北京郵電大學
Beijing University of Posts and Telecommunications

Merak
hacking securily

汇编语言与逆向工程
Assembly Language and Software Reverse Engineering

北京邮电大学
付俊松

2024年4月

第七章 异常处理

- 一. 计算机系统中的异常机制
- 二.程序异常及异常处理
- 三. Windows异常处理流程
- 四. SEH结构化异常处理机制
- 五. SEH反调原理
- 六. SEH逆向分析

（1）计算机系统中的异常机制

- 异常机制在计算机系统中扮演核心角色之一，存在于整个系统的各个层面：
  - 硬件层面
    - 电源异常、时钟异常、I/O异常等、内存页非法访问等
  - 操作系统层面
    - 进程切换、任务调度等
  - 进程层面
    - 除零异常、断点异常等

（1）计算机系统中的异常机制

- 计算机异常
  - Interrupt
    - 用于处理系统I/O产生的中断，通常是硬件异常
  - Trap
    - 用于调用系统内核中的功能
  - Fault
    - 危害程度较低的系统异常，操作系统将尝试修复，修复失败将转为Abort
  - Abort
    - 危害程度较高的系统异常，系统将放弃修复并直接终止程序运行

第七章 异常处理

- 一. 计算机系统中的异常机制
- 二.程序异常及异常处理
- 三. Windows异常处理流程
- 四. SEH结构化异常处理机制
- 五. SEH反调原理
- 六. SEH逆向分析

(2) 程序异常及异常处理

- 1.程序异常
- 2.异常处理

(2) 程序异常及异常处理

- 程序异常
  - 程序异常（Exception）是程序运行过程中由于种种原因导致的意外情况，是程序本身可以处理的“错误”，也是程序必须处理、无法忽略的“错误”
    - 有些是因为程序本身的逻辑错误，如非法除零、非法内存读写等
    - 有些是用户操作导致程序错误，如用户误删文件导致文件无法找到等
    - 还有一些其他异常，如网络访问超时异常
  - 程序异常VS程序错误

(2) 程序异常及异常处理

– JAVA语言中的异常类
- 算术异常类：ArithmeticExecption
- 类型强制转换异常：ClassCastException
- 字符串转换为数字异常：NumberFormatException
- 输入输出异常：IOException
- 文件未找到异常：FileNotFoundException
- 文件已结束异常：EOFException
- 空指针异常类：NullPointerException
- 数组负下标异常：NegativeArrayException
- 数组下标越界异常：ArrayIndexOutOfBoundsException
- 违背安全原则异常：SecturityException
- 方法未找到异常：NoSuchMethodException
- 操作数据库异常：SQLException

(2) 程序异常及异常处理

– 微软提供的winnt.h文件中对程序运行中可能遇到的多种异常进行了定义（VC++6.0 Debug版本）

```
#define STATUS_GUARD_PAGE_VIOLATION      ((DWORD   )0x80000001L)
#define STATUS_DATATYPE_MISALIGNMENT     ((DWORD   )0x80000002L)
#define STATUS_BREAKPOINT                ((DWORD   )0x80000003L)
#define STATUS_SINGLE_STEP               ((DWORD   )0x80000004L)
#define STATUS_ACCESS_VIOLATION          ((DWORD   )0xC0000005L)
#define STATUS_IN_PAGE_ERROR             ((DWORD   )0xC0000006L)
#define STATUS_INVALID_HANDLE            ((DWORD   )0xC0000008L)
#define STATUS_NO_MEMORY                 ((DWORD   )0xC0000017L)
#define STATUS_ILLEGAL_INSTRUCTION       ((DWORD   )0xC000001DL)
#define STATUS_NONCONTINUABLE_EXCEPTION  ((DWORD   )0xC0000025L)
#define STATUS_INVALID_DISPOSITION       ((DWORD   )0xC0000026L)
#define STATUS_ARRAY_BOUNDS_EXCEEDED     ((DWORD   )0xC000008CL)
#define STATUS_FLOAT_DENORMAL_OPERAND    ((DWORD   )0xC000008DL)
#define STATUS_FLOAT_DIVIDE_BY_ZERO      ((DWORD   )0xC000008EL)
#define STATUS_FLOAT_INEXACT_RESULT      ((DWORD   )0xC000008FL)
#define STATUS_FLOAT_INVALID_OPERATION   ((DWORD   )0xC0000090L)
#define STATUS_FLOAT_OVERFLOW            ((DWORD   )0xC0000091L)
#define STATUS_FLOAT_STACK_CHECK         ((DWORD   )0xC0000092L)
#define STATUS_FLOAT_UNDERFLOW           ((DWORD   )0xC0000093L)
#define STATUS_INTEGER_DIVIDE_BY_ZERO    ((DWORD   )0xC0000094L)
#define STATUS_INTEGER_OVERFLOW          ((DWORD   )0xC0000095L)
#define STATUS_PRIVILEGED_INSTRUCTION    ((DWORD   )0xC0000096L)

#define STATUS_STACK_OVERFLOW            ((DWORD   )0xC00000FDL)
```

(2) 程序异常及异常处理

– 五种最具代表性的异常，在程序运行过程中最为常见

异常种类	含义
EXCEPTION_ACCESS_VIOLATION(C0000005)	试图访问不存在或无访问权限的内存区域
EXCEPTION_BREAKPOINT(80000003)	在运行代码中设置断点后，CPU尝试执行该地址处的指令时就会发生断点异常
EXCEPTION_ILLEGAL_INSTRUCTION(C000001D)	CPU遇到无法解析的命令
EXCEPTION_INT_DIVIDE_BY_ZERO(C0000094)	整数除法运算中若分母为零则引发除零异常
EXCEPTION_SINGLE_STEP(80000004)	CPU在单步工作模式下，每执行一条指令就会引发一次单步异常

(2) 程序异常及异常处理

2.异常处理（Exception handling）

- 针对程序异常的应对和解决方法称为异常处理，异常处理通过处理程序运行时可能出现的异常情况，尽可能的使程序不受异常影响稳健运行，一个异常结构完备的系统不会有运行时错误
- 异常处理机制是用于处理程序异常状况的系统化处理方法，它使得程序代码更加简洁、干净，而且不容易漏掉代码中出现的异常
- 在常见的异常处理机制实现过程中，程序通过抛出异常的形式将这些意外情况告诉上级调用者，系统会强制调用者对异常进行处理

(2) 程序异常及异常处理

- 程序代码片段主要通过地址跳转（函数内部）以及函数调用（函数外部）进行前后串联执行，异常处理机制提供了一种新的代码片段有机串联机制
- 从分离代码的角度上看，异常处理机制在功能上近似等价于一种分支处理机制
  - 将异常看做程序运行中不常见的分支情况，异常处理就是对这些不常见的分支情况的处理

（2）程序异常及异常处理

- 尽管异常处理机制可以理解为一种分支机制，但有些异常处理不可以由if/else条件结构替代
  - 条件判断是提前进行的，不一定能准确地发现异常，而对于异常的判断是实时的，是与代码运行同时的
  - 有些异常的发生是可预测的，有些异常的发生是难以预测的
    - 像除零异常这类可预测的异常可以使用条件结构进行判断，但像文件读写、序列化对象等不可预测的异常就难以用条件结构进行控制，应使用异常处理机制

(2) 程序异常及异常处理

- 为了优化异常处理过程，操作系统引进了异常处理机制，将这些不常见的分支归为异常，进行统一的异常处理，便于程序员在编写程序时将注意力集中于正常情况处理

（2）程序异常及异常处理

– 如果没有引入异常处理机制，会产生以下后果

- 程序员需要通过if else等分支语句，考虑所有可能出现的异常情况，包括业务、数据、网络等多方面因素，编程难度进一步增加
- 由于if else等分支语句的增加，代码可读性下降，难以聚焦业务逻辑，代码维护难度增加
- 如果异常本身需要外部函数（而非本地函数）进行处理，异常处理过程需要冲破多层函数，程序异常难度会进一步增加，而异常处理机制将使抛出的异常冲破外层调用，直至达到系统层，此时程序终止

第七章 异常处理

- 一. 计算机系统中的异常机制
- 二.程序异常及异常处理
- 三. Windows异常处理流程
- 四. SEH结构化异常处理机制
- 五. SEH反调原理
- 六. SEH逆向分析

(3) Windows异常处理流程

- Windows程序运行分为正常运行模式与调试运行模式两种情况，在不同的运行情况下，操作系统对程序的异常处理流程也有所不同

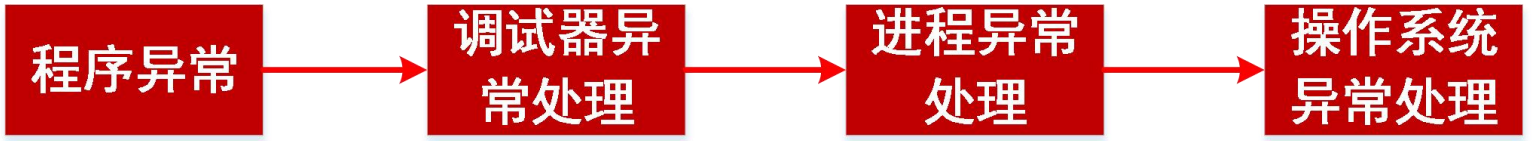
(3) Windows异常处理流程

– 在程序正常运行的情况下，当异常发生时

- 操作系统会先将异常抛给进程处理，进程代码中如果存在具体的异常处理代码，则能顺利处理异常继续运行
- 如果没有，则操作系统启动默认异常处理，终止进程运行

(3) Windows异常处理流程

- 在调试运行的情况下，当异常发生时
  - 操作系统会先把异常抛给调试器进程，由调试人员进一步选择异常处理的方式
  - 调试者在使用调试器处理被调程序异常时有两种方法
    - 直接修改代码、寄存器、内存来修改异常
    - 将异常抛给被调试程序处理
  - 如果这两种方法无法处理异常，则操作系统会使用默认异常处理机制进行处理，终止被调试程序，同时结束调试



第七章 异常处理

- 一. 计算机系统中的异常机制
- 二.程序异常及异常处理
- 三. Windows异常处理流程
- 四. SEH结构化异常处理机制
- 五. SEH反调原理
- 六. SEH逆向分析

(4) SEH结构化异常处理

- 在Windows操作系统中，异常处理机制由结构化异常处理机制(SEH)来实现

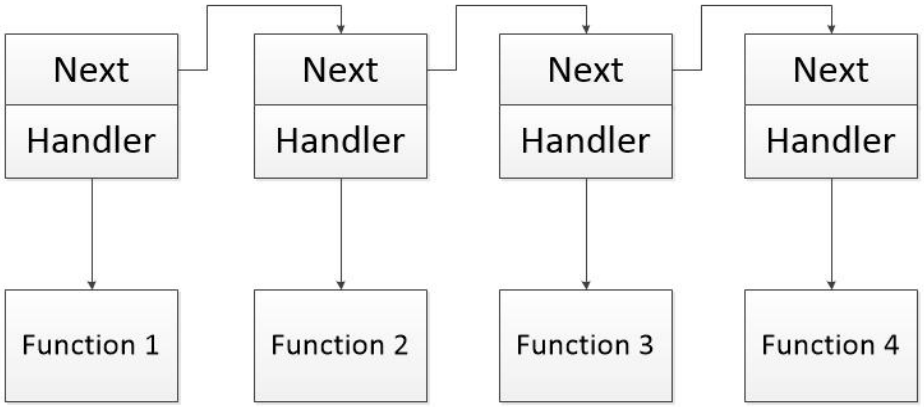
(4) SEH结构化异常处理

- 1. SEH链
- 2. SEH的应用原理
- 3. 在程序中添加SEH

(4) SEH结构化异常处理 — SEH链

1. SEH链

- SEH（Structured Exception Handling，结构化异常处理）是Windows操作系统默认的异常处理机制，在程序源码中使用__try、__except等关键字来实现
- 从数据结构上来看，SEH以链表的形式存在，称为SEH链



(4) SEH结构化异常处理 — SEH链

- SEH链中的每个节点都是一个_EXCEPTION_REGISTRATION_RECORD结构体，称为异常处理器（有时异常处理器也代指异常处理函数）
- 一个异常处理器有两个功能：
  - 指明下一个异常处理器的位置
  - 提供用于处理异常的代码

(4) SEH结构化异常处理 — SEH链

- _EXCEPTION_REGISTRATION_RECORD结构体有Next和Handler两个成员
  - Next成员是一个结构体指针，指向下一个_EXCEPTION_REGISTRATION_RECORD结构体，也就是下一个异常处理器的地址，如果Next的值为FFFFFFFF，则表示SEH链到此结束
  - Handler成员是一个函数指针，指向异常处理函数，异常处理函数是一个回调函数，由系统调用

```
Typedef struct _EXCEPTION_REGISTRATION_RECORD{
        PEXCEPTION_REGISTRATION_RECORD Next;
        PEXCEPTION_DISPOSITION Handler;
}EXCEPTION_REGISTRATION_RECORD,*PEXCEPTION_REGISTRATION_
RECORD;
```

(4) SEH结构化异常处理 — SEH链

– 下面是一个异常处理函数的原型

➢异常处理函数有四个参数，这四个参数用来传递与异常相关的信息，包括异常类型、发生异常的代码地址、异常发生时CPU寄存器的状态等

```
EXCEPTION_DISPOSITION _except_handler{
        EXCEPTION_RECORD *pRecord,
        EXCEPTION_REGISTRATION_RECORD *pFrame,
        CONTEXT *pContext,
        PVOID pValue
};
```

(4) SEH结构化异常处理 — SEH链

- 异常处理函数返回一个名为EXCEPTION_DISPOSITION的枚举类型，用于告知系统异常处理完成后程序应如何继续运行

```
Typedef enum _EXCEPTION_DISPOSITION{
        ExceptionContinueExecution =0,//继续执行异常代码
        ExceptionContinueSearch =1,//运行下一个异常处理器
        ExceptionNestedException =2,//在OS内部使用
        ExceptionCollidedUnwind =3//在OS内部使用
}EXCEPTION_DISPOSITION;
```

(4) SEH结构化异常处理 — SEH的应用原理

2. SEH的应用原理

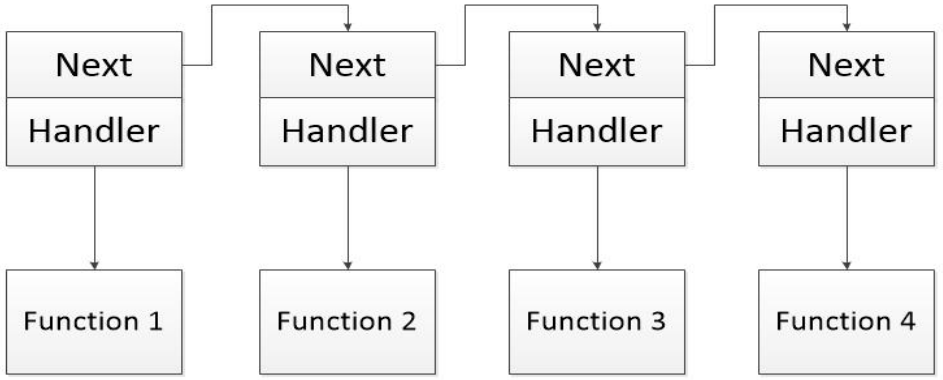
- 系统在使用SEH链时需要知道SEH链的地址
- SEH链的头部地址储存在TEB（Thread Environment Block）中
  - TEB是线程描述块，存储着线程运行所需的各种信息，例如线程的空间大小、寄存器状态、堆栈地址等
  - 在TEB的第一个DWORD成员中存储着SEH链表头的地址，系统就可以通过这个地址找到SEH链

(4) SEH结构化异常处理 — SEH的应用原理

- 当进程发生异常时，系统首先找出发生异常的线程，并根据该线程TEB中的信息获得第一个异常处理器的地址
- 之后，异常被提交给进程的SEH链的第一个异常处理器，由异常处理函数对异常进行处理
- 如果第一个异常处理器不能处理相关异常，则异常会被传递到下一个异常处理器，直到异常得到处理或进程SEH链结束，将异常抛给操作系统
- 如果异常得到处理则程序继续运行，如果异常抛给操作系统，则系统会终止程序运行

(4) SEH结构化异常处理 — SEH的应用原理

- 系统首先通过TEB获取第一个异常处理器的地址，然后将程序异常交由第一个异常处理器处理
- 如果Function1无法处理异常，则将异常传递到第二个异常处理器，如果Function2也无法处理异常，则继续传递下去，直到某个异常处理器能解决该异常
- 如果传递到第四个异常处理器，Function4仍无法处理异常，则该异常将被抛给操作系统，由操作系统默认的异常处理进行处理



(4) SEH结构化异常处理 — SEH的应用原理

- TEB（Thread Environment Block，线程环境块）系统在此TEB中保存频繁使用的线程相关的数据
- 进程中的每个线程都有自己的一个TEB。一个进程的所有TEB都以堆栈的方式，存放在从0x7FFDE000开始的线性内存中，每4KB为一个完整的TEB，不过该内存区域是向下扩展的
- 在用户模式下，当前线程的TEB位于独立的4KB段，可通过CPU的FS寄存器来访问该段，一般存储在[FS:0]

(4) SEH结构化异常处理 — 在程序中添加SEH

3. 在SEH链中添加异常处理器

- C语言中，程序员只需要使用_try将要监视运行的代码包起来，在__except中编写异常处理代码，语法如下所示
- SEH添加的其余工作交由编译器完成

```
__try{
// guarded code
}
__except ( expression ){
// exception handler code
}
```

(4) SEH结构化异常处理 — 在程序中添加SEH

3. 在SEH链中添加异常处理器

- 汇编语言层面，异常处理器的添加有以下三个步骤：
  - 1）编写异常处理函数
  - 2）构造异常处理器
  - 3）将异常处理器从表头添加到SEH链

(4) SEH结构化异常处理 — 在程序中添加SEH

– 汇编语言中，需要程序员自己按照步骤添加SEH

- 首先，将异常处理函数地址压栈
- 然后，将SEH链表表头地址压栈
- 此时，ESP指向了新构建的异常处理器地址，因此将表头地址FS:[0]修改为ESP地址

```
PUSH @Handler                ;将异常处理函数地址压栈: Handler
PUSH DWORD PTR FS:[0]        ;将SEH链表表头地址压到:Next
MOV DWORD PTR FS:[0],ESP     ;将表头地址改为新的表头地址
```

第七章 异常处理

- 一. 计算机系统中的异常机制
- 二.程序异常及异常处理
- 三. Windows异常处理流程
- 四. SEH结构化异常处理机制
- 五. SEH反调原理
- 六. SEH逆向分析

(5) SEH反调原理

- SEH反调试的两个层次
  - 利用SEH对代码片段的非常规链接方式（相比于跳转和函数调用）实现反调
    - 单纯利用调试者的SEH知识盲点，将有效代码放入异常处理函数中，在调试者不对SEH调试的条件下，实现对有效代码的隐私保护
  - 利用程序在正常运行与调试运行的不同工作模式，实现反调试
    - 基于Windows提供的系统函数实现反调试，在调试运行的模式下，某些异常处理函数不会被访问，实现对有效代码的隐私保护

(5) SEH反调原理

- SEH反调原理一
  - 编程人员将程序真正的逻辑，放到异常处理函数中，并在逻辑中添加一些混淆和跳转，使调试变得艰难
    - 正常运行的情况下
      - 该程序运行发生异常后，直接调用异常处理函数，实现真正的功能
    - 在调试运行的情况下
      - 该程序运行发生异常后，异常被调试器OD等捕获，可以通过调试器将异常抛回给程序，调用程序异常处理函数，实现真正的功能
    - 调试者通过查看SEH链可以发现自定义的异常处理函数，这种方式很容易被发现

(5) SEH反调原理

SEH反调原理二

- 为避免在调试状态被发现隐藏于异常处理函数中的有效代码，可使用以下两个系统函数：
  - 使用函数UnhandledExceptionFilter可以设置系统默认的异常处理器
    - 该函数只有在非调试状态下才会被调用
    - 将异常处理器设置为系统默认的异常处理器，使异常发生时直接使用默认异常处理器进行处理
  - 使用函数SetUnhandledExceptionFilter设置自定义的异常处理器
    - 该函数只有在非调试状态下才会被调用
    - 该函数将默认异常处理器设置为自定义的异常处理器

第七章 异常处理

- 一. 计算机系统中的异常机制
- 二.程序异常及异常处理
- 三. Windows异常处理流程
- 四. SEH结构化异常处理机制
- 五. SEH反调原理
- 六. SEH逆向分析

（6）SEH逆向分析

- 实验一：使用C语言添加SEH
- 实验二：使用内联汇编块注册SEH
- 实验三：SEH链分析
- 实验四：使用UnhandledExceptionFilter实现反调

(6) SEH逆向分析 — 实验一：使用c语言添加SEH

- 实验内容
  - 实验一为编程实验，要求在代码中设计异常，并使用c语言的__try、__except添加异常处理逻辑，使程序在出现异常时弹出对话框，并修正原有错误使程序继续运行

(6) SEH逆向分析 — 实验一：使用c语言添加SEH

- 示例代码如下：
  - 代码中设计了除零异常，异常处理会弹出对话框并修正异常代码

(6) SEH逆向分析 — 实验一：使用c语言添加SEH

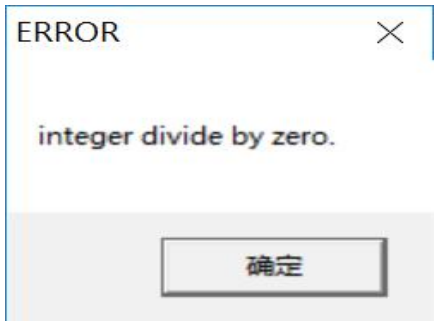
```
#include <stdio.h>
#include <windows.h>
int main(){
        int a =0, b =1, c =0;
          try{
                c = b / a;//触发除零异常
        }
          except(EXCEPTION_EXECUTE_HANDLER){
                MessageBox(NULL, TEXT("integer divide by zero."),
                                                TEXT("ERROR"), MB_OK);

                a =1;//修正的错误
                c = b / a;
        }
        printf("b / a = %d\n", c);
        system("pause");
        return 0;
}
```

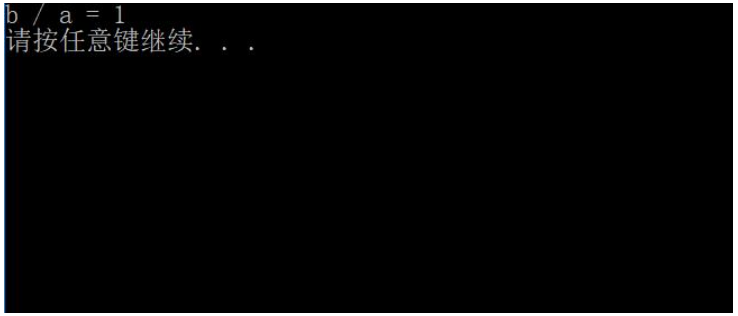
(6) SEH逆向分析 — 实验一：使用c语言添加SEH

– 运行示例程序，程序触发除零异常，启动SEH，运行结果如下

➢启动SEH弹出对话框



➢修正错误后程序继续正常运行



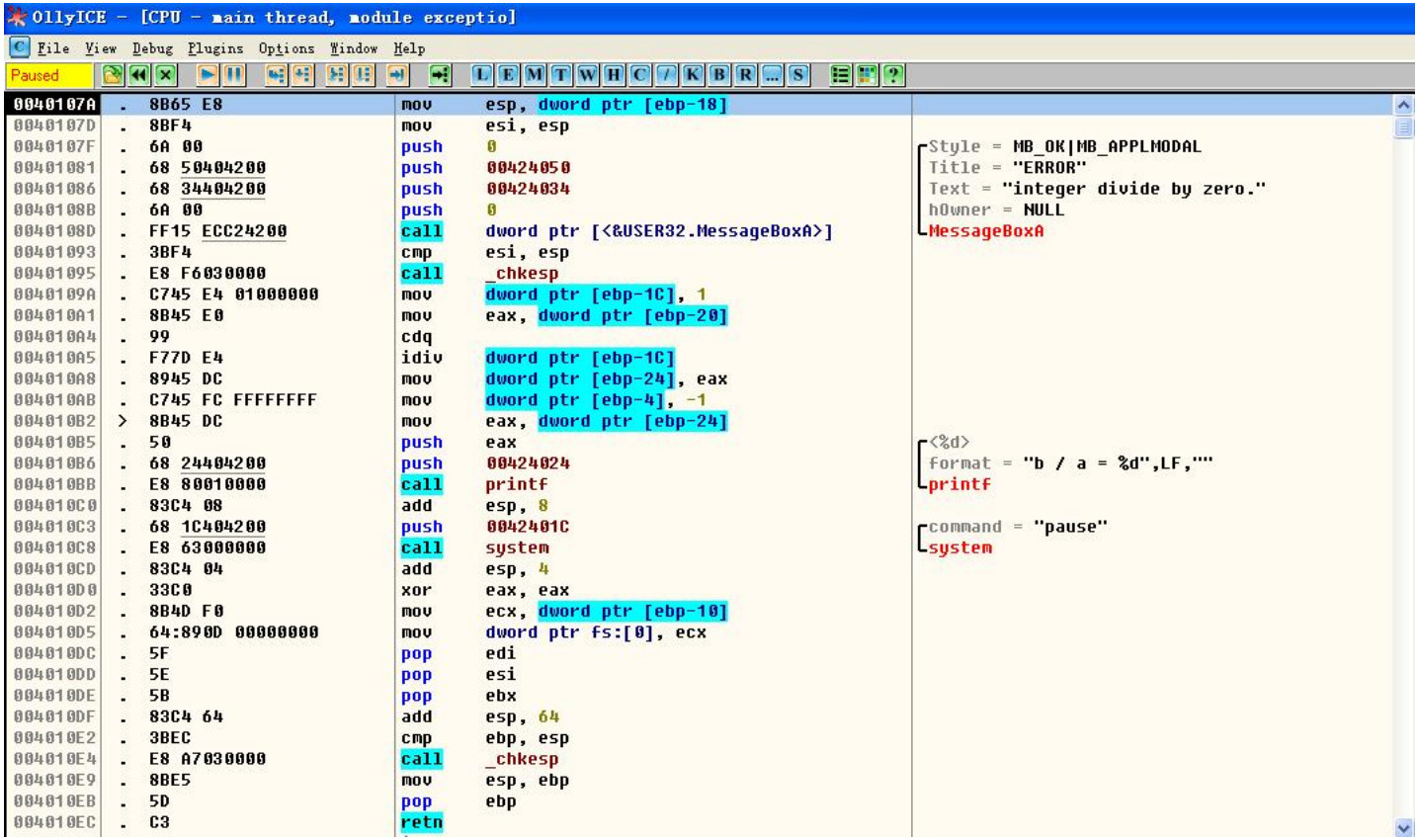
(6) SEH逆向分析 — 实验一：使用c语言添加SEH

- 本实验示例展示了如何使用__try、__except添加SEH
- 在使用关键字添加SEH时，由于编写者没有给出自定义的异常处理函数，编译器会生成一个特殊的函数，这个函数负责将_except中的代码注册为包装后的异常处理函数
- 因为被包装了，因此在SEH中，使用__try、__except没有直接显示出添加的异常处理函数，而是显示为外面包着的那个特殊函数

第七章 异常处理

(6) SEH逆向分析 — 实验一：使用c语言添加SEH

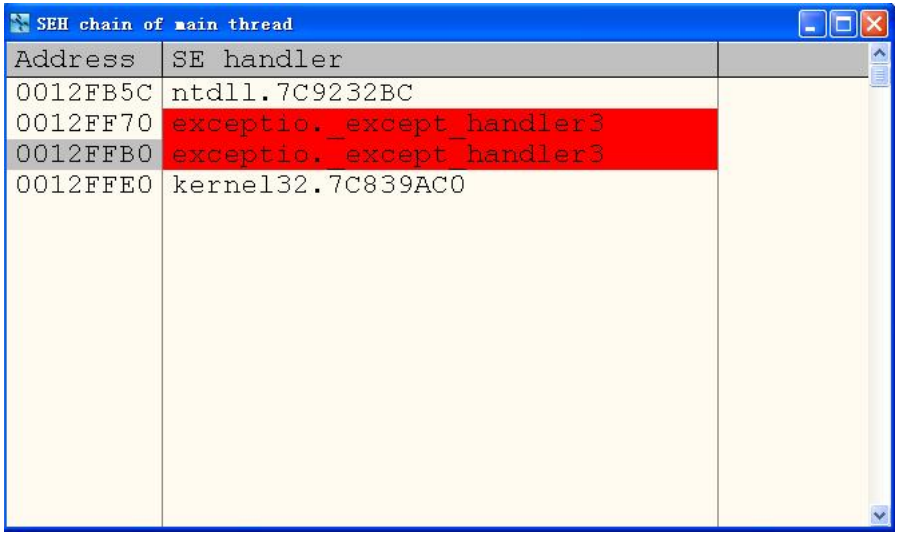
– 分析代码，发现异常处理代码起始地址为0x40107A

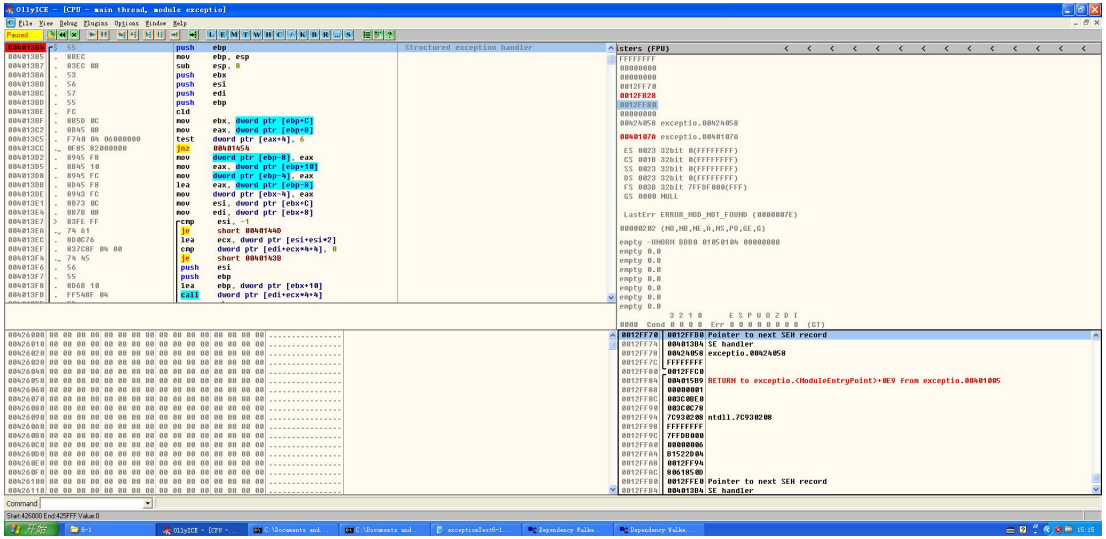


第七章 异常处理

(6) SEH逆向分析 — 实验一：使用c语言添加SEH

- 运行程序，在异常触发时查看程序的SEH链，如图
- 可见SEH链中并没有出现异常处理代码的起始地址（0x40107A），因为异常处理代码的调用被包含在了地址0x4013B4的代码中





(6) SEH逆向分析 — 实验二：使用内联汇编块注册SEH

- 实验内容
  - 实验二为编程实验，实现口令匹配程序
  - 实验要求使用内联汇编块，注册自定义的异常处理函数，在异常处理函数中实现口令匹配功能
  - 因为一般调试人员不调试SEH链，这是利用SEH实现反调功能的一种方式

(6) SEH逆向分析 — 实验二：使用内联汇编块注册SEH

示例

– 异常处理函数

```
EXCEPTION_DISPOSITION
__cdecl
_except_handler(struct _EXCEPTION_RECORD *ExceptionRecord,
void* EstabliSEHrFrame,
struct _CONTEXT *ContextRecord,
void* DispatcherContext )
{
        //异常处理函数中口令匹配逻辑
        if(strcmp(input, pw)==0){
                printf("ok, you are here. Congratulation!\n");
        }
        else{
                printf("ok, you are here. But your pw is wrong!\n");
        }
        system("pause");
        exit(0);//完成口令匹配后结束进程—否则因为没有对异常代码进行处理，所以该进程会一直在触发异常和异常处理之间循环。

        //返回，告知os继续执行异常代码
        return ExceptionContinueExecution;
}
```

(6) SEH逆向分析 — 实验二：使用内联汇编块注册SEH

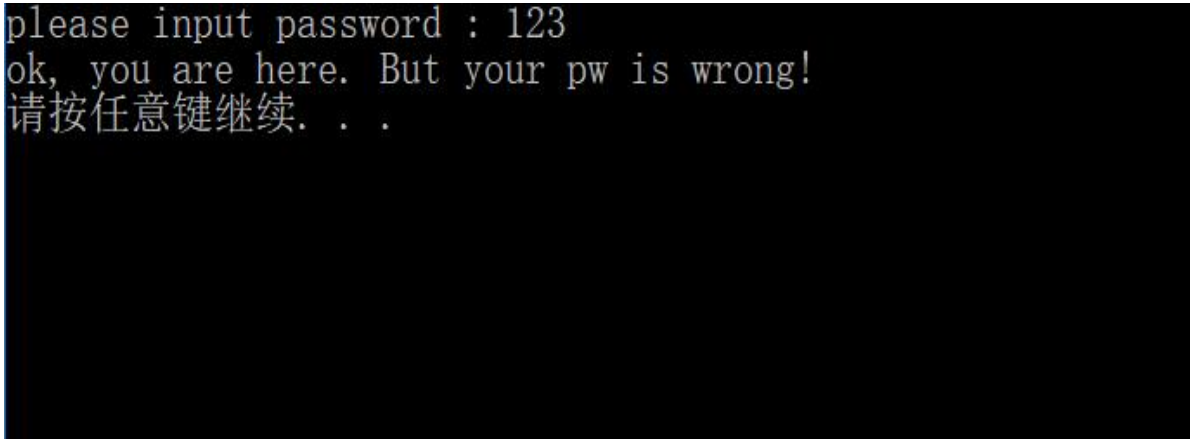
示例

– 使用内联汇编块添加SEH

```
DWORD handler =(DWORD)_except_handler;
//注册异常处理函数
__asm{
        push handler;
        push FS:[0];
        mov FS:[0], ESP;
}
```

(6) SEH逆向分析 — 实验二：使用内联汇编块注册SEH

- 运行示例程序，程序中会产生除零异常。产生异常后，程序会调用异常处理函数。异常处理函数根据口令匹配逻辑进行显示
- 输入错误口令，结果如下：



(6) SEH逆向分析 — 实验二：使用内联汇编块注册SEH

– 输入正确口令，结果如下

```
please input password : SEH
ok, you are here. Congratulation!
请按任意键继续. . .
```

(6) SEH逆向分析 — 实验二：使用内联汇编块注册SEH

- 实验二示例展示了如何利用SEH技术实现程序的反调功能
- 在利用SEH实现反调功能时，程序真正的功能隐藏在异常处理函数中实现，并故意在主逻辑中触发异常来调用异常处理函数

(6) SEH逆向分析 — 实验二：使用内联汇编块注册SEH

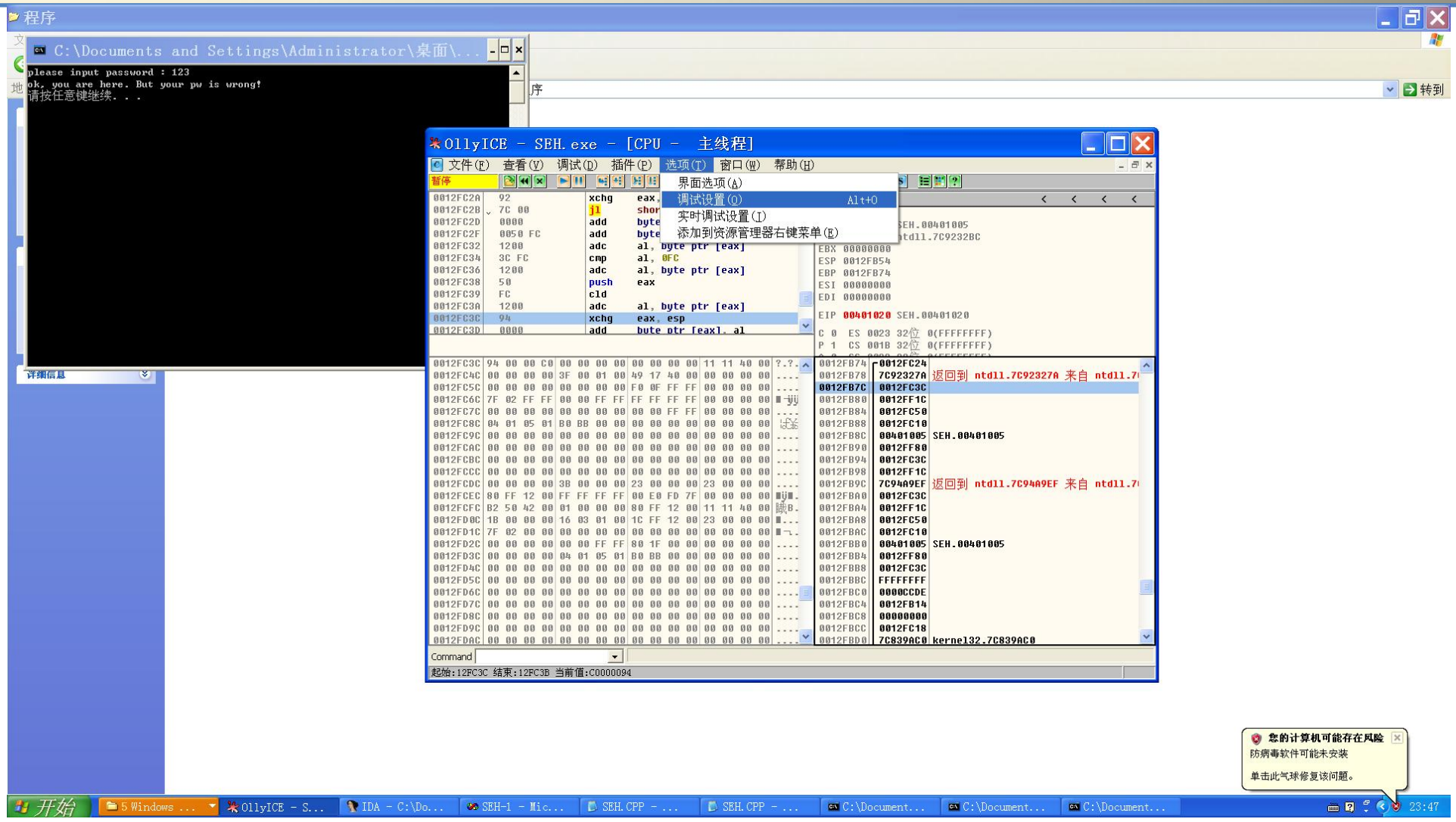
- 这是最简单的反调方式之一，一旦分析者对SEH进行分析，就暴露了，这个简单的隐藏在一般不调试的SEH链中
- 通常情况下，为了反调，程序员一般不会采用内联汇编方式主动注册异常处理函数
  - 通过内联汇编方式主动注册的异常处理函数会直接显示在SEH链中，很容易被发现
  - 而像使用_try、_except添加的异常处理代码不会直接显示在SEH链中，而是编译器编译后添加到SEH链中，隐藏性高，这样会给调试带来一定的难度，从而达到反调的效果
- 实验二这样做一方面使读者熟悉汇编代码添加SEH的操作，另一方面也方便实验三的讲解

(6) SEH逆向分析 — 实验三：SEH链分析

- 实验内容
  - 实验三是分析实验，要求通过分析实验二的示例程序获取正确口令
  - 要求通过实验了解SEH分析的基本流程，熟悉SEH安装的汇编代码，熟悉使用OD进行SEH链表查询和异常处理函数参数查看

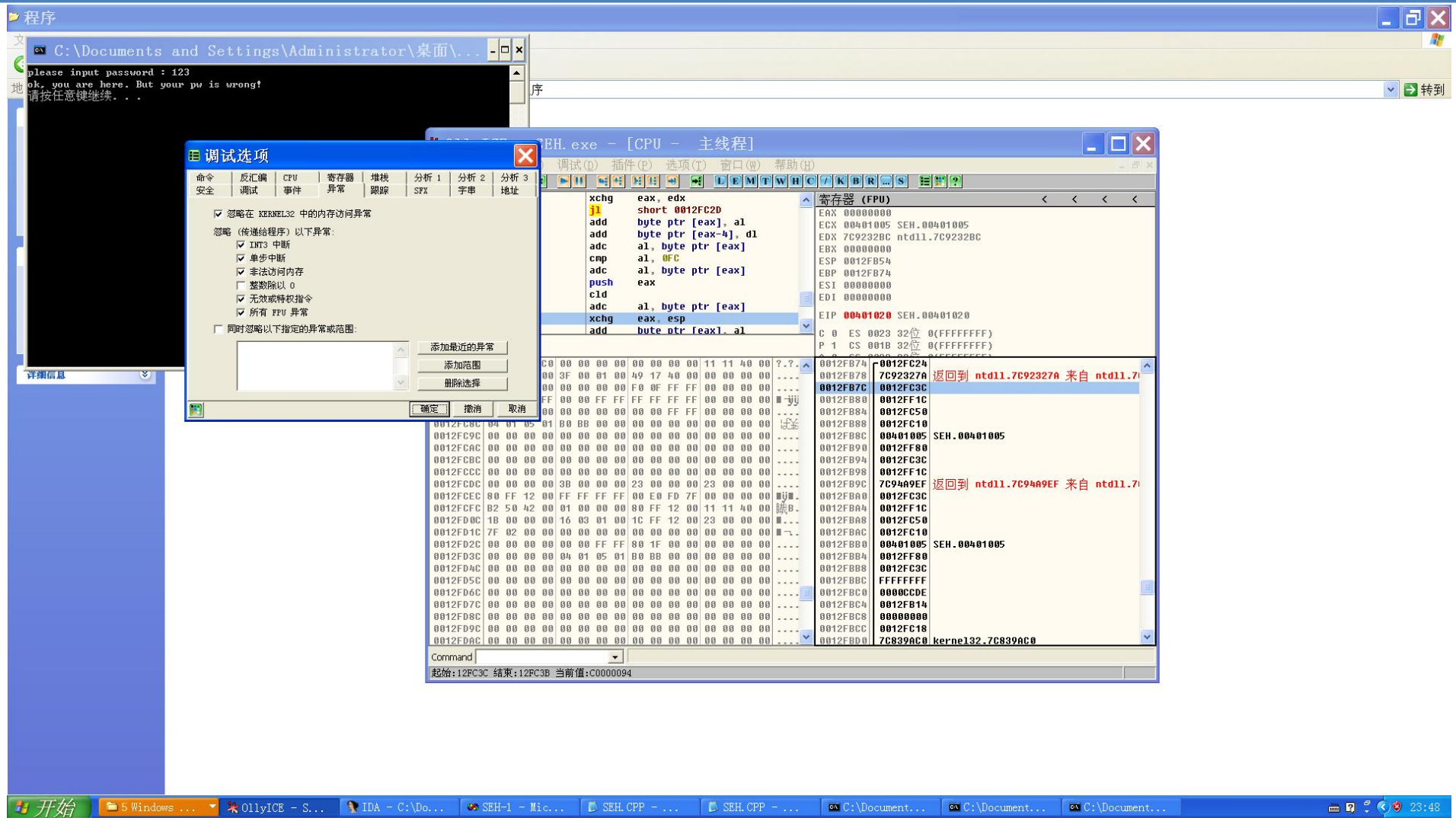
第七章 异常处理

(6) SEH逆向分析 — 实验三：SEH链分析



第七章 异常处理

(6) SEH逆向分析 — 实验三：SEH链分析

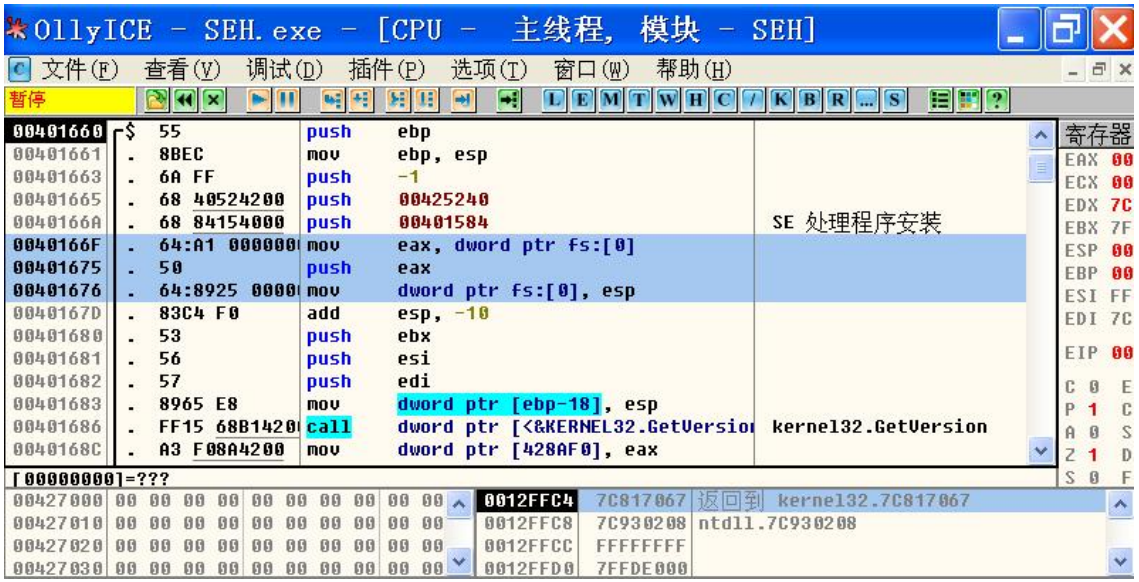


(6) SEH逆向分析 — 实验三：SEH链分析

- 示例
  - 使用OD打开实验二中的程序，可以在程序入口位置看到一个SEH添加操作，这个操作添加的并不是示例程序中自定义的SEH，而是系统提供的默认异常处理器
  - 其中，FS:[0]是SEH链表头的地址
  - SEH节点的添加是在链表头进行添加，因此会将原链表头地址赋给新节点的Next成员（即mov eax dword ptr FS:[0]），然后以新节点地址作为链表头地址(即mov dword ptr FS:[0] esp)

第七章 异常处理

(6) SEH逆向分析 — 实验三：SEH链分析



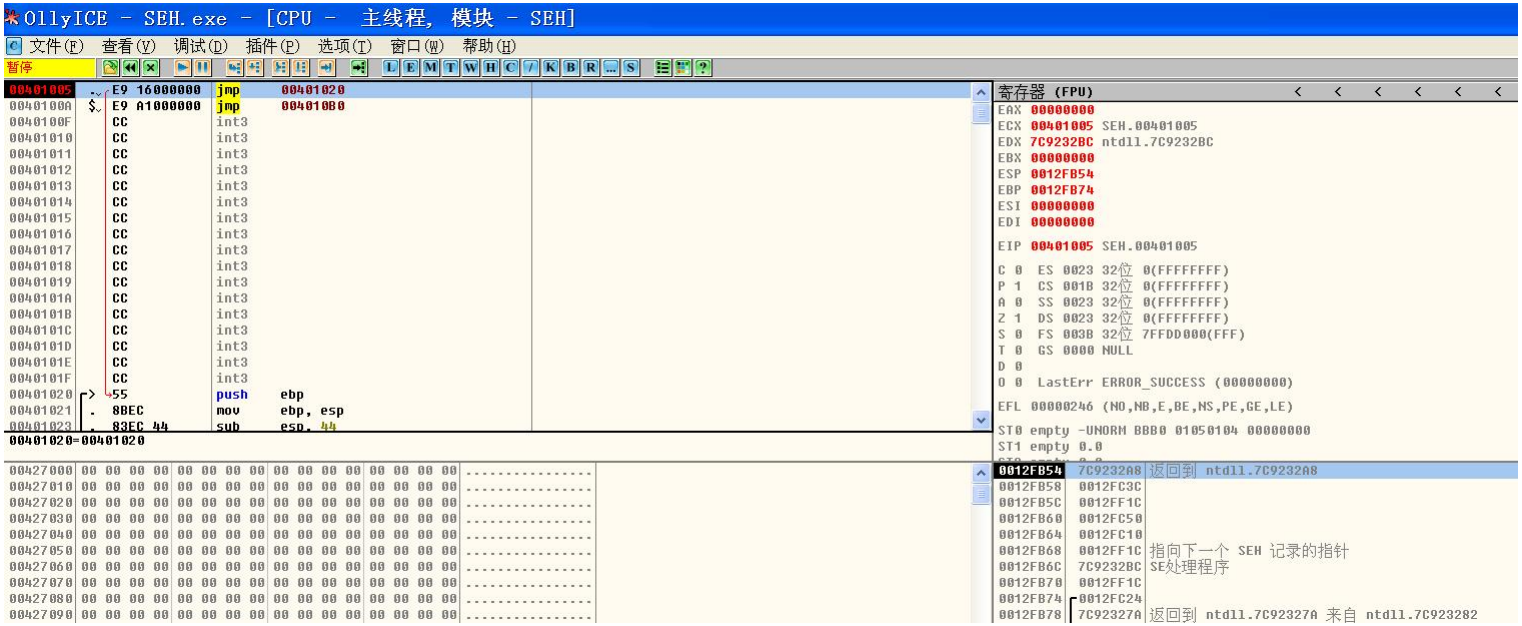
(6) SEH逆向分析 — 实验三：SEH链分析

- 将异常处理函数的地址和Next成员指向的地址压栈后（push eax），此时ESP的地址就是新节点的地址，因此第二条MOV指令将该地址作为新的表头地址赋给FS:[0] (即mov dword ptr FS:[0] esp)
- 在软件分析过程中，如果看到对FS:[0]的操作，一般都与SEH有关。如果看到与FS:[0]有关的MOV指令，则很有可能在进行SEH的添加或删除

第七章 异常处理

(6) SEH逆向分析 — 实验三：SEH链分析

- 直接运行程序，程序提示输入密码，此时并不知道密码，所以随便输入内容。（示例程序没有进行缓冲区溢出处理，输入内容不能过多）
- 继续运行程序，程序触发除零异常，该异常被调试器捕获

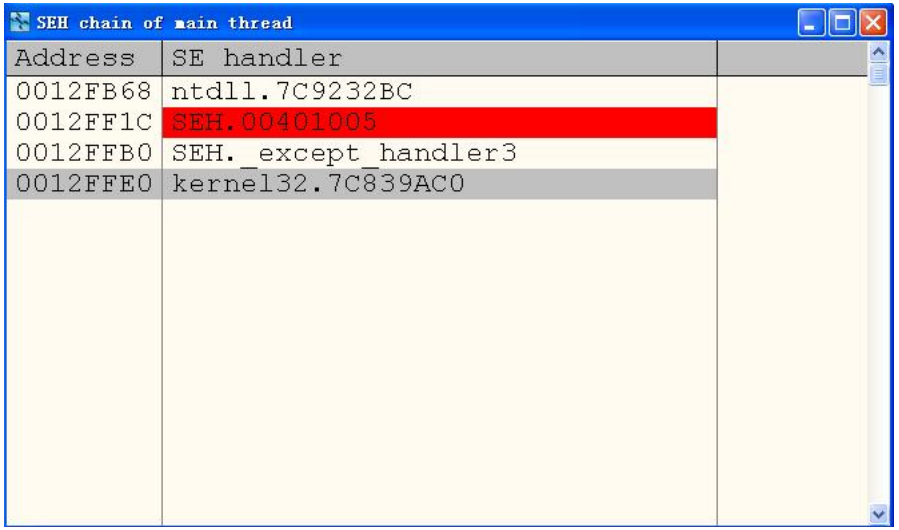


(6) SEH逆向分析 — 实验三：SEH链分析

– 此时查看SEH链，可以直接找到异常函数地址设置断点，OD中可以使用“视图->VEH/SEH链”查看当前进程的SEH链

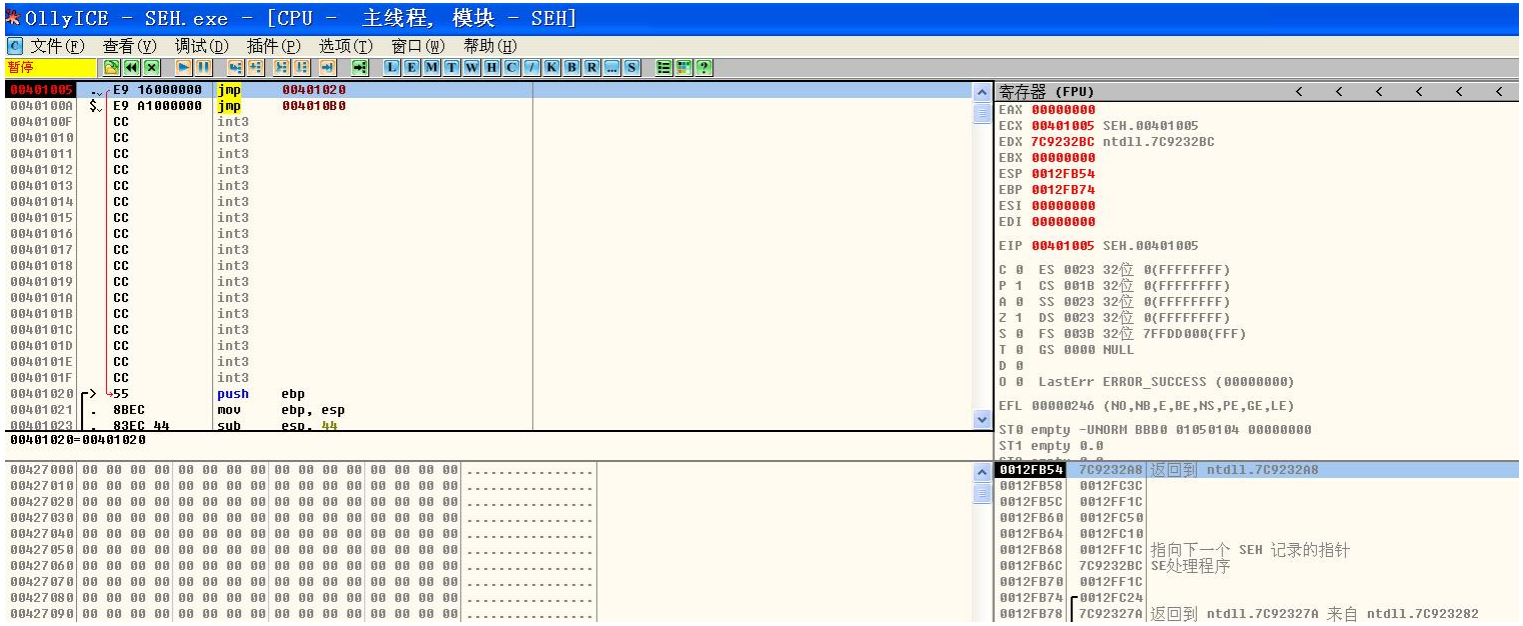
(6) SEH逆向分析 — 实验三：SEH链分析

- 因为SEH节点从链头开始添加，所以示例程序自定义的异常处理函数一定是第一个函数
- 最后一个是windows默认的异常处理函数。
- 在00401006第一个异常处理函数的地址处设置一个断点



(6) SEH逆向分析 — 实验三：SEH链分析

- 使用shift+F7/8/9将异常移交给被调程序，则被调程序会按照SEH链的顺序依次调用异常处理函数
- 因为之前在异常处理函数处设了断点，所以程序会运行至异常处理函数起始地址



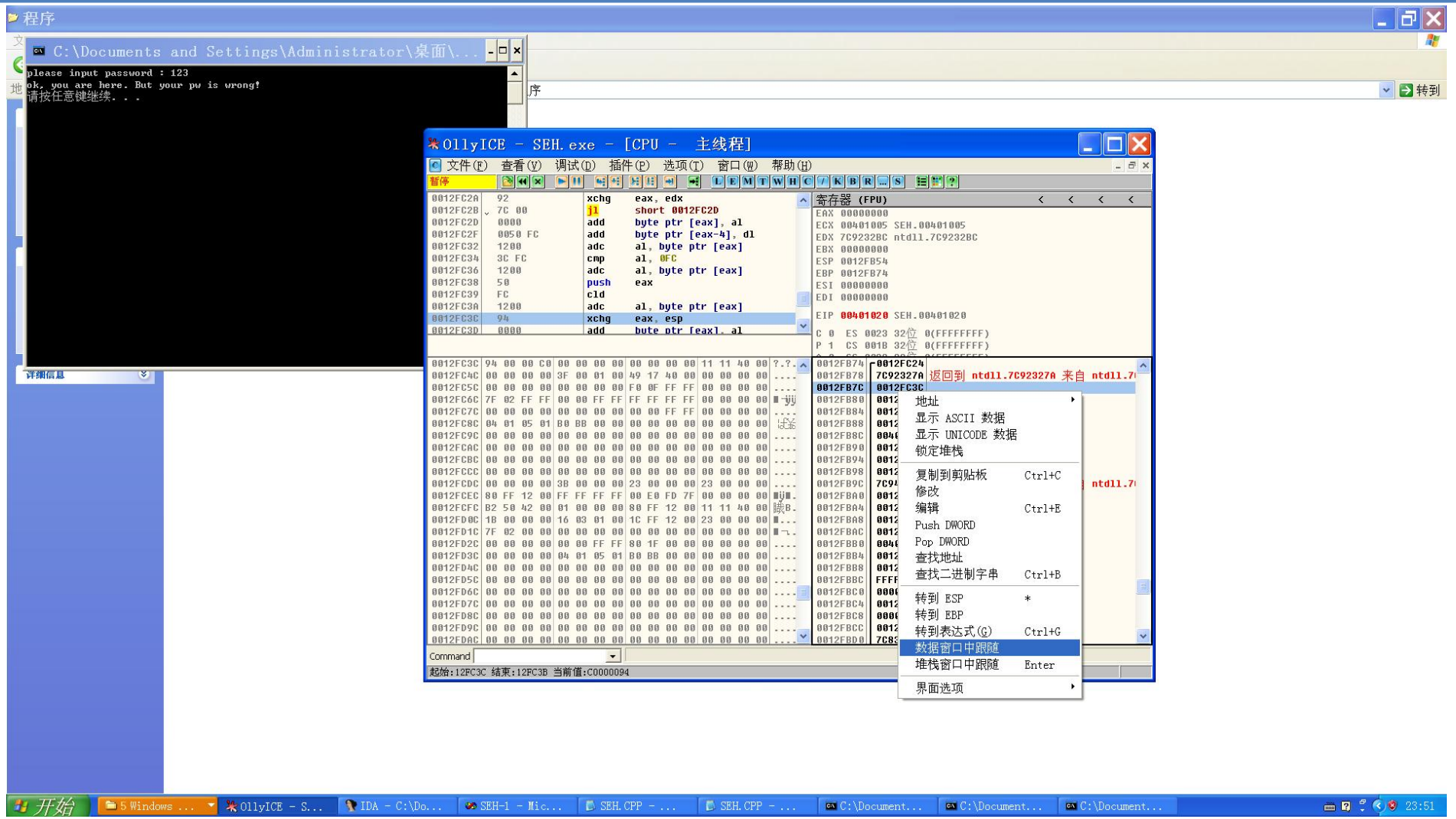
(6) SEH逆向分析 — 实验三：SEH链分析

– 此时查看堆栈即可看到异常处理函数的参数。

➢(一般情况异常处理函数的参数分析意义不大，通常直接对函数代码进行分析，这里仅为了加深理解)

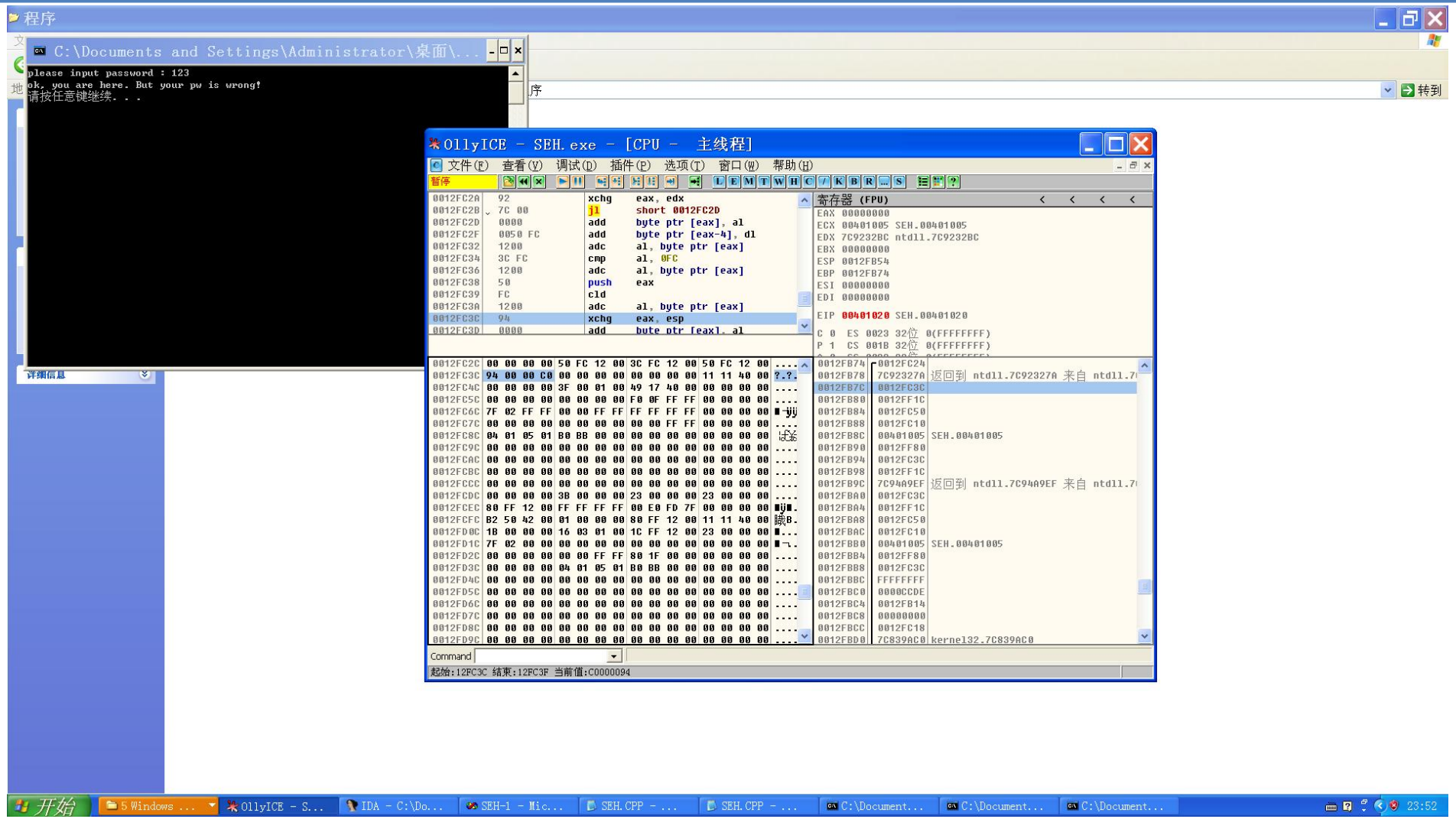
第七章 异常处理

(6) SEH逆向分析 — 实验三：SEH链分析



第七章 异常处理

(6) SEH逆向分析 — 实验三：SEH链分析



(6) SEH逆向分析 — 实验三：SEH链分析

– 下面是一个异常处理函数的原型

➢异常处理函数有四个参数，这四个参数用来传递与异常相关的信息，包括异常类型、发生异常的代码地址、异常发生时CPU寄存器的状态等

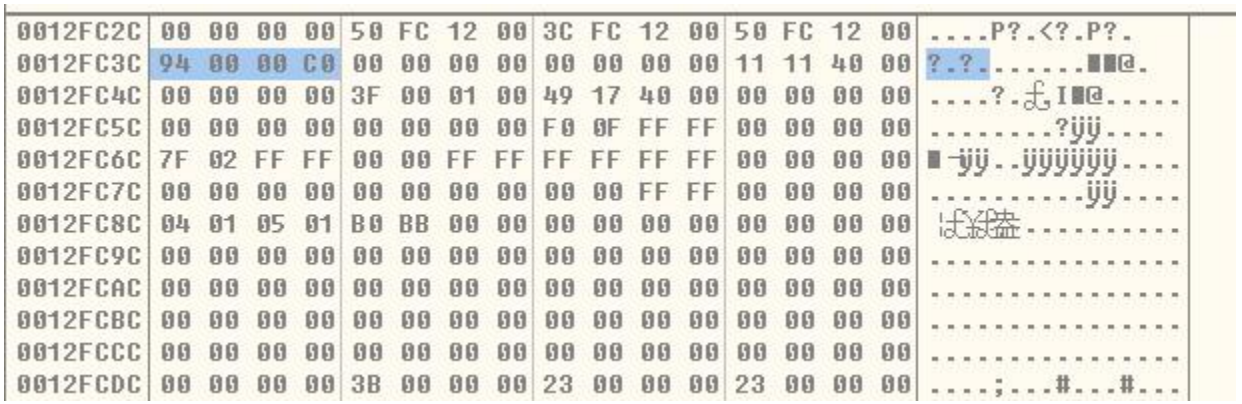
```
EXCEPTION_DISPOSITION _except_handler{
        EXCEPTION_RECORD *pRecord,
        EXCEPTION_REGISTRATION_RECORD *pFrame,
        CONTEXT *pContext,
        PVOID pValue
};
```

(6) SEH逆向分析 — 实验三：SEH链分析

- 第一个参数是指向EXPCEPTION_RECORD结构体的指针，根据指向的地址可以查看结构体中的数据
  - 第一个参数其中的第一个DWORD成员指出了异常类型代码为0xC0000094（即STATS_DIVIDE_BY_ZERO）
- 第二个参数是指向EXCEPTION_REGISTRATION_RECORD结构体的指针，指向SEH链表
- 第三个参数是指向CONTEXT结构体的指针，该结构体描述了异常发生时寄存器状态、异常发生的地址等上下文信息
- 第四个参数仅供系统内部使用

(6) SEH逆向分析 — 实验三：SEH链分析

– 异常处理函数第一个参数是类型

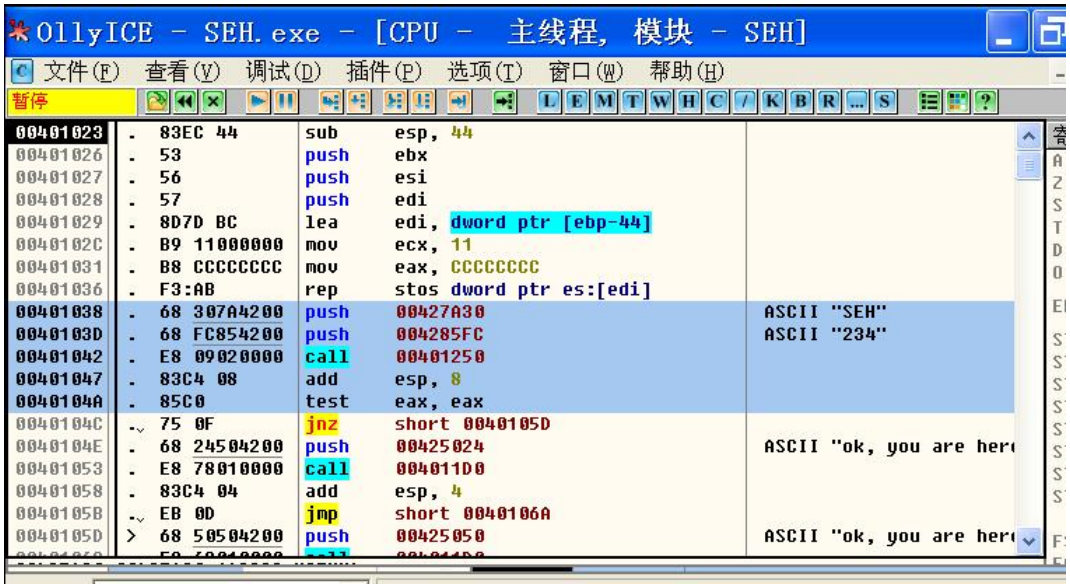


```
0012FC2C  00 00 00 00 50 FC 12 00 3C FC 12 00 50 FC 12 00  ....P?.<?.P?.
0012FC3C  94 00 00 C0 00 00 00 00 00 00 00 00 11 11 40 00  ?.?.........@.
0012FC4C  00 00 00 00 3F 00 01 00 49 17 40 00 00 00 00 00  ....?...I.@.....
0012FC5C  00 00 00 00 00 00 00 00 F0 0F FF FF 00 00 00 00  ........?...
0012FC6C  7F 02 FF FF 00 00 FF FF FF FF FF FF 00 00 00 00  ..  ..
0012FC7C  00 00 00 00 00 00 00 00 00 00 FF FF 00 00 00 00  ..........  ....
0012FC8C  04 01 05 01 B0 BB 00 00 00 00 00 00 00 00 00 00  ..........
0012FC9C  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0012FCAC  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0012FCBC  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0012FCCC  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0012FCDC  00 00 00 00 3B 00 00 00 23 00 00 00 23 00 00 00  ....;...#...#...
```

```
#define STATUS_FLOAT_INEXACT_RESULT         ((DWORD   )0xC000008FL)
#define STATUS_FLOAT_INVALID_OPERATION      ((DWORD   )0xC0000090L)
#define STATUS_FLOAT_OVERFLOW               ((DWORD   )0xC0000091L)
#define STATUS_FLOAT_STACK_CHECK            ((DWORD   )0xC0000092L)
#define STATUS_FLOAT_UNDERFLOW              ((DWORD   )0xC0000093L)
#define STATUS_INTEGER_DIVIDE_BY_ZERO       ((DWORD   )0xC0000094L)
#define STATUS_INTEGER_OVERFLOW             ((DWORD   )0xC0000095L)
#define STATUS_PRIVILEGED_INSTRUCTION       ((DWORD   )0xC0000096L)
```

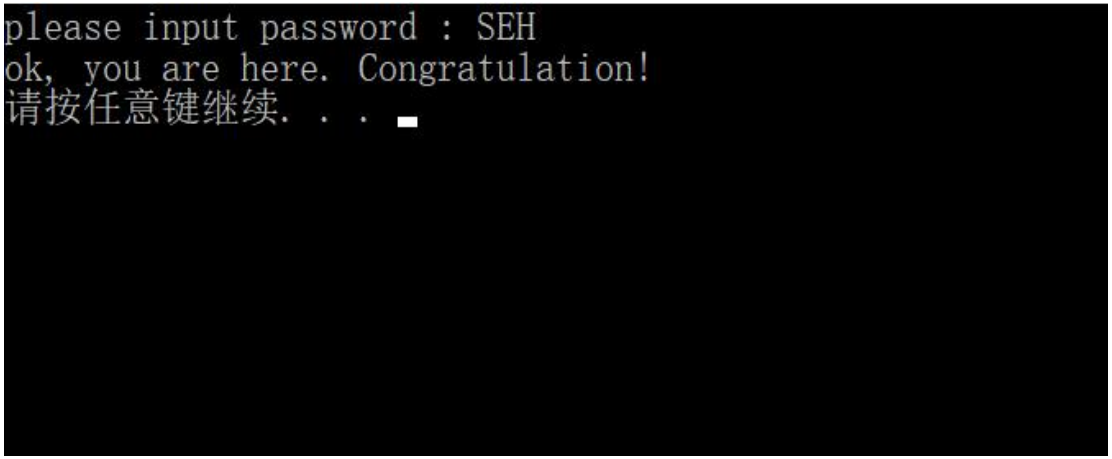
(6) SEH逆向分析 — 实验三：SEH链分析

– 查看异常处理函数代码，可以直接看到口令比对代码。“123”是输入，所以“SEH”就是正确的口令了。



(6) SEH逆向分析 — 实验三：SEH链分析

– 直接运行程序，输入“SEH”即可得到正确结果



(6) 逆向分析 — 实验四：使用UnhandledExceptionFilter实现反调

- 实验内容
  - 实验四为编程实验，实现口令匹配程序
  - 实验要求使用SetUnhandledExceptionFilter注册异常处理函数，在异常处理函数中实现口令匹配功能，也就是利用UnhandledExceptionFilter实现反调功能

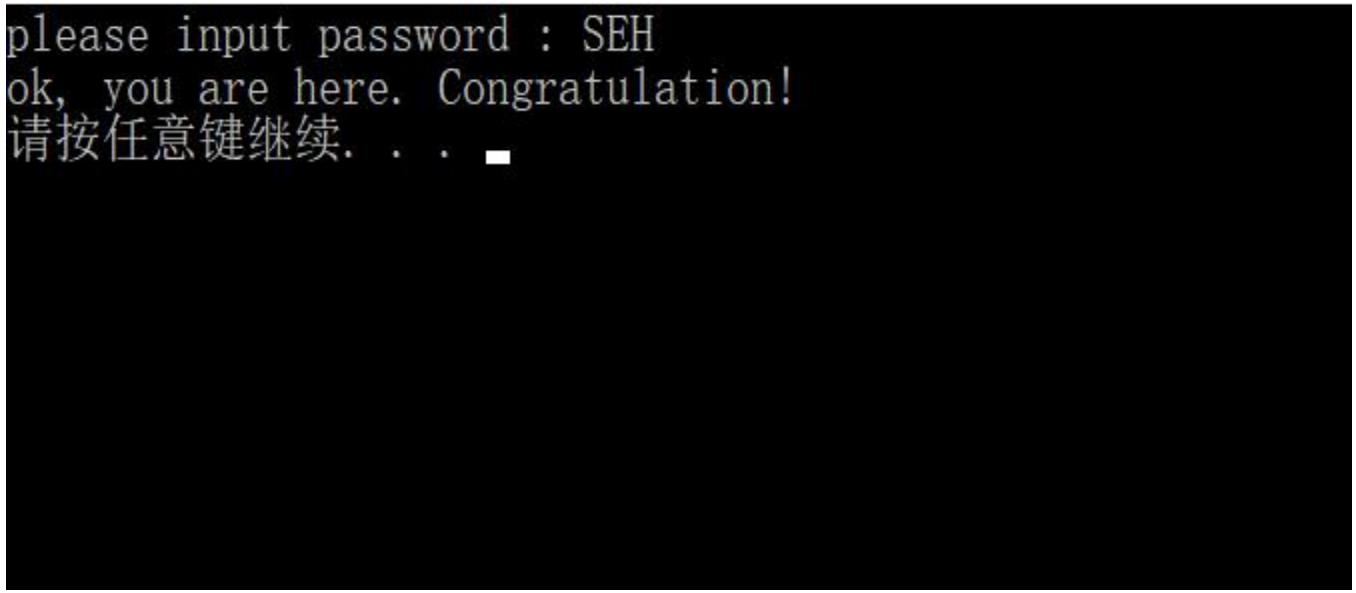
（6）逆向分析 — 实验四：使用UnhandledExceptionFilter实现反调

- 示例
  - 示例程序运行效果与实验二效果相同。输入错误口令，结果如下：

```
please input password : 123
ok, you are here. But your pw is wrong!
请按任意键继续. . .
```

（6）逆向分析 — 实验四：使用UnhandledExceptionFilter实现反调

– 输入正确口令，结果如下：

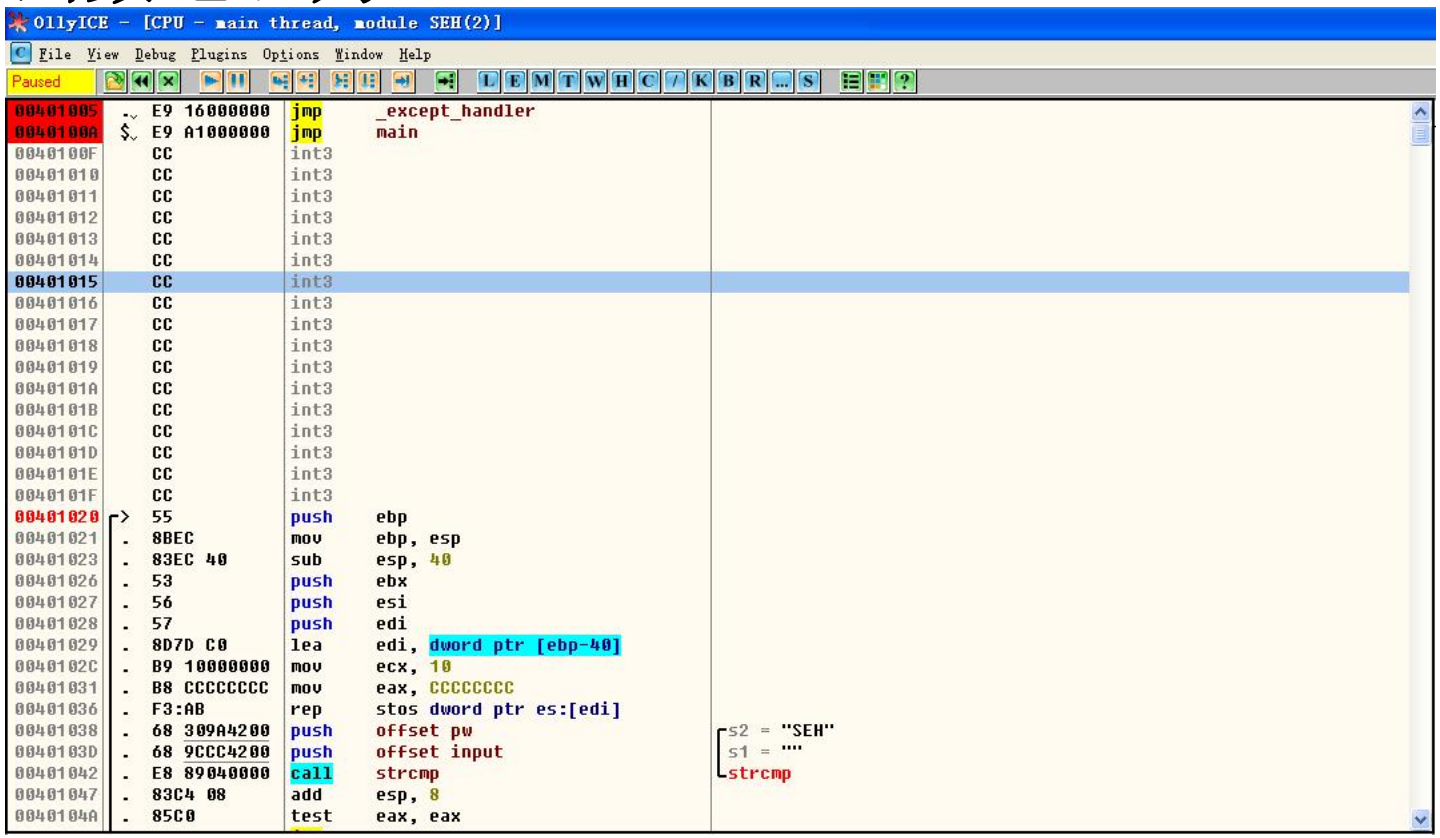


（6）逆向分析 — 实验四：使用UnhandledExceptionFilter实现反调

- 使用OD对示例程序进行分析时,因为SetUnhandledExceptionFilter函数在调试状态下不起作用，所以不会调用自定义的异常处理函数，而且也不会在SEH链中看到自定义的异常处理函数

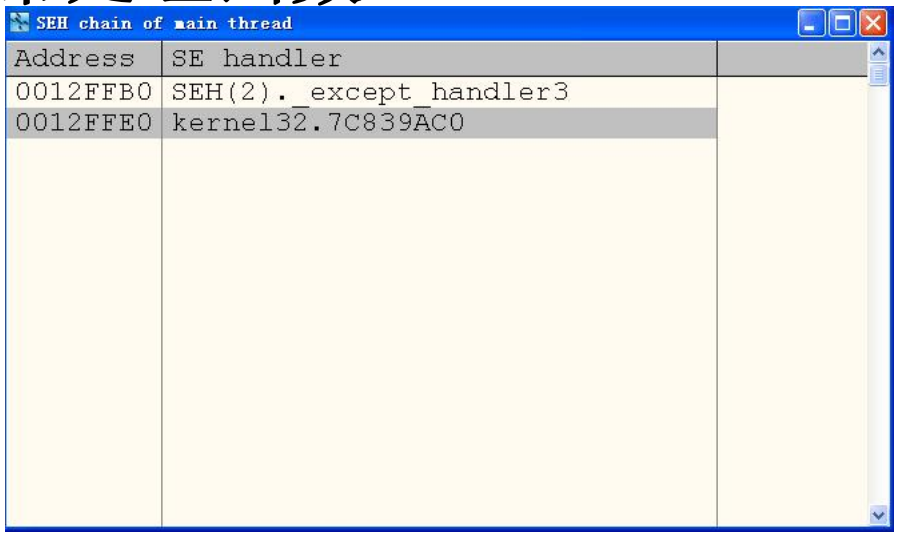
（6）逆向分析 — 实验四：使用UnhandledExceptionFilter实现反调

– 双击字符串所在行跳转到其所在代码，可见异常处理函数地址为0x401020



（6）逆向分析 — 实验四：使用UnhandledExceptionFilter实现反调

- 运行程序，在异常触发时查看SEH链
- 可见0x401020地址不在SEH链中
- 继续运行程序，按shift+F9/F8/F7，调试器会提示应用程序无法处理异常，因为调试状态下不会调用自定义的异常处理函数



谢　谢！