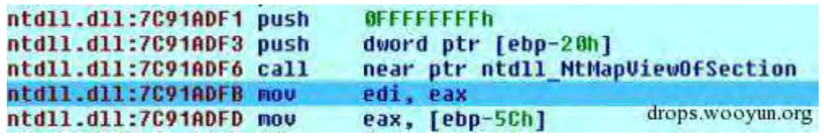
然而这种方式非常低效，经常需要研究人员手动的暂停和继续：先判断最新加载的库是不是想调试的那个，如果不是再继续执行。如果能仅仅运行一条简单的指令，然后坐等想要的文件被加载进来的话，会让我们的调试工作更加惬意。

0x03 Part3 – 条件断点

我们将用一个 dd.dll 的文件作为例子进行讲解。这个文件在运行过程中被提取后，通常会被利用程序注入到另一个进程中。为了能够在 IDA 中动态调 试这个文件，我将会动态加载 rundll32.exe 这个保存在 system32 目录下的执行文件，并且将 dd.dll 文件作为参数（如图 2 所示）。在给 定导出名的情况下，rundll32.exe 文件允许用户加载一个指定的 dll 文件。在这个样本中，我对 dll 文件中的 Setting 函数很有兴趣，于是 我将如下参数传递给 IDA：



下一步是当一个 DLL 文件被加载进来后，找到正确的地方设置断点。为了做到这一点，我先在调试器设置中的”在库被加载或卸载时暂停”的选项上打勾，然后我能够看到有一个断点被设置到了 NtMapViewOfSection 的后面一个指令。



接下来我在 0x7C91ADFB 处下了一个断点。在我的代码中，我使用 add_bpt()和 enable_bpt()去创建和激活这个断点。

```
1 '''
2 ntdll.dll:7C91ADF1 push       0FFFFFFFFh
3 ntdll.dll:7C91ADF3 push       dword ptr [ebp-20h]
4 ntdll.dll:7C91ADF6 call       near ptr ntdll_NtMapViewOfSection
5 ntdll.dll:7C91ADFB mov         edi, eax
6 '''
7
8 address = 0x7C91ADFB # Just after NtMapViewOfSection
9 add_bpt(address, 0, BPT_SOFT)
10enable_bpt(address, True)
```

这个时候，我们已经在 0x7C91ADFB 处下了断点，并且调试器会在每个 dll 被加载的时候暂停。为了确保只有当”dd.dll”文件被加载的时 候才暂停，我们必须创建一个条件断点。条件断点允许一个研究人员使用 IDC 或者 Python 代码去判断一个断点是不是真的被触发。如果返回值为真，那么断 点就会被触发，否则将会被忽略。为了使用 Python，我们先将 Python 设置为默认语言。然后，将我

们的 Python 代码保存在一个变量中，然后在 SetBptCnd()中调用这个变量，这样就能把我们的代码变成断点的条件判断了。当条件被设置好后，我们让调试器继续运行，直到遇到一个暂停事件。伪代码如下：

```
address = 0x7C91ADFB # Just after NtMapViewOfSection
RunPlugin("python", 3) # Python default programming
StartDebugger("","","");

dll = "dd.dll"
condition = """
for m in Modules():
    if "%s".lower() in m.name.lower():
        print "Breaking on", m.name.lower()
        del_bpt(%d)
        return True
return False
""" % (dll, address)

add_bpt(address, 0, BPT_SOFT)
enable_bpt(address, True)
SetBptCnd(address, condition)

continue_process()
GetDebuggerEvent(WFNE_SUSP, -1)
```

实际上用来使用的条件断点的代码如下：

```
for m in Modules():
    if "dd.dll".lower() in m.name.lower():
        print "Breaking on", m.name.lower()
        del_bpt(0x7C91ADFB)
        return True
return False
```

这个代码会遍历所有被加载进来的模块，然后判断”dd.dll”是不是被加载了进来，如果被加载了，一条调试信息将会被打印出来，并且会返回 Ture 并触发断点。当执行的时候，我们能看到如下输出：

在这时候，我们可以设置我们期望的导出函数为”Setting”然后运行程序直到触发断点。为了能够自动的完成这个任务，如下代码将会被用到：

```
def get_names(base, size, desired_name):
    current_address = base
    while current_address &lt;= base+size:
            current_address = NextHead(current_address)
            print hex(current_address)
            if desired_name in Name(current_address):
                return current_address

for m in Modules():
    if 'dd.dll' in m.name.lower():
        base = m.base
        size = m.size
        analyze_area(base, base+size)
        setting = get_names(base, size, "Setting")
        if setting:
            add_bpt(setting, 0, BPT_SOFT)
            enable_bpt(setting, True)
            continue_process()
GetDebuggerEvent(WFNE_SUSP, -1)
```

这段代码会遍历所有加载的模块来寻找”dd.dll”文件。当找到这个文件之后，我们会分析这个 DLL 文件并且遍历代码中的函数名。当我们找 到”Setting”这个函数名后会在这个函数名对应的函数上设置一个断点。随后继续我们继续运行调试器，直到这个断点被触发。当我们执行的时候，发现调 试器暂停在了我们期望的地址上：

```
dd.dll:10003500
dd.dll:10003500 dd_Setting:
dd.dll:10003500 push    ebp
dd.dll:10003501 mov     ebp, esp
dd.dll:10003503 sub     esp, 514h
dd.dll:10003509 xor     eax, eax
dd.dll:1000350B push    206h
dd.dll:10003510 push    eax
dd.dll:10003511 mov     [ebp-30Ch], ax
dd.dll:10003518 lea     eax, [ebp-30Ah]
dd.dll:1000351E push    eax
dd.dll:1000351F call    sub_10002308
dd.dll:10003524 lea     ecx, [ebp-30Ch]
dd.dll:1000352A call    sub_10001E00
dd.dll:1000352F push    offset aRasphone_dll            ; drops.wooyun.org
```

0x04 Part3 – 总结

尽管设置条件断点看起来是个很小的技术，但的确能节省分析人员大量的时间。一小段代码就能解决我们一遍一遍手动的工作，何乐而不为呢。

0x05 Part4 – 序

分析人员经常会遇到复杂度很高的代码，并且在动态运行过程中这些代码并不是显而易见的。使用 IDAPython，我们不光可以确定哪些指令被执行了，还可以确定这些指令被执行了多少次。

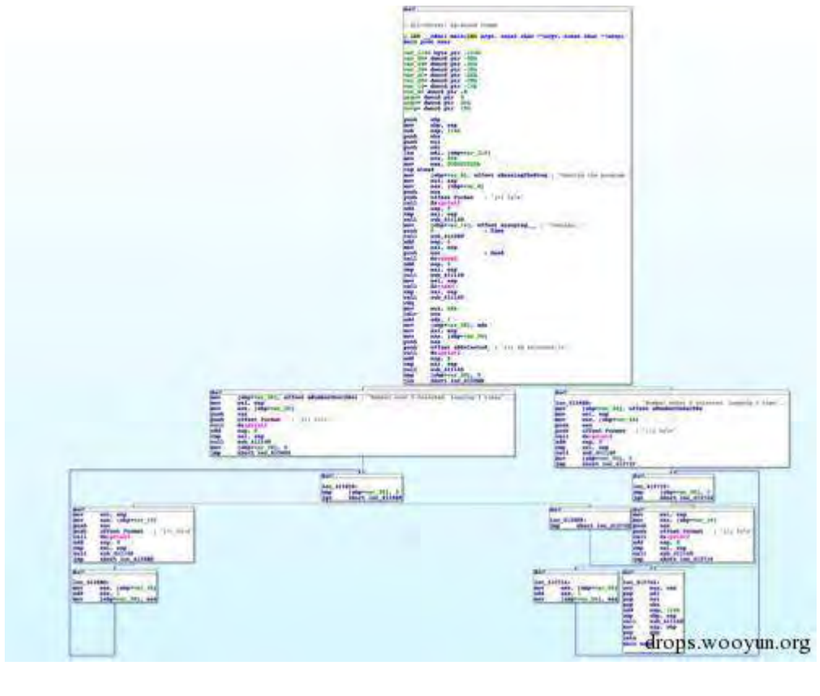
0x06 Part4 – 背景

为了这篇文章，我写了一个简单的 c 程序。代码如下：

```
#include "stdafx.h"
#include <stdlib.h>
#include <time.h>

int _tmain(int argc, _TCHAR* argv[])
{
    char* start = "Running the program.";
    printf("[+] %s\n", start);
    char* loop_string = "Looping...";

    srand (time(NULL));
    int bool_value = rand() % 10 + 1;
    printf("[+] %d selected.\n", bool_value);
    if(bool_value > 5)
    {
        char* over_five = "Number over 5 selected. Looping 2 times.";
        printf("[+] %s\n", over_five);
        for(int x = 0; x < 2; x++)
            printf("[+] %s\n", loop_string);
    }
    else
    {
        char* under_five = "Number under 5 selected. Looping 5 times.";
        printf("[+] %s\n", under_five);
        for(int x = 0; x < 5; x++)
            printf("[+] %s\n", loop_string);
    }
    return 0;
```

29}

当我们把这个文件载入 IDA 后，我们可以看到期望的循环和重定向的语句。如果我们不看底层代码（源代码）的话，通过反汇编，我们依然可以知道会发生什么。如图所示：



但是如果我们想要知道哪一块代码会在运行时执行。我就需要 IDAPython 来完成这个挑战了。

0x07 Part4 – IDAPython 脚本

我们第一个解决的挑战是能够单步处理每一条指令。我们用下面的代码完成这个任务（每一条执行的指令都会通过调试接口输出）：

```
1 RunTo(BeginEA())
2 event =GetDebuggerEvent(WFNE_SUSP, -1)
3
4 EnableTracing(TRACE_STEP, 1)
5 event =GetDebuggerEvent(WFNE_ANY|WFNE_CONT, -1)
6
7 while True:
8     event =GetDebuggerEvent(WFNE_ANY, -1)
9     addr =GetEventEa()
```

```
10    print "Debug: current address", hex(addr), "| debug event", hex(event)
11    if event <= 1: break
```

在上面的代码中，我们先启动调试器，然后利用 Runto(BeginEA())执行到入口处的代码。随后的 GetDebuggerEvent()函 数调用将会在断点触发的时候进行等待。然后我们使用 EnableTracing()函数启动跟踪。随后的 GetDebuggerEvent()将会让调试 器继续单步执行。最后我们进入一个循环然后遍历每个地址，直到我们收到进程结束的信号为止。输出结果如下图所示：

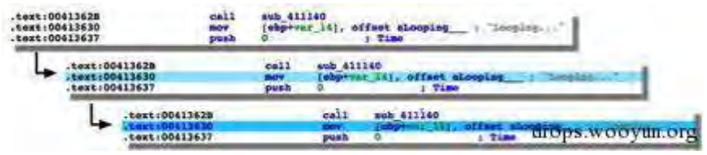
```
Debug: current address 0x41371cL | debug event 0x20
Debug: current address 0x41371fL | debug event 0x20
Debug: current address 0x413723L | debug event 0x20
Debug: current address 0x413742L | debug event 0x20
Debug: current address 0x413744L | debug event 0x20
Debug: current address 0x413745L | debug event 0x20
Debug: current address 0x413746L | debug event 0x20
Debug: current address 0x413747L | debug event 0x20
Debug: current address 0x41374dL | debug event 0x20
Debug: current address 0x41374fL | debug event 0x20
Debug: current address 0x411140L | debug event 0x20
Debug: current address 0x411570L | debug event 0x20
Debug: current address 0x411572L | debug event 0x20
Debug: current address 0x413754L | debug event 0x20
Debug: current address 0x413756L | debug event 0x20
Debug: current address 0x413757L | debug event 0x20
Debug: current address 0x411acfL | debug event 0x20
Debug: current address 0x411ad2L | debug event 0x20
Debug: current address 0x411ad7L | debug event 0x20
Debug: current address 0x411adeL | debug event 0x20
Debug: current address 0x411ae0L | debug event 0x20
Debug: current address 0x411ae5L | debug event 0x20
Debug: current address 0x411ae6L | debug event 0x20
Debug: current address 0x102480a0L | debug event 0x20
Debugger: process has exited (exit code 0)
Debug: current address 0x102480a0L | debug event 0x2
Debug: current address 0x102480a0L | debug event -0x2
```

Python drops.wooyun.org

下一步就是取出和设置每一条执行的指令的地址的颜色。为了做到这一点，我们可以使用 GetColor()和 SetColor()函数。接下来的代码 可以得到目标行指令的颜色，然后判断并设置这行指令的颜色。在这个例子中，我使用了四种不同深度的蓝色。某行指令被执行的越多颜色就会越深（我鼓励读者自 定义自己的偏好）。

```
1 def get_new_color(current_color):
2     colors = [0xffe699, 0xffcc33, 0xe6ac00, 0xb38600]
3     if current_color == 0xFFFFFF:
4         return colors[0]
5     if current_color in colors:
6         pos = colors.index(current_color)
7         if pos == len(colors)-1:
8             return colors[pos]
9         else:
10            return colors[pos+1]
11    return 0xFFFFFF
12
13
14current_color = GetColor(addr, CIC_ITEM)
15new_color = get_new_color(current_color)
16SetColor(addr, CIC_ITEM, new_color)
```

执行上面的代码会让指定行的指令变蓝，如果指定行的指令被执行了多次，会让蓝色变得更深。



我们可以使用接下的代码去除掉之前在 IDA Pro 文件中的颜色。因为设置颜色为 0xFFFFFF 会让颜色变成白色，并且去掉之前设置的所有颜色。

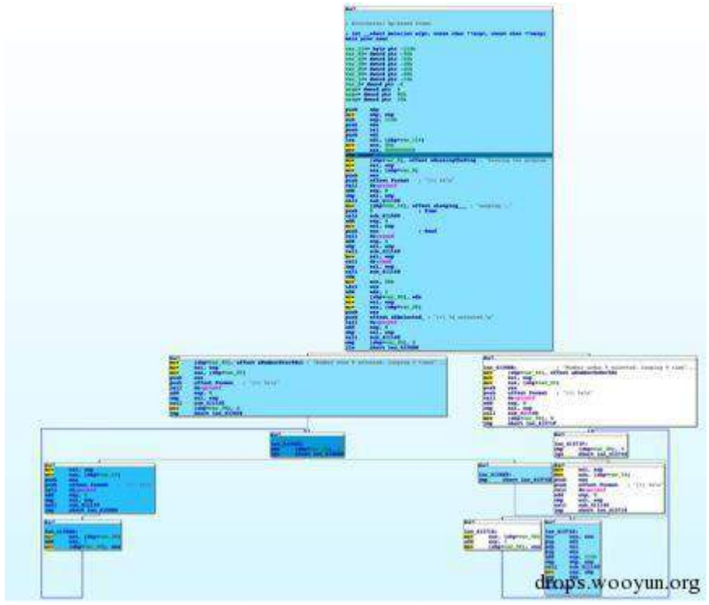
```
heads = Heads(SegStart(ScreenEA()), SegEnd(ScreenEA()))
for i in heads:
    SetColor(i, CIC_ITEM, 0xFFFFFF)
```

将所有的代码放在一起我们可以得到如下结果：

```
heads = Heads(SegStart(ScreenEA()), SegEnd(ScreenEA()))
for i in heads:
        SetColor(i, CIC_ITEM, 0xFFFFFF)

def get_new_color(current_color):
        colors = [0xffe699, 0xffcc33, 0xe6ac00, 0xb38600]
        if current_color == 0xFFFFFF:
                return colors[0]
        if current_color in colors:
                pos = colors.index(current_color)
                if pos == len(colors)-1:
                        return colors[pos]
                else:
                        return colors[pos+1]
        return 0xFFFFFF

RunTo(BeginEA())
event = GetDebuggerEvent(WFNE_SUSP, -1)

EnableTracing(TRACE_STEP, 1)
event = GetDebuggerEvent(WFNE_ANY|WFNE_CONT, -1)
while True:
        event = GetDebuggerEvent(WFNE_ANY, -1)
        addr = GetEventEa()
        current_color = GetColor(addr, CIC_ITEM)
        new_color = get_new_color(current_color)
```

```
27          SetColor(addr, CIC_ITEM, new_color)
28          if event <= 1: break
```

当我们在我们的程序中执行这段代码的时候，我们能看到如下的变化：我们能看到所有被执行的汇编指令都被高亮了。就像下图所示，指令执行的次数越多，颜色就会越深，这会让我们很容易的理解程序执行的流程。



0x08 Part4 – 总结

这篇 blog 介绍了如何利用 IDAPython 来给程序上色，这个技术能够很好的帮助研究人员分析复杂的代码并节省大量的时间。

http://drops.wooyun.org/tips/12060

IDAPython：让你的生活更美好（一）

IDAPython 是 IDA 的一个功能强大的扩展特性，对外提供了大量的 IDA API 调用。另外，还能在使用 python 脚本语言的过程中获得能力提升，所以我强烈推荐所有的逆向工程师使用它。

然而不幸的是，除了下面这几项，关于 IDAPython 的信息和教程实在太少了。

- “The IDA Pro Book” by Chris Eagle
- “The Beginner’s Guide to IDAPython” by Alex Hanel

- “IDAPython Wiki” by Magic Lantern

为了增加 IDAPython 相关的教程资料，在该系列中我将会提供我写的一些有趣的实例代码。 而在第一部分，我会通过编写脚本来解码一个恶意软件里面的大量被混淆字的符串。

背景

在逆向恶意样本的过程中，我遇到了下面这个函数：

```
.text:00405C7D ; ---------------------------------------------------------------------------
.text:00405C7D
.text:00405C7D loc_405C7D:                             ; CODE XREF: sub_405BF0+6B↑j
.text:00405C7D                 mov     eax, 1
.text:00405C82                 cmp     edi, eax
.text:00405C84                 jle     short loc_405CA3
.text:00405C86                 mov     ecx, dword_41C050
.text:00405C8C                 shl     ecx, 9
.text:00405C8F                 lea     ecx, unk_41E477[ecx]
.text:00405C95
.text:00405C95 loc_405C95:                             ; CODE XREF: sub_405BF0+B1↓j
.text:00405C95                 mov     dl, [eax+esi]
.text:00405C98                 xor     dl, [esi]
.text:00405C9A                 inc     eax
.text:00405C9B                 cmp     eax, edi
.text:00405C9D                 mov     [ecx+eax-1], dl
.text:00405CA1                 jl      short loc_405C95
.text:00405CA3
.text:00405CA3 loc_405CA3:                             ; CODE XREF: sub_405BF0+8B↑j
.text:00405CA3                                         ; sub_405BF0+94↑j
.text:00405CA3                 push    offset CriticalSection ; lpCriticalSection
.text:00405CA8                 call    ds:LeaveCriticalSection
.text:00405CAE                 mov     eax, dword_41C050
.text:00405CB3                 shl     eax, 9
.text:00405CB6                 add     eax, offset unk_41E478
.text:00405CBB                 pop     edi
.text:00405CBC                 retn
.text:00405CBC sub_405BF0      endp
.text:00405CBC
.text:00405CBC ; ---------------------------------------------------------------------------
```

Figure 1 String decryption function

基于过去的经验，我觉得这个函数应该是用来解密二进制数据的。这个函数的交叉引用次数证明了我的猜测：

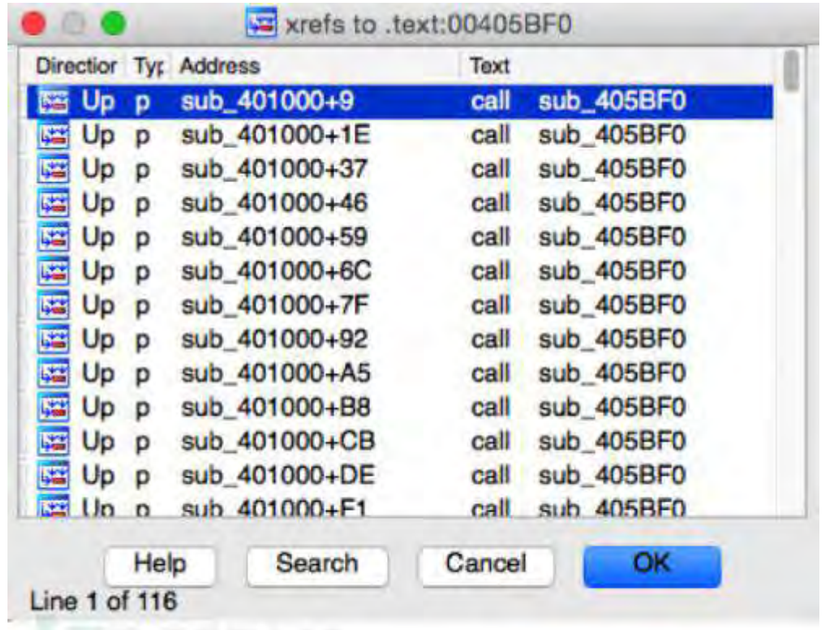


Figure 2 High number of references to suspect function

正如图 2 所示，这个特殊的函数被调用了 116 次。这个函数的每一次调用，都有一个二进制数据对象通过 ESI 寄存器作为参数传入。

```
.text:00401004                  mov     esi, offset unk_418BE0
.text:00401009                  call    sub_405BF0
.text:0040100E                  mov     ebp, ds:LoadLibraryA
.text:00401014                  push    eax             ; lpLibFileName
.text:00401015                  call    ebp ; LoadLibraryA
.text:00401017                  mov     esi, offset unk_418BF0
.text:0040101C                  mov     edi, eax
.text:0040101E                  call    sub_405BF0
.text:00401023                  push    eax             ; lpProcName
.text:00401024                  push    edi             ; hModule
.text:00401025                  mov     edi, ds:GetProcAddress
.text:0040102B                  call    edi ; GetProcAddress
.text:0040102D                  mov     esi, offset unk_418C08
.text:00401032                  mov     dword_41D020, eax
.text:00401037                  call    sub_405BF0
.text:0040103C                  push    eax             ; lpLibFileName
.text:0040103D                  call    ebp ; LoadLibraryA
.text:0040103F                  mov     esi, offset unk_418C18
.text:00401044                  mov     ebx, eax
.text:00401046                  call    sub_405BF0
.text:0040104B                  push    eax             ; lpProcName
.text:0040104C                  push    ebx             ; hModule
.text:0040104D                  call    edi ; GetProcAddress
.text:0040104F                  mov     esi, offset unk_418C2C
.text:00401054                  mov     dword_41D01C, eax
.text:00401059                  call    sub_405BF0
.text:0040105E                  push    eax             ; lpProcName
.text:0040105F                  push    ebx             ; hModule
.text:00401060                  call    edi ; GetProcAddress
.text:00401062                  mov     esi, offset unk_418C44
.text:00401067                  mov     dword_41D018, eax
.text:0040106C                  call    sub_405BF0
```

Figure 3 Instances where the suspect function (405BF0) is called

通过这一分析，我更加确定这个函数是恶意软件运行时用来解密字符串的。面对这一情况，我可以选择下面这几种解决方案：

1. 手动解密并重命名这些被混淆的字符串
2. 运行这些样本，遇到这些字符串的时候进行重命名
3. 编写一个脚本来解密并重命名这些字符串

如果恶意软件只是解密少量的字符串，我会选择第一种或者第二种方案。然而，正如我们前面了解到的，这个函数被调用了 116 次，所以编写一个脚本似乎更有意义。

编写 IDAPYTHON 脚本

解决混淆字符串问题的第一个步骤是找到并重写解密函数。幸运的是，这里的解密函数比较简单。这个函数简单的将二进制数组中的第一个字符与剩下的数据逐字节的进行异或。

E4 91 96 88 89 8B 8A CA 80 88 88

在上面这个例子中，取出 0XE4 跟剩下的其他数据进行异或。解密的结果是’urlmon.dll’。我们可以用 python 这样实现：

```
def decrypt(data):
    length = len(data)
    c = 1
    o = ""
    while c < length:
```

```
        o  +=  chr(ord(data[0])  ^  ord(data[c]))
        c  +=  1
    return  o
```

运行这段代码，我们得到了我们预期的结果

```
>>>  from  binascii  import  *
>>>  d  =  unhexlify("E4  91  96  88  89  8B  8A  CA  80  88  88".replace("  ",''))
>>>  decrypt(d)
'urlmon.dll'
```

接下来就是找出代码中引用了解密函数的地方，提取作为参数出入的数据。 通过 IDA 找到函数的引用比较简单，IDA 提供的 API 函数 XrefsTo()完美的解决了这个问题。下面这个脚本中，我将地址硬编码到解密脚本中。 下面的代码能够找出解密函数的引用地址。下面这个测试，我简单的将地址用 16 进制的格式打印出来。

```
for  addr  in  XrefsTo(0x00405BF0,  flags=0):
    print  hex(addr.frm)

Result:
0x401009L
0x40101eL
0x401037L
0x401046L
0x401059L
0x40106cL
0x40107fL
<truncated>
```

从交叉引用处识别参数并提取原始数据稍微复杂一点，但显然不是不可能的。首先我们要得到字符串解密函数调用点之前最近的一个 ‘mov esi, offset unk_??’指令的偏移地址。为了达到这个目的，我们会对字符串解密函数的每一处调用，逐条指令的回溯去查找 ‘mov esi, offset [addr]’ 指令。我们可以使用 GetOperandValue()函数（API）来获取真正的偏移地址。

下面是代码实现：

```
def  find_function_arg(addr):
    while  True:
        addr  =  idc.PrevHead(addr)
        if  GetMnem(addr)  ==  "mov"  and  "esi"  in  GetOpnd(addr,  0):
            print  “We  found  it  at  0x%x”  %  GetOperandValue(addr,  1)
            break

Example  Results:
Python>find_function_arg(0x00401009)
We  found  it  at  0x418be0
```

现在我们只要简单的将偏移地址处的字符串提取出来。 通常我们会使用 GetString()函数，然而，由于混淆过的字串是原始的二进制数据，这个函数无法得到我们期望的结果。 所以我们通过逐字节的重复读取，直到遇到 null（0×00）结束符为止。

下面是代码实现：

```
def  get_string(addr):
    out  =  ""
    while  True:
        if  Byte(addr)  !=  0:
            out  +=  chr(Byte(addr))
        else:
            break
        addr  +=  1
    return  out
```

接下来就是将之前实现的功能整合到一起：

```
def  find_function_arg(addr):
    while  True:
        addr  =  idc.PrevHead(addr)
        if  GetMnem(addr)  ==  "mov"  and  "esi"  in  GetOpnd(addr,  0):
            return  GetOperandValue(addr,  1)
    return  ""

def  get_string(addr):
    out  =  ""
    while  True:
        if  Byte(addr)  !=  0:
            out  +=  chr(Byte(addr))
        else:
            break
        addr  +=  1
    return  out

def  decrypt(data):
    length  =  len(data)
    c  =  1
    o  =  ""
    while  c  <  length:
        o  +=  chr(ord(data[0])  ^  ord(data[c]))
        c  +=  1
    return  o
```

```
print "[*] Attempting to decrypt strings in malware"
for x in XrefsTo(0x00405BF0, flags=0):
    ref = find_function_arg(x.frm)
    string = get_string(ref)
    dec = decrypt(string)
    print "Ref Addr: 0x%x | Decrypted: %s" % (x.frm, dec)

Results:
[*] Attempting to decrypt strings in malware
Ref Addr: 0x401009 | Decrypted: urlmon.dll
Ref Addr: 0x40101e | Decrypted: URLDownloadToFileA
Ref Addr: 0x401037 | Decrypted: wininet.dll
Ref Addr: 0x401046 | Decrypted: InternetOpenA
Ref Addr: 0x401059 | Decrypted: InternetOpenUrlA
Ref Addr: 0x40106c | Decrypted: InternetReadFile
<truncated>
```

我们可以不用运行恶意软件也能看到所有加密后的字符串。下一个步骤我们可以将解密后的字符串以注释的形式写到引用处，让字符串明文与密文同时存在，这样就很便于分析了。我们使用 MakeComm() 函数来实现这一功能。将下面这两行代码加入到上面代码 print 语句的后面：

```
MakeComm(x.frm, dec)
MakeComm(ref, dec)
```

如下图所示，我们能看到的，通过这一额外的步骤，我们可以结合交叉引用看得更清楚。现在我们能够很容易的找到被引用的特殊字符串。

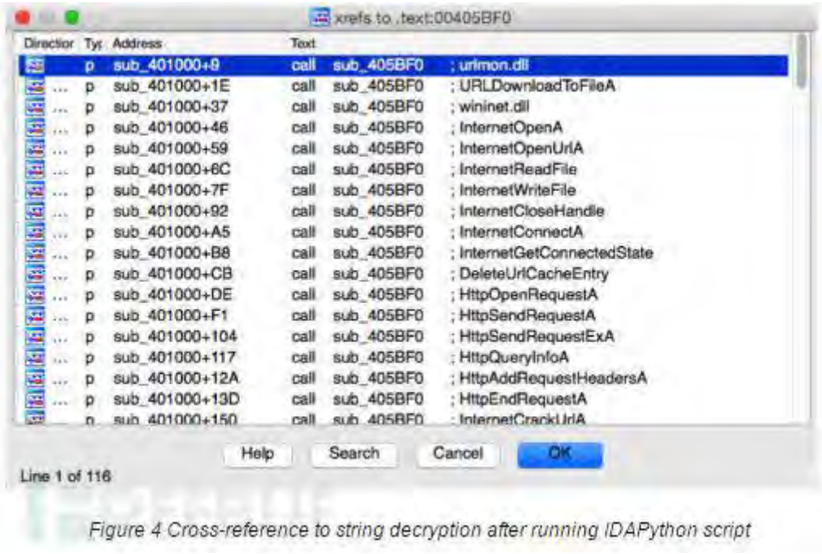


Figure 4 Cross-reference to string decryption after running IDAPython script

另外的，通过反编译，我们可以看到解密后的字符串作为注释存在与代码中。

```
.text:00401004                  mov     esi, offset unk_418BE0
.text:00401009                  call    sub_405BF0      ; urlmon.dll
.text:0040100E                  mov     ebp, ds:LoadLibraryA
.text:00401014                  push    eax             ; lpLibFileName
.text:00401015                  call    ebp ; LoadLibraryA
.text:00401017                  mov     esi, offset unk_418BF0
.text:0040101C                  mov     edi, eax
.text:0040101E                  call    sub_405BF0      ; URLDownloadToFileA
.text:00401023                  push    eax             ; lpProcName
.text:00401024                  push    edi             ; hModule
.text:00401025                  mov     edi, ds:GetProcAddress
.text:0040102B                  call    edi ; GetProcAddress
.text:0040102D                  mov     esi, offset unk_418C08
.text:00401032                  mov     dword_41D020, eax
.text:00401037                  call    sub_405BF0      ; wininet.dll
.text:0040103C                  push    eax             ; lpLibFileName
.text:0040103D                  call    ebp ; LoadLibraryA
.text:0040103F                  mov     esi, offset unk_418C18
.text:00401044                  mov     ebx, eax
.text:00401046                  call    sub_405BF0      ; InternetOpenA
.text:0040104B                  push    eax             ; lpProcName
.text:0040104C                  push    ebx             ; hModule
.text:0040104D                  call    edi ; GetProcAddress
.text:0040104F                  mov     esi, offset unk_418C2C
.text:00401054                  mov     dword_41D01C, eax
.text:00401059                  call    sub_405BF0      ; InternetOpenUrlA
.text:0040105E                  push    eax             ; lpProcName
.text:0040105F                  push    ebx             ; hModule
.text:00401060                  call    edi ; GetProcAddress
.text:00401062                  mov     esi, offset unk_418C44
.text:00401067                  mov     dword_41D018, eax
.text:0040106C                  call    sub_405BF0      ; InternetReadFile
```

Figure 5 Assembly after IDAPython script is run

总结

通过使用 IDAPython，我们完成一个困难的任务—恶意二进制样本中的 161 个加密字符串的解密。正如我们看到的，IDAPython 对于逆向工程来说是一款强大的工具，简化任务，节省宝贵的时间。

***原文链接：researchcenter.paloaltonetworks，东二门陈冠希/编译，转载请注明来自 FreeBuf 黑客与极客（FreeBuf.COM）**

http://www.freebuf.com/sectool/92107.html

IDAPython：让你的生活更美好（二）

继续我们的主题—使用 IDAPython　让逆向工程师的生活变得更美好。

这一部分，我们将着手处理一个非常常见的问题：shellcode 和恶意软件使用 hash 算法混淆加载的函数和链接库，这项技术被广泛使用。使用 IDAPython，我们能够很容易的解决这个很有挑战性的问题。

背景

逆向人员经常在 shellcode 里遇到名字被混淆的函数。 总体来说这个过程比较简单：代码运行之后会在初始化阶段加载 kernel32.dll 链接库，然后继续用这个加载的映像来标识和存储 LoadLibraryA 的函数，而这个函数又会去加载附加的链接库和函数 。

这项特殊的技术使用 hash 算法来标识一个函数，该 hash 算法一般使用 CRC32, 其他变种中 ROR13 也经常被使用。

在逆向恶意软件的过程中，我们找到下面这段代码：

```
.text:004125A0    push    1
.text:004125A2    push    7695D1CCh
.text:004125A7    push    edx
.text:004125A8    call    load_function
.text:004125AD    add     esp, 0Ch
.text:004125B0    mov     dword_41A59C, eax
.text:004125B5    cmp     eax, ebx
.text:004125B7    jz      loc_4124E6
.text:004125BD    push    1
.text:004125BF    push    0E62E824Dh
.text:004125C4    push    esi
.text:004125C5    call    load_function
.text:004125CA    add     esp, 0Ch
.text:004125CD    mov     dword_41A3D8, eax
.text:004125D2    cmp     eax, ebx
.text:004125D4    jz      loc_4124E6
.text:004125DA    push    1
.text:004125DC    push    9A80E589h
.text:004125E1    push    esi
.text:004125E2    call    load_function
.text:004125E7    add     esp, 0Ch
.text:004125EA    mov     dword_41A56C, eax
.text:004125EF    cmp     eax, ebx
.text:004125F1    jz      loc_4124E6
.text:004125F7    push    1
.text:004125F9    push    0F3B07FCCh
.text:004125FE    push    edi
.text:004125FF    call    load_function
.text:00412604    add     esp, 0Ch
.text:00412607    mov     dword_41A380, eax
.text:0041260C    cmp     eax, ebx
.text:0041260E    jz      loc_4124E6
.text:00412614    mov     edi, [ebp+var_18]
.text:00412617    push    1
.text:00412619    push    301BF0h
.text:0041261E    push    edi
.text:0041261F    call    load_function
.text:00412624    add     esp, 0Ch
.text:00412627    mov     dword_41A544, eax
.text:0041262C    cmp     eax, ebx
.text:0041262E    jz      loc_4124E6
.text:00412634    mov     eax, [ebp+var_4]
.text:00412637    push    1
.text:00412639    push    0A9290135h
.text:0041263E    push    eax
.text:0041263F    call    load_function
.text:00412644    add     esp, 0Ch
.text:00412647    mov     dword_41A38C, eax
```

Figure 1 Malware loading functions dynamically using CRC32 hash

在上面的例子中， 我们可以通过找到包含 0xEDB88320 来快速标识出使用 CRC32 算法的地方：

```
.text:00405590 crc32           proc near               ; CODE XREF: sub_405640+C p
.text:00405590                 cmp     byte_41AA9E, 0
.text:00405597                 jnz     locret_405631
.text:0040559D                 mov     byte_41AA9E, 1
.text:004055A4                 xor     ecx, ecx
.text:004055A6
.text:004055A6 loc_4055A6:                             ; CODE XREF: crc32+9B j
.text:004055A6                 mov     eax, ecx
.text:004055A8                 shr     eax, 1
.text:004055AA                 test    cl, 1
.text:004055AD                 jz      short loc_4055B4
.text:004055AF                 xor     eax, 0EDB88320h
.text:004055B4
.text:004055B4 loc_4055B4:                             ; CODE XREF: crc32+1D'j
.text:004055B4                 test    al, 1
.text:004055B6                 jz      short loc_4055C1
.text:004055B8                 shr     eax, 1
.text:004055BA                 xor     eax, 0EDB88320h
.text:004055BF                 jmp     short loc_4055C3
.text:004055C1 ; ---------------------------------------------------------------------------
.text:004055C1
.text:004055C1 loc_4055C1:                             ; CODE XREF: crc32+26'j
.text:004055C1                 shr     eax, 1
.text:004055C3
.text:004055C3 loc_4055C3:                             ; CODE XREF: crc32+2F'j
.text:004055C3                 test    al, 1
.text:004055C5                 jz      short loc_4055D0
.text:004055C7                 shr     eax, 1
.text:004055C9                 xor     eax, 0EDB88320h
.text:004055CE                 jmp     short loc_4055D2
.text:004055D0 ; ---------------------------------------------------------------------------
.text:004055D0
.text:004055D0 loc_4055D0:                             ; CODE XREF: crc32+35'j
.text:004055D0                 shr     eax, 1
.text:004055D2
.text:004055D2 loc_4055D2:                             ; CODE XREF: crc32+3E'j
.text:004055D2                 test    al, 1
.text:004055D4                 jz      short loc_4055DF
.text:004055D6                 shr     eax, 1
.text:004055D8                 xor     eax, 0EDB88320h
.text:004055DD                 jmp     short loc_4055E1
.text:004055DF ; ---------------------------------------------------------------------------
```

Figure 2 CRC32 algorithm identified

现在算法和函数已经被标识出来了，我们可以使用交叉引用来看看这个函数被调用了多少次。

在这个特殊的样本中，这个函数被调用了 190 次。而我们不想手动的去解码这些 hash 值，那么像上一部分一样，我们可以使用 IDAPython 来简化我们的工作。

编写 IDAPython 脚本

第一步其实没有使用到 IDAPython， 但是用到了 Python。 为了标识出 hash 值与函数的匹配关系，我们需要生成一张微软的 windows 操作系统最常使用函数的 hash 值表。为了达成这个目的，我们可以抓取 windows 操作系统常用的链接库列表，然后遍历里面的函数。

```
def  get_functions(dll_path):
    pe  =  pefile.PE(dll_path)
    if  ((not  hasattr(pe,  'DIRECTORY_ENTRY_EXPORT'))  or  (pe.DIRECTORY_ENTRY_EXPORT  is  None)):
        print  "[*]  No  exports  for  %s"  %  dll_path
        return  []
    else:
        expname  =  []
        for  exp  in  pe.DIRECTORY_ENTRY_EXPORT.symbols:
            if  exp.name:
```

```
                expname.append(exp.name)
        return  expname
```

我们可以用下面代码来计算这些函数的 CRC32 的值。

```
def  calc_crc32(string):
    return  int(binascii.crc32(string)  &  0xFFFFFFFF)
```

最后，我们将结果写道 JSON 格式的文件中，命名为 output.json。这些 JSON 数据包含了一个很大的字典，格式如下：

```
HASH  =>  NAME
```

完成的脚本代码下载链接

https://github.com/pan-unit42/public_tools/blob/master/ida_scripts/gen_function_json.py

生成了这个文件之后，我们回到 IDA 中，剩下的脚本将会在这里完成。 首先我们的脚本会从之前生成的 output.json 文件中读取 JSON 数据。 不幸的是，JSON 对象不支持整数型的键值，所以数据加载之后，我们将键值转换为整数类型来替代字符串类型。

```
for  k,v  in  json_data.iteritems():
    json_data[int(k)]  =  json_data.pop(k)
```

数据完全载入后，我们创建一个新的枚举类型（enumeration）来存储‘hash 值-函数名’的映射。（更多关于的枚举类型的资料与工作原理，可以阅读下面的这个教程

http://www.cprogramming.com/tutorial/enum.html）

使用枚举类型，我们能够将一个整数值（如 CRC32）与字符串（如函数名）建立映射关系。为了在 IDA 中建立新的枚举，我们将使用到 AddEnum()函数。为了让脚本能处理更多的情况，我们先使用 GetEnum()函数来校验枚举是否已经存在。

```
enumeration  =  GetEnum("crc32_functions")
if  enumeration  ==  0xFFFFFFFF:
    enumeration  =  AddEnum(0,  "crc32_functions",  idaapi.hexflag())
```

这个枚举以后会被修改。 下一步就是找到 hash 转换函数的交叉引用。这个跟第一部分提到的很相似。当分析参数如何传递到函数中时， 我们可以看到 CRC32 格式的 hash 值被作为第二个参数。

```
.text:004124F3                  push    1
.text:004124F5                  push    4C8A5B22h
.text:004124FA                  push    edx
.text:004124FB                  call    load_function
```

我们将遍历函数调用前面的指令，找到第二个 push 指令的使用处。一旦找到，我们将会将 CRC32 的 hash 值与之前从 output.json 里面读取出来的 JSON 数据进行匹配，看是否有匹配的函数名。

```
for  x  in  XrefsTo(load_function_address,  flags=0):
        current_address  =  x.frm
```

```
        addr_minus_20  =  current_address-20
        push_count  =  0
        while  current_address  >=  addr_minus_20:
            current_address  =  PrevHead(current_address)
            if  GetMnem(current_address)  ==  "push":
                push_count  +=  1
                data  =  GetOperandValue(current_address,  0)
                if  push_count  ==  2:
                    if  data  in  json_data:
                        name  =  json_data[data]
```

在下面这个步骤中，我们使用 AddConstEx()函数将 CRC32 hash 值与函数名加入到之前创建的枚举中。

```
AddConstEx(enumeration,  str(name),  int(data),  -1)
```

一旦数据被加入到枚举中， 我们就能够将 CRC32 hash 值转换为对应的枚举名。 下面两个函数能够用来获取枚举的第一个实例，以及获取转换为枚举的数据的地址。

```
def  get_enum(constant):
    all_enums  =  GetEnumQty()
    for  i  in  range(0,  all_enums):
        enum_id  =  GetnEnum(i)
        enum_constant  =  GetFirstConst(enum_id,  -1)
        name  =  GetConstName(GetConstEx(enum_id,  enum_constant,  0,  -1))
        if  int(enum_constant)  ==  constant:  return  [name,  enum_id]
        while  True:
            enum_constant  =  GetNextConst(enum_id,  enum_constant,  -1)
            name  =  GetConstName(GetConstEx(enum_id,  enum_constant,  0,  -1))
            if  enum_constant  ==  0xFFFFFFFF:
                break
            if  int(enum_constant)  ==  constant:  return  [name,  enum_id]
    return  None

def  convert_offset_to_enum(addr):
    constant  =  GetOperandValue(addr,  0)
    enum_data  =  get_enum(constant)
    if  enum_data:
        name,  enum_id  =  enum_data
        OpEnumEx(addr,  0,  enum_id,  0)
        return  True
    else:
        return  False
```

这个枚举转换发生之后，我们将会关注到函数后面 DWORD 加载的地址。

```
.text:004124F3        push    1
.text:004124F5        push    4C8A5B22h
.text:004124FA        push    edx
.text:004124FB        call    load_function
.text:00412500        add     esp, 0Ch
.text:00412503        mov     dword_41A36C, eax
```

Figure 4 Storing function address to DWORD after it is loaded

我们将会从函数开始往后遍历，去找到将 eax 的值移动到 DWORD 的指令。找到之后，我们会将 DWORD 值重命名为函数名。为了避免命名冲突，我们将会使用’d_’来作为名字前缀。

```
address = current_address
while address <= address_plus_30:
    address = NextHead(address)
    if GetMnem(address) == "mov":
        if 'dword' in GetOpnd(address, 0) and 'eax' in GetOpnd(address, 1):
            operand_value = GetOperandValue(address, 0)
            MakeName(operand_value, str("d_"+name))
```

将这几个步骤结合在一起，就可以将之前不可读的反汇编代码用更容易理解的形式替换。

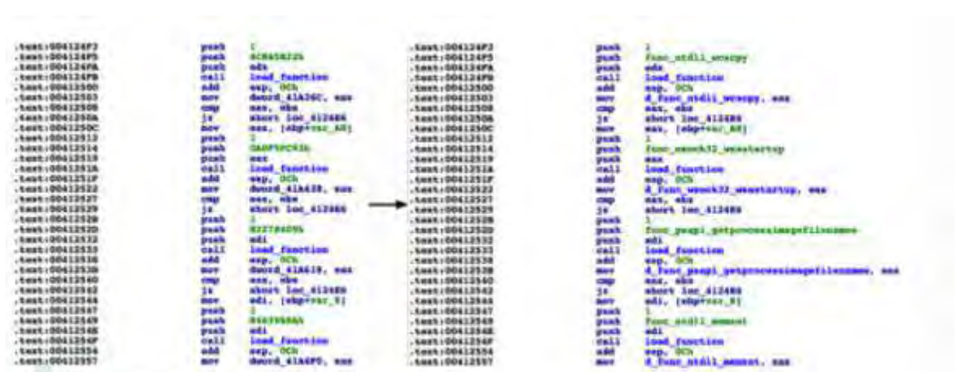


Figure 5 Changes after running script

当我们查看用来存储这些信息的 DWORDs 时，我们将会得到下面这样的函数名列表。这些数据可以被用来进行恶意软件的静态分析。

```
.data:0041A510 d_func_advapi32_cryptgethashparam dd ?  ; DATA XREF: sub_40ADB0+68'r
.data:0041A510                                         ; sub_4122E0+12E4'w
.data:0041A514 d_func_kernel32_resumethread dd ?       ; DATA XREF: sub_4122E0+928'w
.data:0041A518 d_func_shlwapi_urlescapea dd ?          ; DATA XREF: sub_4121E0+78'r
.data:0041A518                                         ; sub_4122E0+1532'w
.data:0041A51C d_func_wininet_internetconnecta dd ?    ; DATA XREF: sub_40B7F0+73'r
.data:0041A51C                                         ; sub_4122E0+CC1'w
.data:0041A520 d_func_user32_flashwindow dd ?          ; DATA XREF: sub_404DB0+75'r
.data:0041A520                                         ; sub_4122E0+7F1'w
.data:0041A524 d_func_kernel32_getlasterror dd ?       ; DATA XREF: sub_4030E0+1FB'r
.data:0041A524                                         ; sub_403F40+683'r ...
.data:0041A528 d_func_msvcrt_malloc dd ?               ; DATA XREF: .text:00406A89'r
.data:0041A528                                         ; sub_406BE0+1B7'r ...
.data:0041A52C d_func_kernel32_getmodulefilenamea dd ? ; DATA XREF: sub_4030E0+62'r
.data:0041A52C                                         ; sub_4122E0+8CE'w ...
.data:0041A530 d_func_kernel32_thread32first dd ?      ; DATA XREF: sub_4122E0+CA1'w
.data:0041A534 d_func_ntdll_memcmp dd ?                ; DATA XREF: sub_403F40+58B'r
.data:0041A534                                         ; sub_409890+130'r ...
.data:0041A538 d_func_ntdll_ntquerysysteminformation dd ? ; DATA XREF: sub_404710+33'r
.data:0041A538                                         ; sub_4122E0+FEE'w
.data:0041A53C d_func_ntdll_memcpy dd ?                ; DATA XREF: sub_402A90+A5'r
.data:0041A53C                                         ; sub_402A90+167'r ...
.data:0041A540 d_func_ntdll_strncat dd ?               ; DATA XREF: sub_40BD80+9C'r
.data:0041A540                                         ; sub_4122E0+512'w
.data:0041A544 d_func_ntdll_rwqueueapcthread dd ?      ; DATA XREF: sub_40B220+CE'r
.data:0041A544                                         ; sub_4122E0+347'w
.data:0041A548 d_func_kernel32_process32first dd ?     ; DATA XREF: sub_40C160+42'r
.data:0041A548                                         ; sub_4122E0+649'w
.data:0041A54C d_func_kernel32_getcurrentprocessid dd ? ; DATA XREF: sub_402EF0+61'r
.data:0041A54C                                         ; sub_404A20:loc_404AFB'r ...
.data:0041A550 d_func_advapi32_regenumvaluew dd ?      ; DATA XREF: sub_4122E0+6C3'w
.data:0041A554 d_func_ntdll_rtlgetversion dd ?         ; DATA XREF: sub_404A20+6D'r
.data:0041A554                                         ; sub_4122E0+1643'w
.data:0041A558 d_func_kernel32_createtoolhelp32snapshot dd ? ; DATA XREF: sub_40C160+E'r
.data:0041A558                                         ; sub_4122E0+73D'w
.data:0041A55C d_func_ole32_coinitializesecurity dd ?  ; DATA XREF: sub_405DD0+36'r
.data:0041A55C                                         ; sub_4122E0+293'w
.data:0041A560 d_func_oleaut32_sysallocstring dd ?     ; DATA XREF: sub_405EE0+7E'r
```

Figure 6 DWORD naming after script runs

完整的 IDAPython　脚本代码：

https://github.com/pan-unit42/public_tools/blob/master/ida_scripts/crc32_conversion.py

总结：

我们又一次用使用 IDAPython 解决了一个困难的问题（将 190 处 CRC32 hash 值转换为有意义的函数名）。枚举结构在这个问题的解决过程中起到了非常重要的作用。IDAPython 能够很方便的操作枚举变量，进行创建修改，为我们节省大量的时间。另外的，当我们在逆向其他样本遇到相同问题的时候，这些枚举数据可以导出或导入到 IDA 项目中。

*** 原文链接：researchcenter.paloaltonetworks,转载请注明来自 FreeBuf 黑客与极客（FreeBuf.COM）**

http://www.freebuf.com/sectool/92168.html

IDAPython：让你的生活更美好（三）

在过去两个部分中，我们已经讨论了使用 IDAPython 让逆向工程更容易一些。这一部分我们来看一下条件断点。

当在 IDA 中调试时，分析者经常会遇到希望可以在一个特殊的地址中断下来的情况，但这只有在一些特定的情况能够触发。

一个典型的例子：只有在特殊的参数传递进去的时候，才能断到一个特殊的函数的调用处。另外一个实例：我希望我的分析虚拟机加载一个特定的链接库时能够产生中断。

今天，我们来看看如何用 IDAPython 来解决这个特殊的问题。

背景

在进行分析时经常会逆向 DLL。在这类实例中，这些文件通常都是被其他可执行文件加载的。解决这个问题的一个方案是确保调试器在每个链接库加载时都能中断下来，然后 DLL 或者驱动最后加载完的时候才能停下来。

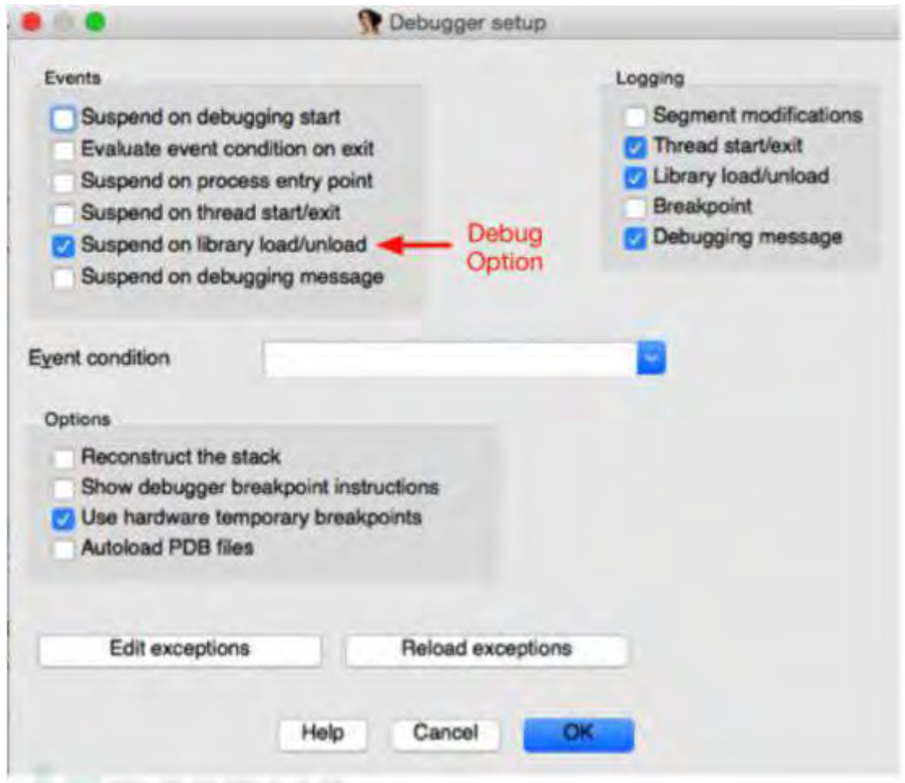


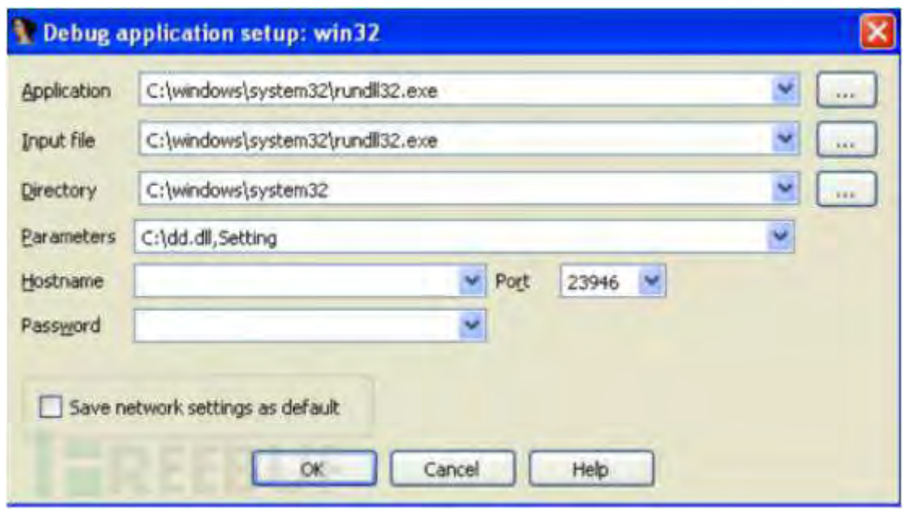
Figure 1 Debug option to break on each library load

但事实上这种方法其实是很低效的。通常需要分析者人工停止和开始：继续执行，判断哪些最近使用的链接库和驱动是想要断下来的。

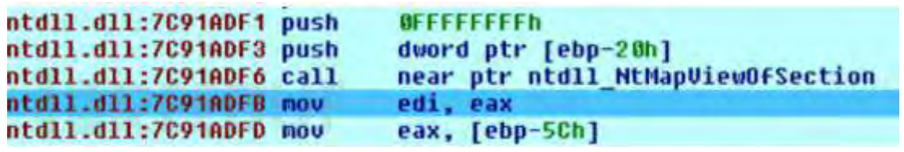
其实简单的运行一条命令，然后坐着等待感兴趣的文件被加载进来，会让分析工作变得更简单。

条件断点

在这个特殊的例子中，我们将会使用’dd.dll’文件。这个特殊的文件在运行时被提取之后会执行一个利用过程注入到其他进程中。为了在 IDA 中运行这个动态链接库，我将会加载可执行文件 rundll32.exe 将其关联到 system32 目录，这样能够将 dd.dll 文件作为参数传进来，就像下面这张图能够看到的。然后 rundll32.exe 可以让用户通过给定的导出名，加载一个特定的 DLL。在这个特定的样本中，我对 DLL 的导出函数’Setting’比较感兴趣。我在 IDA 中填入了下面这些参数：



下一个步骤是，在 DLL 被可执行文件加载进来之后找到正确的地方设置断点。 为了达到这个目的，我先在调试器中勾选了‘Suspend on library load/unload’选项。当暂停在第一个 DLL 实例被加载进来之后，我们可以看到断点设在了 NtMapViewOfSection 函数调用的后一句。



接下来我在 0x7C91ADFB 地址处下了一个断点。 在我的代码中，我调用 add_bpt() 和 enable_bpt() 函数来创建和生效断点。

```
,,,
ntdll.dll:7C91ADF1  push        0FFFFFFFFh
ntdll.dll:7C91ADF3  push        dword  ptr  [ebp-20h]
ntdll.dll:7C91ADF6  call        near  ptr  ntdll_NtMapViewOfSection
ntdll.dll:7C91ADFB  mov          edi,  eax
,,,


address  =  0x7C91ADFB  #  Just  after  NtMapViewOfSection
add_bpt(address,  0,  BPT_SOFT)
enable_bpt(address,  True)
```

在这个步骤，我已经在 0x7C91ADFB 处设置了断点，每次 DLL 加载进来的时候调试器都会中断下来。为了确保只有在’dd.dll’加载进来的时候才会中断，我们必须使用条件断点。分析者可以使用代码来判断条件断点是否真的被触发（不管用 IDC 或者 Python 都行）。如果代码返回值为 True，断点会被触发，否则，断点将会被忽略。 我们首选 Python 作为编程语言。 然后我们将 Python 代码存储到变量中，然后将这个变量传递给 SetBptCnd()函数，这个函数会将这个代码作为条件设置到断点上。 这个条件设置之后，我们让调试器继续运行，然后等待调试器暂停事件触发。伪代码如下所示：

```
address  =  0x7C91ADFB  #  Just  after  NtMapViewOfSection
RunPlugin("python",  3)  #  Python  default  programming
StartDebugger("","","");

dll  =  "dd.dll"
condition  =  """
for  m  in  Modules():
    if  "%s".lower()  in  m.name.lower():
        print  "Breaking  on",  m.name.lower()
        del_bpt(%d)
        return  True
return  False
"""  %  (dll,  address)

add_bpt(address,  0,  BPT_SOFT)
enable_bpt(address,  True)
SetBptCnd(address,  condition)

continue_process()
GetDebuggerEvent(WFNE_SUSP,  -1)
```

现实中使用的条件代码如下所示：

```
for  m  in  Modules():
    if  "dd.dll".lower()  in  m.name.lower():
        print  "Breaking  on",  m.name.lower()
        del_bpt(0x7C91ADFB)
        return  True
return  False
```

这个代码会反复的查找’dd.dll’是否被加载进来。 如果是，会打印调试信息，最开始的断点被删除掉了，返回的布尔值为 True 时，断点就会被触发。我们运行程序，IDA 窗口中输出如下信息：

下面这个步骤，我们会对导出函数’Setting’设置断点然后运行程序直到断点触发。为了自动完成这个步骤，我们可以使用下面的代码：

```
def  get_names(base,  size,  desired_name):
    current_address  =  base
    while  current_address  &lt;=  base+size:
            current_address  =  NextHead(current_address)
            print  hex(current_address)
            if  desired_name  in  Name(current_address):
                return  current_address

for  m  in  Modules():
    if  'dd.dll'  in  m.name.lower():
        base  =  m.base
        size  =  m.size
        analyze_area(base,  base+size)
        setting  =  get_names(base,  size,  "Setting")
        if  setting:
            add_bpt(setting,  0,  BPT_SOFT)
            enable_bpt(setting,  True)
            continue_process()
            GetDebuggerEvent(WFNE_SUSP,  -1)
```

这段代码反复的在加载的模块中查找’dd.dll’。找到之后，分析包含 DLL 的代码。 然后在这段代码中寻找’Setting’函数。一旦找到，就在这个地址设置断点，让调试器继续执行，等待断点触发。执行程序，我们可以看到正如我们预期的那样断点在我们想要的地址上中断下来。

```
dd.dll:10003500
dd.dll:10003500 dd_Setting:
dd.dll:10003500 push    ebp
dd.dll:10003501 mov     ebp, esp
dd.dll:10003503 sub     esp, 514h
dd.dll:10003509 xor     eax, eax
dd.dll:1000350B push    206h
dd.dll:10003510 push    eax
dd.dll:10003511 mov     [ebp-30Ch], ax
dd.dll:10003518 lea     eax, [ebp-308h]
dd.dll:1000351E push    eax
dd.dll:1000351F call    sub_10002308
dd.dll:10003524 lea     ecx, [ebp-30Ch]
dd.dll:1000352A call    sub_10001E00
dd.dll:1000352F push    offset aRasphone_dll          ; "rasphone.dll"
```

结论：

设置条件断点看起来是一个非常小的技术点， 但是能够节省分析者大量的时间。一个很小的代码片段能够代替我们完成一次次手动的寻找我们想要的断点这么繁琐的工作。我们又一次借助 IDAPython 的力量来简化我们的逆向工作，帮助我们节省了大量的时间和精力。

***原文：Paloalto 东二门陈冠希/编译，转载请注明来自 FreeBuf 黑客与极客（FreeBuf.COM）**

http://www.freebuf.com/articles/system/92488.html

IDAPython：让你的生活更美好（四）

前三部分已经验证了用 IDAPython 能够让工作变的更简单，这一部分让我们看看逆向工程师如何使用 IDAPython 的颜色和强大的脚本特性。

分析者经常需要面对越来越复杂的代码，而且有时候无法轻易看出动态执行的时候执行的代码。而通过 IDAPython 的强大功能，我们不但能静态的标识指令，并且能够统计出对应的指令被使用了多少次。

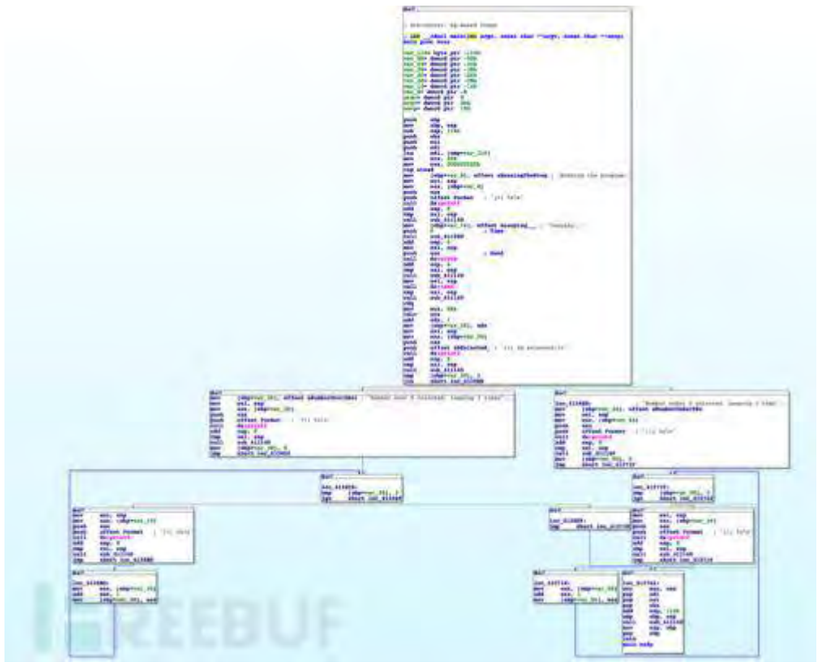
背景

在这一部分中，我用 C 语言写了一个简单的程序。下面的代码是为了这次的练习而编写和编译的：

```
#include  "stdafx.h"
#include  <stdlib.h>
#include  <time.h>

int  _tmain(int  argc,  _TCHAR*  argv[])
{
    char*  start  =  "Running  the  program.";
```

```
printf("[+] %s\n", start);
char* loop_string = "Looping...";

srand (time(NULL));
int bool_value = rand() % 10 + 1;
printf("[+] %d selected.\n", bool_value);
if(bool_value > 5)
{
    char* over_five = "Number over 5 selected. Looping 2 times.";
    printf("[+] %s\n", over_five);
    for(int x = 0; x < 2; x++)
        printf("[+] %s\n", loop_string);
}
else
{
    char* under_five = "Number under 5 selected. Looping 5 times.";
    printf("[+] %s\n", under_five);
    for(int x = 0; x < 5; x++)
        printf("[+] %s\n", loop_string);
}
return 0;
}
```

当我们将这个二进制加载到 IDA 中时，我们可以看到预期的循环与代码重定向语句。如果我们在不知道源码的情况下来看这个例子，通过静态分析能够大概判断代码实现的功能。



然而，如果我们想知道运行的时候执行了哪个区块的代码呢？这个问题可以用 IDAPython 来解决哦！

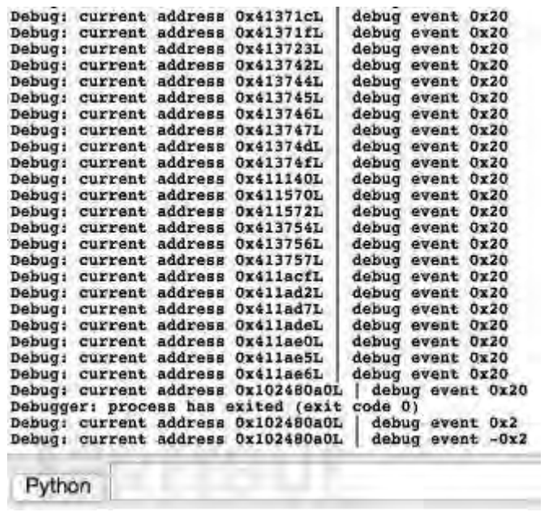
编写 IDAPYTHON 脚本

我们第一个需要处理的难题是如何逐句遍历每一条指令，以下代码将可以帮助我们来解决：（调试信息会输出已经被执行的指令）

```
RunTo(BeginEA())
event  =  GetDebuggerEvent(WFNE_SUSP,  -1)

EnableTracing(TRACE_STEP,  1)
event  =  GetDebuggerEvent(WFNE_ANY|WFNE_CONT,  -1)

while  True:
    event  =  GetDebuggerEvent(WFNE_ANY,  -1)
    addr  =  GetEventEa()
    print  "Debug:  current  address",  hex(addr),  "|  debug  event",  hex(event)
    if  event  <=  1:  break
```

在上面的代码中我们首先启动了调试器并且执行到函数的入口处，通过调用‘RunTo(BeginEA())’函数。接下来调用的 GetDebuggerEvent()函数会等待直到断点到达。

接着我们调用 EnableTracing()函数来打开 IDA 的跟踪功能，然后 GetDebuggerEvent()函数调用会继续执行调试器，配置跟踪步骤。最后，我们会进入一个循环遍历每一个地址直到遇到结束条件。这个脚本在 IDA 中的输出如下所示：

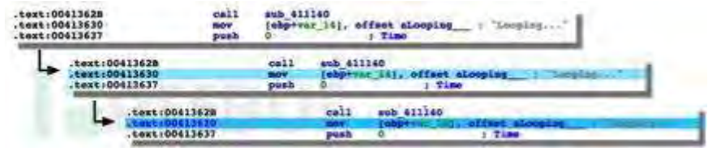


下一个步骤是检索出执行的每一行并标识颜色。我们可以使用 GetColor()和 SetColor()函数来分别解决这两个问题。下面的代码会获取给定行数代码的当前颜色，并决定将代码设置成什么颜色，并进行设定。

在这个例子，我使用四种不同深浅的蓝色。深蓝色表示这一行被重复执行。（读者可以根据跟人喜好修改这部分代码）

```
def  get_new_color(current_color):
    colors  =  [0xffe699,  0xffcc33,  0xe6ac00,  0xb38600]
    if  current_color  ==  0xFFFFFF:
        return  colors[0]
    if  current_color  in  colors:
        pos  =  colors.index(current_color)
        if  pos  ==  len(colors)-1:
            return  colors[pos]
        else:
            return  colors[pos+1]
    return  0xFFFFFF


current_color  =  GetColor(addr,  CIC_ITEM)
new_color  =  get_new_color(current_color)
SetColor(addr,  CIC_ITEM,  new_color)
```

运行上面的代码能够将无颜色的行数修改为高亮的蓝色。另外，如果同一行代码运行多次会变成深蓝色。



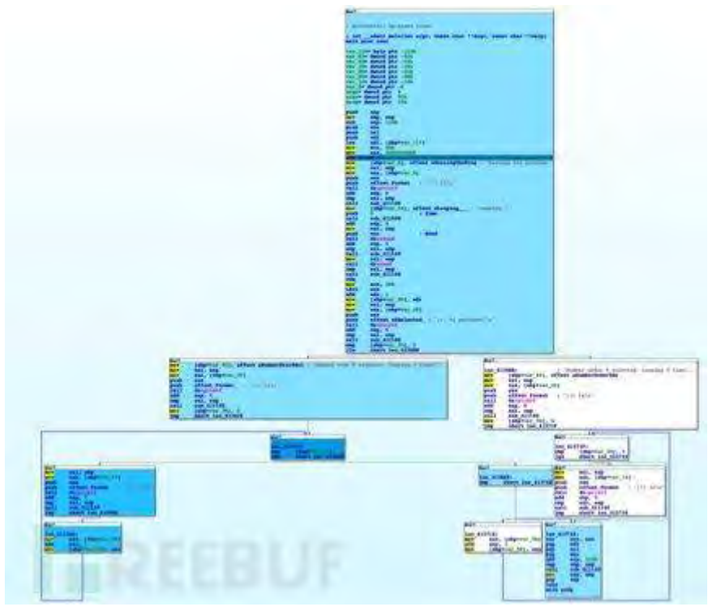
可以使用下面的代码来删除 IDA 文件中之前设置的所有颜色。将颜色设置成 0XFFFFFF 将会变成白色，或者高效的将之前设置的所有颜色删除。

```
heads = Heads(SegStart(ScreenEA()), SegEnd(ScreenEA()))
for i in heads:
    SetColor(i, CIC_ITEM, 0xFFFFFF)
```

将所有的代码合到一起，我们会得到如下结果：

```
heads = Heads(SegStart(ScreenEA()), SegEnd(ScreenEA()))
for i in heads:
    SetColor(i, CIC_ITEM, 0xFFFFFF)

def get_new_color(current_color):
    colors = [0xffe699, 0xffcc33, 0xe6ac00, 0xb38600]
    if current_color == 0xFFFFFF:
        return colors[0]
    if current_color in colors:
        pos = colors.index(current_color)
        if pos == len(colors)-1:
            return colors[pos]
        else:
            return colors[pos+1]
    return 0xFFFFFF

RunTo(BeginEA())
event = GetDebuggerEvent(WFNE_SUSP, -1)

EnableTracing(TRACE_STEP, 1)
event = GetDebuggerEvent(WFNE_ANY|WFNE_CONT, -1)
while True:
    event = GetDebuggerEvent(WFNE_ANY, -1)
    addr = GetEventEa()
    current_color = GetColor(addr, CIC_ITEM)
    new_color = get_new_color(current_color)
    SetColor(addr, CIC_ITEM, new_color)
    if event <= 1: break
```

当我在程序用运行这段代码时，我们看到执行过的反汇编指令被高亮了。如下图所示，多次执行的指令被设置成了深蓝色，让我们能够更容易理解代码执行流程。



总结

这一部分演示的例子确实很简单，结合了 IDA 调试功能与颜色相关 API 的使用。这项技术能够让分析者在复杂应用代码分析中节省大量的时间。

*原文：Paloaltone，东二门陈冠希/编译，转载请注明来自 FreeBuf 黑客与极客（FreeBuf.COM）

http://www.freebuf.com/articles/system/92505.html

IDAPython：让你的生活更美好（五）

我们继续 IDAPython 让生活更美好序列，这一部分我们解决逆向工程师日常遇到的问题：提取执行的内嵌代码。

恶意软件会用各种方式存储内嵌可执行代码，有些恶意软件将内嵌代码加到文件附加段，包括 PE 资源区段，或者存放在恶意软件的缓冲区中。

当遇到这个情况，恶意软件分析者可以有几个选择：

可以动态运行样本在写入和提取的后面下断点或者如果文件存储在资源段，可以使用一些工具比如 CFFExplorer 提取资源数据，在 IDA 中可以高亮选取可疑的二进制数据，然后右键保存想要的提取的数据。

虽然这几个方法都可行，但是都有一些限制。而自动化提取内嵌代码可以节省分析者大量的时间。为了实现这个目的，我们会用到 IDAPython 的第三方链接库组件‘pefile’。而这里也会带来一些挑战：

1. 我们必须在 IDA 环境中用 PIP 安装第三方 python 链接库
2. 已经标识出了内嵌代码
3. 需要计算要提取的可执行代码的大小

让我们一次性的解决这些问题吧。

在 IDA PRO 中加入第三方 PYTHON 链接库

在 IDA 中用 PIP 安装第三方 python 链接库之后，如何让其生效是一个有趣的挑战。而如果不修改的话是没有办法加载第三方链接库的，比如 pefile 的 IDAPython 解释中会出现如下错误。

```
Python>import pefile
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named pefile
Python>
```

为了修正这个错误，我们必须将 PIP 的‘site-packages’目录加到 PYTHON 的环境变量中。可以使用如下代码实现：

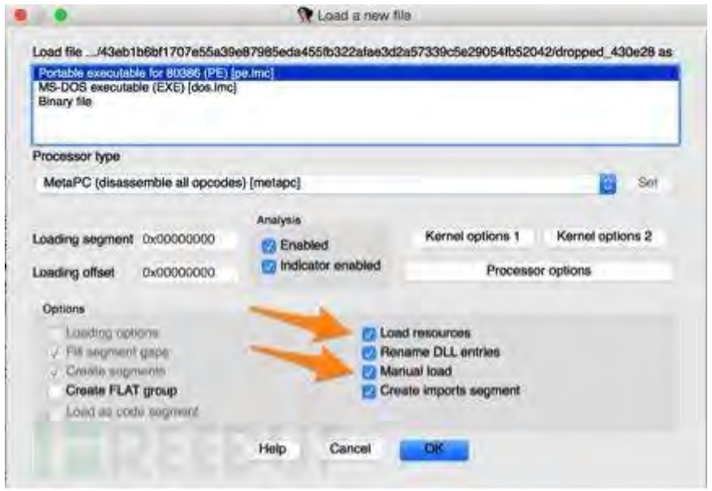
```
import  sys
print  sys.path

Result:
['/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python27.zip',  '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7',  '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-darwin',  '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac',  '<Truncated>']
```

而这里为了包含 PIP 安装链接库，我们可以简单的将‘site-packages’目录加到 pefile 包含声明数组中。但是这个方案不太好，因为这需要分析者人工识别‘site-packages’目录，但是我没要找到跨平台的解决方案。加入相关代码之后，我们就能够加载 pefile 链接库了。

```
Python>import sys
Python>sys.path.append('/usr/local/lib/python2.7/site-packages')
Python>import pefile
Python>
```

需要内嵌代码

为了找到恶意软件包含的所有内嵌代码，我们基于 MZ 头的已知字符串对二进制进行搜索。分析者请确认已经勾选了‘Load resources’选项这样才能够读取到作为资源存储的所有数据。另外的，如果内嵌文件包含在附加段中，为了在 IDA 中能够看到数据一定要勾选’Manual load’选项。



现在我们已经有了 IDA 中加载的必要信息了，现在我们可以开始在 PE32 文件中搜索数据了。我们有好几种方法可以实现，但我选择搜索所有 MZ 头中都会包含的静态信息，如下所示：

```
Python>find_embedded_exes()
[*] Identified embedded executable at the following offset: 0x1000b1c0
[*] Identified embedded executable at the following offset: 0x1001a9c0
```

为了找到 IDA 中所有的字符串事件，我们可以使用循环调用 FindBinary()函数来寻找二进制字符串的每一次实例。代码如下：

```
def  find_string_occurrences(string):
    results  =  []
    base  =  idaapi.get_imagebase()  +  1024
    while  True:
        ea  =  FindBinary(base,  SEARCH_NEXT|SEARCH_DOWN|SEARCH_CASE,  '"%s"'  %  string)
        if  ea  !=  0xFFFFFFFF:
            base  =  ea+1
```

```
        else:
            break
        results.append(ea)
    return  results
```

当在 PE32 文件中寻找 MZ 头字符串标识时，我们需要验证‘MZ’字符存在于 MZ 头的开始处。由于我们之前找的字符串在静态偏移是固定的，我们只需要简单的确定‘MZ’的已知偏移就可以了。

```
def  find_embedded_exes():
    results  =  []
    exes  =  find_string_occurrences("!This  program  cannot  be  run  in  DOS  mode.")
    if  len(exes)  >  1:
        for  exe  in  exes:
            m  =  Byte(exe-77)
            z  =  Byte(exe-76)
            if  m  ==  ord("M")  and  z  ==  ord("Z"):
                mz_start  =  exe-77
                print  "[*]  Identified  embedded  executable  at  the  following  offset:  0x%x"  %  mz_start
                results.append(mz_start)
    return  results
```

将上面的代码组合到一起，来找到 IDA 中所有的内嵌代码。

```
Python>find_embedded_exes()
[*] Identified embedded executable at the following offset: 0x1000b1c0
[*] Identified embedded executable at the following offset: 0x1001a9c0
```

确定可执行代码的大小

为了确定找到的内嵌代码的大小，我们将使用前面提到的 python 第三方 pefile 链接库。这个链接库可以解析各种可执行文件头，这样我们就能够计算 PE 文件的大小了。为了实现这个目的，我们会在可选头中加入’SizeOfHeaders’参数，连同每个段的’SizeOfRawData’字段。下面的代码会读出标识出的内嵌代码的前 1024 字节，用 pefile 解析这些数据，计算各个段的大小。

```
def  calculate_exe_size(begin):
    buff  =  ""
    for  c  in  range(0,  1024):
        buff  +=  chr(Byte(begin+c))
    pe  =  pefile.PE(data=buff)
    total_size  =  0
    #  Add  total  size  of  headers
    total_size  +=  pe.OPTIONAL_HEADER.SizeOfHeaders
    #  Iterate  through  each  section  and  add  section  size
    for  section  in  pe.sections:
        total_size  +=  section.SizeOfRawData
    return  total_size
```

最后，我们可以使用这些大小值来提取可执行代码数据然后写入我们选择的文件中。

```
def  extract_exe(name,  begin,  size):
    buff  =  ""
    for  c  in  range(0,  size):
        buff  +=  chr(Byte(begin+c))
    f  =  open(name,  'wb')
    f.write(buff)
    f.close()
```

结论

将这些组合到一起，我们将会得到下面的脚本。

https://github.com/pan-unit42/public_tools/blob/master/ida_scripts/idapython_pt5.py

在恶意样本中运行这个例子，将会得到下面的结果：

```
Python>
[*] Identified embedded executable at the following offset: 0x1000b1c0
[*] Identified embedded executable at the following offset: 0x1001e9c0
[*] Wrote /Users/jgrunzweig/Desktop/dropped_0x1000b1c0
[*] Wrote /Users/jgrunzweig/Desktop/dropped_0x1001e9c0
```

正如我们看到的，我们能够在 IDA 中自动提取 PE 文件了。通过一些小修改，这个实现能够应用到其他类型的文件中。我希望这个教程能够让逆向工程师知道 IDAPython 能够实现很多难以置信的功能。

***原文地址：Paloaltonetworks，东二门陈冠希/编译，转载请注明来自 FreeBuf 黑客与极客（FreeBuf.COM）**

http://www.freebuf.com/articles/system/93440.html

ida idc 脚本实现加密指令修改

移动应用中对 so 文件，有些函数用了 mprotect 进行保护，即将加密数据解密后再在内存执行，然后再将内存数据加密后写回原位置，通常解密后数据具有反调试功能。
破解应对措施：ida 调试跟踪后内存加密数据解密还原及加密算法提取完毕后，将密钥的内存数据和修改后的内存数据地址作为 idc 脚本输入，打印输出生成的加密数据或指令，从而利用 UE 修改 so，实现永久修改的目的，后续直接跟踪调试 so 即可。

将下面的脚本保存为 xx.idc 后在 ida 中 shrift+F2 导入执行即可。

```
import idaapi
import struct
```

```
#input
def dump24c8(start, len, key, target):
  rawInc = idaapi.dbg_read_memory(start, len)
  offset = start-0x70d1a4c8
  rawIncByte = bytearray(rawInc)
  rawIncHex = struct.unpack('<I', rawInc)[0]
  print 'rawIncHex is ' + str(hex(rawIncHex))

  rawkey = idaapi.dbg_read_memory(key, 0x6C)
  rawkeyByte = bytearray(rawkey)
  #rawkeyHex = struct.unpack('<I', rawkey)[0]
  #print 'rawkeyHex is ' + str(hex(rawkeyHex))
  #rawdex = "hello"
  count = 0
  offset %= 0x6C
  fd = open(target, 'wb')
  while (count < len):
      print 'The count is:', count
      rawIncByte[count] ^= rawkeyByte[(offset + count) % 0x6C]
      fd.write(rawInc)
      count = count + 1
  fd.close()
  rawIncHex = struct.pack('<BBBB', rawIncByte[0],rawIncByte[1],rawIncByte[2],rawIncByte[3])
  rawIncHex1 = struct.unpack('<I', rawIncHex)[0]
  print 'rawIncHex is ' + str(hex(rawIncHex1))

def dump14a4(start, len, key, target):
  rawInc = idaapi.dbg_read_memory(start, len)
  print "rawInc is " + rawInc
  offset = start-0x70d194a4
  print "offset is " + str(offset)
  rawIncByte = bytearray(rawInc)
  #rawIncByte[0] = 0x39
  #rawIncByte[1] = 0x00
  #rawIncByte[2] = 0x00
  #rawIncByte[3] = 0x1A
  rawIncHex = struct.unpack('<I', rawInc)[0]
  print 'rawIncHex is ' + str(hex(rawIncHex))

  rawkey = idaapi.dbg_read_memory(key + 0x6C, 0x6C)
  rawkeyByte = bytearray(rawkey)
  #rawkeyHex = struct.unpack('<I', rawkey)[0]
```

```
  #print 'rawkeyHex is ' + str(hex(rawkeyHex))
  #rawdex = "hello"
  count = 0
  offset %= 0x6C
  print "offset is " + str(offset)
  fd = open(target, 'wb')
  while (count < len):
      print 'The count is:', count
      rawIncByte[count] ^= rawkeyByte[(offset + count) % 0x6C]
      fd.write(rawInc)
      count = count + 1
  fd.close()
  rawIncHex = struct.pack('<BBBB', rawIncByte[0],rawIncByte[1],rawIncByte[2],rawIncByte[3])
  rawIncHex1 = struct.unpack('<I', rawIncHex)[0]
  print 'rawIncHex is ' + str(hex(rawIncHex1))

def getdexlen(start):
  pos = start + 0x20
  mem = idaapi.dbg_read_memory(pos, 4)
  len = struct.unpack('<I', mem)[0]
  print 'len is ' + str(hex(len))
  return int(len)


#input start is 0x78960 len is 0x200 target is c:\\xx.raw
start = AskAddr(0, 'Input instructor start addr in hex: ')
len = AskLong(0, 'Input instructor len in hex: ')
key = AskAddr(0, 'Input key addr in hex: ')
target = AskStr('c:\\ins.txt', 'Input the dump file path')

print('start is ' + str(hex(start)) + " len is " + str(len) + "key is " +    str(hex(key)) + " target is " + target )

if len > 0 and start >= 0x0 and key >= 0 and target and AskYN(1, 'start is 0x%0x, len is %d, enc dump to %s' % (start, len, target)) == 1:
  dump14a4(start, len, key,target)
  print('Dump Finish')
```

http://blog.csdn.net/jiayanhui2877/article/details/49779925