软件安全实验

北京邮电大学信息安全中心
张淼
zhangmiao@bupt.edu.cn

虚函数攻击及SEH

- 虚函数攻击

- SEH攻击

一、虚函数攻击

- 虚函数攻击
- SEH攻击

虚函数攻击

多态是面向对象的一个重要特性，在C++中，这个特性主要靠对虚函数的动态调用来实现。

由于我们的重点是讲解利用虚函数进行攻击，所以对虚函数和动态联编等概念不太了解的同学可以课下学习一下。

在仅仅关注漏洞利用的前提下，我们可以简单地把虚函数和虚表理解为一下几个要点。

1）C++类的成员函数在声明时，若使用关键字virtual进行修饰，则被称为虚函数。

虚函数攻击

2）一个类中可能有很多个虚函数。

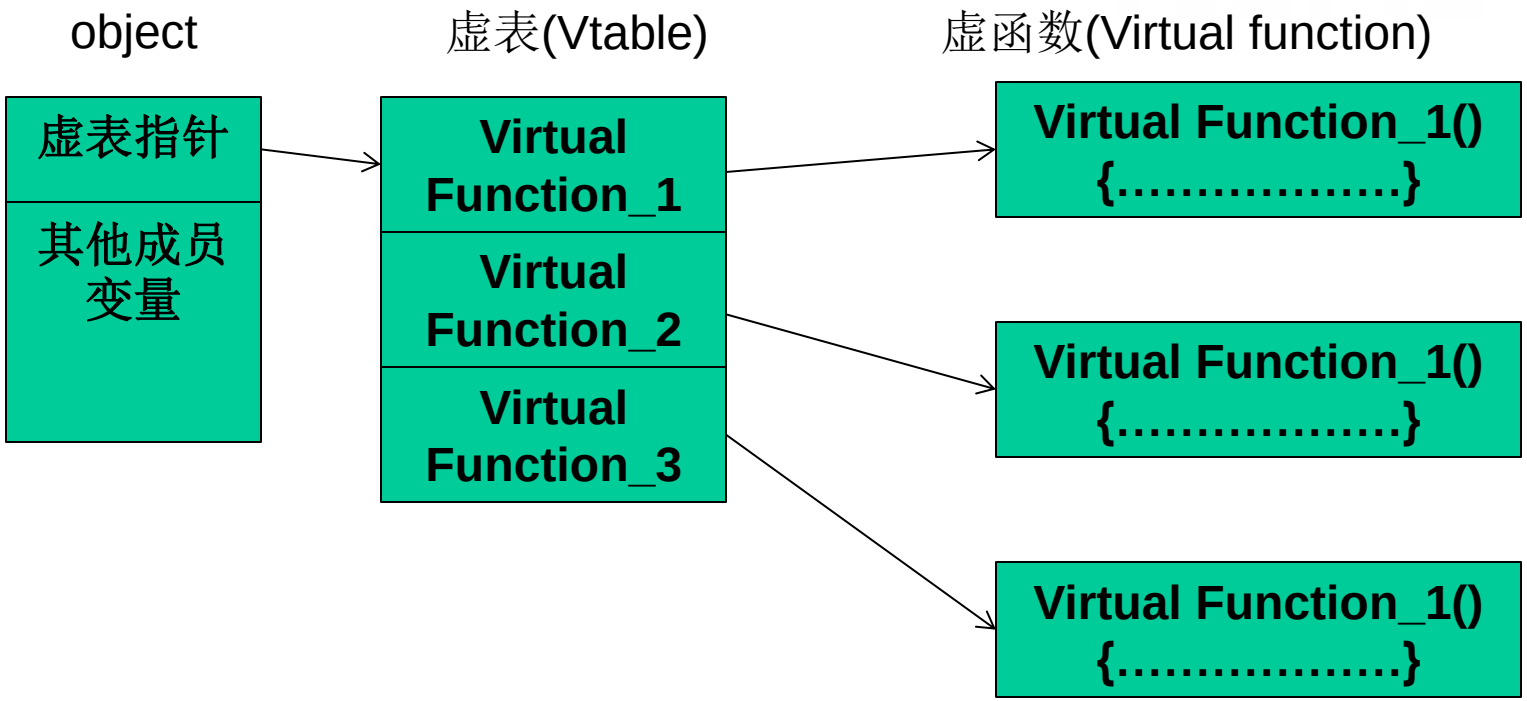
3）虚函数的入口地址被统一保存在虚表(Vtable)中。

4）对象在使用虚函数时，先通过虚表指针找到虚表，然后从虚表中取出最终的函数入口地址进行调用。

5）虚表指针保存在对象的内存空间中，紧接着虚表指针的是其他成员变量。

6）虚函数只有通过对象指针的引用才能显示出其动态调用的特性。

虚函数攻击



虚函数的实现

虚函数攻击

我们来写一个利用虚函数执行shellcode的程序

```
char[] shellcode="……";
class vf
{
        public:
        char buf[200];
        virtual void test(void)
        {
                cout<<"Class Vtable::test()"<<endl;
        }
};
vf overflow, *p;
```

这段程序声明了一个类，具有buf和虚函数test，声明它的实例overflow和指针p。

虚函数攻击

```
void main(void)
{
        LoadLibrary("user32.dll");
        char * p_vtable;
        p_vtable=overflow.buf-4;//point to virtual
table
        __asm int 3
        //reset fake virtual table to 0x0042E430
        //the address may need to adjusted via
runtime debug
```

虚函数攻击

```
        p_vtable[0]=0x30;
        p_vtable[1]=0xE4;
        p_vtable[2]=0x42;
        p_vtable[3]=0x00;
        strcpy(overflow.buf,shellcode1);//set fake
                virtual function pointer
        p=&overflow;
        p->test();
}
```

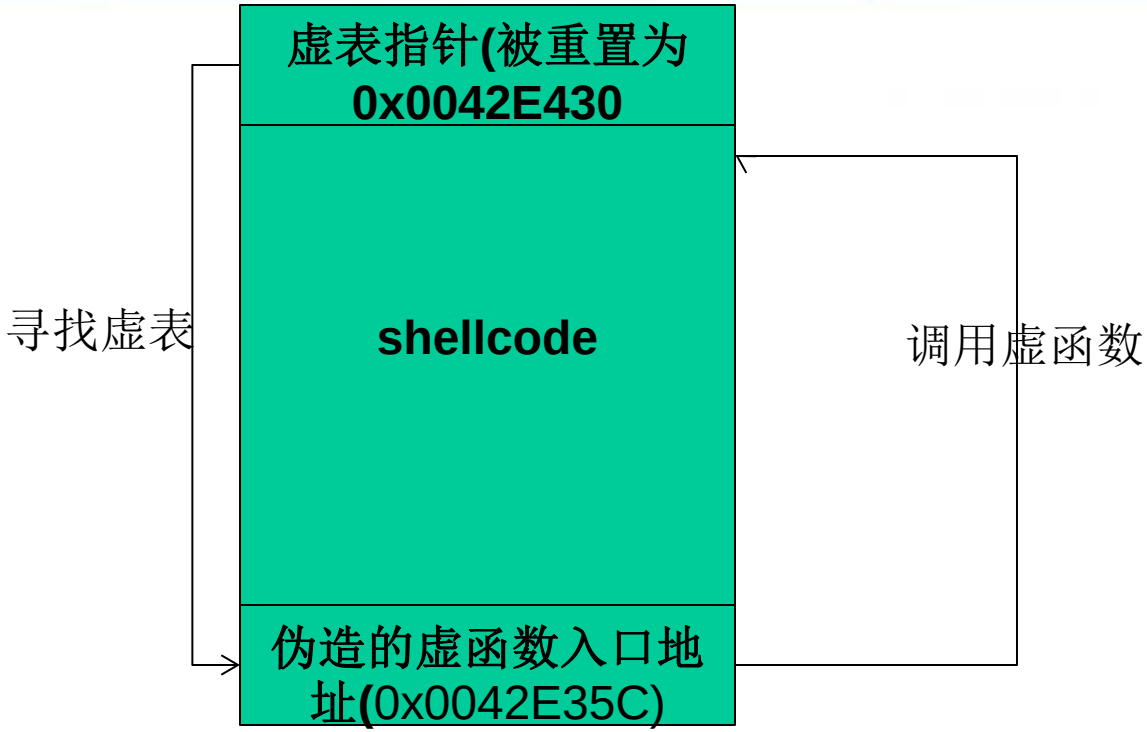
虚函数攻击

1）虚表指针位于成员变量char buf[200]之前，程序中通过p_vtable=overflow.buf-4定位到这个指针。

2）修改虚表指针指向缓冲区的0x0042E430处（第一次得不到，需要通过调试得到）。

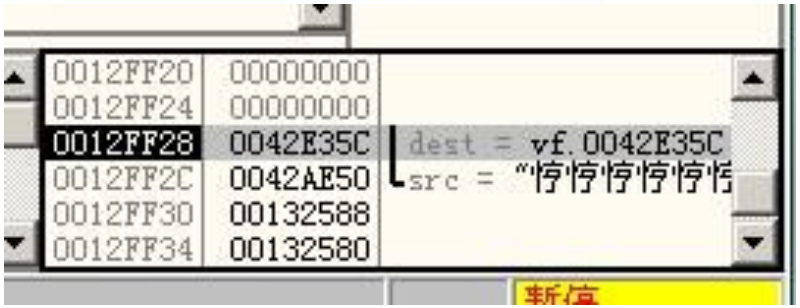
3）程序执行到p->test()时，将按照伪造的虚函数指针去0x0042E430寻找虚表，这里正好是缓冲区里shellcode的末尾。在这里填上shellcode的起始位置0x0042E35C作为伪造的虚函数入口地址，程序将最终跳去执行shellcode。

虚函数攻击



利用虚表的示意图

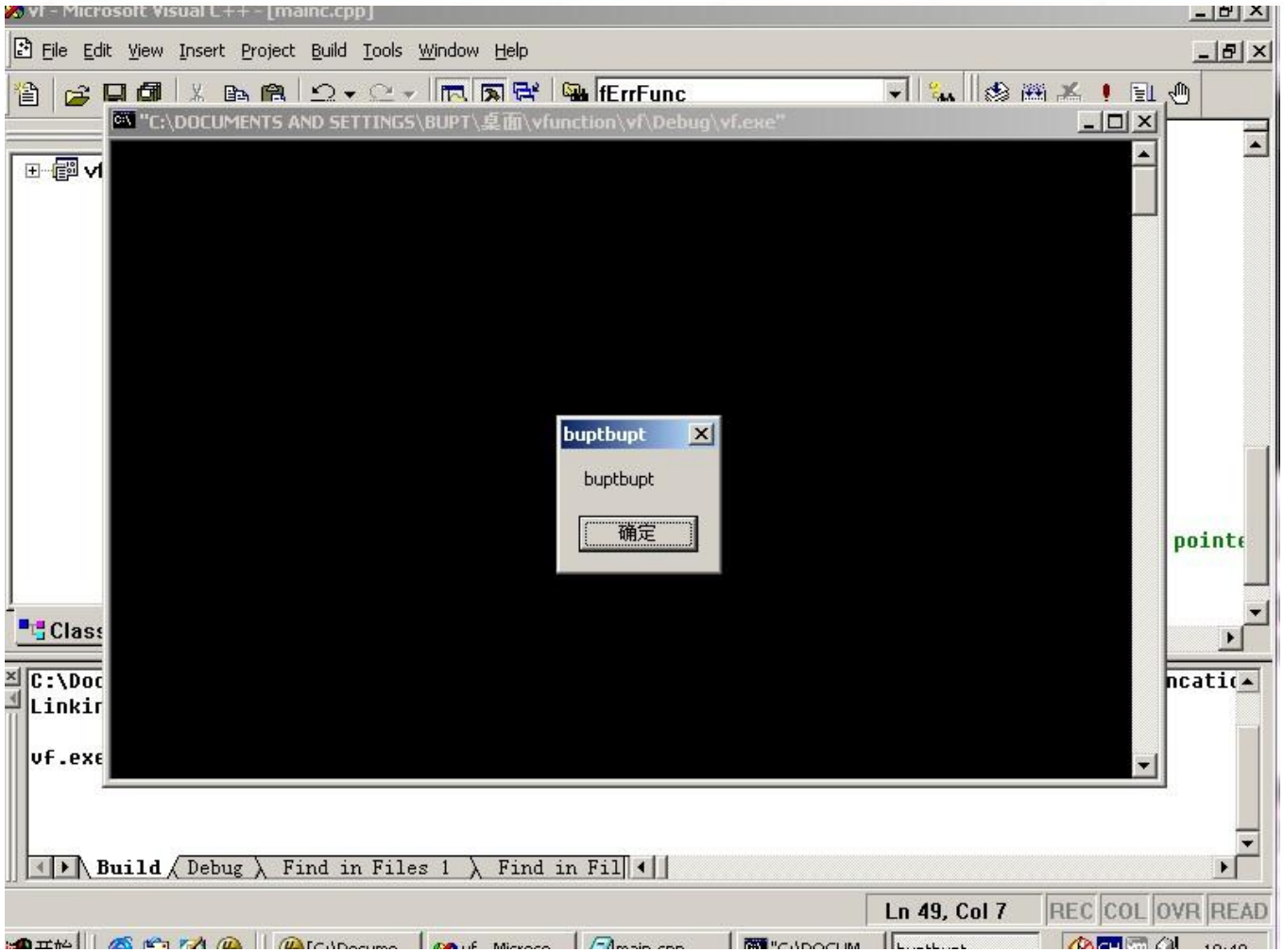
虚函数攻击



Int3触发后我们可以在strcpy函数上看到shellcode的起始地址0x0042E35C,然后可以计算出shellcode的末尾后四个字节地址是0X0042E430。

得到这两个地址后，修改程序，将vtable和shellcode那两个部分进行修改，然后就可以运行程序了

虚函数攻击



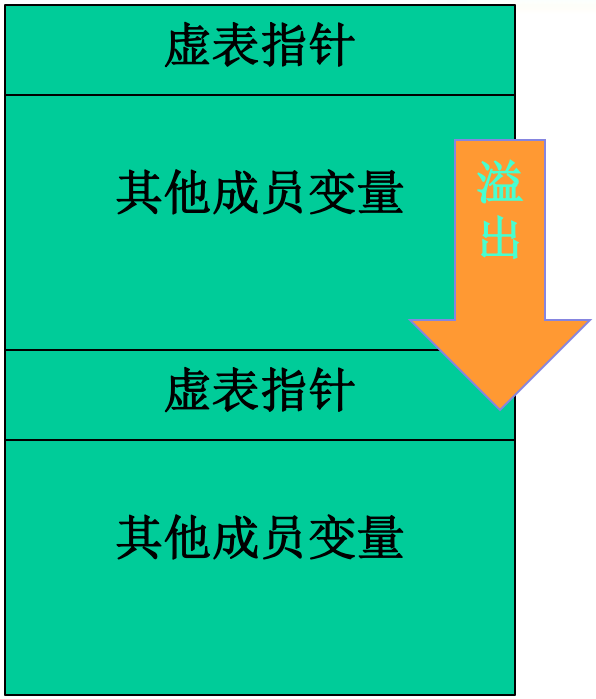
运行程序后可以看到这个结果。

虚函数攻击

由于虚表指针位于成员变量之前，溢出只能向后覆盖数据，所以很可惜这种利用方式在“栈溢出”场景下有一定局限性。

当然，如果内存中存在多个对象且能够溢出到下一个对象空间中去，“连续性覆盖”还是有攻击的机会的，比如下图这种情况。

虚函数攻击



溢出邻接对象的虚表

虚函数攻击

说到这里，大家应该明白所谓的虚函数、面向对象在指令层次上和C语言是没有质的区别的，以漏洞利用的眼光来看这些东西，其实都是指针。

- 虚函数攻击
- SEH攻击

S.E.H概述

操作系统或程序在运行时，难免会遇到各种各样的错误，如除0、非法内存访问、文件打开错误、内存不足、此版读写错误、外设操作失败等。

S.E.H概述

为了保证系统在遇到错误时不至于崩溃，仍能够见状稳定地继续运行下去，Windows会对运行在其中的程序提供一次补救的机会来处理错误，这种机制就是异常处理机制。

S.E.H概述

S.E.H即异常处理结构体(Structure Exception Handler)

它是Windows异常处理机制所采用的重要数据结构。

每个S.E.H包含两个DWORD指针：S.E.H链表指针和异常处理函数句柄，共8个字节，如图所示。

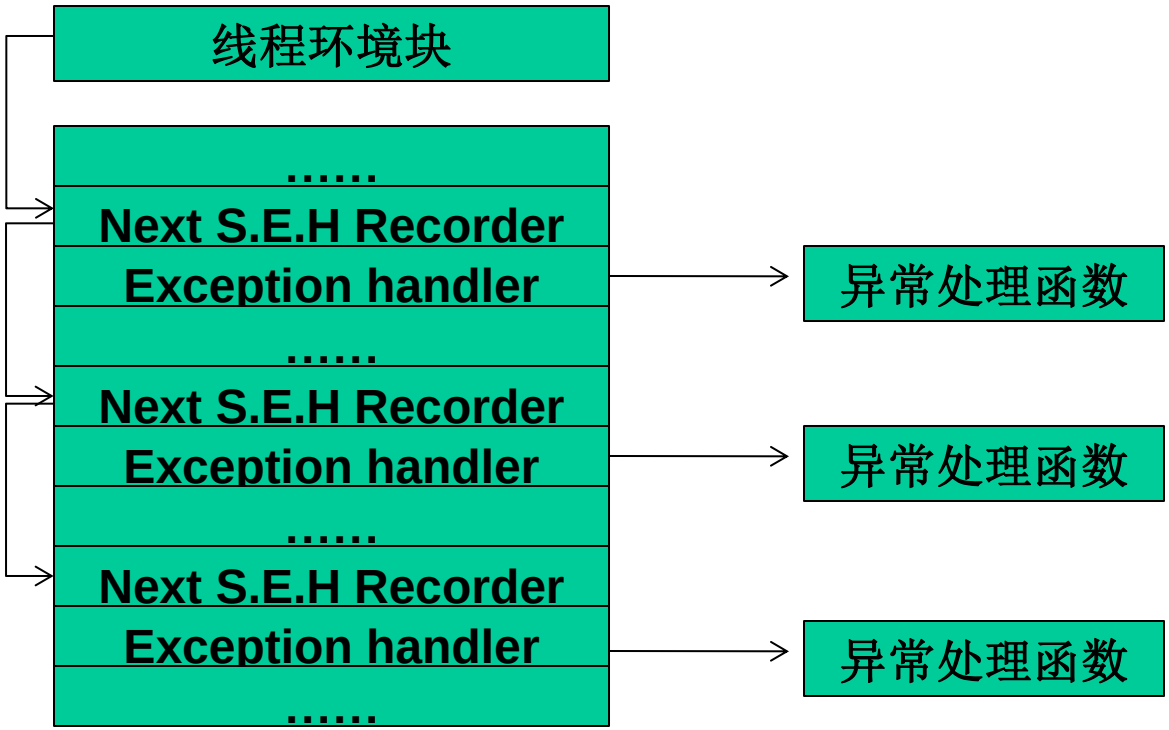
S.E.H概述

DWORD:Next S.E.H recorder
DWORD:Exception handler

S.E.H结构体

S.E.H概述

作为对S.E.H的初步了解，我们现在需要知道以下几个要点，S.E.H链表如下图所示。



S.E.H概述

1）S.E.H结构体存放在系统栈中。

2）当线程初始化时，会自动向栈中安装一个S.E.H，作为线程默认的异常处理。

3）如果程序源代码中使用了__try{}__except{}或者Assert宏等异常处理机制，编译器将最终通过向当前函数栈帧中安装一个S.E.H来实现异常处理。

4）栈中一般会同时存在多个S.E.H。

S.E.H概述

5）栈中的多个S.E.H通过链表指针在栈内有栈顶向栈底串成单向链表，位于链表最顶端的S.E.H通过T.E.B(线程环境块)0字节偏移处的指针标识。

6）当异常发生时，操作系统会中断程序，并首先从T.E.B的0字节偏移处取出距离栈顶最近的S.E.H，使用异常处理函数句柄所指向的代码来处理异常。

S.E.H概述

7）当离“事故现场”最近的异常处理函数运行失败时，将顺着S.E.H链表依次尝试其他的异常处理函数。

8）如果程序安装的所有异常处理函数都不能处理，系统将采用默认的异常处理函数。通常，这个函数会弹出一个对话框，然后强制关闭程序。

S.E.H概述

上面所叙述的只是对S.E.H进行了一下简单的概述，省略了很多细节。

例如，系统对异常处理函数的调用可能不止一次；

对同一个函数内的多个__try或者嵌套的__try需要进行S.E.H展开操作；

执行异常处理函数前会进行若干判定操作等等。

S.E.H概述

从程序设计的角度来讲，S.E.H就是在系统关闭程序之前，给程序一个执行预先设定的回调函数的机会。

大概明白了S.E.H的工作原理之后，同学们能否设计出利用S.E.H攻击的思路呢？

S.E.H概述

我们一步一步来思考。

首先，S.E.H存放在栈内，我们学过的栈溢出漏洞可能会派上用场。

其次，我们可以精心制造出溢出数据来把S.E.H的异常处理的函数地址替换为shellcode的起始地址。

S.E.H概述

再次，溢出后错误的栈帧或堆块数据往往会触发异常，或者我们能查到哪些能触发异常的片段。

最后，当Windows开始处理溢出后的异常时，会调用什么呢？对，就是我们的shellcode，它会把shellcode当作异常函数来处理异常。

S.E.H概述

以上就是我们利用S.E.H来进行攻击的思路。

接下来我们来实验一下，来在自己写的有漏洞的小程序上，在栈溢出中利用S.E.H执行shellcode。

S.E.H概述

我们先来看一下我们的源代码

```
#include <windows.h>
char   shellcode[]="\x90\x90…";// 由 于 我 们 要 得 到
  //shellcode起始位置，S.E.H地址等信息，所以我们
//先用0x90来进行填充
```

S.E.H概述

```
DWORD MyExceptionhandler(void)
{
        printf("got an exception,press Enter to kill process!\n");
        getchar();
        ExitProcess(1);
}//这是我们自己编写的异常处理函数
```

S.E.H概述

```
void test(char* input)
{
  char buf[200];
  int zero=0;
  __asm int 3 //int 3指令可以中断进程开始调试
  __try
  {
        strcpy(buf,input);//产生栈溢出。未完，接下页
```

S.E.H概述

```
        zero=4/zero;//除0产生异常
    }
    __except(MyExceptionhandler()){}
}
int main(){
        test(shellcode);
        return 0;
}
```

S.E.H概述

对代码进行一下简要的解释。

1）函数test中存在典型的栈溢出漏洞。

2）__try{}会在test的函数栈帧中安装一个S.E.H结构。

3）__try中的除0操作会产生一个异常

4）当strcpy操作没有产生溢出时，除0操作的异常将最终被MyExceptionHandler函数处理。

S.E.H概述

对代码进行一下简要的解释。

5）当strcpy操作产生溢出，并精确地将栈帧中的S.E.H异常处理句柄修改为shellcode的入口地址时，操作系统将会错误地使用shellcode去处理除0异常，代码植入成功。

S.E.H概述

对代码进行一下简要的解释。

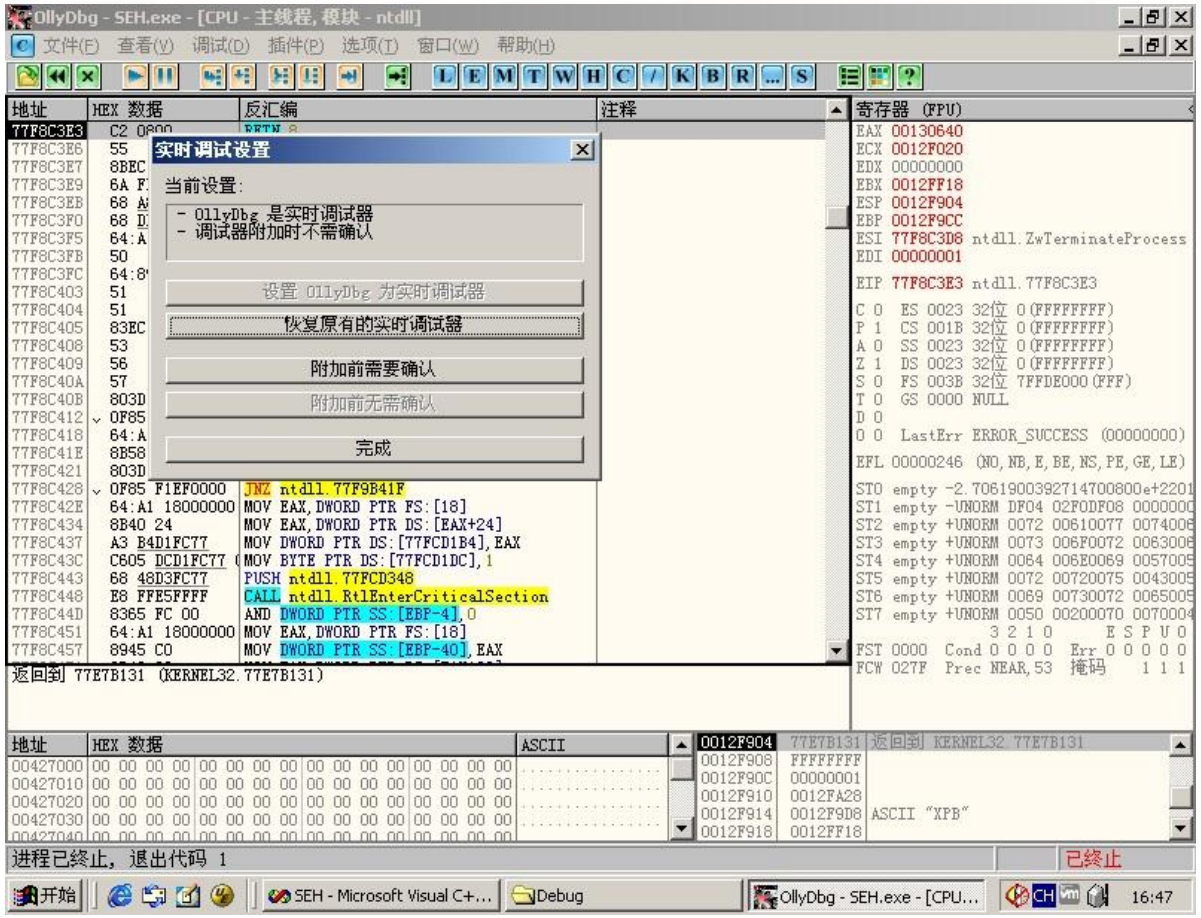
6）此外，异常处理机制会检测进程是否处于调试状态。如果直接用调试器加载程序，异常处理会进入调试状态下的处理流程。因此我们在代码中加入断点__asm int 3，让进程中断，然后用调试器attach的方法进行调试

S.E.H概述

	推荐使用的环境	备注
操作系统	Win 2000	虚拟机实体机均可
编译器	Vc 6.0	
编译选项	默认编译选项	
Build版本	Release版本	必须使用release版本进行调试

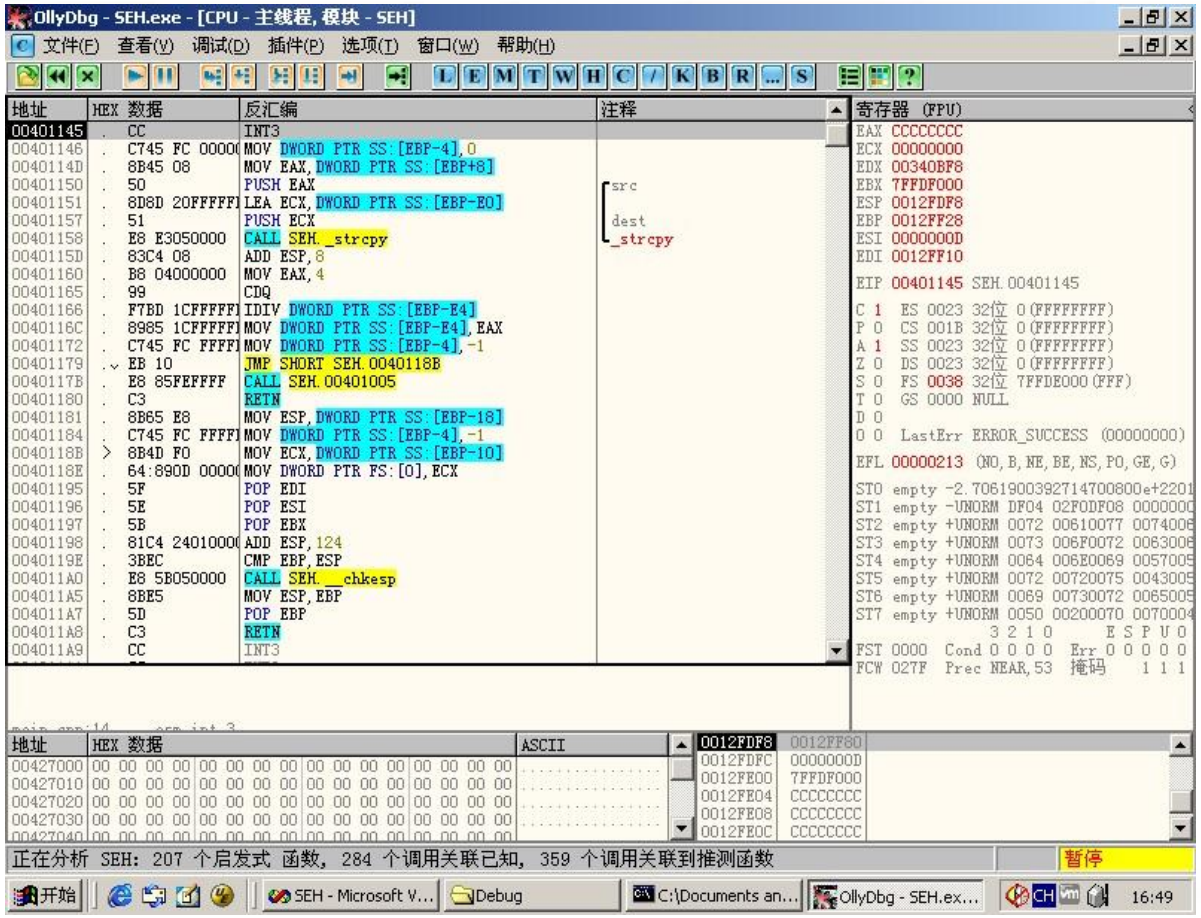
关于实验环境必须要说明，只在win2000下才能进行实验，
winXP SP2和Win 2003以及之后的版本都对SEH进行了优化，
会导致实验的失败，具体优化细节我们可能会在以后的课上进行讲述

S.E.H概述



为了能触发int 3断点时启动OllyDbg，我们选择选项中的实时调试设置选择设置OllyDbg为实时调试器，然后当我们运行exe文件时，int 3 断点触发后就会启动OllyDbg

S.E.H概述



成功在 int3上启动了Ollydbg，有可能启动时要求设置UDD和Plugin目录，我们设置一下即可

S.E.H概述

在执行strcpy函数之前，我们可以看到shellcode的起始地址

0012FE48

S.E.H概述

执行完strcpy后，确认0012FE48是我们shellcode的起始

S.E.H概述

我们点击查看菜单中的S.E.H链，我们会看到这个S.E.H链的情况，我们主要针对第一个地址。

S.E.H概述

我们去看一下那个地址的记录，果然，是一个指向下一个SEH的指针，接着是异常处理程序，我们只需要把0012FF1C这个地址的内容改成我们的shellcode起始地址就可以了

S.E.H概述

我们查得了shellcode的起始地址为

0x0012FE48

第一个S.E.H地址为

0x0012FF18(指向下一个S.E.H的指针)

0x0012FF1C(异常处理地址)

那么我们的shellcode应该有

0x0012FF1C-0x0012FE48=212的字节进行填充

S.E.H概述

我们还使用我们曾经使用过的buptbupt弹出对话框(静态地址版，不是jmp esp的那个版本)的那个shellcode内容，作为我们shellcode的核心内容。

剩下的空间我们用0x90来补齐至212字节，然后我们在213-216字节使用0x0012FE48填充(不要忘了内存的大端小端的问题)。

S.E.H概述

我们再来修改一下源代码，在main函数中加入

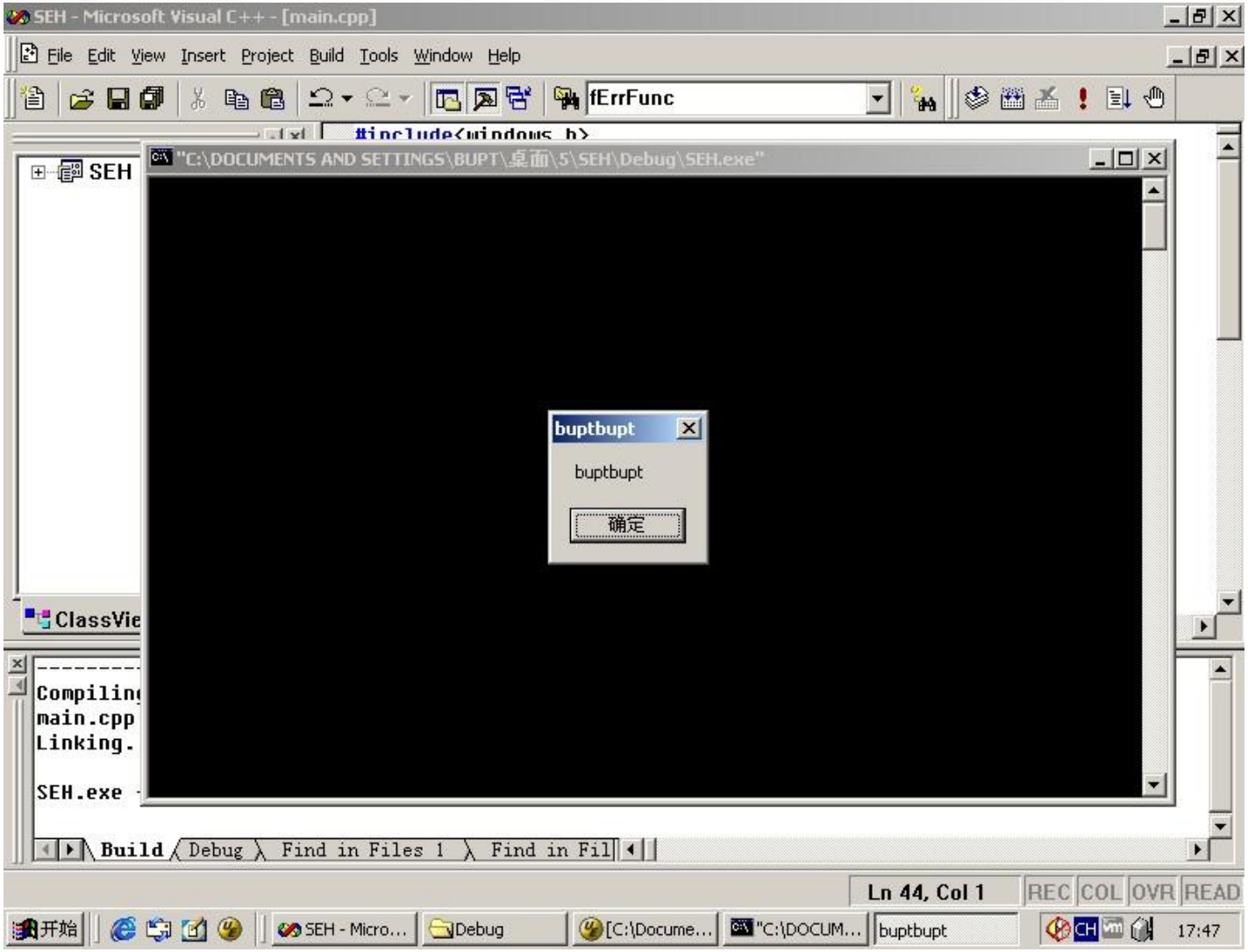
LoadLibrary(“user32.dll”);//为了我们能够顺利调用出

//对话框

取消__asm int 3

然后我们直接启动程序

S.E.H概述


SEH - Microsoft Visual C++ - [main.cpp]
File Edit View Insert Project Build Tools Window Help
fErrFunc
SEH
"C:\DOCUMENTS AND SETTINGS\BUPT\桌面\5\SEH\Debug\SEH.exe"
buptbupt
buptbupt
确定
ClassVie
Compiling
main.cpp
Linking.
SEH.exe
Build Debug Find in Files 1 Find in Fil
Ln 44, Col 1
REC COL OVR READ
开始
SEH - Micro...
Debug
[C:\Docume...
"C:\DOCUM...
buptbupt
17:47

S.E.H概述

我们看到我们的shellcode已经被执行，但是我们点击确定却没有反应，因为我们的shellcode已经被当作系统异常处理来进行了，所以会有所不同。

S.E.H概述

在S.E.H方面，微软也做了一些改进，比如SafeSEH和SEHOP。

在Windows XP SP2及后续版本的操作系统中，微软引入了著名的S.E.H校验机制SafeSEH。

SafeSEH的原理很简单，在程序调用异常处理函数之前，对要调用的异常处理函数进行一系列的有效性校验，当发现异常处理函数不可靠时将终止异常处理函数的调用。

S.E.H概述

SafeSEH需要操作系统和编译器的双重支持，二者缺一都会降低SafeSEH的保护能力。

SafeSEH的保护措施：

1）检查异常处理链是否位于当前程序的栈中。如果不在当前栈中，程序将终止异常处理函数的调用。

2）检查异常处理函数指针是否指向当前程序的栈中，如果指向当前栈中，程序将终止异常处理函数的调用。

S.E.H概述

3）在前面两项检查都通过后，程序调用一个全新的函数RtlIsValidHandler()，来对异常处理函数的有效性进行验证，稍后我们会详细介绍RtlIsValidHandler()函数。

S.E.H概述

首先该函数判断异常处理函数地址是不是在加载模块的内存空间，如果属于加载模块的内存空间，校验函数将依次进行如下校验。

1）判断程序是否设置了IMAGE_DLLCHARACTERISTICS_NO_SEH标识。如果设置了这个标识，这个程序内的异常会被忽略。所以当这个标识被设置时，直接返回失败。

S.E.H概述

2）检验程序是否包含安全S.E.H表。如果程序包含安全S.E.H表，则将当前的异常处理函数地址与该表进行匹配，匹配成功则返回校验成功，匹配失败则返回校验失败。

3）判断程序是否设置Ilonly标识。如果设置了这个标识，说明该程序只包含.NET编译人中间语言，函数直接返回校验失败。

S.E.H概述

4）判断异常处理函数地址是否位于不可执行页上。当异常处理函数地址位于不可执行页上时，校验函数将检测DEP是否开启，如果系统未开启DEP则返回校验成功，否则程序抛出访问违例的异常。

S.E.H概述

如果异常处理函数的地址没有包含在加载模块的内存空间，校验函数将直接进行DEP相关检测，函数依次进行如下校验。

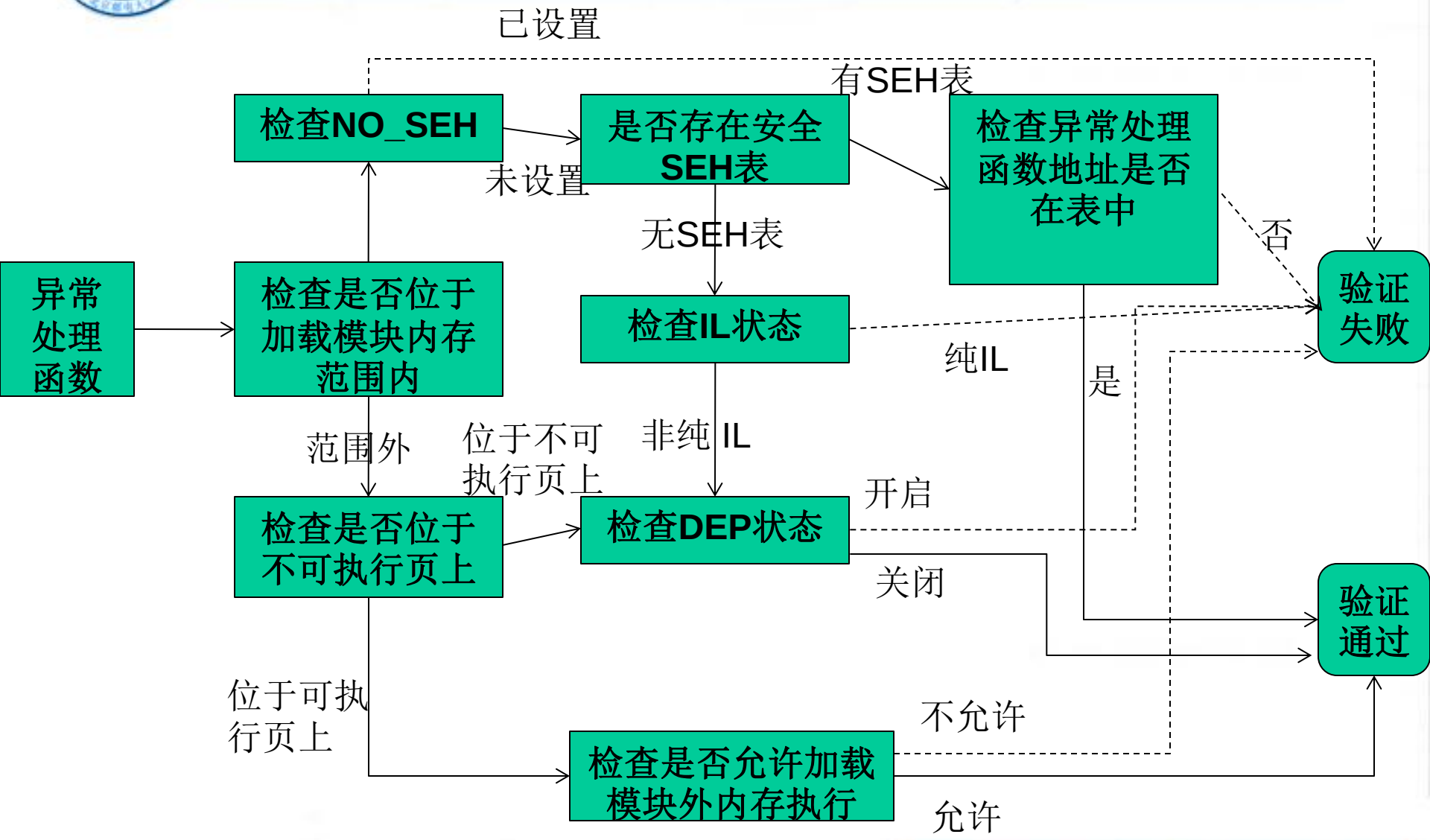
1）判断异常处理函数地址是否位于不可执行页上。但异常处理函数位于不可执行页上时，校验函数将检测DEP是否开启，如果系统未开启DEP则返回校验成功，否则抛出访问违例异常。

S.E.H概述

2）判断系统是否允许跳转到加载模块的内存空间外执行，如果允许则返回校验成功，否则返回校验失败。

那我们来看看下面的流程图：

S.E.H概述

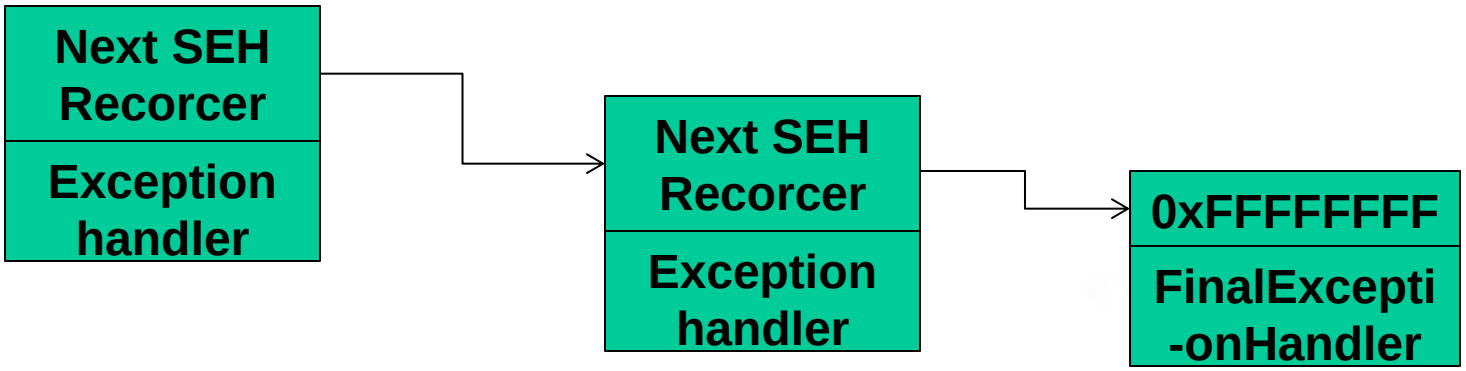

已设置
有SEH表
检查NO_SEH
是否存在安全SEH表
检查异常处理函数地址是否在表中
未设置
无SEH表
否
异常处理函数
检查是否位于加载模块内存范围内
检查IL状态
验证失败
纯IL
是
范围外
位于不可执行页上
非纯 IL
开启
检查是否位于不可执行页上
检查DEP状态
关闭
验证通过
位于可执行页上
不允许
检查是否允许加载模块外内存执行
允许

S.E.H概述

SEHOP是一种比SafeSEH更为严厉的保护机制。

目前Vista SP1和Windows 7支持SEHOP。

我们来看看SEHOP是干什么的。我们知道S.E.H函数是以单链表的形式存放在栈中的，末端是默认异常处理。



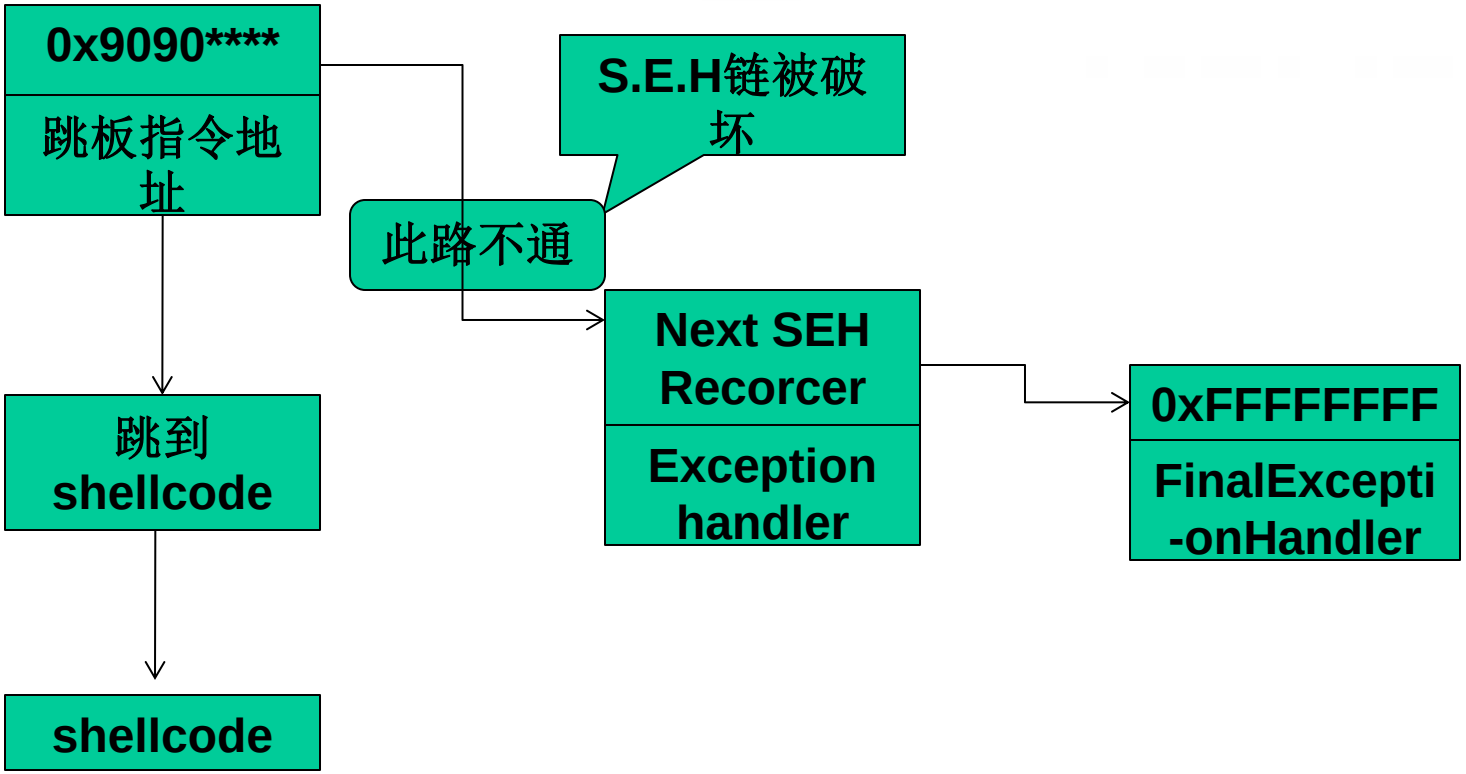
典型的S.E.H链

S.E.H概述

SEHOP的核心任务就是检查这条链的完整性，在程序转入异常处理前SEHOP会检查S.E.H链上最后一个异常处理函数是否为系统固定的终极异常处理函数。

如果是，说明这个链没有被破坏，可以去执行当前的异常处理函数；如果检测不是，则说明被破坏，程序不会去执行当前的异常处理函数。

S.E.H概述



典型的S.E.H溢出过程

通过这节课的学习，我们知道了微软操作系统的安全性是不断在进步的，也知道了我们好多的溢出手段已经在较新的操作系统是无法使用的了。

我们课上讲解的攻击手段只是基础和原理，希望学有余力的同学能够继续进行深入的研究。

谢谢观赏！
Thanks！