第7章 代码混淆

北京邮电大学
崔宝江

第7章 代码混淆

- 一. 代码混淆简介
- 二. 花指令混淆
- 三. SMC代码自修改
- 四. OLLVM混淆

北邮网安学院　崔宝江

一. 代码混淆简介

- 1 代码混淆概念
- 2 代码混淆种类以及工具介绍

一. 代码混淆简介

- 1 代码混淆概念
  - 为了在一定程度上阻止逆向人员进行分析，在源代码和中间层代码层面产生了代码混淆的概念。
  - 通过技术手段实现计算机程序的代码转换，形成一种功能上等价，但是从代码层分析相对复杂的程序，增加逆向分析人员的分析成本。
  - 代码混淆的根本目的就是干扰逆向工程，保护知识产权等。而保护程序不被逆向又有多种方式，代码混淆(Obfuscation)是其中的一种。

一. 代码混淆简介

- 2代码混淆种类以及工具介绍
  - 从增加逆向工程难度的角度上来说，主要有
    - （1）源码级混淆
    - （2）机器码混淆

混淆种类	逆向分析难度
混淆	容易
源码级混淆	中等
机器码混淆	最难

一. 代码混淆简介

- （1）源码级混淆
  - 标识符重命名
  - 等价表达式
  - 代码重排
  - 花指令
  - 自解密

一. 代码混淆简介

- （1）源码级混淆
  - 标识符重命名
    - 即将代码中的各种元素，如变量，函数，类的名字改写成无意义的名字。比如改写成单个字母或者数字，又或者字母和数字的组合等，使得阅读的人无法根据名字猜测其用途。
  - 等价表达式
    - 重写代码中的部分逻辑，将其变成功能上等价，但是更难理解的形式。比如将循环改成递归，精简中间变量等。

一. 代码混淆简介

- （1）源码级混淆
  - 代码重排
    - 打乱原有代码格式。比如将多行代码挤到一行代码中。
  - 花指令
    - 通过构造字节码插入程序的适当位置，使得反汇编器出错，产生无法反编译或者反编译出错的情况。
  - 自解密
    - 通过对程序部分进行加密，在即将运行时代码进行自解密，然后执行解密之后的代码。

一. 代码混淆简介

- （2）机器码混淆
  - 平坦控制流
    - 就是将程序原有的顺序、选择、循环结构统一重构为switch结构，使得程序的结构图从原有正常形态转变为扁平状
  - 伪造控制流
    - 构造出根本不会去真正执行的控制流，从而在静态分析时，会对分析形成强烈的干扰，增加逆向人员的分析工作量
  - 指令替换
    - 将原有的一条或者几条能形成某种功能的指令，替换为大量的指令，从而增加逆向分析的代码量，从视觉上产生一种复杂感，增加逆向分析的难度

一. 代码混淆简介

- 程序员们开发出了混淆器对代码进行混淆
  - 对JS进行混淆的工具有诸如YUI Compressor，Google Closure Compiler,UglifyJS，JScrambler等
  - 在windows下进行代码混淆最出名的工具
    - VMProtect，VMP能够实现代码虚拟化，将部分或者全部代码使用VMP混淆，将该部分代码转换为自身才能解释执行的代码，按照VMP自身实现的虚拟机架构去解释执行
      - 相当于将中文转换为了英文，你只有掌握了英文的词，语法和句，才能看懂英文写的文章。

一. 代码混淆简介

- 程序员们开发出了混淆器对代码进行混淆
  - OLLVM混淆器
    - LLVM 命名最早源自于底层虚拟机（Low Level Virtual Machine），是伊利诺伊大学的一个研究项目
      - 提供了一套中立的中间代码IR和编译基础设施，并围绕这些设施提供了一套全新的编译策略（使得优化能够在编译、连接、运行环境执行过程中，以及安装之后以有效的方式进行）
    - Obfuscator-LLVM是一个开源项目
      - 适用于LLVM所支持的所有语言(C,C++,Objective-C,Ada和Fortran)和目标平台(x86, x86-64, PowerPC, PowerPC-64, ARM, Thumb, SPARC, Alpha, CellSPU, MIPS, MSP430, SystemZ,和Xcore)

第7章 代码混淆

- 一. 代码混淆简介
- 二. 花指令混淆
- 三. SMC代码自修改
- 四. OLLVM混淆

北邮网安学院　崔宝江

二. 花指令混淆

- 花指令原理
  - 花指令通常用来抵御静态分析，通过花指令混淆的程序，会扰乱汇编代码的可读性。在静态分析下，使反汇编器无法正常解析，反编译器无法正常反编译。

二. 花指令混淆

- 花指令原理
  - 在反汇编的过程中，存在一个数据与代码的区分问题
  - 而不同字节码包含的字节数不同，有单字节指令，也有多字节指令。
  - 如果首字节是多字节指令，反汇编器在确定了第一个指令，也就是操作码以后，就会确定该指令是包含多少字节码的指令，然后将这些字节码转化为一条汇编指令。
    - 举个例子，0xE8是x86中call指令的操作码，它后面通常要跟4个字节码，当反汇编器解析到0xE8这个字节码之后，会将后4个字节码连同0xE8一起转化为一条call指令。

二. 花指令混淆

- 说明
  - 汇编码（**Assembly Code**）是用 人类可读的 汇编语言助记符 书写的代码。
  - 机器码（**Machine Code**）是用 硬件可执行的 二进制 表示的代码。
  - 十六进制码（**Hexadecimal Code**） 是用 人类可读的 十六进制 表示的代码

二. 花指令混淆

- 说明
  - 几种跳转指令和对应的机器码
    - 0xE8　CALL　后面的四个字节是地址
    - 0xE9　JMP　后面的四个字节是偏移
    - 0xEB　JMP　后面的二个字节是偏移

二. 花指令混淆

- 两类反汇编算法
  - 线性扫描算法(Linear Sweep)
    - 将一条指令的结束作为另一条指令的开始
    - 从第一个字节开始，以线性模式扫描整个代码段，逐条反汇编每条指令，直到完成整个代码段的分析。但是却没有考虑代码中可能混有的数据，容易出错。
  - 递归行进算法(Recursive traversal)
    - 对代码可能的执行路径进行扫描，当解码出分支指令后，就把这个分支指令的地址记录下来，并反汇编各个分支中的指令。
    - 这种算法可以避免将代码中的数据作为指令解析。

二. 花指令混淆

- 花指令原理
  - 在正常的程序中巧妙嵌入数据，使得反汇编器在解析的时候，误认为是代码一同解析，从而在静态分析层面干扰了逆向分析者，这就是花指令。
  - 花指令需要逆向分析者花时间将这些数据从程序中剔除掉，还原原有正常程序，从而正常地实现反汇编和反编译。

二. 花指令混淆

- 常见花指令混淆手段
  - Call, Jmp类指令形成的混淆
  - 更改IDA识别的栈指针，破坏栈平衡
  - 更改IDA识别的函数参数个数

二.花指令混淆

- 1 Call, Jmp类指令形成的混淆
  - 没有做过任何混淆的程序代码

```
#include<stdio.h>
#include<Windows.h>
int main(int argc,char*argv[])
{
	printf("******************So Let's play a Game!******************\n");
	printf("******************So Let's play a Game!******************\n");
	system("pause");
	return 0;
}
```


二.花指令混淆

- 使用IDA打开程序进行查看，并记住混淆前的程序

```
int main_0()
{
  char v1; // [sp+Ch] [bp-40h]@1

  memset(&v1, 0xCCu, 0x40u);
  printf("********************So Let's play a Game!********************\n");
  printf("********************So Let's play a Game!********************\n");
  system("pause");
  return 0;
}
```


二.花指令混淆

- （1）Call指令混淆
- （2）Jmp Short类型混淆
- （3）Jmp类型混淆

二. 常见的花指令混淆手段

- （1）Call指令混淆
  - Call代码进行混淆，混淆源代码如下

```
#include<stdio.h>
#include<Windows.h>
int main(int argc,char*argv[])
{
    printf("*****************So Let's play a Game!*****************\n");
    __asm
{
xor eax,eax
        jz __label    //jz跳转到下面的__label，导致__emit 0xE8无法执行，即混淆了
        __emit 0xE8
}
__label:
    printf("*****************So Let's play a Game!*****************\n");
    system("pause");
return0;
}
```

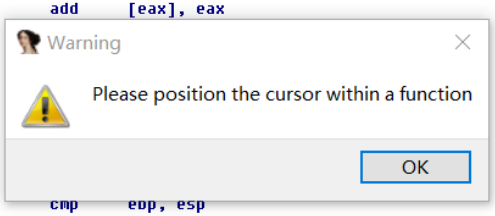
二. 常见的花指令混淆手段

- 说明
  - __asm 关键字用于调用内联汇编程序，并且可在 C 或 C++ 语句合法时出现
  - __emit 可以使数据以代码的形式写入到代码段，也就是shellcode中。用emit就是在当前位置直接插入数据（实际上是指令），一般是用来直接插入汇编里面没有的特殊指令

二. 常见的花指令混淆手段

- 修改后的程序，IDA已经无法正常进行反编译（快捷键F5），成功地实现了混淆IDA反编译的作用。

```
.text:00401010 loc_401010:                             ; CODE XREF: .text:_main↑j
.text:00401010                 push    ebp
.text:00401011                 mov     ebp, esp
.text:00401013                 sub     esp, 40h
.text:00401016                 push    ebx
.text:00401017                 push    esi
.text:00401018                 push    edi
.text:00401019                 lea     edi, [ebp-40h]
.text:0040101C                 mov     ecx, 10h
.text:00401021                 mov     eax, 0CCCCCCCCh
.text:00401026                 rep stosd
.text:00401028                 push    offset aSoLetSPlayAGam ; "*****************So Let's play a Game!*"...
.text:0040102D                 call    _printf
.text:00401032                 add     esp, 4
.text:00401035                 xor     eax, eax
.text:00401037                 jz      short near ptr loc_401039+1
.text:00401039
.text:00401039 loc_401039:                             ; CODE XREF: .text:00401037↑j
.text:00401039                 call    near ptr 42603A86h
.text:0040103E                 add     al, ch
.text:00401040                 dec     esp
.text:00401041                 add     [eax], eax
.text:00401043
.text:00401049
.text:0040104C
.text:00401051
.text:00401054
.text:00401056
.text:00401057
.text:00401058
.text:00401059
.text:0040105C                 cmp     ebp, esp
.text:0040105E                 call    __chkesp
.text:00401063                 mov     esp, ebp
.text:00401065                 pop     ebp
.text:00401066                 retn
```



二. 常见的花指令混淆手段

- 原理分析
  - 反汇编器的一种反汇编算法是线性扫描算法，当IDA解析到0xE8的时候，自动与其后4个字节码组成了一个call指令，从而混淆了代码。
  - E8后面需要4个字节，由于程序解析多字节指令时，区分不了数据和代码产生的错误。

二. 常见的花指令混淆手段

- （2） Jmp Short类型混淆
  - 接着看看Jmp Short类型混淆

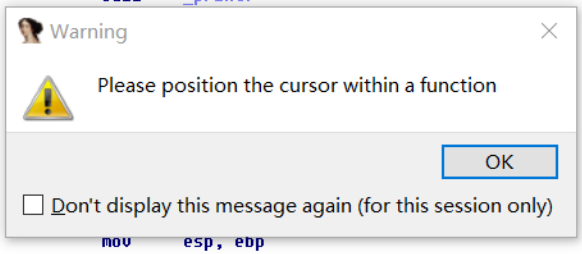
```
#include<stdio.h>
#include<Windows.h>
int main(int argc,char*argv[])
{
    printf("*****************So Let's play a Game!*****************\n");
    __asm
{
xor eax,eax
        jz __label
        __emit 0xEB
}
__label:
    printf("*****************So Let's play a Game!*****************\n");
    system("pause");
return0;
}
```

一. 代码混淆简介

- 说明
  - 几种跳转指令和对应的机器码
    - 0xE8　CALL　后面的四个字节是地址
    - 0xE9　JMP　后面的四个字节是偏移
    - 0xEB　JMP　后面的二个字节是偏移

二. 常见的花指令混淆手段

```
.text:00401010                push    ebp
.text:00401011                mov     ebp, esp
.text:00401013                sub     esp, 40h
.text:00401016                push    ebx
.text:00401017                push    esi
.text:00401018                push    edi
.text:00401019                lea     edi, [ebp-40h]
.text:0040101C                mov     ecx, 10h
.text:00401021                mov     eax, 0CCCCCCCCh
.text:00401026                rep stosd
.text:00401028                push    offset aSoLetSPlayAGam ; "**********So Let's play a Game!**"...
.text:0040102D                call    _printf
.text:00401032                add     esp, 4
.text:00401035                xor     eax, eax
.text:00401037                jz      short near ptr loc_401039+1
.text:00401039
.text:00401039 loc_401039:                            ; CODE XREF: .text:00401037↑j
.text:00401039                jmp     short loc_4010A3
.text:00401039 ; ---------------------------------------------------------------------------
.text:0040103B                dd offset aSoLetSPlayAGam ; "**********So Let's play a Game!**"...
.text:0040103F ; ---------------------------------------------------------------------------
.text:0040103F                call    _printf
.text:00401044
.text:00401047
.text:0040104C
.text:00401051
.text:00401054
.text:00401056
.text:00401057
.text:00401058
.text:00401059
.text:0040105C
.text:0040105E
.text:00401063                mov     esp, ebp
.text:00401065                pop     ebp
.text:00401066                retn
```



北邮网安学院　崔宝江

二. 常见的花指令混淆手段

- 原理分析
  - 当IDA解析到0xEB的时候，自动与其后2个字节码组成了一个JMP 偏移地址的指令，从而混淆了代码。

二. 常见的花指令混淆手段

❒（3）Jmp类型混淆

```
#include<stdio.h>
#include<Windows.h>
int main(int argc,char*argv[])
{
    printf("*****************So Let's play a Game!*****************\n");
    __asm
{
xor eax,eax
        jz __label
        __emit 0xE9
}
__label:
    printf("*****************So Let's play a Game!*****************\n");
    system("pause");
return0;
}
```

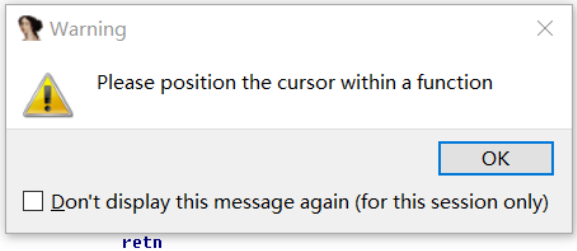
一. 代码混淆简介

- 说明
  - 几种跳转指令和对应的机器码
    - 0xE8　CALL　后面的四个字节是地址
    - 0xE9　JMP　后面的四个字节是偏移
    - 0xEB　JMP　后面的二个字节是偏移

二. 常见的花指令混淆手段

- E9后面同样需要4个字节，程序解析多字节指令时，区分不了数据和代码产生的错误

```
.text:00401010                push    ebp
.text:00401011                mov     ebp, esp
.text:00401013                sub     esp, 40h
.text:00401016                push    ebx
.text:00401017                push    esi
.text:00401018                push    edi
.text:00401019                lea     edi, [ebp-40h]
.text:0040101C                mov     ecx, 10h
.text:00401021                mov     eax, 0CCCCCCCCh
.text:00401026                rep stosd
.text:00401028                push    offset aSoLetSPlayAGam ; "*****************So Let's play a Game!*"...
.text:0040102D                call    _printf
.text:00401032                add     esp, 4
.text:00401035                xor     eax, eax
.text:00401037                jz      short near ptr loc_401039+1
.text:00401039
.text:00401039 loc_401039:                            ; CODE XREF: .text:00401037↑j
.text:00401039                jmp     near ptr 426034A6h
.text:00401039 ; ---------------------------------------------------------------------------
.text:0040103E                dw 0E800h
.text:00401040                dd 14Ch, 6804C483h
.text:00401048                dd offset aPause        ; "pause"
.text:0040104C ; ---------------------------------------------------------------------------
.text:0040104C
.text:00401051
.text:00401054
.text:00401056
.text:00401057
.text:00401058
.text:00401059
.text:0040105C
.text:0040105E
.text:00401063
.text:00401065
.text:00401066                retn
.text:00401066 ; ---------------------------------------------------------------------------
```

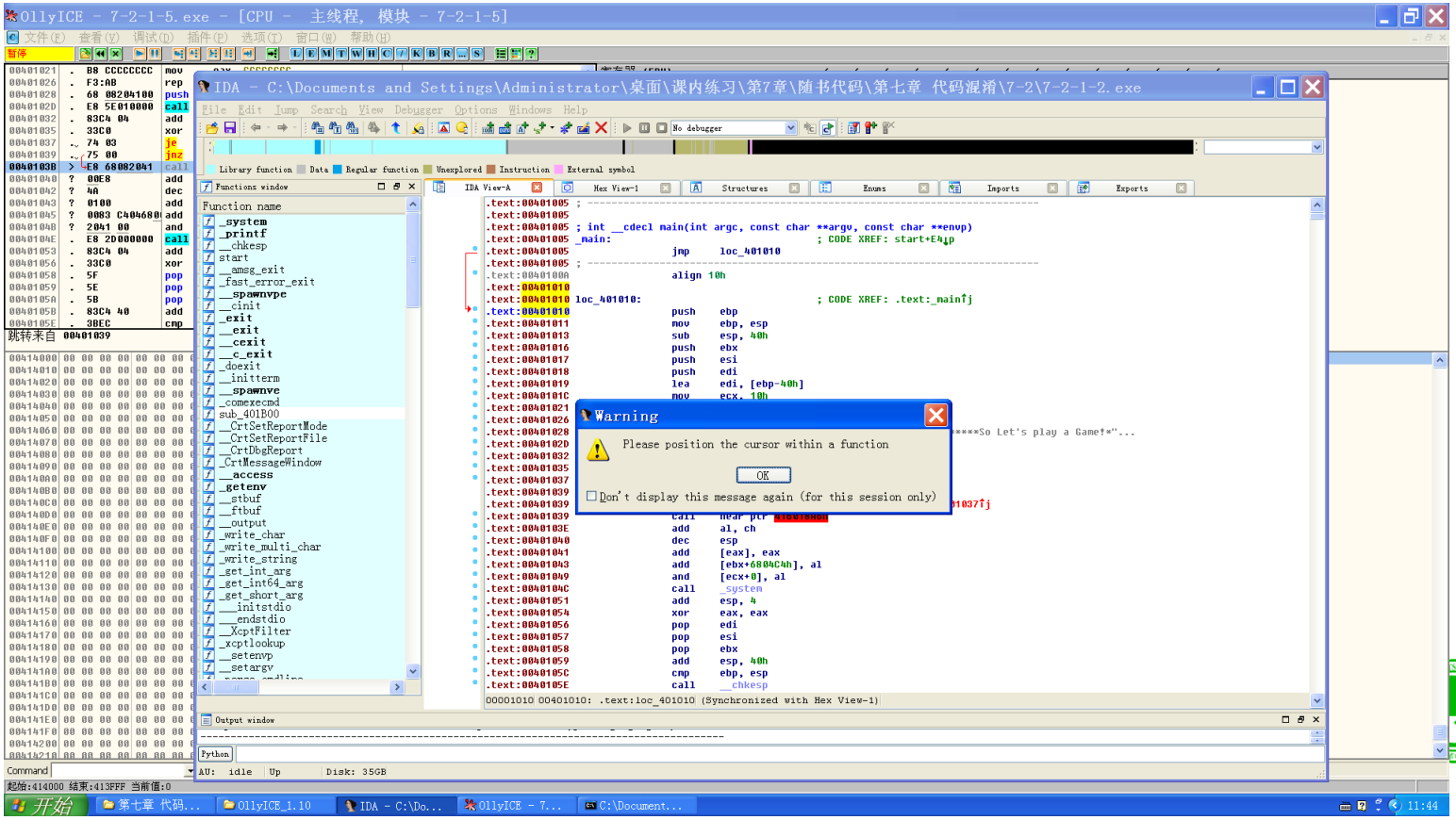


二. 常见的花指令混淆手段

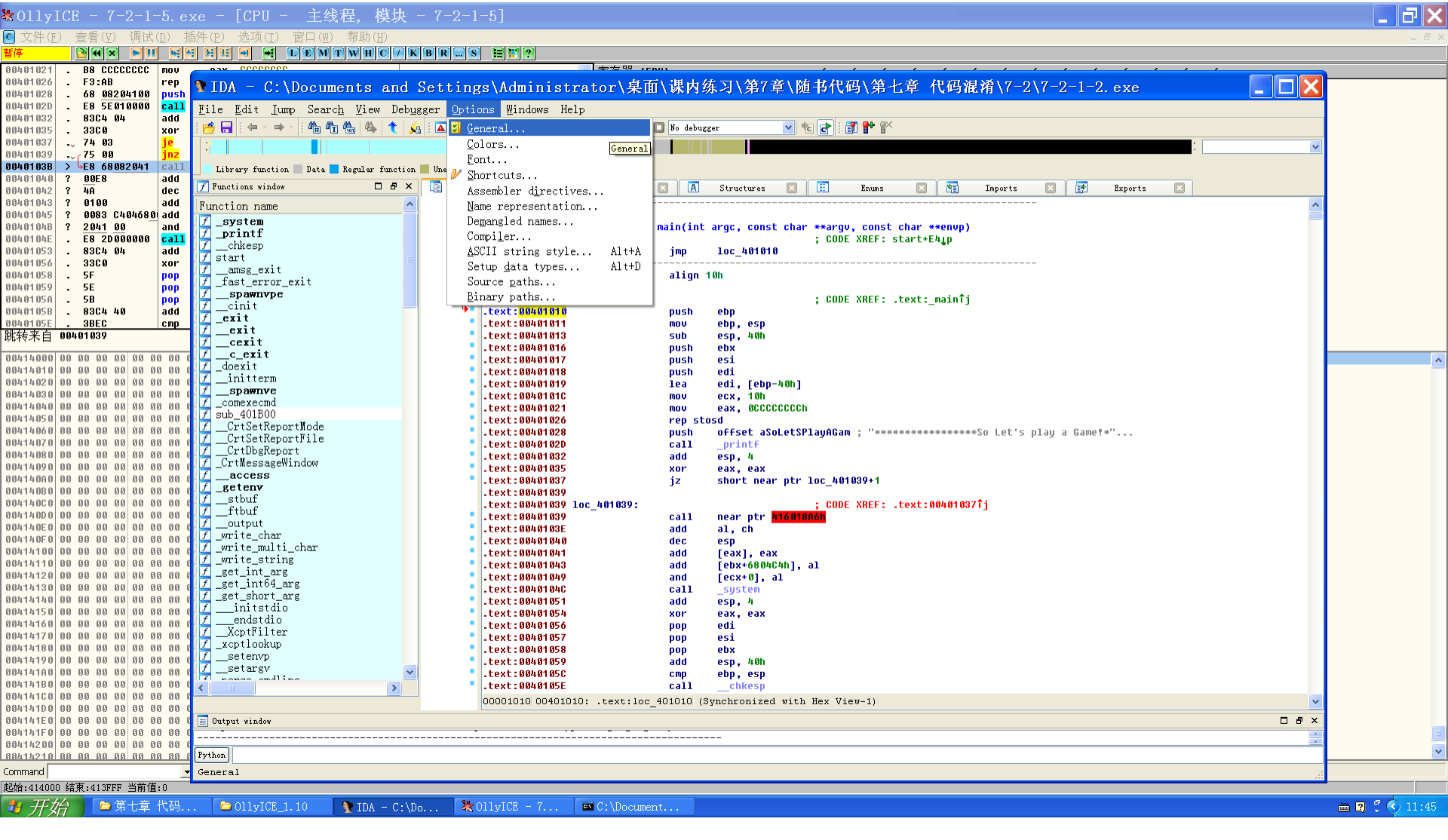
- 反混淆方法
  - 只需要根据混淆产生原理将对应的**E8,EB**和**E9**字节码**NOP**掉，就可以达到反混淆的目的。

二. 常见的花指令混淆手段

- 反混淆方法
  - 以Call指令混淆为例

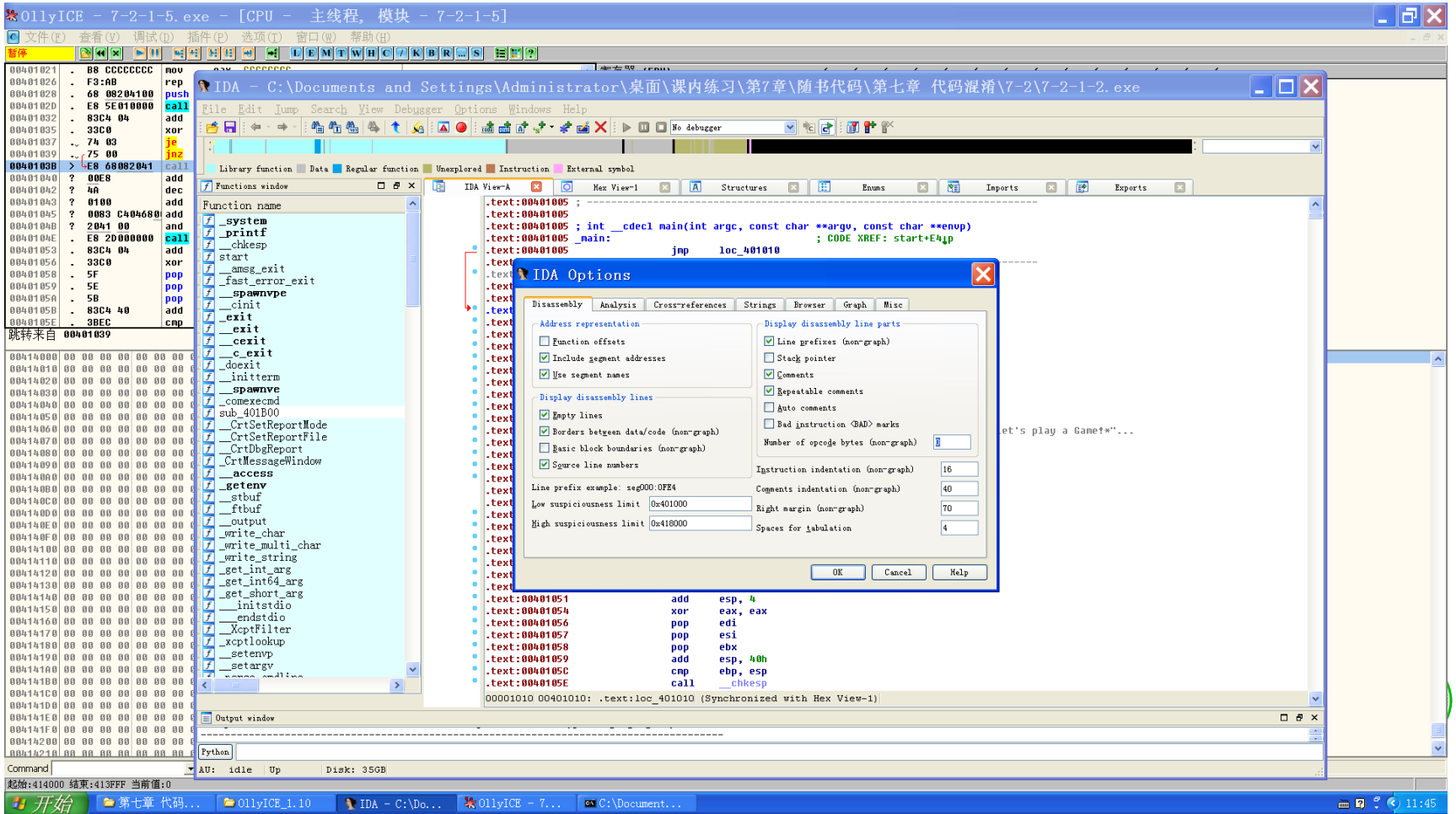


二. 常见的花指令混淆手段

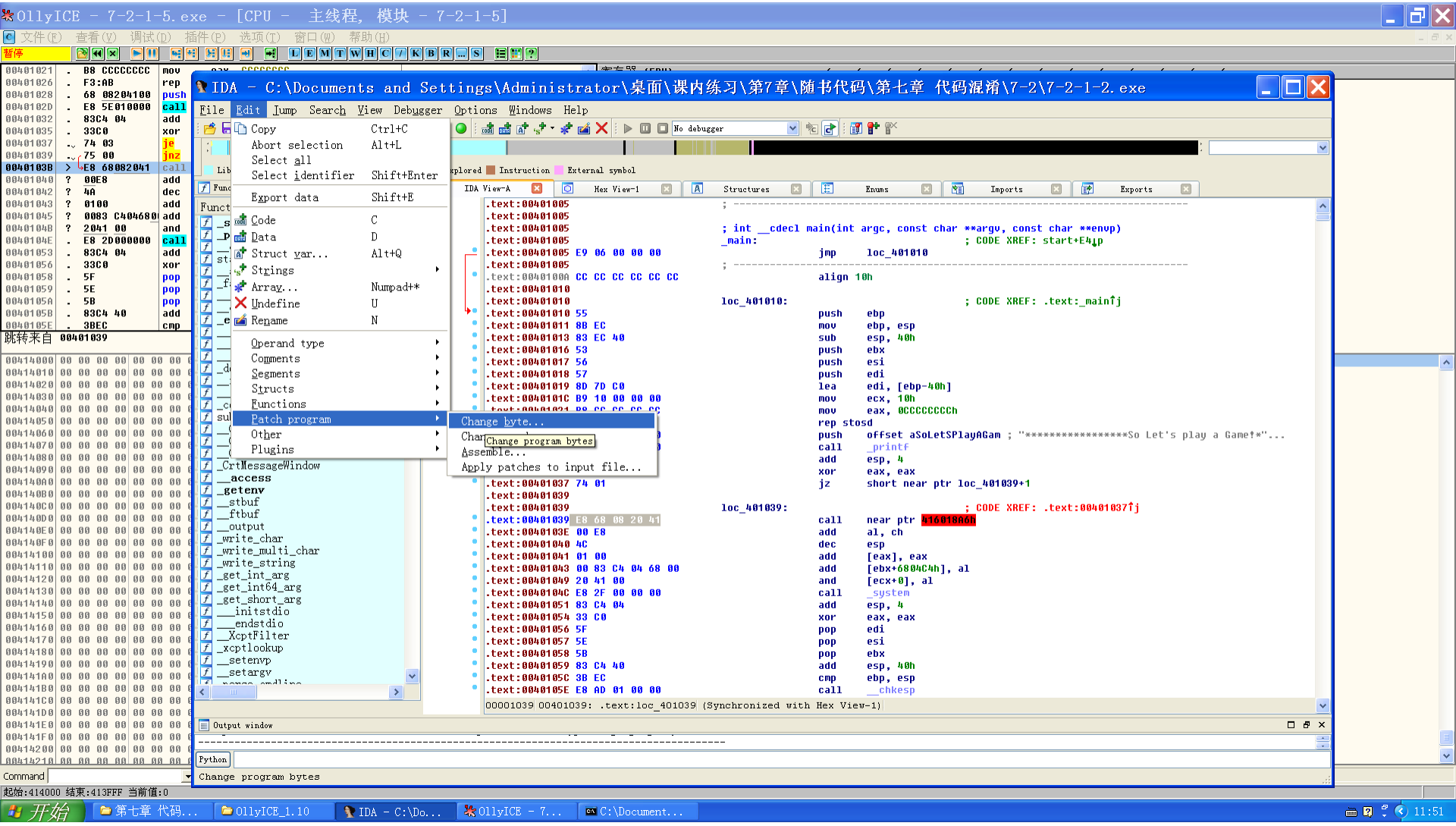


二. 常见的花指令混淆手段

- 显示的字节码数量Number of Opcode bytes :从0改为8

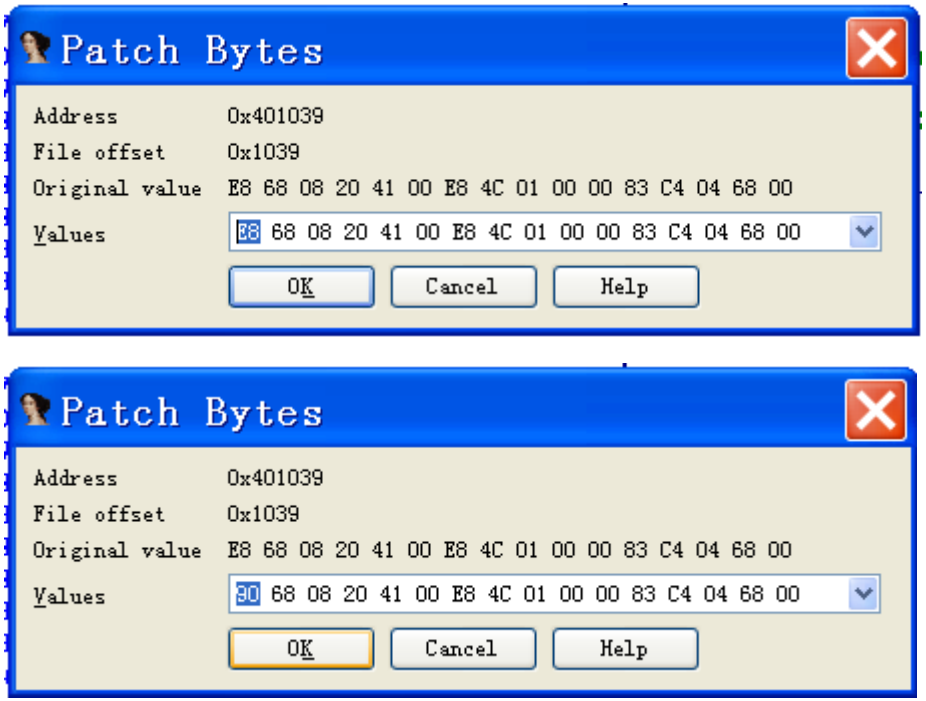


二. 常见的花指令混淆手段



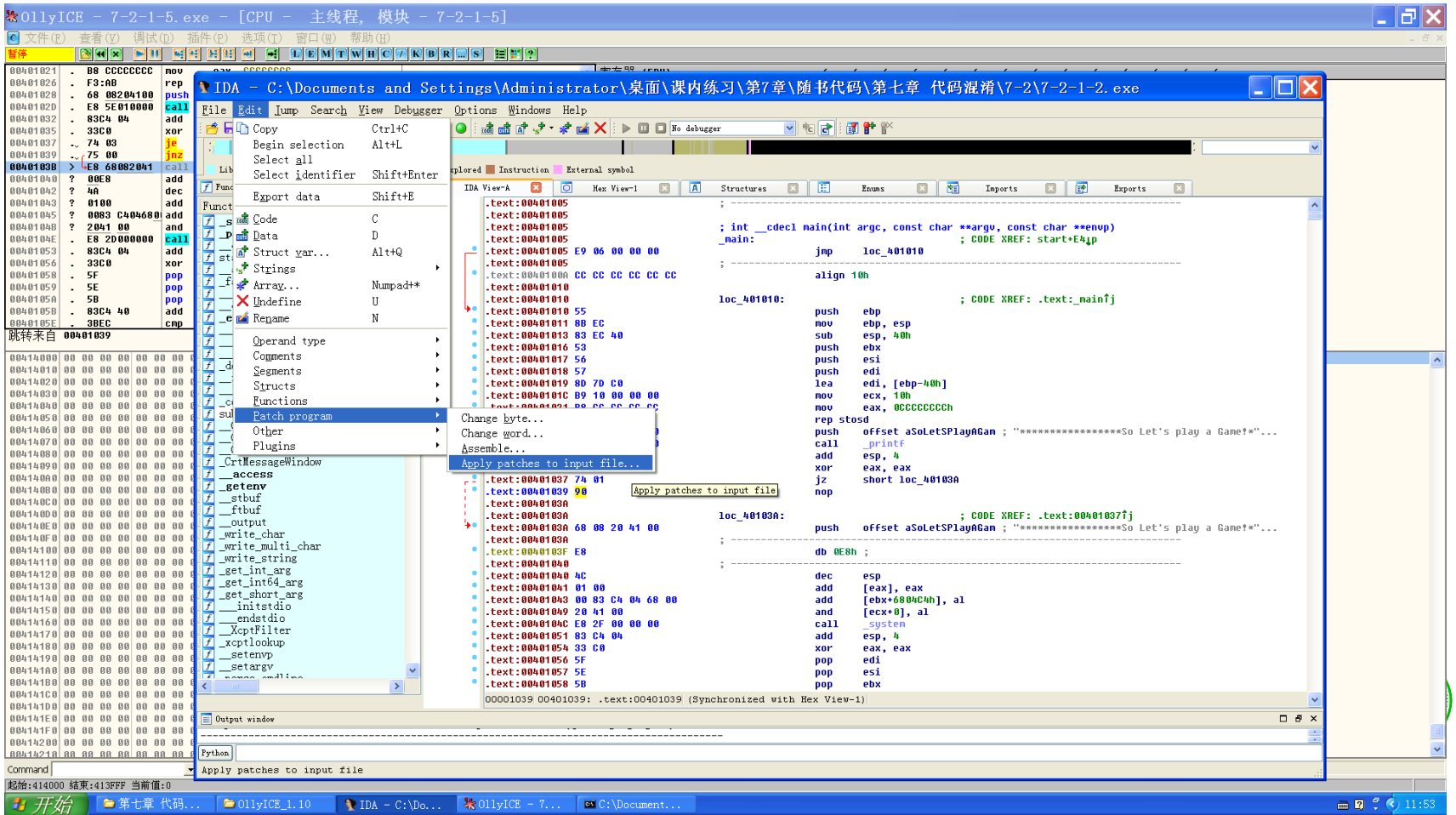
二. 常见的花指令混淆手段

- 反混淆方法
  - E8-→90



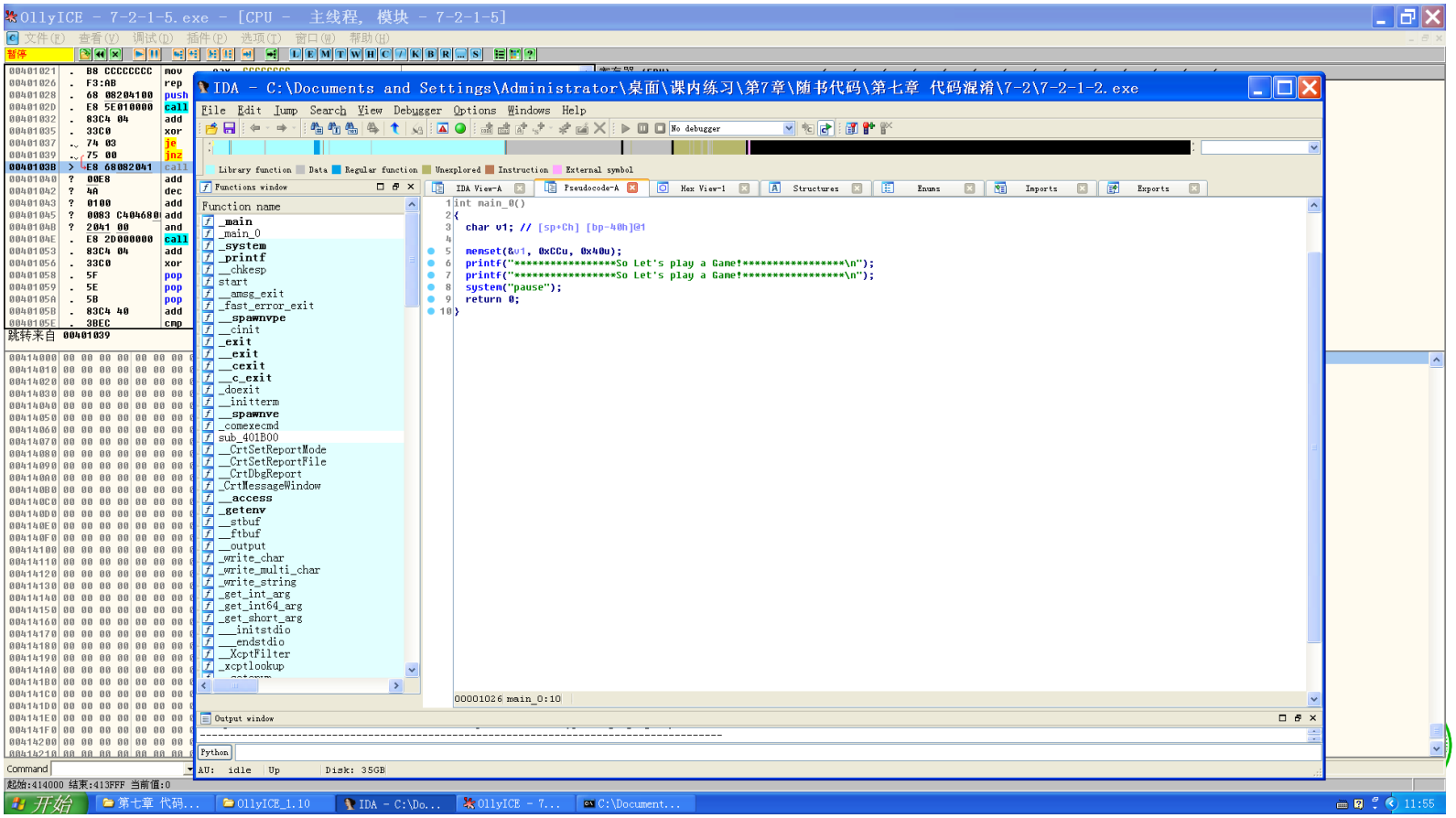
二. 常见的花指令混淆手段

- 保存一下



二. 常见的花指令混淆手段

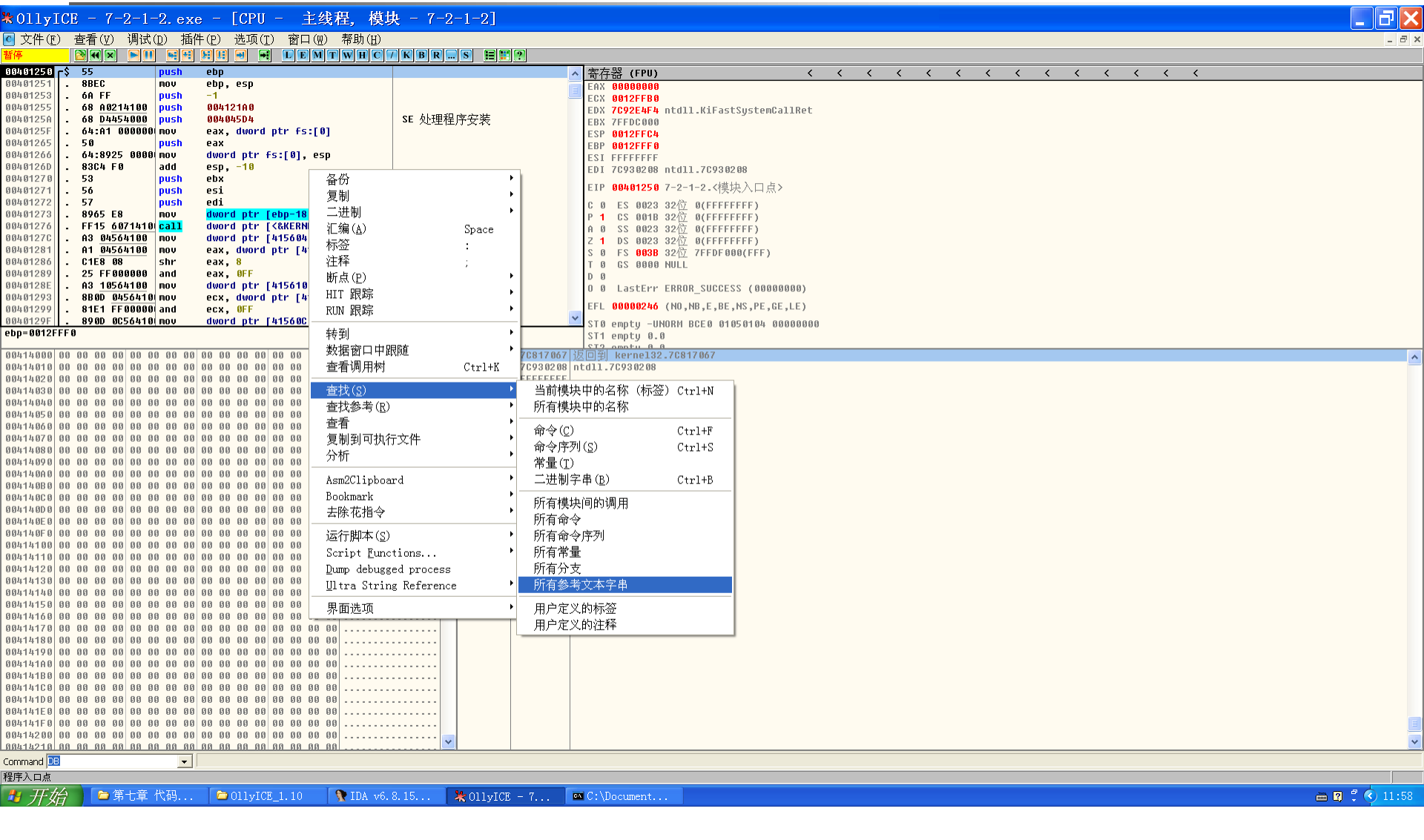
- 重新打开，可正常反编译了



二. 常见的花指令混淆手段

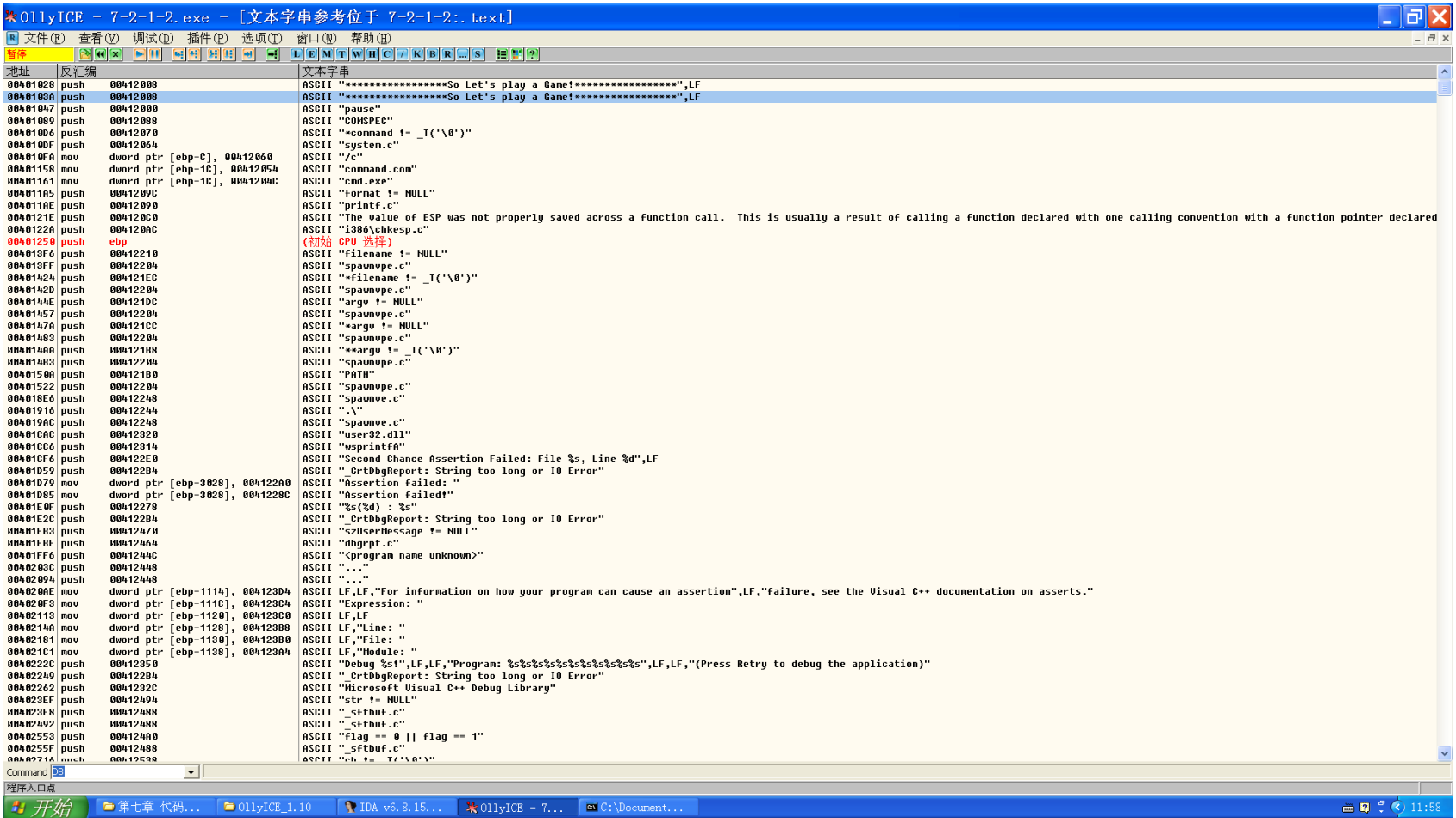
- 用OllyDbg来进行调试看看
  - OllyDbg则是使用的递归行进扫描
  - 来看看使用OllyDbg打开Call指令类混淆的二进制程序

二. 常见的花指令混淆手段



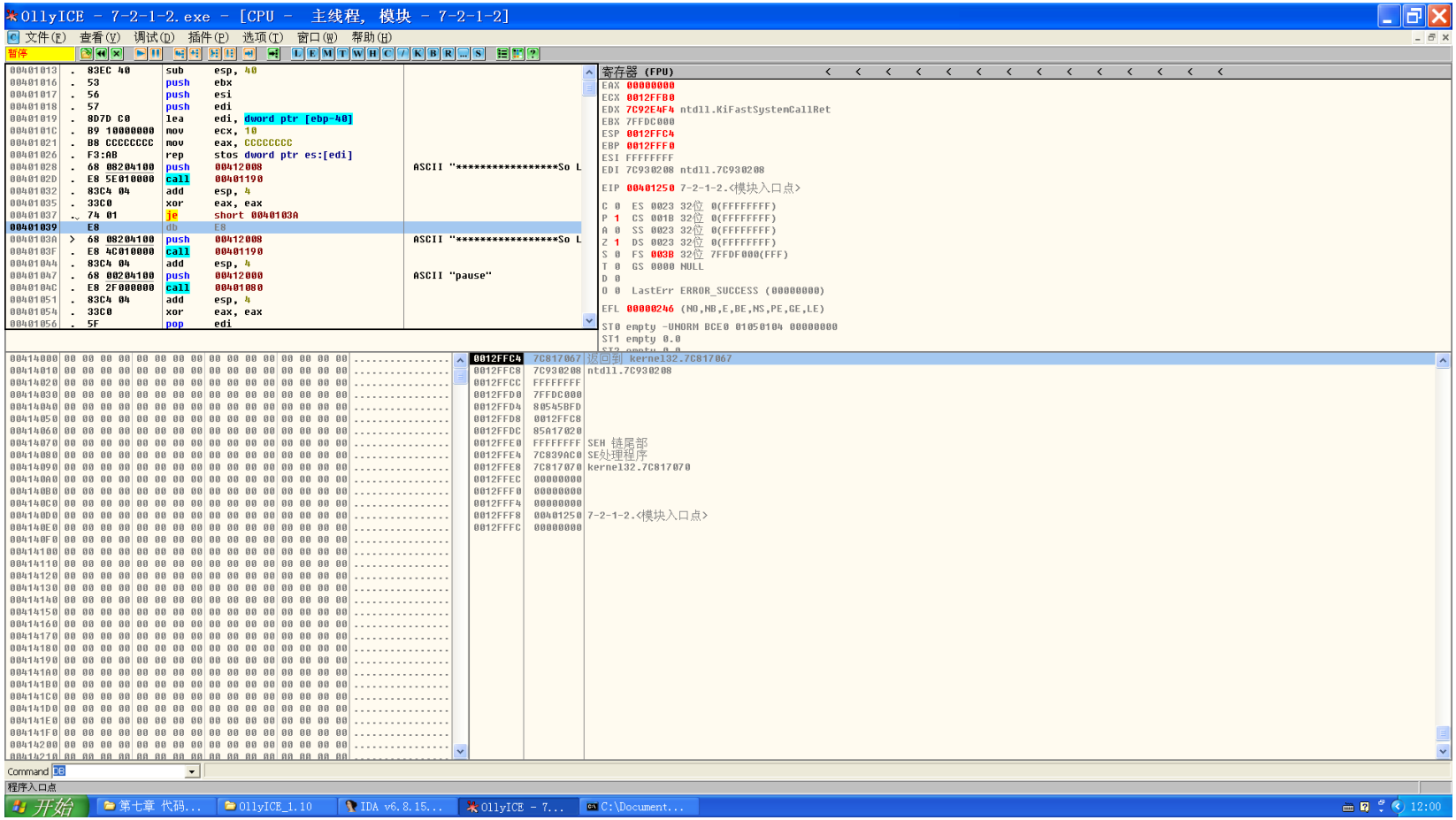
二. 常见的花指令混淆手段

- 找到main函数中的输出，双击，定位到main函数



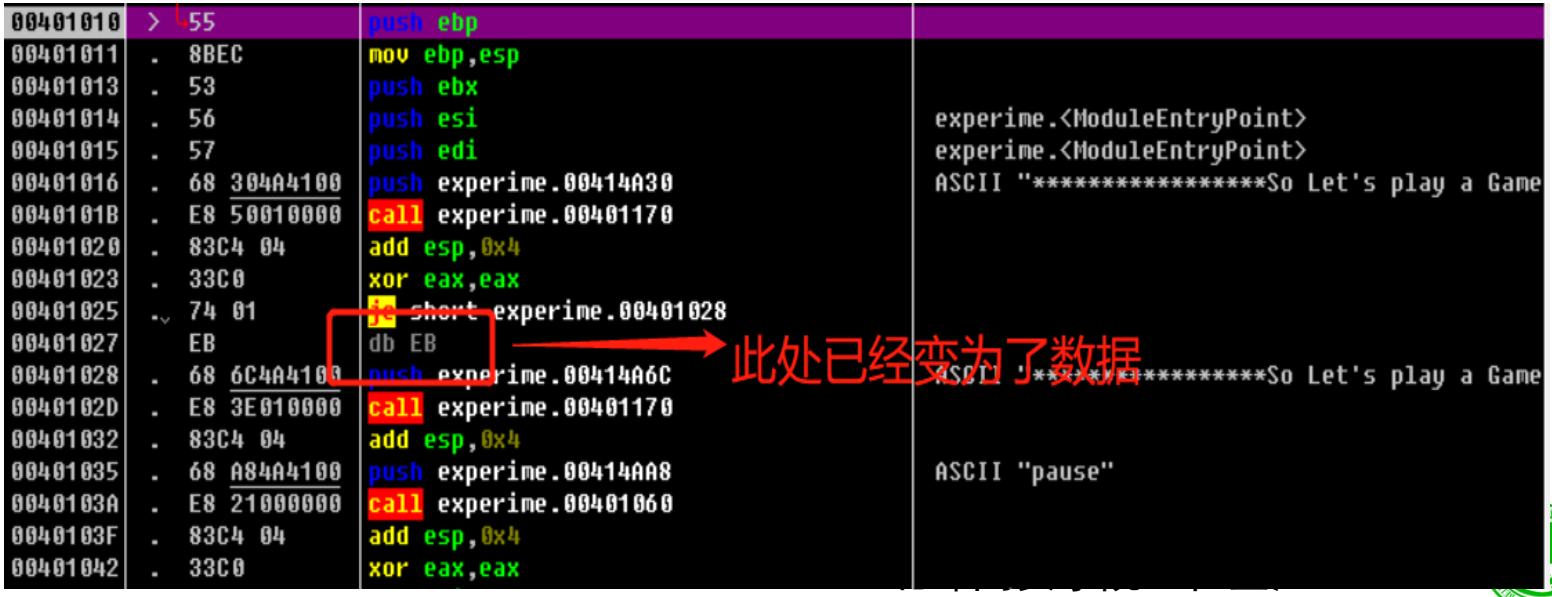
二. 常见的花指令混淆手段

- 可以看到DB E8



二. 常见的花指令混淆手段

- 用OllyDbg来进行调试看看
  - OllyDbg则是使用的递归行进扫描
  - 来看看使用OllyDbg打开Call指令类混淆的二进制程序



二. 常见的花指令混淆手段

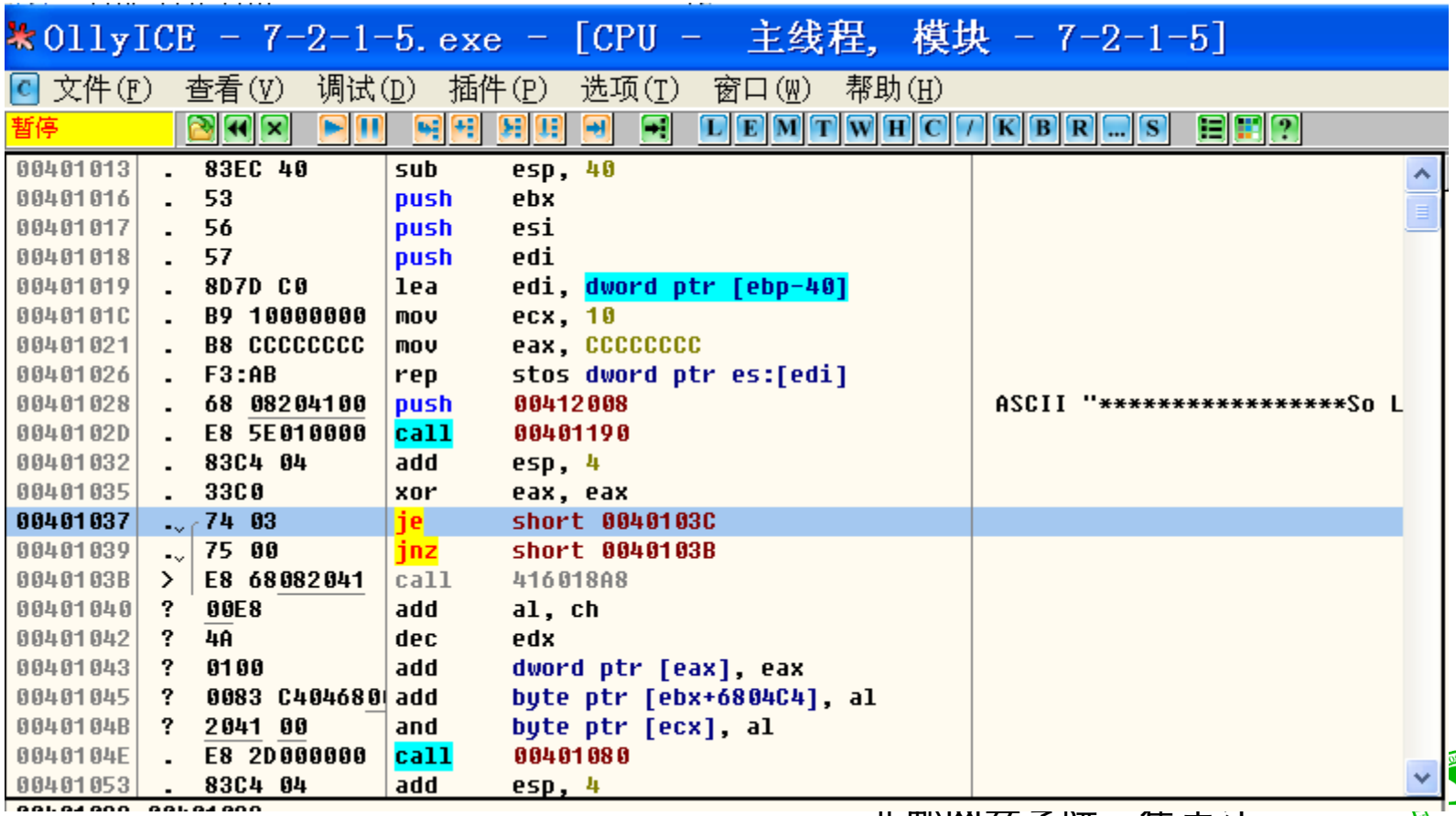
- 使用递归行进扫描的OllyDbg,成功地把花指令识别了出来，因为递归行进扫描中对于任一条控制转移指令，其转移的目的地址都要能确定
- 要迷惑这类反汇编器，只需要让其转移的地址不确定即可，创建一个指向无效数据地址的汇编语句

二. 常见的花指令混淆手段

```
#include<stdio.h>
#include<Windows.h>
int main(int argc,char*argv[])
{
    printf("*****************So Let's play a Game!*****************\n");
    __asm
{
xor eax,eax
        jz __label   //jz直接跳转到__label，不会执行下一个jnz指令
        jnz __label2   //动态实际不会执行此语句，只是在给静态分析增加迷惑性
__label2:
        __emit 0xE8
}
__label:
    printf("*****************So Let's play a Game!*****************\n");
    system("pause");
return0;
}
```

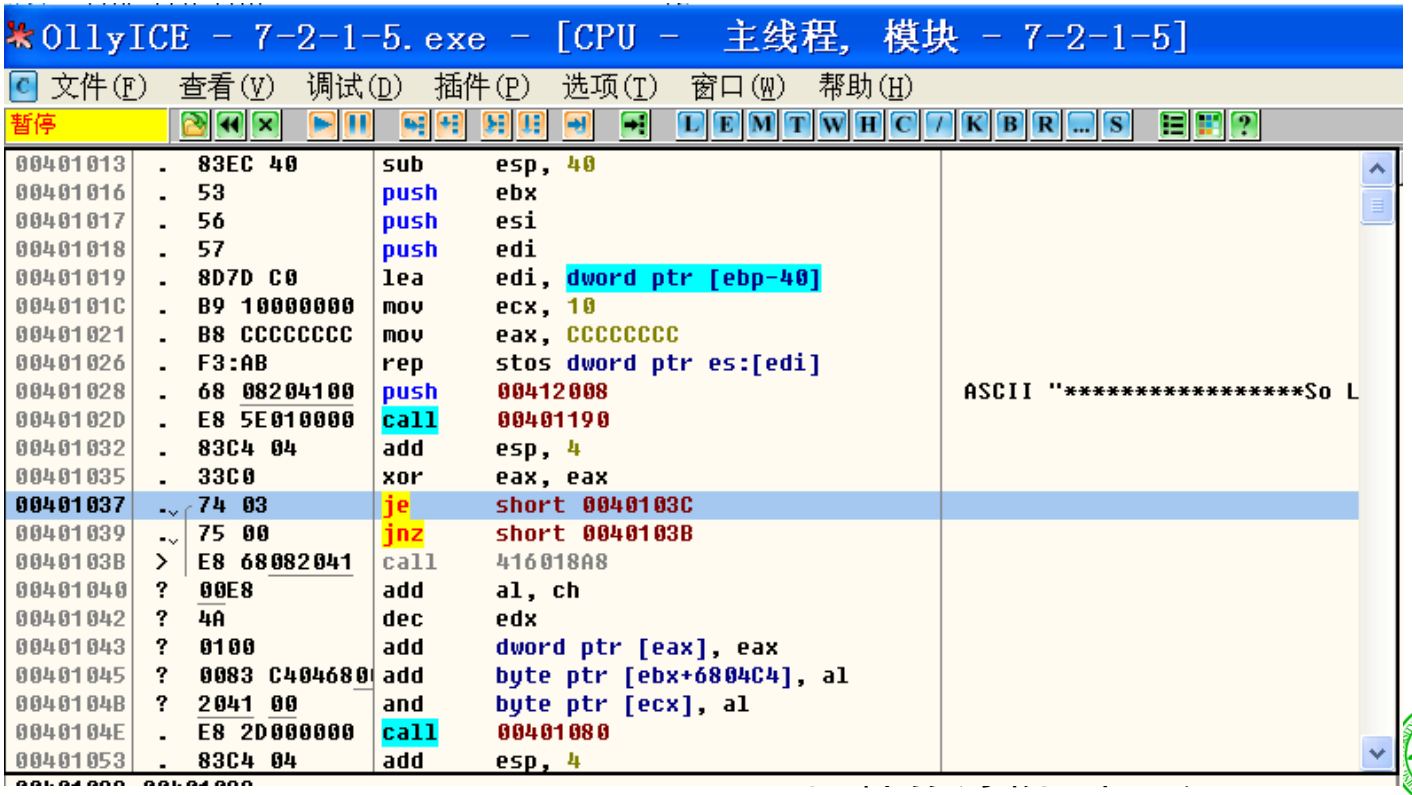
二. 常见的花指令混淆手段

- 用OD打开，查找显示的字符，定位main函数



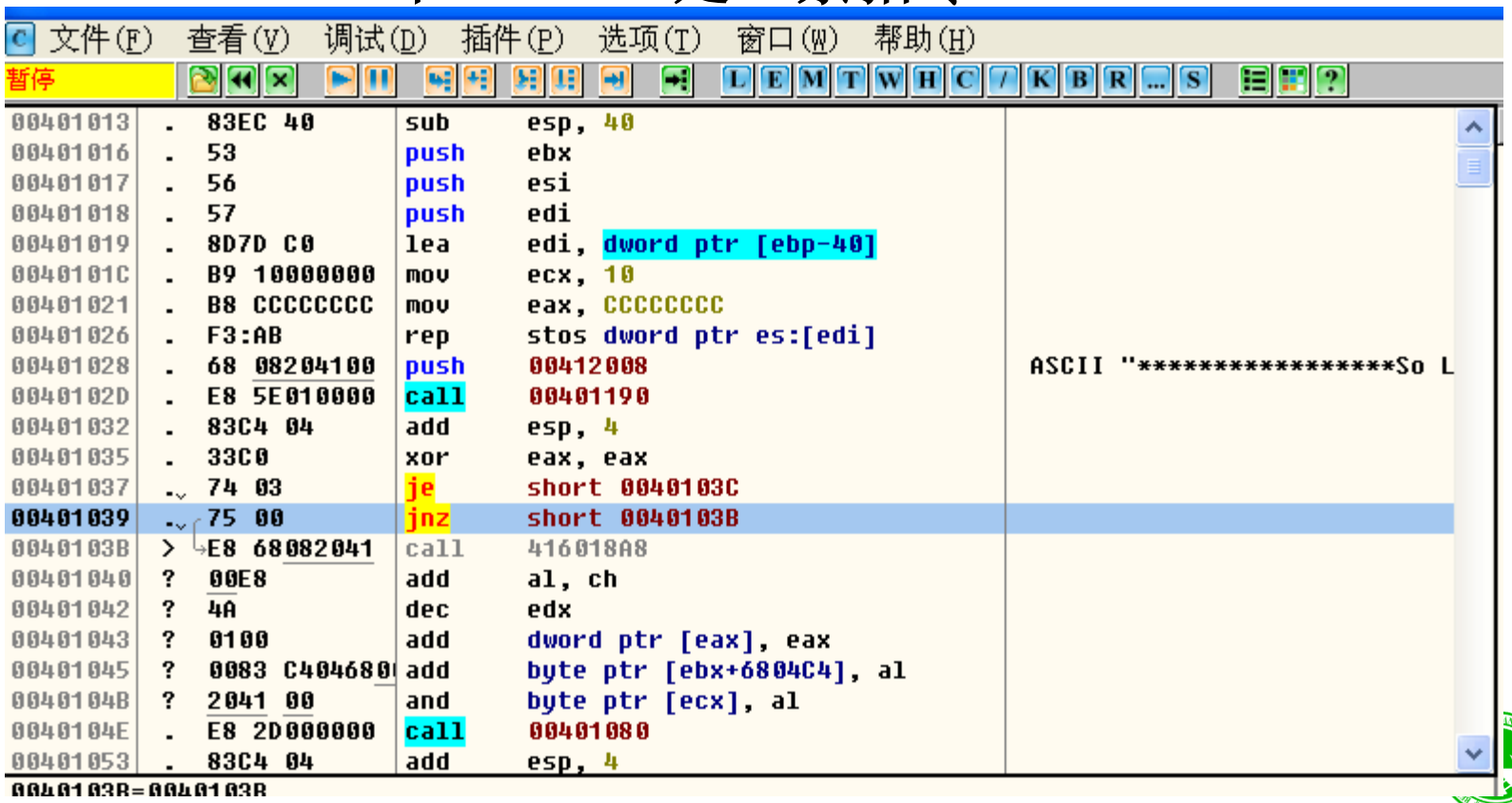
二. 常见的花指令混淆手段

- 用OD打开，查找显示的字符，定位main函数
- Je跳转到40103C



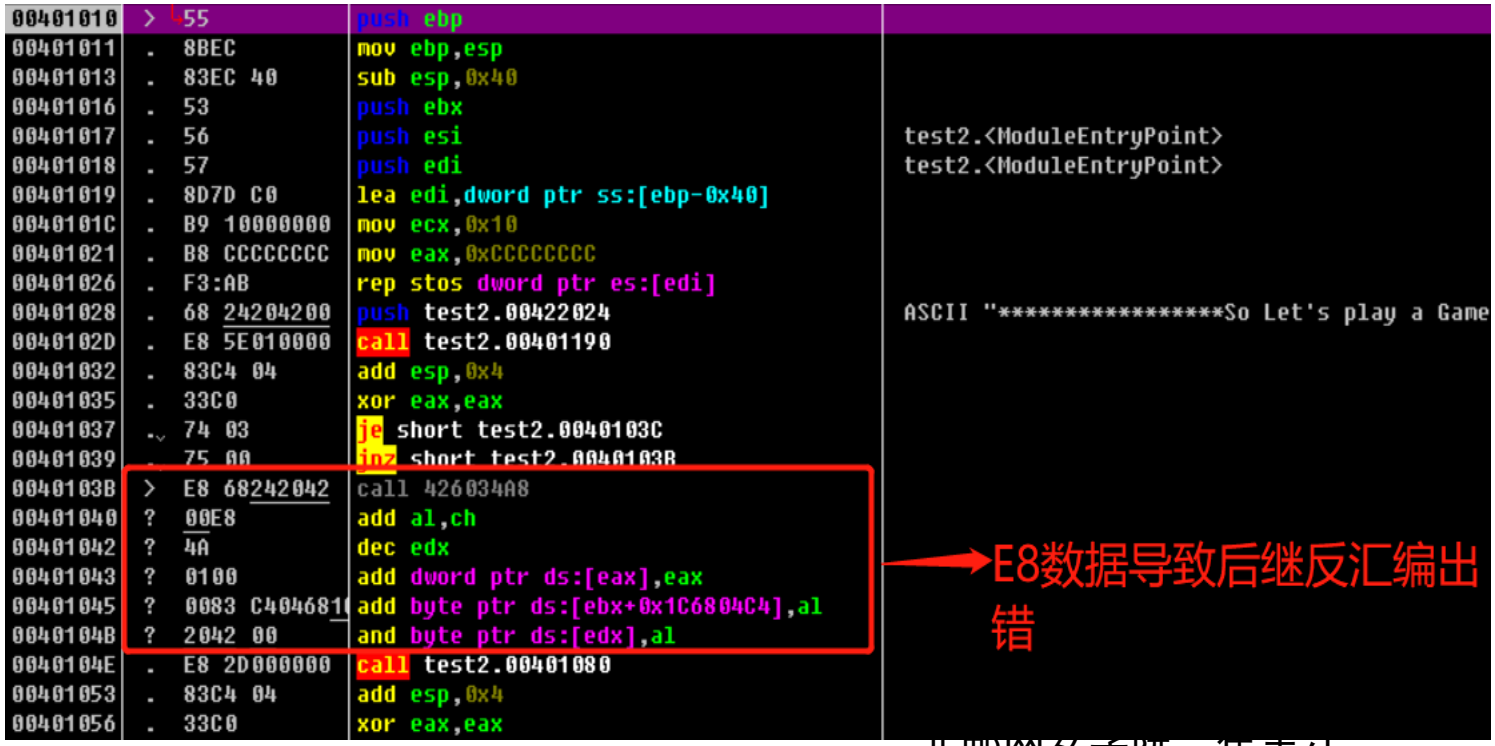
二. 常见的花指令混淆手段

- jnz跳转到40103B，这里是无效的数据E8
- 40103C和40103B是一条指令E8 68082041



二. 常见的花指令混淆手段

- 创建了一个控制转移指令jnz，让其指向了我们的无效数据0xE8



二. 常见的花指令混淆手段

- 成功蒙骗了OllyDbg，使得代码混淆成功
- 对于call这一条指令，je跳转到了0x40103C的位置，而jnz跳转到了0x40103B的位置，两处控制跳转指令均指向了同一条汇编指令，使得混淆成功

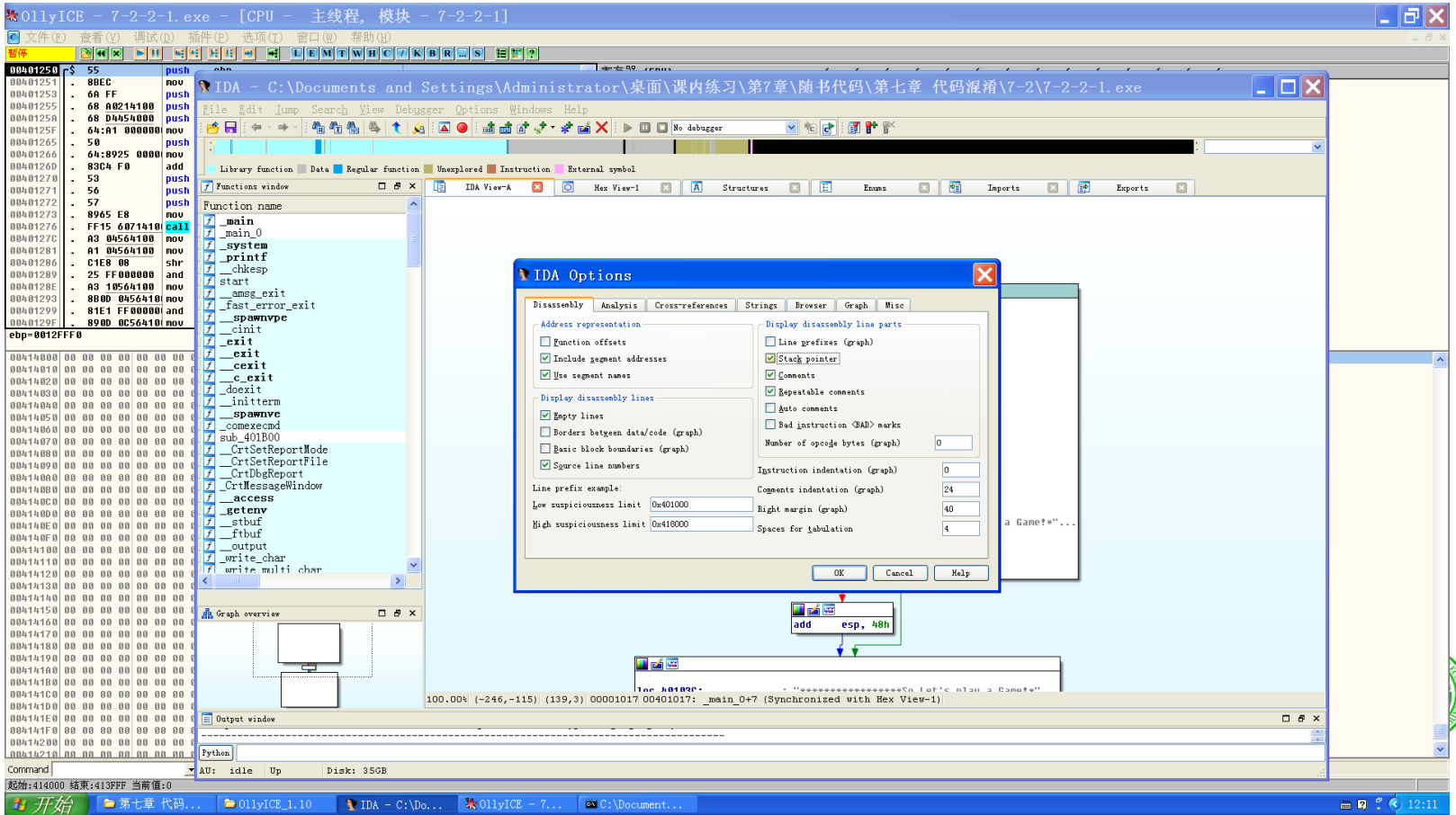
二. 常见的花指令混淆手段

❒2 更改IDA识别的栈指针，破坏栈平衡

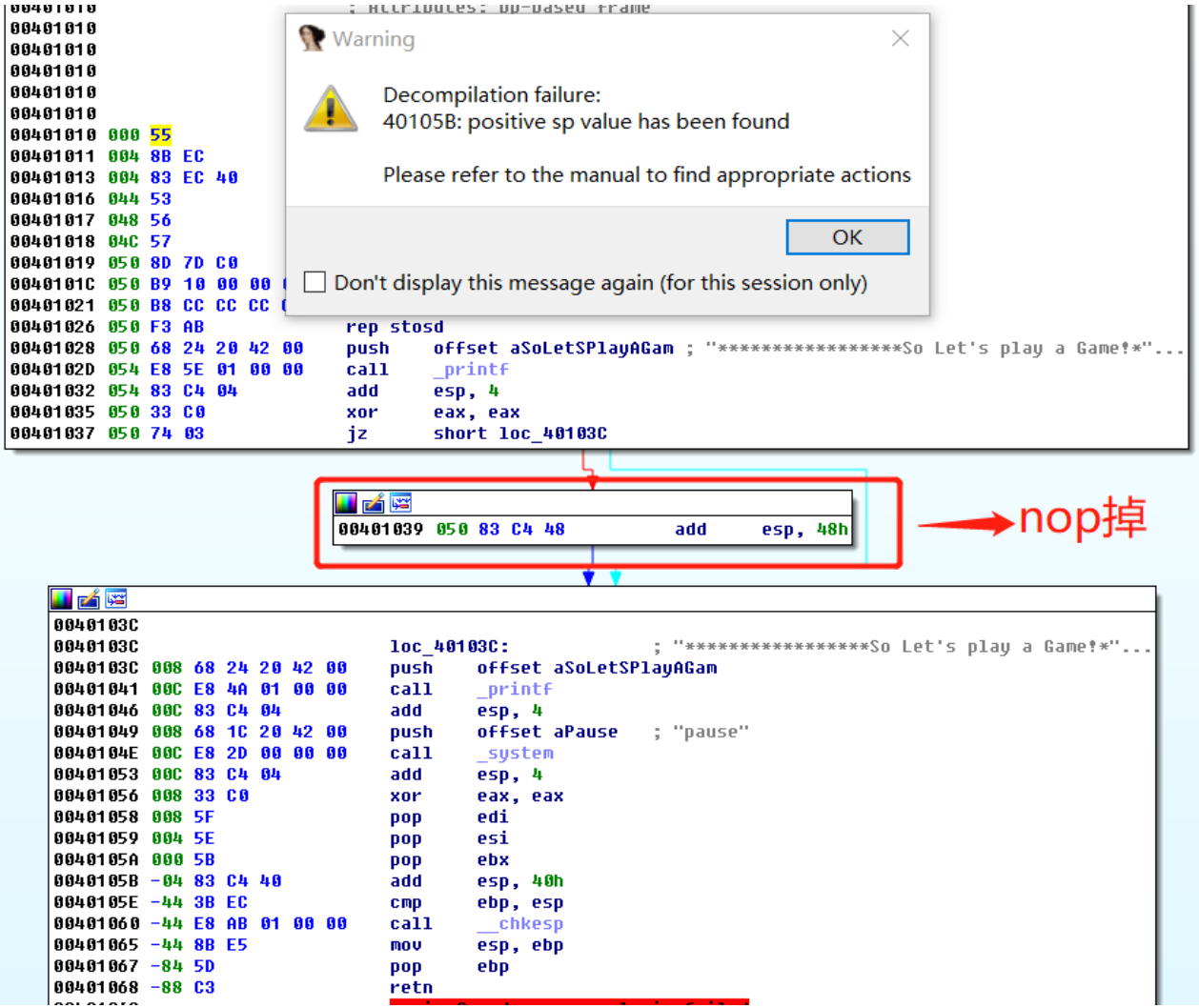
```
#include<stdio.h>
#include<Windows.h>
int main(int argc,char*argv[])
{
    printf("*****************So Let's play a Game!*****************\n");
    __asm
{
        xor eax,eax
        jz __label
        add esp,0x48
}
__label:
    printf("*****************So Let's play a Game!*****************\n");
    system("pause");
return0;
}
```

二. 常见的花指令混淆手段

- Option→general→选中stack pointer，可以查看栈帧的偏移地址

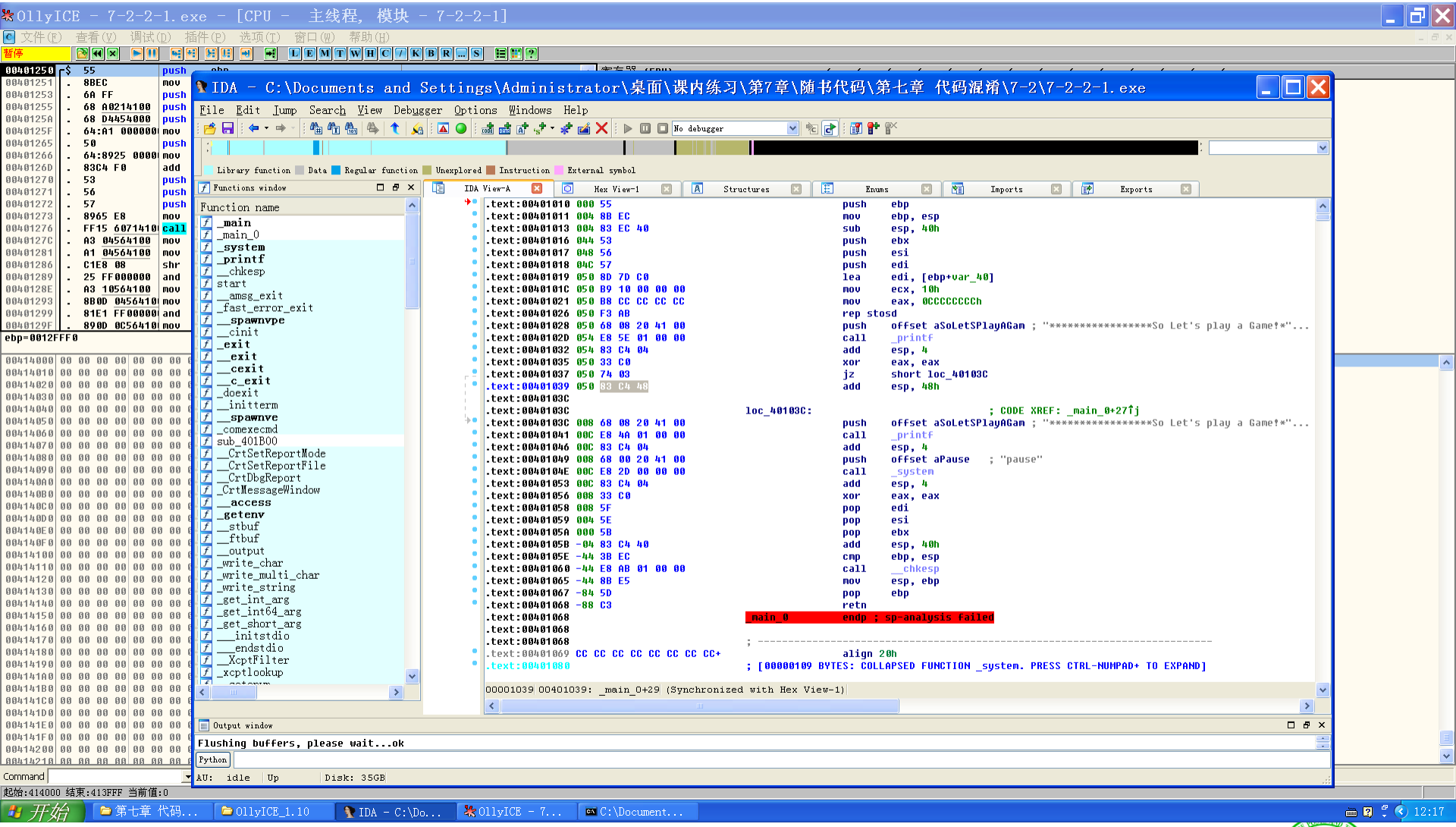


二. 常见的花指令混淆手段

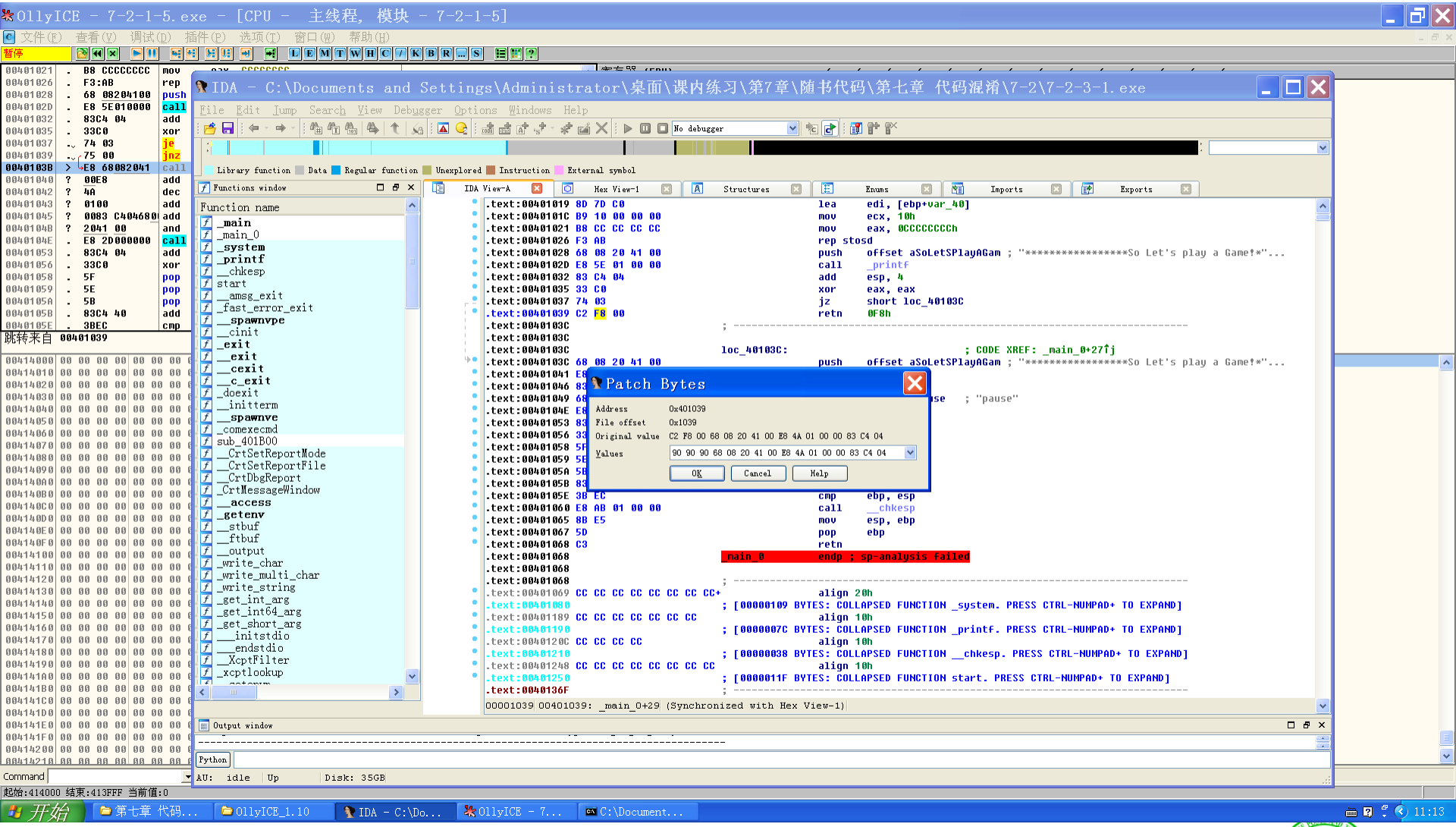


二. 常见的花指令混淆手段

- 对于这种混淆方式，**f5**以后程序显示**Decompilation failure**，原因在于**positive sp**，也就是堆栈没有平衡
- 可以看到**retn**这条汇编指令的最左边显示的绿色值为**-88h**，正常情况下子函数返回时，堆栈弹出所有参数后，堆栈平衡，偏移为**00h**。
- 当**nop**掉导致这一切的**add esp,0x48**以后，栈指针即可恢复正常


OllyICE - 7-2-2-1.exe - [CPU - 主线程, 模块 - 7-2-2-1]
IDA - C:\Documents and Settings\Administrator\桌面\课内练习\第7章\随书代码\第七章 代码混淆\7-2\7-2-2-1.exe
File Edit Jump Search View Debugger Options Windows Help
Function name
_main
_system
_printf
start
IDA View-A
Hex View-1
Structures
Enums
Imports
Exports
.text:00401010 000 55 push ebp
.text:00401011 004 8B EC mov ebp, esp
.text:00401013 004 83 EC 40 sub esp, 40h
.text:00401016 044 53 push ebx
.text:00401017 048 56 push esi
.text:00401018 04C 57 push edi
.text:00401019 050 8D 7D C0 lea edi, [ebp+var_40]
.text:0040101C 050 B9 10 00 00 00 mov ecx, 10h
.text:00401021 050 B8 CC CC CC CC mov eax, 0CCCCCCCCh
.text:00401026 050 F3 AB rep stosd
.text:00401028 050 68 08 20 41 00 push offset aSoLetSPlayAGam ; "**********So Let's play a Game!*"...
.text:0040102D 054 E8 5E 01 00 00 call _printf
.text:00401032 054 83 C4 04 add esp, 4
.text:00401035 050 33 C0 xor eax, eax
.text:00401037 050 74 03 jz short loc_40103C
.text:00401039 050 83 C4 48 add esp, 48h
.text:0040103C
.text:0040103C loc_40103C: ; CODE XREF: _main_0+27↑j
.text:0040103C 008 68 08 20 41 00 push offset aSoLetSPlayAGam ; "**********So Let's play a Game!*"...
.text:00401041 00C E8 4A 01 00 00 call _printf
.text:00401046 00C 83 C4 04 add esp, 4
.text:00401049 008 68 00 20 41 00 push offset aPause ; "pause"
.text:0040104E 00C E8 2D 00 00 00 call _system
.text:00401053 00C 83 C4 04 add esp, 4
.text:00401056 008 33 C0 xor eax, eax
.text:00401058 008 5F pop edi
.text:00401059 004 5E pop esi
.text:0040105A 000 5B pop ebx
.text:0040105B -04 83 C4 40 add esp, 40h
.text:0040105E -44 3B EC cmp ebp, esp
.text:00401060 -44 E8 AB 01 00 00 call __chkesp
.text:00401065 -44 8B E5 mov esp, ebp
.text:00401067 -84 5D pop ebp
.text:00401068 -88 C3 retn
.text:00401068 _main_0 endp ; sp-analysis failed
.text:00401069 CC CC CC CC CC CC CC CC+ align 20h
.text:00401080 ; [00000109 BYTES: COLLAPSED FUNCTION _system. PRESS CTRL-NUMPAD+ TO EXPAND]
Output window
Flushing buffers, please wait...ok
Python
AU: idle Up Disk: 35GB
Command
开始
第七章 代码...
OllyICE_1.10
IDA - C:\Do...
OllyICE - 7...
C:\Document...
12:17

北邮网安学院　崔宝江


OllyICE - 7-2-1-5.exe - [CPU - 主线程, 模块 - 7-2-1-5]
IDA - C:\Documents and Settings\Administrator\桌面\课内练习\第7章\随书代码\第七章 代码混淆\7-2\7-2-3-1.exe
Patch Bytes
Address 0x401039
File offset 0x1039
Original value C2 F8 00 68 08 20 41 00 E8 4A 01 00 00 83 C4 04
Values 90 90 90 68 08 20 41 00 E8 4A 01 00 00 83 C4 04
OK
Cancel
Help

北邮网安学院　崔宝江

二. 常见的花指令混淆手段

❒3 更改IDA识别的函数参数个数

```
#include<stdio.h>
#include<Windows.h>
int main(int argc,char*argv[])
{
    printf("*****************So Let's play a Game!*****************\n");
    __asm
{
        xor eax,eax
        jz __label
        ret 248
}
__label:
    printf("*****************So Let's play a Game!*****************\n");
    system("pause");
return0;
}
```

二. 常见的花指令混淆手段

- 反编译使用的反编译器是IDA7.0

```
int __stdcall sub_4113D0(int a1, int a2, int a3, int a4, int a5, int a6, int a7, int a8, int a9, int a10, int a11, int a12, int a13, int a14, int a15, int a16, int a17, int a18, i
{
  int v62; // eax@1
  int v63; // eax@1
  int v64; // eax@1
  int v66; // [sp-Ch] [bp-CCh]@1
  char v67; // [sp+0h] [bp-C0h]@1

  memset(&v67, 0xCCu, 0xC0u);
  v62 = printf("******************So Let's play a Game!*****************\n");
  sub_41113B(&v66 == &v66, v62);
  v63 = printf("******************So Let's play a Game!*****************\n");
  sub_41113B(&v66 == &v66, v63);
  v64 = system("pause");
  sub_41113B(&v66 == &v66, v64);
  return sub_41113B(1, 0);
}
```

共62个参数

二. 常见的花指令混淆手段

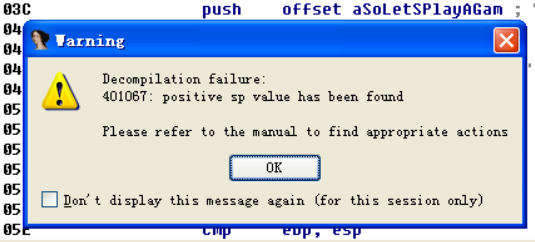
- 当程序正常反编译以后，程序的参数变成了62个，这是因为ret 248指令，其中248 = 4 * 62
- stdcall函数调用约定，子函数压栈多少个参数（字节），ret返回参数时，就返回多个参数对应的字节。在stdcall调用约定当中，这个平衡栈帧操作由被调用的子函数负责，因此，使得IDA才在反编译时，出现返回62个参数。
- 如果先要修复代码，只需要将ret 248 NOP掉即可。

二. 常见的花指令混淆手段

- 不同编译器编译出来的效果不尽相同，对于上述的例子，使用的是vc6.0进行编译，IDA6.8反汇编的提示框将会和7.2.2节中提示框提示的信息是一样
- 具体的情况还需要自行尝试，亲自实践

二. 常见的花指令混淆手段

```
.text:00401010 _main_0         proc near               ; CODE XREF: _main↑j
.text:00401010
.text:00401010 var_40          = byte ptr -40h
.text:00401010
.text:00401010                 push    ebp
.text:00401011                 mov     ebp, esp
.text:00401013                 sub     esp, 40h
.text:00401016                 push    ebx
.text:00401017                 push    esi
.text:00401018                 push    edi
.text:00401019                 lea     edi, [ebp+var_40]
.text:0040101C                 mov     ecx, 10h
.text:00401021                 mov     eax, 0CCCCCCCCh
.text:00401026                 rep stosd
.text:00401028                 push    offset aSoLetSPlayAGam ; "**********So Let's play a Game!**"...
.text:0040102D                 call    _printf
.text:00401032                 add     esp, 4
.text:00401035                 xor     eax, eax
.text:00401037                 jz      short loc_40103C
.text:00401039                 retn    0F8h
.text:0040103C ; ---------------------------------------------------------------------------
.text:0040103C
.text:0040103C loc_40103C:                             ; CODE XREF: _main_0+27↑j
.text:0040103C                 push    offset aSoLetSPlayAGam ; "**********So Let's play a Game!**"...
.text:00401041
.text:00401042
.text:00401043
.text:00401044
.text:00401045
.text:00401050
.text:00401051
.text:00401052
.text:00401053
.text:00401054
.text:00401055
.text:0040105E                 cmp     ebp, esp
00001010 00401010: _main_0 (Synchronized with Hex View-1)
```



北邮网安学院　崔宝江

二. 常见的花指令混淆手段

- 花指令绝不仅仅限于此，通过巧妙地构造数据，在静态分析下，能够让正常的程序产生奇妙的变化
- 课后，可以尝试组合这些混淆方式，应用到自己开发的程序当中，体验代码混淆带来的乐趣

第7章 代码混淆

- 一. 代码混淆简介
- 二. 花指令混淆
- 三. SMC代码自修改
- 四. OLLVM混淆

北邮网安学院　崔宝江

三. SMC代码自修改

- 1.SMC原理
- 2.逆向分析

三. SMC代码自修改

- 1.SMC原理
  - SMC(self-Modifying Code)
  - 在真正执行某一段代码时，程序会对自身的该段代码进行自修改，只有在修改后的代码才是可汇编，可执行的。
  - 在程序未对该段代码进行修改之前，在静态分析状态下，均是不可读的字节码，IDA之类的反汇编器无法识别程序的正常逻辑。

三. SMC代码自修改

- 通过先解密代码块，后执行的方式让大家来认识了解SMC

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<string.h>
#include<windows.h>
```

未加密的机器码

```
char step2[]=
"\x55\x8B\xEC\x51\x51\x53\x8B\x5D\x08\x8D\x45\xF8\x56\x57\x33\xD2"
"\xC7\x45\xF8\x98\xA4\xA9\x93\x8B\xFB\x88\x55\xFC\x8B\xF2\x2B\xF8"
"\x8D\x4D\xF8\x03\xCE\x8A\x04\x0F\x34\xCC\x3A\x01\x75\x23\x46\x83"
"\xFE\x04\x7C\xEC\x8B\x4D\x0C\x80\x34\x0A\x55\x42\x81\xFA\x5B\x01"
"\x00\x00\x7C\xF3\x8D\x43\x04\x68\x20\x30\x40\x00\x50\xFF\xD1\x59"
"\x59\x5F\x5E\x33\xC0\x5B\x8B\xE5\x5D\xC3";
```

加密的机器码

```
char step1[]=
"\x28\xF6\x91\xF6\x38\x75\xFD\x45\x1B\x08\x49\xFD\x05\x7C\x11\x08"
"\x53\xFD\x05\x7F\x1C\x08\x55\xFD\x05\x7E\x1A\x08\x5F\xFD\x05\x79"
"\x06\x08\x61\xF6\x28\x71\x4E\xB4\xFD\x49\x6C\x3E\x3C\xFE\x84\x27"
"\x01\x8B\xFE\xBD\x78\x15\x6D\x4C\x3D\x7D\x2D\x82\xAF\x24\x24\x4E"
"\xBD\x20\xBE";
```

三. SMC代码自修改

```
int main(int argc, TCHAR *argv[])
{
int length =0;
char flag[100]={0,};
    scanf("%100s", flag);
    length = strlen(flag);
int(*Check)(char*,char*);
if(length !=28)
{
        printf("Try Again......\n");
return0;
}
if(flag[length -1]==0x7d)
{
for(int i =0; i <67; i++)
{
            step1[i]= step1[i]^0x7d;  //下一步执行step1之前，解密step1
}
        Check =(int(*)(char*,char*))&step1;
        Check(flag, step2);
}
else
{
return0;
}
return0;
}
```

三. SMC代码自修改

- 上述代码是在vc6.0中编译
  - 如果使用其他编译器，比如VS，请关闭数据执行保护(DEP)，缓冲区安全监测机制(GS)，地址随机化(ASLR)这三个安全保护机制
    - 数据执行保护，能够阻止除.text段外其他段数据的可执行，在windows中，在开启了数据执行保护之后，只有使用VirtualProtect改变相应数据段的可执行属性，才能在该数据段执行我们的代码。而本程序中写的解密后执行的代码是在.data段上，所以要关闭数据执行保护

三. SMC代码自修改

- 对于GS而言，在拥有GS保护机制的函数当中，当函数返回的时候，会调用系统的GS系统检测函数检测GS保护机制是否被破坏。
  - 如果解密后的函数中存在GS保护机制，在我们每一次编译的过程中，随着代码量的增加，代码布局可能会发生变化，系统API的地址可能发生变化，这样再去call这个API地址，就有可能引发内存访问错误，造成程序的崩溃。就算关闭了地址随机化(ASLR)，也是有可能产生这种问题的。

三. SMC代码自修改

- 开启了**ASLR**，系统**API**的地址，甚至常量，全局变量，字符串等的地址都会发生变化，这对于**SMC**来说，极容易引发内存访问错误。
  - 解密完以后的代码之所以能够成功执行，在于设计者在设计时，是将正常程序需要加密的代码抠出来，对其进行加密的。如果存在**call**某处地址，就一定要保证这处地址是固定的，不会变化的。

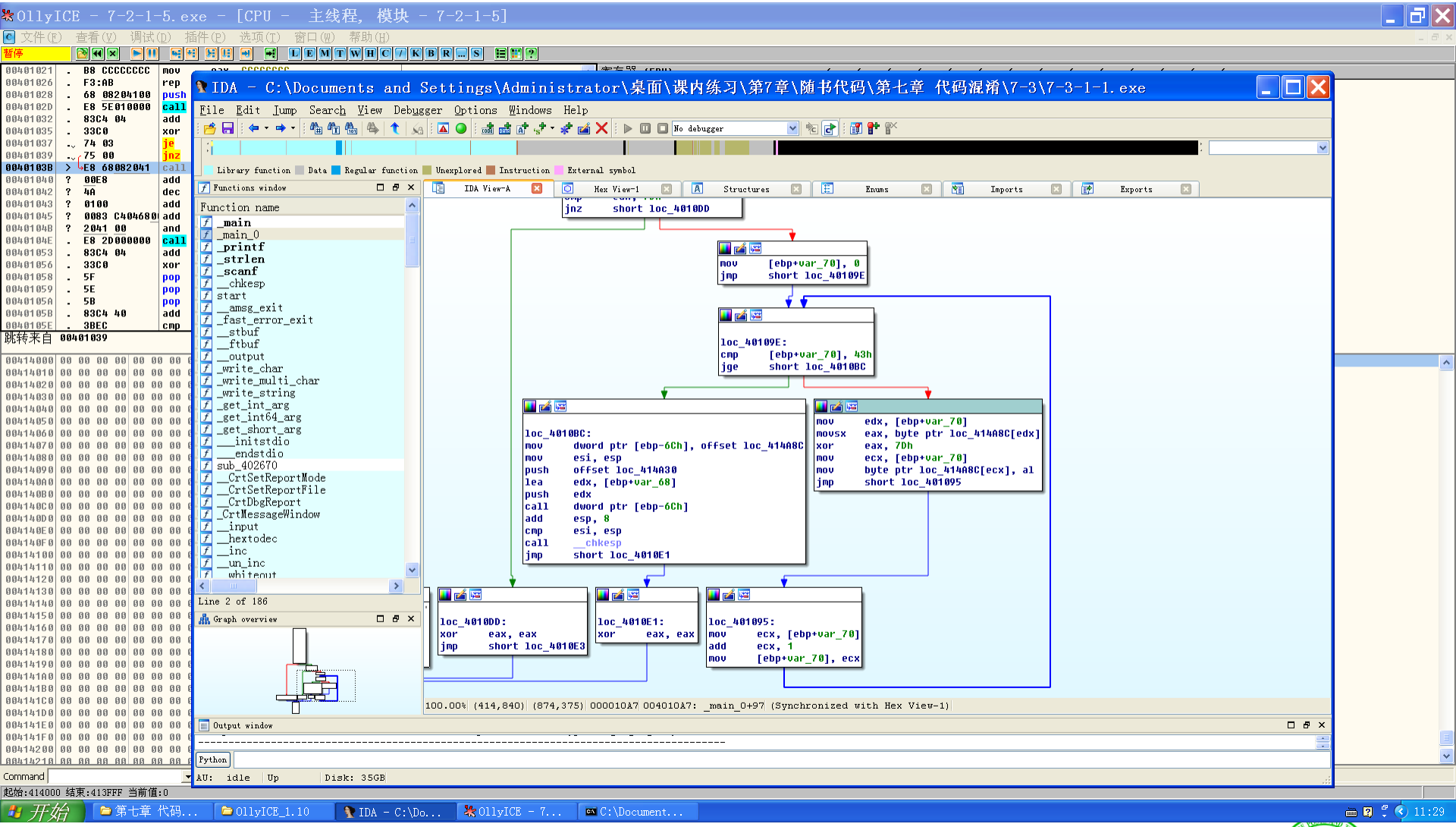
三. SMC代码自修改

- 在写类似SMC代码保护的时候，不要引入系统API，系统API的功能也最好自己使用语言去实现，避免最后生成的源程序，运行时产生不必要的错误
  - 系统API加载时偏移地址不同，字节码不同，在上面的程序中把API字节码固定了，所以要关闭ASDL。
  - 本程序避免使用API,都使用自己的程序

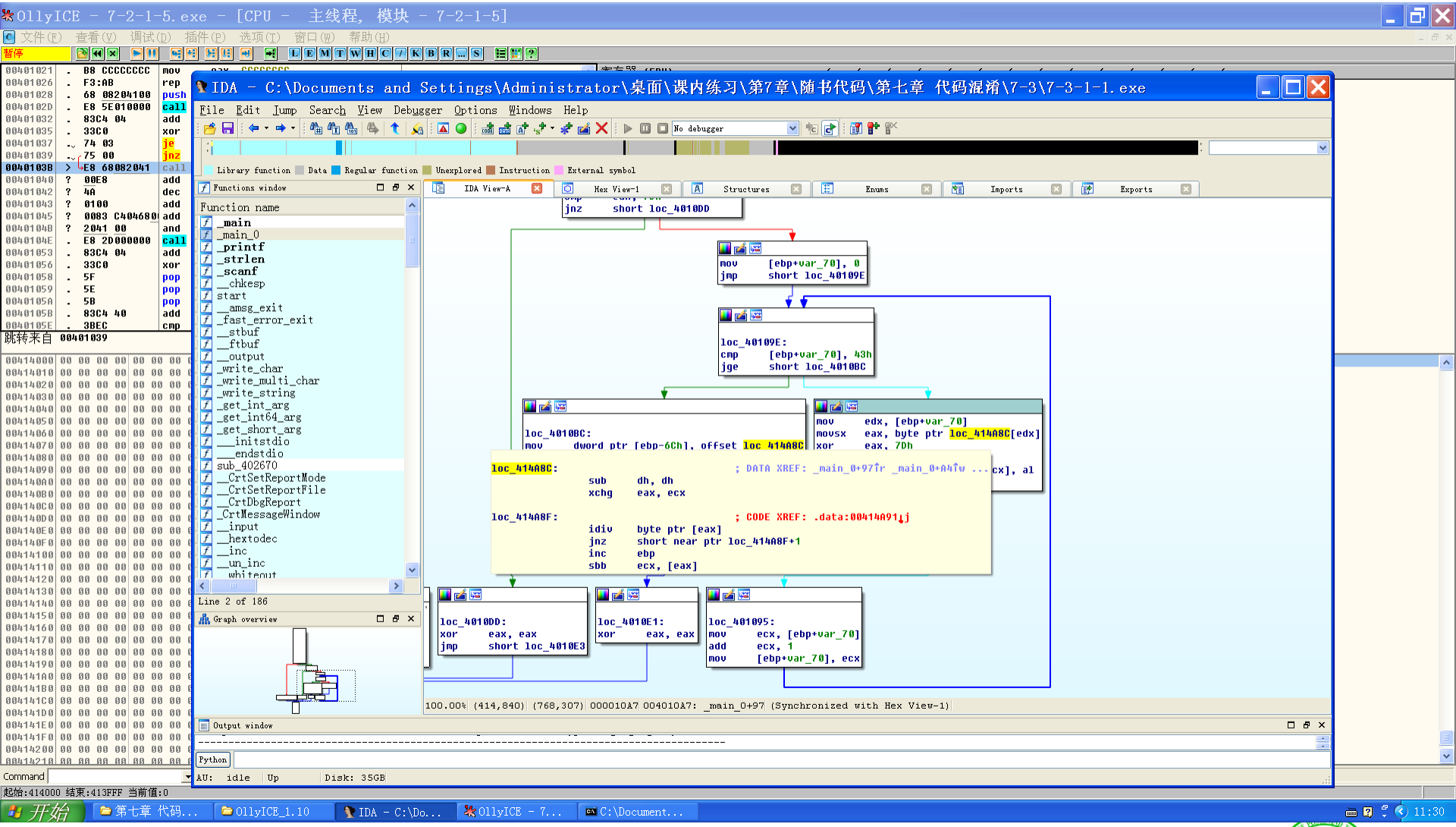
三. SMC代码自修改

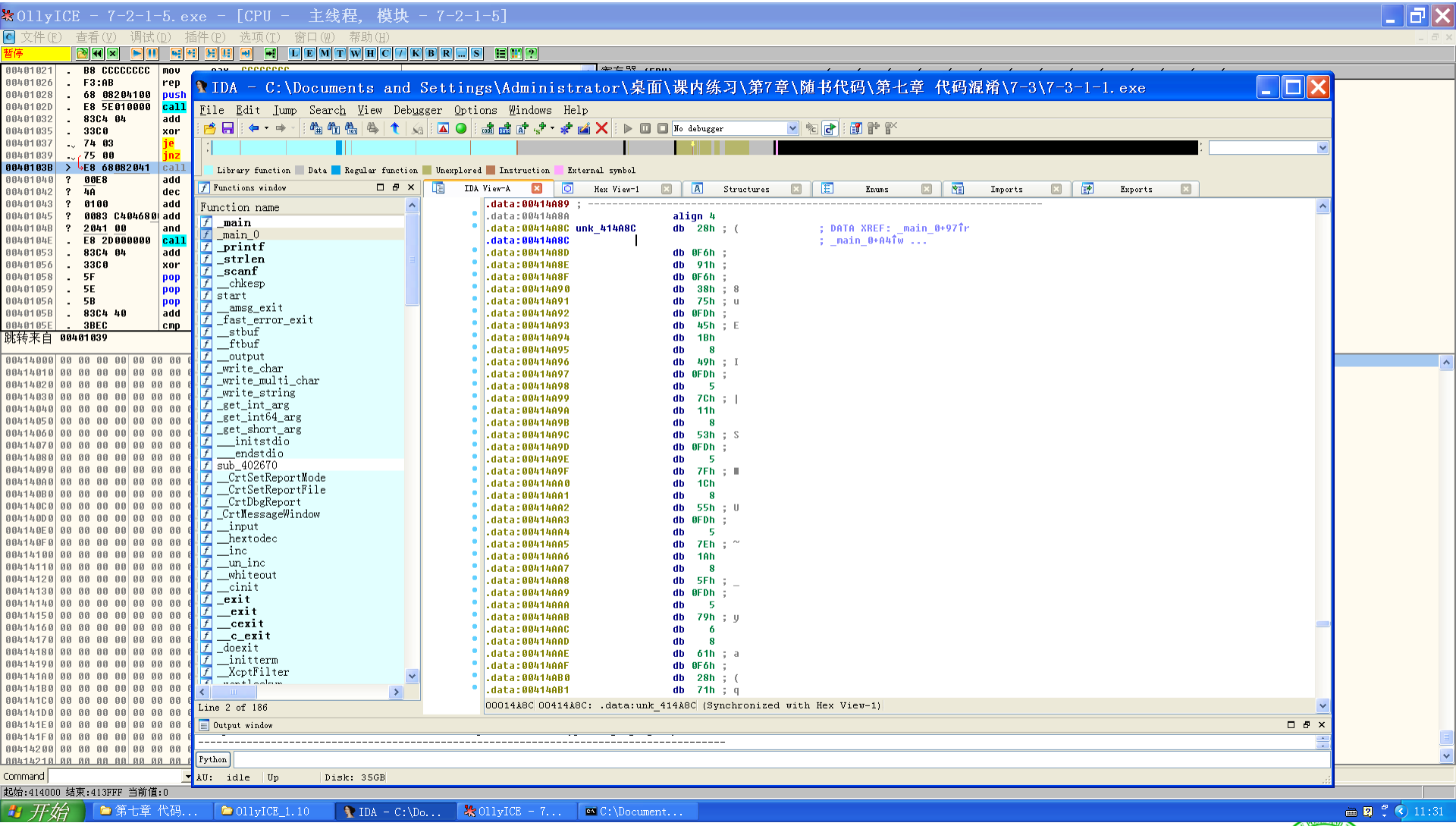
- 编译好的源程序在IDA中的展现
  - step1处加密数据

```
.data:00414A8C ; char loc_414A8C[]
.data:00414A8C loc_414A8C:                             ; DATA XREF: _main_0+97↑r
.data:00414A8C                                         ; _main_0+A4↑w ...
.data:00414A8C                 sub     dh, dh
.data:00414A8E                 xchg    eax, ecx
.data:00414A8F
.data:00414A8F loc_414A8F:                             ; CODE XREF: .data:00414A91↓j
.data:00414A8F                 idiv    byte ptr [eax]
.data:00414A91                 jnz     short near ptr loc_414A8F+1
.data:00414A93                 inc     ebp
.data:00414A94                 sbb     ecx, [eax]
.data:00414A96                 dec     ecx
.data:00414A97                 std
.data:00414A98                 add     eax, 5308117Ch
.data:00414A9D                 std
.data:00414A9E                 add     eax, 55081C7Fh
.data:00414AA3                 std
.data:00414AA4                 add     eax, 5F081A7Eh
.data:00414AA9                 std
.data:00414AAA                 add     eax, 61080679h
.data:00414AAF                 imul    byte ptr [eax]
.data:00414AB1                 jno     short near ptr off_414B00+1
.data:00414AB1 ; ---------------------------------------------------------------------------
.data:00414AB3                 db 0B4h ;
.data:00414AB4                 db 0FDh ;
.data:00414AB5                 db  49h ; I
.data:00414AB6                 db  6Ch ; l
.data:00414AB7                 db  3Eh ; >
.data:00414AB8                 db  3Ch ; <
.data:00414AB9                 db 0FEh ;
.data:00414ABA                 db  84h ;
.data:00414ABB                 db  27h ; '
.data:00414ABC                 db    1
.data:00414ABD                 db  8Bh ;
.data:00414ABE                 db 0FEh ;
.data:00414ABF                 db 0BDh ;
.data:00414AC0                 db  78h ; x
```


OllyICE - 7-2-1-5.exe - [CPU - 主线程, 模块 - 7-2-1-5]
IDA - C:\Documents and Settings\Administrator\桌面\课内练习\第7章\随书代码\第七章 代码混淆\7-3\7-3-1-1.exe
File Edit Jump Search View Debugger Options Windows Help
Function name
_main
_main_0
_printf
_strlen
_scanf
__chkesp
start
__amsg_exit
_fast_error_exit
__stbuf
__ftbuf
__output
_write_char
_write_multi_char
_write_string
_get_int_arg
_get_int64_arg
_get_short_arg
___initstdio
___endstdio
sub_402670
__CrtSetReportMode
__CrtSetReportFile
__CrtDbgReport
_CrtMessageWindow
__input
__hextodec
__inc
__un_inc
loc_4010DD:
xor eax, eax
jmp short loc_4010E3
loc_4010E1:
xor eax, eax
loc_401095:
mov ecx, [ebp+var_70]
add ecx, 1
mov [ebp+var_70], ecx
loc_4010BC:
mov dword ptr [ebp-6Ch], offset loc_414A8C
mov esi, esp
push offset loc_414A30
lea edx, [ebp+var_68]
push edx
call dword ptr [ebp-6Ch]
add esp, 8
cmp esi, esp
call __chkesp
jmp short loc_4010E1
mov edx, [ebp+var_70]
movsx eax, byte ptr loc_414A8C[edx]
xor eax, 7Dh
mov ecx, [ebp+var_70]
mov byte ptr loc_414A8C[ecx], al
jmp short loc_401095
loc_40109E:
cmp [ebp+var_70], 43h
jge short loc_4010BC
mov [ebp+var_70], 0
jmp short loc_40109E
jnz short loc_4010DD
Output window
Python
Command
AU: idle Up Disk: 35GB

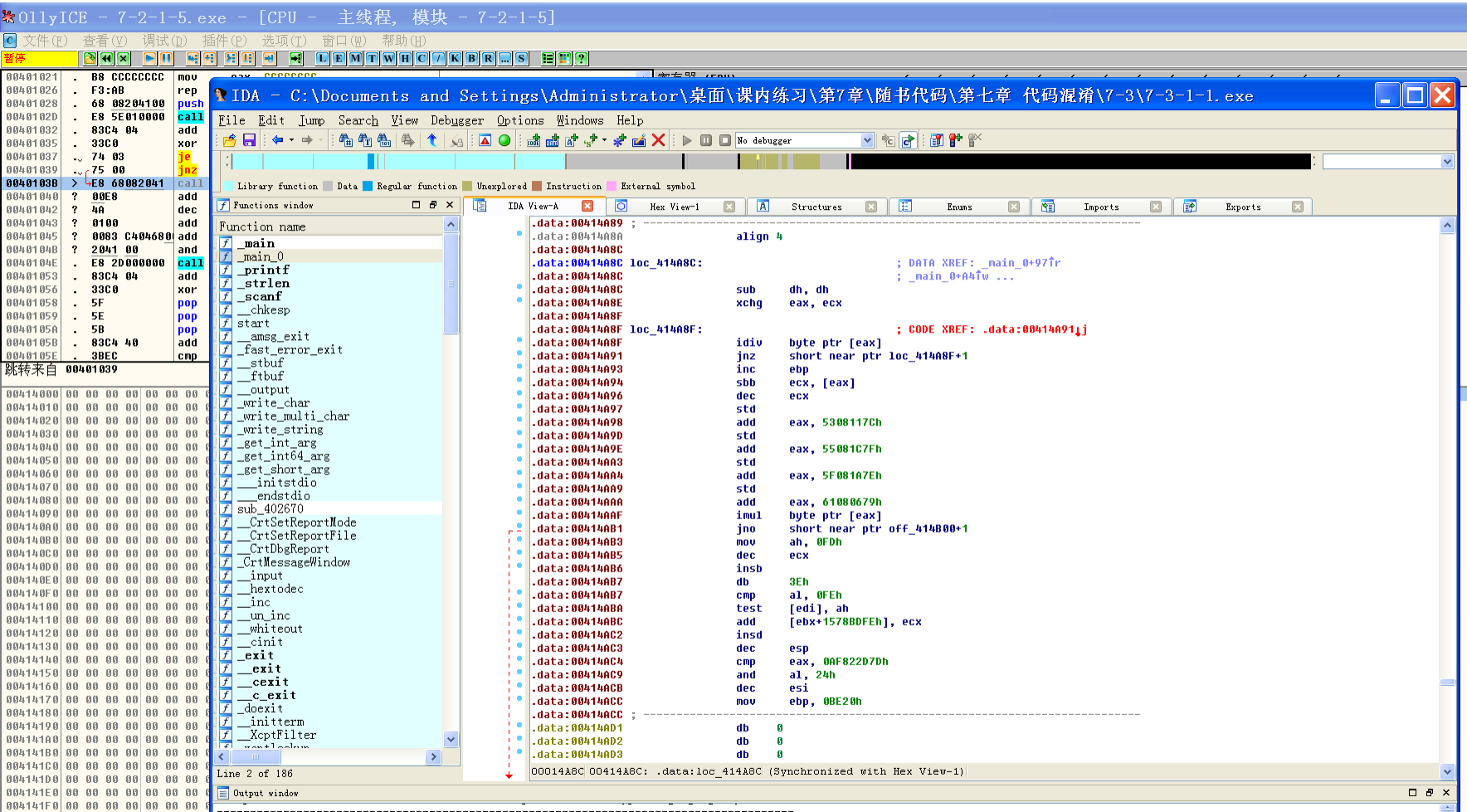
北邮网安学院　崔宝江


OllyICE - 7-2-1-5.exe - [CPU - 主线程, 模块 - 7-2-1-5]
IDA - C:\Documents and Settings\Administrator\桌面\课内练习\第7章\随书代码\第七章 代码混淆\7-3\7-3-1-1.exe
File Edit Jump Search View Debugger Options Windows Help
Functions window
Function name
_main
_main_0
_printf
_strlen
_scanf
__chkesp
start
__amsg_exit
_fast_error_exit
__stbuf
__ftbuf
__output
_write_char
_write_multi_char
_write_string
_get_int_arg
_get_int64_arg
_get_short_arg
___initstdio
___endstdio
sub_402670
__CrtSetReportMode
__CrtSetReportFile
__CrtDbgReport
_CrtMessageWindow
__input
__hextodec
__inc
__un_inc
Line 2 of 186
Graph overview
IDA View-A
Hex View-1
Structures
Enums
Imports
Exports
jnz short loc_4010DD
mov [ebp+var_70], 0
jmp short loc_40109E
loc_40109E:
cmp [ebp+var_70], 43h
jge short loc_4010BC
loc_4010BC:
mov edx, [ebp+var_70]
movsx eax, byte ptr loc_414A8C[edx]
xor eax, 7Dh
loc_414A8C: ; DATA XREF: _main_0+97↑r _main_0+A4↑w ...
sub dh, dh
xchg eax, ecx
loc_414A8F: ; CODE XREF: .data:00414A91↓j
idiv byte ptr [eax]
jnz short near ptr loc_414A8F+1
inc ebp
sbb ecx, [eax]
loc_4010DD:
xor eax, eax
jmp short loc_4010E3
loc_4010E1:
xor eax, eax
loc_401095:
mov ecx, [ebp+var_70]
add ecx, 1
mov [ebp+var_70], ecx
100.00% (414,840) (768,307) 000010A7 004010A7: _main_0+97 (Synchronized with Hex View-1)
Output window
Python
AU: idle Up Disk: 35GB
Command
起始:414000 结束:413FFF 当前值:0
开始
第七章 代码...
OllyICE_1.10
IDA - C:\Do...
OllyICE - 7...
C:\Document...
11:30


OllyICE - 7-2-1-5.exe - [CPU - 主线程, 模块 - 7-2-1-5]
IDA - C:\Documents and Settings\Administrator\桌面\课内练习\第7章\随书代码\第七章 代码混淆\7-3\7-3-1-1.exe
File Edit Jump Search View Debugger Options Windows Help
Functions window
Function name
_main
_main_0
_printf
_strlen
_scanf
__chkesp
start
__amsg_exit
_fast_error_exit
__stbuf
__ftbuf
__output
_write_char
_write_multi_char
_write_string
_get_int_arg
_get_int64_arg
_get_short_arg
___initstdio
___endstdio
sub_402670
__CrtSetReportMode
__CrtSetReportFile
__CrtDbgReport
_CrtMessageWindow
__input
__hextodec
__inc
__un_inc
__whiteout
__cinit
_exit
__exit
__cexit
__c_exit
_doexit
__initterm
__XcptFilter
IDA View-A
Hex View-1
Structures
Enums
Imports
Exports
.data:00414A89 ; ---
.data:00414A8A align 4
.data:00414A8C unk_414A8C db 28h ; (; DATA XREF: _main_0+97↑r
.data:00414A8C ; _main_0+A4↑w ...
.data:00414A8D db 0F6h ;
.data:00414A8E db 91h ;
.data:00414A8F db 0F6h ;
.data:00414A90 db 38h ; 8
.data:00414A91 db 75h ; u
.data:00414A92 db 0FDh ;
.data:00414A93 db 45h ; E
.data:00414A94 db 1Bh
.data:00414A95 db 8
.data:00414A96 db 49h ; I
.data:00414A97 db 0FDh ;
.data:00414A98 db 5
.data:00414A99 db 7Ch ; |
.data:00414A9A db 11h
.data:00414A9B db 8
.data:00414A9C db 53h ; S
.data:00414A9D db 0FDh ;
.data:00414A9E db 5
.data:00414A9F db 7Fh ;
.data:00414AA0 db 1Ch
.data:00414AA1 db 8
.data:00414AA2 db 55h ; U
.data:00414AA3 db 0FDh ;
.data:00414AA4 db 5
.data:00414AA5 db 7Eh ; ~
.data:00414AA6 db 1Ah
.data:00414AA7 db 8
.data:00414AA8 db 5Fh ; _
.data:00414AA9 db 0FDh ;
.data:00414AAA db 5
.data:00414AAB db 79h ; y
.data:00414AAC db 6
.data:00414AAD db 8
.data:00414AAE db 61h ; a
.data:00414AAF db 0F6h ;
.data:00414AB0 db 28h ; (
.data:00414AB1 db 71h ; q
Line 2 of 186
00014A8C 00414A8C: .data:unk_414A8C (Synchronized with Hex View-1)
Output window
Python
Command
AU: idle Up Disk: 35GB
起始:414000 结束:413FFF 当前值:0
开始
第七章 代码...
OllyICE_1.10
IDA - C:\Do...
OllyICE - 7...
C:\Document...
11:31

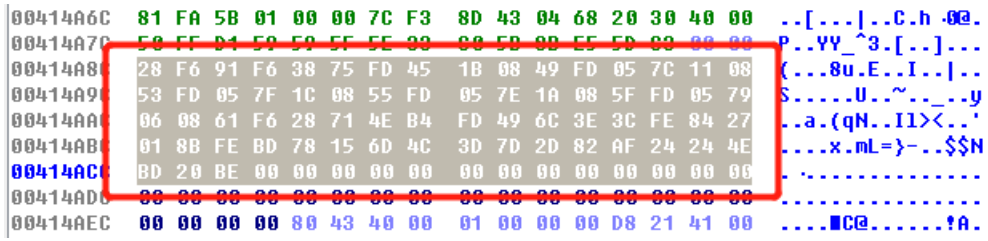
北邮网安学院　崔宝江

❍按快捷键C，将数据转成汇编，汇编代码看不懂



三. SMC代码自修改

- **step1处加密数据的十六进制**

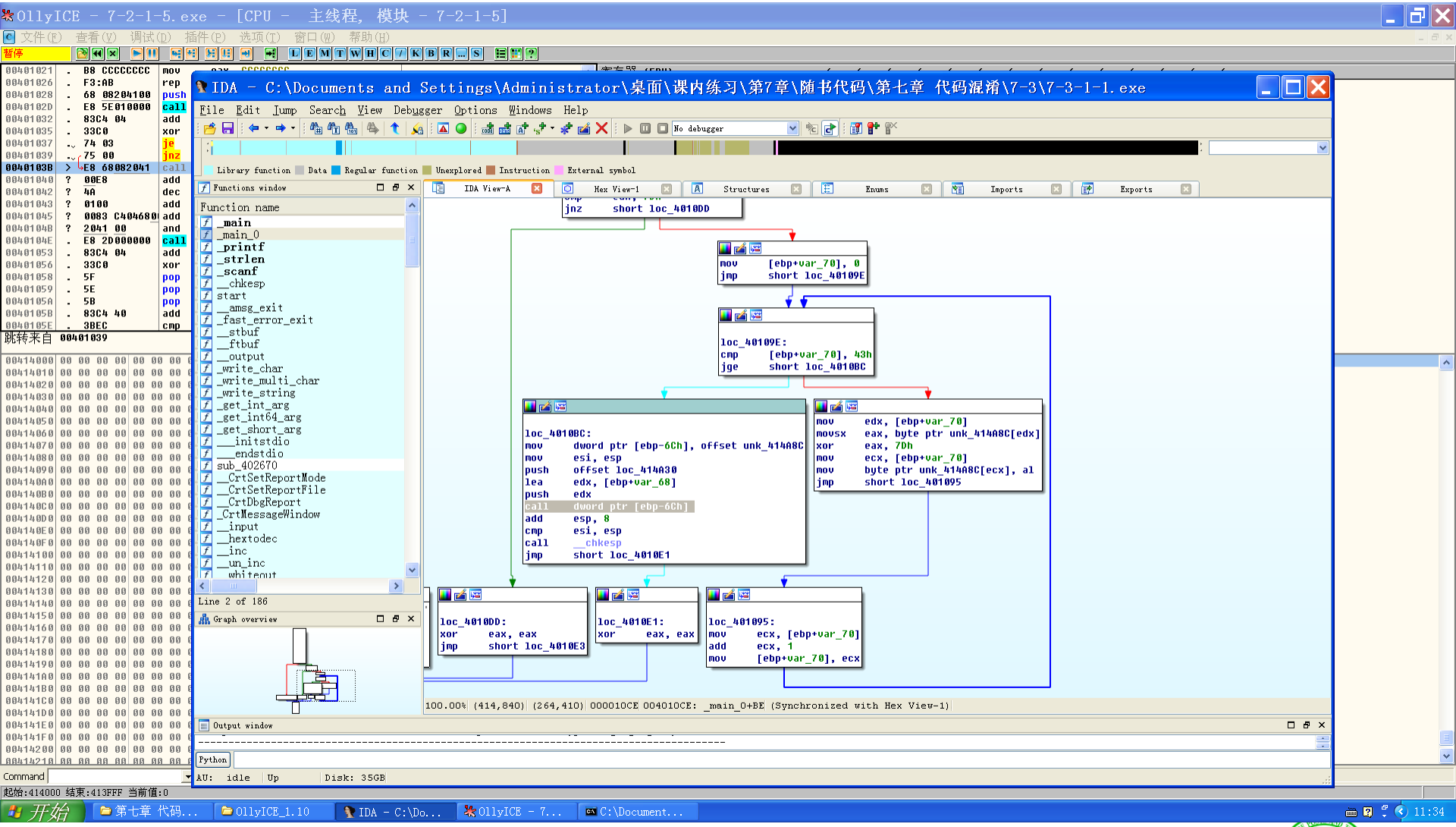


三. SMC代码自修改

- 在未解密之前，静态分析加密处的代码，是不可见的
- 程序在执行这部分代码之前，会对这部分代码进行自解密，具体可看一下解密处的汇编

三. SMC代码自修改

```
.text:00401095 loc_401095:; CODE XREF: _main_0+AAj
.text:00401095                 mov     ecx,[ebp+var_70]
.text:00401098                 add     ecx,1
.text:0040109B                 mov     [ebp+var_70], ecx
.text:0040109E
.text:0040109E loc_40109E:; CODE XREF: _main_0+83j
.text:0040109E                 cmp     [ebp+var_70],43h
.text:004010A2                 jge     short loc_4010BC
.text:004010A4                 mov     edx,[ebp+var_70]
.text:004010A7                 movsx   eax, byte ptr loc_414A8C[edx]
.text:004010AE                 xor     eax,7Dh
.text:004010B1                 mov     ecx,[ebp+var_70]
.text:004010B4                 mov     byte ptr loc_414A8C[ecx], al
.text:004010BA                 jmp     short loc_401095
.text:004010BC;---------------------------------------------------------------
.text:004010BC
.text:004010BC loc_4010BC:; CODE XREF: _main_0+92j
.text:004010BC                 mov     dword ptr [ebp-6Ch], offset loc_414A8C
.text:004010C3                 mov     esi, esp
.text:004010C5                 push    offset sub_414A30
.text:004010CA                 lea     edx,[ebp+var_68]
.text:004010CD                 push    edx
.text:004010CE                 call    dword ptr [ebp-6Ch]
```


OllyICE - 7-2-1-5.exe - [CPU - 主线程, 模块 - 7-2-1-5]
IDA - C:\Documents and Settings\Administrator\桌面\课内练习\第7章\随书代码\第七章 代码混淆\7-3\7-3-1-1.exe
File Edit Jump Search View Debugger Options Windows Help
Function name
_main
_main_0
_printf
_strlen
_scanf
loc_4010BC:
loc_4010DD:
loc_4010E1:
loc_401095:
loc_40109E:

北邮网安学院　崔宝江

三. SMC代码自修改

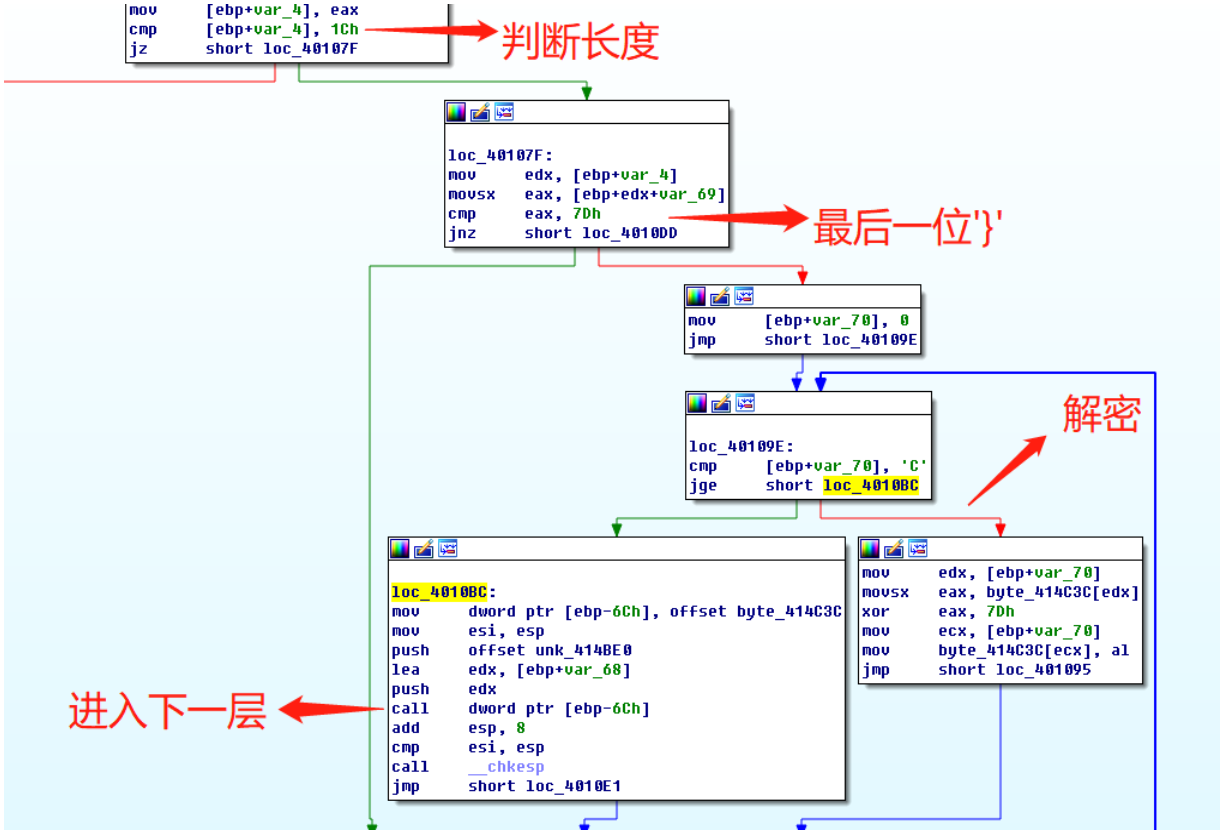
- 最后的Call dword ptr [ebp-6Ch]即会跳转到解密后的代码执行
- 知道了这个原理，在混淆代码时，可灵活应用SMC。
  - ① 可以多层嵌套SMC，增加逆向的难度。
  - ② 每一层代码的解密，不一定要设计成一个解密函数进行解密，可以多个解密函数对同一处代码进行解密，最终才能执行。
  - ③ 在解密函数中加入算法或者是反调试方法，阻止逆向人员逆向，增加逆向成本，提高逆向难度。

三. SMC代码自修改

- 2.逆向分析
  - 通过一道多层嵌套的简单SMC题目，进一步理解和使用SMC
  - 学生练习

三. SMC代码自修改

- 首先判断了长度必须为0x1C，最后一位为'}'，通过以后便解密第一段加密的代码，解密完成进入下一层



三. SMC代码自修改

- 进入第二层，右键起始地址，选择**Create Function**，即可反编译函数，该层的主要作用则是判断前**5**个字符分别是什么，判断成功，解密加密代码，然后进入下一层代码执行

```
int __cdecl sub_414C3C(int a1, void (__cdecl *a2)(int, void *))
{
  signed int v2; // ecx@6

  if ( *(_BYTE *)a1 == 102
    && *(_BYTE *)(a1 + 1) == 108
    && *(_BYTE *)(a1 + 2) == 97
    && *(_BYTE *)(a1 + 3) == 103
    && *(_BYTE *)(a1 + 4) == 123 )
  {
    v2 = 0;
    do
      *((_BYTE *)a2 + v2++) ^= 0x43u;
    while ( v2 < 90 );
    a2(a1 + 5, &unk_414A84);
  }
  return 0;
}
```


三. SMC代码自修改

- 第三层的主要作用则是判断接下来的4个字符与0xCC异或后的结果要与一串硬编码的值相同

```
int __cdecl sub_414BE0(int a1, void (*a2)(_DWORD, const char *, ...))
{
  signed int v2; // edx@1
  int v3; // esi@1
  int v5; // [sp+Ch] [bp-8h]@1
  char v6; // [sp+10h] [bp-4h]@1

  v2 = 0;
  v5 = -1817598824;
  v6 = 0;
  v3 = 0;
  while ( (*((_BYTE *)&v5 + v3 + a1 - (_DWORD)&v5) ^ 0xCC) == *((_BYTE *)&v5 + v3) )
  {
    if ( ++v3 >= 4 )
    {
      do
        *((_BYTE *)a2 + v2++) ^= 0x55u;
      while ( v2 < 347 );
      a2(a1 + 4, (const char *)&unk_414A30);
      return 0;
    }
  }
  return 0;
}
```


三. SMC代码自修改

- 第四层主要是**base64**编码，将我们输入的后续字符的一部分**base64**编码，然后与**cmVhbEN0Rl8=**作比较

```
  *((_BYTE *)&v20 + v3++) = v2++;
while ( v2 <= 90 );
v4 = 97;
do
  *((_BYTE *)&v20 + v3++) = v4++;
while ( v4 <= 122 );
v5 = 48;
do
  *((_BYTE *)&v20 + v3++) = v5++;
while ( v5 <= 57 );
*(__int16 *)((char *)&v20 + v3) = 12075;
v21[v3] = 0;
v6 = &v22;
v22 = 0;
v23 = 0;
v26 = 1750494563;
v27 = 810435938;
v28 = 1027107922;
v24 = 0;
v29 = 0;
v25 = 0;
v7 = 0;
do
{
  v8 = *(_BYTE *)(a1 + v7 + 1);
  v9 = *(_BYTE *)(a1 + v7 + 2);
  v10 = *(_BYTE *)(a1 + v7) & 3;
  v11 = (unsigned int)*(_BYTE *)(a1 + v7) >> 2;
  v7 += 3;
  *v6 = *((_BYTE *)&v20 + v11);
  v6[1] = *((_BYTE *)&v20 + (((unsigned int)v8 >> 4) | 16 * v10));
  v6[2] = *((_BYTE *)&v20 + (((unsigned int)v9 >> 6) | 4 * (v8 & 0xF)));
  v6[3] = *((_BYTE *)&v20 + (v9 & 0x3F));
  v6 += 4;
}
while ( v7 < 6 );
```

三. SMC代码自修改

- 第五层即，将最后的字符中每一个字符加1，要与一串硬编码的值相同，即可通关，获得最终的flag
- 通过5关之后，得到最后的flag:flag{The_realCtF_just_B3g!n}

```
int __cdecl sub_414A30(_BYTE *a1)
{
  signed int v1; // ebx@1
  signed int v2; // edx@1
  _BYTE *v3; // esi@4
  int v5; // [sp+Ch] [bp-Ch]@1
  int v6; // [sp+10h] [bp-8h]@1
  __int16 v7; // [sp+14h] [bp-4h]@1
  char v8; // [sp+16h] [bp-2h]@1

  v1 = 0;
  v5 = 1970566763;
  v6 = 1748255584;
  v2 = -1;
  v7 = 28450;
  v8 = 0;
  do
    ++v2;
  while ( *((_BYTE *)&v5 + v2) );
  if ( v2 > 0 )
  {
    v3 = a1;
    do
    {
      if ( *v3 != v3[(char *)&v5 - a1] - 1 )
        break;
      ++v1;
      ++v3;
    }
    while ( v1 < v2 );
  }
  return 0;
}
```

崔宝江

第7章 代码混淆

- 一. 代码混淆简介
- 二. 花指令混淆
- 三. SMC代码自修改
- 四. OLLVM混淆

北邮网安学院　崔宝江

四. OLLVM混淆

- 1.OLLVM简介
- 2.OLLVM编译与使用

北邮网安学院　崔宝江

四. OLLVM混淆

- 1.OLLVM简介
  - OLLVM(Obfuscator-LLVM)是瑞士西部应用科学大学安全小组于2010年6月发起的一个项目
  - 该项目的目的是提供一套开源的基于LLVM的代码混淆工具，通过代码混淆和防篡改提供更高的软件安全性

四. OLLVM混淆

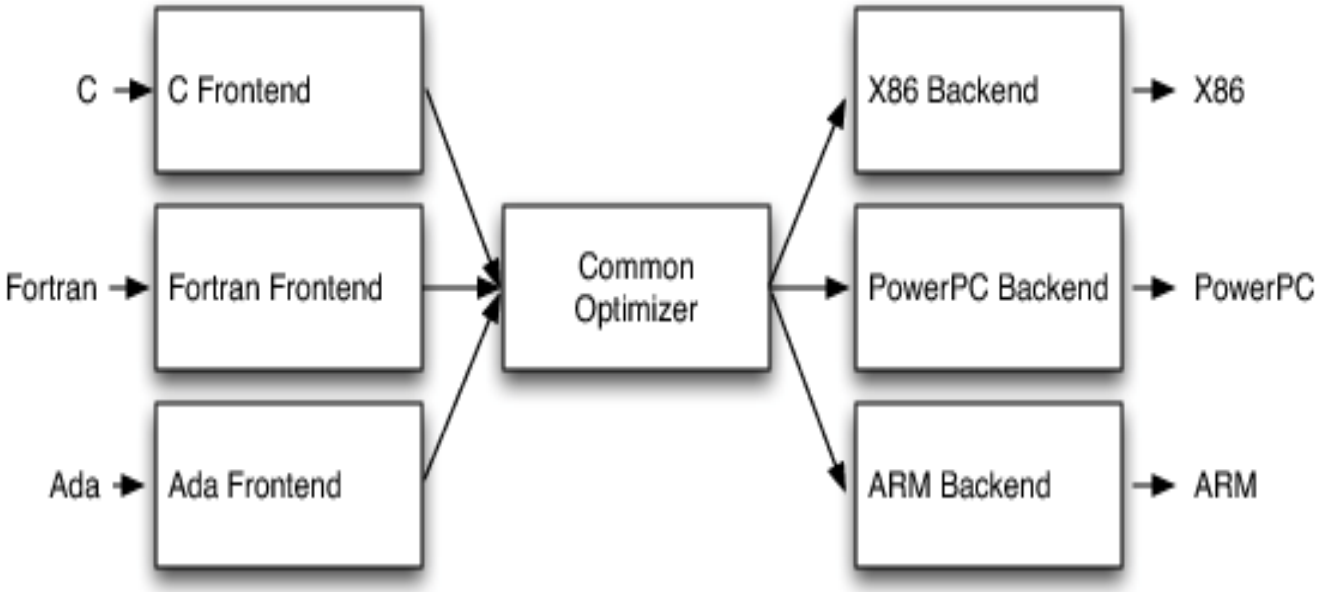
- 1.OLLVM简介
  - LLVM是一个开源的编译器架构，利用虚拟技术提供现代化的源代码与目标无关的优化和针对多种CPU的代码生成功能。
  - LLVM核心库提供了与编译器相关的支持，可以作为多种语言编译器的后台来使用，能够进行程序语言的编译期优化、链接优化、在线编译优化、代码生成。

四. OLLVM混淆

- LLVM采用经典的编译器三段式设计，分为
  - 前端
    - 前端解析源代码，由语法分析器和语义分析协同工作，检查语法错误，并构建语言的抽象语法树AST来表示输入代码，然后将分析好的代码转化为LLVM的中间表示IR（Intermediate Representation）；
    - 注：通常但不总是先构建AST，然后将AST转换为LLVM IR
  - 优化器
    - 优化器通过一系列的Pass对中间代码IR进行优化，改善代码的运行时间使代码更高效；
  - 后端
    - 后端负责将优化器优化后的中间代码IR转换为目标机器的代码

四. OLLVM混淆

- LLVM实现三段式设计



四. OLLVM混淆

- **LLVM IR**
  - **LLVM IR是LLVM的中间表示，用来在编译器中表示代码的形式，优化器通过一系列的Pass对IR进行优化操作，直至生成后端可用的IR。**

四. OLLVM混淆

- OLLVM工作原理
  - OLLVM工作在LLVM IR中间表示层，通过编写Pass来混淆IR，后端依照IR转换的目标机器的代码也就达到了混淆的目的。
  - 因为OLLVM是基于LLVM设计的，因此它支持LLVM支持的所有编程语言（C，C ++，Objective-C，Ada和Fortran）以及目标平台（x86，x86-64，PowerPC，PowerPC-64，ARM，Thumb，SPARC，Alpha，CellSPU，MIPS，MSP430，SystemZ和XCore）。

四. OLLVM混淆

- OLLVM提供三种混淆模式
  - 指令替换（Instructions Substitution）
    - 参数为-sub
  - 虚假控制流程（Bogus Control Flow）
    - 参数为- bcf
  - 控制流平展（Control Flow Flattening）
    - 参数为- fla
  - 注：三个参数可以单独使用，也可以一起配合使用

四. OLLVM混淆

- OLLVM目前正在开发函数合并、防篡改等混淆功能，未来可能出现常量加密、垃圾代码插入、反调试技巧插入等混淆模式

四. OLLVM混淆

- 2.OLLVM编译与使用
  - OLLVM可以在Windows和Linux平台编译与使用，这里选择的平台为Linux，操作系统为Ubuntu，版本为14.04 LTS 64位。

四. OLLVM混淆

（1）OLLVM的编译

- 安装OLLVM前需要安装git和cmake

```
sudo apt-get install git
sudo pip install cmake
```

- OLLVM的安装命令

```
git clone -b llvm-4.0 https://github.com/obfuscator-llvm/obfuscator.git
mkdir build
cd build
cmake -G "Unix Makefiles"-DCMAKE_BUILD_TYPE=Release -DLLVM_INCLUDE_TESTS=OFF ../obfuscator
make-j7
```


四. OLLVM混淆

- （2）OLLVM的使用
  - 以C程序来说明OLLVM的三种混淆模式
    - ① 指令替换
      - 通过功能上等价的但更复杂的指令序列，替换标准二元运算符（如加法、减法或布尔运算符），当有多个可用的等效指令序列时，随机选择一个
    - ② 虚假控制流程
      - 通过在当前基本块之前添加基本块来修改程序的控制流图，原始的基本块也会被克隆，并插入随机的垃圾指令。
    - ③ 控制流平展
      - 使用该模式后，程序的控制流图被完全压扁。

四. OLLVM混淆

- （2）OLLVM的使用
  - 编写一个简单的程序进行测试，分别使用OLLVM的三种混淆模式对该程序进行代码混淆，比较它在混淆前后的汇编代码

四. OLLVM混淆

```
#include<stdio.h>
#include<stdio.h>
int main(){
int a =200;
int b =100;
int c =0;
int d =0;
int e =0;
int i =10;
while(i >0){
if(a > b){
            c = a + b;
            d = a & b;
            e = a ^ b;
}
else{
            c = a - b;
            d = a | b;
}
        i = i -1;
        a = a -20;
        b = b -5;
}
    printf("c = %d\nd = %d\ne = %d\n",c,d,e);
return0;
}
```

程序逻辑非常简单，进行了10次循环操作，每次比较a和b的大小关系，根据比较结果进行相应的运算

北邮网安学院　崔宝江

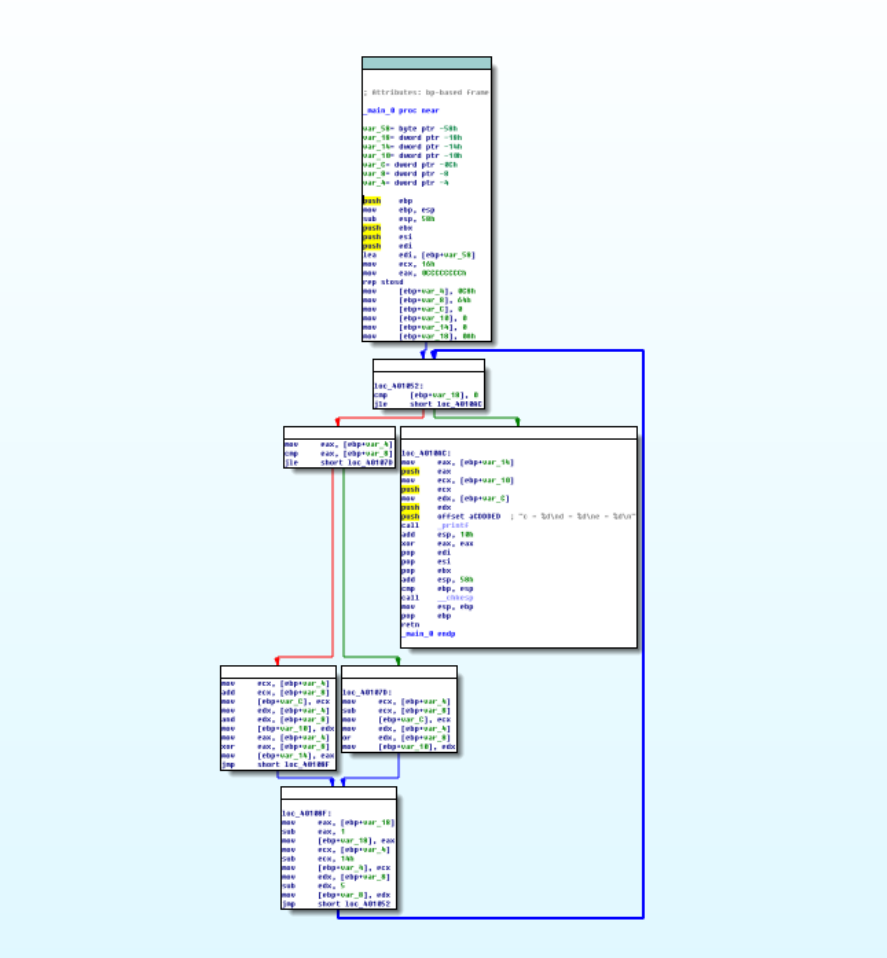
四. OLLVM混淆

- 在进行混淆前，使用IDA打开test.exe，可以看到test.exe的与运算相关的汇编代码

```
.text:00401044                  mov     [ebp+var_14], 0
.text:0040104B                  mov     [ebp+var_18], 0Ah
.text:00401052
.text:00401052 loc_401052:                              ; CODE XREF: _main_0+9A↓j
.text:00401052                  cmp     [ebp+var_18], 0
.text:00401056                  jle     short loc_4010AC
.text:00401058                  mov     eax, [ebp+var_4]
.text:0040105B                  cmp     eax, [ebp+var_8]
.text:0040105E                  jle     short loc_40107D
.text:00401060                  mov     ecx, [ebp+var_4]    ecx = a , ecx = a + b
.text:00401063                  add     ecx, [ebp+var_8]
.text:00401066                  mov     [ebp+var_C], ecx
.text:00401069                  mov     edx, [ebp+var_4]    edx = a , edx = a & b
.text:0040106C                  and     edx, [ebp+var_8]
.text:0040106F                  mov     [ebp+var_10], edx
.text:00401072                  mov     eax, [ebp+var_4]    eax = a , eax = a ^ b
.text:00401075                  xor     eax, [ebp+var_8]
.text:00401078                  mov     [ebp+var_14], eax
.text:0040107B                  jmp     short loc_40108F
```


四. OLLVM混淆

- 控制流图结构比较简单清晰



崔宝江

四. OLLVM混淆

① 指令替换（Instructions Substitution）

- 通过功能上等价的但更复杂的指令序列替换标准二元运算符（如加法、减法或布尔运算符），当有多个可用的等效指令序列时，随机选择一个
- 例如： **a=b+c**
- 等价表达式

```
a =(b -(-c))
a =-(-b +(-c))
r = rand(); a = b + r; a = a + c; a = a - r
r = rand(); a = b - r; a = a + c; a = a + r
```

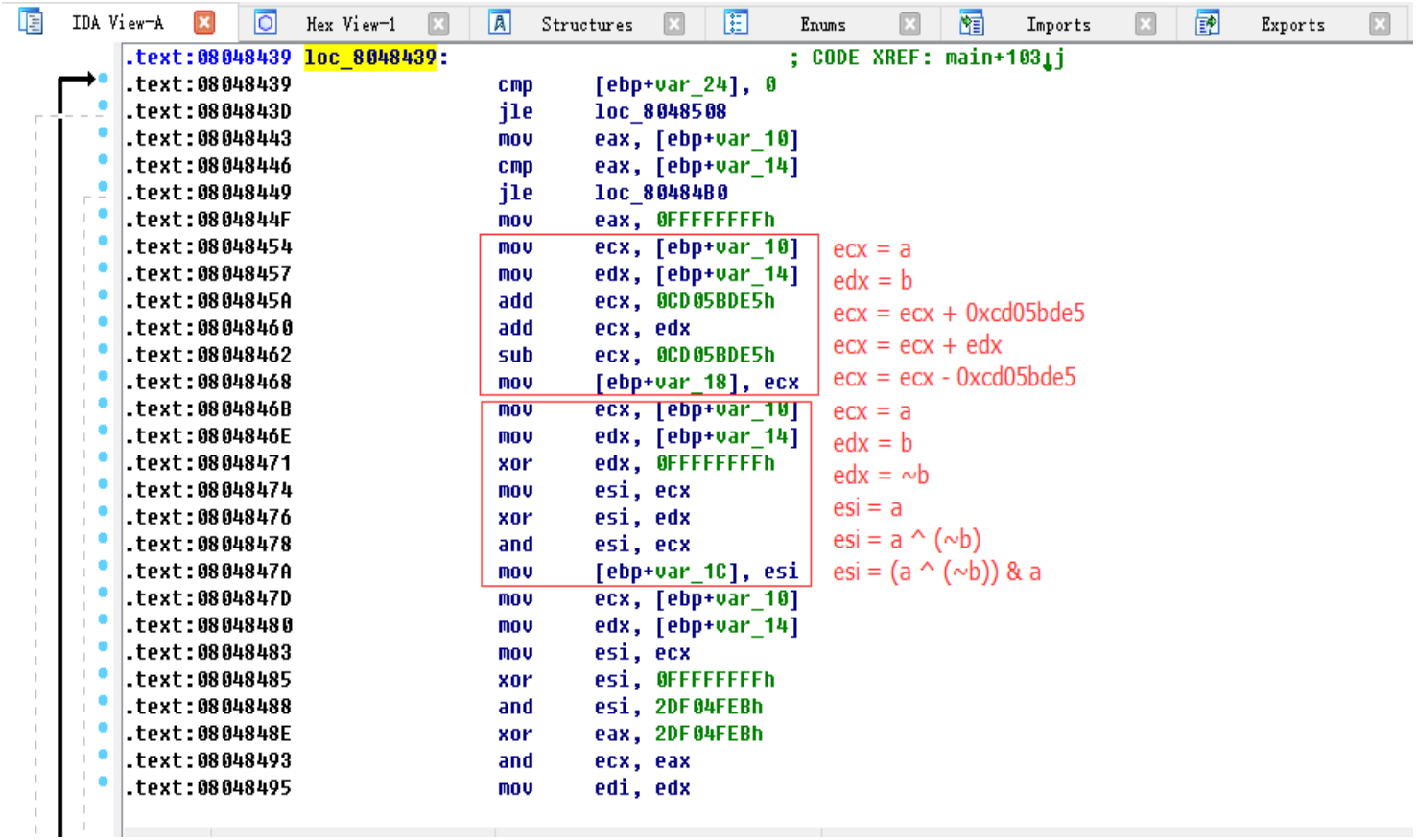

四. OLLVM混淆

- 目前，OLLVM支持只有整数参与（浮点数会带来舍入误差和不必要的数值不准确性）的加法、减法、与、或、异或运算
- 使用以下命令对test.c进行指令替换操作

```
#将test.c 置于 your_ollvm_path/build/test文件夹下
cd your_ollvm_path/build/bin
./clang -m32 ../test/test.c -o ../test/test_sub -mllvm -sub
```


四. OLLVM混淆

- 使用IDA打开test_sub，可以看到在相关运算处做了指令替换的混淆



四. OLLVM混淆

```
.text:08048462                sub     ecx, 0CD05BDE5h
.text:08048468                mov     [ebp+var_18], ecx
.text:0804846B                mov     ecx, [ebp+var_10]
.text:0804846E                mov     edx, [ebp+var_14]
.text:08048471                xor     edx, 0FFFFFFFFh
.text:08048474                mov     esi, ecx
.text:08048476                xor     esi, edx
.text:08048478                and     esi, ecx
.text:0804847A                mov     [ebp+var_1C], esi
.text:0804847D                mov     ecx, [ebp+var_10]
.text:08048480                mov     edx, [ebp+var_14]
.text:08048483                mov     esi, ecx
.text:08048485                xor     esi, 0FFFFFFFFh
.text:08048488                and     esi, 2DF04FEBh
.text:0804848E                xor     eax, 2DF04FEBh
.text:08048493                and     ecx, eax
.text:08048495                mov     edi, edx
.text:08048497                xor     edi, 0FFFFFFFFh
.text:0804849A                and     edi, 2DF04FEBh
.text:080484A0                and     edx, eax
.text:080484A2                or      esi, ecx
.text:080484A4                or      edi, edx
.text:080484A6                xor     esi, edi
.text:080484A8                mov     [ebp+var_20], esi
.text:080484AB                jmp     loc_80484D0
```

esi = (~a) & 0x2df04feb
ecx = a & (~0x2dd04feb)

edi = (~b) & 0x2df04feb
edx = b & (~0x2df04feb)

a ^ b = (esi | ecx) ^ (edi | edx)

北邮网安学院　崔宝江

四. OLLVM混淆

- 加法、与运算、异或运算的指令转换为

```
c = a + b => c = a +0xCD05BDE5;c = c + b;c = c -0xCD0DBDE5
c = a & b => c =(a ^~b)% a
c = a ^ b => m = rand();a =((~a)& m)|(a &(~m));b =((~b)& m)|(b &(~m));c = a ^ b
```


四. OLLVM混淆

② 虚假控制流程

- 虚假控制流程（**Bogus Control Flow**）：通过在当前基本块之前添加基本块来修改程序的控制流图，原始的基本块也会被克隆，并插入随机的垃圾指令。
- 使用以下命令对**test.c**进行虚假控制流程混淆操作

```
#将test.c 置于 your_ollvm_path/build/test文件夹下
cd your_ollvm_path/build/bin
./clang -m32 ../test/test.c -o ../test/test_bcf -mllvm -bcf
```

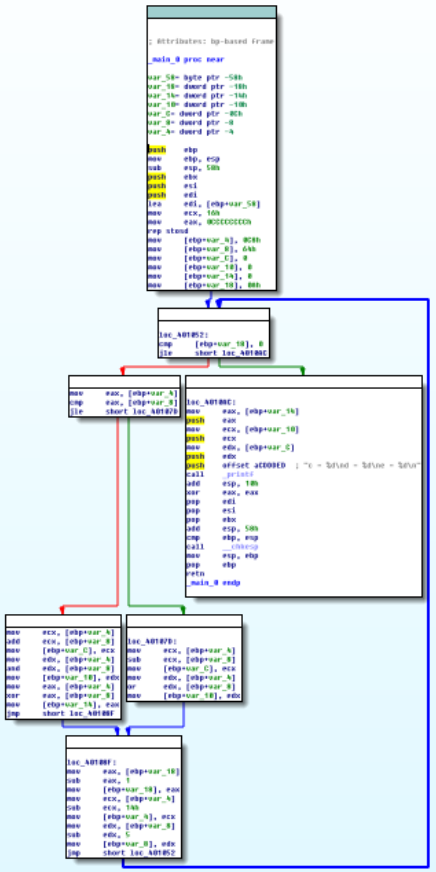

四. OLLVM混淆

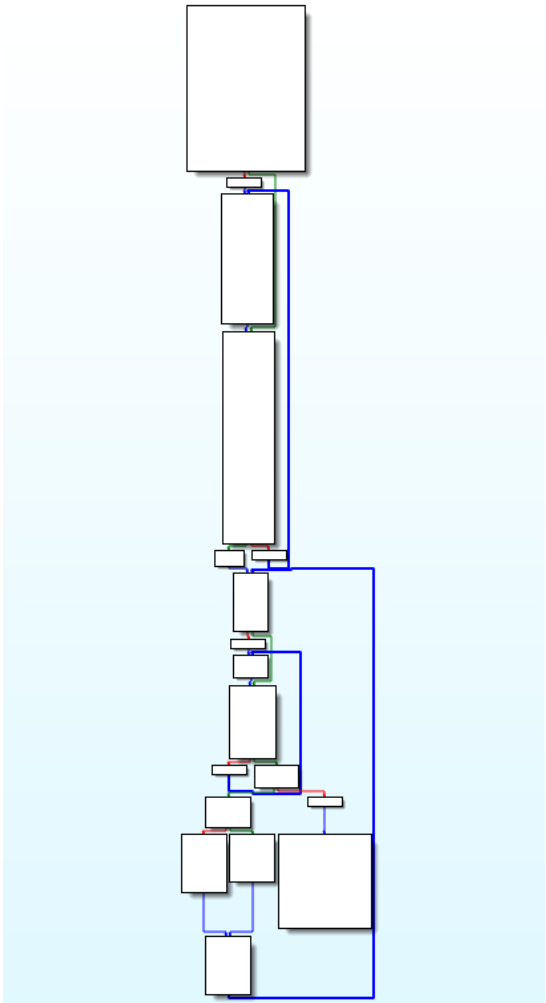
- 使用IDA打开test_bcf，可以看到该程序原来的基本块里被插入了很多垃圾指令

```
.text:08048481                mov     dword ptr [edx], 64h
.text:08048487                mov     dword ptr [esi], 0
.text:0804848D                mov     dword ptr [edi], 0
.text:08048493                mov     dword ptr [ebx], 0
.text:08048499                mov     eax, [ebp+var_14]
.text:0804849C                mov     dword ptr [eax], 0Ah
.text:080484A2                mov     eax, ds:x
.text:080484A7                mov     [ebp+var_18], eax
.text:080484AA                mov     eax, ds:y
.text:080484AF                mov     [ebp+var_1C], eax
.text:080484B2                mov     eax, [ebp+var_18]
.text:080484B5                sub     eax, 1
.text:080484B8                mov     [ebp+var_20], eax
.text:080484BB                mov     eax, [ebp+var_18]
.text:080484BE                mov     [ebp+var_24], ecx
.text:080484C1                mov     ecx, [ebp+var_20]
.text:080484C4                imul    eax, ecx
.text:080484C7                and     eax, 1
.text:080484CA                cmp     eax, 0
.text:080484CD                setz    al
.text:080484D0                mov     ecx, [ebp+var_1C]
.text:080484D3                cmp     ecx, 0Ah
.text:080484D6                setl    ah
.text:080484D9                or      al, ah
.text:080484DB                test    al, 1
.text:080484DD                mov     [ebp+var_28], ebx
.text:080484E0                mov     [ebp+var_2C], edi
.text:080484E3                mov     [ebp+var_30], edx
```

四. OLLVM混淆

- 程序的控制流程变得复杂





四. OLLVM混淆

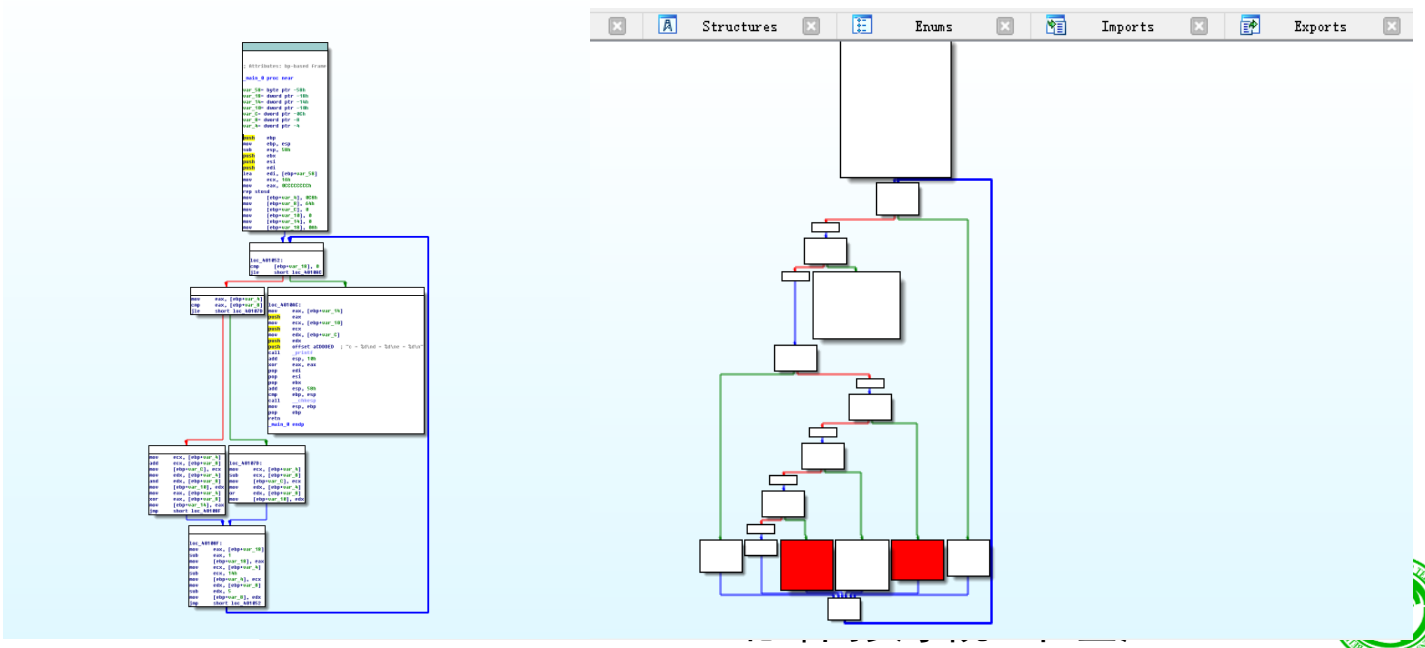
③ 控制流平展

- 控制流平展（**Control Flow Flattening**）：使用该模式后，程序的控制流图被完全压扁。
- 使用以下命令对**test.c**进行控制流平展混淆操作

```
#将test.c 置于 your_ollvm_path/build/test文件夹下
cd your_ollvm_path/build/bin
./clang -m32 ../test/test.c -o ../test/test_fla -mllvm -fla
```


四. OLLVM混淆

- 使用IDA打开test_fla，可以看到程序的控制流程被压扁，相比于虚假控制流程混淆模式，该模式没有插入垃圾指令，只是出现很多代码分支，控制流程被打乱，红色的是原来的if-else基本块



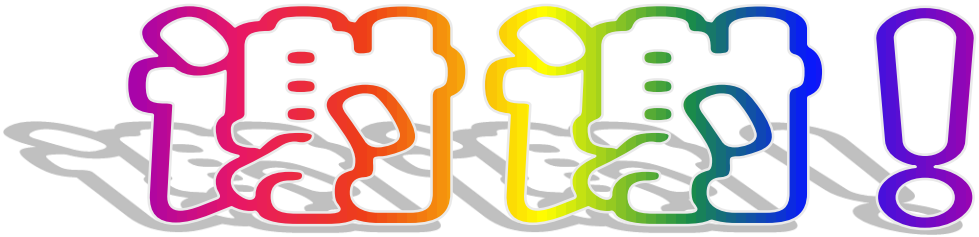
四. OLLVM混淆

- 使用快捷键**F5**反编译后可以看到原来的**if-else**逻辑变成了**switch**循环

IDA View-A | Pseudocode-A | Hex View-1 | Structures | Enums

```
30      break;
31    switch ( v6 )
32    {
33      case -943850392:
34        v3 = -1932896546;
35        if ( v7 > 0 )
36          v3 = -2027726344;
37        v6 = v3;
38        break;
39      case -297877476:
40        v10 = v12 - v11;      c = a - b
41        v9 = v11 | v12;       d = a | b
42        v6 = 839835941;
43        break;
44      case 839835941:
45        --v7;
46        v12 -= 20;
47        v11 -= 5;
48        v6 = -943850392;
49        break;
50      case 872273810:
51        v10 = v11 + v12;      c = a + b
52        v9 = v11 & v12;       d = a & b
53        v8 = v11 ^ v12;       e = a ^ b
54        v6 = 839835941;
55        break;
56    }
57  }
```

Q & A



北邮网安学院　崔宝江