第10讲 其它内存攻击的机理与防御

系统安全与可信计算研究所 陈泽茂

chenzemao@whu.edu.cn

目 录

- Windows异常处理机制
- Windows异常溢出漏洞利用与防御
- C++对象溢出漏洞机理与利用
- 整数溢出漏洞机理与利用
- 格式化串漏洞机理与利用

一、Windows异常处理机制

1.1 Windows的异常处理编程模型

- 异常：CPU或者程序发生的某种错误
  - 硬件异常： 硬件异常是由CPU发现的异常，例：除零异常、内存访问异常
  - 软件异常：由操作系统或应用程序抛出的异常，比如C++关键字throw，Windows的RaiseException函数。
- 异常处理：异常发生之后，系统用于处理所产生错误的一段程序。

1.1 Windows的异常处理编程模型

__try	Begins a guarded body of code. Used with the **__except** keyword to construct an exception handler, or with the **__finally** keyword to construct a termination handler.
__except	Begins a block of code that is executed only when an exception occurs within its associated **__try** block.
__finally	Begins a block of code that is executed whenever the flow of control leaves its associated **__try** block.
__leave	Allows for immediate termination of the **__try** block without causing abnormal termination and its performance penalty.

如果程序源代码中使用了__try{}/__except{}、try{}/catch{}等异常处理机制，编译器将会在当前函数栈帧中安装一个结构化异常处理器（S.E.H）来实现异常处理。

1.1 Windows的异常处理编程模型

```
void vulfun_1(char *input)
{
    char buf[8];

    __try {
        strcpy(buf, input);  已用时间 <= 3ms
    }
    __except (GetExceptionCode()) {
        printf("vulfun_1: Exception caught.\n");
    }
}
```

无异常处理代码时的反汇编结果

```
void vulfun_1(char *input)
{
    char buf[8];
    strcpy(buf, input);
}
```

```
void vulfun_1(char *input)
{
00171C30 55                  push        ebp
00171C31 8B EC               mov         ebp,esp
00171C33 83 EC 48            sub         esp,48h
00171C36 53                  push        ebx
00171C37 56                  push        esi
00171C38 57                  push        edi
    char buf[8];
```

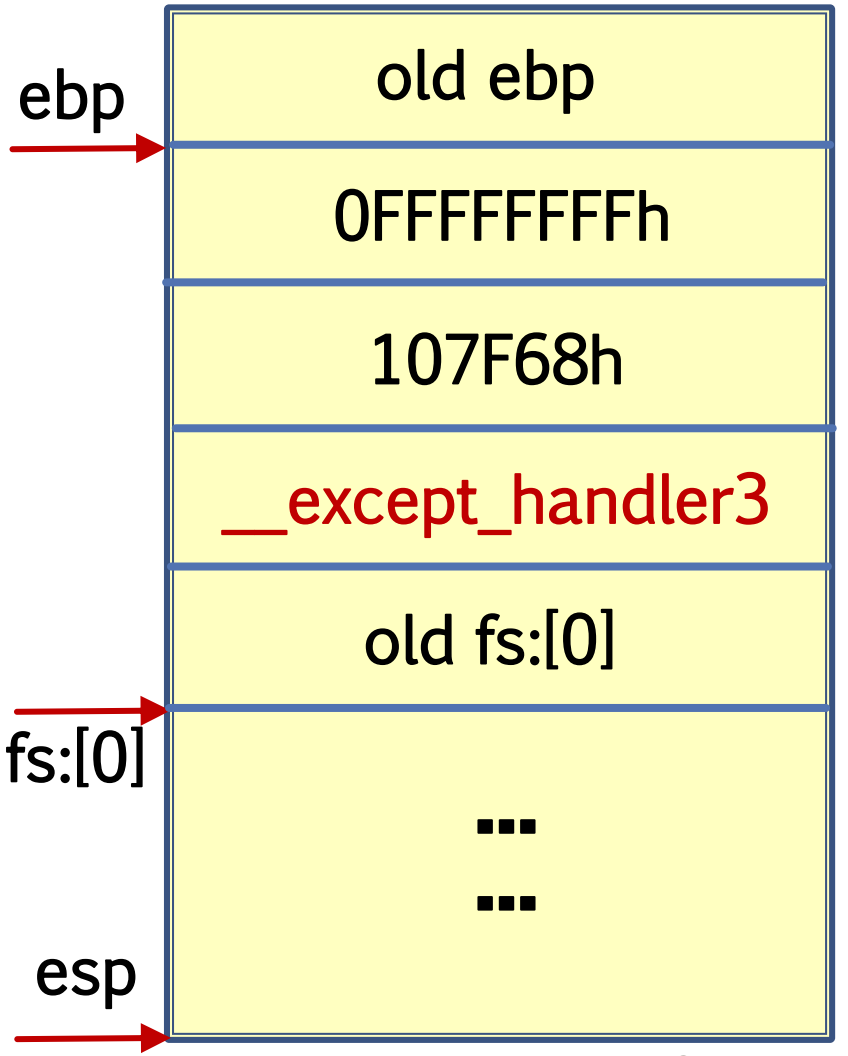
含异常处理代码时的反汇编结果

```
void vulfun_1(char *input)
{
00101C30 55                   push        ebp
00101C31 8B EC                mov         ebp,esp
00101C33 6A FF                push        0FFFFFFFFh
                              push        107F68h
                              push        offset __except_handler3 (010
                              mov         eax,dword ptr fs:[00000000h]
                              push        eax
                              mov         dword ptr fs:[0],esp
                              add         esp,0FFFFFFACh
                              push        ebx
00101C51 56                   push        esi
00101C52 57                   push        edi
00101C53 89 65 E8             mov         dword ptr [ebp-18h],esp
    char buf[8];
```

```
void vulfun_1(char *input)
{
    char buf[8];

    __try {
        strcpy(buf, input);  已用时间<=3ms
    }
    __except (GetExceptionCode()) {
        printf("vulfun_1: Exception caught.\n");
    }
}
```

1.2 带异常处理的函数栈帧

```
push        ebp
mov         ebp, esp
push        0FFFFFFFFh
push        107F68h
push        offset __except_handler3 (010484Eh)
mov         eax, dword ptr fs:[00000000h]
push        eax
mov         dword ptr fs:[0], esp
add         esp, 0FFFFFFACh
push        ebx
push        esi
push        edi
mov         dword ptr [ebp-18h], esp
```

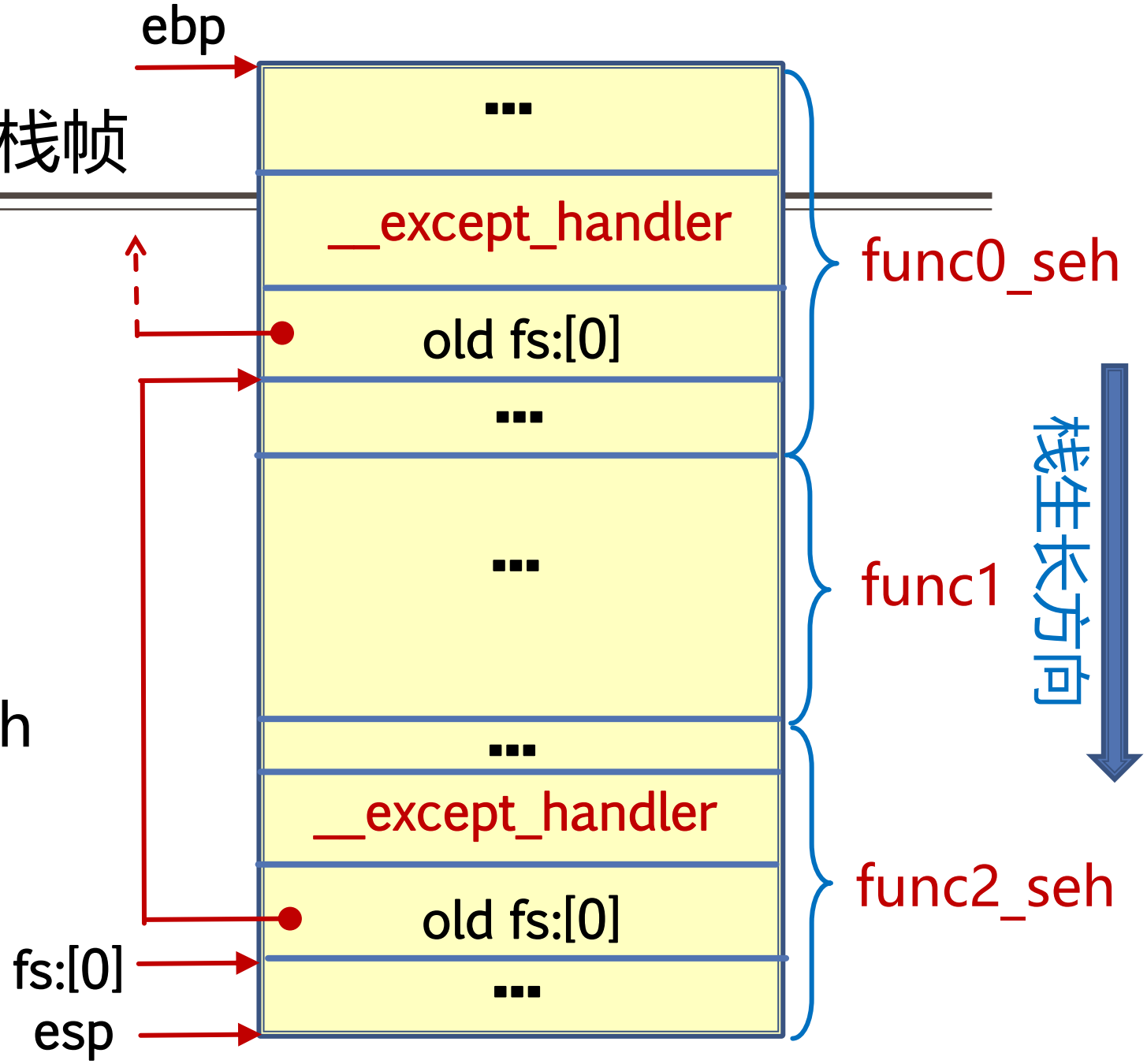


1.2 带异常处理的函数栈帧

函数栈帧分析

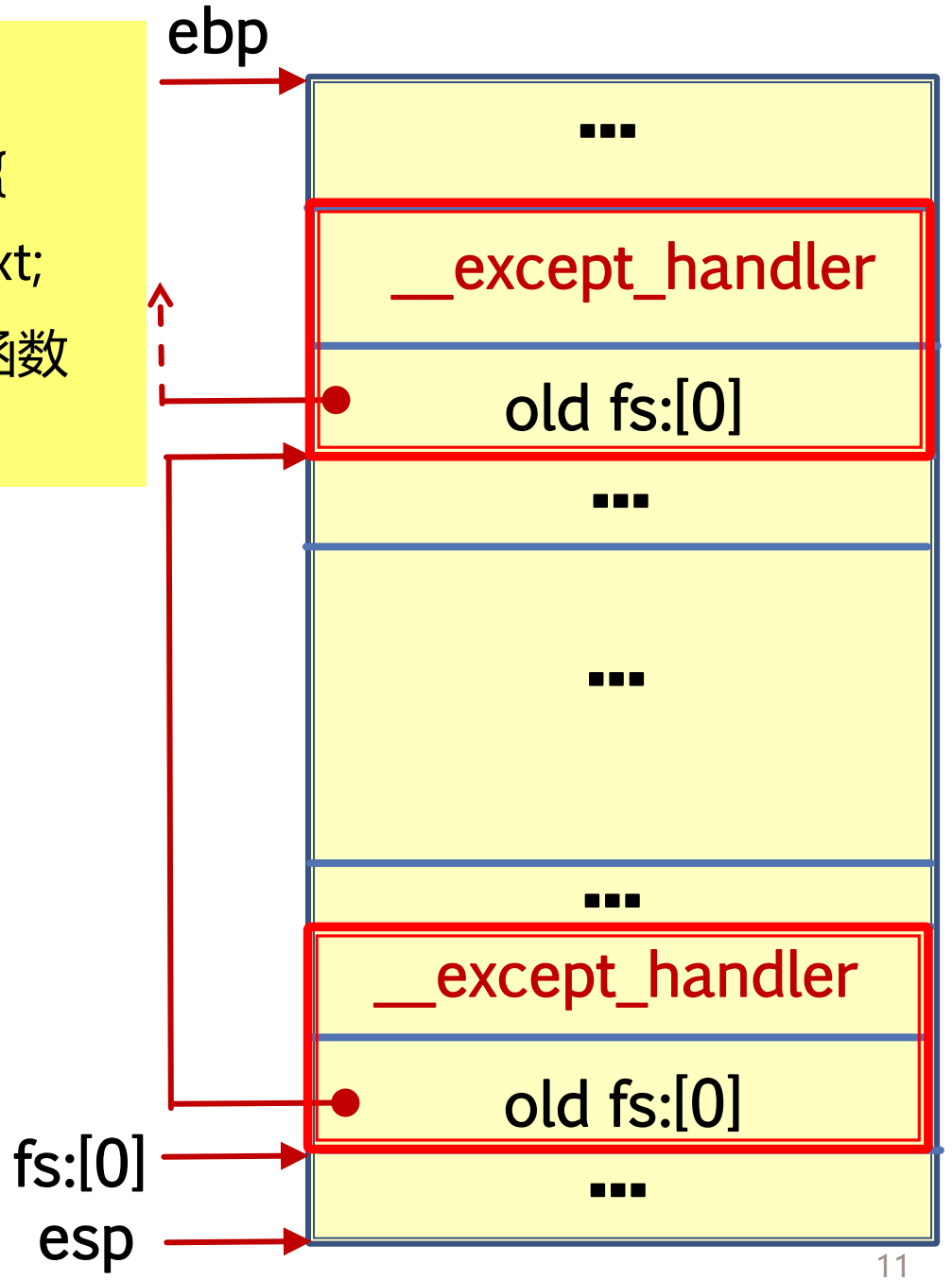
overflow_seh

func0_seh→func1→func2_seh



SEH Record的数据结构

```
typedef struct _EXCEPTION_REGISTRATION_RECORD {
    struct _EXCEPTION_REGISTRATION_RECORD *Next;
    PEXCEPTION_ROUTINE Handler;  //指向异常处理函数
} EXCEPTION_REGISTRATION_RECORD;
```

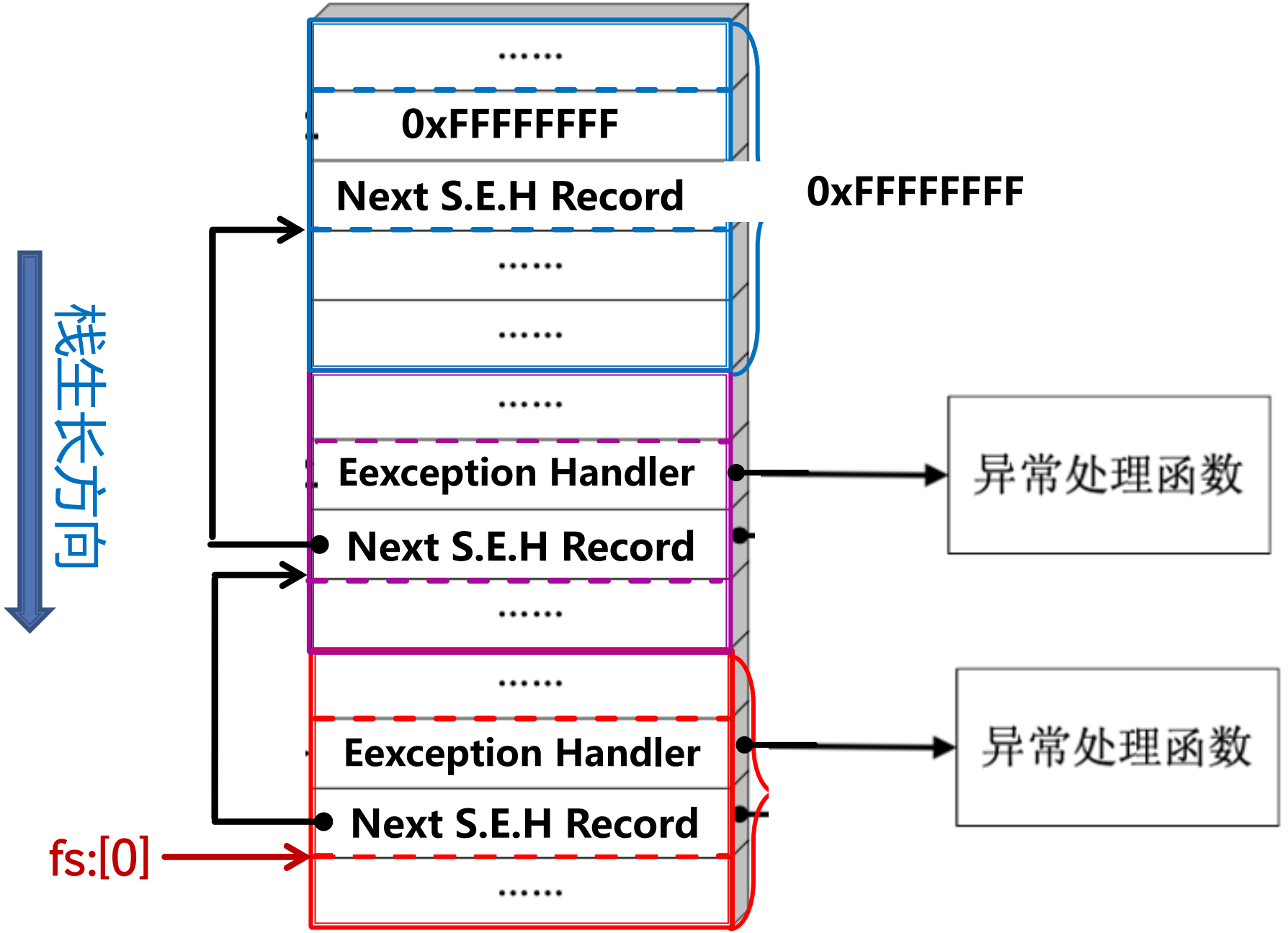


SEH Record的数据结构

```
typedef struct _EXCEPTION_REGISTRATION_RECORD {
    struct _EXCEPTION_REGISTRATION_RECORD *Next;
    PEXCEPTION_ROUTINE Handler;  //指向异常处理函数
} EXCEPTION_REGISTRATION_RECORD;
```

1.3 线程栈中S.E.H链

S.E.H链的最后节点的Next字段为0xFFFFFFFF



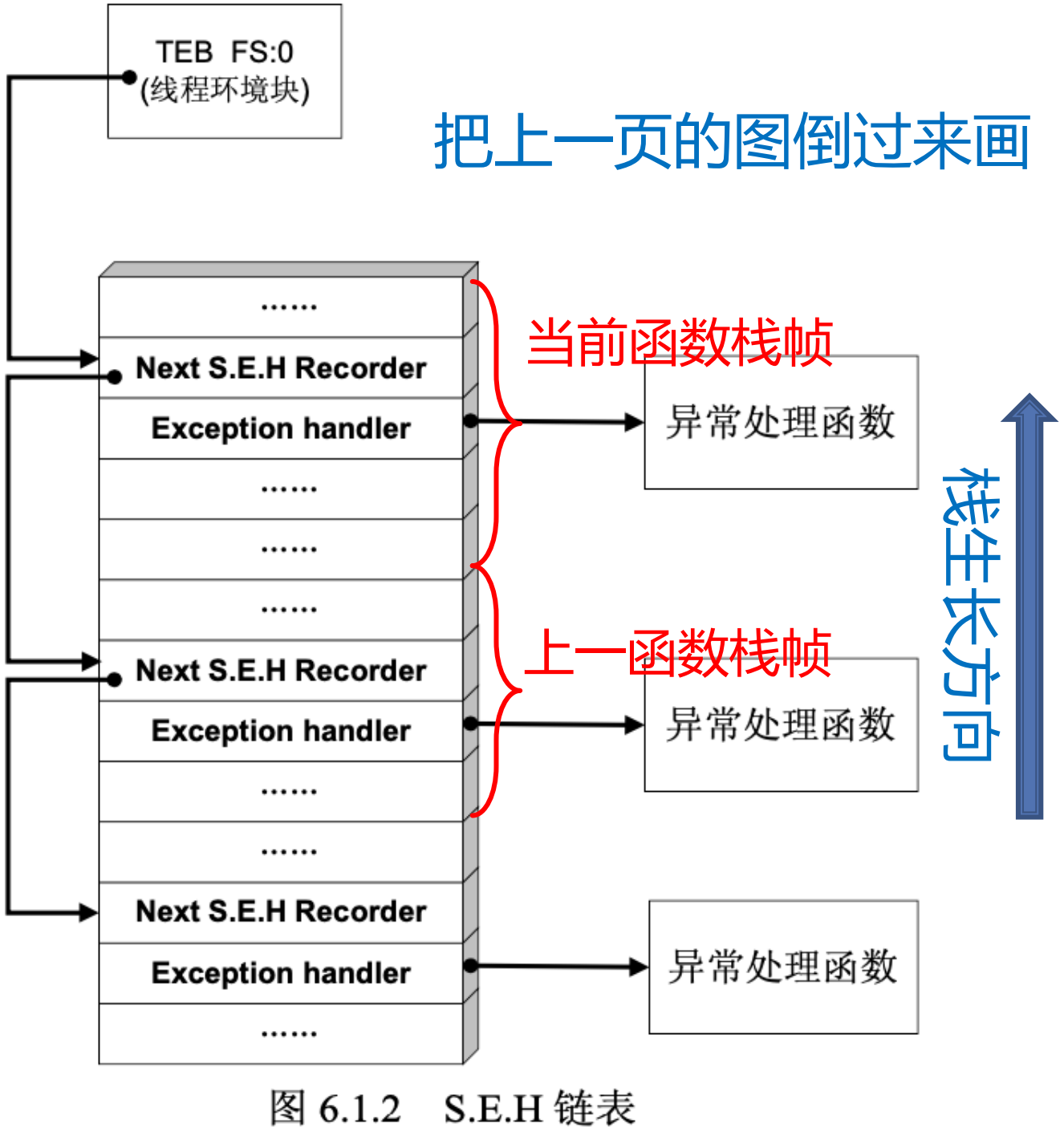
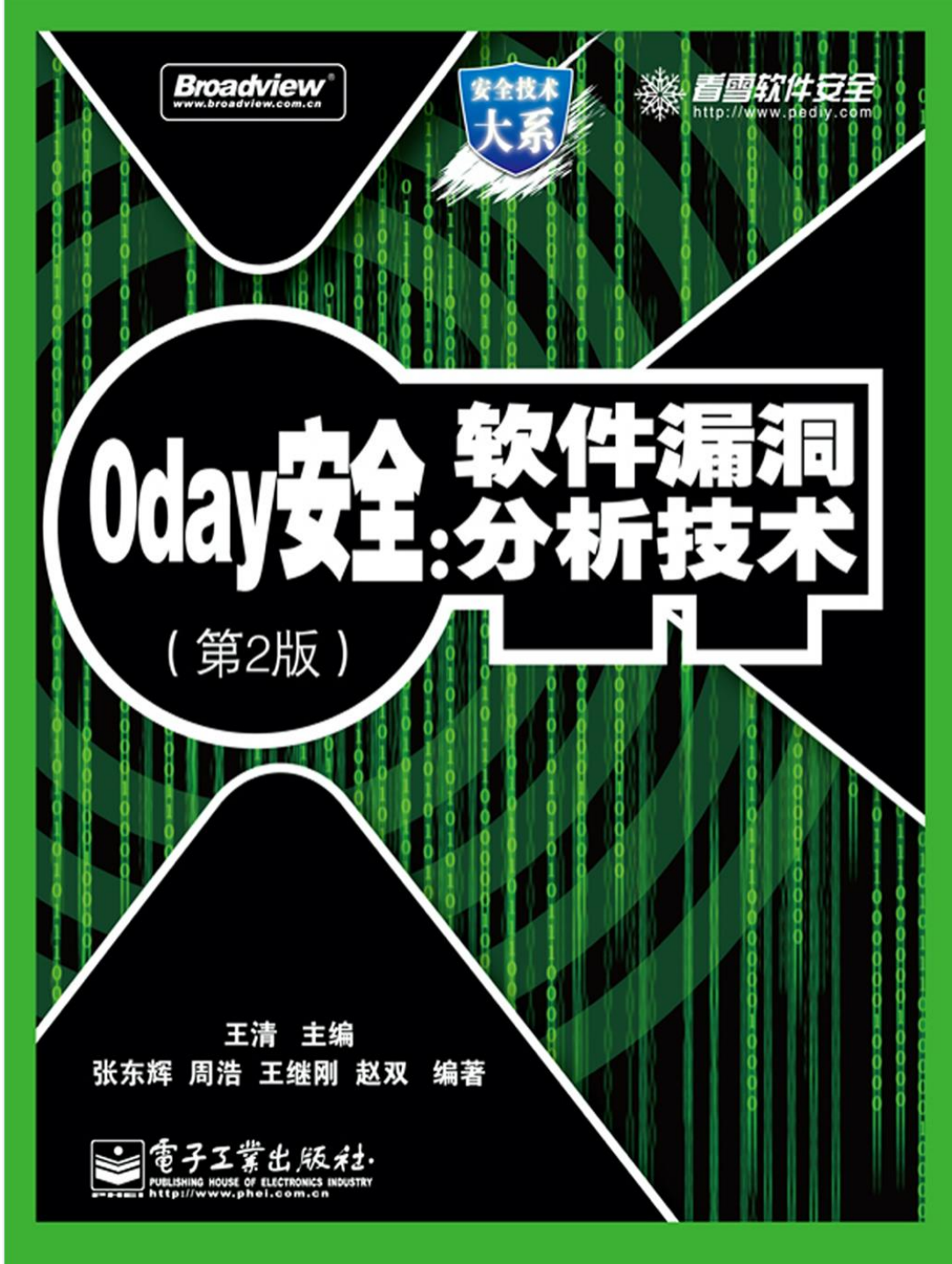
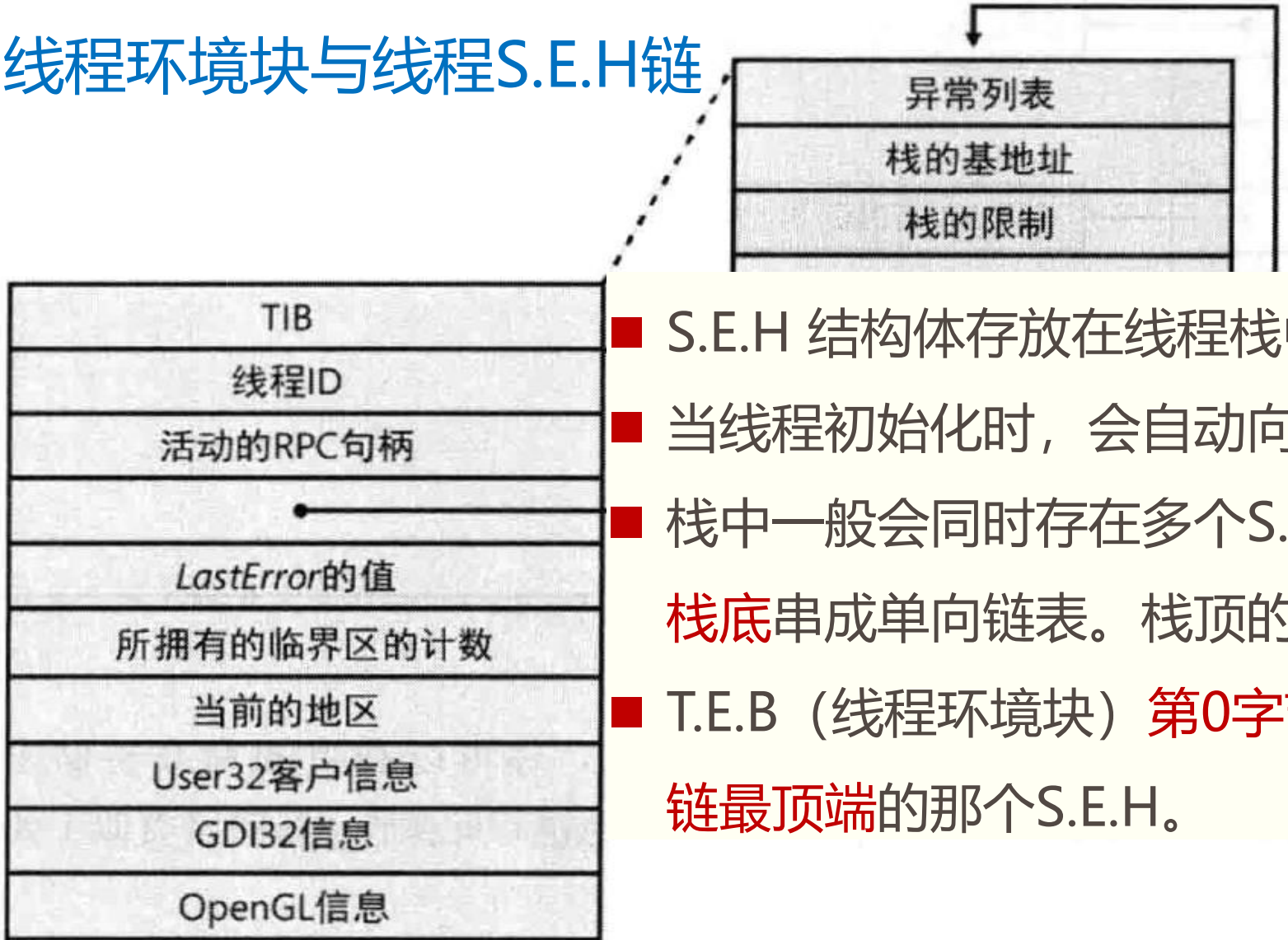

TEB FS:0
(线程环境块)
把上一页的图倒过来画
.....
Next S.E.H Recorder
Exception handler
.....
.....
.....
Next S.E.H Recorder
Exception handler
.....
.....
Next S.E.H Recorder
Exception handler
.....
当前函数栈帧
上一函数栈帧
异常处理函数
异常处理函数
异常处理函数
栈生长方向

图 6.1.2　S.E.H 链表


Broadview
www.broadview.com.cn
安全技术大系
看雪软件安全
http://www.pediy.com
0day安全：软件漏洞分析技术
（第2版）
王清　主编
张东辉　周浩　王继刚　赵双　编著
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

1.3 线程栈中S.E.H链

线程环境块与线程S.E.H链



- S.E.H 结构体存放在线程栈中。
- 当线程初始化时，会自动向栈中安装一个默认的S.E.H。
- 栈中一般会同时存在多个S.E.H，这些S.E.H由栈顶向栈底串成单向链表。栈顶的S.E.H最靠近异常发生点。
- T.E.B（线程环境块）第0字节偏移处的指针指向S.E.H链最顶端的那个S.E.H。

1.4 Windows操作系统的异常处理流程

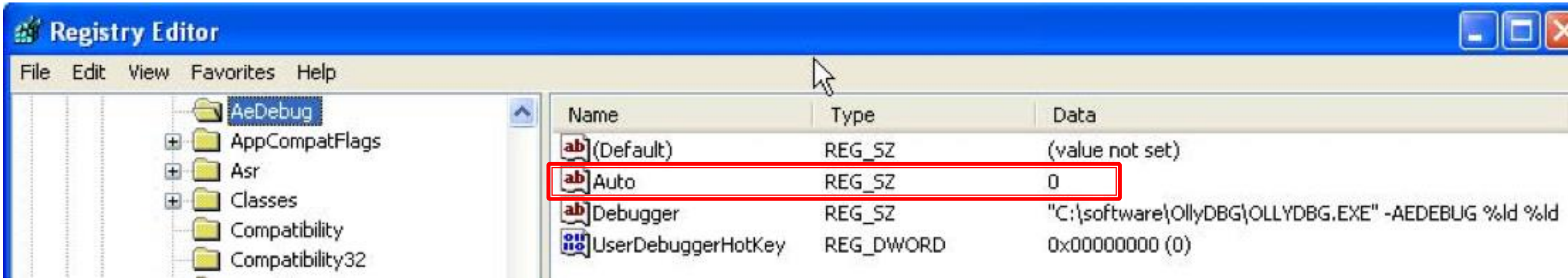
- 当异常发生时，首先由线程自身处理，操作系统从T.E.B的第 0 字节处（fs:[0]）取出S.E.H，开始执行异常处理。
- 若失败，将顺着 S.E.H 链表依次尝试其它异常处理函数。
- 若线程S.E.H链中所有的异常处理函数都没能处理异常，则执行进程中的异常处理。
- 如果进程异常处理失败或者用户没有注册进程异常处理，操作系统默认异常处理函数UnhandledExceptionFilter()会被调用。UnhandledExceptionFilter()简称为U.E.F 。

1.4 Windows操作系统的异常处理流程

UnhandledExceptionFilter () 将检查注册表

HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\AeDebug

下的Auto 键。

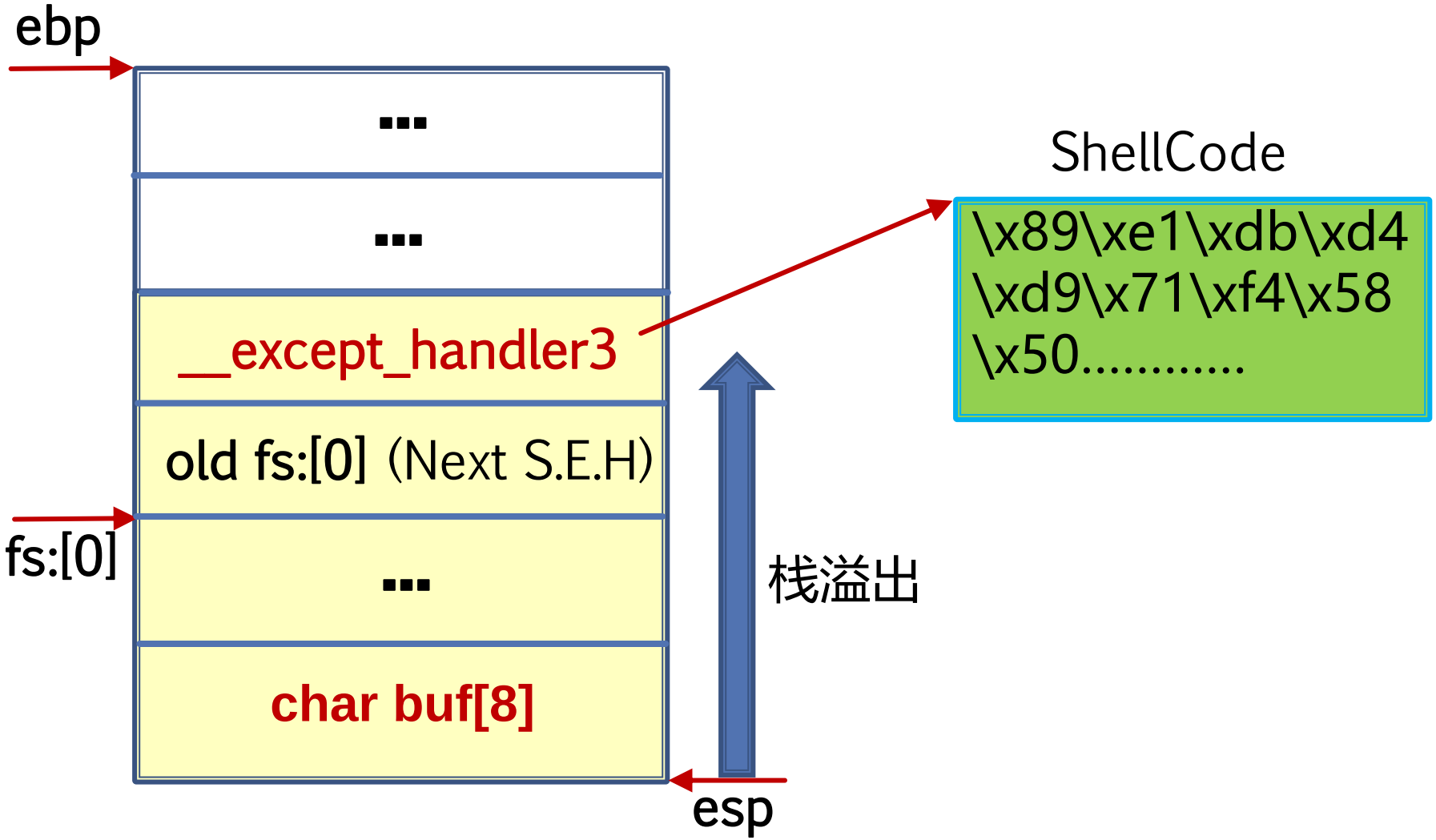


Auto=1: 不弹出错误对话框, 直接结束程序;

Auto=其余值: 弹出错误对话框。

二、Windows异常溢出漏洞利用与防御

2.1 覆盖栈帧中的异常处理函数地址



2.1 覆盖栈帧中的异常处理函数地址

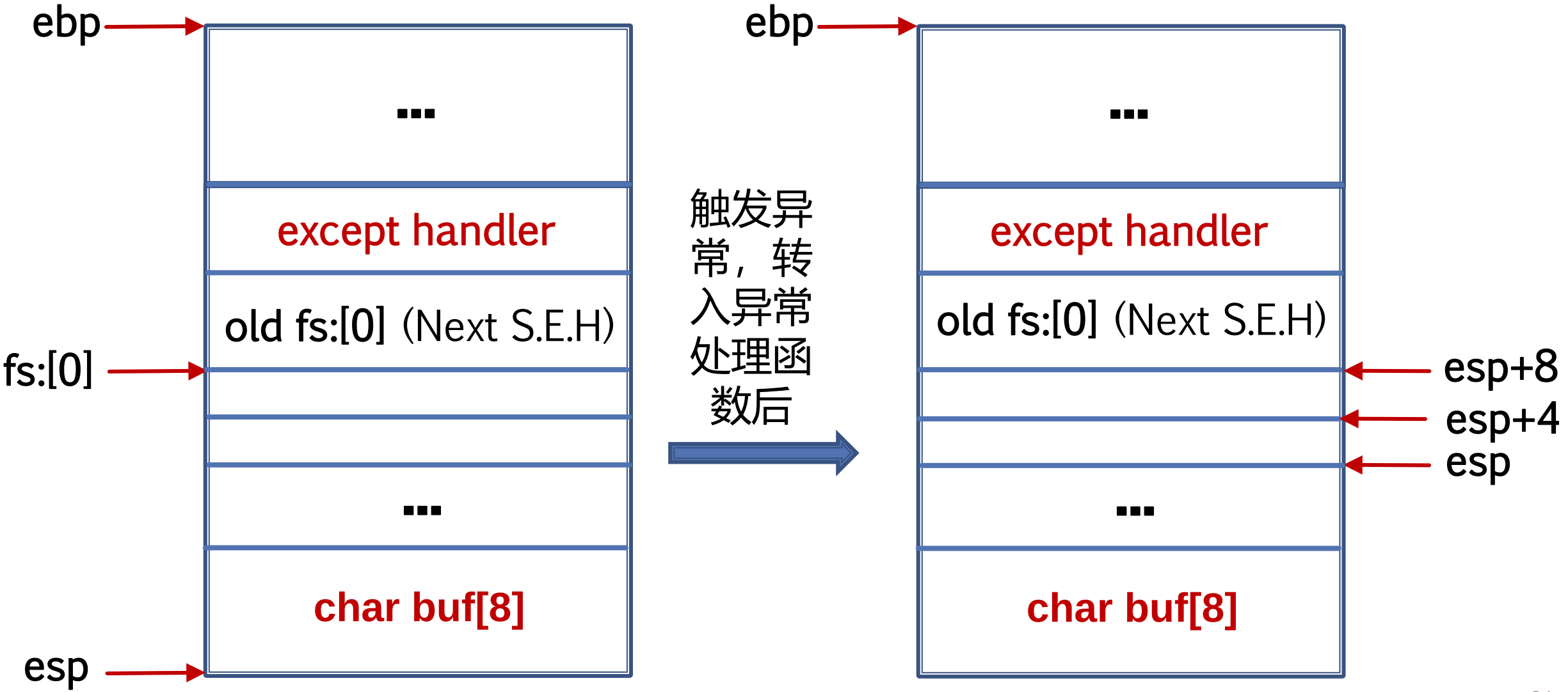
代码演示

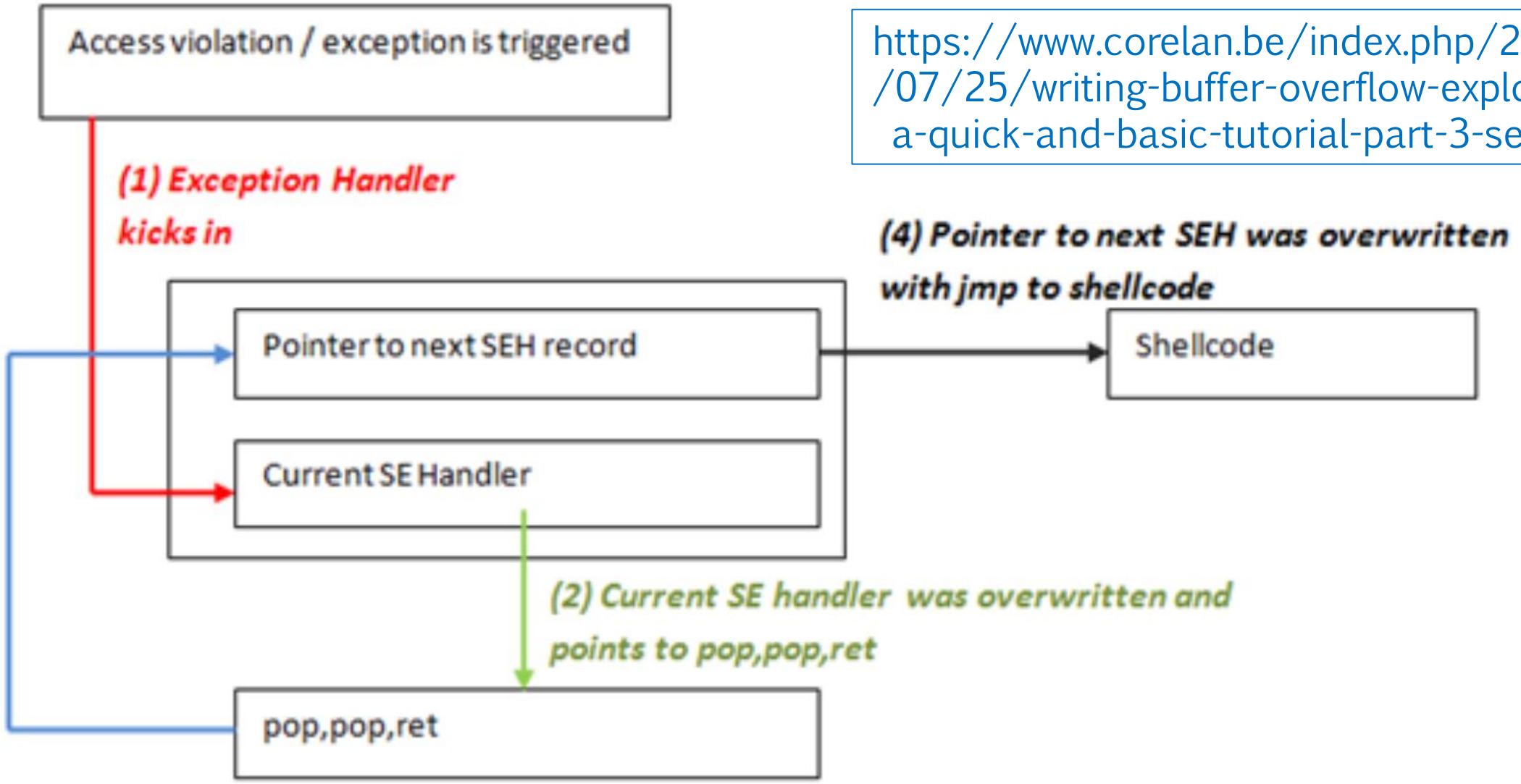
overflow_seh

探测异常处理函数地址：detect();

覆盖异常处理函数地址实现代码执行：vulfun_1();

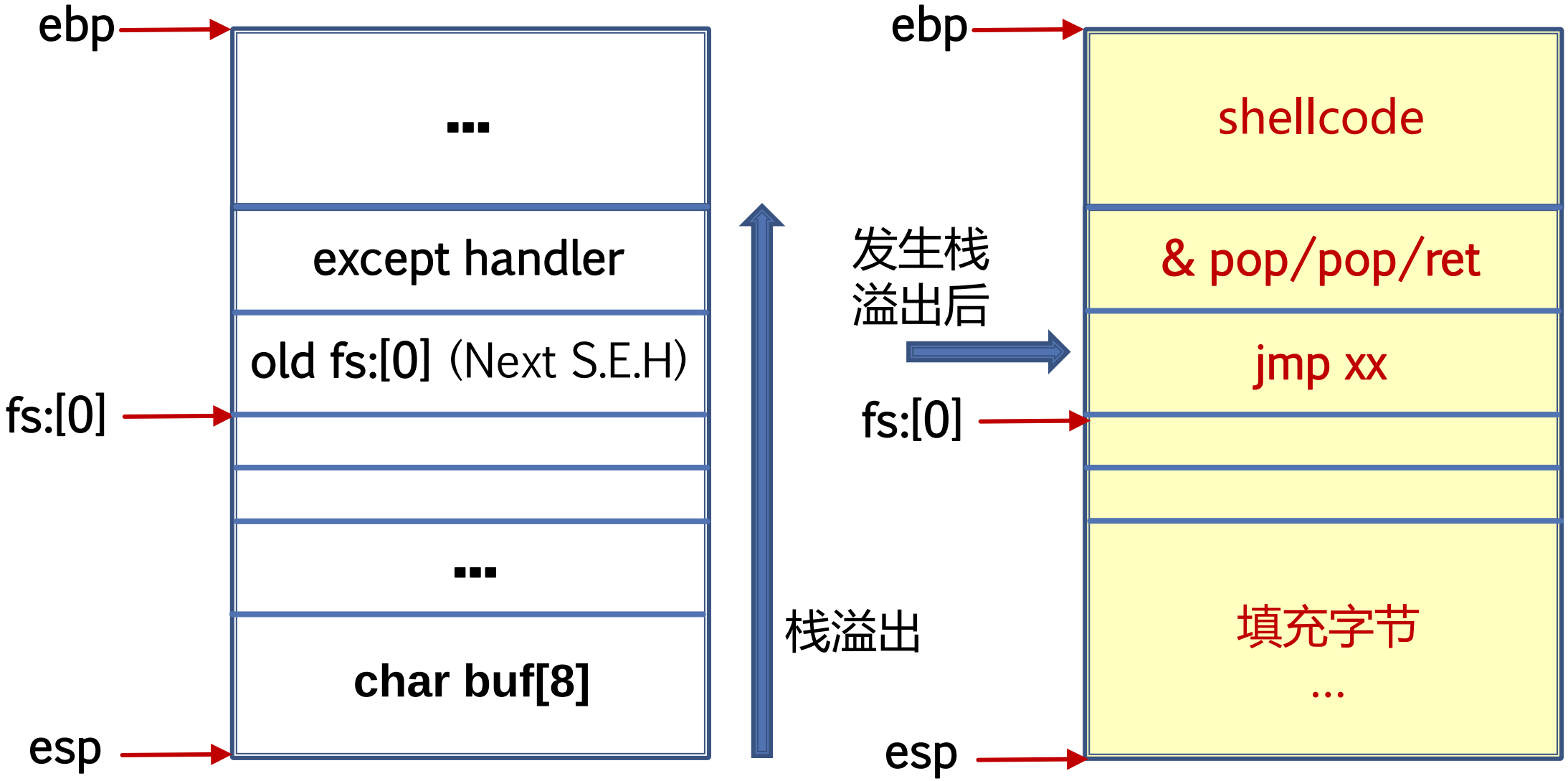
2.2 设计异常溢出时的栈帧结构



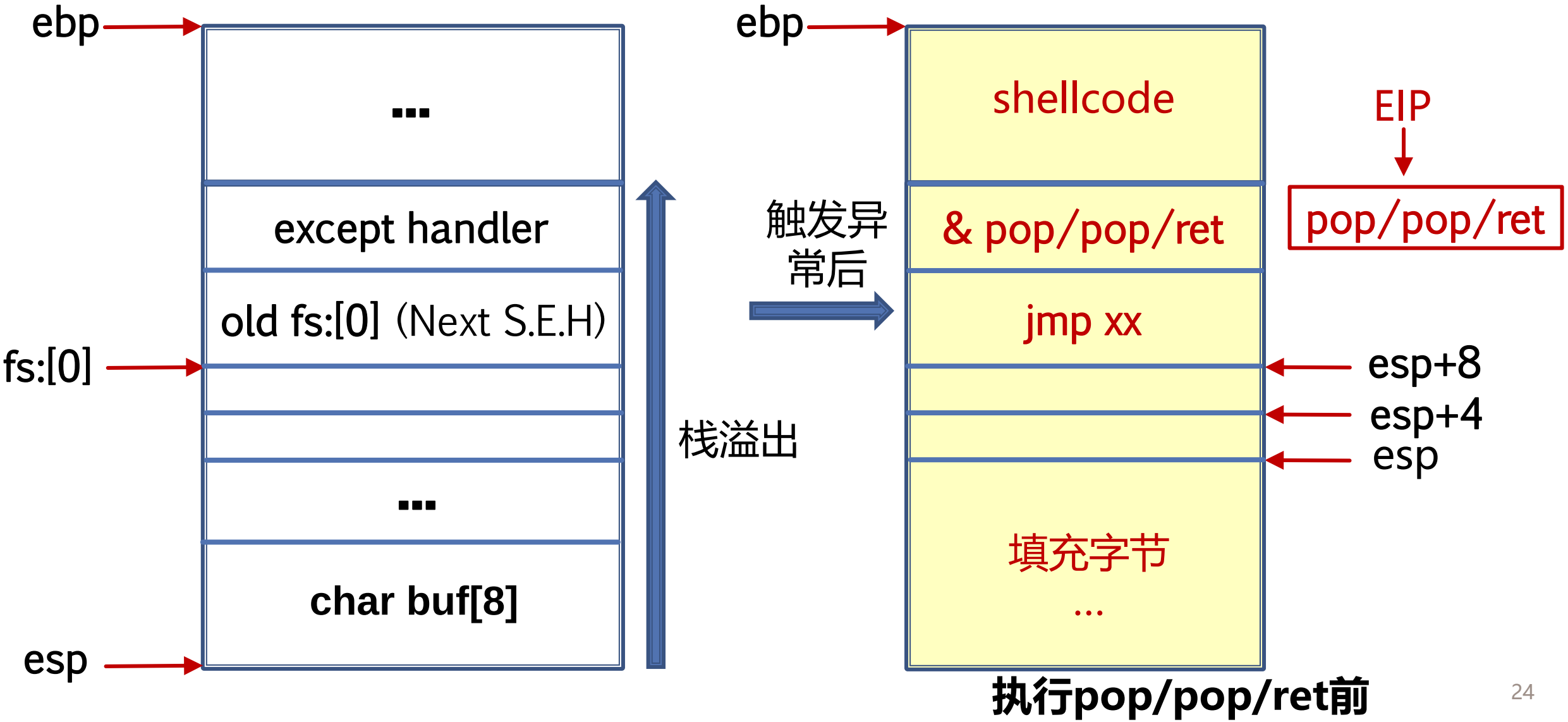


https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/

2.2 设计异常溢出时的栈帧结构

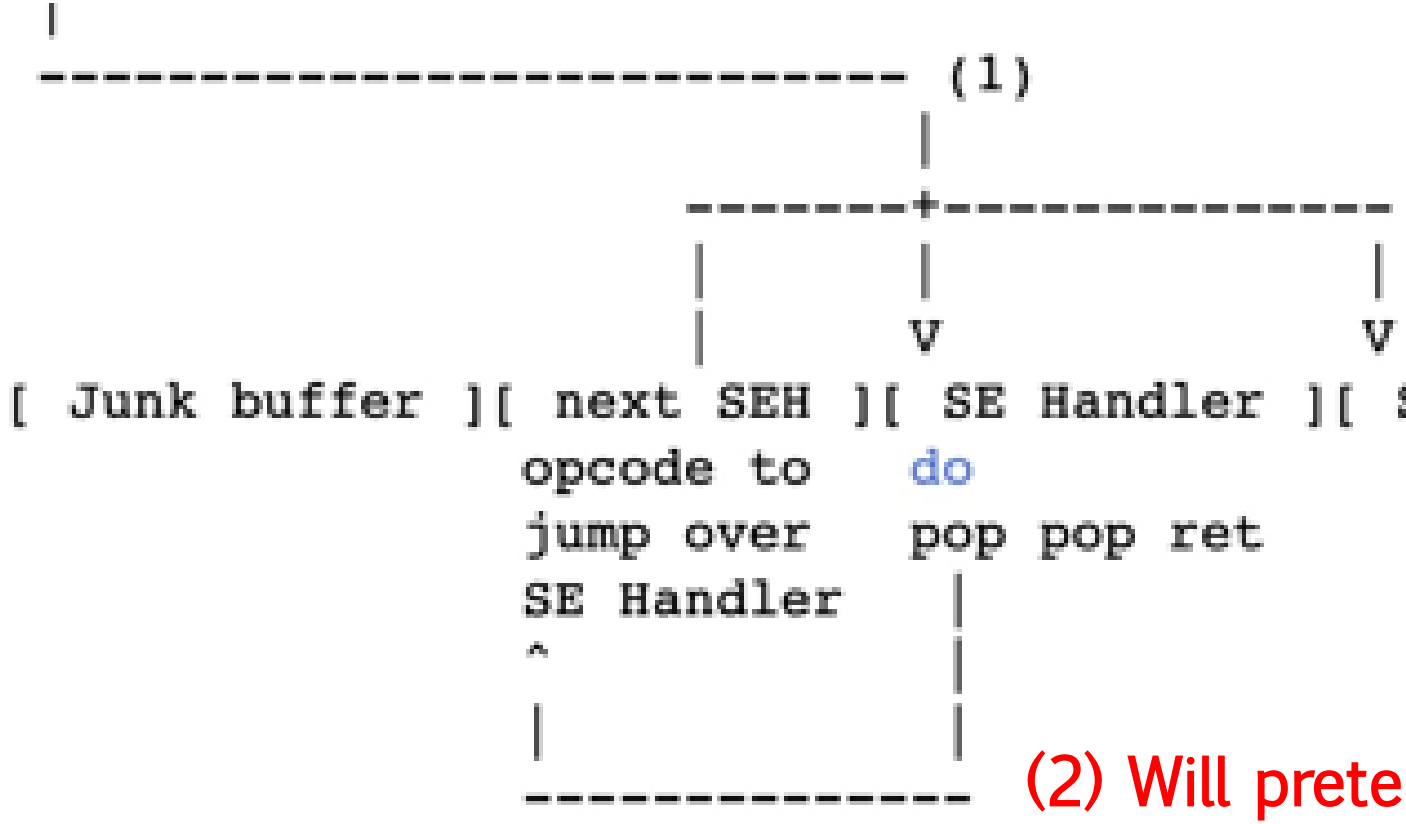


2.3 利用异常溢出触发shellcode



异常溢出利用数据的布局

(1) 1st exception occurs:



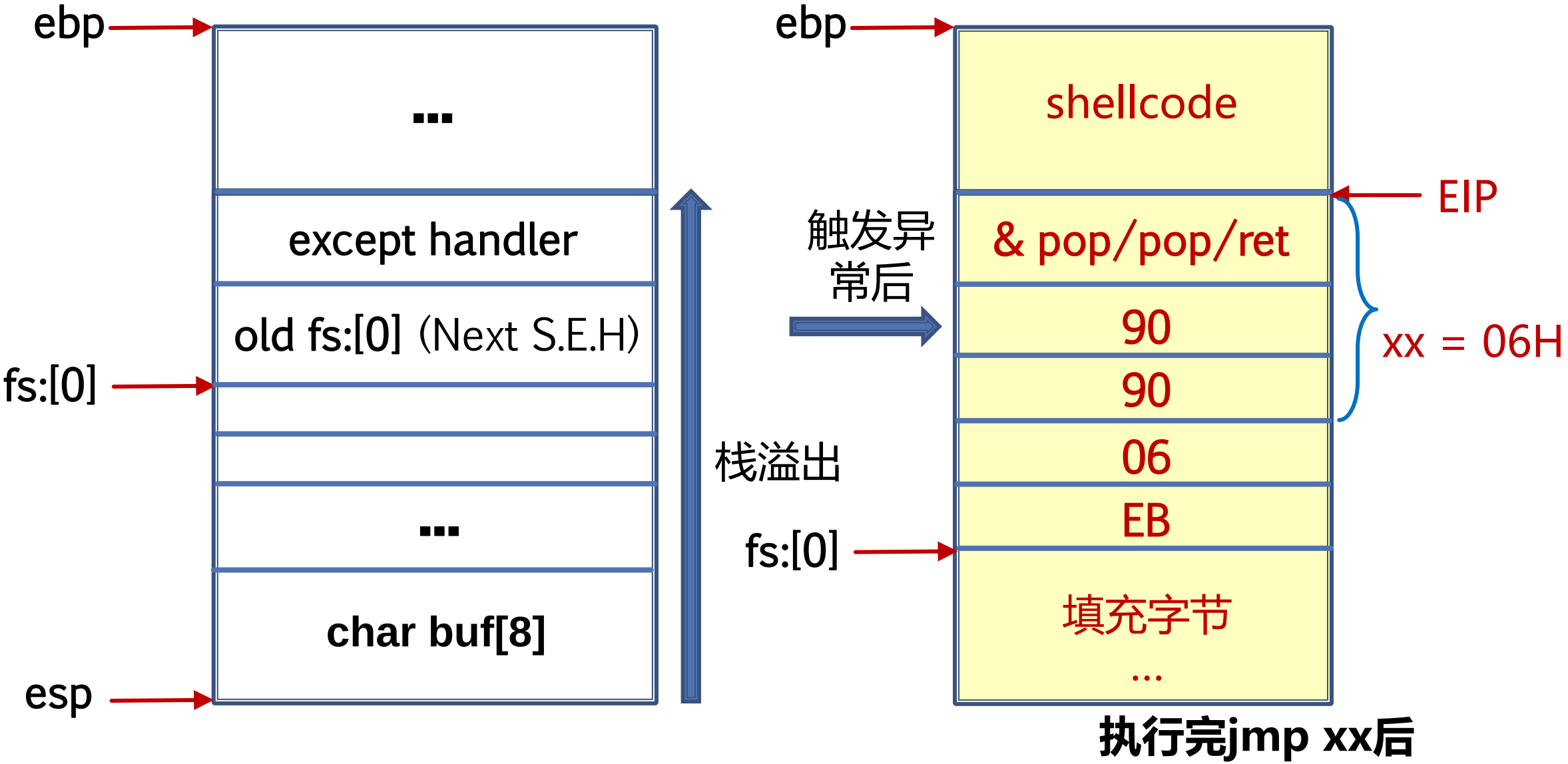
(3) Opcode in next SHE: jump over SE handler to the shellcode

(4) Shellcode gets executed

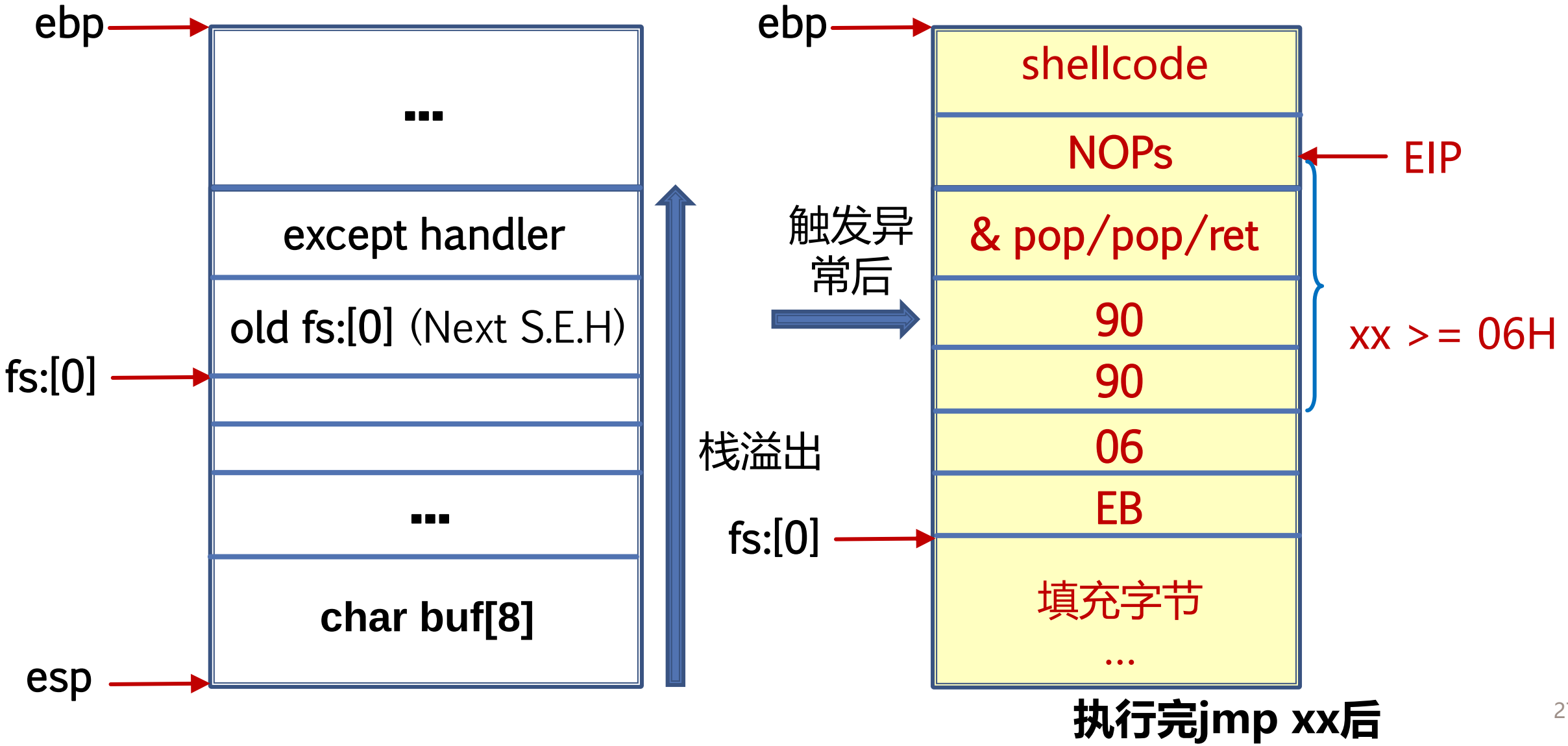
(2) Will pretend there is a second exception, puts address of next SHE location in EIP, so opcode gets executed.

https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/

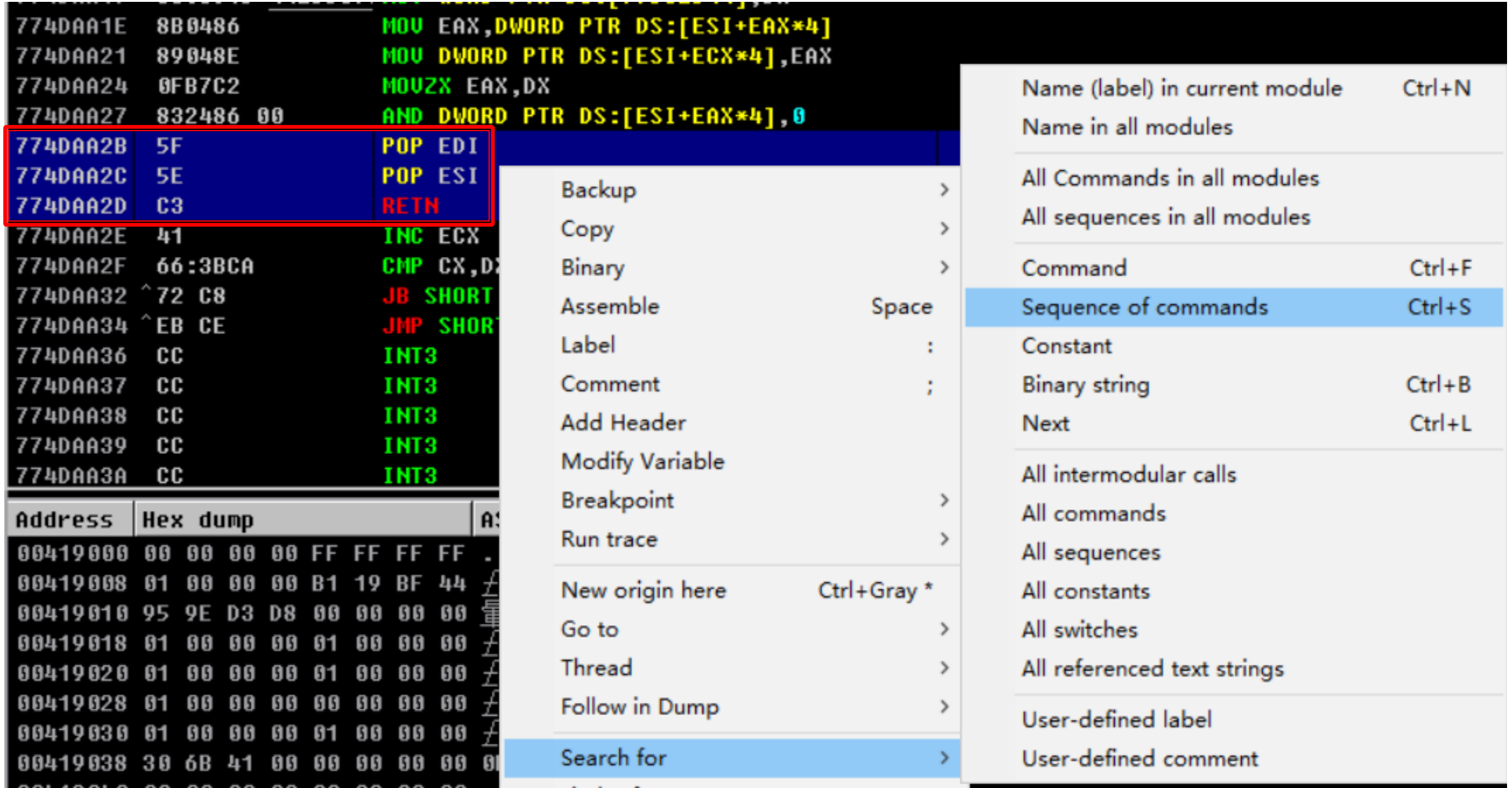
2.3 利用异常溢出触发shellcode



2.3 利用异常溢出触发shellcode



小技巧：查找pop pop ret指令序列



2.4 异常溢出防范技术——SafeSEH

- 防范思路
  - Windows XP SP2 introduced the SafeSEH protection mechanism in which validated exception handlers are registered and stored in a table.

    事先注册并保存经过验证的异常处理函数
  - The addresses in this table are checked prior to executing a given exception handler to ensure it is deemed “safe”.

    在执行给定的异常处理函数之前，基于事先构建的异常处理函数表，检查该函数是否是安全的。

2.4 异常溢出防范技术——SafeSEH

- 编译器层面的实现：启用/SafeSEH链接选项
  - 该选项在VS2003 及后续版本默认启用。
  - 启用该链接选项后，编译时将程序中所有异常处理函数地址编入一张安全S.E.H表，并将该表放到程序的映像里面。
  - 当调用异常处理函数时，将检查所调用函数的地址是否位于安全 S.E.H 表中。

2.4 异常溢出防范技术——SafeSEH

- 操作系统层面的实现
  - 检查异常处理链是否位于当前程序的栈中，若否，程序将终止异常处理函数的调用。
  - 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。
  - 在前面两项检查都通过后，程序调用RtlIsValidHandler()，对异常处理函数的有效性进行验证

2.4 异常溢出防范技术——SafeSEH

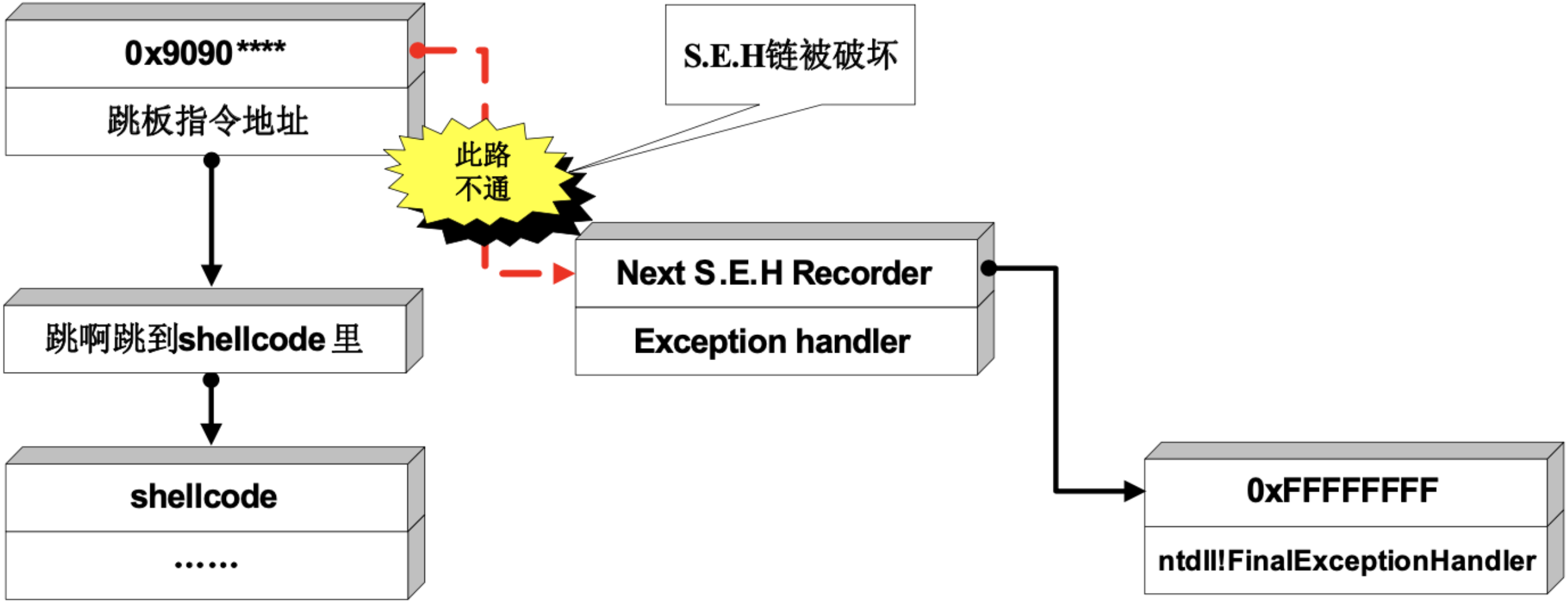
- SafeSEH的缺点
  - Requires changing and rebuilding/compiling executables.
  - Application modules are not typically compiled with SafeSEH by default. Any module loaded by an application that was not compiled with SafeSEH can be used for your SEH overwrite.

2.5 异常溢出防范技术——SEHOP

- SEHOP: Structured Exception Handling Overwrite Protection
- Rather than require code changes, SEHOP works at run time.
- SEHOP的技术思路：
  - verifies that a thread's exception handler chain is intact (can be navigated from top to bottom) before calling an exception handler.

2.5 异常溢出防范技术——SEHOP

- SEHOP does this by
  - adding a custom record to the end of the SEH chain.
  - Prior to executing an exception handler, the OS ensures this custom record can be reached by walking the chain from top to bottom.



As a result, overwriting the SEH address would break the chain and trigger SEHOP, rendering the SEH exploit attempt ineffective.

2.5 异常溢出防范技术——SEHOP

- SEHOP的应用情况

  Was introduced in Windows Vista SP1 and is available on subsequent desktop and servers versions. It is enabled by default on Windows Server Editions (from 2008 on) and disabled by default on desktop versions.

三、 C++对象溢出漏洞机理与利用

3.1 C++对象模型中的虚表机制

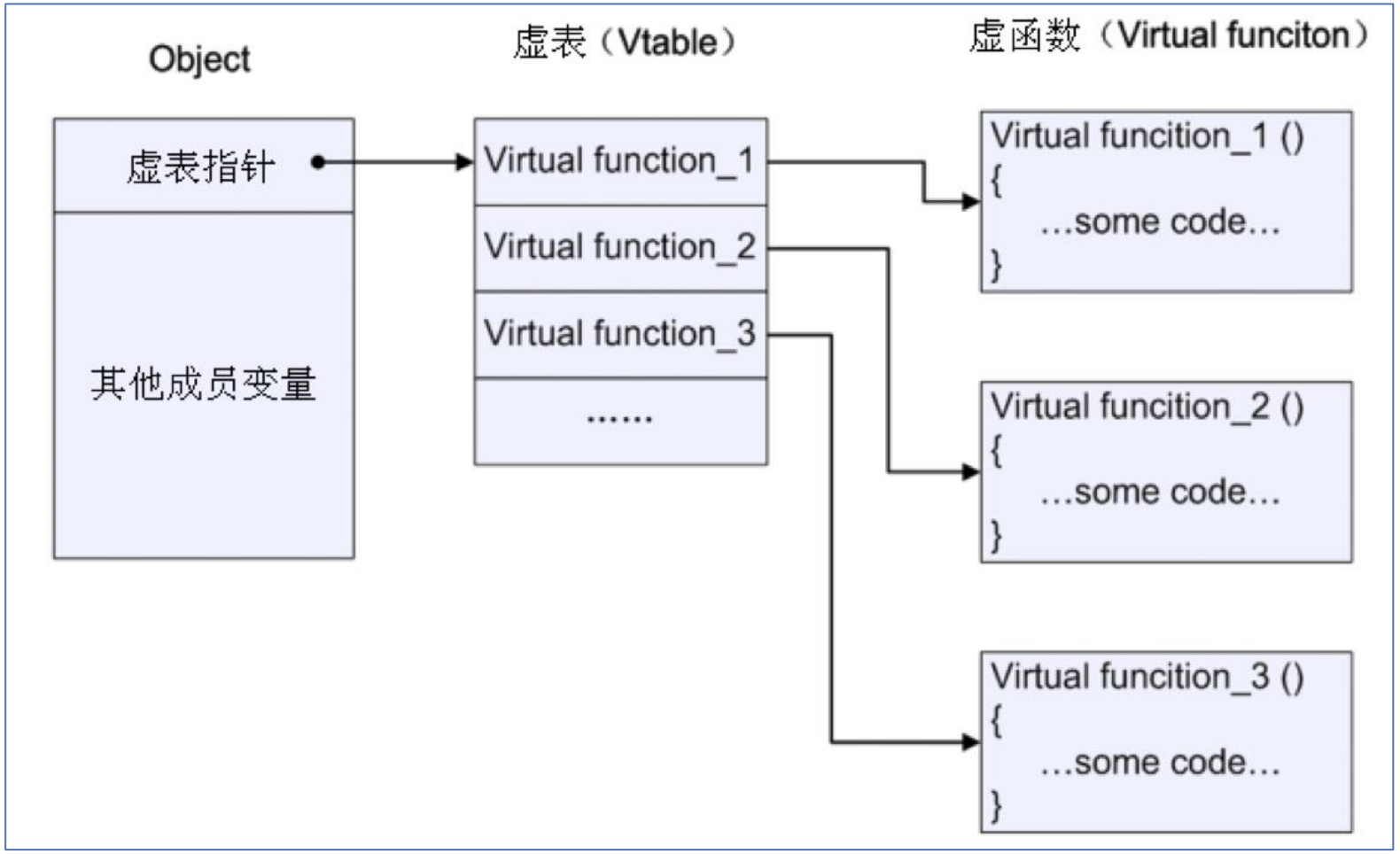
- C++类的成员函数在声明时,若使用关键字 virtual 进行修饰,则被称为虚函数。
- 一个类中可能有多个虚函数。虚函数的入口地址被统一保存在虚表(Vtable)中。

3.1 C++对象模型中的虚表机制

- 子类的虚表与父类的虚表
  - 若子类未重写虚函数，则其虚表中的虚函数地址指向父类中的对应函数。
  - 若子类重写了虚函数，则其虚表中的虚函数地址指向自身的虚函数实现。如果子类自己新增了虚函数，那么其虚表中就会增加该项。
  - 派生类的虚表中虚函数地址的排列顺序和基类的虚表中虚函数地址排列顺序相同。

3.1 C++对象模型中的虚表机制

- 每一个对象内部都有一个虚表指针，指向本类的虚表。
- 虚表指针保存在对象的内存空间中，紧挨着它的是其它成员变量。
- 对象在调用虚函数时，先通过虚表指针找到虚表，然后从虚表中取出要调用函数的地址。
- 不管对象类型如何转换，该对象内部的虚表指针不变，从而实现动态的对象函数调用。



```
65  int main(int argc, const char * argv
66      WhuDemo target;
67      char vulbuf[64];
68      unsigned long dwFakeVt = (unsigne
69
```

3 % 0 项更改 | 0 名作者，0 项更改

视 1

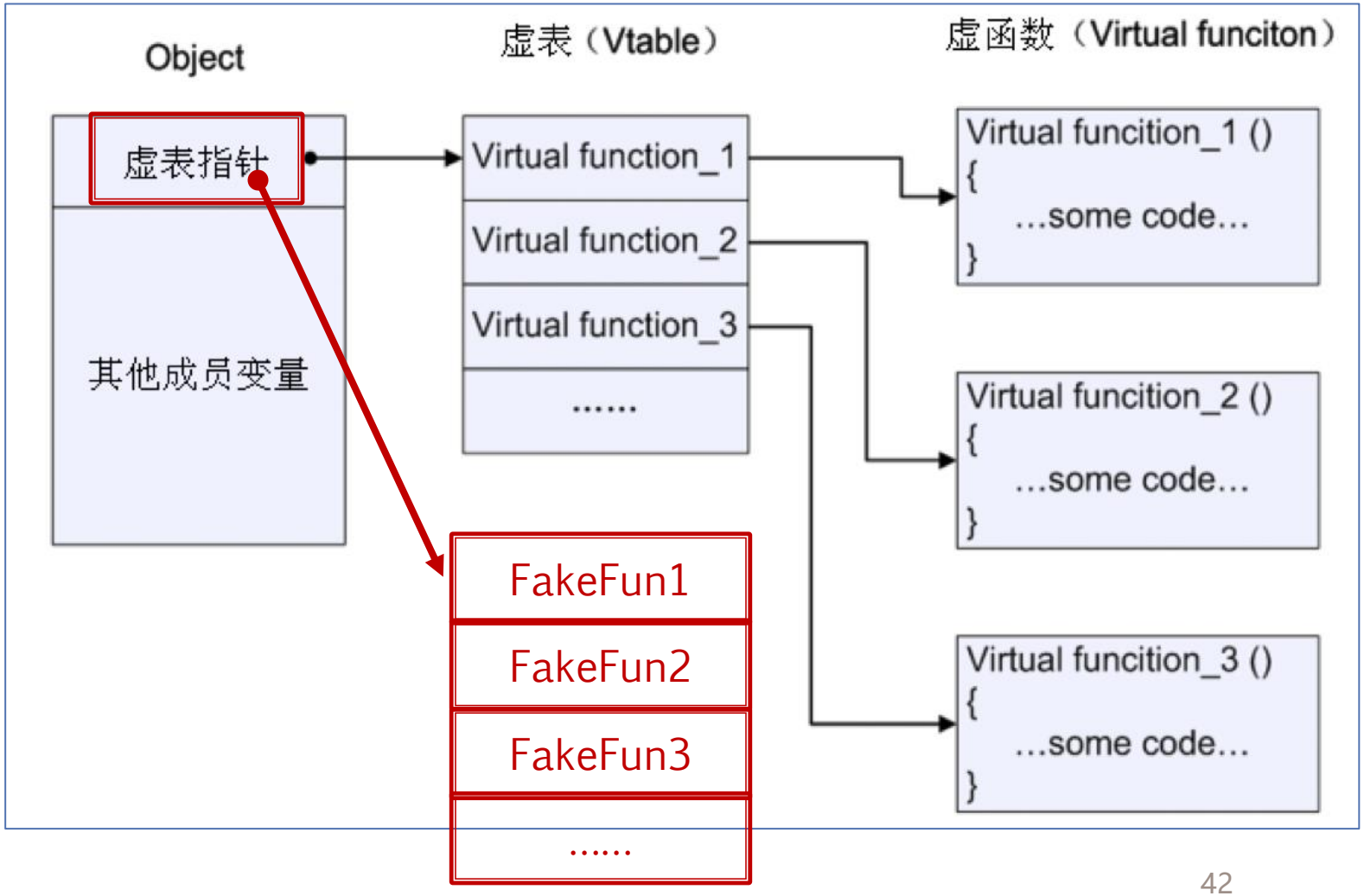
名称	值
target	{m_objectId=0x0019fee4 "S$A" }
__vfptr	0x00418bf0 {overflow_cpp.exe!void(* WhuDemo::`vftab
[0x00000000]	0x004110d2 {overflow_cpp.exe!WhuDemo::test0(void)}
[0x00000001]	0x004110e6 {overflow_cpp.exe!WhuDemo::test1(void)}
[0x00000002]	0x0041115e {overflow_cpp.exe!WhuDemo::test2(void)}
m_objectId	0x0019fee4 "S$A"

```
class WhuDemo {
public:
    char m_objectId[16];
    virtual void test0(void) {
        cout << "WhuDemo::test0()" << endl;
    }

    virtual void test1(void) {
        cout << "WhuDemo::test1()" << endl;
    }

    virtual void test2(void) {
        cout << "WhuDemo::test2()" << endl;
    }
};
```

3.2 C++对象虚表溢出攻击

- 情形一：修改虚表指针

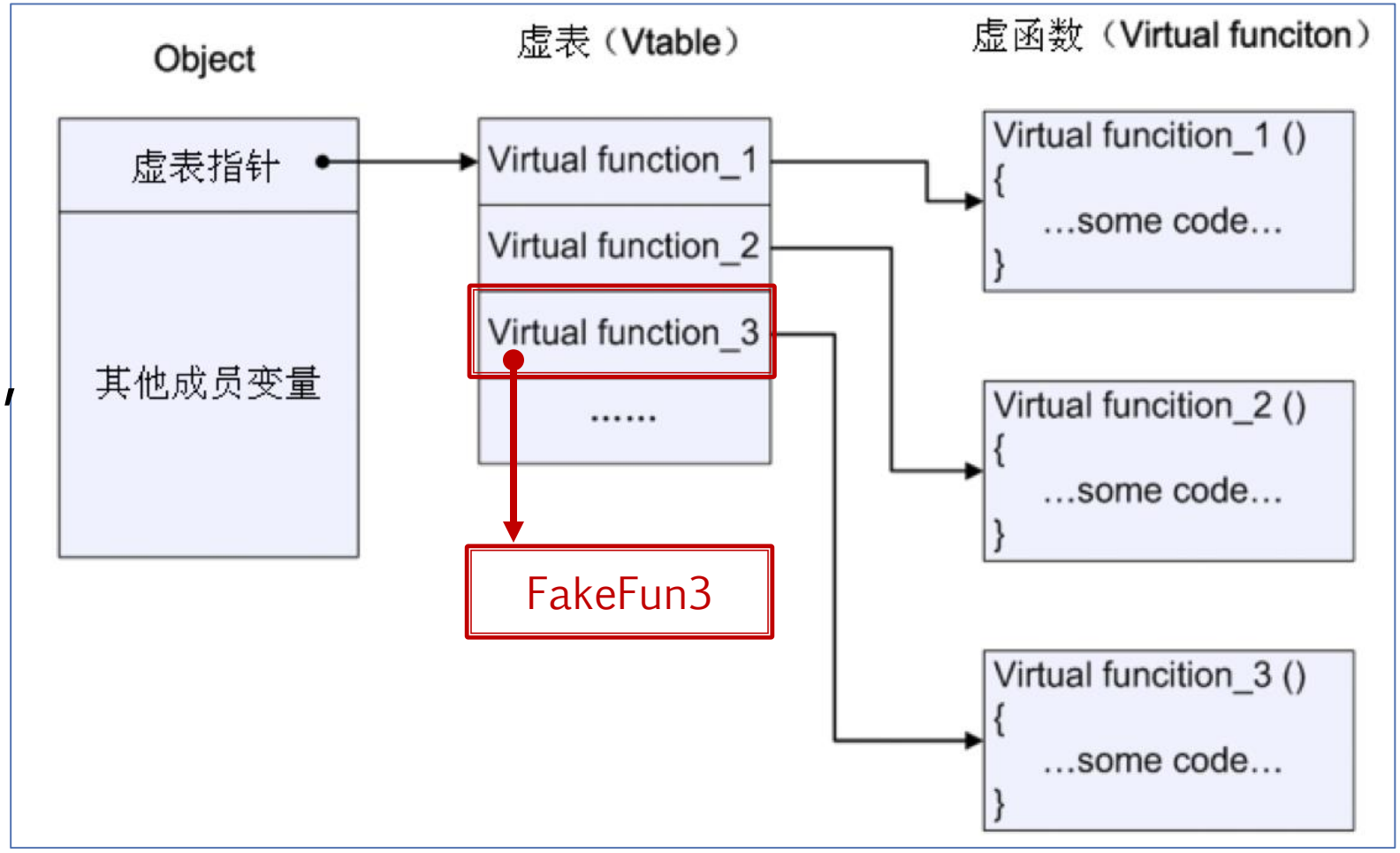
  如果有机会修改对象中的虚表指针，那么在程序调用虚函数时就有机会执行攻击代码。



3.2 C++对象虚表溢出攻击

- 情形二：修改虚表中的虚函数指针

  如果有机会修改虚表中的虚函数指针，那么在程序调用虚函数时就有机会执行攻击代码。



四、整数溢出漏洞机理与利用

4.1 什么是整数溢出

- An integer overflow or wraparound occurs when an integer value is incremented to a value that is too large to store in the associated representation.
- When this occurs, the value may wrap to become a very small or negative number.
- While this may be intended behavior in circumstances that rely on wrapping, it can have security consequences if the wrap is unexpected.

2019 CWE Top 25 Most Dangerous Software Errors

Rank	Name	Score
[1]	Improper Restriction of Operations within the Bounds of a Memory Buffer	75.56
[2]	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.69
[3]	Improper Input Validation	43.61
[4]	Information Exposure	32.12
[5]	Out-of-bounds Read	26.53
[6]	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	24.54
[7]	Use After Free	17.94
[8]	Integer Overflow or Wraparound	17.35
…	……	……

4.2 整数溢出的几种情形

上溢：一个整数运算后所得的结果超过了它能表示的上限，结果变成一个很小的数或负数。

例1：unsigned short a = 65535;

a = a + 5的计算结果为4

分析：$(65535)_{10} + (5)_{10} = (1111\ 1111\ 1111\ 1111)_2 + (101)_2$

$= (1\ 0000\ 0000\ 0000\ 0100)_2$

unsigned short 长度为16位，最高位的1被丢弃，只保留低16位，即$(0000000000000100)_2 = (4)_{10}$

复习：整数的二进制表示

- 原码

  一个整数，按照绝对值大小转换成的二进制数，称为原码。

- 反码

  将二进制数按位取反，所得的新二进制数称为原二进制数的反码。

- 补码

  反码加1称为补码。

- 负数的二进制表示

  负数以其绝对值的补码形式表达。

4.2 整数溢出的几种情形

例2：short a = 32767;

a = a+5 的计算结果为-32764

分析：

$(32767)_{10} + (5)_{10} = (0111111111111111)_2 + (0000000000000101)_2$

$= (1000000000000100)_2$

由于short最高位为符号位，0表示正数，1表示负数，上式运算后超出了short类型的数据范围，造成上溢，即：

$(1000\ 0000\ 0000\ 0100)_2 = (-32764)_{10}$

4.2 整数溢出的几种情形

下溢：一个整数运算后所得的数低于它能表 示的下限，结果变成一个很大的数。

例3： short a = -4;

unsigned short b = a;

实际结果为: b = 65532

分析：负数在计算机中以其绝对值的补码表示

$(-4)_{10} = (1111111111111100)_2$

b为无符号数，所有位全部用来表示数据，因此

$b = (1111111111111100)_2 = (65532)_{10}$

4.2 整数溢出的几种情形

截断：将数据放入了比它本身小的存储空间中， 从而导致溢出。

例4：int a = 65537;

short b = a;

实际结果为：b = 1

分析：

$(65537)_{10} = (1\ 0000\ 0000\ 0000\ 0001)_2$

截断后：b = $(\cancel{1}0000\ 0000\ 0000\ 0001)_2 = 1$

4.2 整数溢出的几种情形

符号问题引发的整数溢出：

当无符号整数和有符号整数进行比较或运算时，会将有符号整数转化成无符号整数，之后再进行比较或运算，可能发生的上溢和下溢会导致与事实相反的结论。

例5：short a = -5;

unsigned short b = 5;

实际比较结果：a > b

分析：

-5转化成无符号整数后等于65531

4.3 整数溢出导致危害的情形

- The integer overflow can be triggered using user-supplied inputs.
- This becomes security-critical when the result is used to
  - control looping, （作为循环变量）
  - make a security decision, （作为安全判决条件）
  - or determine the offset or size in behaviors such as memory allocation, copying, concatenation, etc.

    （作为指示内存分配、拷贝和连接等操作的位置和大小）

4.4 整数溢出漏洞举例

```
1  BOOL fun(size_t cbSize)
2  {
3      if( cbSize > 1024 )
4      {
5          return FALSE;
6      }
7      char *pBuf = new char[cbSize-1];      //cbSize = 0
8      // 存在溢出隐患
9      // memset第三个参数是无符号数
10     // 此时cbSize - 1 = 0xFFFFFFFF空间过大导致程序崩溃
11     memset(pBuf, 0x90, cbSize-1);
12     //...
13     return TRUE;
14 }
```

4.4 整数溢出漏洞举例

```
1  int copy_something(char *buf, int len)  //int len有符号数
2  {
3      char szBuf[800];
4      if(len > sizeof(szBuf))  //len为有符号的负数，跳过if
5      {
6          return -1;
7      }
8      // 存在溢出隐患
9      return memcpy(szBuf, buf, len);  //len参数作为无符号解析，变为很大的数
10 }
```

4.4 整数溢出漏洞举例

```
1  BOOL fun(byte *name, DWORD cbBuf)  //cbBuf = 0x0010 0020
2  {
3      unsigned short cbCalculatedBufSize = cbBuf;  // cbCxxx = 0x0020
4      byte *buf = new byte(cbCalculatedBufSize);   //buf长度为0x20
5      if(buf == NULL)
6      {
7          return FALSE;
8      }
9      // 存在溢出隐患
10     memcpy(buf, name, cbBuf);   //发生溢出
11     //...
12     return TRUE;
13 }
```

- 如果cbBuf值大于65535，该值存入cbCalculatedBufSize时将被截断并变小。
- buf的大小为cbCalculatedBufSize，但向buf拷入的数据长度却仍然为cbBuf，因此发生溢出。

4.4 整数溢出漏洞举例

```
nresp = packet_get_int();
if (nresp > 0) {
        response = xmalloc(nresp*sizeof(char*));
        for (i = 0; i < nresp; i++)
                response[i] = packet_get_string(NULL);
}
```

分析：

- 当32位整数 nresp = 1073741824，sizeof(char*) =4，nresp*sizeof(char*) =0x100000000，发生溢出，实际计算结果为0。
- 此时发生了xmalloc(0)调用。
- xmalloc允许分配大小为0的内存，从而导致后面for循环中的response赋值发生堆溢出。

OpenSSH 3.3中的一个整数溢出漏洞代码

五、 格式化串漏洞机理与利用

5.1 格式化函数

- 常见的格式化函数
  - printf — prints to the ‘stdout’ stream
  - sprintf — prints into a string
  - fprintf — prints to a FILE stream
  - snprintf — prints into a string with length checking
  - vfprintf — print to a FILE stream from a va_arg structure
  - vprintf — prints to ‘stdout’ from a va_arg structure
  - vsprintf — prints to a string from a va_arg structure
  - vsnprintf — prints to a string with length checking from a va_arg structure

5.1 格式化函数

- 格式化函数的用途
  - used to convert simple C datatypes to a string representation
  - allow to specify the format of the representation
  - process the resulting string (output to stderr, stdout, syslog, ...)

5.2 格式串在格式化函数中的作用

int printf (const char * format, ...);

specifier	Output
%d *or* %i	Signed decimal integer
%u	Unsigned decimal integer
%o	Unsigned octal
%x	Unsigned hexadecimal integer
%X	Unsigned hexadecimal integer (uppercase)
%f	Decimal floating point, lowercase
%F	Decimal floating point, uppercase
%e	Scientific notation (mantissa/exponent), lowercase
%E	Scientific notation (mantissa/exponent), uppercase

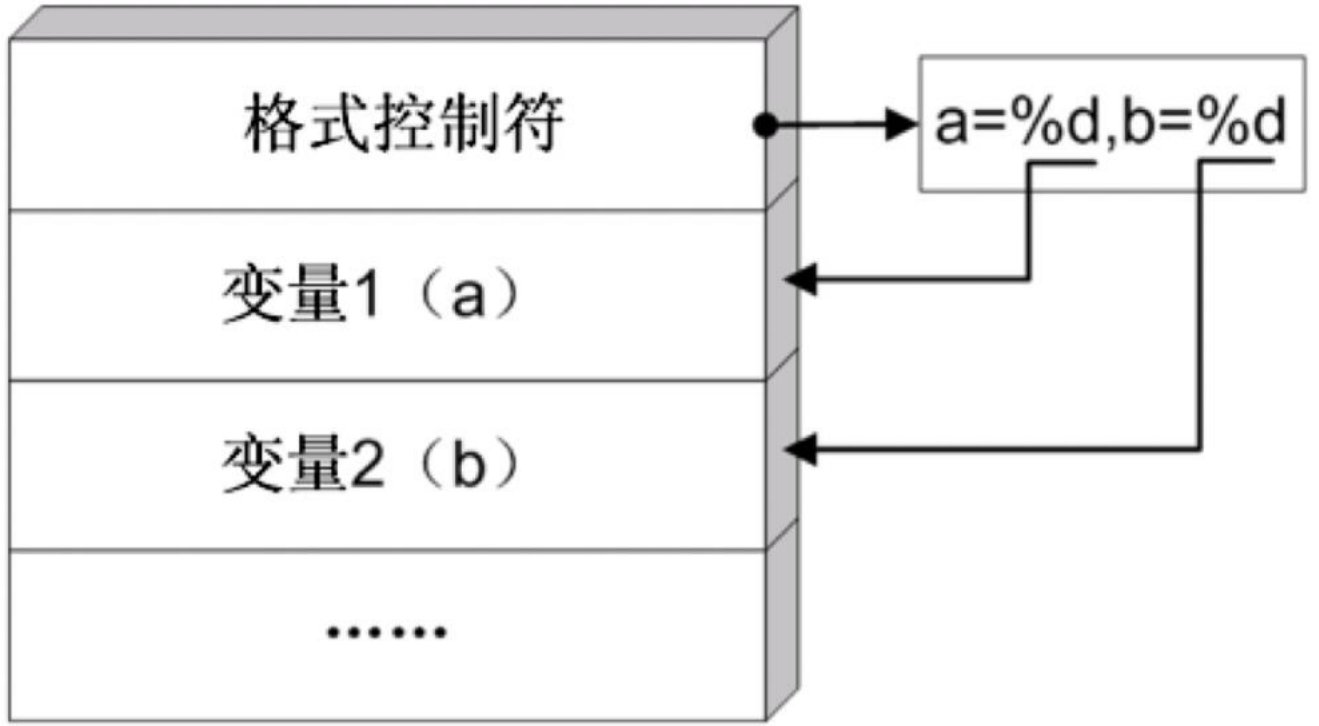
specifier	Output
%g	Use the shortest representation: %e or %f
%G	Use the shortest representation: %E or %F
%a	Hexadecimal floating point, lowercase
%A	Hexadecimal floating point, uppercase
%c	Character
%s	String of characters
%p	Pointer address
%n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.

http://www.cplusplus.com/reference/cstdio/printf/

5.3 格式串与格式化函数的栈帧

printf (”a=%d, b=%d\n”, a, b);

低址

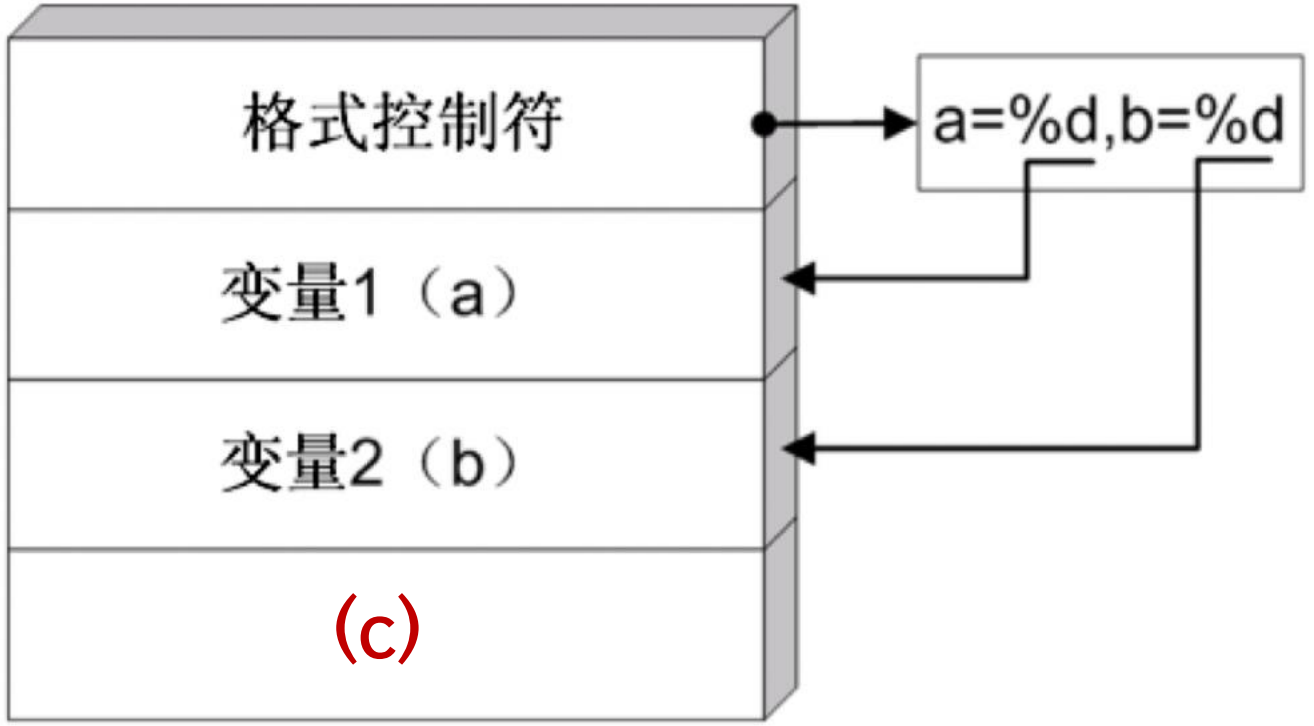


高址

5.3 格式串与格式化函数的栈帧

printf (”a=%d, b=%d, c=%d\n”, a, b);

栈的低址



栈的高址

格式串与格式化函数实际调用参数不匹配会发生什么?

① 编译出错?

② 调用失败?

③ ……

5.4 格式串漏洞可能导致的安全问题

- 栈内信息泄露
- 引发缓冲区溢出

代码演示

overflow_format_string

参考资料

Matt Pietrek
https://blog.csdn.net/LG1259156776/article/details/51472597

二进制各种漏洞原理实战分析总结
https://mp.weixin.qq.com/s/6GVhuXOjChDj19SITqpM7Q

THE END