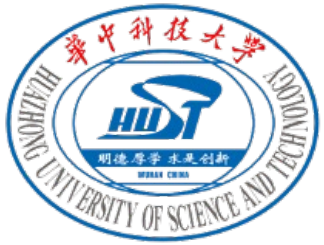


华中科技大学
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

网络空间安全学院

4.1 缓冲区溢出之栈溢出

网络空间安全学院 慕冬亮

Email : dzm91@hust.edu.cn

[1] Smashing The Stack For Fun And Profit

缓冲区溢出的历史

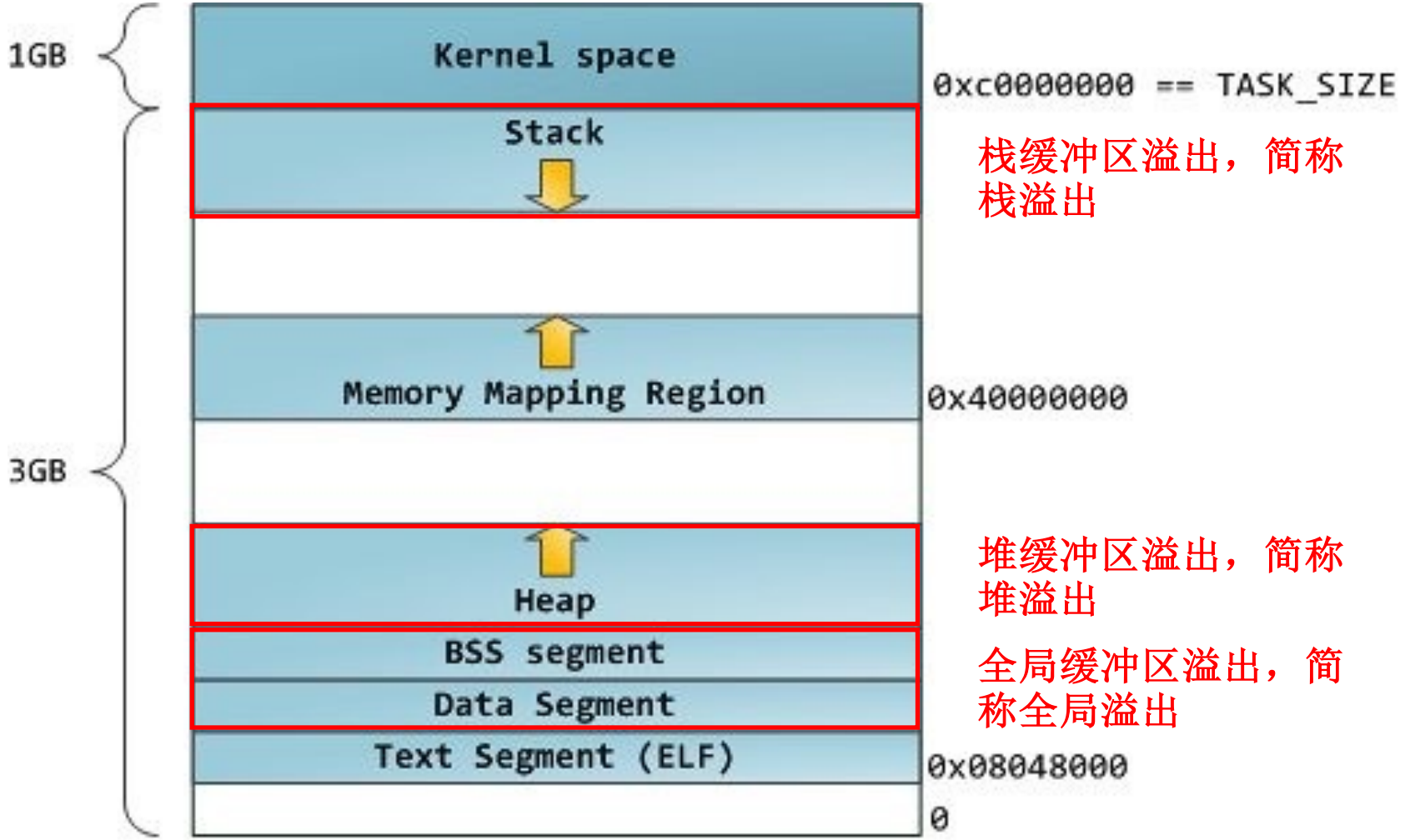
- 缓冲区溢出最早发现于1970s
- 首次应用于1988年的 Morris 蠕虫，并对造成大范围的攻击
- 直到今天，它仍然是一个没有完全解决的安全问题

fingerd

Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection
2017	14714	3157	3004	2491	745	508
2018	16557	1853	3041	2121	400	517
2019	17344	1344	3201	1263	488	551
2020	18325	1351	3248	1561	409	462
2021	20149	1838	3846	1679	484	738
2022	5670	651	1118	541	86	249
Total	172110	27875	41858	21538	6511	9446

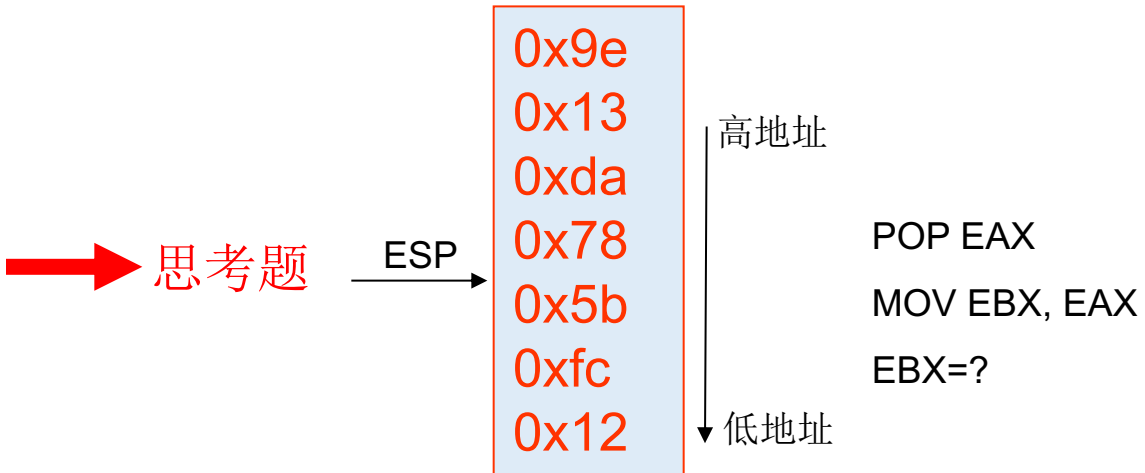
[1] https://www.cvedetails.com/vulnerabilities-by-types.php

进程地址空间分布


1GB
3GB
Kernel space
Stack
Memory Mapping Region
Heap
BSS segment
Data Segment
Text Segment (ELF)
0xc0000000 == TASK_SIZE
0x40000000
0x08048000
0
栈缓冲区溢出，简称栈溢出
堆缓冲区溢出，简称堆溢出
全局缓冲区溢出，简称全局溢出

栈

- 栈是一块连续的内存空间
  - 先入后出 （FILO, First In Last Out）
  - 生长方向与内存的生长方向正好相反, 从高地址向低地址生长
- 每一个线程有自己的栈
  - 提供一个暂时存放数据的区域
- 使用POP/PUSH指令来对栈进行操作
- 使用ESP/RSP寄存器指向栈顶，EBP/RBP指向栈帧底



栈内容

- 函数的参数
- 函数返回地址
- 当前正在执行的函数的局部变量
- EBP/RBP的值
- 一些通用寄存器的值

三个重要的寄存器

- SP(ESP,RSP)
  - 即栈顶指针，随着数据入栈出栈而发生变化
- BP(EBP,RBP)
  - 即基地址指针，用于标识栈中一个相对稳定的位置。通过BP,可以方便地引用函数参数以及局部变量
- IP(EIP,RIP)
  - 即指令寄存器，在将某个函数的栈帧压入栈中时，其中就包含当前的IP值，即函数调用返回后下一个执行语句的地址

函数调用过程

- 把参数压入栈
- 保存指令寄存器中的内容，作为返回地址
- 放入堆栈当前的基址寄存器
- 把当前的栈指针(ESP)拷贝到基址寄存器，作为新的基地址
- 为本地变量留出一定空间，把ESP减去适当的数值

函数调用类型

函数调用规则指的是调用者和被调用函数间传递参数及返回参数的方法，常用的有stdcall,cdecl,Fast Call.

- __cdecl C调用规则：

1．在后面的参数先进入堆栈；

2．在函数返回后，调用者要负责清除堆栈．所以这种调用常会生成较大的可执行程序.

- __stdcall 又称为WINAPI， 其调用规则：

1．在后面的参数先进入堆栈；

2．被调用的函数在返回前自行清理堆栈，所以生成的代码比cdecl小.

- Fast Call

1. 是把函数参数列表的前2个参数放入寄存器,其他参数压栈
2. 后面参数先入栈

函数调用类型

- __cdecl C调用规则

```
int __cdecl CFunc(int i, int j)  {  return i+j;  }
```

- __stdcall 又称为WINAPI

```
int _stdcall DFunc(int i, int j)  {  return i*j;  }
```

- Fast Call

```
int _fastcall EFunc(int i,int j,int k,int l,int m)  {  return i*j*k*l*m;  }
```

函数调用中栈的工作过程

- 调用函数前
  - 压入栈
    - 上级函数传给A函数的参数
    - 返回地址(EIP)
    - 当前的EBP
    - 函数的局部变量
- 调用函数后
  - 恢复EBP
  - 恢复EIP
  - 局部变量不作处理

例子

```
int main()
{
    AFunc(5, 6);
    return 0;
}
```

```
int AFunc(int i,int j) {
    int m = 3;
    int n = 4;

    m = i;
    n = j;

    BFunc(m,n);

    return 8;

}
```

```
int BFunc (int i, int j) {

    int m = 1;

    int n = 2;

    m = i;

    n = j;

    return m;

}
```

详见实例 test_stack_layout

当缓冲区溢出发生时……

```
int AFunc(int i, int j)
{
    int m = 3;
    int n = 4;
    char szBuf[8] = {0};
    strcpy(szBuf, “stack overflow!”);
    m = i;
    n = j;
    BFunc(m,n);
    return 8;
}
```

看实例 test_stack_overflow!

```
char *strcpy(char *dest, const char *src);

char *strncpy(char *dest, const char *src, size_t n);
```

Description

The **strcpy**() function copies the string pointed to by *src*, including the terminating null byte ('\0'), to the buffer pointed to by *dest*. The strings may not overlap, and the destination string *dest* must be large enough to receive the copy. *Beware of buffer overruns!* (See BUGS.)

如果 strcpy 的第二个参数变长之后，如“let us test stack overflow”，程序会如何？

栈溢出特点

- 特点
  - 缓冲区在栈中分配
  - 拷贝的数据过长
  - 覆盖了缓冲区相邻的一些重要数据结构、函数指针

缓冲区溢出的根本原因

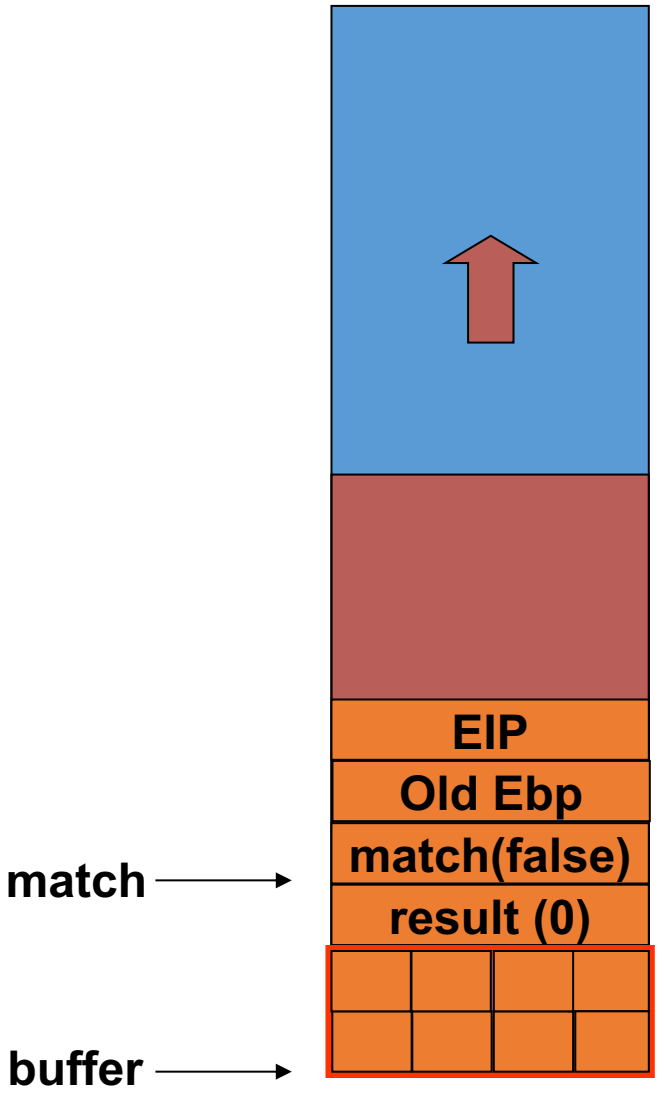
- 没有内嵌支持的边界保护
  - User funcs
  - Ansi C/C++: strcat(), strcpy(), sprintf(), vsprintf(), bcopy(), gets(), scanf()…
- 程序员安全编程技巧和意识缺乏
- 那么栈缓冲区溢出到底如何利用呢？
  - 这取决于栈上相邻数据 （无通用利用方法）
    - 局部变量
    - 函数指针 等
  - 除此之外，栈上面有一种漏洞利用的敏感数据

栈溢出的利用方式(1)

```
bool match = false; int result;
char buffer[8];

printf("3 + 5 = ?\n");
scanf("%s", buffer); // vulnerable scanf
result = atoi(buffer);
if (result == 3+5) match = true;

if (match) {
   printf("The answer is correct\n");
} else {
   printf("The answer is incorrect\n");
}
```



看实例 test_overflow_local_variable!

栈溢出的利用方式(2)

```
int AFunc(int i, char* password)
{
   int m = 3;
   char buffer[8] = {0};
    strcpy(buffer, password);
    *(int *)((char *)buffer+X) = BFunc;
    m = i;
    BFunc(m,6);
    return 8;
}
```

用BFunc的地址替换正常的AFunc返回地址，使程序运行至Bfunc, X=?

看实例 test_pwn/x86_64

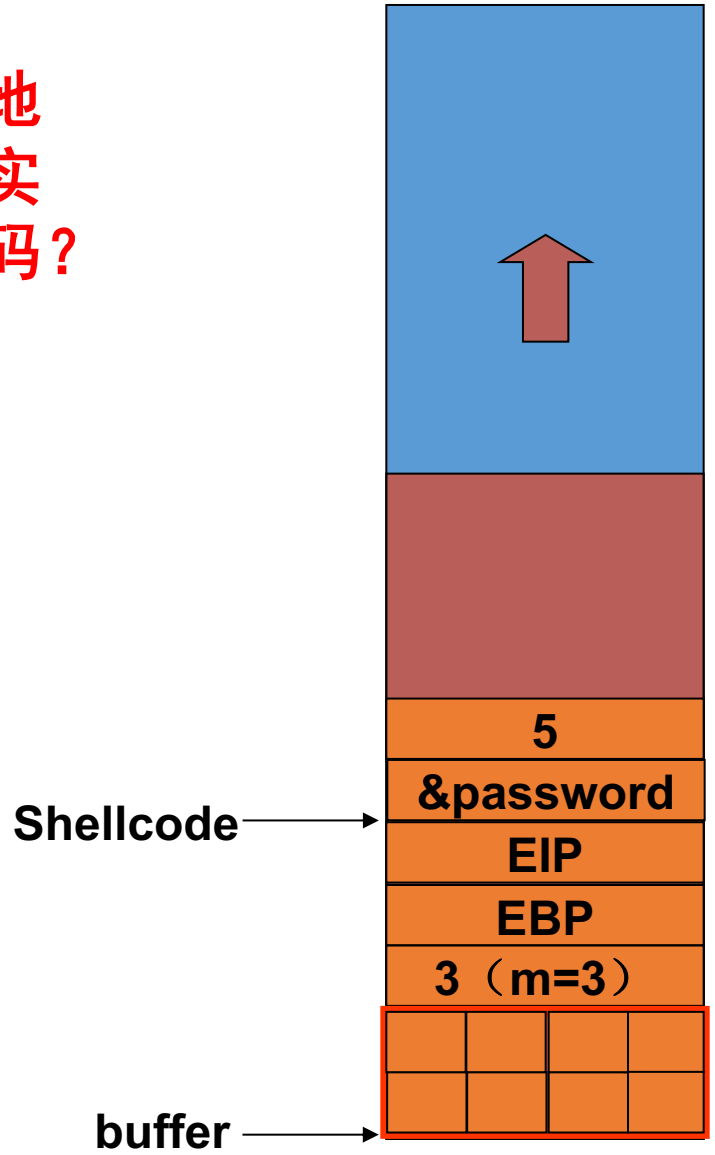
栈溢出的利用方式(2)

前面的例子是在有源代码的情况下进行地址赋值，进入到自定义函数执行，但现实中没有源代码，如何才能执行自定义代码？

char buffer[8] = {0};

strcpy(buffer, password);

- password 的内容：
  - 对 EIP 的填充
  - Shellcode
- 如何进入到 ShellCode?

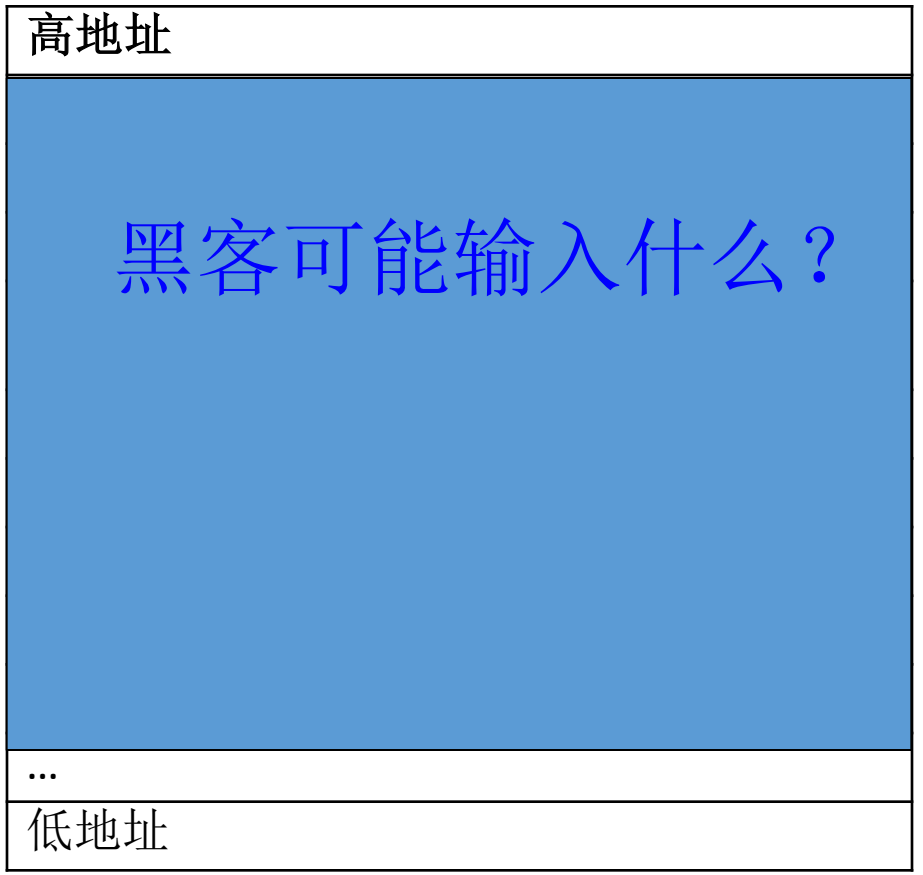


栈溢出利用

- 正常栈帧结构

高地址
…
&password
i
函数返回地址
Old EBP
m
buffer[4-7]
buffer[0-3]
…
低地址

精心设计输入内容（Payload）



- 黑客输入字串导致的异常栈帧

ShellCode 与 Payload

- Shellcode，是指能完成特殊任务的自包含的二进制代码，根据不同的任务可能是发出一条系统调用或建立一个高权限的Shell，Shellcode也就由此得名。
  - 它的最终目的是取得目标机器的控制权，所以一般被攻击者利用系统的漏洞送入系统中执行，从而获取特殊权限的执行环境，或给自己设立有特权的帐户。

- 与Shellcode相关的还有Payload，在漏洞利用时，一般把Shellcode以及实现跳转到Shellcode的那部分填充代码合称为Payload。

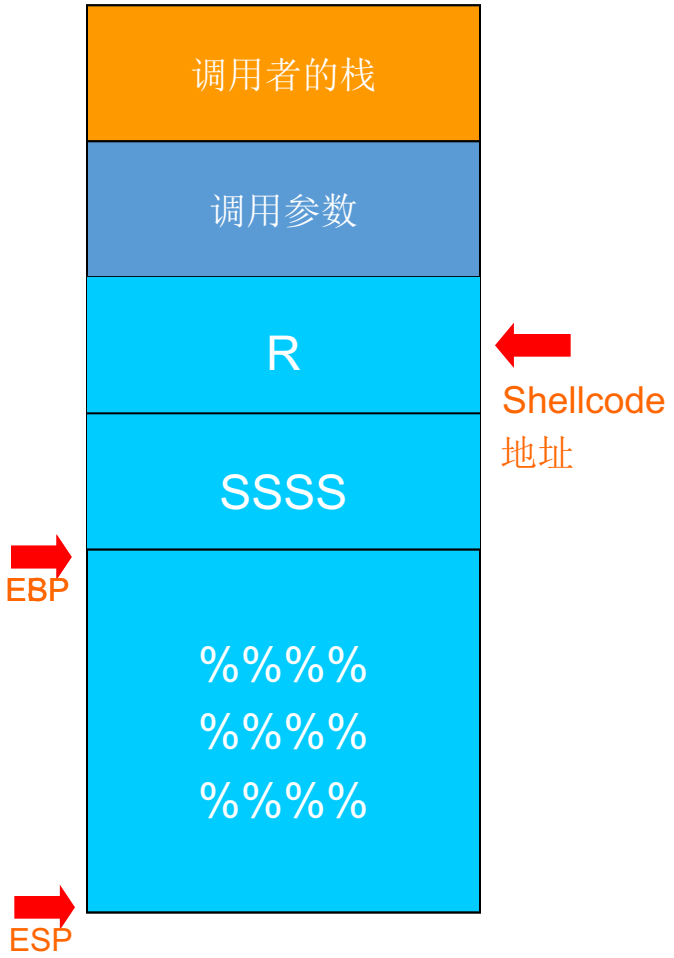
- 由于两者意义相差不大，现在也有很多人将Payload简称为Shellcode。

以何种姿势设计Payload？

- NSR模式

R指向了Shellcode地址，但执行“mov esp,ebp”恢复调用者栈信息时，Win32会在被废弃的栈中填入一些随机数据。

WE LOST SHELLCODE!!!



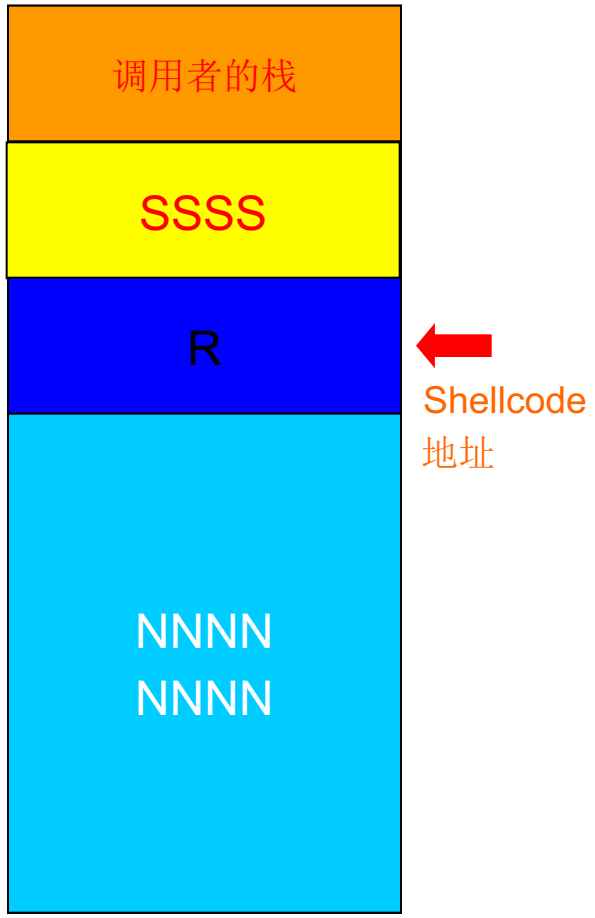
Win32栈地址含有空字节

➢NRS模式

栈在1G(~0x00FFFFFF)以下

如果R直接指向Shellcode，则在R中必然含有空字节‘\0’.

地址R中含空字节



Shellcode将被截断

we lost shellcode AGAIN!!

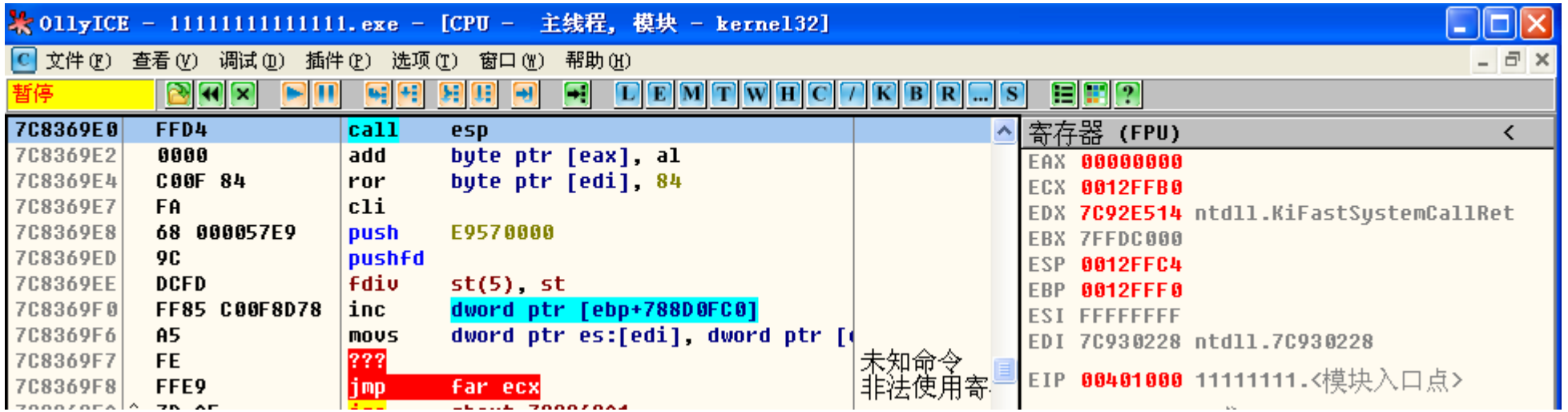
如何解决?

- 通过Jmp/Call ESP指令跳转
  - 1998: Dildog-提出利用栈指针的方法完成跳转
  - 1999: Dark Spyrit-提出使用系统核心DLL中的Jmp ESP指令完成跳转

- 跳转指令在哪?
  - 代码页里的地址: 受系统版本及SP影响。
  - 应用程序加载的用户DLL，取决于具体的应用程序，可能较通用。
  - 系统未变的DLL，特定发行版本里不受SP影响，但不同语言版本加载基址可能会不同。

返回地址的选择

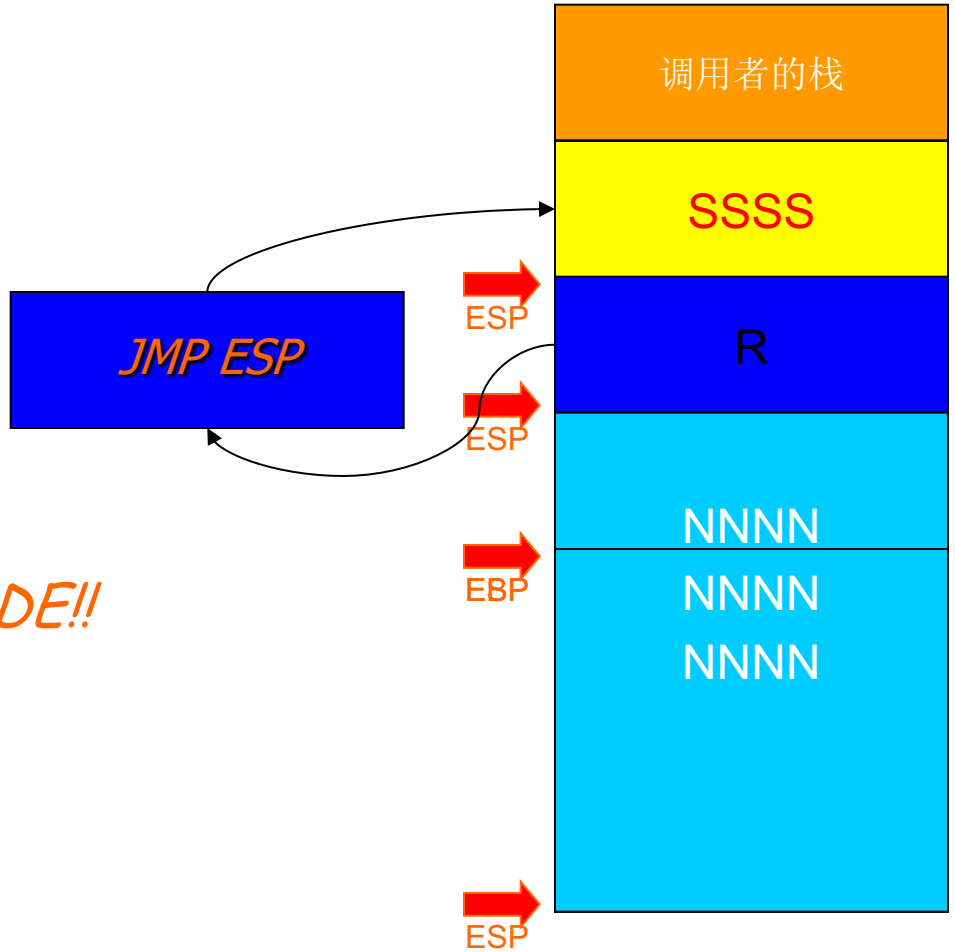
- Jmp esp, 机器码为0x FF E4
- 同理，call esp[0xFFD4], jmp ebp，其他指向栈帧的寄存器都可以，如EAX, EBX, ESI。
- ESP or EBP 指向哪里?



通过跳转指令执行Shellcode

- 如何利用跳转指令让漏洞程序正确执行我们的Shellcode

```
0040100F  |. E8 0C000000    CALL
00401014  |. 83C4 08        ADD ESP,8
00401017  |. 8BE5           MOV ESP,EBP
00401019  |. 5D             POP EBP
0040101A  \. C3             RETN
```



NOW ESP POINTS TO SHELLCODE!!

栈溢出利用的其它方法

- JMP ESP在有些情况下不会成功

  比如：

  (1)当JMP ESP回来时候，开始的Shellcode可能被修改，则无法按照要求继续执行.

  (2)函数在返回之前发生了异常，根本无法调用返回语句.

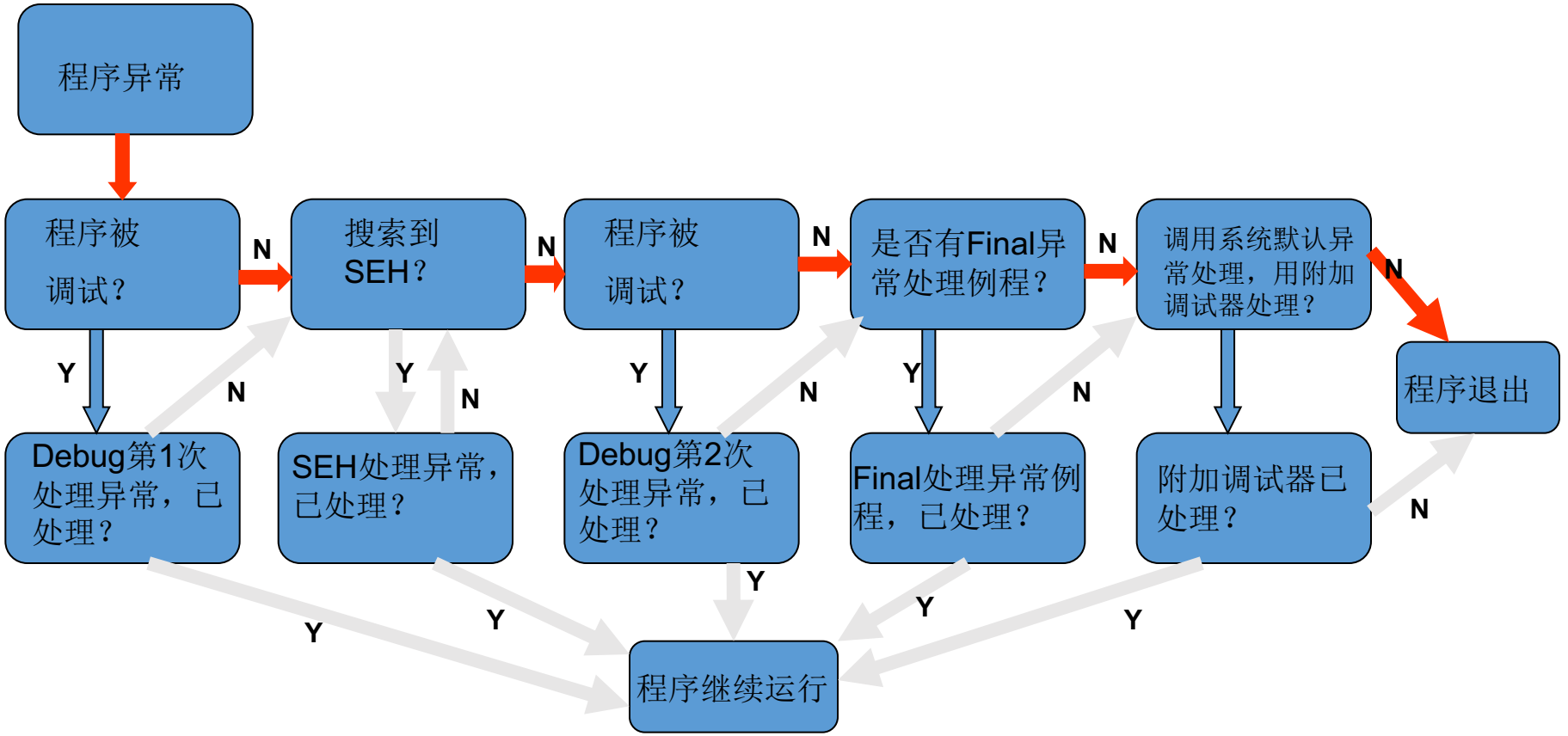
- 可采用SEH进行攻击与利用

  (1)SEH为结构化异常处理

  (2)SEH结构一般在栈空间附近，可以被覆盖到

SEH一般处理流程

- 程序发生异常时候系统处理流程



SEH一般处理流程

- 异常处理SEH结构示意图

(1)异常处理初始地址放在fs:[0];

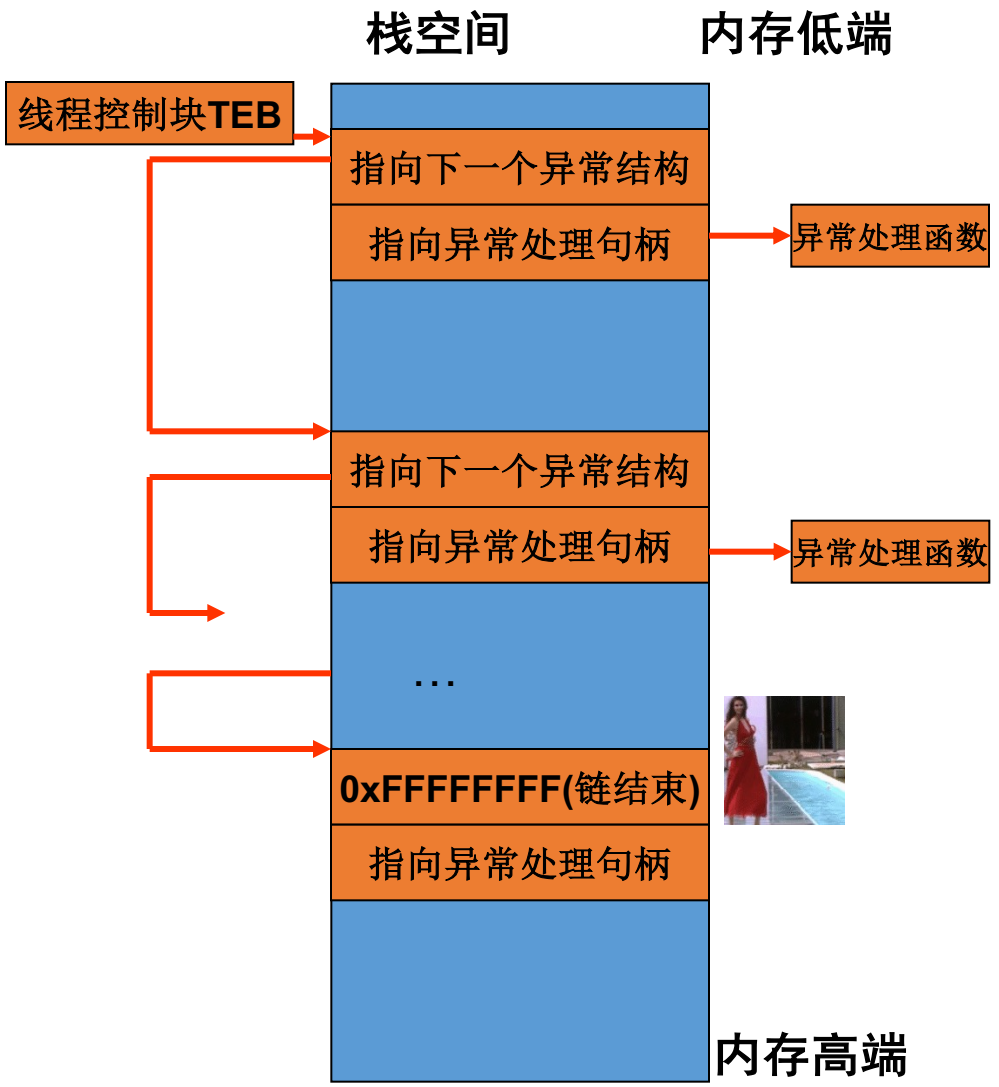
(2)在分析漏洞时候，可以查看fs:[0]与栈空间的距离，以决定需要覆盖的距离;

(3)如果系统还没有进入到异常处理，

win2000:EBX则放的是第一个异常处理函数地址的前4个字节的地址;

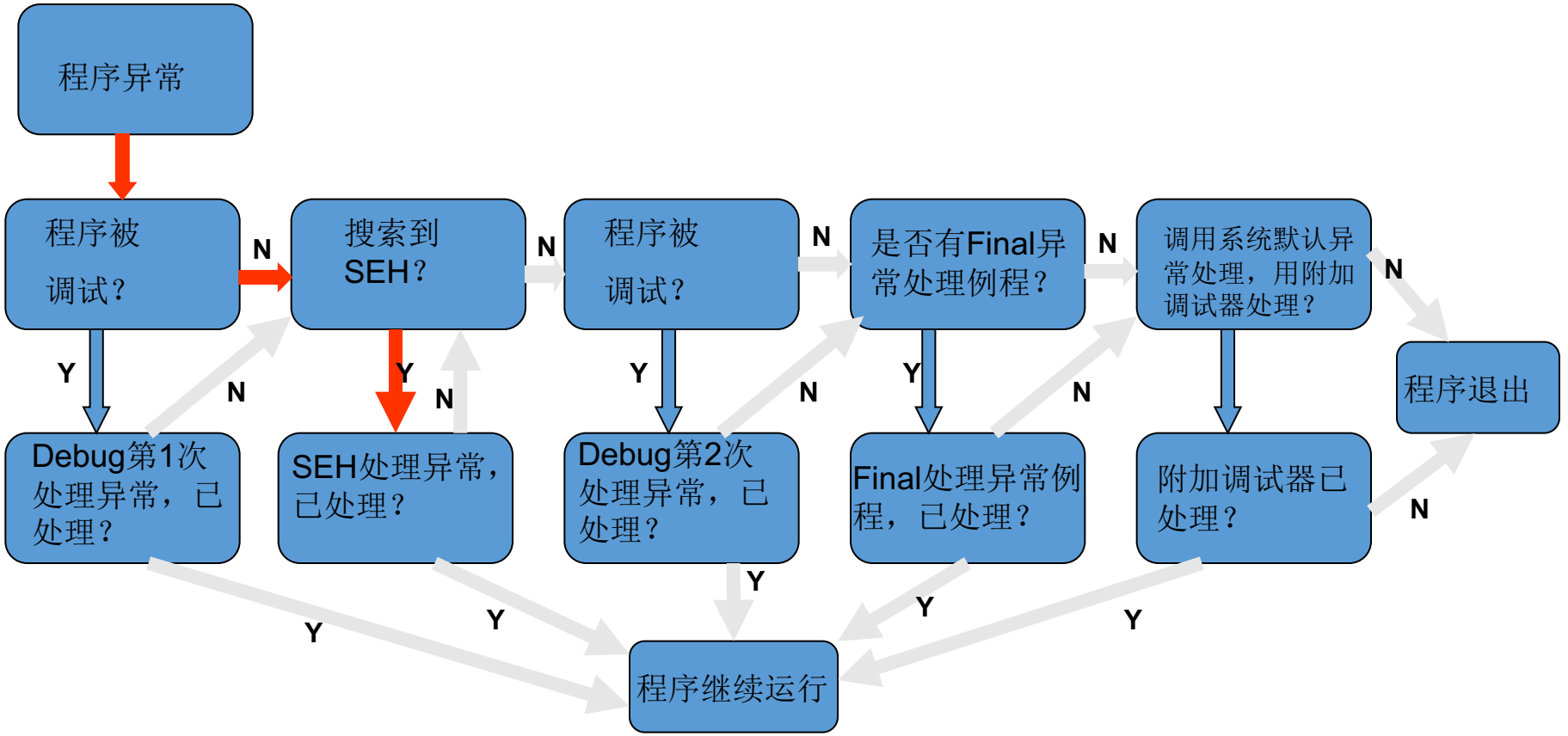
win7/8/10:以当时分析的为准。

需要实时分析才行!



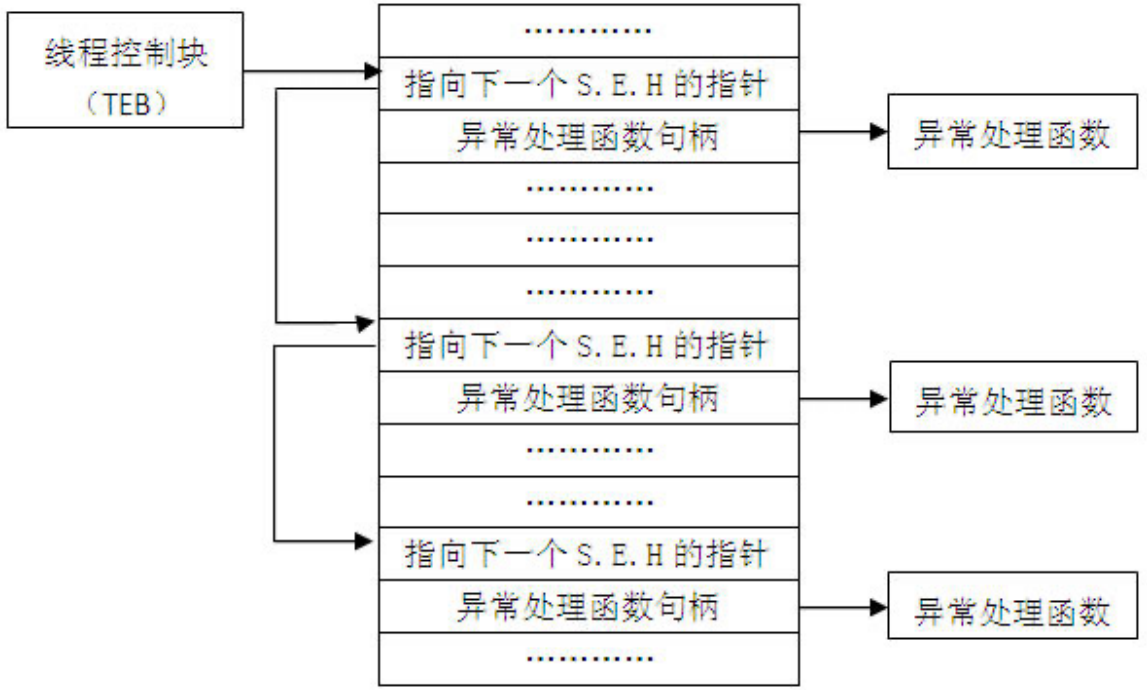
SEH一般处理流程

- 覆盖异常处理后处理流程



S.E.H结构覆盖

- SHE 保存在栈中，对其进行覆盖.
- Exploit时返回前异常触发，程序流进入异常处理
- 异常处理句柄赋值为 类似于JMP ESP的跳转指令,指向ShellCode



ShellCode

- Shellcode一般是作为数据形式发送给服务器制造溢出得以执行代码并获取控制权的。不同的漏洞利用方式，对于数据包的格式都会有特殊的要求，Shellcode必须首先满足被攻击程序对于数据报格式的特殊要求。

- 从总体上来讲，Shellcode具有如下一些特点：
  - 长度受限
  - 不能使用特定字符，例如\x00等
  - API函数自搜索和重定位能力。由于shellcode没有PE头，因此shellcode中使用的API和数据必须由shellcode自己进行搜索和重定位
  - 一定的兼容性。为了支持更多的操作系统平台，shellcode需要具有一定的兼容性 。

ShellCode功能

- 常见功能
  - 下载程序并运行
  - 添加管理员账号
  - 开启Shell(正向、反向)
  - …

[1] https://www.exploit-db.com/shellcodes

课程小结

- 对子函数的调用过程
  - 子函数的调用与返回
  - 栈帧的创建与销毁
  - 局部变量空间的动态分配与定位
- 栈溢出的机理与利用方法-精心设计

  JMP ESP　SEH
- 栈溢出SEH利用方式
- ShellCode

漏洞利用的难度很高

但是只要计算机还是冯诺依曼体系架构，

漏洞被利用只是时间的问题…

下回继续…