PWN

二进制漏洞挖掘与利用

Part0 PWN?

- 概述
- 一次简单的hack

- 破解、利用成功（程序的二进制漏洞）
- 攻破（设备、服务器）
- 控制（设备、服务器）

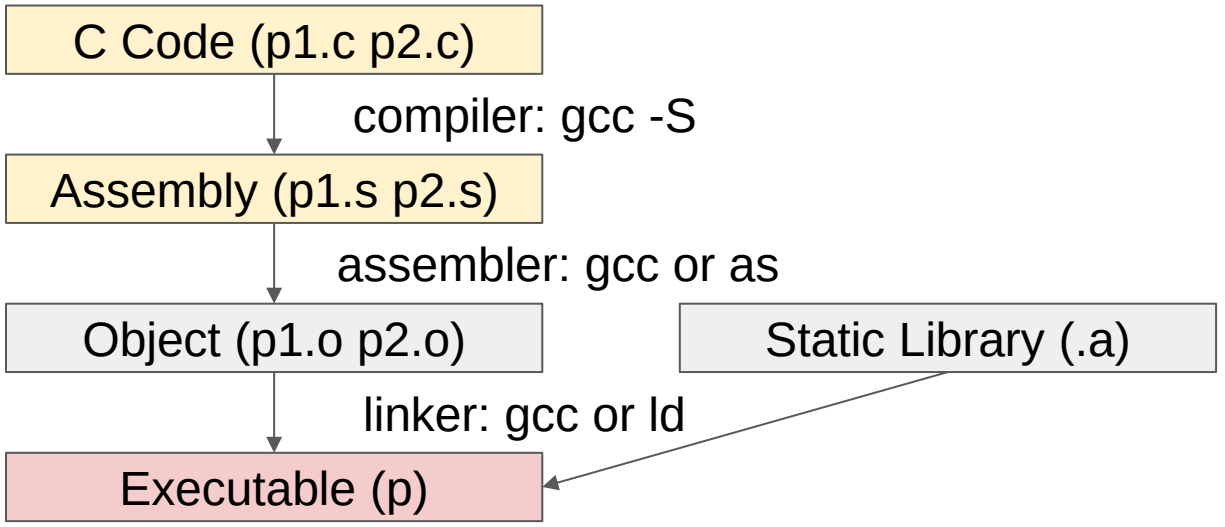
- exploit
  - 用于攻击的脚本与方案
- payload
  - 攻击载荷，是的目标进程被劫持控制流的数据
- shellcode
  - 调用攻击目标的shell的代码

Part1 二进制基础

- 程序的编译与链接
- Linux下的可执行文件格式ELF
- 进程虚拟地址空间
- 程序的装载与进程的执行
- x86&amd64汇编简述

从C源代码到可执行文件的生成过程

- 编译
  - 由C语言代码生成汇编代码
- 汇编
  - 由汇编代码生成机器码
- 链接
  - 将多个机器码的目标文件链接成一个可执行文件



C语言代码

```
int sum(int x, int y)
{
      int t = x + y;
      return t;
}
```

编译

汇编代码

```
sum:
      push ebp
      mov ebp, esp
      mov eax, [ebp+12]
      add eax, [ebp+8]
      pop ebp
      ret
```

汇编

机器码

```
sum:
      0x55
      0x89 0xe5
      0x8b 0x45 0x0c
      0x03 0x45 0x08
      0x5d
      0xc3
```

什么是可执行文件?

- 广义：文件中的数据是可执行代码的文件
  - .out、.exe、.sh、.py
- 狭义：文件中的数据是机器码的文件
  - .out、.exe、.dll、.so

可执行文件的分类

- Windows：PE（Portable Executable）
  - 可执行程序
    - .exe
  - 动态链接库
    - .dll
  - 静态链接库
    - .lib
- Linux：ELF（Executable and Linkable Format）
  - 可执行程序
    - .out
  - 动态链接库
    - .so
  - 静态链接库
    - .a

Dissected file

```
~$uname -p
i686
~$./simple.elf
Hello World!
```

simple.elf

SHA-1: 67259687b87b749b13fd59b0344a5890ebcadc3b
download @ elf101.corkami.com

header 1/2
Technical details for identification and execution

sections
Contents of the executable

header 2/2
Technical details for linking (ignored for execution)

ELF header
identify as an ELF type
specify the architecture

Program Header table
Execution information

Code
Executable information

Data
Information used by the code

Sections' names

Section Header table
Linking (connecting program objects) information

Hexadecimal dump / ASCII dump

```
7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00  .ELF............
02 00 03 00 01 00 00 00 60 00 00 08 40 00 00 00  ........`...@...
C0 00 00 00 00 00 00 00 34 00 20 00 01 00 28 00  +.......4.....(.
04 00 03 00                                      ....
```

1

Fields	Values	Explanation
e_ident		
EI_MAG	0x7F, "ELF"	constant signature
EI_CLASS, EI_DATA	1 ELFCLASS32, 1 ELFDATA2LSB	32 bits, Little-Endian
EI_VERSION	1 EV_CURRENT	Always 1
e_type	2 ET_EXEC	Executable
e_machine	3 EM_386	Intel 386 (and later)
e_version	1 EV_CURRENT	Always 1
e_entry	0x8000060	Address where execution starts
e_phoff	0x40	Program Headers' offset
e_shoff	0xC0	Section Headers' offset
e_ehsize	0x34	Elf header's size
e_phentsize	0x20	Size of a single Program Header
e_phnum	1	Count of Program Headers
e_shentsize	0x28	Size of a single Section Header
e_shnum	4	Count of Section Headers
e_shstrndx	3*	Index of the names' section in the table

```
Offset:0x40/Address:0x8000040
01 00 00 00 00 00 00 00 00 00 00 00 08 00 00 00 08  ................
A0 00 00 00 A0 00 00 00 05 00 00 00 00 00 00 00  ................
```

2

p_type	1 PT_LOAD	The segment should be loaded in memory
p_offset	0	Offset where it should be read
p_vaddr	0x8000000	Virtual address where it should be loaded
p_paddr	0x8000000	Physical address where it should be loaded
p_filesz	0xA0	Size on file
p_memsz	0xA0	Size in memory
p_flags	5 PF_R|PF_X	Readable and eXecutable

```
Offset:0x60/Address:0x8000060
B9 90 00 00 08 BA 0D 00 00 00 BB 01 00 00 00 B8  ................
04 00 00 00 CD 80 BB 01 00 00 00 B8 01 00 00 00  ................
CD 80                                            ..
```

3

x86 assembly / Equivalent C code

```
mov ecx, 0x8000090   ; msg
mov edx, 0xD         ; msg_len
mov ebx, 1           ; STDOUT_FILENO
mov eax, 4           ; SYS_WRITE
int 0x80             ; system call   → write(STDOUT_FILENO, "Hello World!\n", len("Hello World!\n"));

mov ebx, 1           ; return code
mov eax, 1           ; SYS_EXIT
int 0x80             ; system call   → exit(1);
```

Strings

```
Offset:0x90/Address:0x8000090
48 65 6C 6C 6F 20 57 6F 72 6C 64 21 0A 00        Hello.World!..
```

"Hello World!\n", 0

Section names

```
Offset:0xA0
00 2E 73 68 73 74 72 74 61 62 00 2E 74 65 78 74  ..shstrtab..text
00 2E 72 6F 64 61 74 61 00                       ..rodata.
```

"" .shrtrtab .text .rodata

```
Offset:0xC0
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 0B 00 00 00 01 00 00 00  ................
06 00 00 00 60 00 00 08 60 00 00 00 22 00 00 00  ....`...`..."...
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
11 00 00 00 01 00 00 00 02 00 00 00 90 00 00 08  ................
90 00 00 00 0D 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 01 00 00 00 03 00 00 00  ................
00 00 00 00 00 00 00 00 A0 00 00 00 19 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

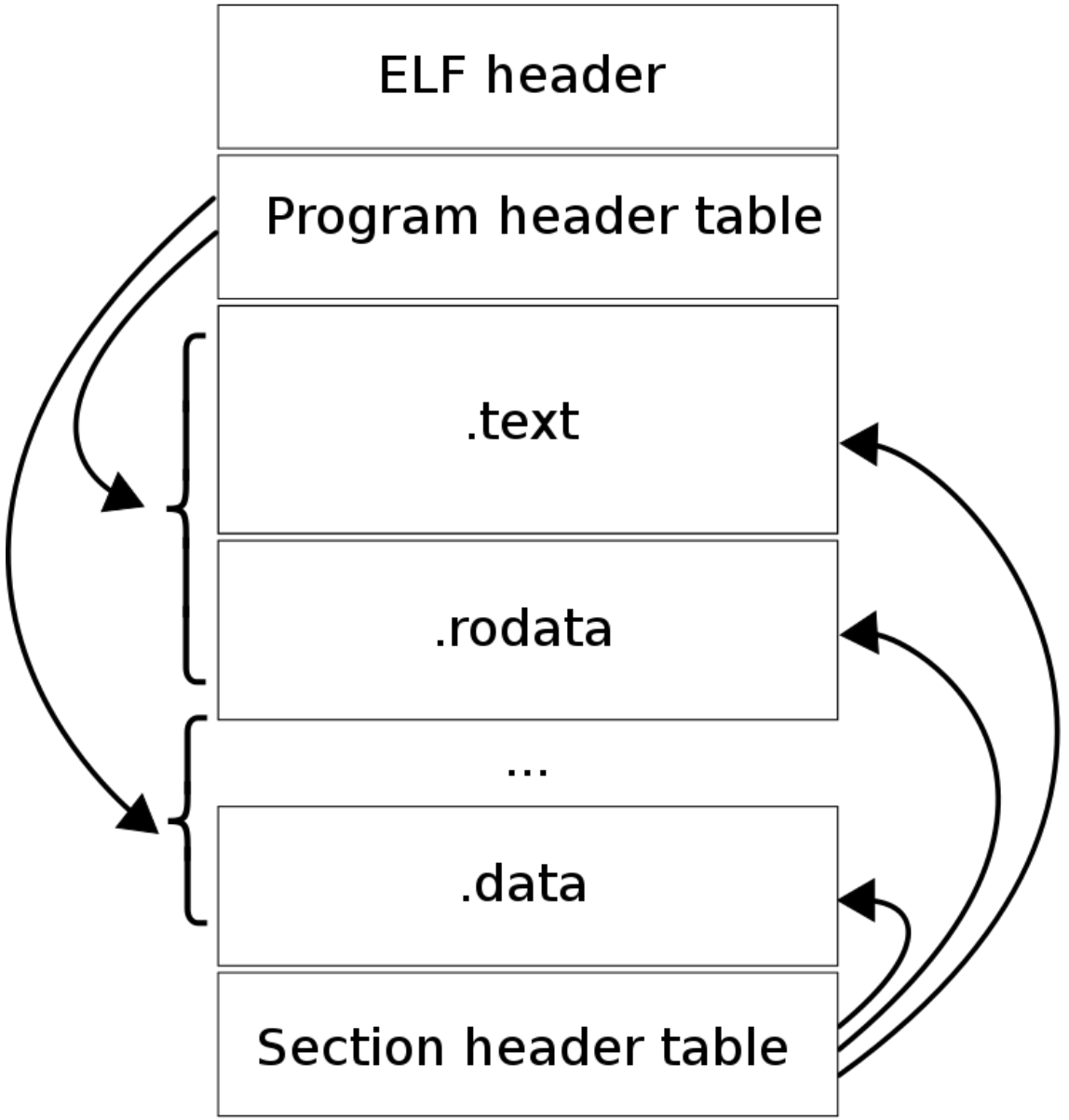
Section Header table

Index	Name (sh_name)	Type (sh_type)	Flags (sh_flags)	Address (sh_addr)	Offset (sh_offset)	Size (sh_size)
0	<null>	0 SHT_NULL (inactive)				
1	.text	1 SHT_PROGBITS (program)	6 SHF_ALLOC (allocated) SHF_EXECINSTR (executable)	0x8000060	0x60	0x22
2	.rodata	1 SHT_PROGBITS (program)	2 SHF_ALLOC (allocated)	0x8000090	0x90	0x0D
3*	.shrtrtab	3 SHT_STRTAB (string table)			0xA0	0x19

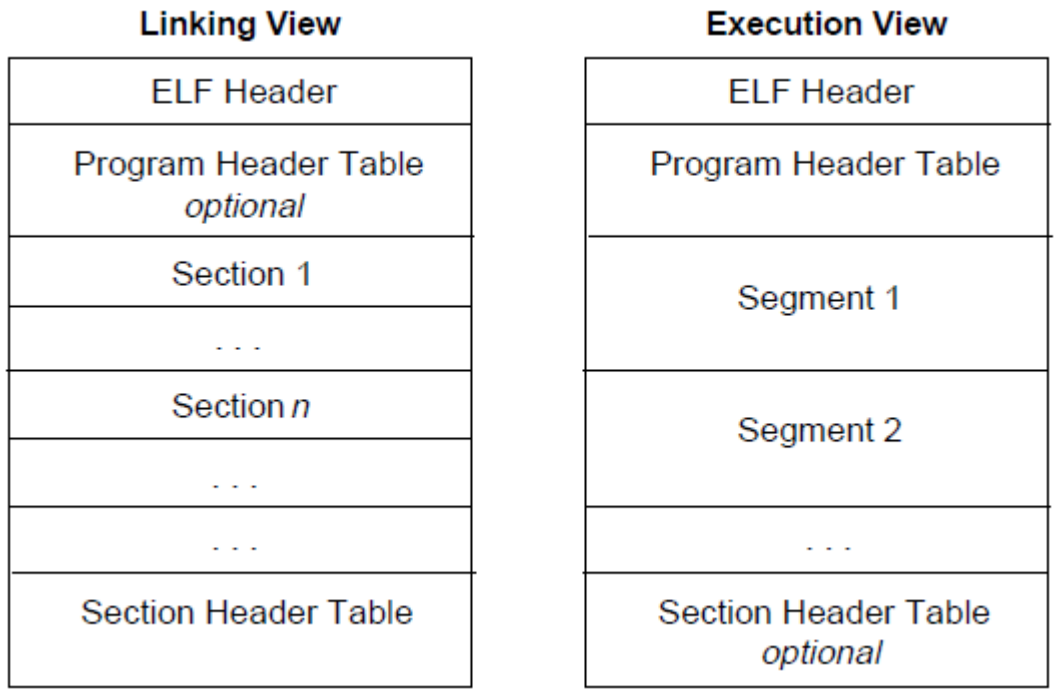
relative offsets in names' section

ELF 文件结构

- ELF文件头表（ELF header）
  - 记录了ELF文件的组织结构
- 程序头表/段表（Program header table）
  - 告诉系统如何创建进程
  - 生成进程的可执行文件必须拥有此结构
  - 重定位文件不一定需要
- 节头表（Section header table）
  - 记录了ELF文件的节区信息
  - 用于链接的目标文件必须拥有此结构
  - 其它类型目标文件不一定需要

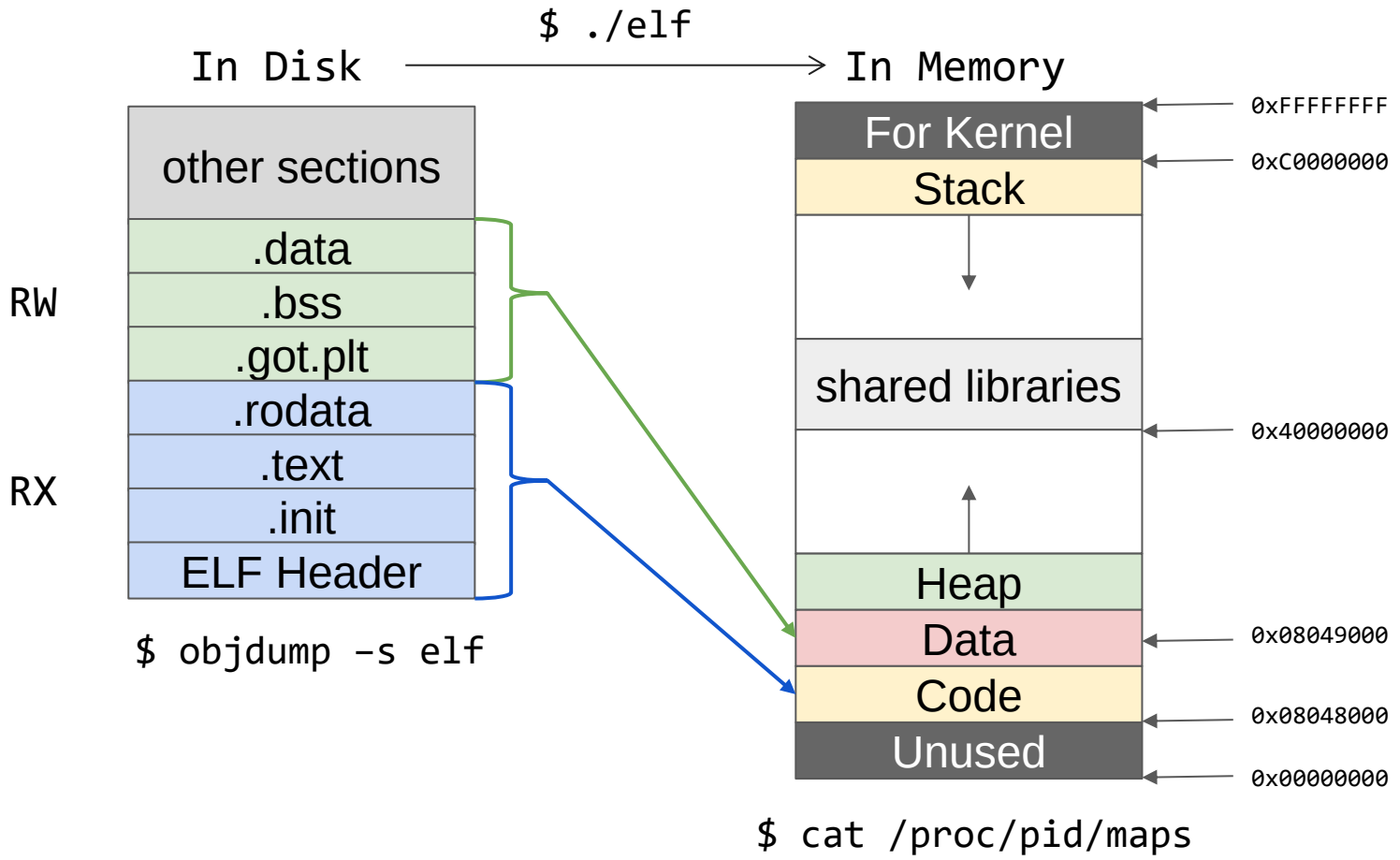


磁盘中的ELF（可执行文件）与内存中的ELF（进程内存映像）

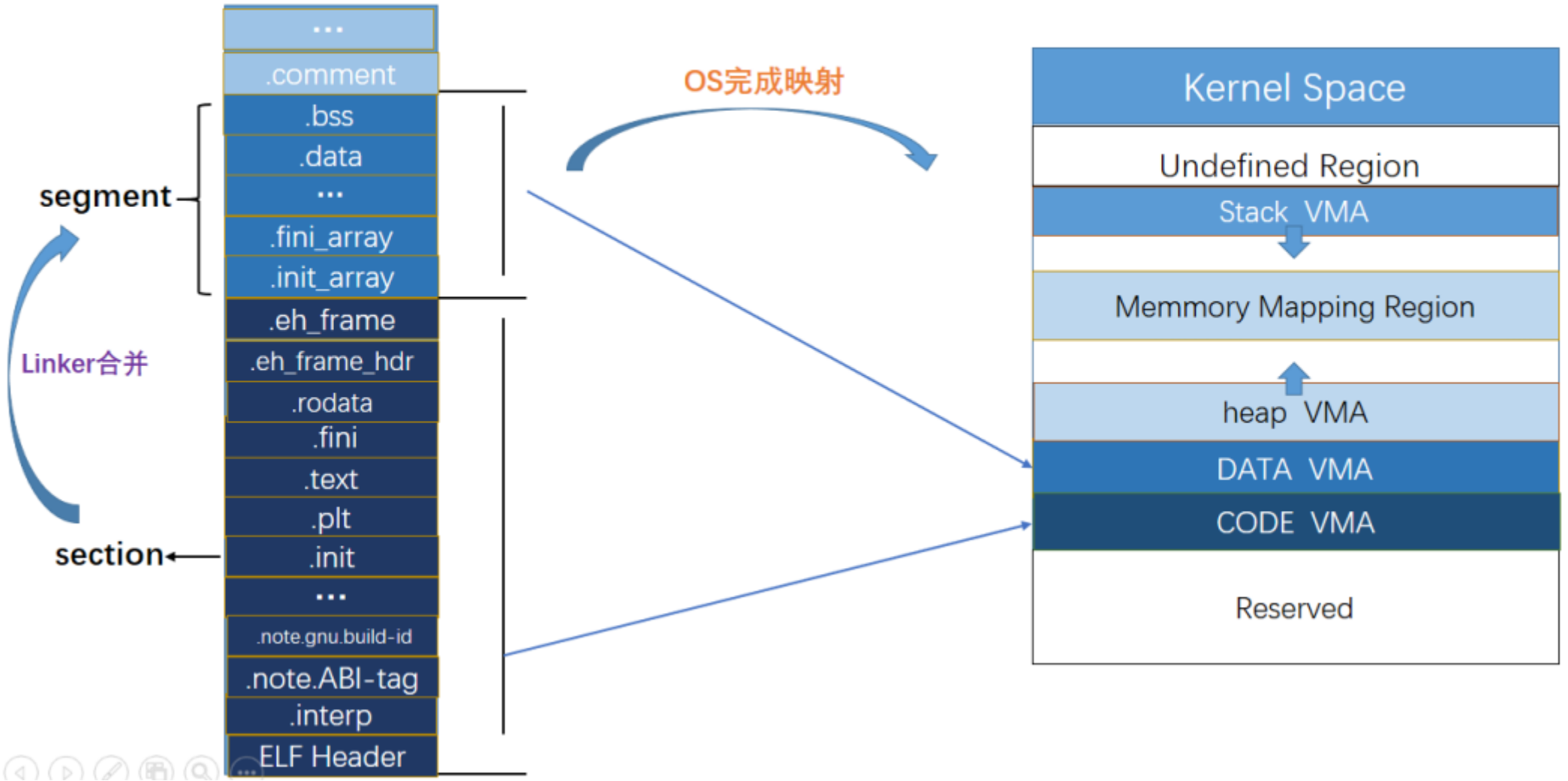


节视图　　段视图

磁盘中的ELF（可执行文件）与内存中的ELF（进程内存映像）



ELF文件到虚拟地址空间的映射



地址以字节编码

1Byte = 8bits

常以16进制表示

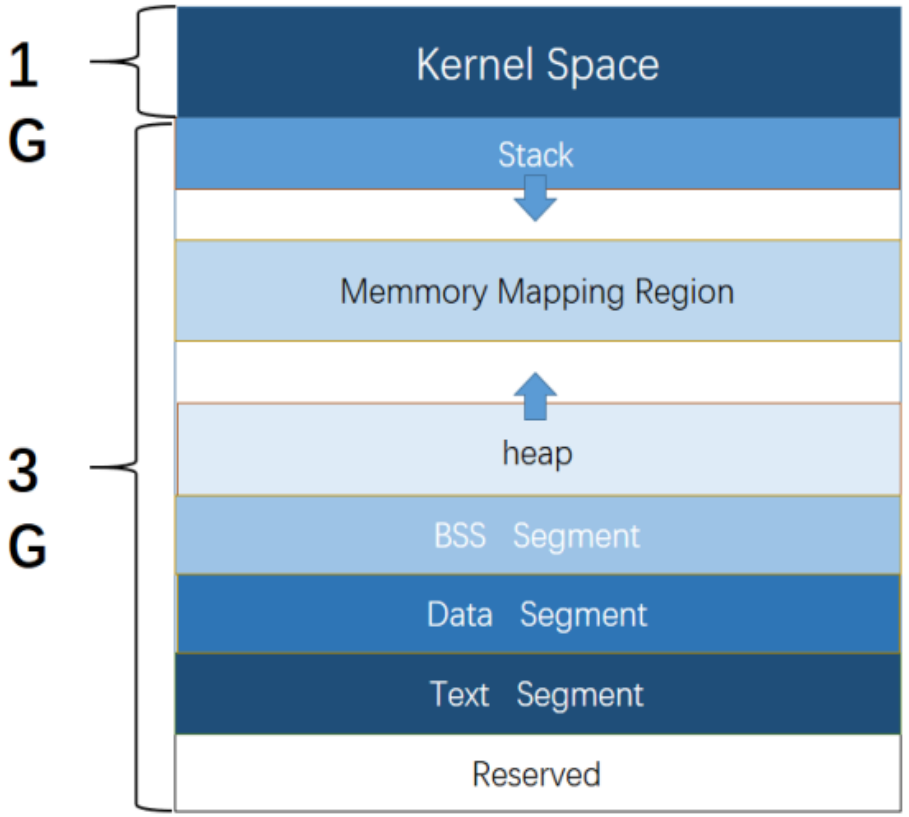
0x3c = 0011 1100

虚拟内存用户空间每个进程一份

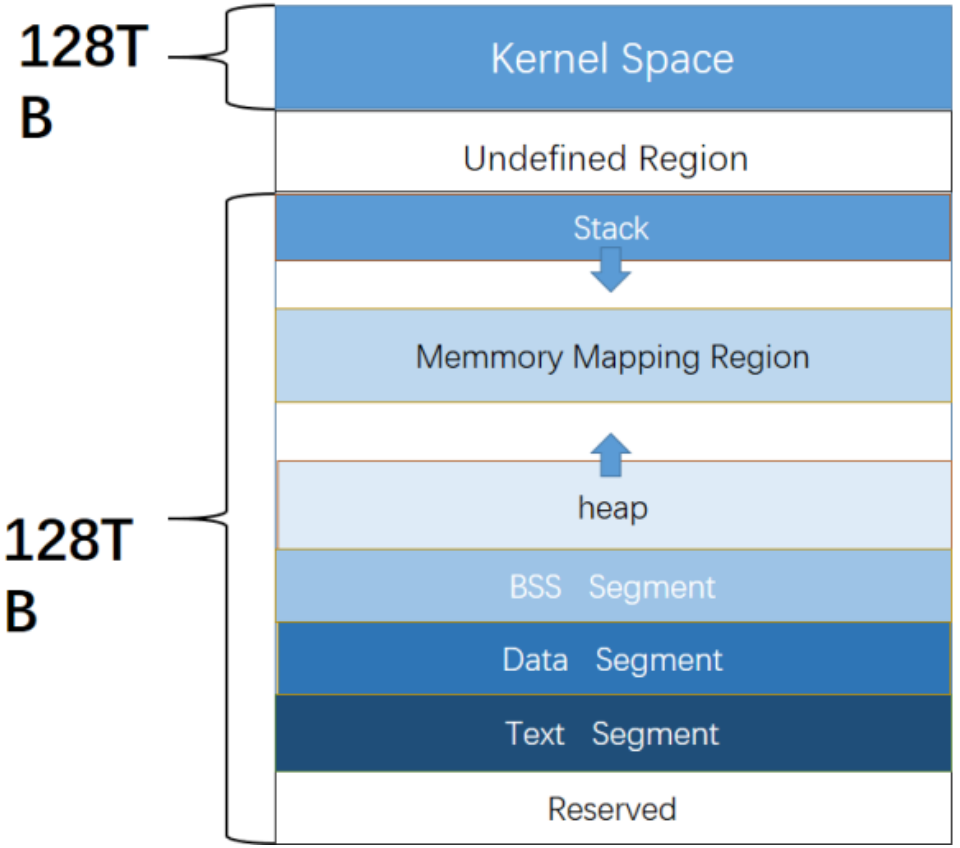
虚拟内存内核空间所有进程共享一份

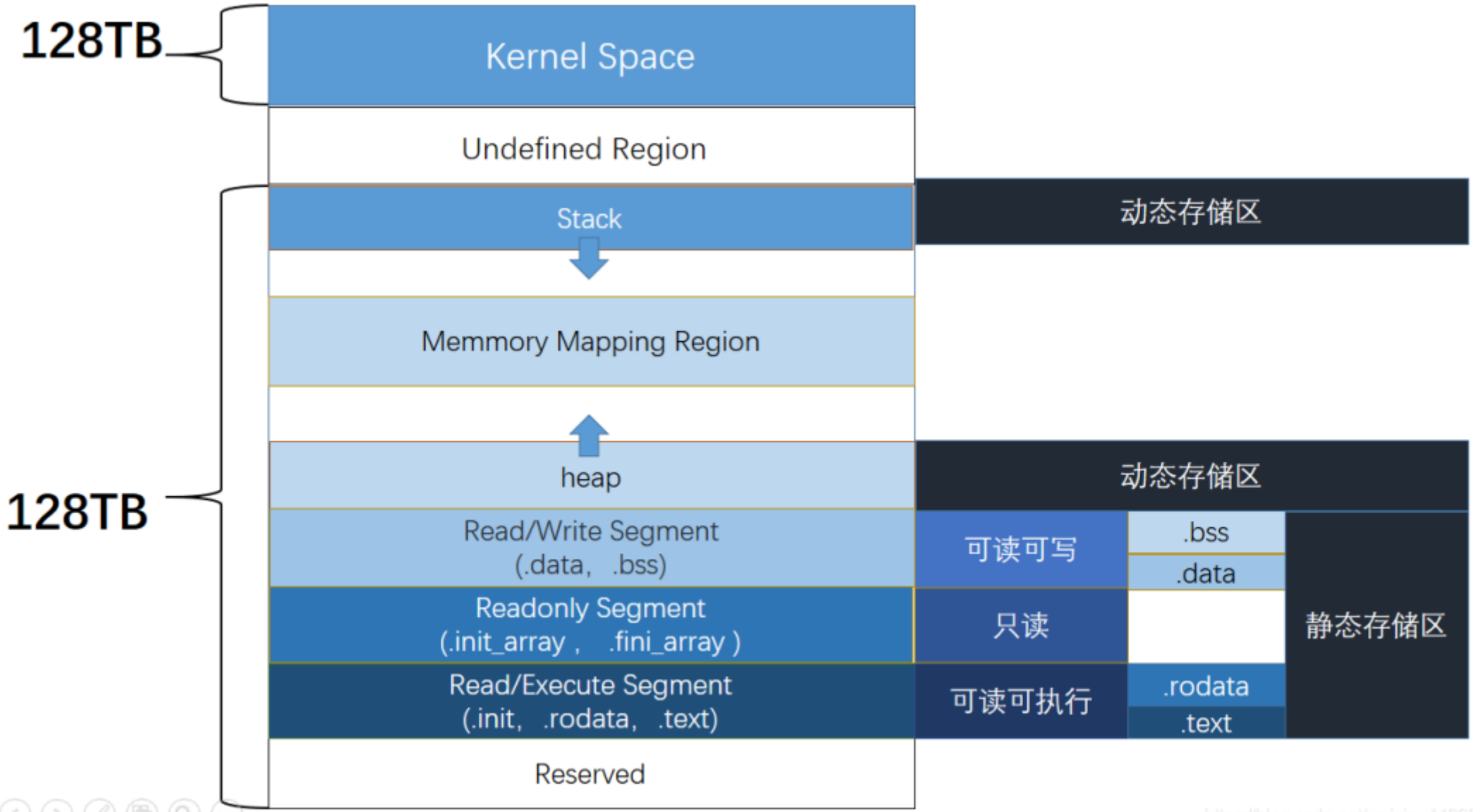
虚拟内存 mmap 段中的动态链接库仅在物理内存中装载一份

32位的进程虚拟地址空间



64位进程虚拟地址空间




128TB
Kernel Space
Undefined Region
128TB
Stack
Memmory Mapping Region
heap
Read/Write Segment
(.data, .bss)
Readonly Segment
(.init_array , .fini_array)
Read/Execute Segment
(.init, .rodata, .text)
Reserved
动态存储区
动态存储区
可读可写
.bss
.data
只读
可读可执行
.rodata
.text
静态存储区

段（segment）与节（section）

- 代码段（Text segment）包含了代码与只读数据
  - .text 节
  - .rodata 节
  - .hash 节
  - .dynsym 节
  - .dynstr 节
  - .plt 节
  - .rel.got 节
  - ……
- 数据段（Data segment）包含了可读可写数据
  - .data 节
  - .dynamic 节
  - .got 节
  - .got.plt 节
  - .bss 节
  - ……
- 栈段（Stack segment）

- 一个 段 包含多个 节
- 段视图用于进程的内存区域的 rwx权限划分
- 节视图用于ELF文件 编译链接时 与 在磁盘上存储时 的文件结构的组织

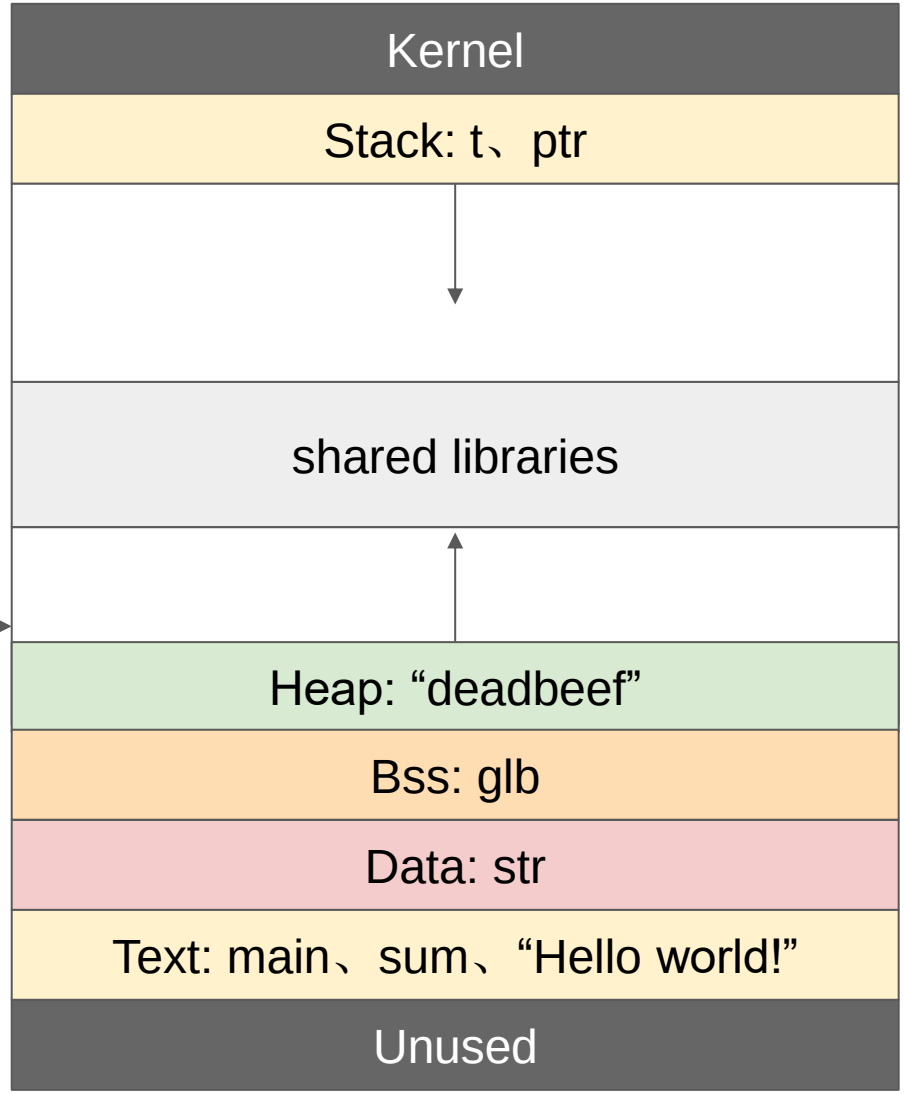
程序数据是如何在内存中组织的

C语言源代码

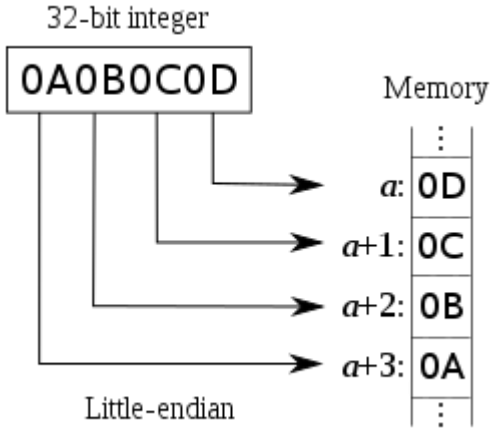
```
int glb;
char* str = “Hello world!”;

int sum(int x, int y)
{
      int t = x + y;
      return t;
}

int main()
{
    sum(1, 2);
    void* ptr = malloc(0x100);
    read(0, ptr, 0x100);  // input “deadbeef”
    return 0;
}
```

编译链接执行

载入内存

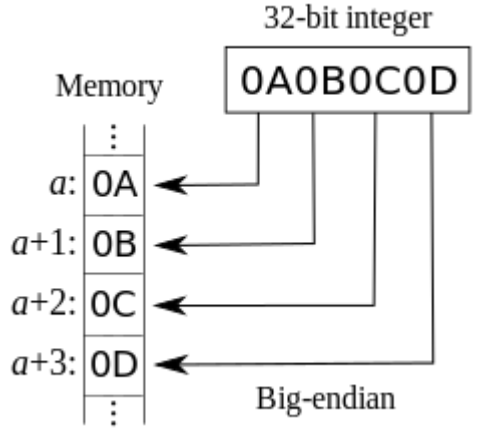


大端序与小端序



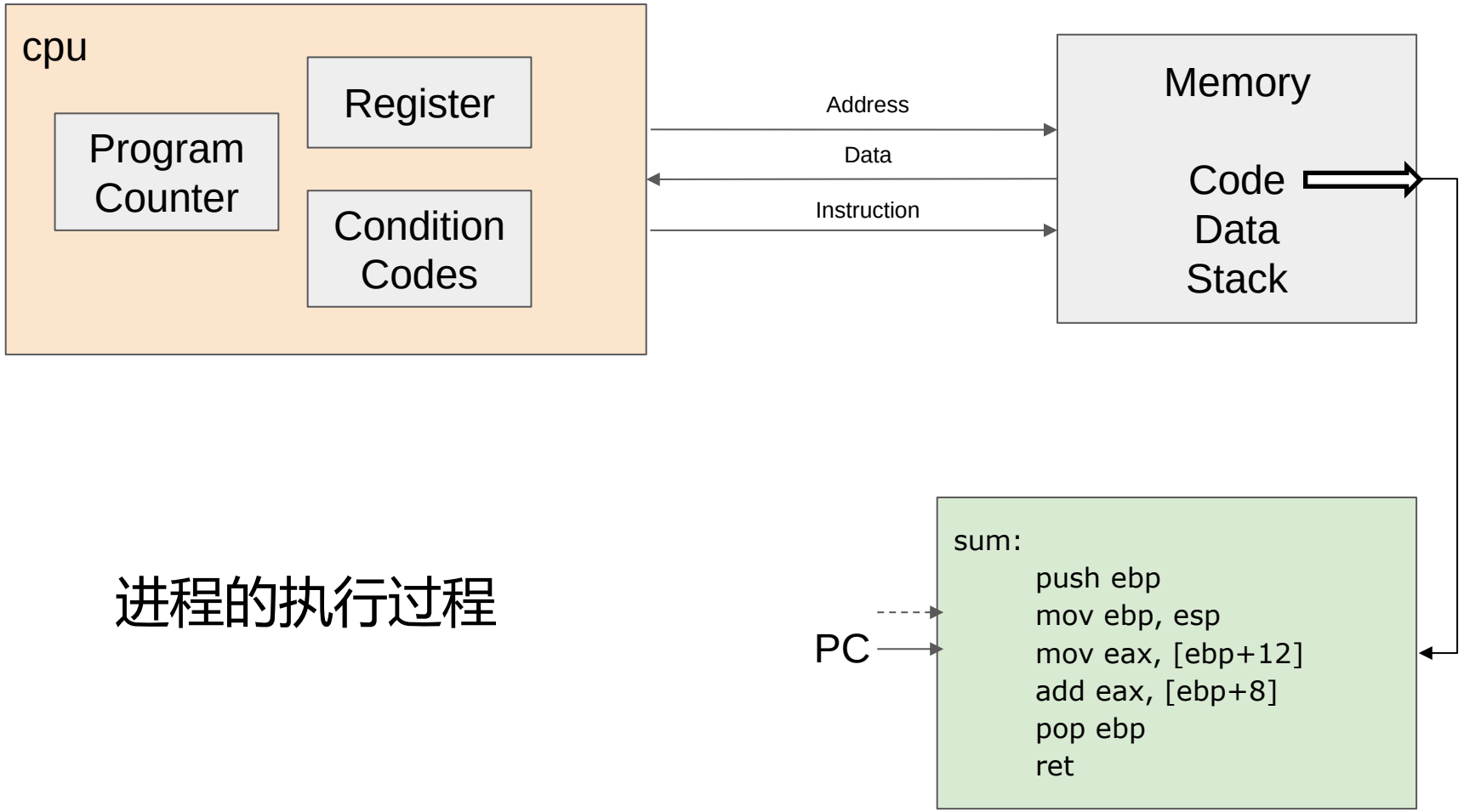
小端序

- 低地址存放数据低位、高地址存放数据高位
- 我们所主要关注的格式



大端序

- 低地址存放数据高位、高地址存放数据低位


cpu
Register
Program Counter
Condition Codes
Address
Data
Instruction
Memory
Code
Data
Stack
sum:
push ebp
mov ebp, esp
mov eax, [ebp+12]
add eax, [ebp+8]
pop ebp
ret
PC

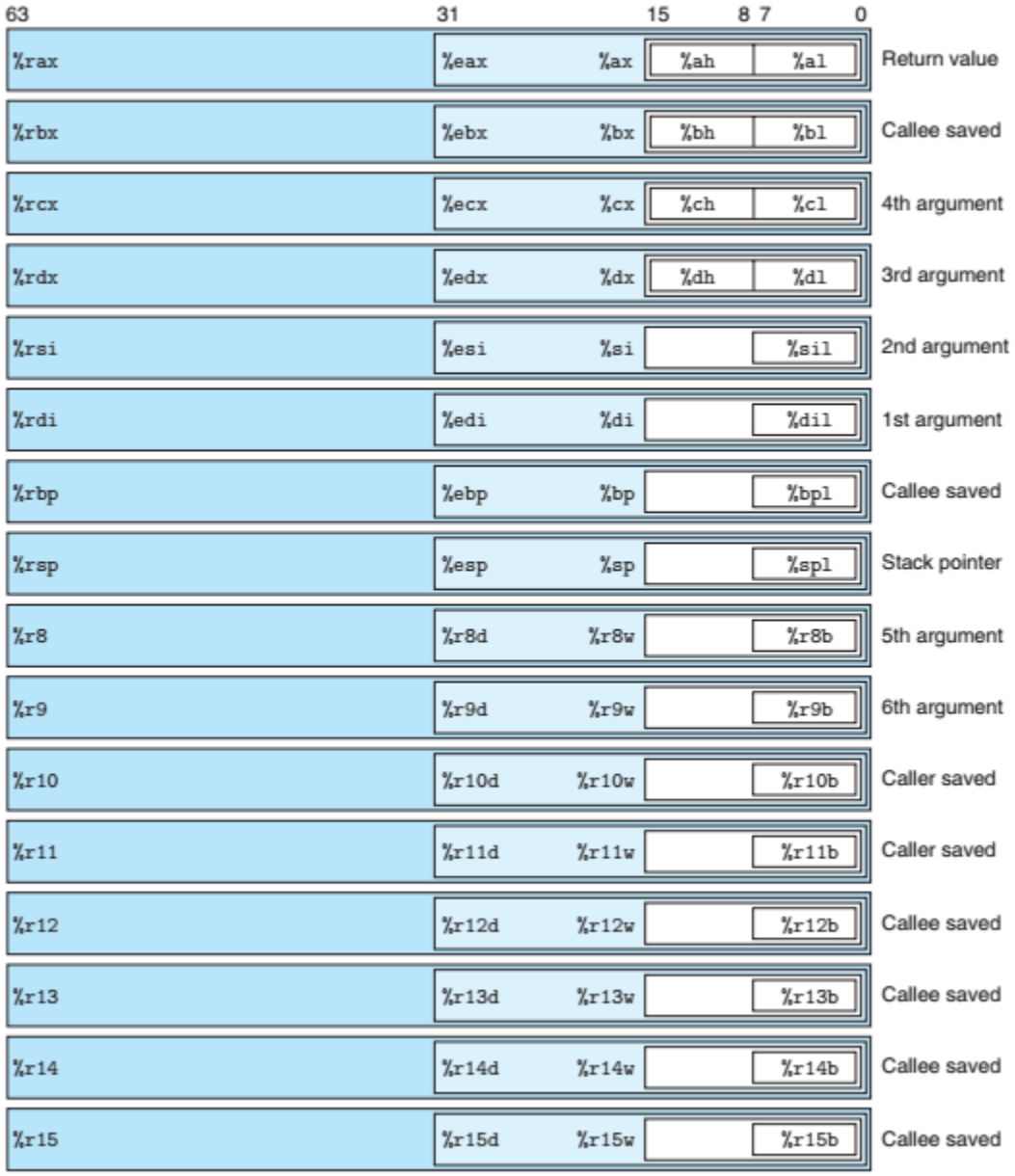
进程的执行过程

amd64寄存器结构

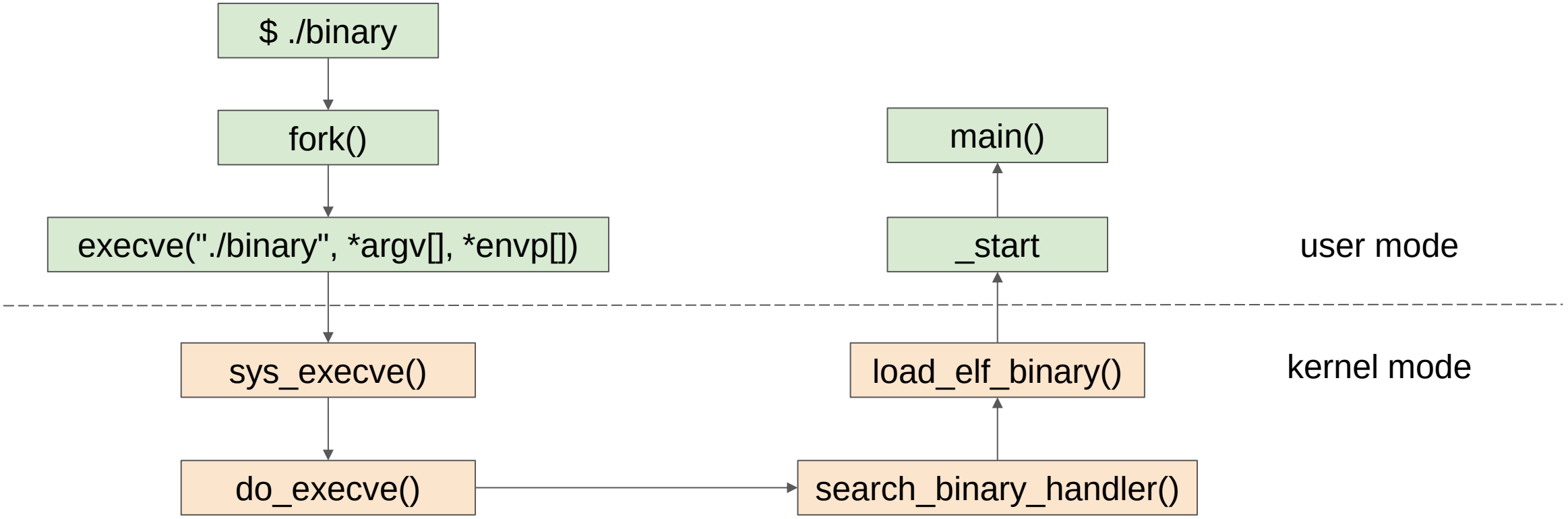
- rax：8Bytes
- eax：4Bytes
- ax：2Bytes
- ah：1Bytes
- al：1Bytes

部分寄存器的功能

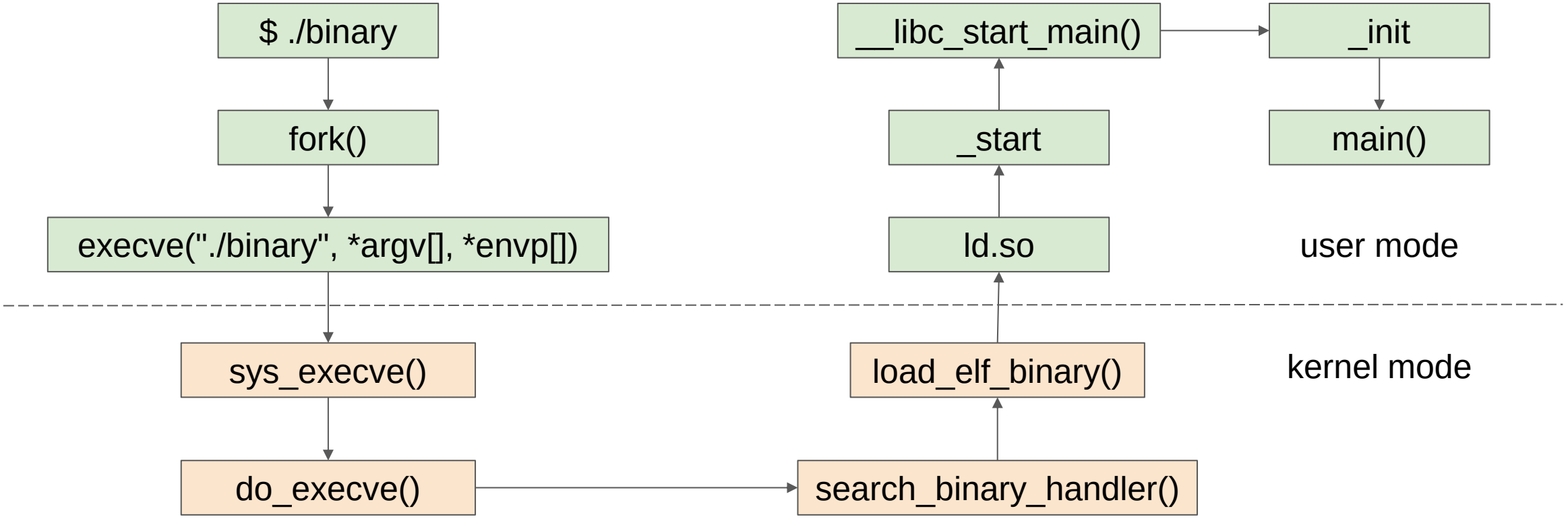
- RIP
  - 存放当前执行的指令的地址
- RSP
  - 存放当前栈帧的栈顶地址
- RBP
  - 存放当前栈帧的栈底地址
- RAX
  - 通用寄存器。存放函数返回值



静态链接的程序的执行过程



动态链接的程序的执行过程



常用汇编指令

- MOV
- LEA
- ADD/SUB
- PUSH
- POP
- CMP
- JMP
- J[Condition]
- CALL
- LEAVE
- RET
- ……

MOV

MOV DEST, SRC ；把源操作数传送给目标

- MOV EAX,1234H ；执行结果（EAX） = 1234H
- MOV EBX, EAX
- MOV EAX, [00404011H] ；[ ] 表示取地址内的值
- MOV EAX, [ESI]

LEA

LEA REG, SRC ；把源操作数的有效地址送给指定的寄存器

- LEA EBX, ASC ；取 ASC 的地址存放至 EBX 寄存器中
- LEA EAX, 6[ESI] ；把 ESI+6 单元的32位地址送给 EAX

PUSH

PUSH VALUE ；把目标值压栈，同时SP指针-1字长

- PUSH 1234H
- PUSH EAX

POP

POP DEST ；将栈顶的值弹出至目的存储位置，同时SP指针+1字长

- POP EAX

LEAVE

- 在函数返回时，恢复父函数栈帧的指令
- 等效于：
  - MOV ESP, EBP
  - POP EBP

RET

- 在函数返回时，控制程序执行流返回父函数的指令
- 等效于：
  - POP RIP（这条指令实际是不存在的，不能直接向RIP寄存器传送数据）

两种汇编格式

intel	AT&T
`mov eax, 8`	`movl $8, %eax`
`mov ebx, 0ffffh`	`movl $0xffff, %ebx`
`int 80h`	`int $80`
`mov eax, [ecx]`	`movl (%ecx), %eax`

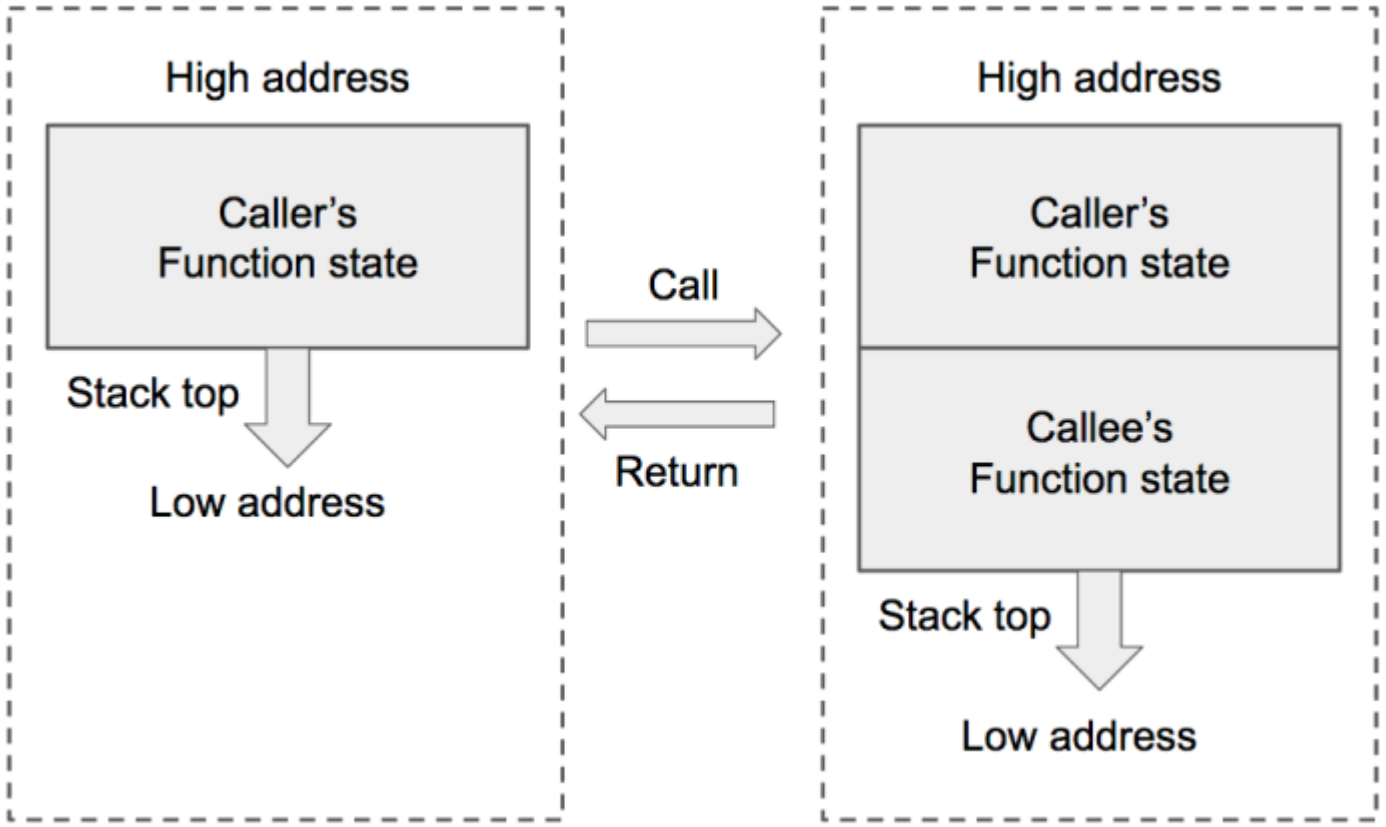
```
sum:
      push ebp
      mov ebp, esp
      mov eax, [ebp+12]
      add eax, [ebp+8]
      pop ebp
      retn
```

```
sum:
      pushl %ebp
      movl %esp,%ebp
      movl 12(%ebp),%eax
      addl 8(%ebp),%eax
      popl %ebp
      ret
```

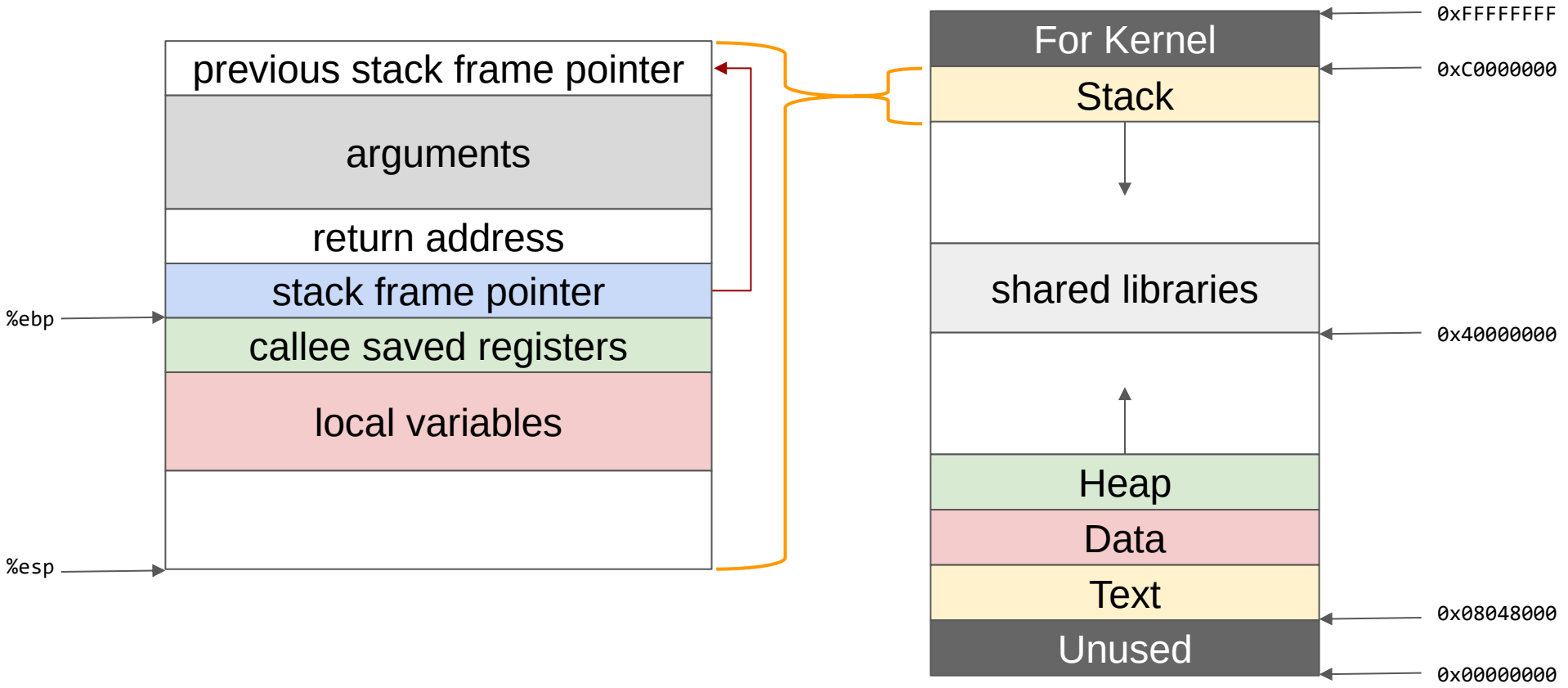
Part2 栈溢出基础

- C语言函数调用栈
- ret2text
- ret2shellcode

- 函数调用栈是指程序运行时内存一段连续的区域
- 用来保存函数运行时的状态信息，包括函数参数与局部变量等
- 称之为“栈”是因为发生函数调用时，调用函数（caller）的状态被保存在栈内，被调用函数（callee）的状态被压入调用栈的栈顶
- 在函数调用结束时，栈顶的函数（callee）状态被弹出，栈顶恢复到调用函数（caller）的状态
- 函数调用栈在内存中从高地址向低地址生长，所以栈顶对应的内存地址在压栈时变小，退栈时变大



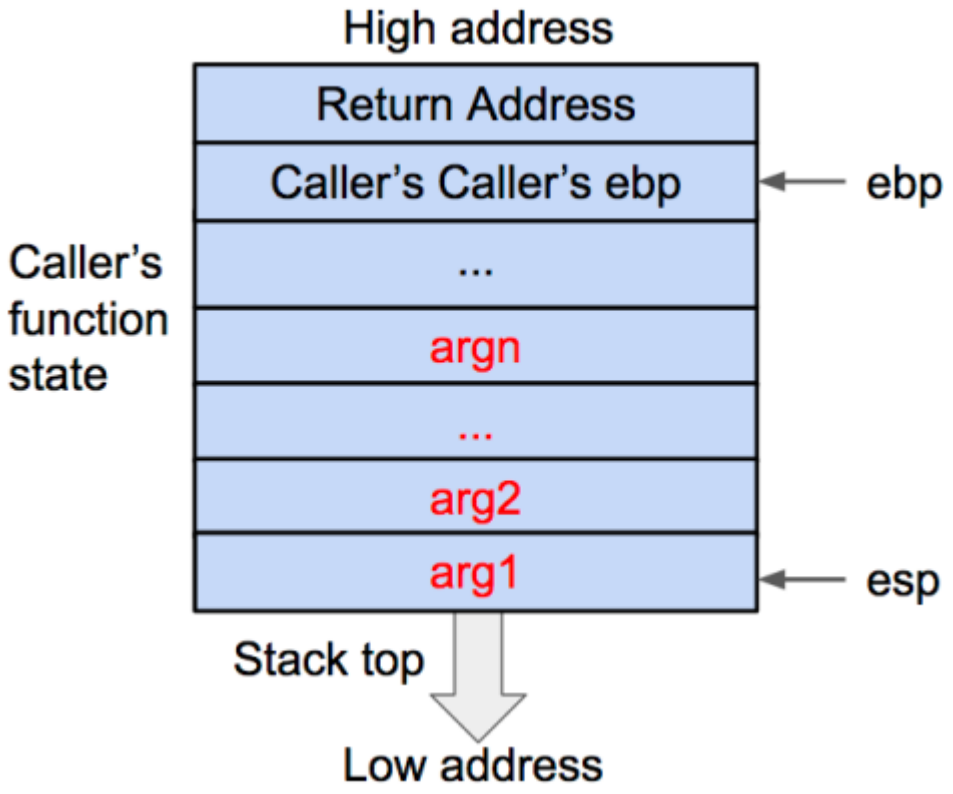
栈帧结构概览


previous stack frame pointer
arguments
return address
stack frame pointer
callee saved registers
local variables
%ebp
%esp
For Kernel
Stack
shared libraries
Heap
Data
Text
Unused
0xFFFFFFFF
0xC0000000
0x40000000
0x08048000
0x00000000

- 函数状态主要涉及三个寄存器 —— esp，ebp，eip。esp 用来存储函数调用栈的栈顶地址，在压栈和退栈时发生变化。ebp 用来存储当前函数状态的基地址，在函数运行时不变，可以用来索引确定函数参数或局部变量的位置。eip 用来存储即将执行的程序指令的地址，cpu 依照 eip 的存储内容读取指令并执行，eip 随之指向相邻的下一条指令，如此反复，程序就得以连续执行指令。

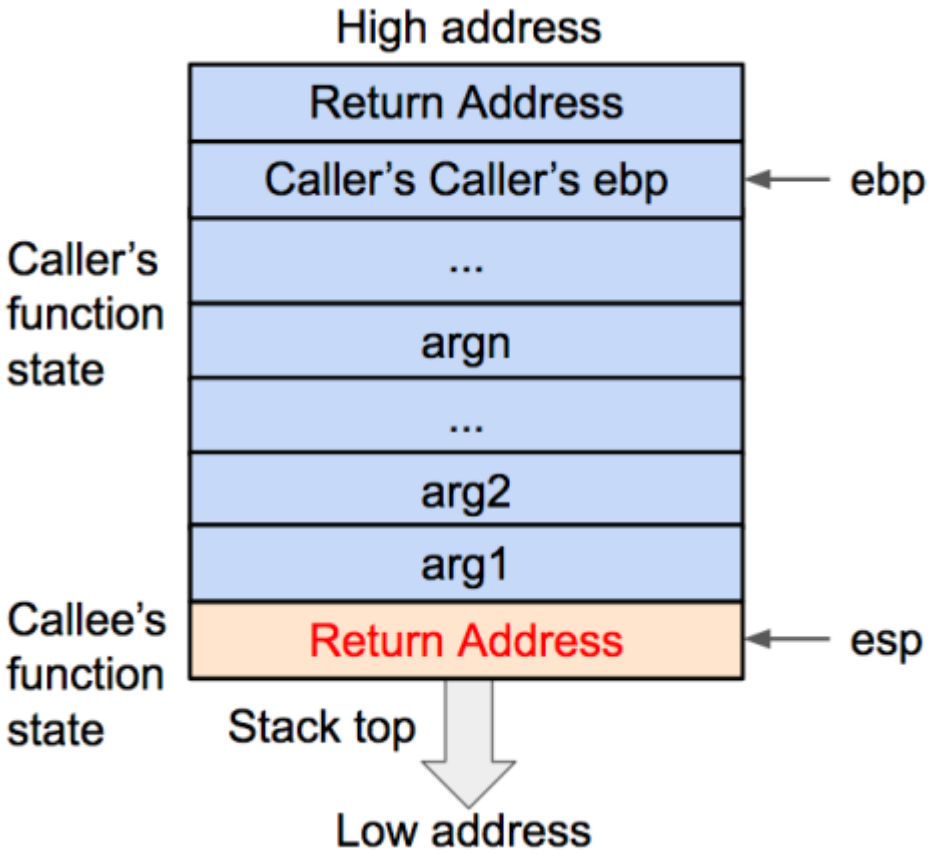
- 下面让我们来看看发生函数调用时，栈顶函数状态以及上述寄存器的变化。变化的核心任务是将调用函数（caller）的状态保存起来，同时创建被调用函数（callee）的状态。

- 首先将被调用函数（callee）的参数按照逆序依次压入栈内。如果被调用函数（callee）不需要参数，则没有这一步骤。这些参数仍会保存在调用函数（caller）的函数状态内，之后压入栈内的数据都会作为被调用函数（callee）的函数状态来保存。



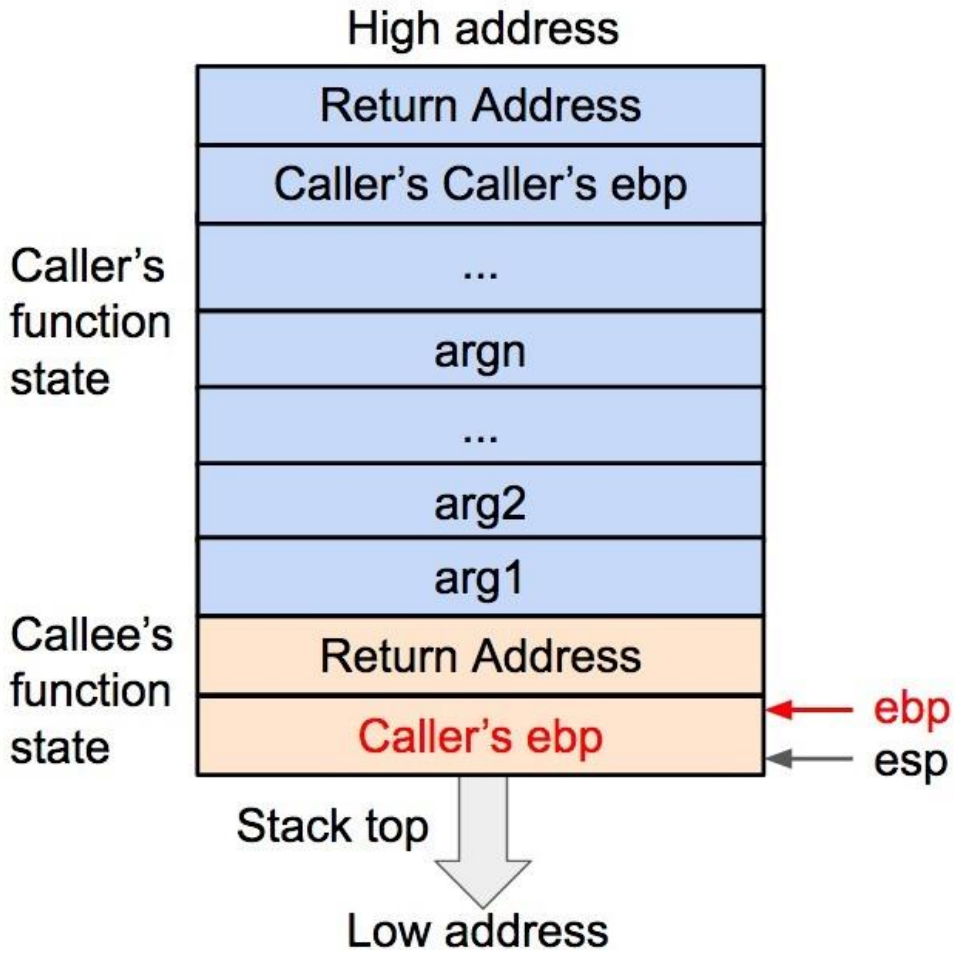
将被调用函数的参数压入栈内

- 然后将调用函数（caller）进行调用之后的下一条指令地址作为返回地址压入栈内。这样调用函数（caller）的eip（指令）信息得以保存。



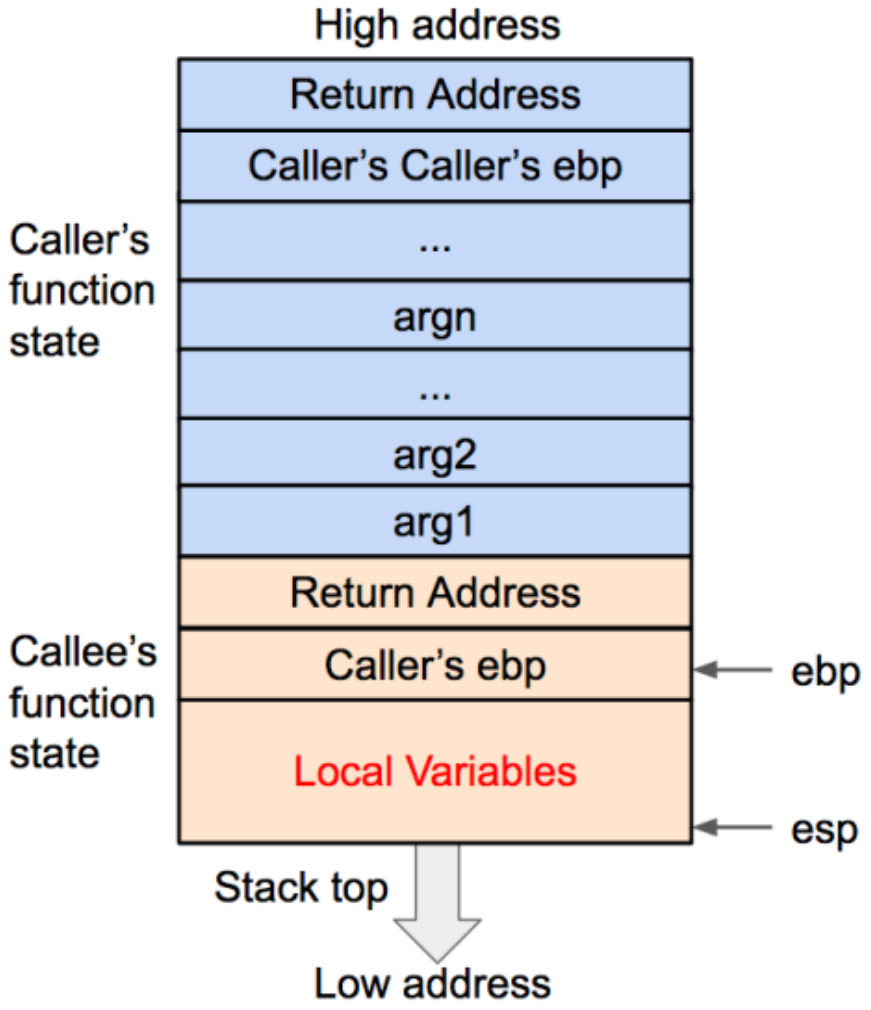
将被调用函数的返回地址压入栈内

- 再将当前的ebp 寄存器的值（也就是调用函数的基地址）压入栈内，并将 ebp 寄存器的值更新为当前栈顶的地址。这样调用函数（caller）的 ebp（基地址）信息得以保存。同时，ebp 被更新为被调用函数（callee）的基地址。



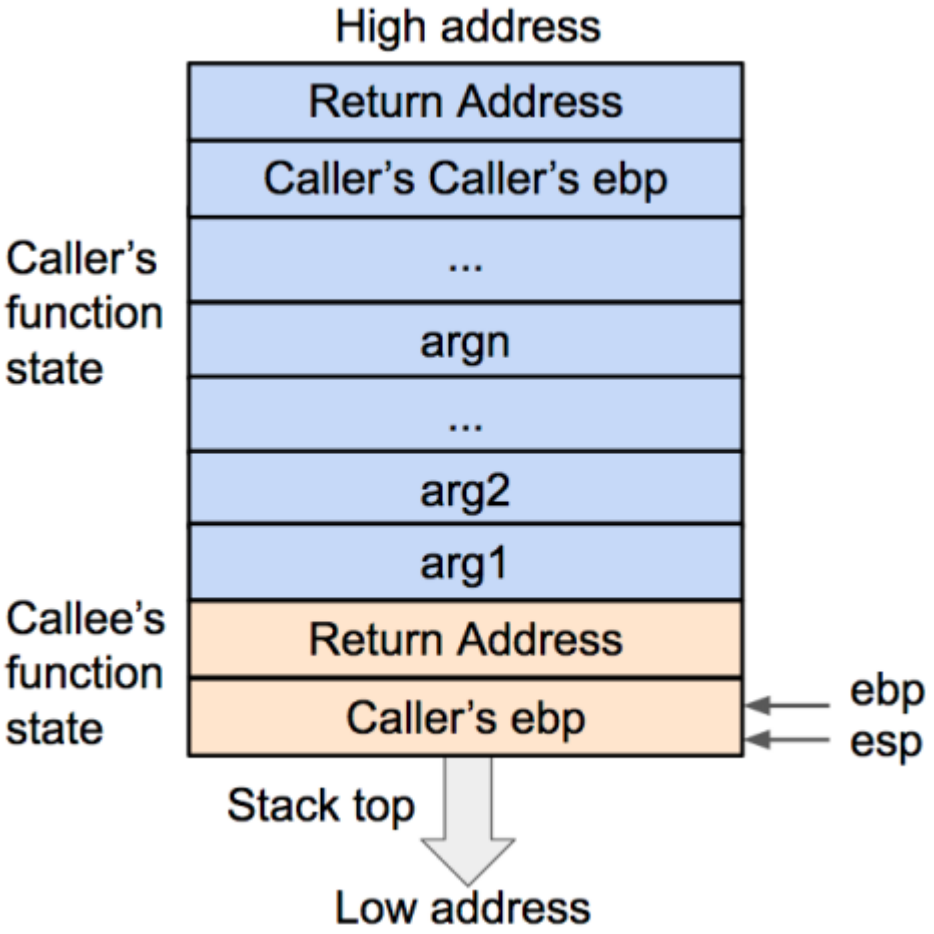
将调用函数的基地址（ebp）压入栈内，并将当前栈顶地址传到 ebp 寄存器内

- 再之后是将被调用函数（callee）的局部变量等数据压入栈内。



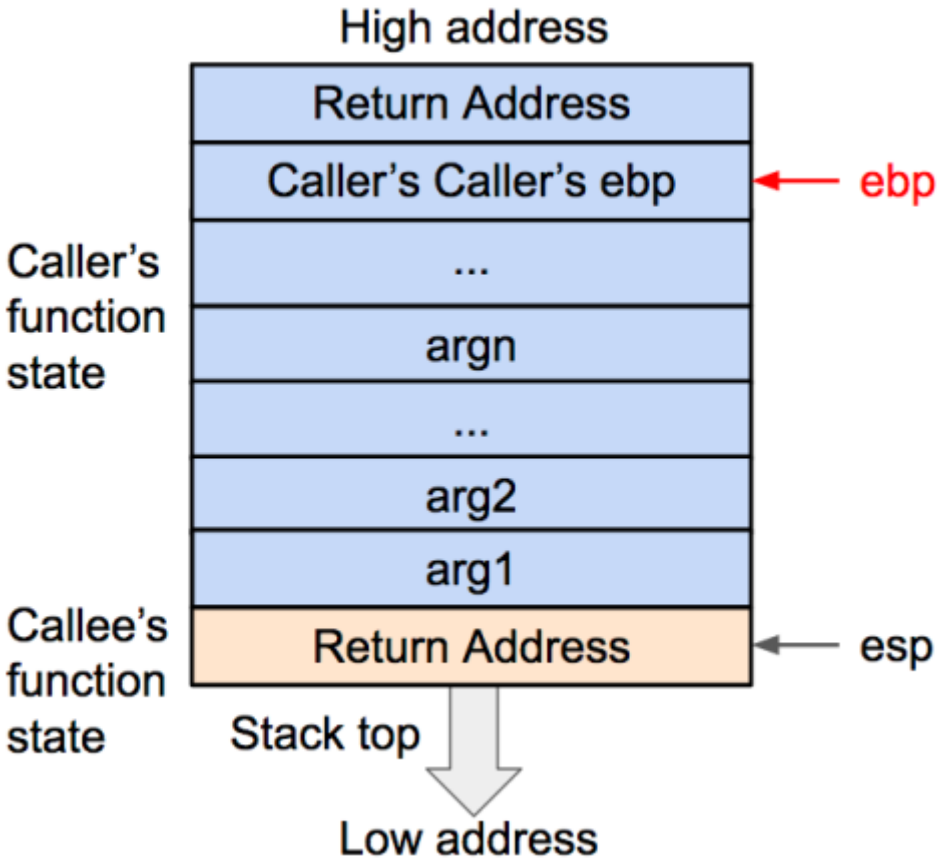
将被调用函数的局部变量压入栈内

- 在压栈的过程中，esp 寄存器的值不断减小（对应于栈从内存高地址向低地址生长）。压入栈内的数据包括调用参数、返回地址、调用函数的基地址，以及局部变量，其中调用参数以外的数据共同构成了被调用函数（callee）的状态。在发生调用时，程序还会将被调用函数（callee）的指令地址存到 eip 寄存器内，这样程序就可以依次执行被调用函数的指令了。
- 看过了函数调用发生时的情况，就不难理解函数调用结束时的变化。变化的核心任务是丢弃被调用函数（callee）的状态，并将栈顶恢复为调用函数（caller）的状态。
- 首先被调用函数的局部变量会从栈内直接弹出，栈顶会指向被调用函数（callee）的基地址。



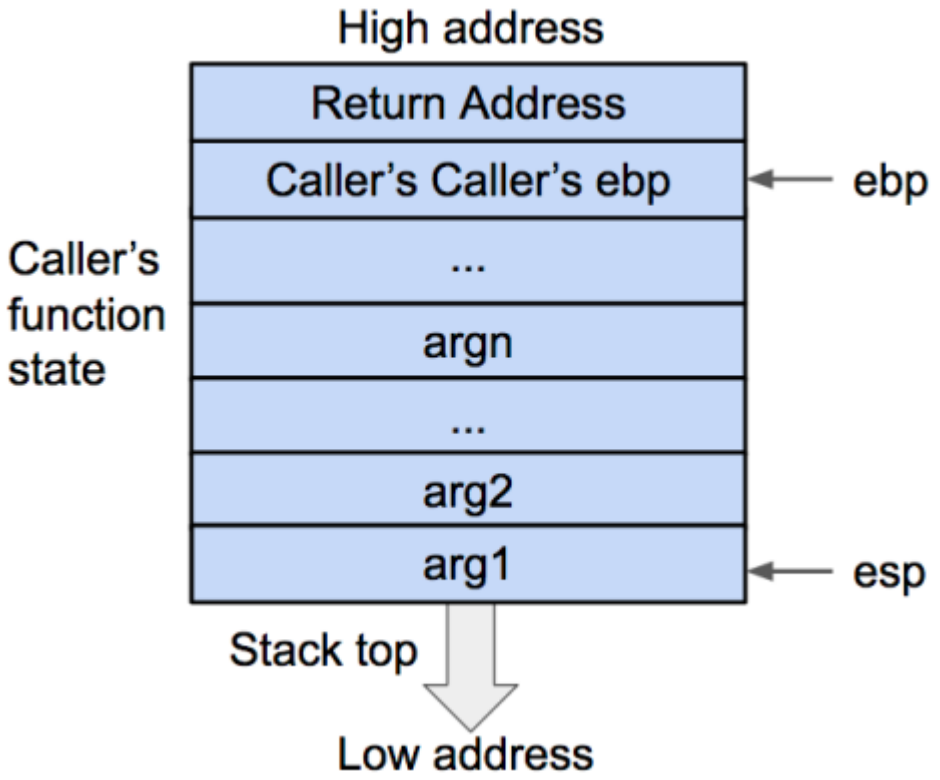
将被调用函数的局部变量弹出栈外

- 然后将基地址内存储的调用函数（caller）的基地址从栈内弹出，并存到 ebp 寄存器内。这样调用函数（caller）的 ebp（基地址）信息得以恢复。此时栈顶会指向返回地址。



将调用函数（caller）的基地址（ebp）弹出栈外，并存到 ebp 寄存器内

- 再将返回地址从栈内弹出，并存到 eip 寄存器内。这样调用函数（caller）的 eip（指令）信息得以恢复。
- 至此调用函数（caller）的函数状态就全部恢复了，之后就是继续执行调用函数的指令了。



将被调用函数的返回地址弹出栈外，并存到 eip 寄存器内

函数调用栈的工作方式（cdecl）

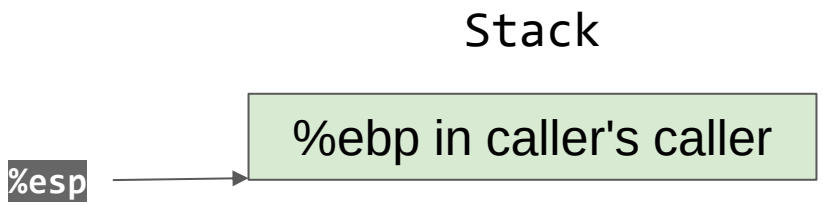
```
int callee(int a, int b, int c) {
    return a + b + c;
}

int caller(void) {
    int ret;
    ret = callee(1, 2, 3);
    ret += 4;
    return ret;
}
```

```
00000012 <caller>:
  12:   55                push   %ebp
  13:   89 e5             mov    %esp,%ebp
  15:   83 ec 10          sub    $0x10,%esp
  18:   6a 03             push   $0x3
  1a:   6a 02             push   $0x2
  1c:   6a 01             push   $0x1
  1e:   e8 fc ff ff ff    call   1f <caller+0xd>
  23:   83 c4 0c          add    $0xc,%esp
  26:   89 45 fc          mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04       addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc          mov    -0x4(%ebp),%eax
  30:   c9                leave
  31:   c3                ret

00000000 <callee>:
   0:   55           push   %ebp
   1:   89 e5        mov    %esp,%ebp
   3:   8b 55 08     mov    0x8(%ebp),%edx
   6:   8b 45 0c     mov    0xc(%ebp),%eax
   9:   01 c2        add    %eax,%edx
   b:   8b 45 10     mov    0x10(%ebp),%eax
   e:   01 d0        add    %edx,%eax
  10:   5d           pop    %ebp
  11:   c3           ret
```

- x86
  - 使用栈来传递参数
  - 使用 eax 存放返回值
- amd64
  - 前6个参数依次存放于 rdi、rsi、rdx、rcx、r8、r9 寄存器中
  - 第7个以后的参数存放于栈中

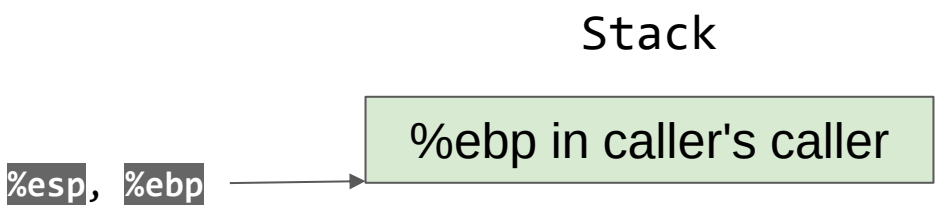
函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
%eip →
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <caller+0xd>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55          push   %ebp
   1:   89 e5       mov    %esp,%ebp
   3:   8b 55 08    mov    0x8(%ebp),%edx
   6:   8b 45 0c    mov    0xc(%ebp),%eax
   9:   01 c2       add    %eax,%edx
   b:   8b 45 10    mov    0x10(%ebp),%eax
   e:   01 d0       add    %edx,%eax
  10:   5d          pop    %ebp
  11:   c3          ret
```



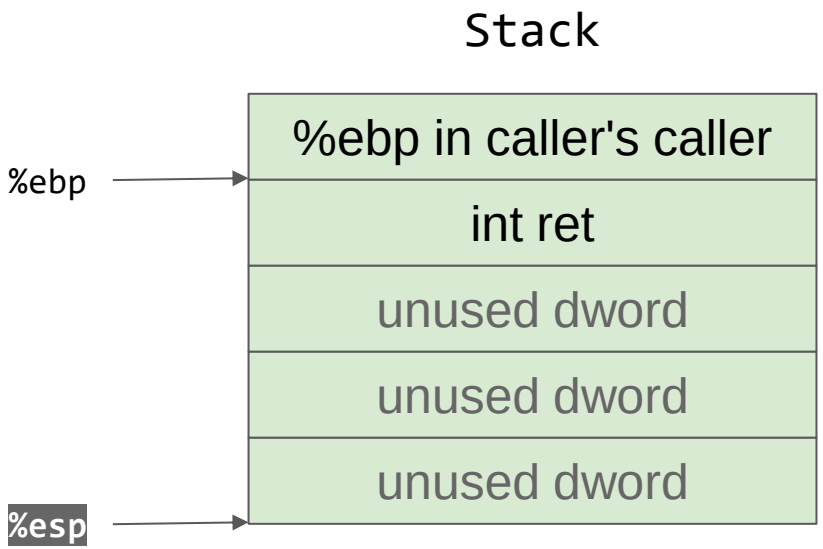
函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
%eip →
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <caller+0xd>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55          push   %ebp
   1:   89 e5       mov    %esp,%ebp
   3:   8b 55 08    mov    0x8(%ebp),%edx
   6:   8b 45 0c    mov    0xc(%ebp),%eax
   9:   01 c2       add    %eax,%edx
   b:   8b 45 10    mov    0x10(%ebp),%eax
   e:   01 d0       add    %edx,%eax
  10:   5d          pop    %ebp
  11:   c3          ret
```



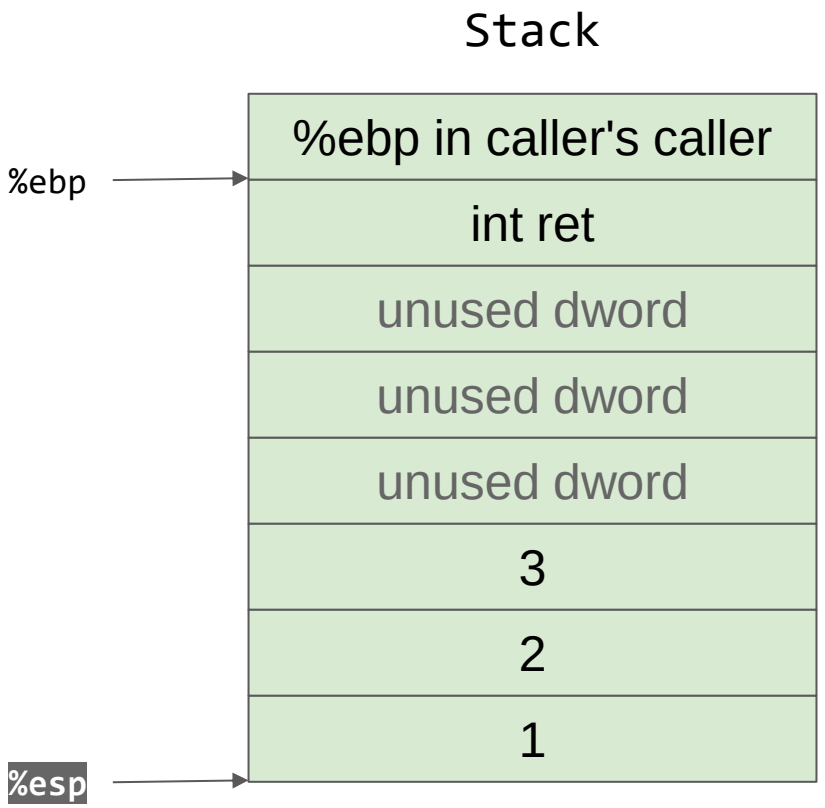
函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
%eip ->18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <caller+0xd>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55          push   %ebp
   1:   89 e5       mov    %esp,%ebp
   3:   8b 55 08    mov    0x8(%ebp),%edx
   6:   8b 45 0c    mov    0xc(%ebp),%eax
   9:   01 c2       add    %eax,%edx
   b:   8b 45 10    mov    0x10(%ebp),%eax
   e:   01 d0       add    %edx,%eax
  10:   5d          pop    %ebp
  11:   c3          ret
```



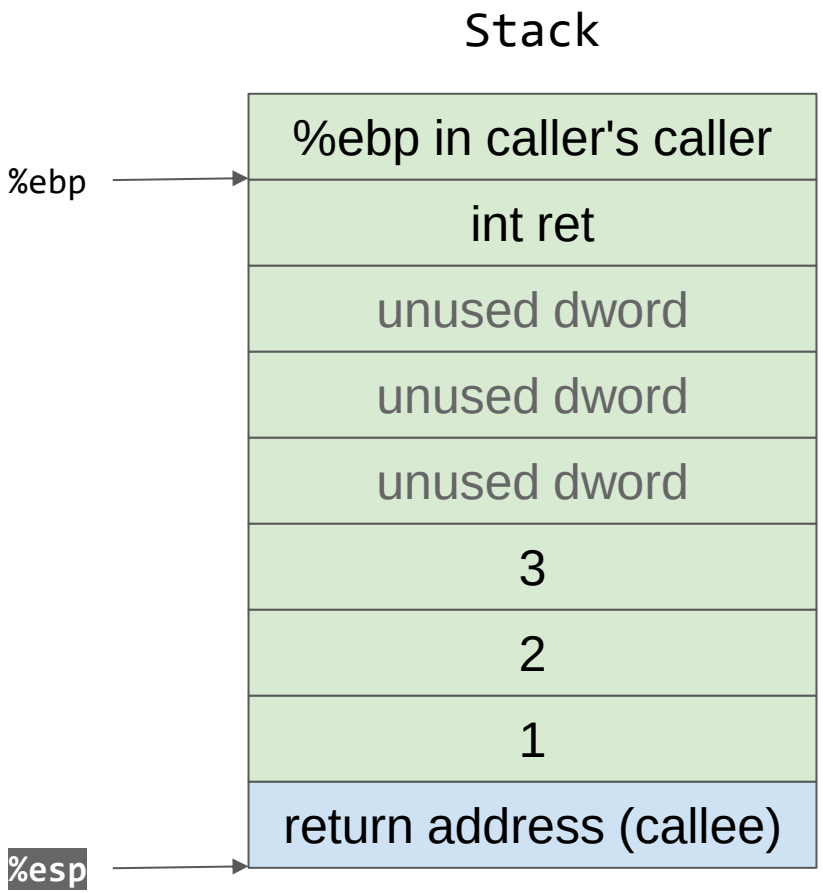
函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
%eip →1e:   e8 fc ff ff ff  call   1f <caller+0xd>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55          push   %ebp
   1:   89 e5       mov    %esp,%ebp
   3:   8b 55 08    mov    0x8(%ebp),%edx
   6:   8b 45 0c    mov    0xc(%ebp),%eax
   9:   01 c2       add    %eax,%edx
   b:   8b 45 10    mov    0x10(%ebp),%eax
   e:   01 d0       add    %edx,%eax
  10:   5d          pop    %ebp
  11:   c3          ret
```



函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
%eip →  0:   55           push   %ebp
   1:   89 e5        mov    %esp,%ebp
   3:   8b 55 08     mov    0x8(%ebp),%edx
   6:   8b 45 0c     mov    0xc(%ebp),%eax
   9:   01 c2        add    %eax,%edx
   b:   8b 45 10     mov    0x10(%ebp),%eax
   e:   01 d0        add    %edx,%eax
  10:   5d           pop    %ebp
  11:   c3           ret
```

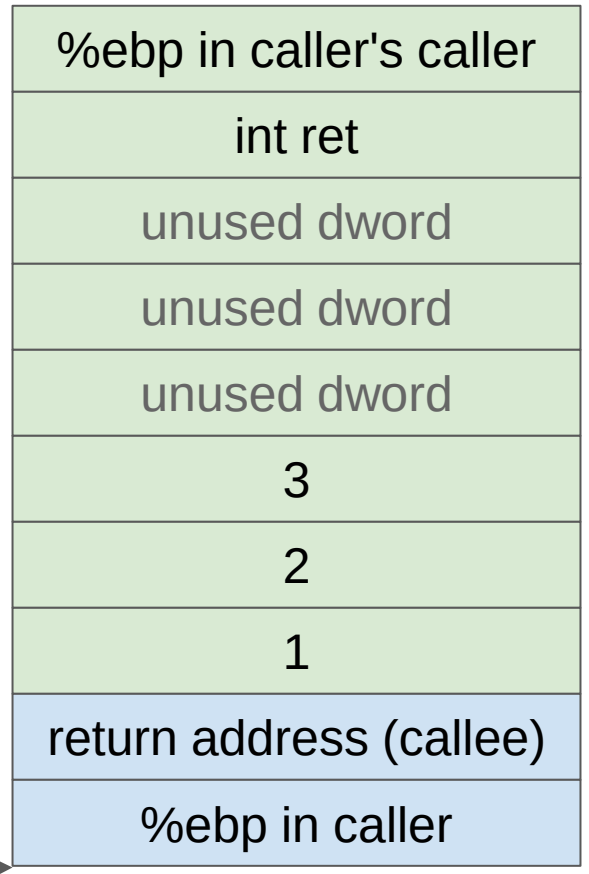


函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55              push   %ebp
   1:   89 e5           mov    %esp,%ebp
   3:   8b 55 08        mov    0x8(%ebp),%edx
   6:   8b 45 0c        mov    0xc(%ebp),%eax
   9:   01 c2           add    %eax,%edx
   b:   8b 45 10        mov    0x10(%ebp),%eax
   e:   01 d0           add    %edx,%eax
  10:   5d              pop    %ebp
  11:   c3              ret
```

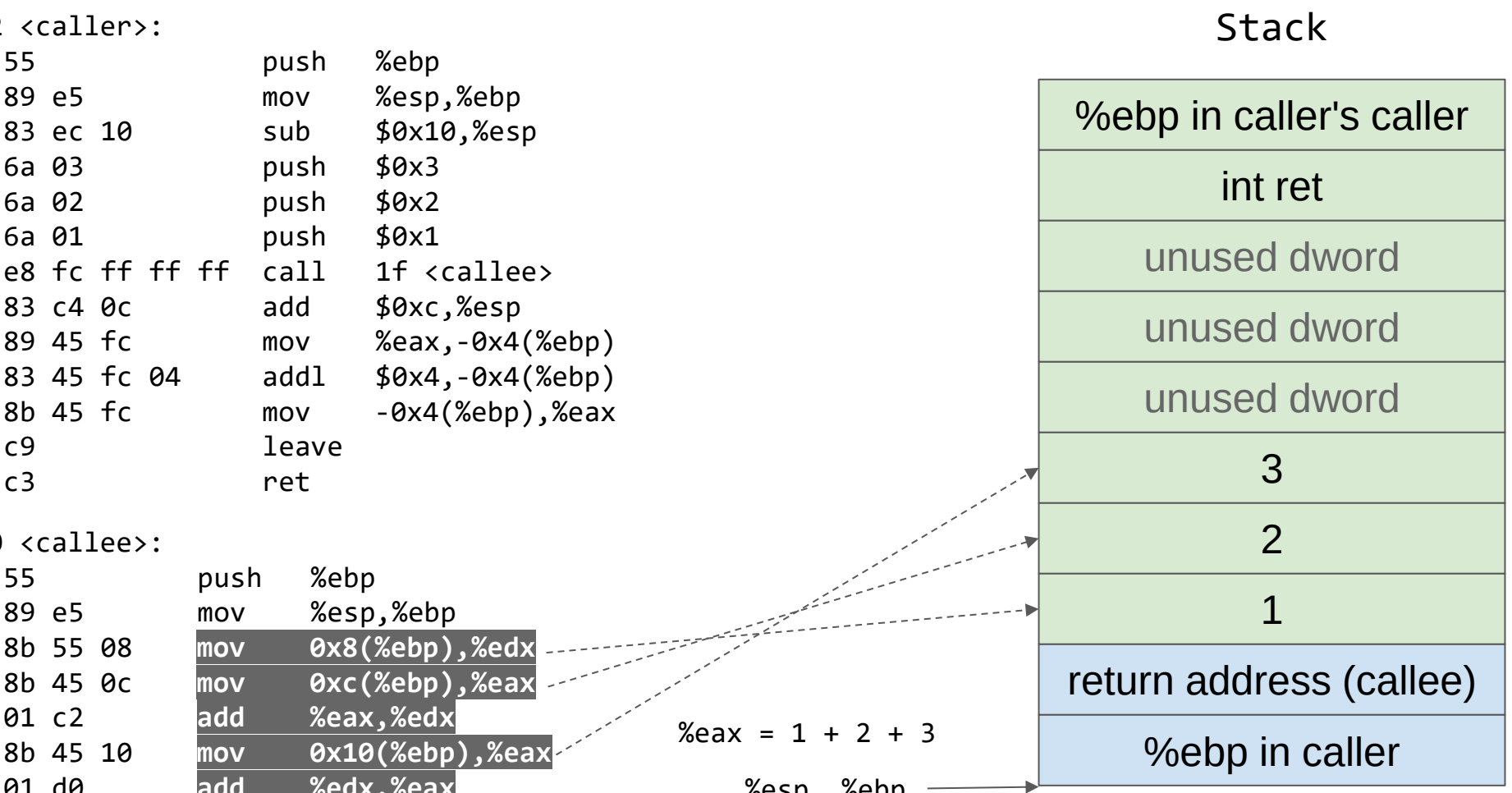
%eip →

Stack



函数调用栈的工作方式（cdecl）

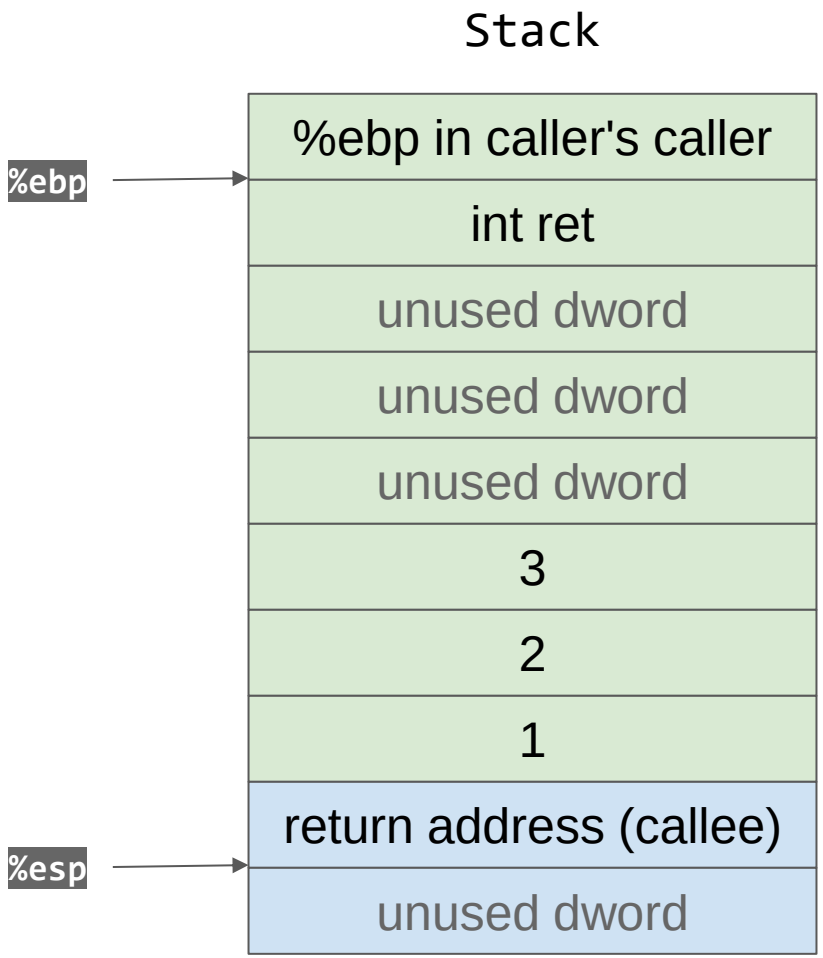
```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55          push   %ebp
   1:   89 e5       mov    %esp,%ebp
   3:   8b 55 08    mov    0x8(%ebp),%edx
   6:   8b 45 0c    mov    0xc(%ebp),%eax
   9:   01 c2       add    %eax,%edx
   b:   8b 45 10    mov    0x10(%ebp),%eax
   e:   01 d0       add    %edx,%eax
%eip → 10:   5d     pop    %ebp
  11:   c3          ret
```



函数调用栈的工作方式（cdecl）

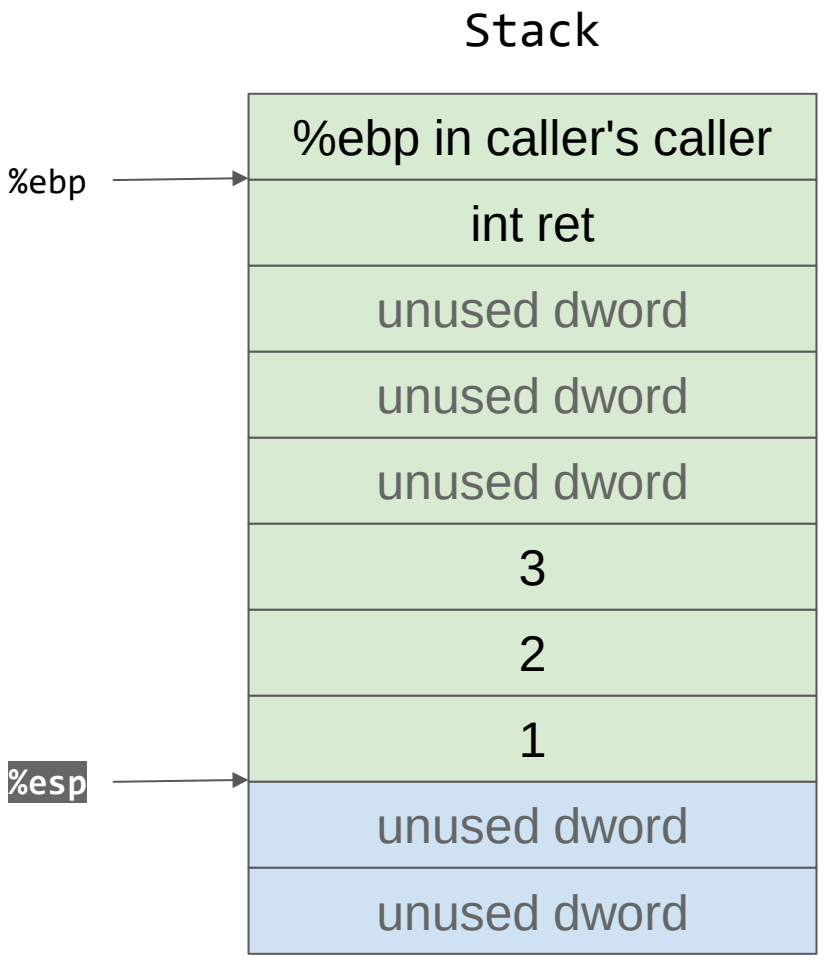
```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55          push   %ebp
   1:   89 e5       mov    %esp,%ebp
   3:   8b 55 08    mov    0x8(%ebp),%edx
   6:   8b 45 0c    mov    0xc(%ebp),%eax
   9:   01 c2       add    %eax,%edx
   b:   8b 45 10    mov    0x10(%ebp),%eax
   e:   01 d0       add    %edx,%eax
  10:   5d          pop    %ebp
  11:   c3          ret
```

%eip →



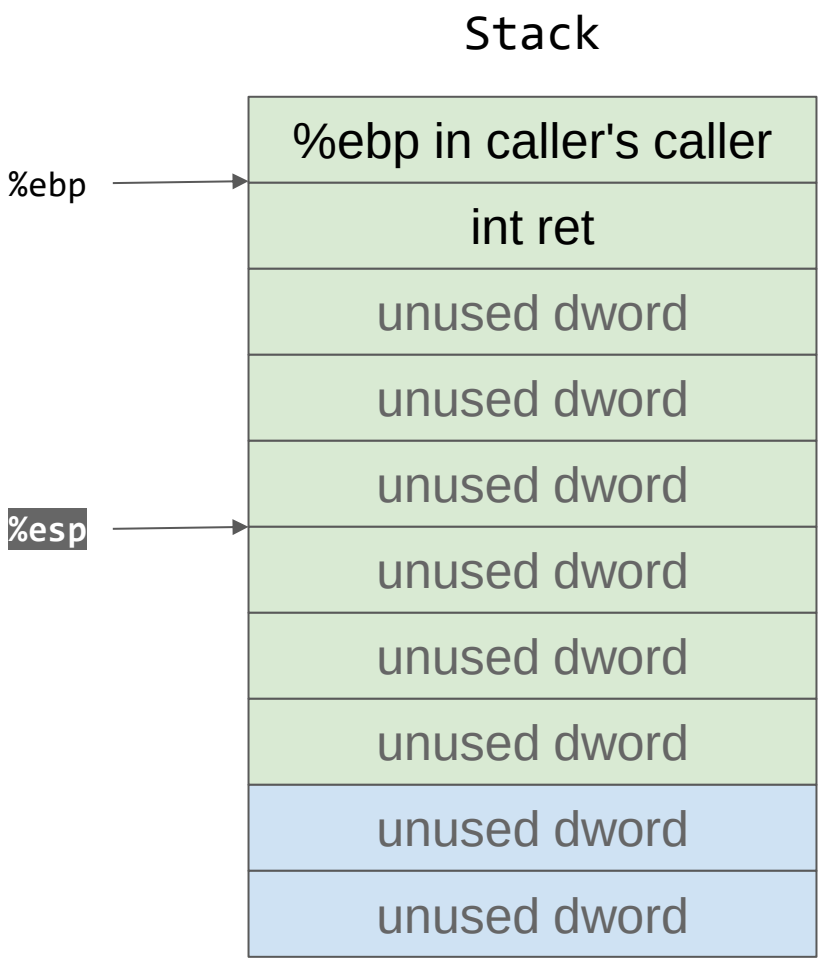
函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
%eip→ 23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55          push   %ebp
   1:   89 e5       mov    %esp,%ebp
   3:   8b 55 08    mov    0x8(%ebp),%edx
   6:   8b 45 0c    mov    0xc(%ebp),%eax
   9:   01 c2       add    %eax,%edx
   b:   8b 45 10    mov    0x10(%ebp),%eax
   e:   01 d0       add    %edx,%eax
  10:   5d          pop    %ebp
  11:   c3          ret
```



函数调用栈的工作方式（cdecl）

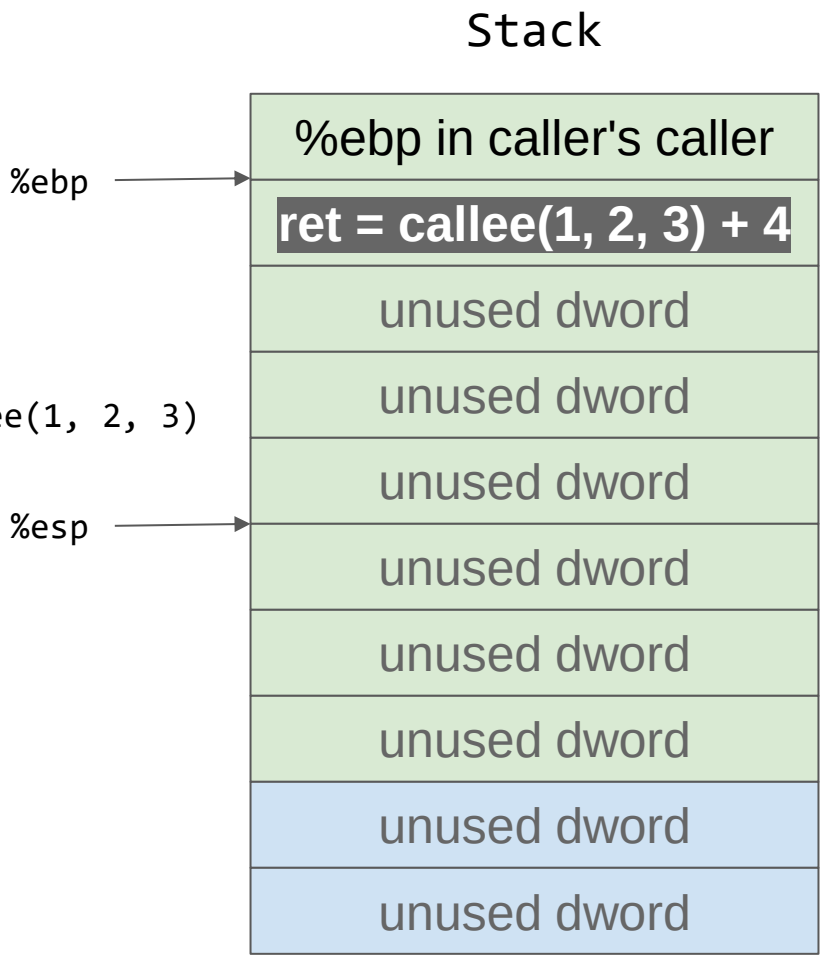
```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
%eip ->
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55          push   %ebp
   1:   89 e5       mov    %esp,%ebp
   3:   8b 55 08    mov    0x8(%ebp),%edx
   6:   8b 45 0c    mov    0xc(%ebp),%eax
   9:   01 c2       add    %eax,%edx
   b:   8b 45 10    mov    0x10(%ebp),%eax
   e:   01 d0       add    %edx,%eax
  10:   5d          pop    %ebp
  11:   c3          ret
```



函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
%eip → 30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55              push   %ebp
   1:   89 e5           mov    %esp,%ebp
   3:   8b 55 08        mov    0x8(%ebp),%edx
   6:   8b 45 0c        mov    0xc(%ebp),%eax
   9:   01 c2           add    %eax,%edx
   b:   8b 45 10        mov    0x10(%ebp),%eax
   e:   01 d0           add    %edx,%eax
  10:   5d              pop    %ebp
  11:   c3              ret
```

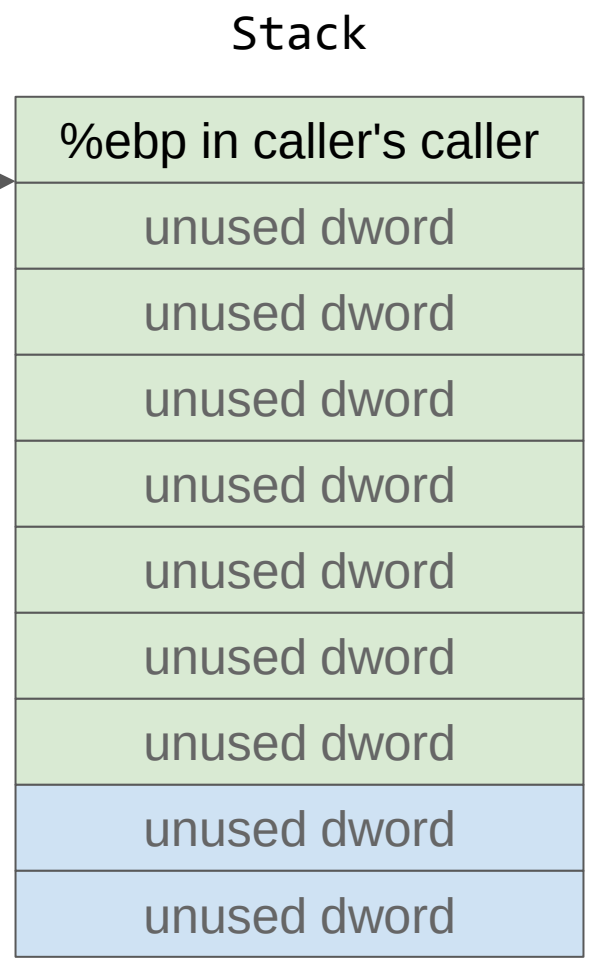
%eax = callee(1, 2, 3)



函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
%eip
  31:   c3              ret

00000000 <callee>:
   0:   55              push   %ebp
   1:   89 e5           mov    %esp,%ebp
   3:   8b 55 08        mov    0x8(%ebp),%edx
   6:   8b 45 0c        mov    0xc(%ebp),%eax
   9:   01 c2           add    %eax,%edx
   b:   8b 45 10        mov    0x10(%ebp),%eax
   e:   01 d0           add    %edx,%eax
  10:   5d              pop    %ebp
  11:   c3              ret
```

```
mov   %ebp, $esp
pop   %ebp
```

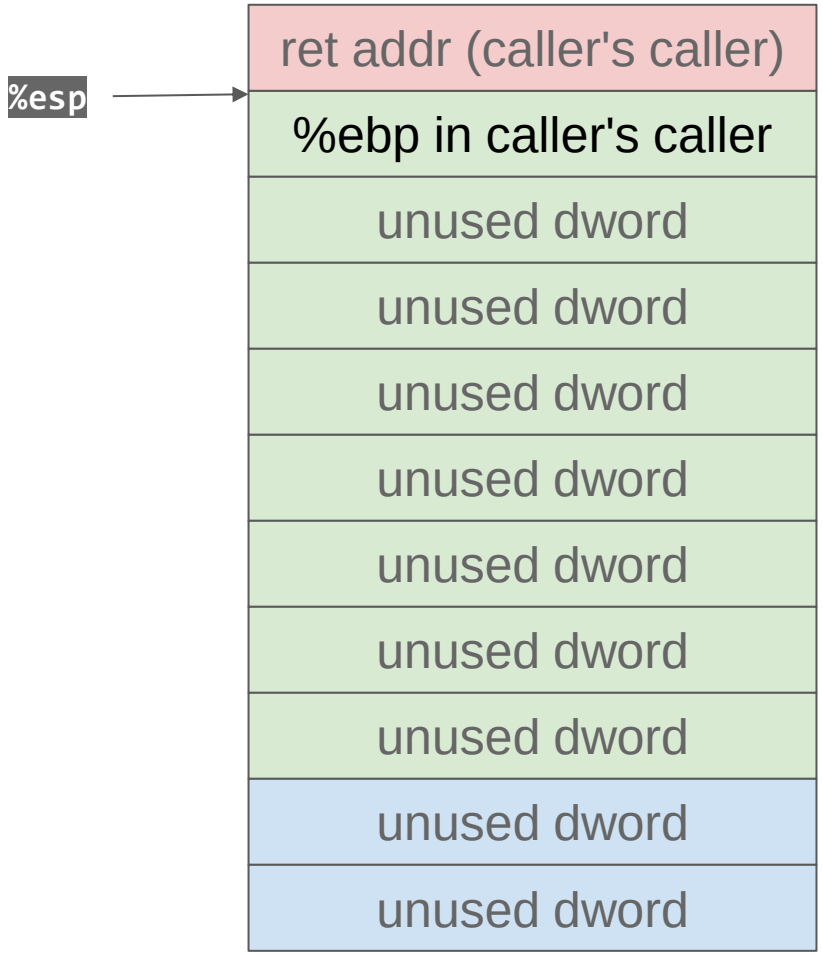


函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
%eip
  31:   c3              ret
```

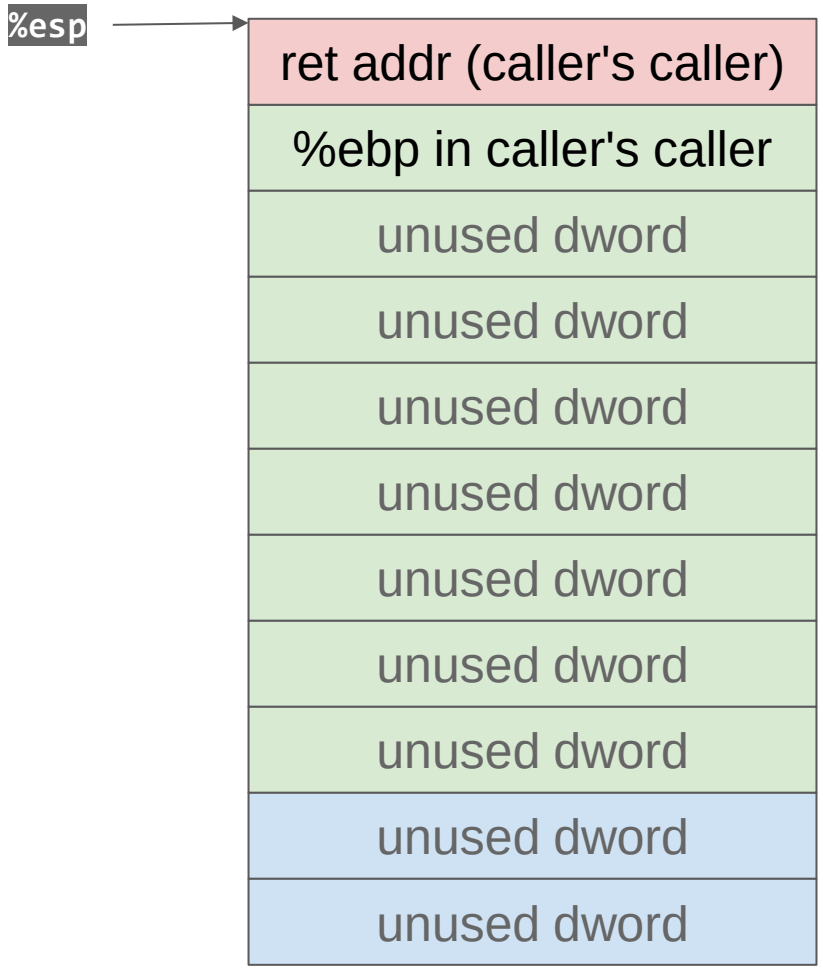
leave →
```
mov   %ebp, $esp
pop   %ebp
```

```
00000000 <callee>:
   0:   55          push   %ebp
   1:   89 e5       mov    %esp,%ebp
   3:   8b 55 08    mov    0x8(%ebp),%edx
   6:   8b 45 0c    mov    0xc(%ebp),%eax
   9:   01 c2       add    %eax,%edx
   b:   8b 45 10    mov    0x10(%ebp),%eax
   e:   01 d0       add    %edx,%eax
  10:   5d          pop    %ebp
  11:   c3          ret
```



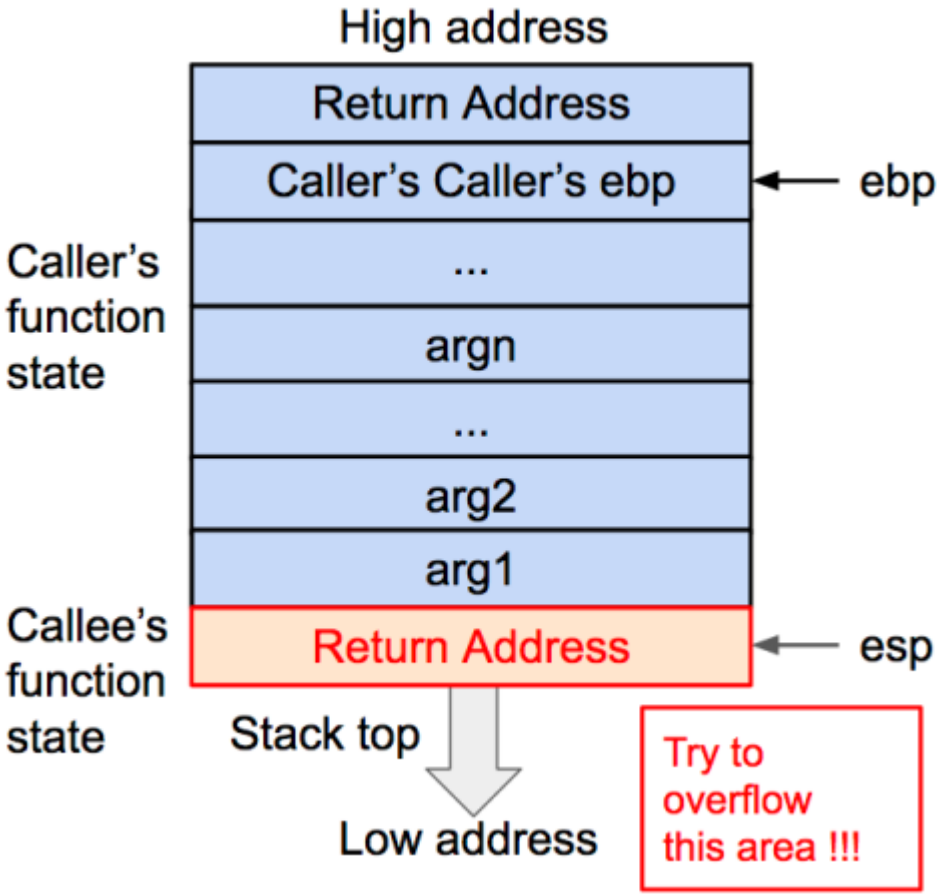
函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55          push   %ebp
   1:   89 e5       mov    %esp,%ebp
   3:   8b 55 08    mov    0x8(%ebp),%edx
   6:   8b 45 0c    mov    0xc(%ebp),%eax
   9:   01 c2       add    %eax,%edx
   b:   8b 45 10    mov    0x10(%ebp),%eax
   e:   01 d0       add    %edx,%eax
  10:   5d          pop    %ebp
  11:   c3          ret
```



- 介绍完背景知识，就可以继续回归栈溢出攻击的主题了。当函数正在执行内部指令的过程中我们无法拿到程序的控制权，只有在发生函数调用或者结束函数调用时，程序的控制权会在函数状态之间发生跳转，这时才可以通过修改函数状态来实现攻击。而控制程序执行指令最关键的寄存器就是 eip，所以我们的目标就是让 eip 载入攻击指令的地址。

- 先来看看函数调用结束时，如果要让 eip 指向攻击指令，需要哪些准备？首先，在退栈过程中，返回地址会被传给 eip，所以我们只需要让溢出数据用攻击指令的地址来覆盖返回地址就可以了。其次，我们可以在溢出数据内包含一段攻击指令，也可以在内存其他位置寻找可用的攻击指令。

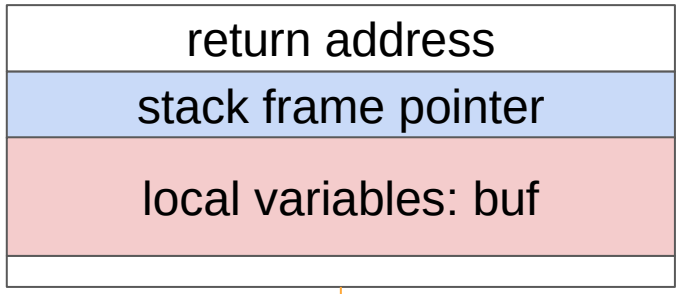


缓冲区溢出（Buffer overflow）

本质是向定长的缓冲区中写入了超长的数据，造成超出的数据覆写了合法内存区域

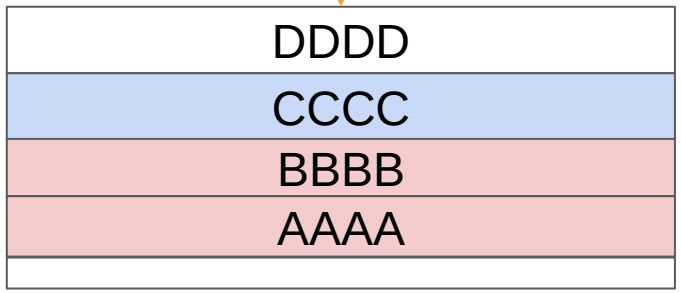
- 栈溢出（Stack overflow）
  - 最常见、漏洞比例最高、危害最大的二进制漏洞
  - 在 CTF PWN 中往往是漏洞利用的基础
- 堆溢出（Heap overflow）
  - 堆管理器复杂，利用花样繁多
  - CTF PWN 中的常见题型
- Data段溢出
  - 攻击效果依赖于 Data段 上存放了何种控制数据

栈溢出



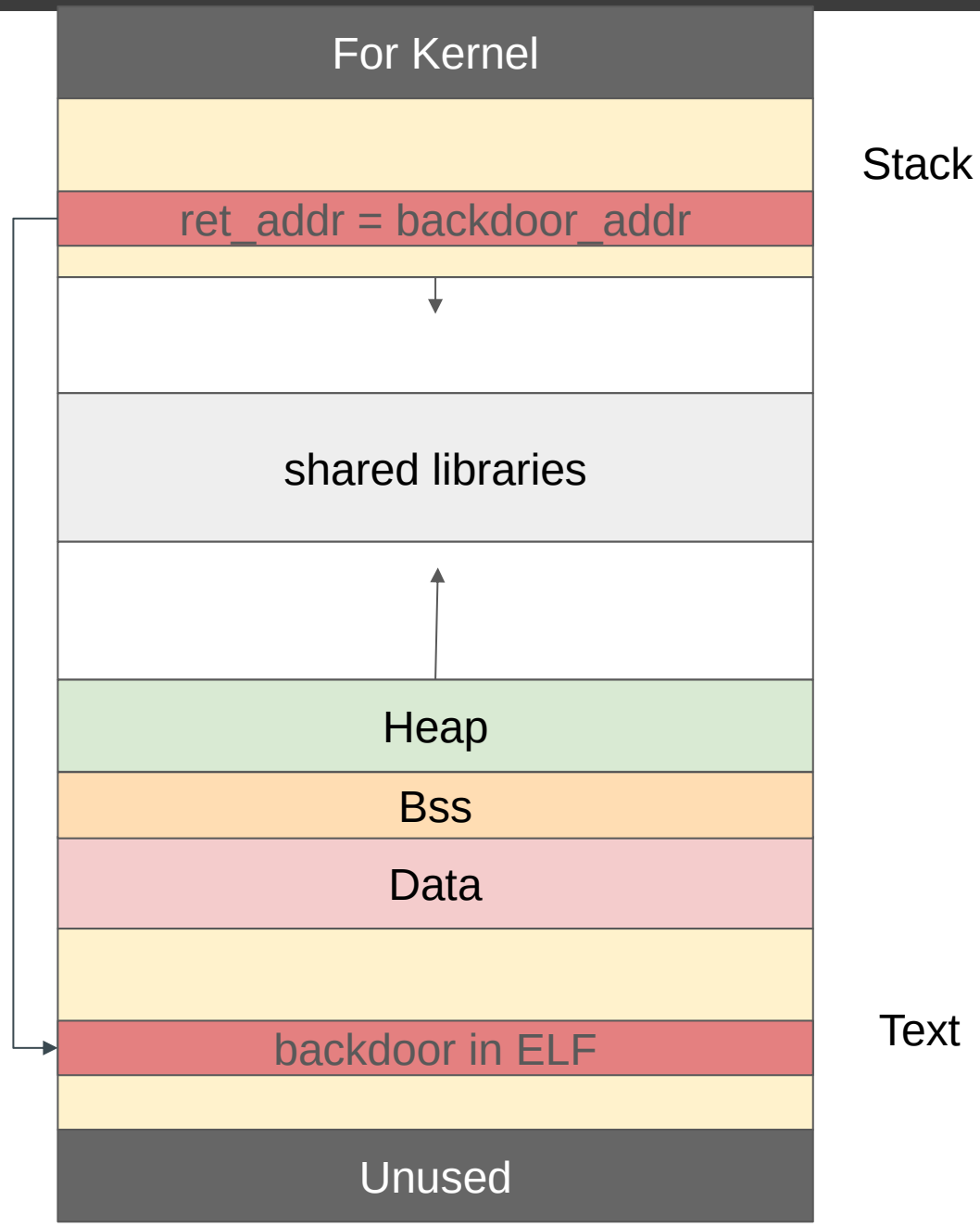

```
int overflow()
{
      char buf[8];          ←
      read(0, buf, 16);
}
```

输入：AAAABBBBCCCCDDDD



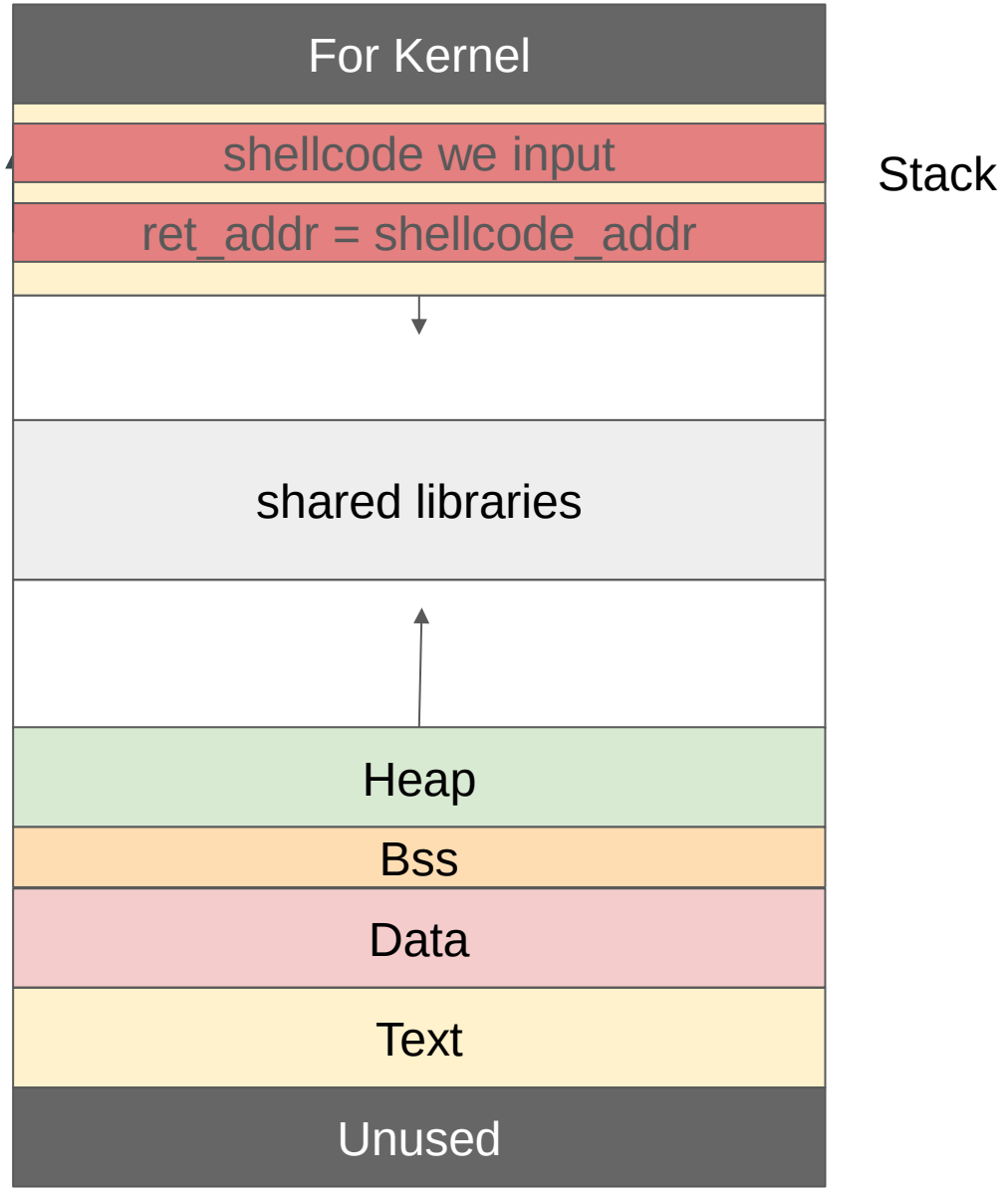

```
int overflow()
{
      char buf[8];
      read(0, buf, 16);     ←
}
```

- 篡改栈帧上的返回地址为程序中已有的后门函数



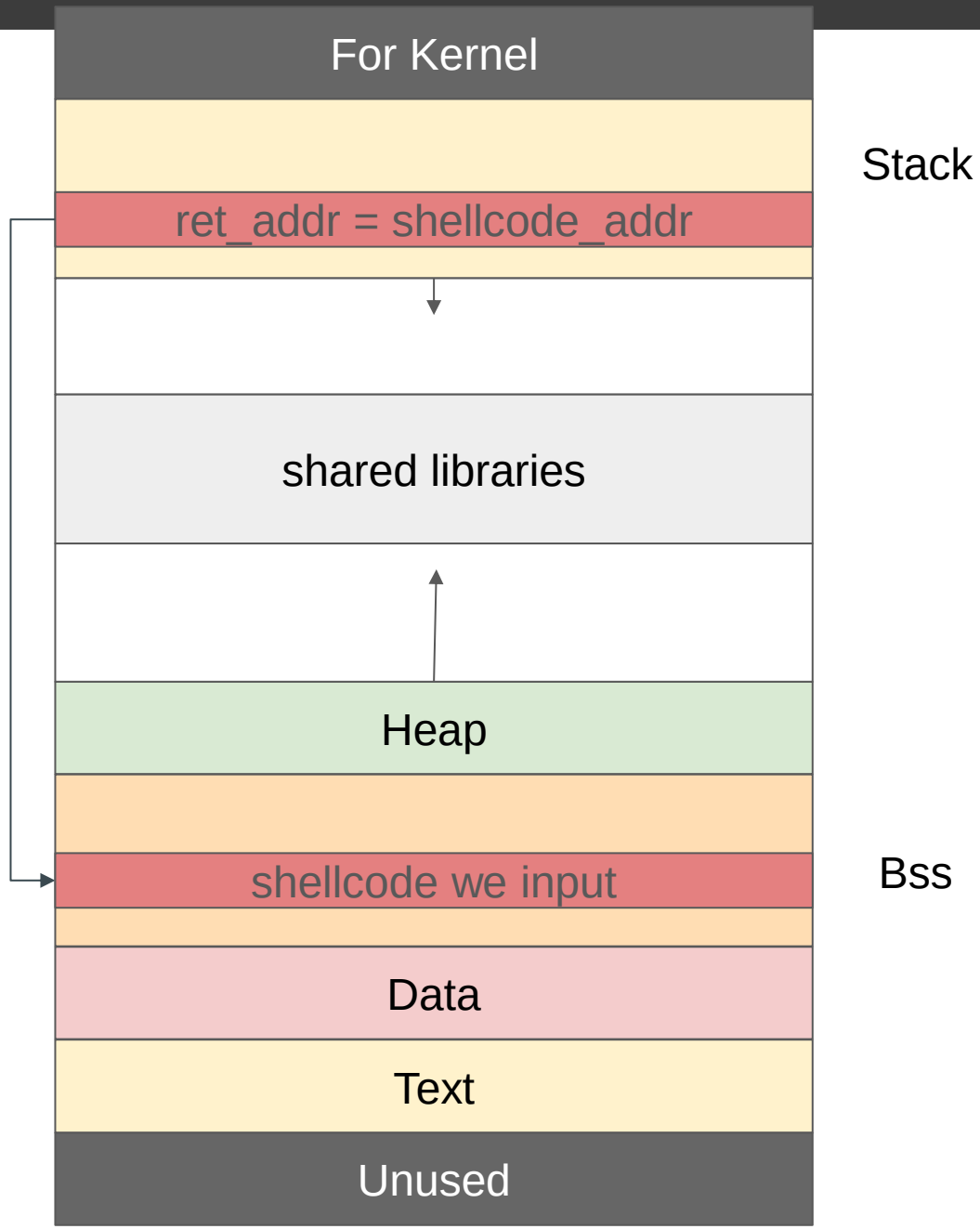
栈溢出基础 | ret2shellcode

- 篡改栈帧上的返回地址为攻击者手动传入的 shellcode 所在缓冲区地址
- 初期往往将 shellcode 直接写入栈缓冲区
- 目前由于 the NX bits 保护措施的开启，栈缓冲区不可执行，故当下的常用手段变为向 bss 缓冲区写入 shellcode 或向堆缓冲区写入 shellcode 并使用 mprotect 赋予其可执行权限



栈溢出基础 | ret2shellcode

- 篡改栈帧上的返回地址为攻击者手动传入的 shellcode 所在缓冲区地址
- 初期往往将 shellcode 直接写入栈缓冲区
- 目前由于 the NX bits 保护措施的开启，栈缓冲区不可执行，故当下的常用手段变为向 bss 缓冲区写入 shellcode 或向堆缓冲区写入 shellcode 并使用 mprotect 赋予其可执行权限



Part3 返回导向编程

- ret2syscall
- 动态链接过程
- ret2libc
- 其它的ROP技巧

什么是系统调用?

- 操作系统提供给用户的编程接口
- 是提供访问操作系统所管理的底层硬件的接口
- 本质上是一些内核函数代码，以规范的方式驱动硬件
- x86 通过 int 0x80 指令进行系统调用、amd64 通过 syscall 指令进行系统调用

举例

- my_puts() -> write() -> sys_write()
  - my_puts(“Hello world!”);
    - 程序 ELF 中的用户代码
  - write(1, &”Hello world!”, 12);
    - libc 中的用户代码
  - [eax = 4; ebx = 1; ecx = &”Hello world!”; edx = 12;] + int 0x80; => **sys_write()**
    - Linux 内核中的内核代码

Q:

可是在程序中没有已存在的一段代码是:

```
mov eax, 0xb
mov ebx, [“/bin/sh”]
mov ecx, 0
mov edx, 0
int 0x80
=> execve("/bin/sh",NULL,NULL)
```

我们仍然要执行 execve("/bin/sh",NULL,NULL)
该怎么做呢?

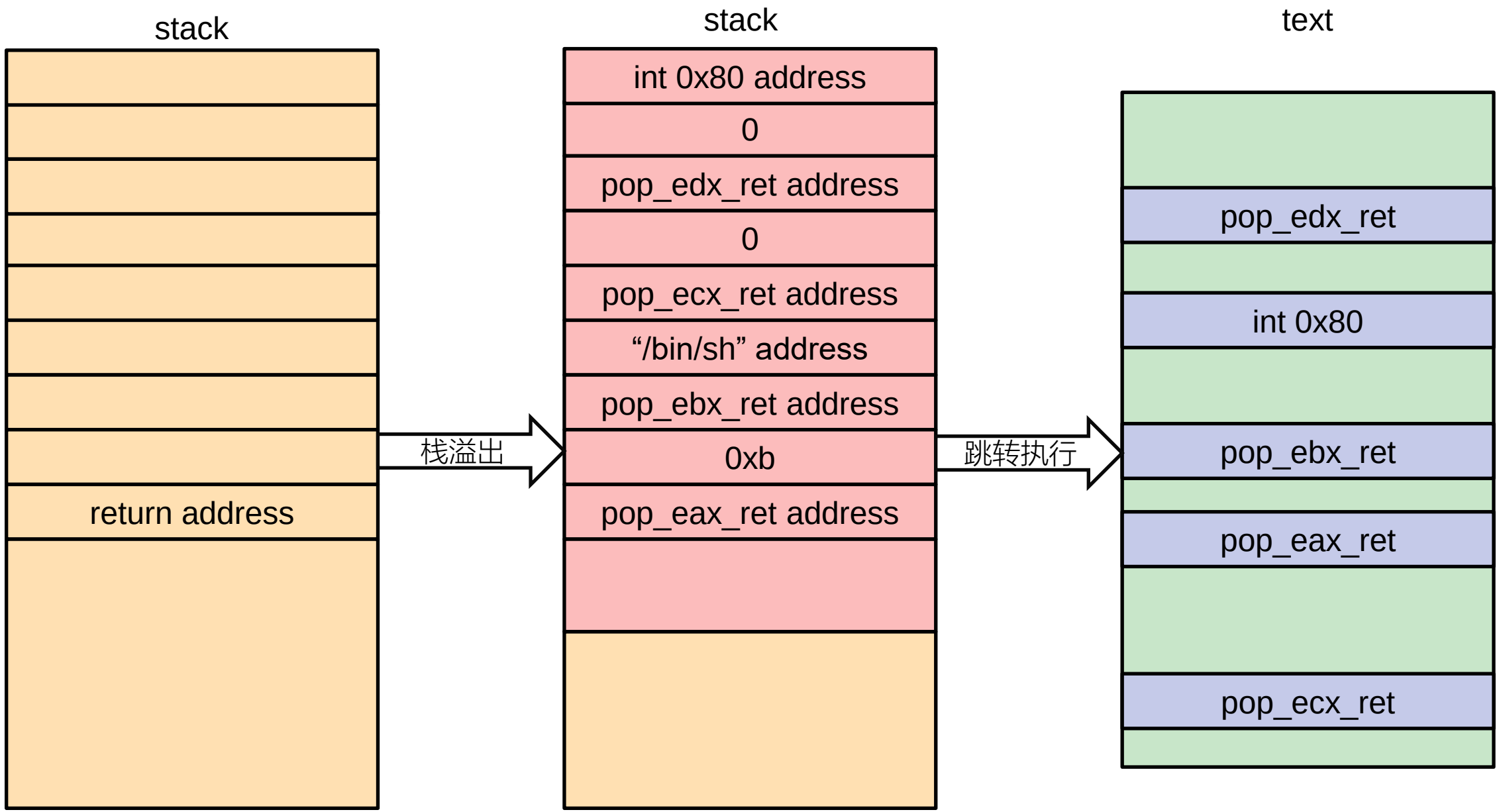
A:

ROP

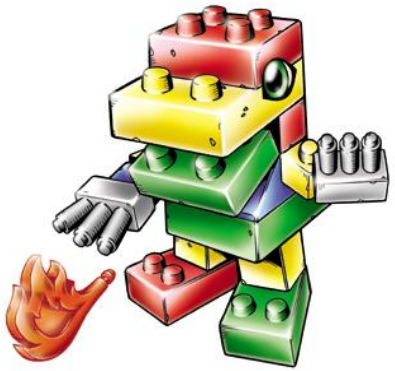
Return Oriented Programming

返回导向编程

返回导向编程 | ret2syscall



gadget

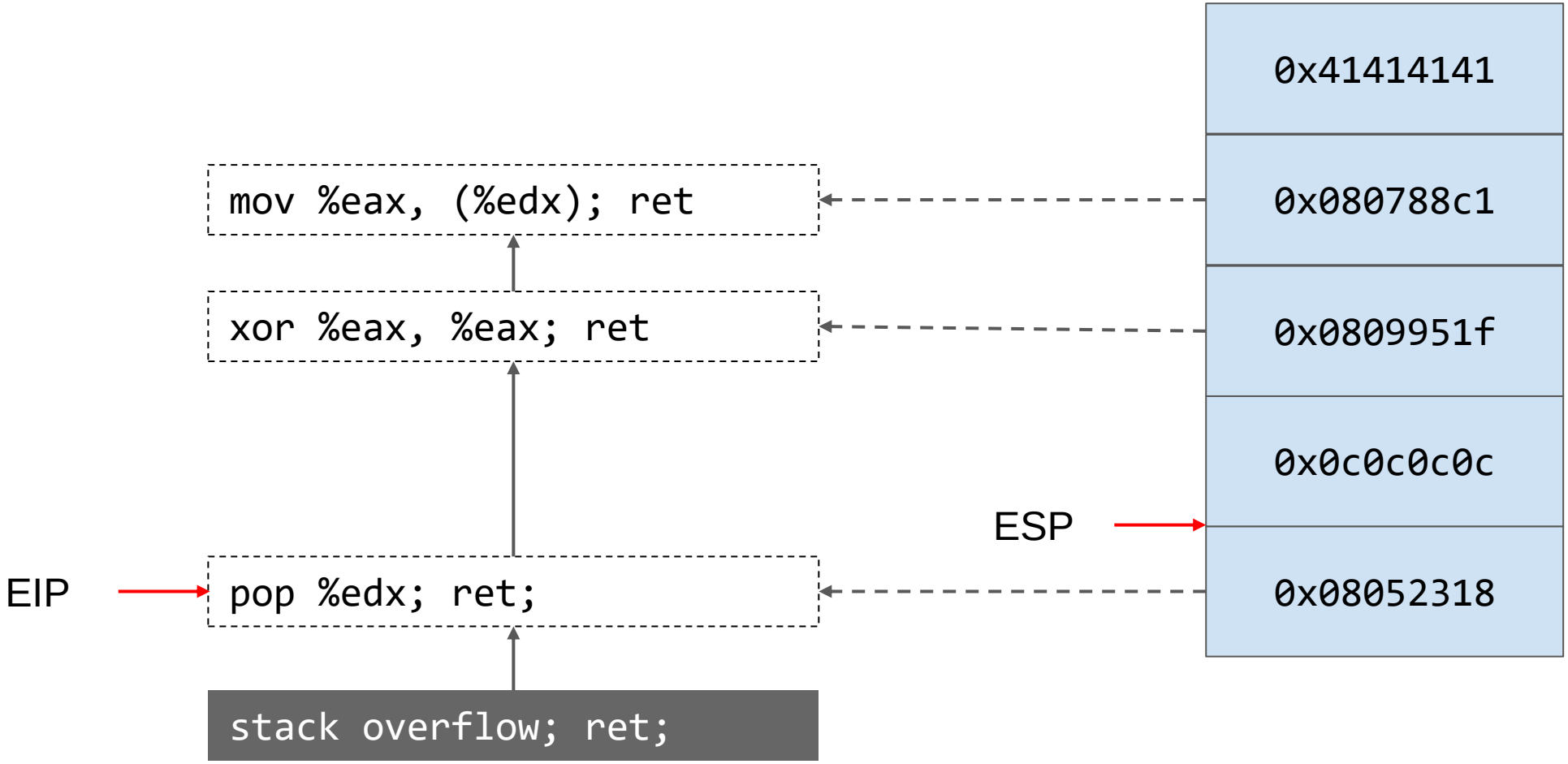


payload

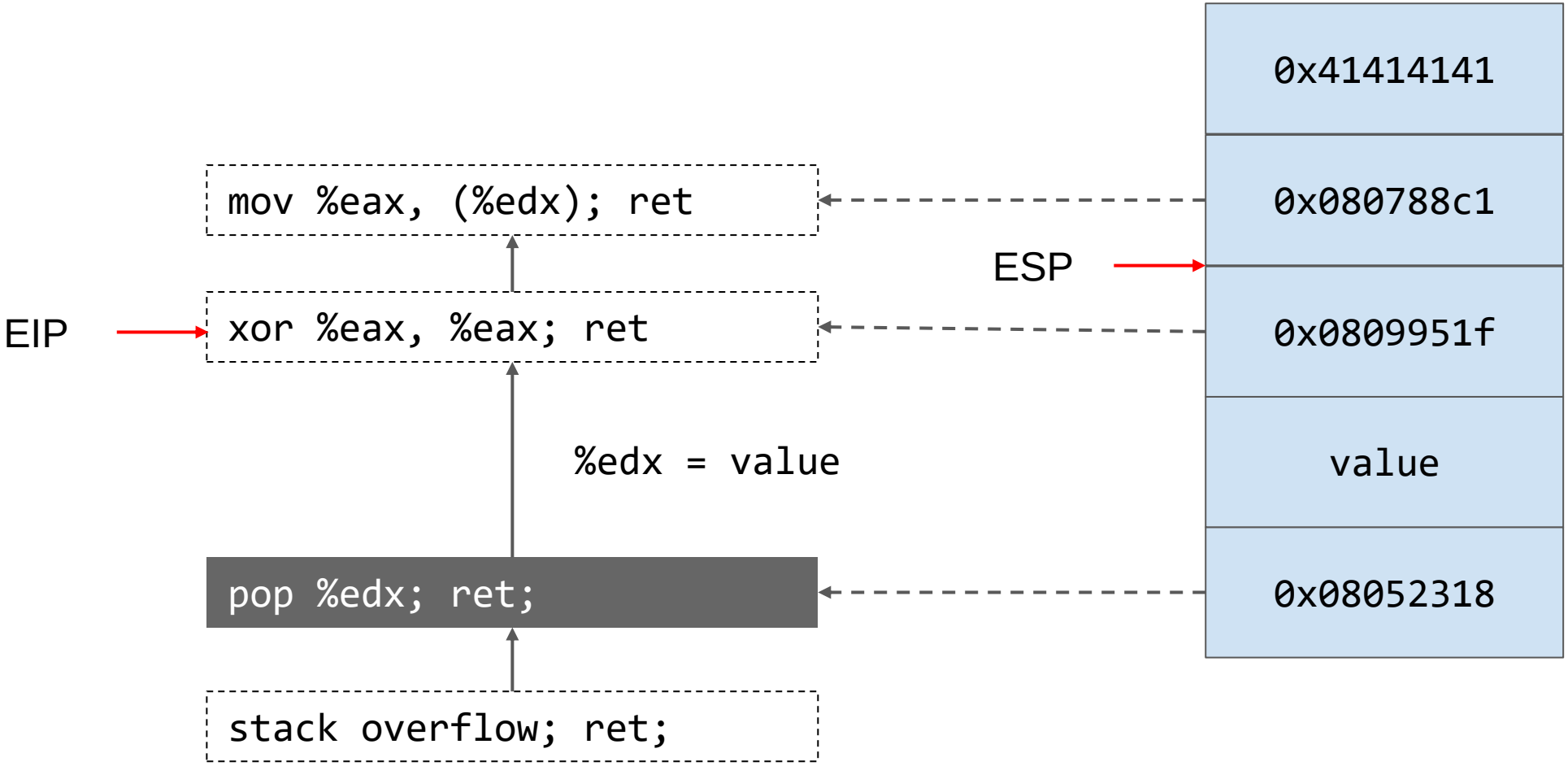
gadget

- pop_edx_ret
- int 0x80
- pop_ebx_ret
- pop_eax_ret
- pop_ecx_ret

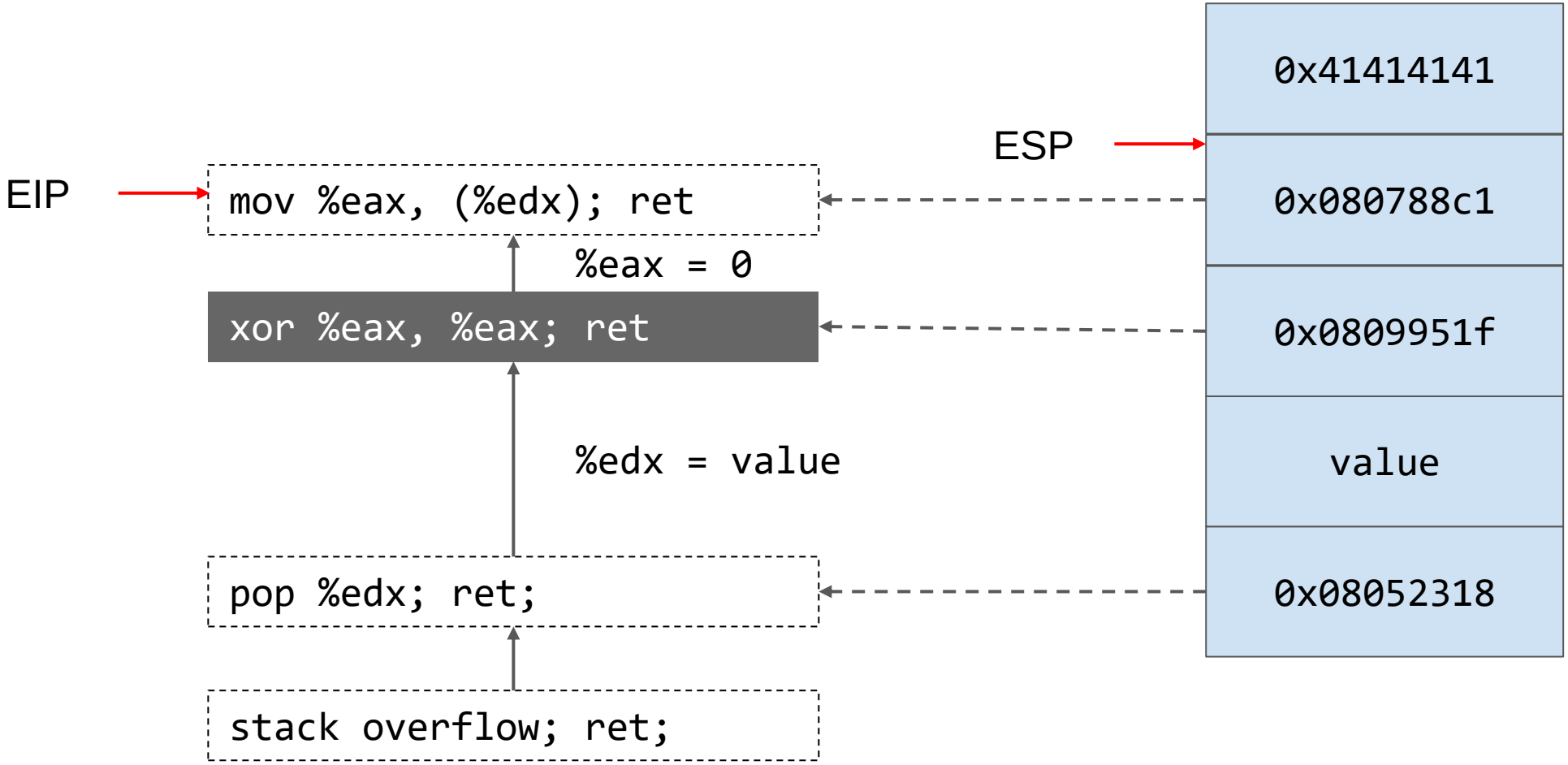
ROP(Return Oriented Programming)



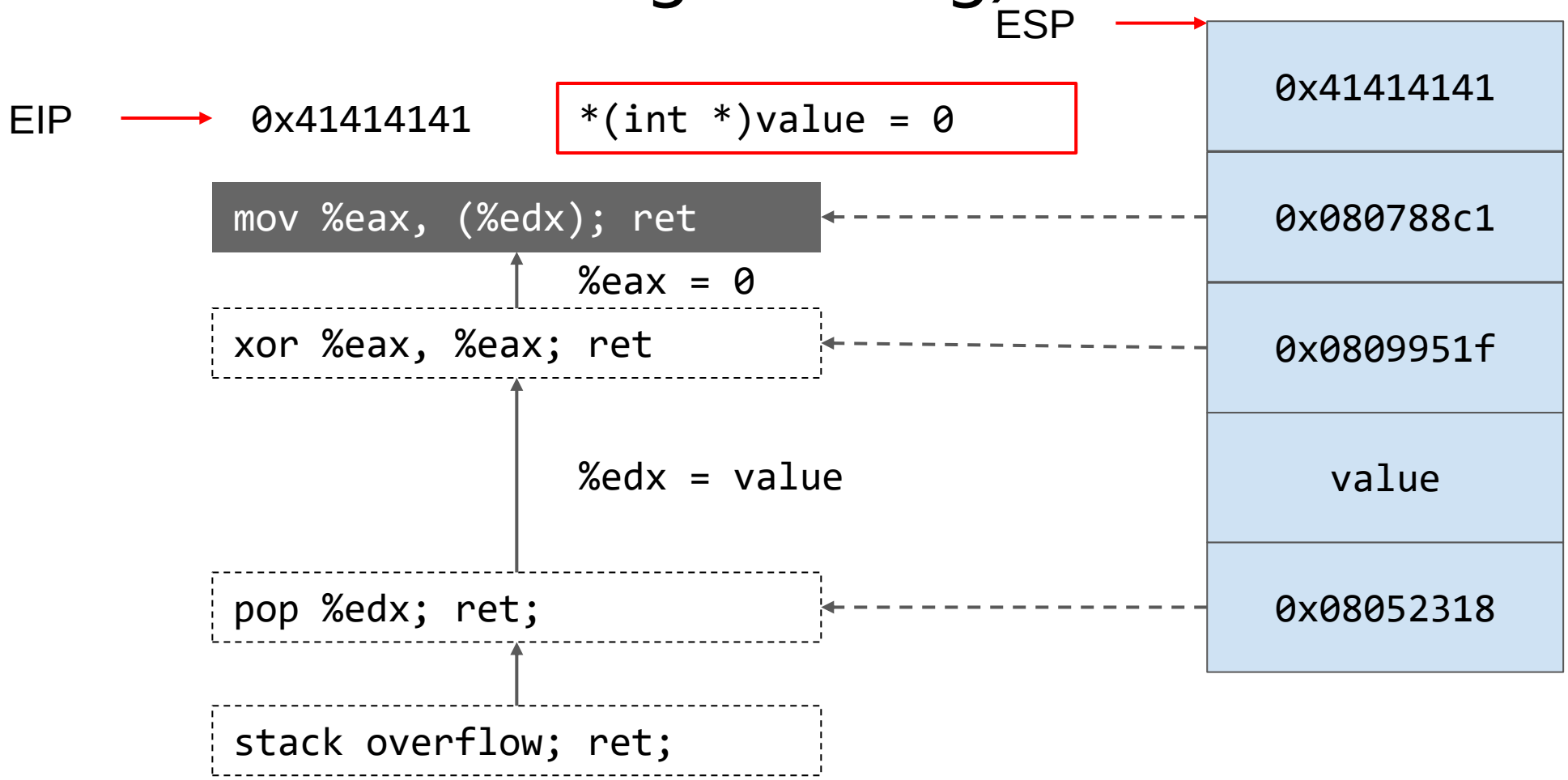
ROP(Return Oriented Programming)



ROP(Return Oriented Programming)

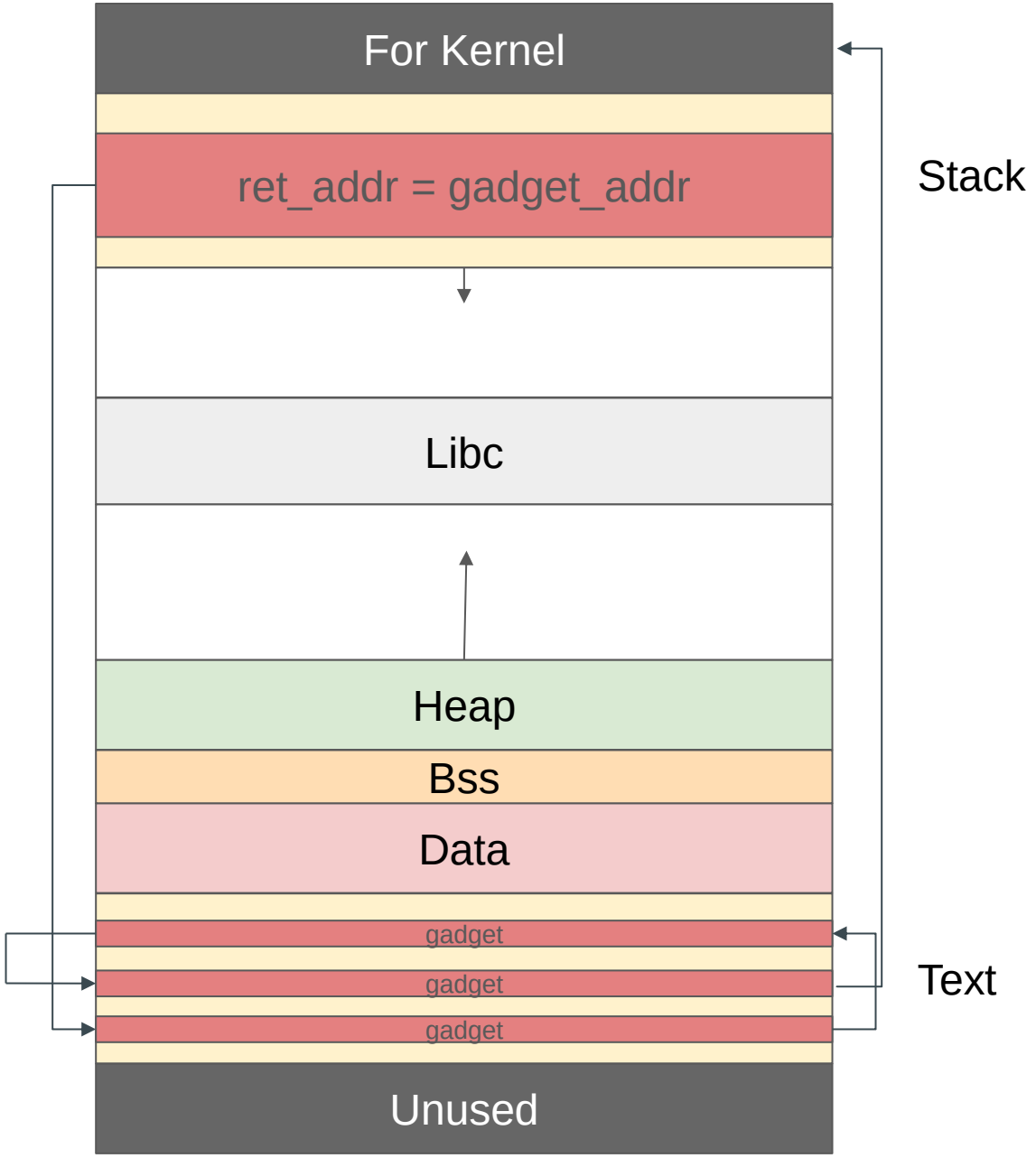


ROP(Return Oriented Programming)



返回导向编程 | ret2syscall

- 篡改栈帧上自返回地址开始的一段区域为一系列 gadget 的地址，最终调用目标系统调用

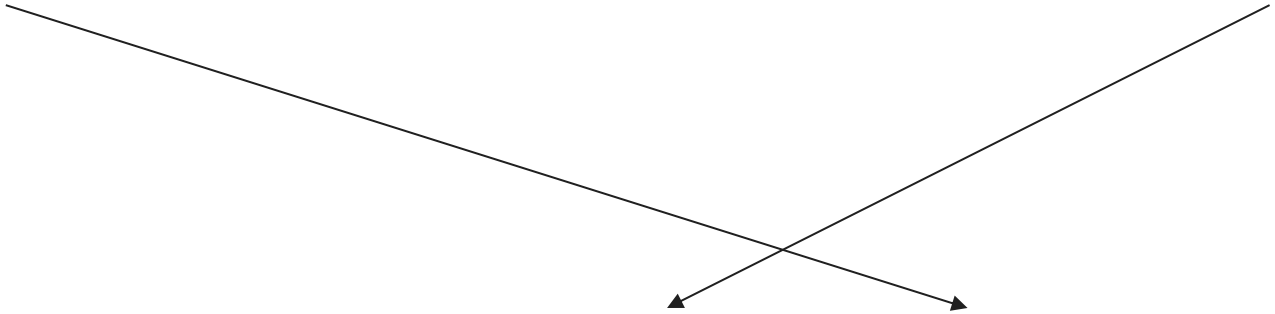


动态链接

- 程序装载进入内存时加载库代码解析外部引用

静态链接

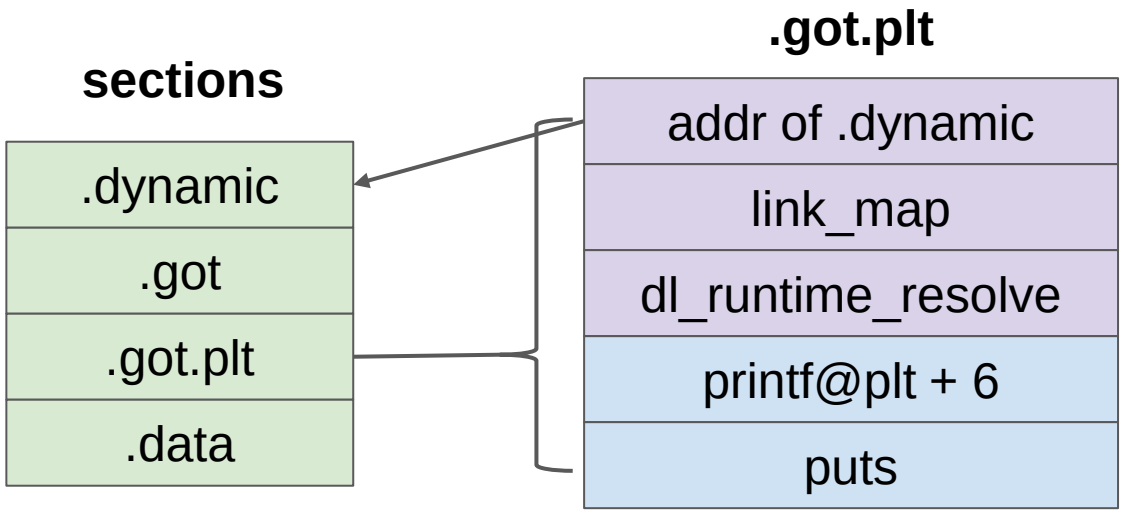
- 链接器在编译链接时将库代码加入到可执行程序中



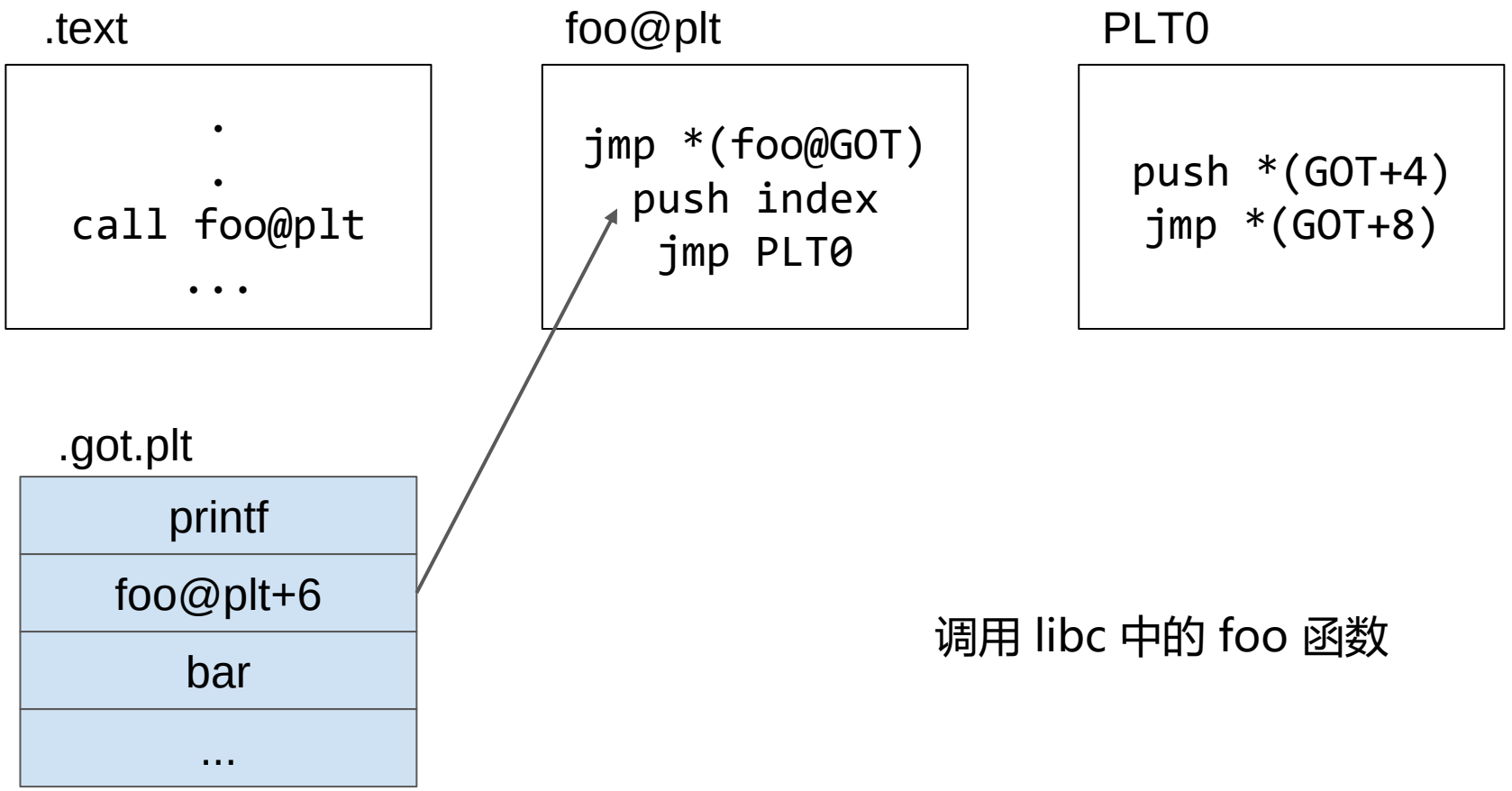
编译 → 汇编 → 链接 → 装载 → 执行

C代码 → 汇编代码 → 目标文件 → 可执行文件 → 进程映像

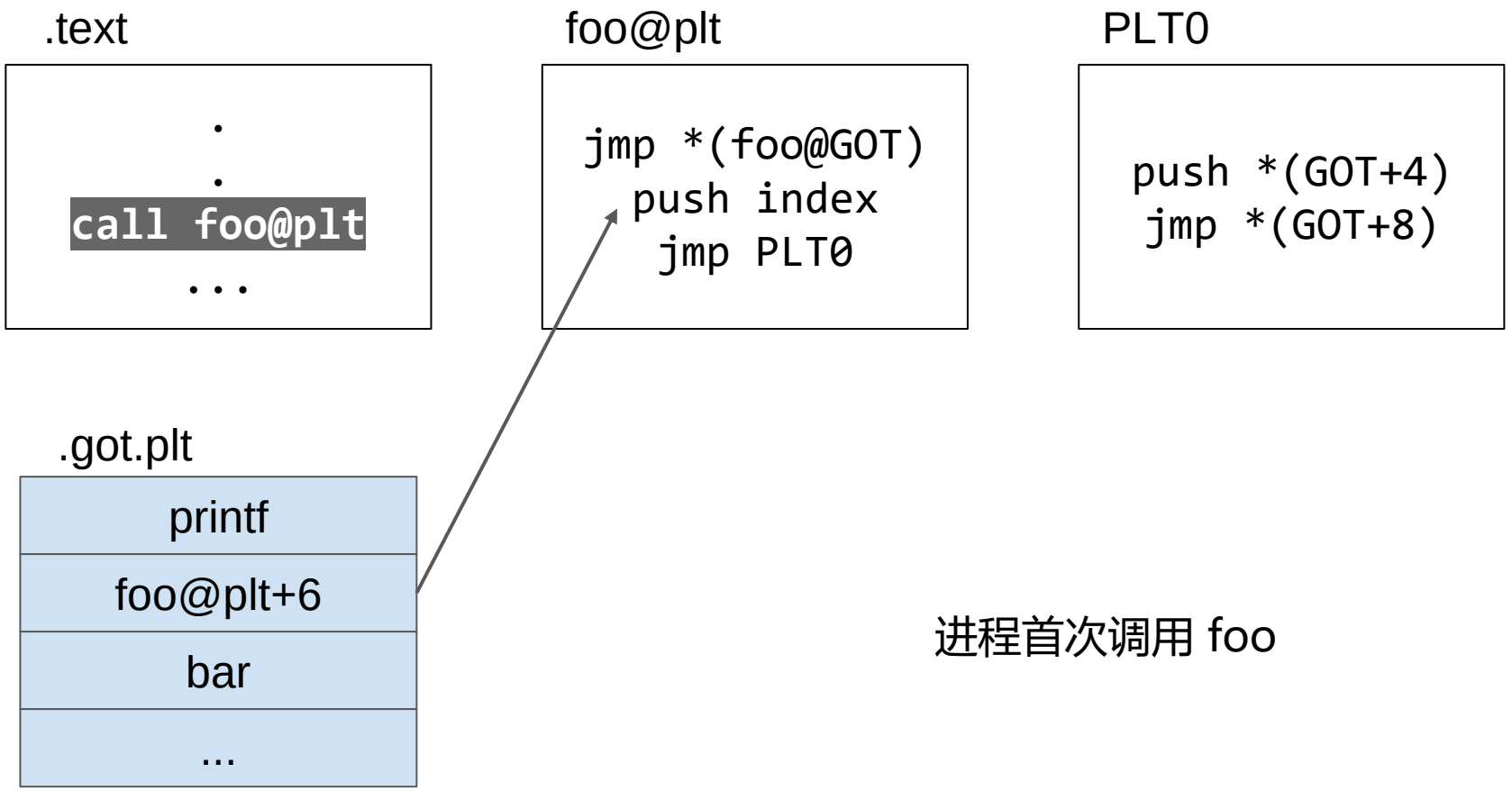
动态链接相关结构



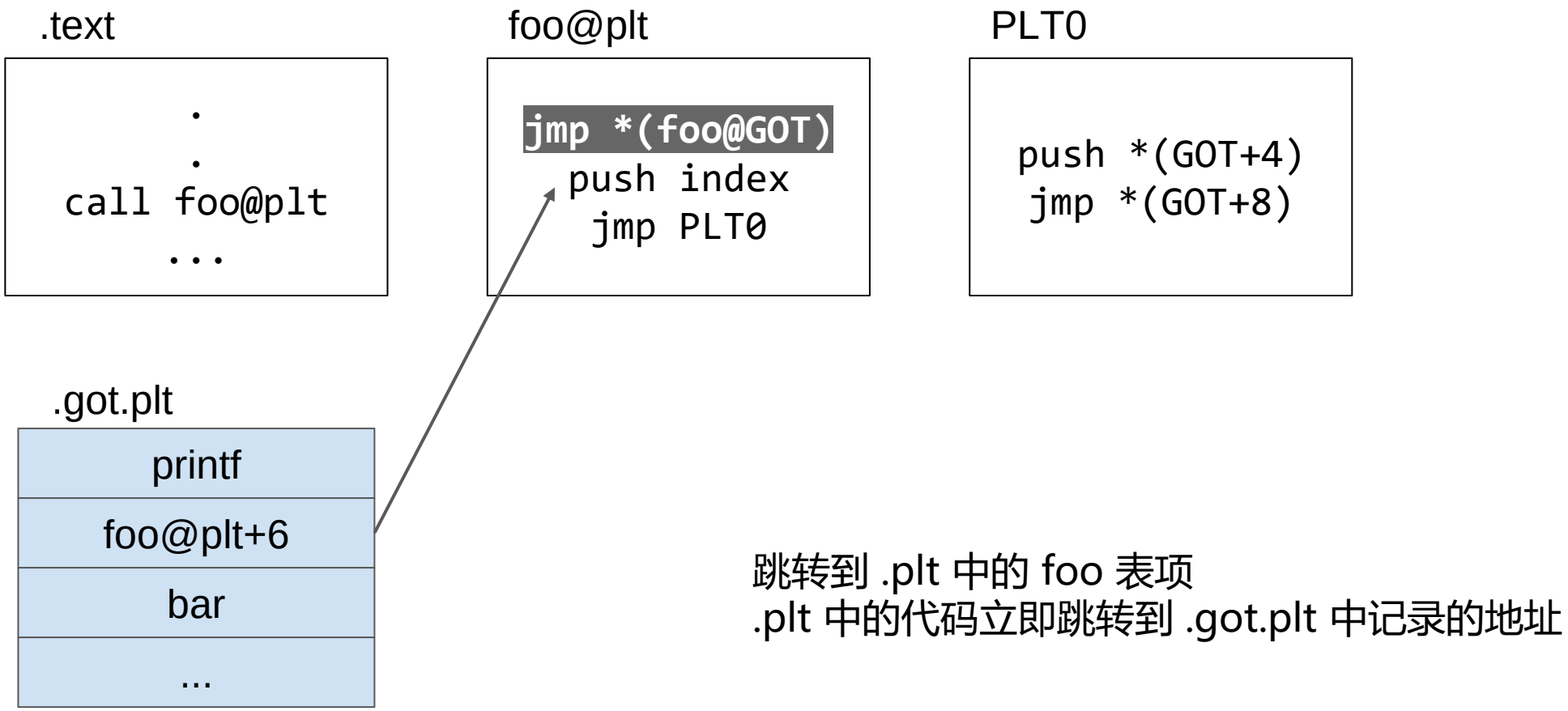
- .dynamic section
  - 提供动态链接相关信息
- link_map
  - 保存进程载入的动态链接库的链表
- __dl_runtime_resolve
  - 装载器中用于解析动态链接库中函数的实际地址的函数



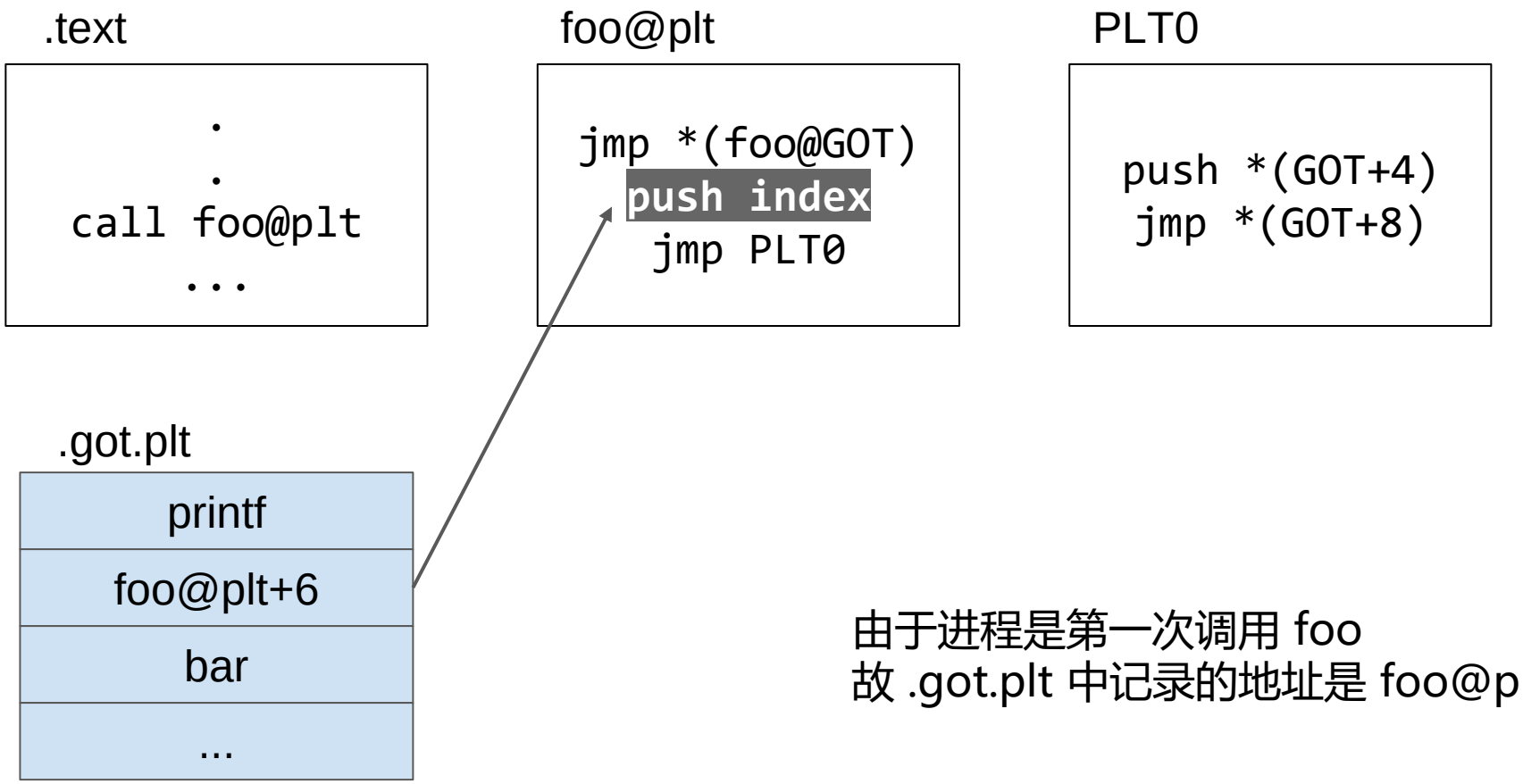
调用 libc 中的 foo 函数


.text
.
.
call foo@plt
...
foo@plt
jmp *(foo@GOT)
push index
jmp PLT0
PLT0
push *(GOT+4)
jmp *(GOT+8)
.got.plt
printf
foo@plt+6
bar
...

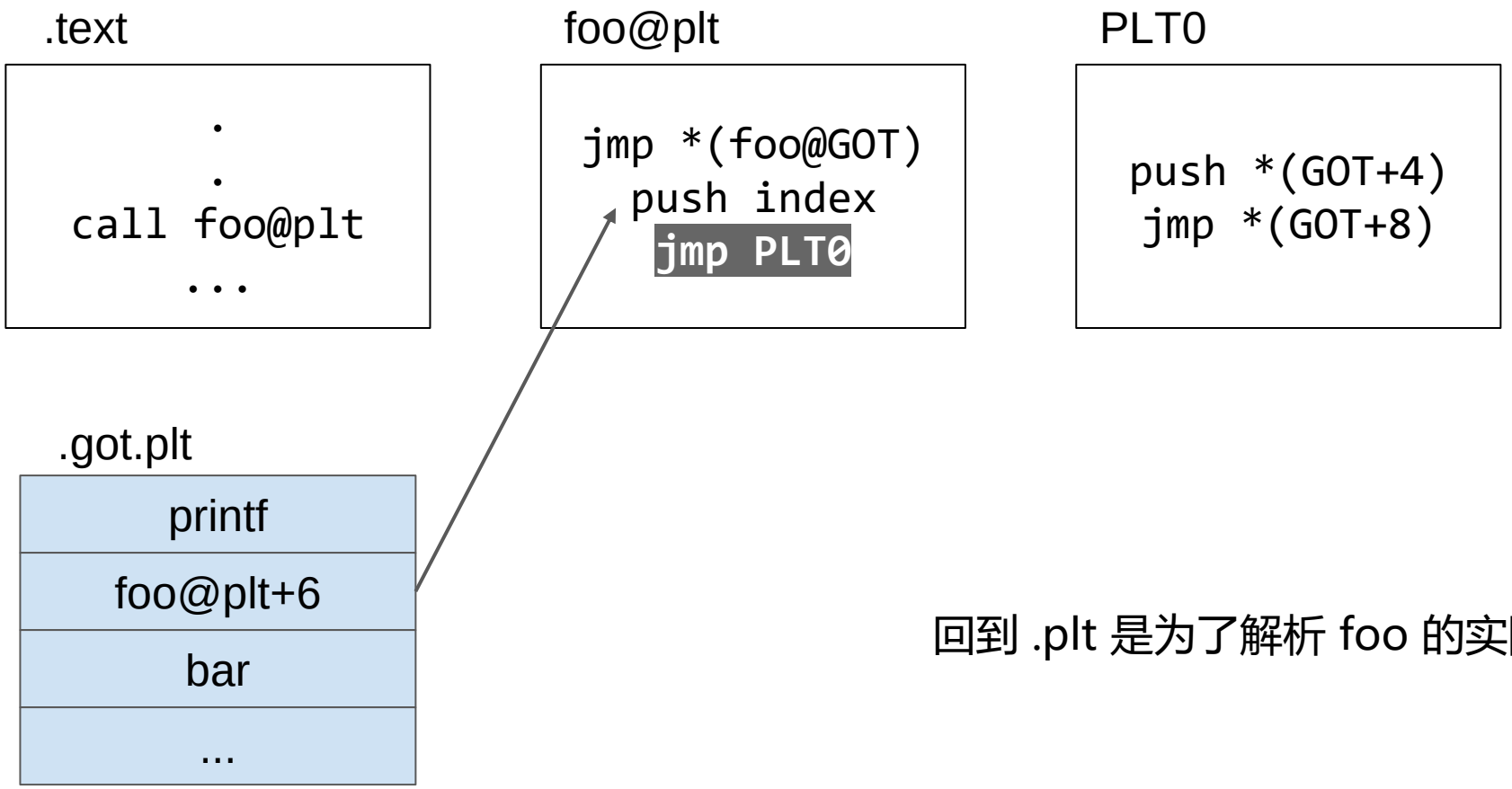
进程首次调用 foo



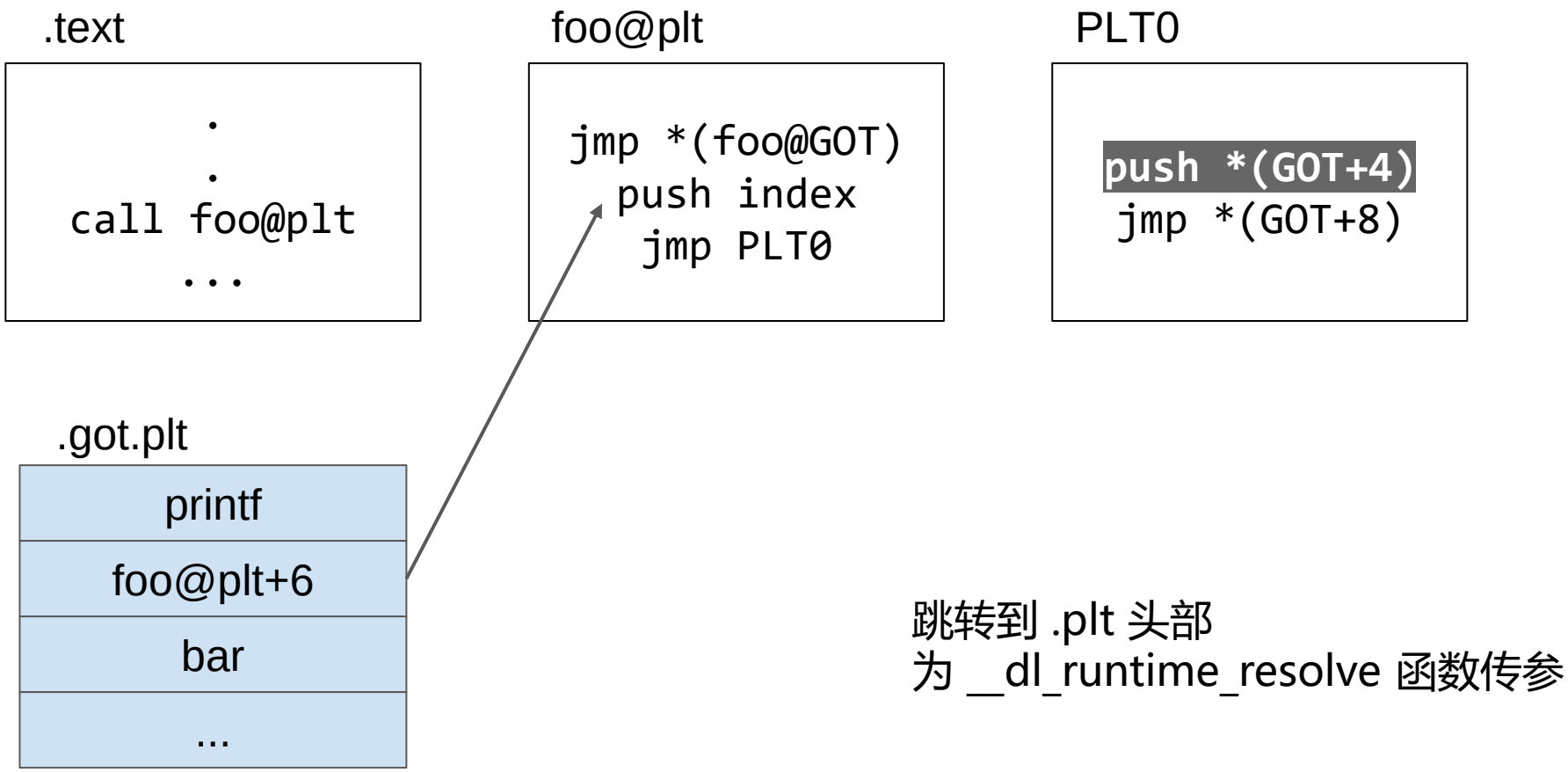
跳转到 .plt 中的 foo 表项
.plt 中的代码立即跳转到 .got.plt 中记录的地址



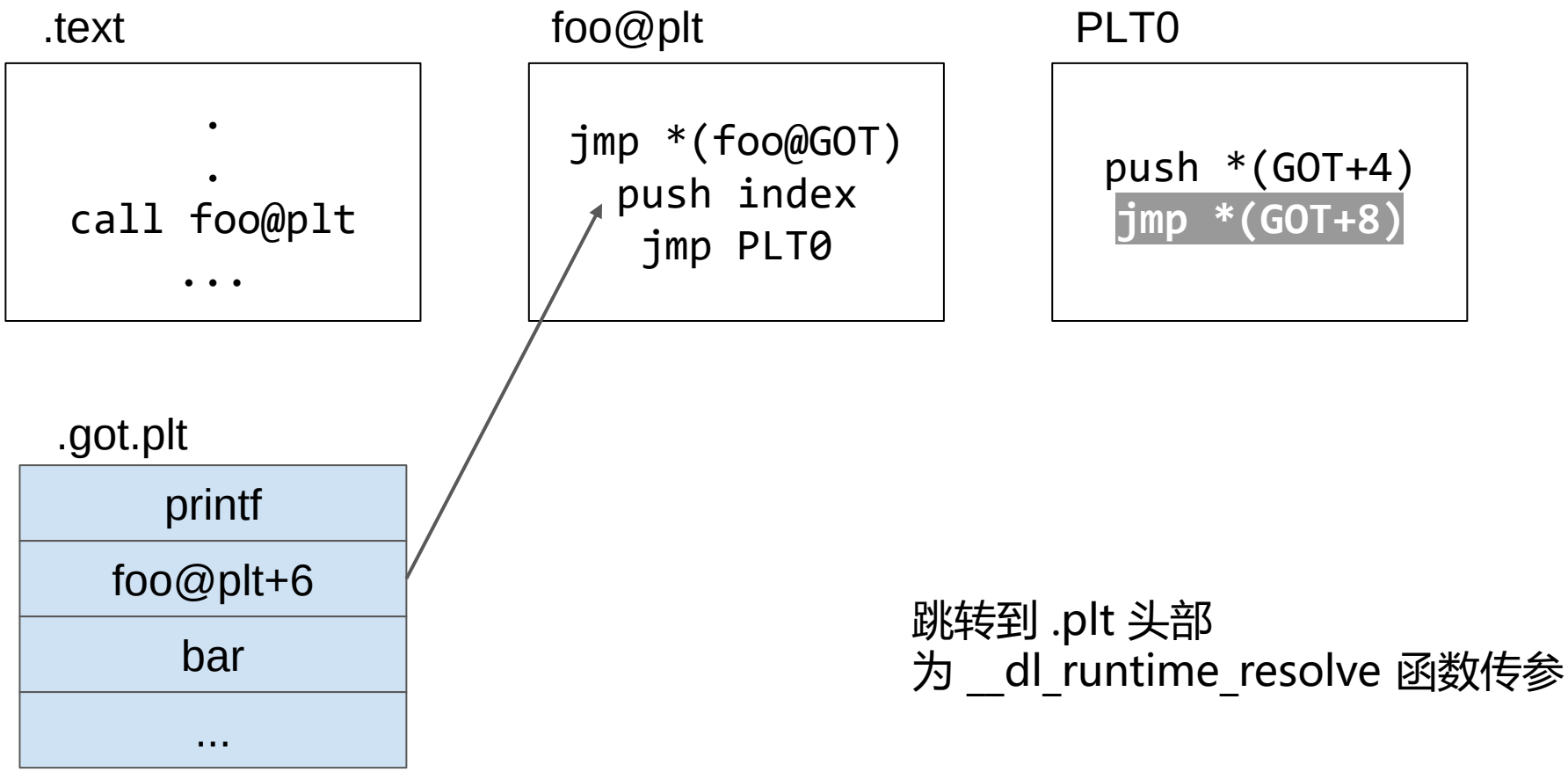
由于进程是第一次调用 foo
故 .got.plt 中记录的地址是 foo@plt+1



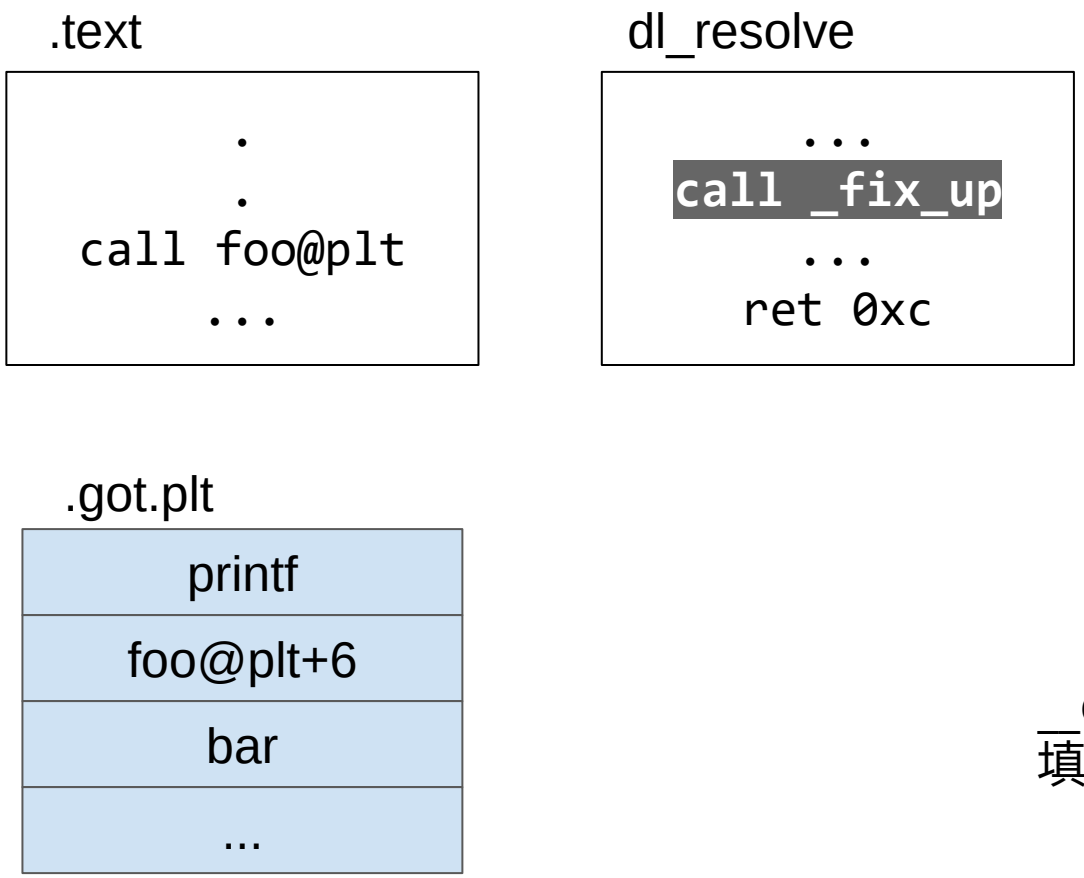
回到 .plt 是为了解析 foo 的实际地址



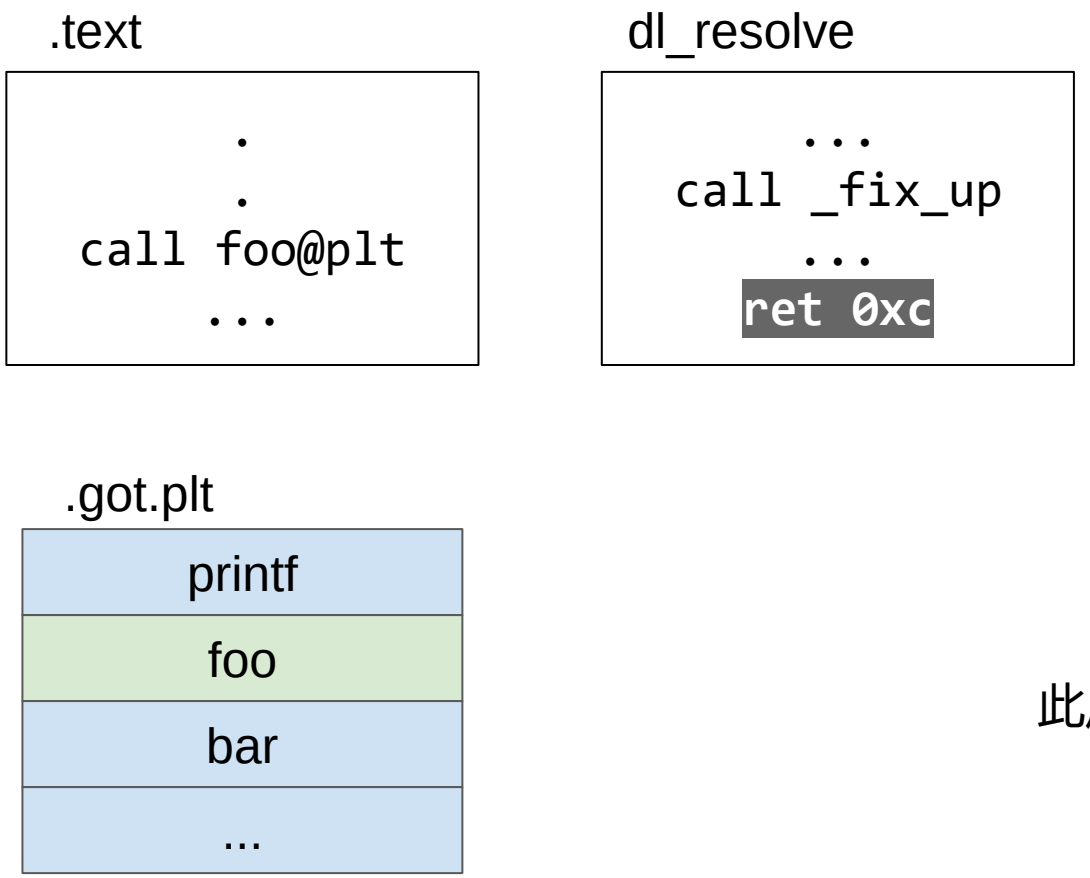
跳转到 .plt 头部
为 __dl_runtime_resolve 函数传参



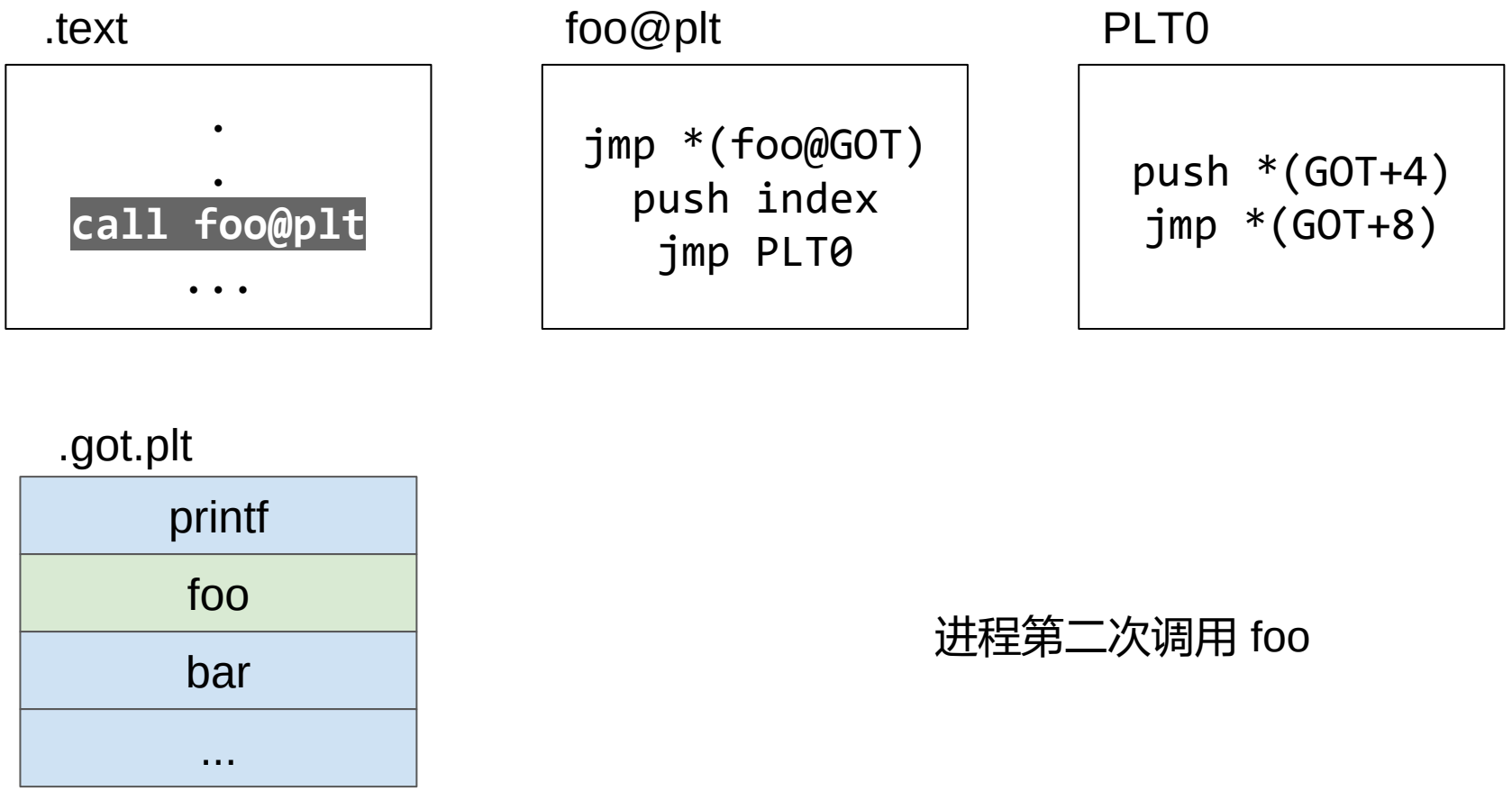
跳转到 .plt 头部
为 __dl_runtime_resolve 函数传参



__dl_runtime_resolve 函数解析 foo 的真正地址
填入 .got.plt 中



此后 .got.plt 中保存的是 foo 的真实地址


.text
.
.
call foo@plt
...
foo@plt
jmp *(foo@GOT)
push index
jmp PLT0
PLT0
push *(GOT+4)
jmp *(GOT+8)
.got.plt
printf
foo
bar
...
进程第二次调用 foo

.text

```
.
.
call foo@plt
...
```

foo@plt

```
jmp *(foo@GOT)
push index
jmp PLT0
```

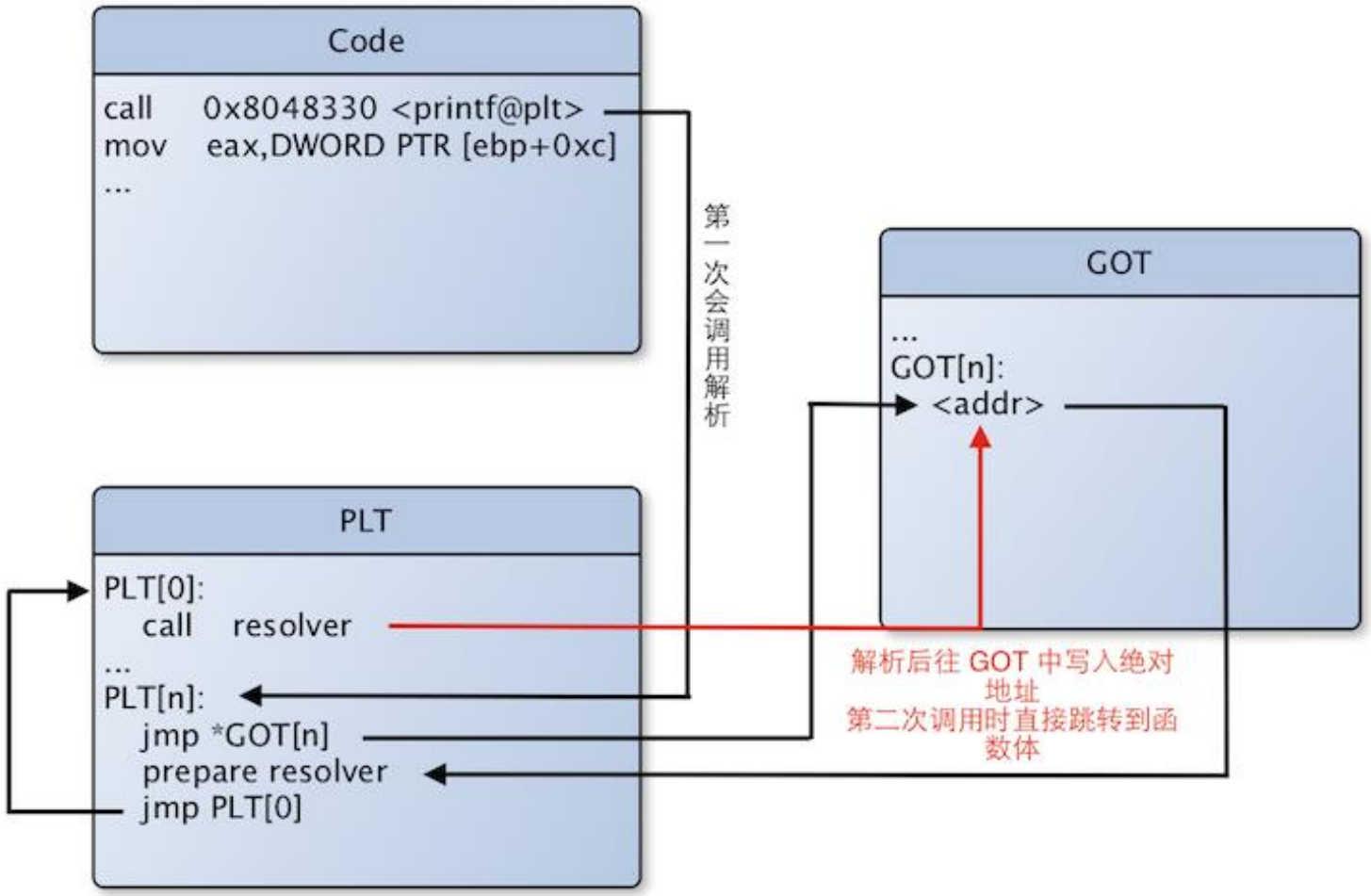
PLT0

```
push *(GOT+4)
jmp *(GOT+8)
```

.got.plt

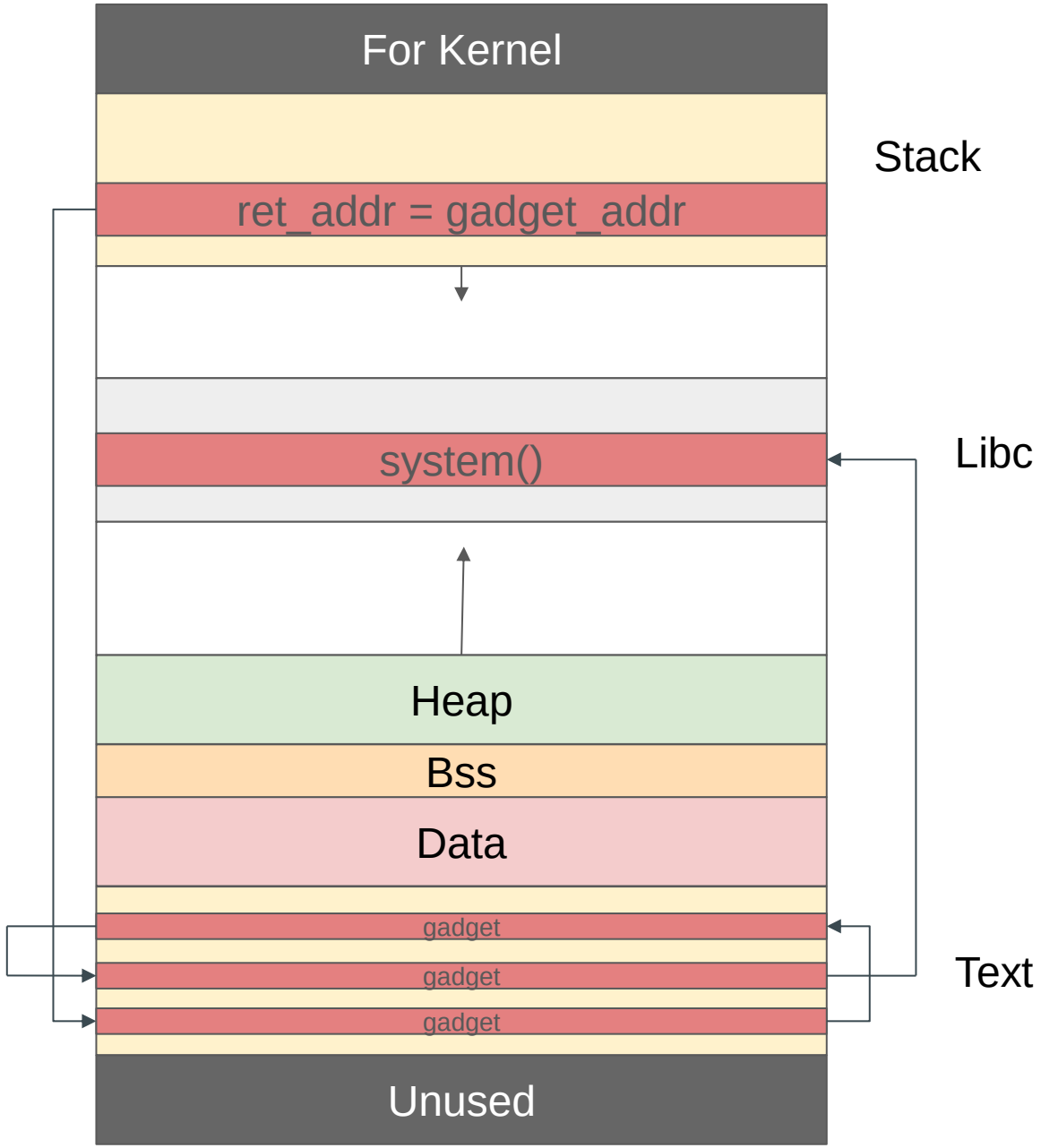
printf
foo
bar
...

直接自 .got.plt 跳转到 foo 的真实地址
没有了第一次的解析地址过程

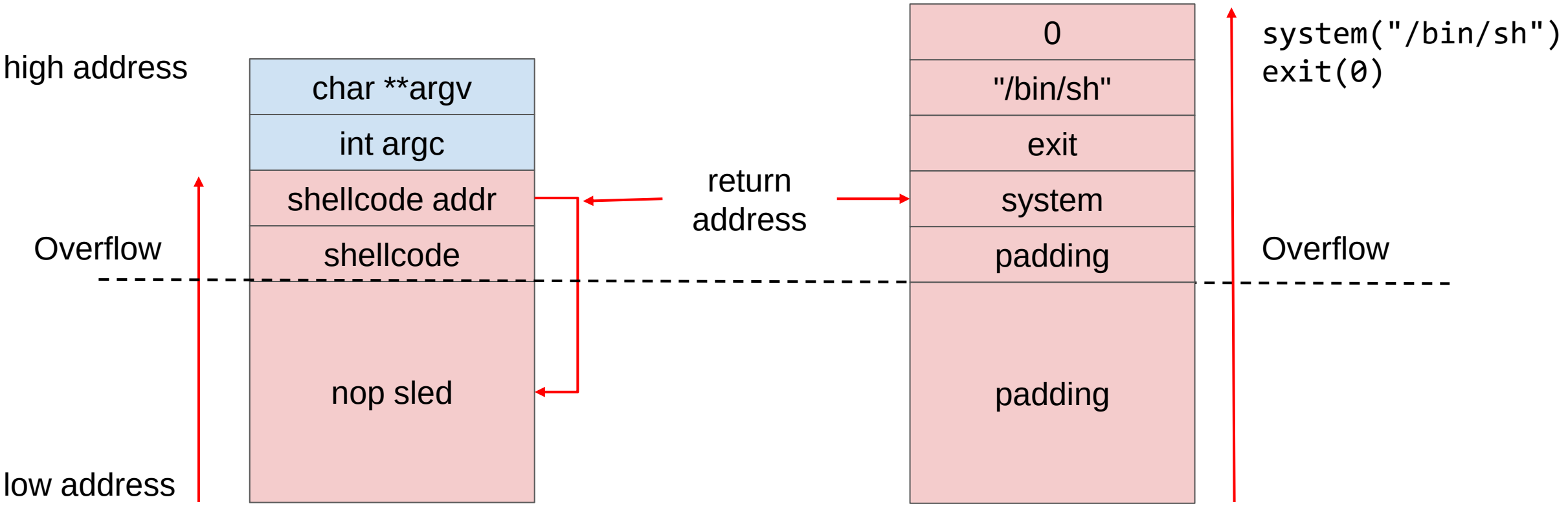

Code
call 0x8048330 <printf@plt>
mov eax,DWORD PTR [ebp+0xc]
...
第一次会调用解析
GOT
...
GOT[n]:
<addr>
PLT
PLT[0]:
call resolver
...
PLT[n]:
jmp *GOT[n]
prepare resolver
jmp PLT[0]
解析后往 GOT 中写入绝对地址
第二次调用时直接跳转到函数体

返回导向编程 | ret2libc

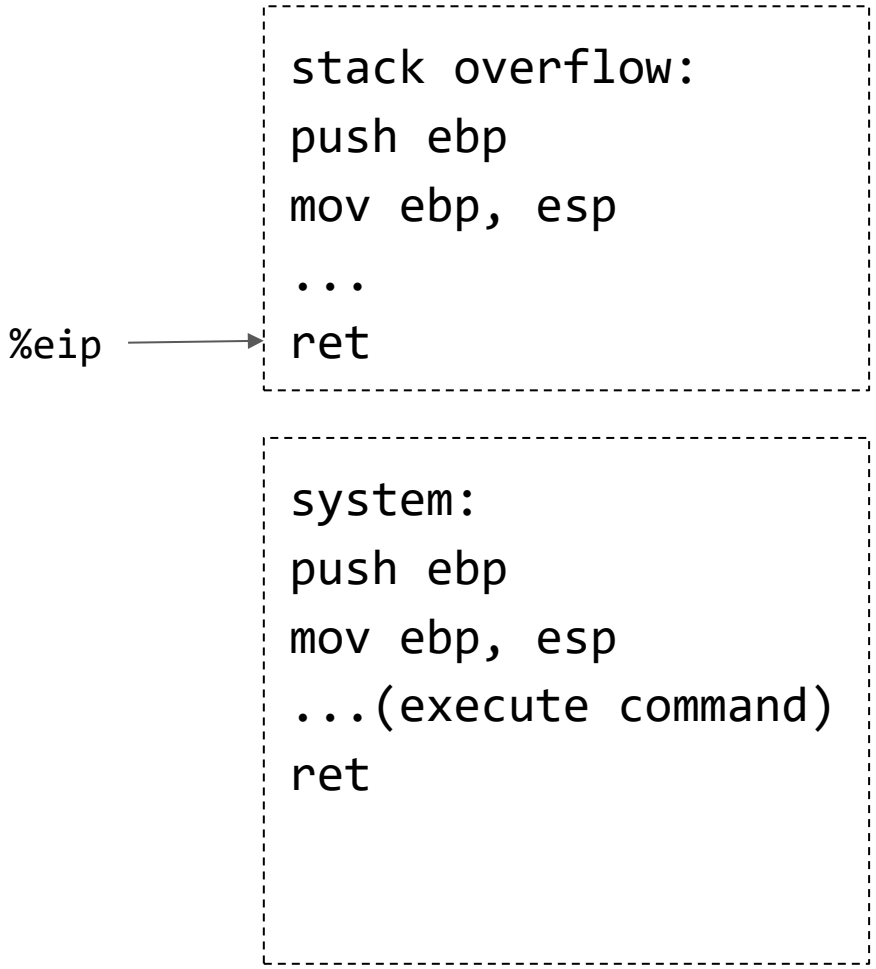
- 篡改栈帧上自返回地址开始的一段区域为一系列 gadget 的地址，最终调用 libc 中的函数获取 shell

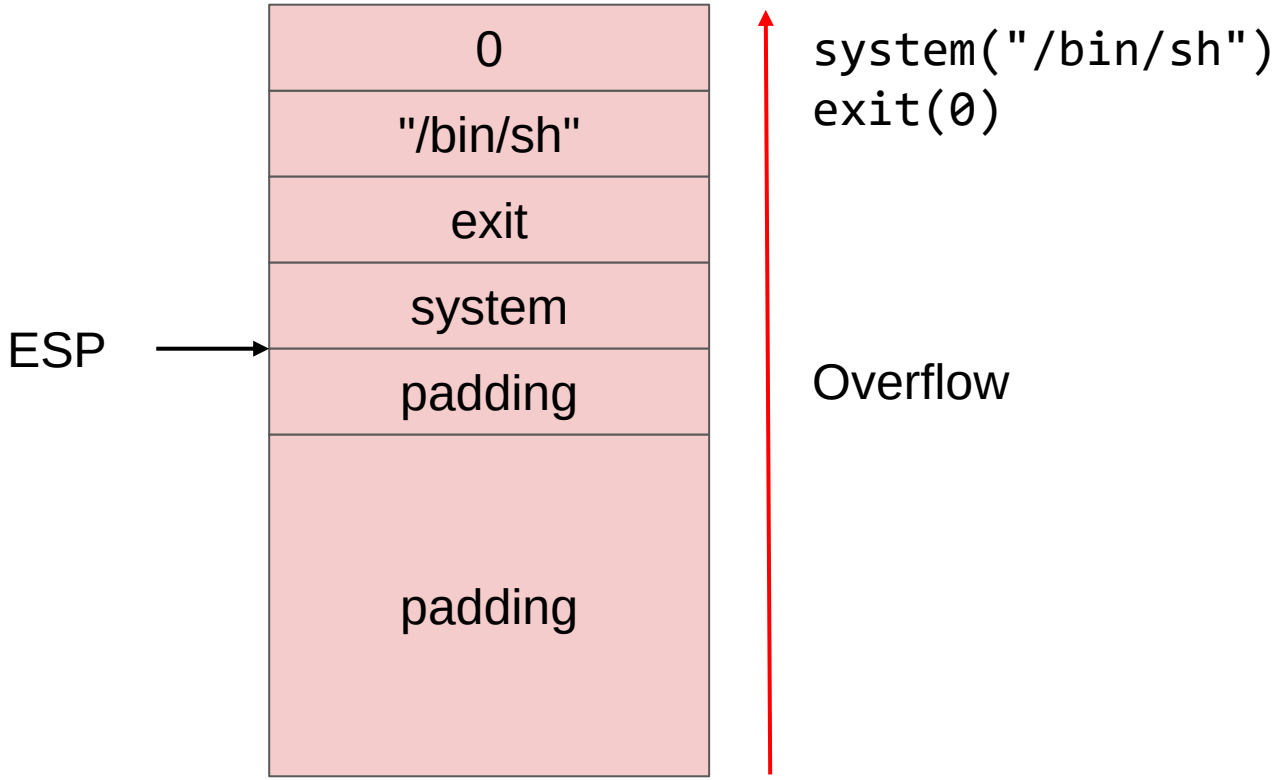


返回导向编程 | ret2libc

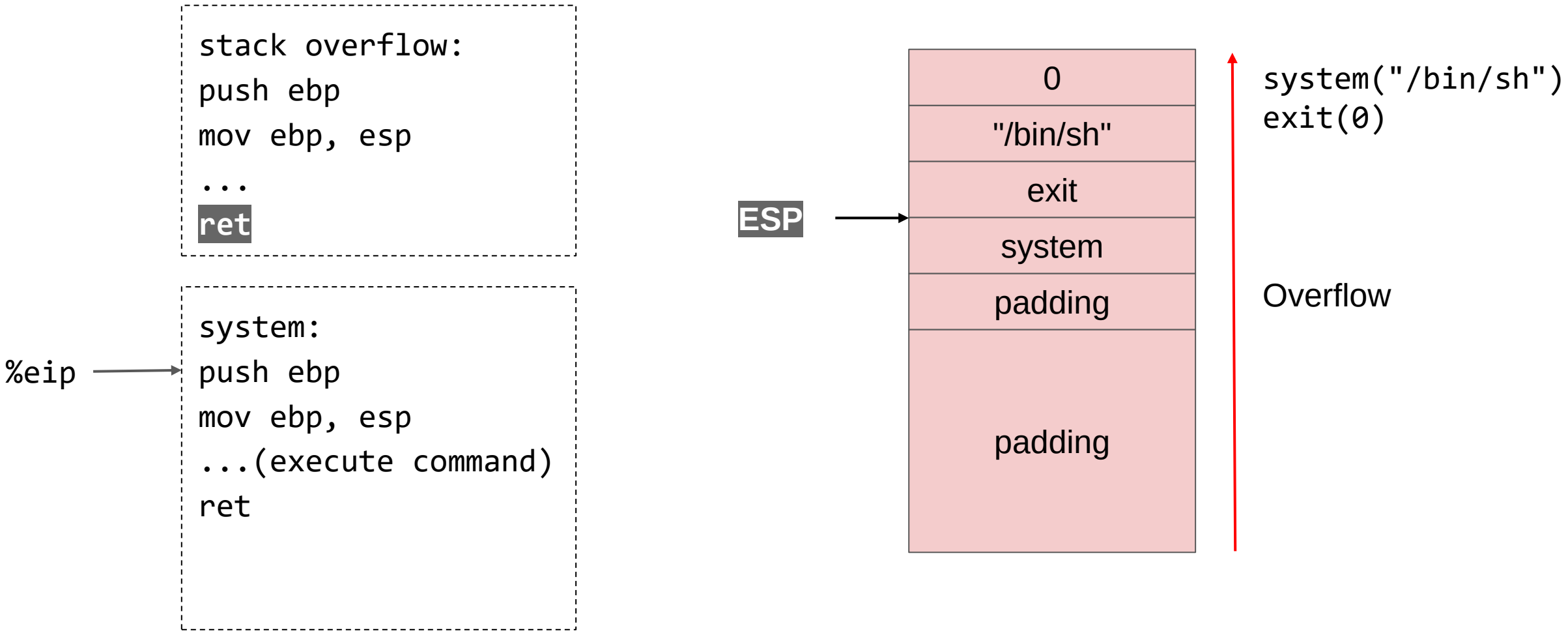


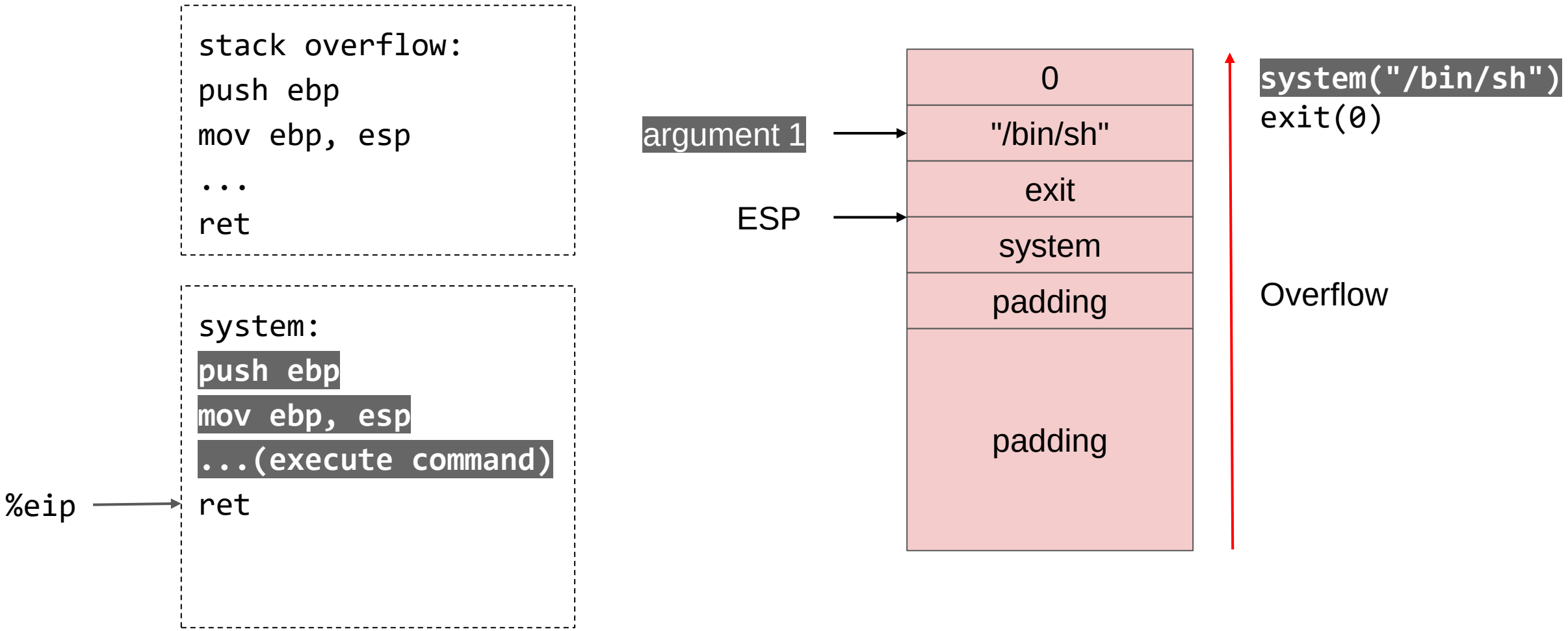
返回导向编程 | ret2libc



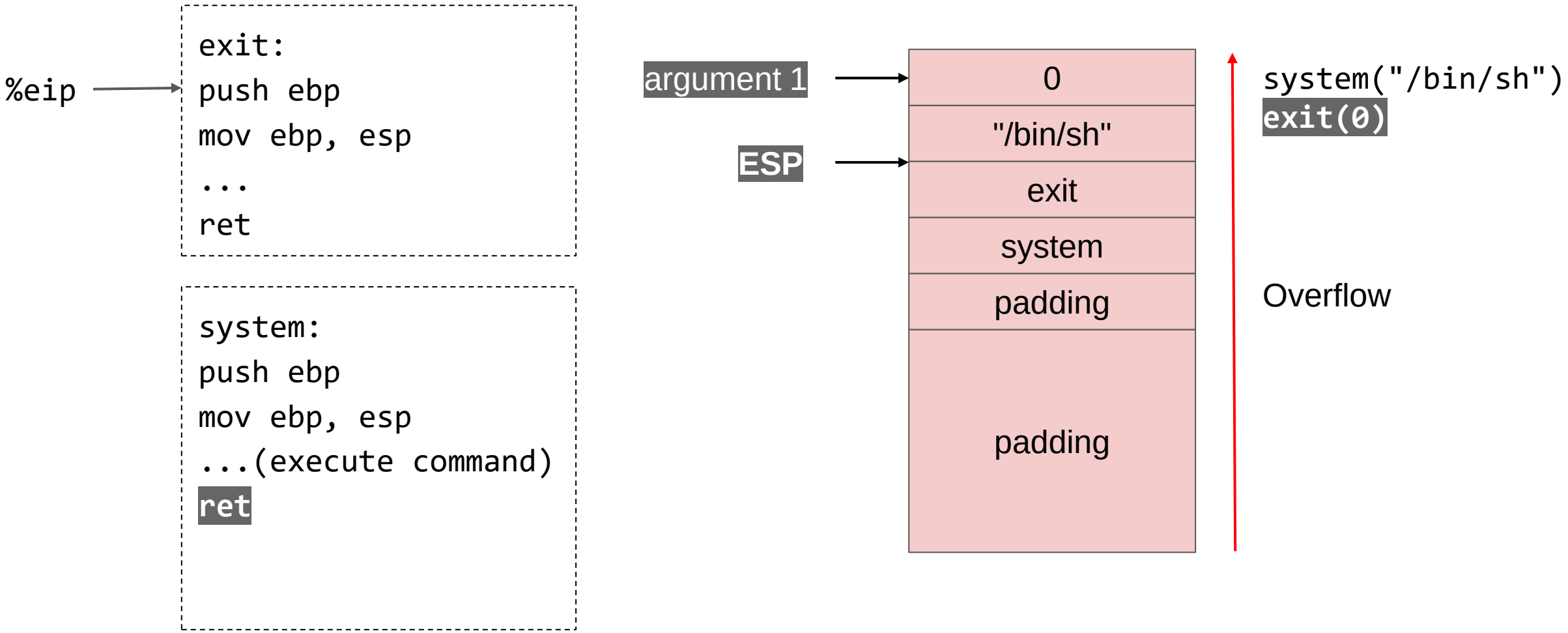


返回导向编程 | ret2libc

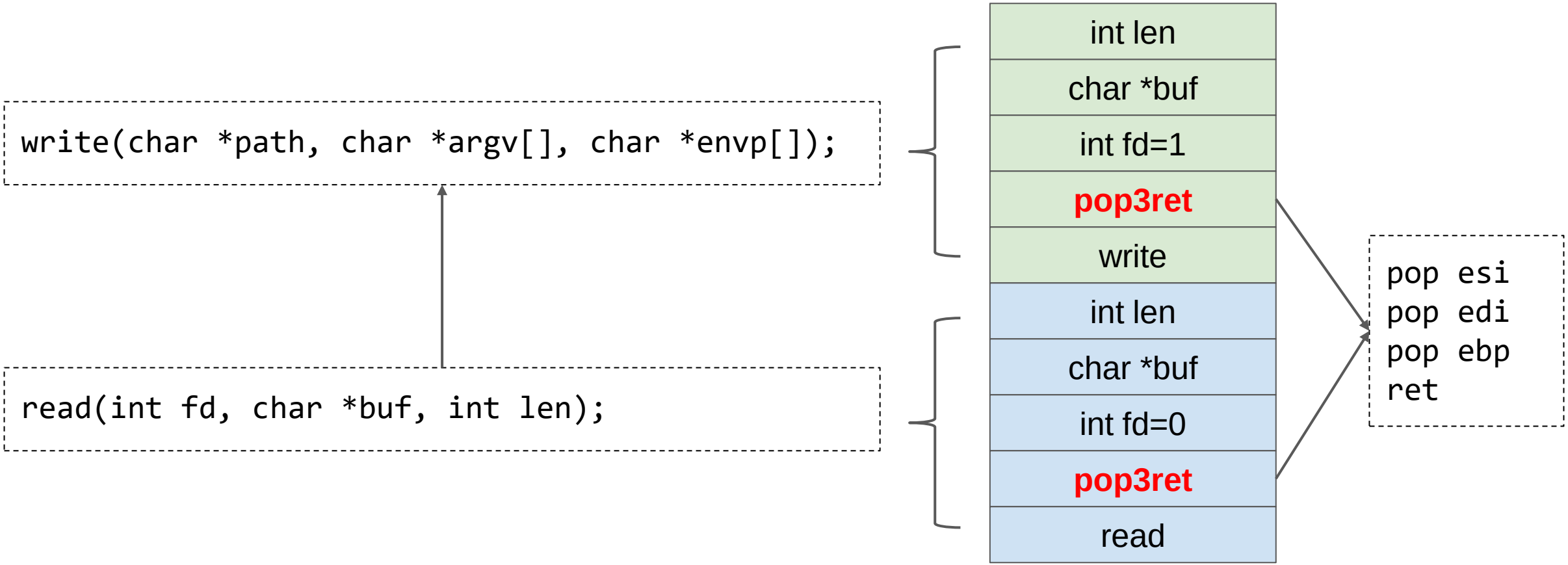



stack overflow:
push ebp
mov ebp, esp
...
ret
system:
push ebp
mov ebp, esp
...(execute command)
ret
%eip
argument 1
ESP
0
"/bin/sh"
exit
system
padding
padding
system("/bin/sh")
exit(0)
Overflow

返回导向编程 | ret2libc



ROP连续调用多个libc函数



- 到目前为止我们只接触了内存空间分布已知的ROP，但在绝大部分赛题中，由于ASLR与PIE保护的开启，在发送攻击载荷（payload）之前，往往需要我们先泄露内存分布信息，接下来来看一道题目。

栈迁移

Definition

用 gadget改变 esp 的值

Application

栈溢出长度不足以使用直接 ROP

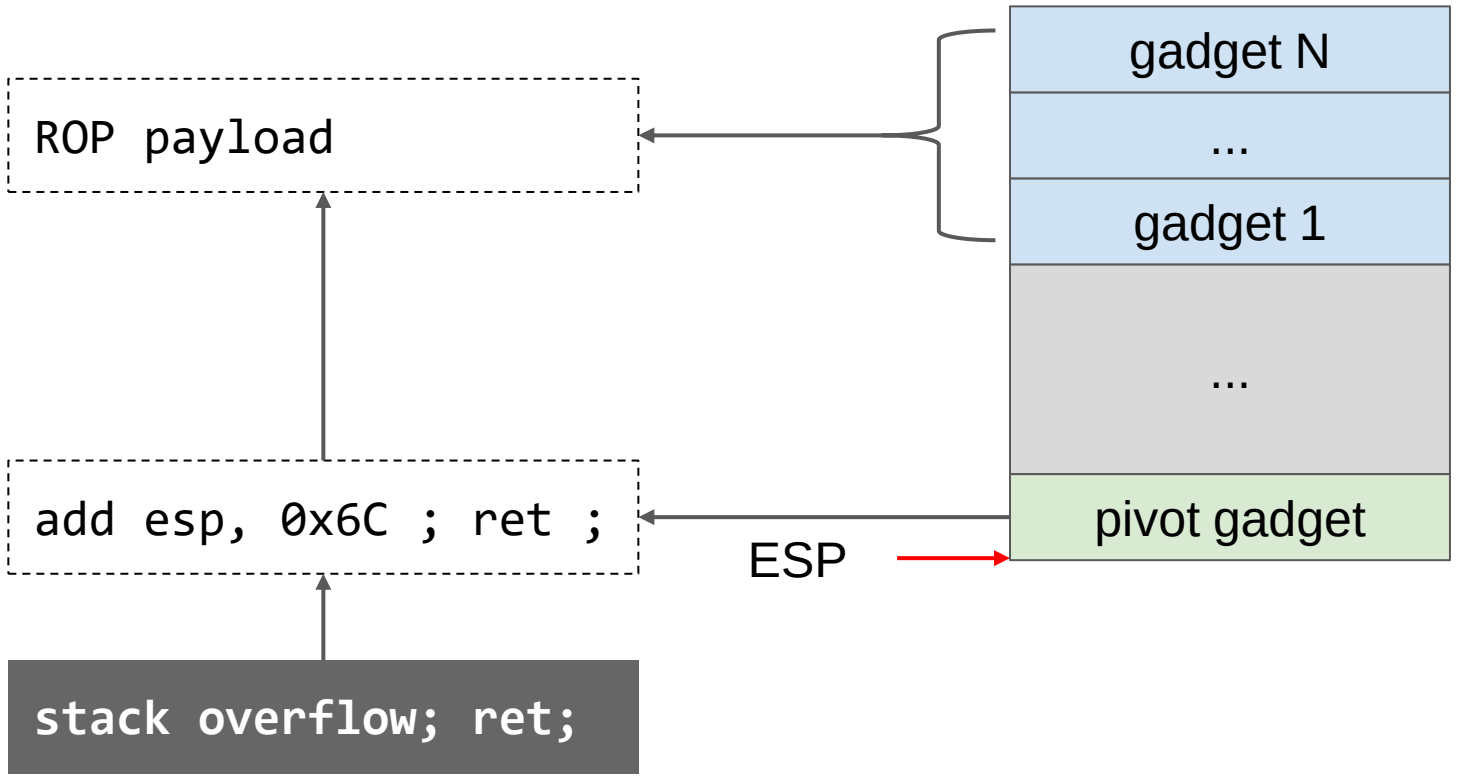
栈溢出 payload 会出现空字符截断，且gadget地址含有空字符

在泄露地址信息后需要新的 ROP payload

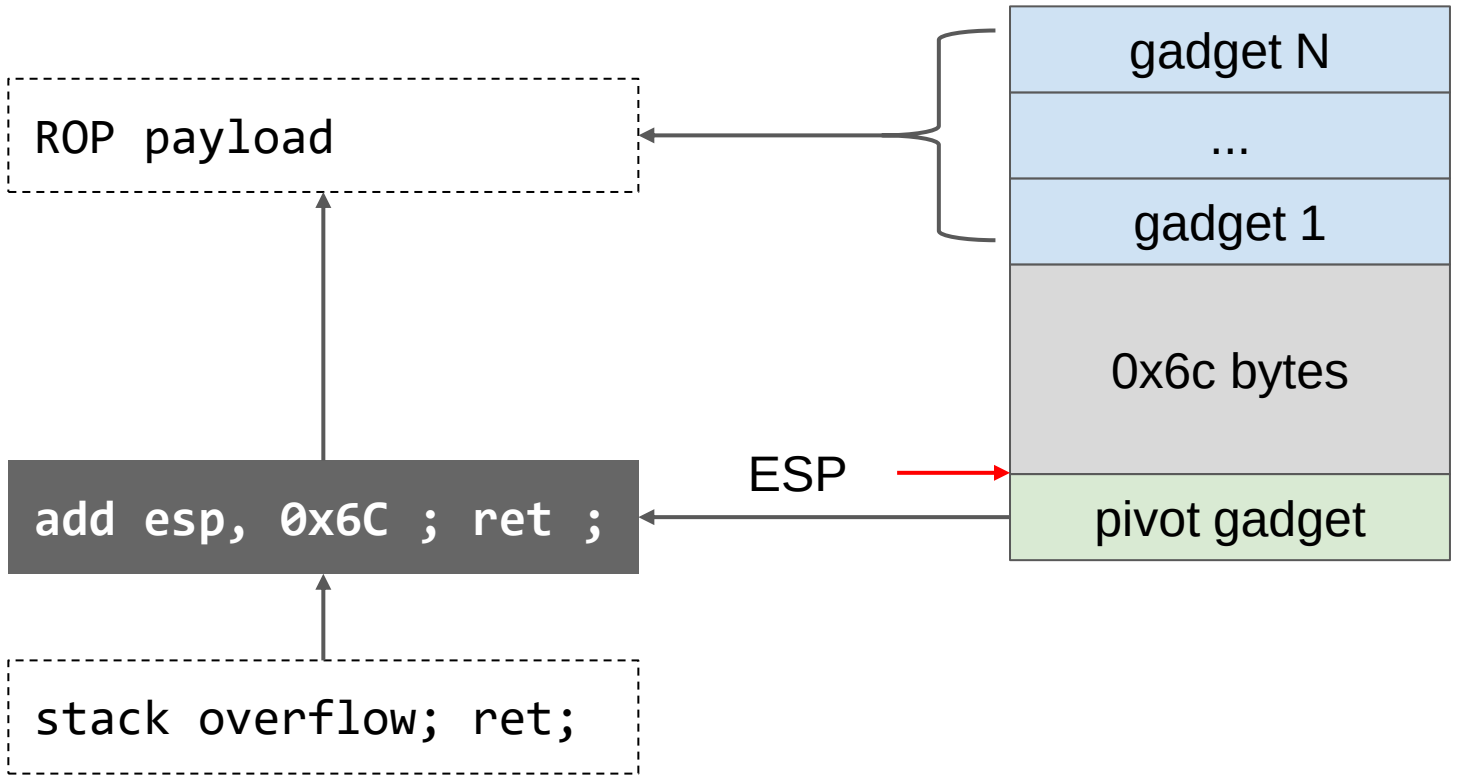
```
void stack_overflow(char *user) {
    char dst[512];
        if (strlen(user) > 536)
    return;
    // 536-512=24 bytes overflow!
    strcpy(dst, user);
}
```

```
void stack_overflow(char *user) {
    char dst[512];
    sprintf(dst, "%s", user);
}
x64 assembly:
0x406113:           55          push   %rbp
0x406114:           41 89 d4    mov
%edx,%r12d
...
```

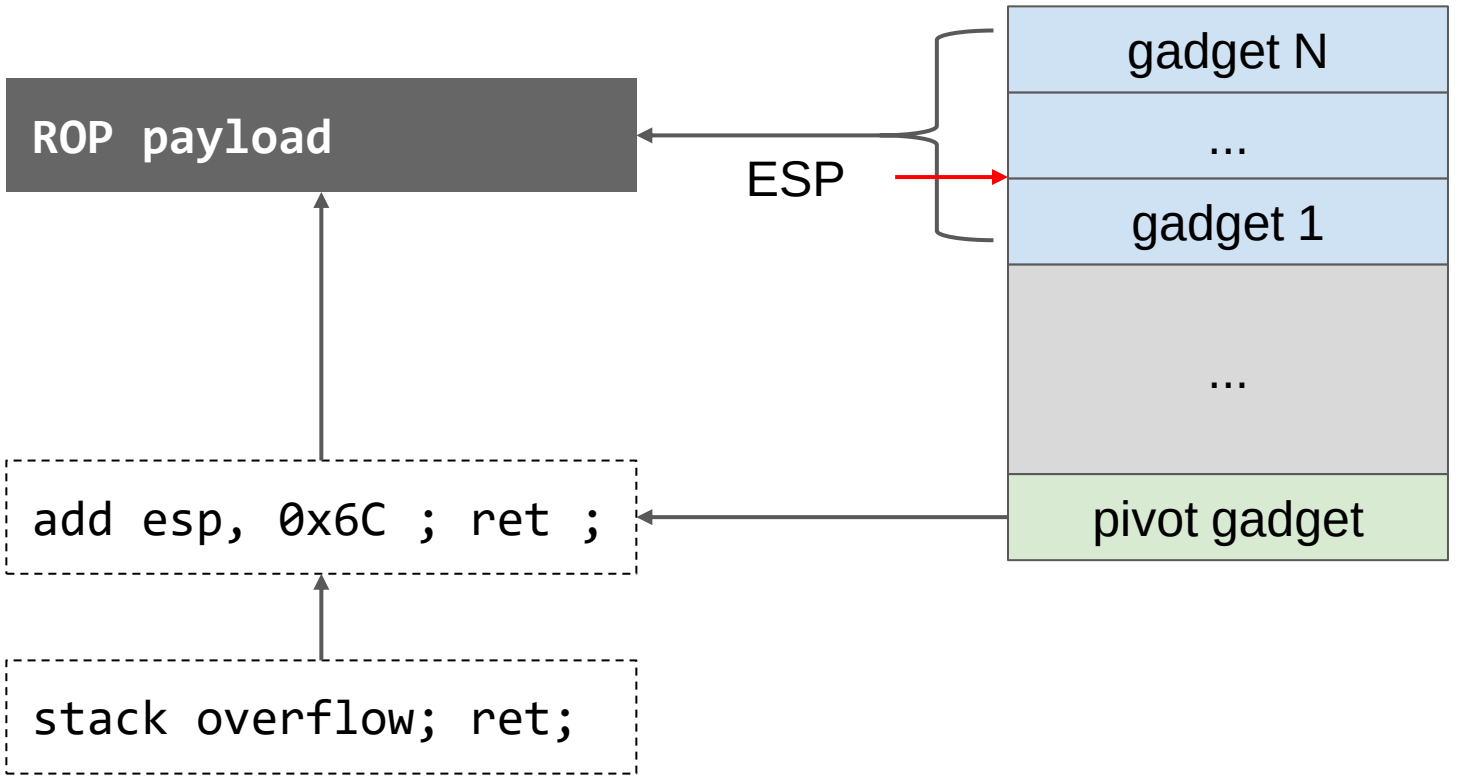
栈迁移


gadget N
...
gadget 1
...
pivot gadget
ROP payload
add esp, 0x6C ; ret ;
ESP
stack overflow; ret;

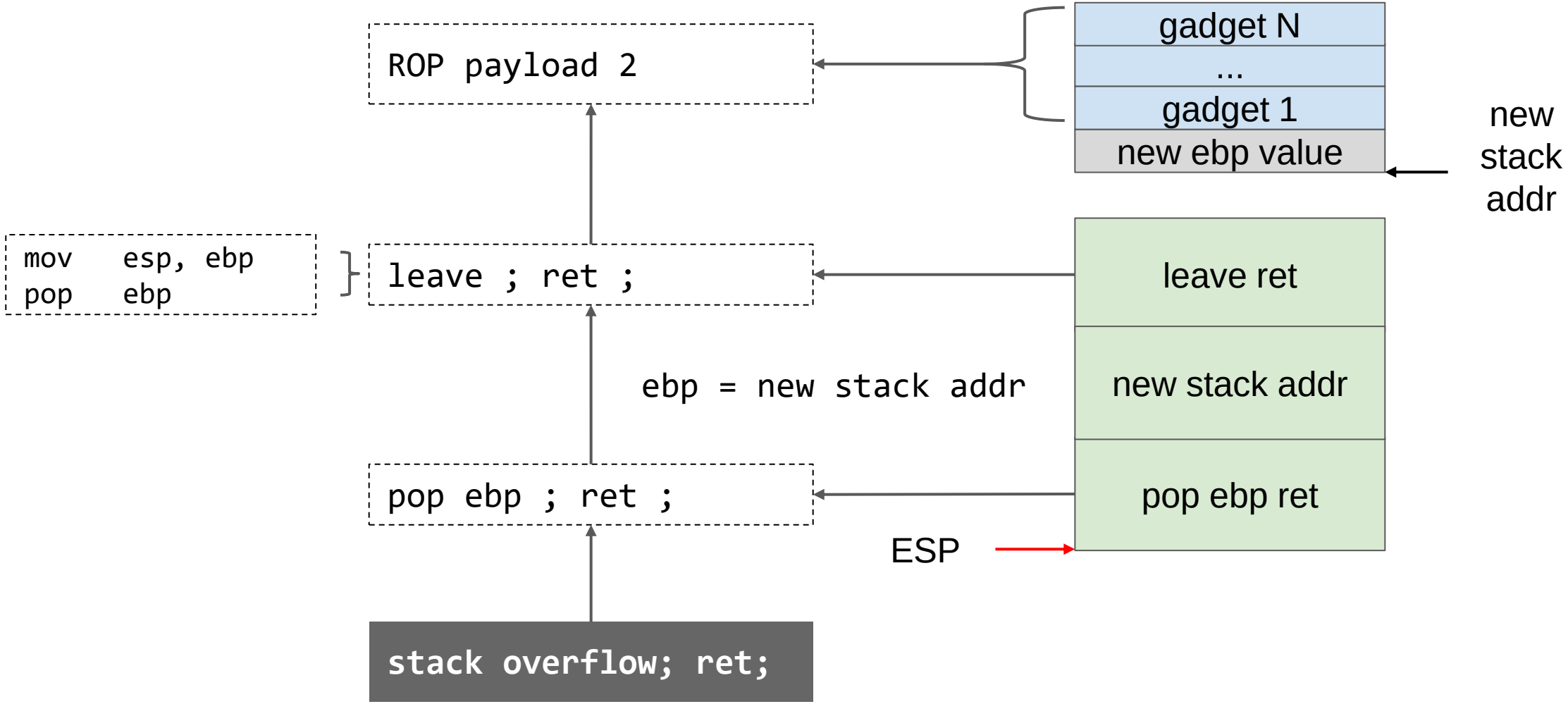
栈迁移


ROP payload
gadget N
...
gadget 1
0x6c bytes
ESP
add esp, 0x6C ; ret ;
pivot gadget
stack overflow; ret;

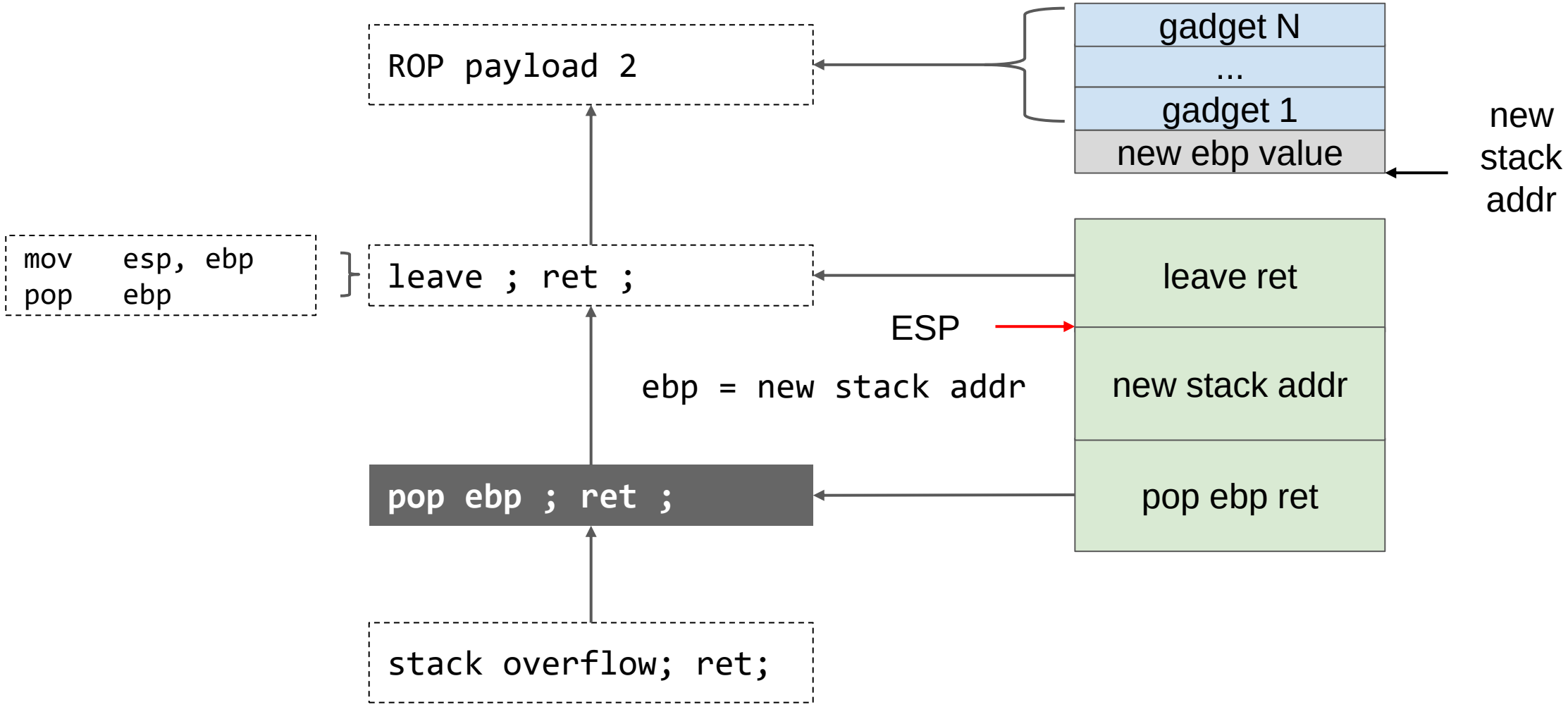
栈迁移


ROP payload
ESP
gadget N
...
gadget 1
...
pivot gadget
add esp, 0x6C ; ret ;
stack overflow; ret;

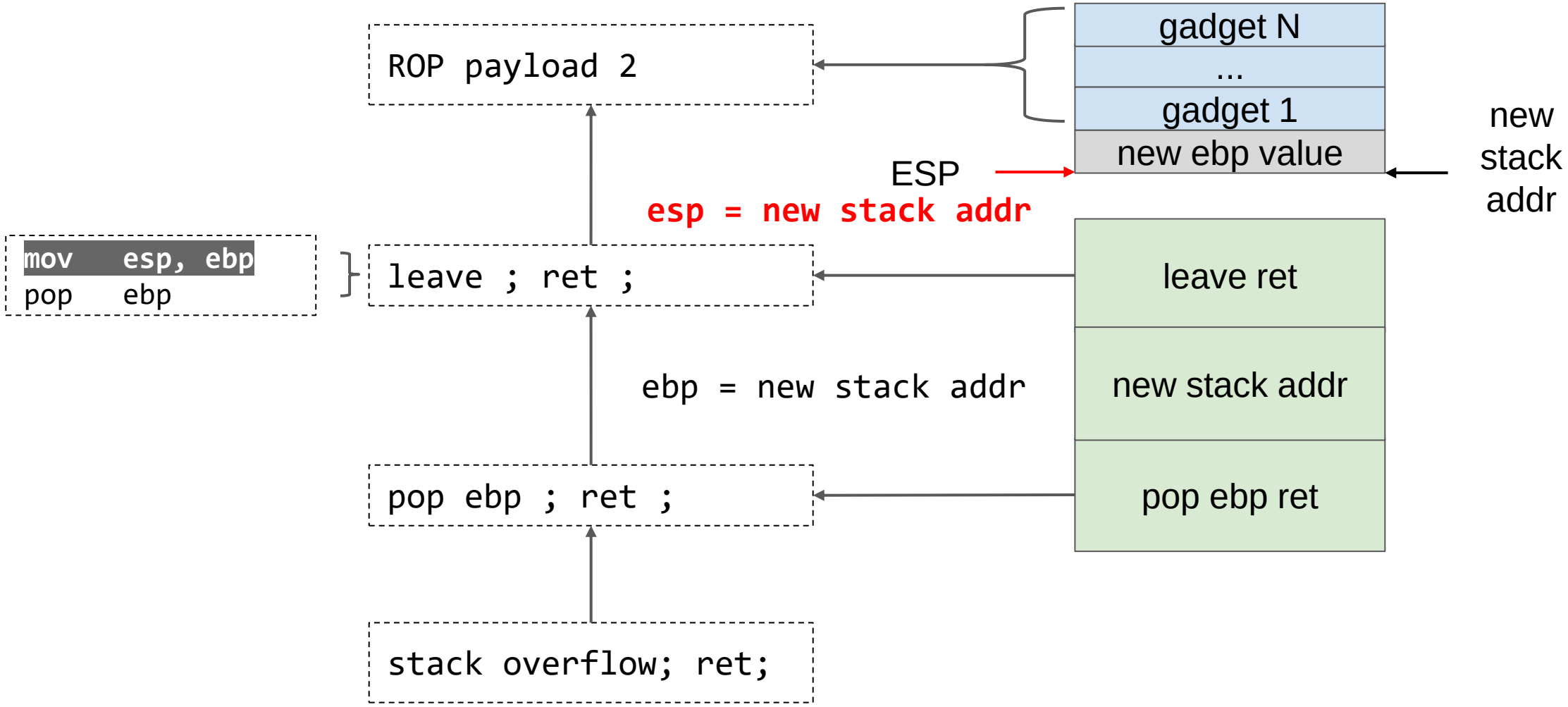
栈迁移: "pop ebp ret" + "leave ret"



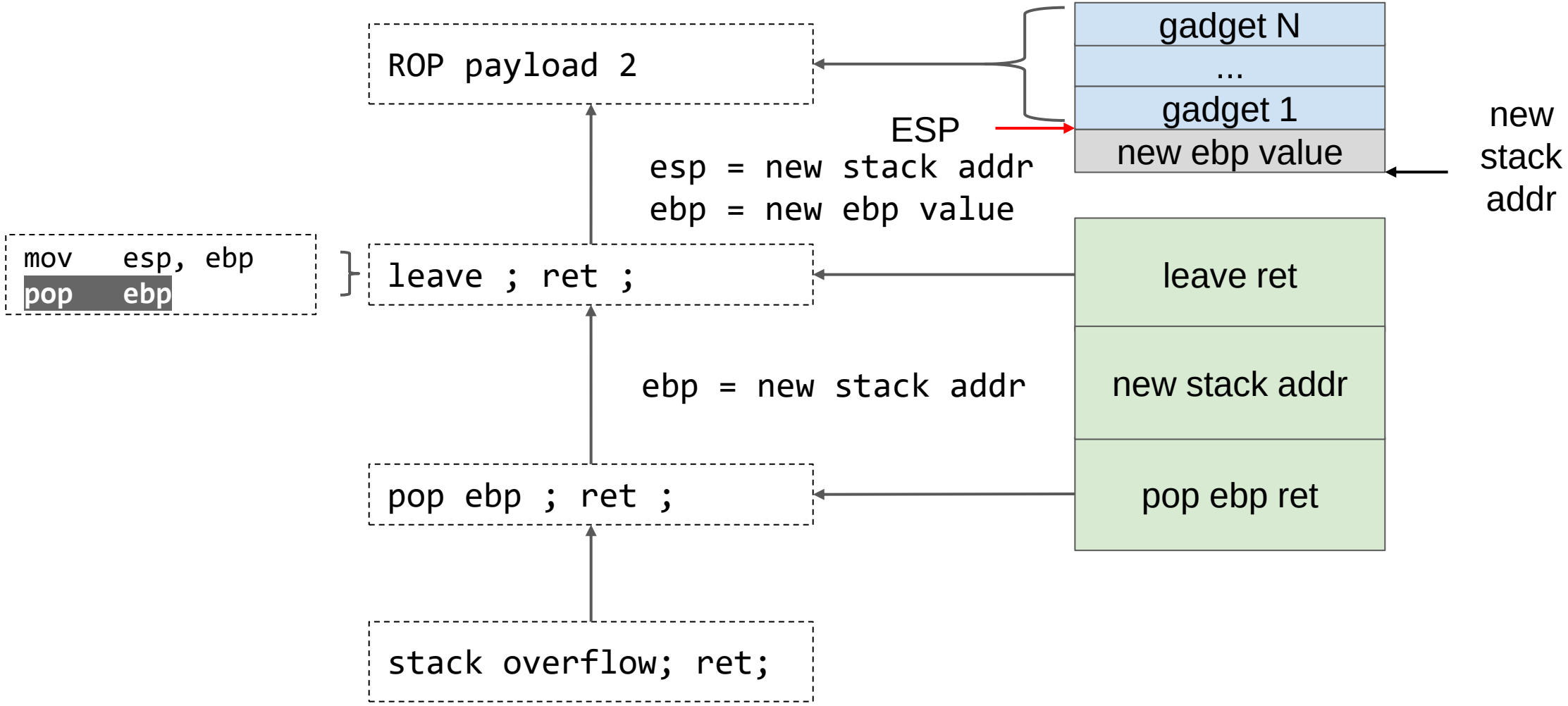
栈迁移: "pop ebp ret" + "leave ret"



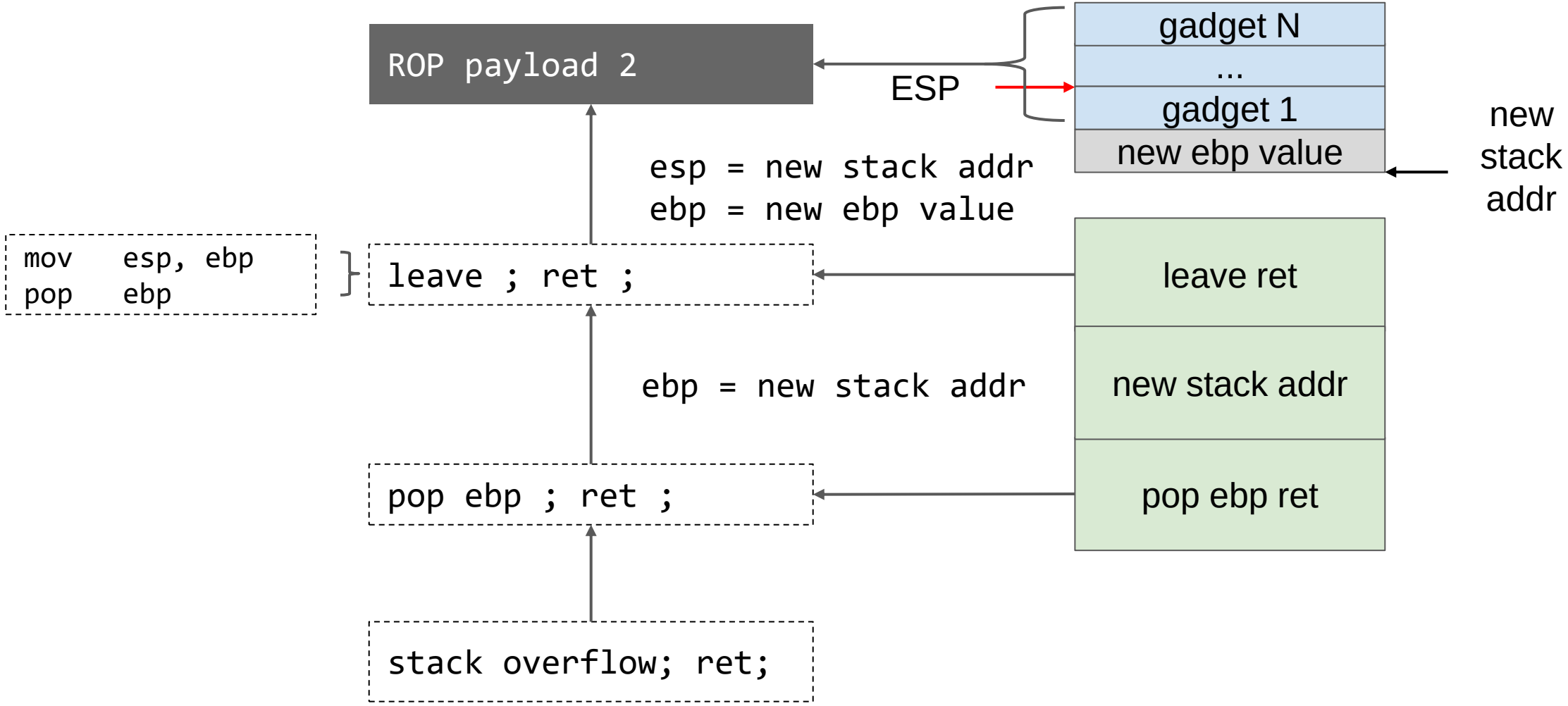
栈迁移: "pop ebp ret" + "leave ret"



栈迁移: "pop ebp ret" + "leave ret"



栈迁移: "pop ebp ret" + "leave ret"



x64 ROP

使用 gadget 传递参数

pop rdi ; ret

pop rsi ; pop r15 ; ret ;

setting rdx and rcx is hard, no trivial gadgets

ret2csu

__libc_csu_init Gadgets for x64

```
.text:0000000000400690 loc_400690:                             ; CODE XREF: __libc_csu_init+54 j
.text:0000000000400690                 mov     rdx, r13
.text:0000000000400693                 mov     rsi, r14
.text:0000000000400696                 mov     edi, r15d
.text:0000000000400699                 call    qword ptr [r12+rbx*8]
.text:000000000040069D                 add     rbx, 1
.text:00000000004006A1                 cmp     rbx, rbp
.text:00000000004006A4                 jnz     short loc_400690
.text:00000000004006A6
.text:00000000004006A6 loc_4006A6:                             ; CODE XREF: __libc_csu_init+34 j
.text:00000000004006A6                 add     rsp, 8
.text:00000000004006AA                 pop     rbx
.text:00000000004006AB                 pop     rbp
.text:00000000004006AC                 pop     r12
.text:00000000004006AE                 pop     r13
.text:00000000004006B0                 pop     r14
.text:00000000004006B2                 pop     r15
.text:00000000004006B4                 retn
.text:00000000004006B4 __libc_csu_init endp
```

Part4 格式化字符串漏洞

- 泄露栈内存
- 泄露任意地址内存
- 篡改栈内存
- 篡改任意地址内存

Integer Overflow

History
European Space Agency spacecraft Cluster I's satellites exploded because of integer overflow (1996)
M. Dowd, C. Spencer, N. Metha, N. Herath, and H. Flake. Advanced Software Vulnerability Assessment.
In Blackhat USA, August 2002
IO2BO
Array index out of bounds

IO2BO

```
void safe_memcpy(char *src, int size) {
    char dst[512];
    if (size < 512) {
        memcpy(dst, src, size);
    }
}
```

Array index OOB

```
  void safe_set_element(char *arr, int
index, char value, int arr_size) {
      if (index < arr_size) {
          arr[index] = value;
      }
  }
```

Integer Overflow in Linux kernel

CVE-2013-1763 Linux Kernel netlink message family number overflow

```
net/core/sock_diag.c View file @ 6e601a5
@@ -121,6 +121,9 @@ static int
__sock_diag_rcv_msg(struct sk_buff *skb, struct
nlmsghdr *nlh)
    if (nlmsg_len(nlh) < sizeof(*req))
      return -EINVAL;
+   if (req->sdiag_family >= AF_MAX)
+     return -EINVAL;
    hndl = sock_diag_lock_handler(req->sdiag_family);
    if (hndl == NULL)
            err = -ENOENT;
      else
 err = hndl->dump(skb, nlh);
...
```

```
static const inline struct
sock_diag_handler
*sock_diag_lock_handler(int family)
{
            if (sock_diag_handlers[family]
== NULL)
                        request_module("net-
pf-%d-proto-%d-type-%d", PF_NETLINK,

NETLINK_SOCK_DIAG, family);


            mutex_lock(&sock_diag_table_mute
x);
            return
sock_diag_handlers[family];
}
```

Integer Overflow Quiz

- Integer Overflow vulnerability can be regarded as some kind of logic bug.
- Integer overflow itself cannot be exploited, it is usually used to trigger buffer overflow/heap overflow or used with ROP, ret2libc, etc.

Quiz: can you find the vuln?

```
int test(short param, int value) {
    int * mybuf = (int *)malloc(65536 * sizeof(int));
    if (param < 0) {
        param = -param;
    }
    mybuf[param] = value;
    return mybuf;
}
```

Format String

History
Discovered and public since June 1999
Shocked the security community in the second half of 2000(influenced lots of programs)
Format String Attacks, Newsham (2001)
use %x and %n to achieve arbitrary read and write

Format String Vuln Discovered(year2000)

Application	*Found by*	*Impact*	*years*
wu-ftpd 2.*	security.is	remote root	> 6
Linux rpc.statd	security.is	remote root	> 4
IRIX telnetd	LSD	remote root	> 8
Qualcomm Popper 2.53	security.is	remote user	> 3
Apache + PHP3	security.is	remote user	> 2
NLS / locale	CORE SDI	local root	?
screen	Jouko Pynnōnen	local root	> 5
BSD chpass	TESO	local root	?
OpenBSD fstat	ktwo	local root	?

Format String related Functions

- fprintf — prints to a FILE stream
- printf — prints to the ‘stdout’ stream
- sprintf — prints into a string
- 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
- setproctitle — set argv[]
- syslog — output to the syslog facility

Format String syntax

-%s print null terminated string pointed by arg pointer
-%x print hex value of the arg (arbitrary read)
-%n store the bytes_written to location pointed by arg pointer
-%hn - store 2 bytes (short integer)
-%hhn - store 1 byte
-%<positive integer>c print arbitrary count of chars controlled by integer
-Example: %123c (filled by space) %0512c (filled by 0)
-Useful to change bytes_written so far
-%<positive integer>$<fmt> specify arg index
-Example: %12$n

Format String Example

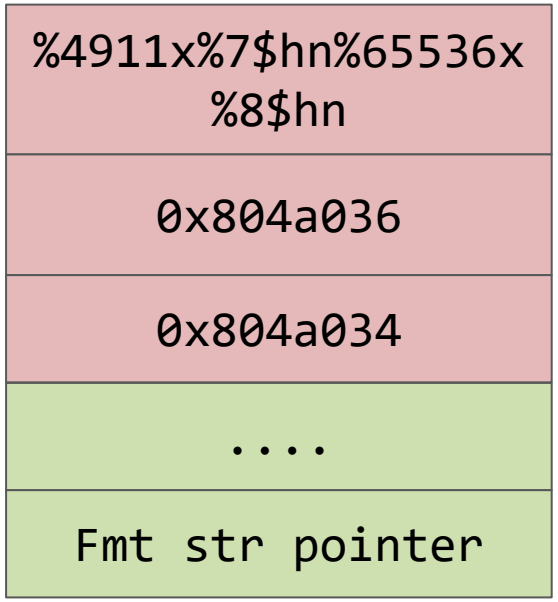
```
 1 #include <stdio.h>
 2 #include <string.h>
 3
 4 int flag = 0;
 5
 6 int main(int argc, char **argv) {
 7         char buf[1024];
 8         if (argc < 2) return 1;
 9         strncpy(buf, argv[1], min(sizeof(buf) - 1, 1024));
10         printf(buf);
11         printf("\n");
12
13         if (flag == 0x13371337) {                    // How to win
14         printf("You Win!\n");
15         }
16
17         return 0;
18 }
```

Format String Example

```
$ ./fmt %x:%x:%x:%x:%x:%x
ffffda9c:3ff:1b1ea4:253a7825:78253a78:3a78253a

$ ./fmt %s:%s:%s:%s:%s:%s
Segmentation fault (core dumped)
```

Format String Example

High Addr



Low Addr

```
 1 #include <stdio.h>
 2 #include <string.h>
 3
 4 int flag = 0;
 5
 6 int main(int argc, char **argv) {
 7         char buf[1024];
 8         if (argc < 2) return 1;
 9         strncpy(buf, argv[1], min(sizeof(buf) - 1, 1024));
10         printf(buf);
11         printf("\n");
12
13         if (flag == 0x13371337) {                    // How
to win
14         printf("You Win!\n");
15         }
16
17         return 0;
18 }
```

Format String Example

```
./fmt $(python2 -c 'print
"\x34\xa0\x04\x08\x36\xa0\x04\x08%4911x%7$hn%65536x%8$hn"')
```

How to execute
arbitrary code?

```
...

You Win!
```

Format String Code Exec

- ret address on stack
- .got.plt
- .dtors

readelf -s a.out |grep DTOR

```
    29: 08049518     0 OBJECT  LOCAL  DEFAULT   19 __DTOR_LIST__
    56: 0804951c     0 OBJECT  GLOBAL HIDDEN   19 __DTOR_END__
```

- __atexit
- ...you name it

Format String Usage

1.Read / write any position if map permission is allowed
2.GOT hijacking
3.Write variable value
4.Leak libc base address and calculate offset to get another function address
5.Leak libc version
6.Leak stack address

Format String Strategy

1.Leak Info
a.Use %x %p to leak stack info
b.To leak arbitrary addr info
i.If buf on stack, write addr in buf
ii.If buf not on stack, write addr pointer on stack first, then use another printf to leak
2.Write Data
a.If dest addr not known, leak addr first(mostly stack)
b.If buf on stack, use addr in buf (Libformatstr can help)
c.If buf not on stack
i.Common way: use ebp->ebp pair (or sth similar) to change arbitrary addr
ii.If using snprintf, then write addr first using %n (suitable for 0x0804xxx)

Format String tricks and gotchas

-Q: How to calculate positional param?

- A: Libc said that the behavior is undefined if positional param position is larger than provided non-positional param count. But for most case, the addr should be `word_size x position .` So it's more easier to open your gdb and use pattern string to find the actual position instead of calculating. Besides, %0$x is not recognised.

- Q: What if the second %n needs a small value to be written while bytes_written just goes larger?
- A: Just use integer overflow. (For example, if %hn applies, `second_padding=(second_size+65536-first_padding) % 65536 )`

Format String tricks and gotchas

- Q: When can I use %n and when shouldn't?
- A: For printf, it'll be too large to print out in your console(or socket if socat/xinetd used). But for snprintf, you can use %n with not too large values. (0x0804xxxx can be considered not too large) snprintf won't overflow, but will calculate the actual value and return actual bytes_written.

- Q: Can I modify more than 1 values on stack within a single format string?
- A: Definitely you can.But if you want the second value addr to be based on the first modified value, there are some tricks here: The first %n must not be positional param. That's because In libc implementation, the positional values are copied into an internal struct in function `printf_positional` before applying fmt transformation.

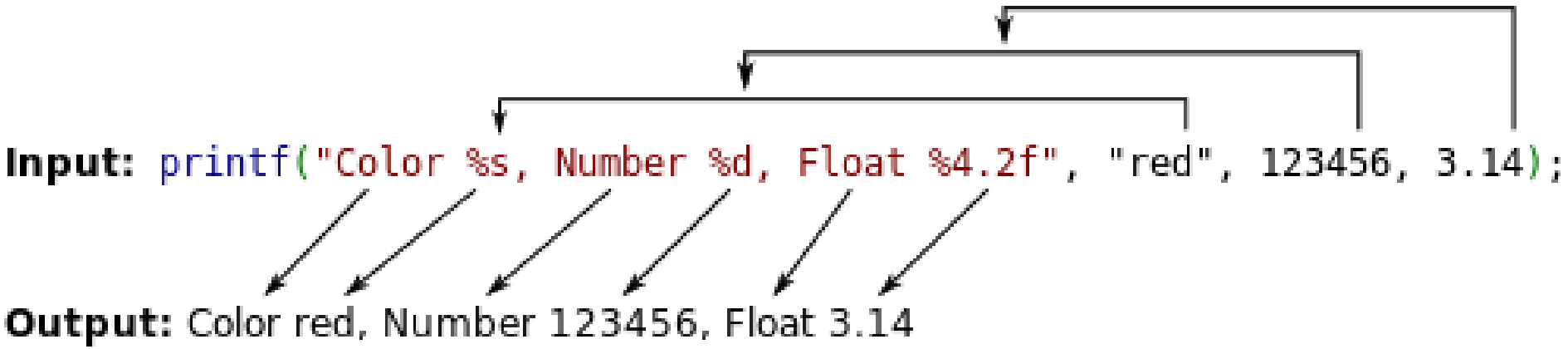
Format String Challenge

```
// simplified version of SCTF 2016 pwn200
static s[0x400];
static buf[0x400];

void run_system() { system("/bin/sh"); }

void main() {
        for (int i = 0; i < 2; i++) {
            fgets(buf, 1024, stdin);
            snprintf(s, 1024, buf);
            puts(s);
            fflush(stdout);
        }
}
```

Can you pwn using different solutions?

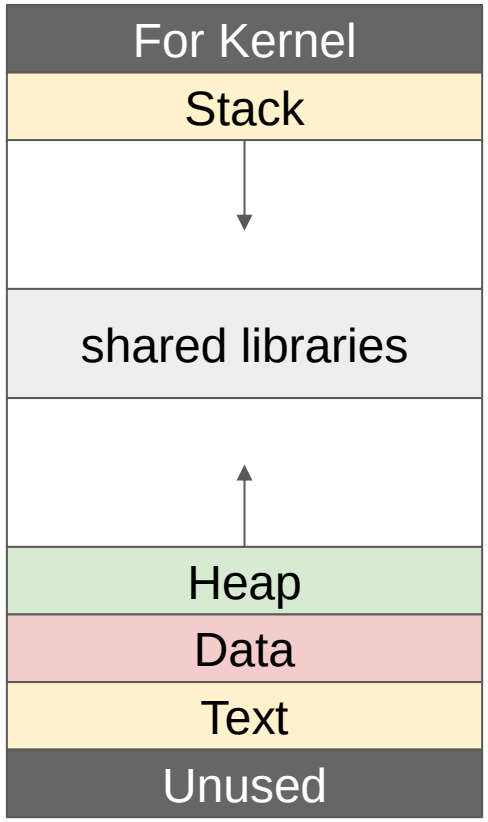
Format String Tools and Ref

-https://github.com/hellman/libformatstr
-https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf
-http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf


Input: printf("Color %s, Number %d, Float %4.2f", "red", 123456, 3.14);
Output: Color red, Number 123456, Float 3.14

Part5 堆利用

- 堆管理器
  - 堆概述
  - Areana
  - Chunk
  - Bin
- 堆分配策略
  - Malloc
  - free
- 堆漏洞与其利用



什么是堆?

- 是虚拟地址空间的一块连续的线性区域
- 提供动态分配的内存，允许程序申请大小未知的内存
- 在用户与操作系统之间，作为动态内存管理的中间人
- 响应用户的申请内存请求， 向操作系统申请内存，然后将其返回给用户程序
- 管理用户所释放的内存，适 时归还给操作系统

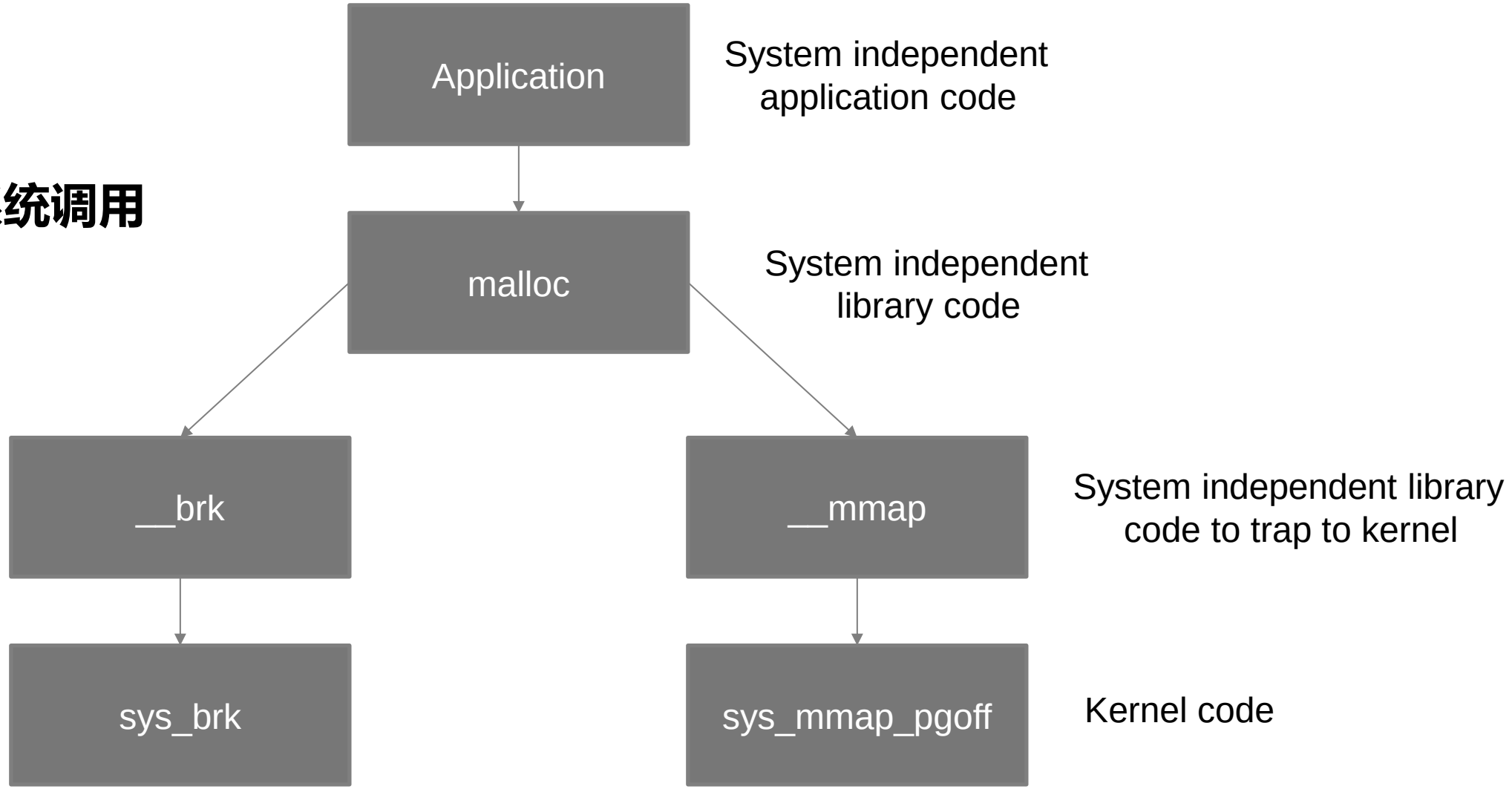
各种堆管理器

- dlmalloc – General purpose allocator
- ptmalloc2 – glibc （敲黑板）
- jemalloc – FreeBSD and Firefox
- tcmalloc – Google libumem – Solaris

堆管理器并非由操作系统实现，而是由libc.so.6链接库实现。 封装了一些系统调用，为用户提供方便的动态内存分配接口的同时，力求高效地管理由系统调用申请来的内存。

申请内存的系统调用

- brk
- mmap

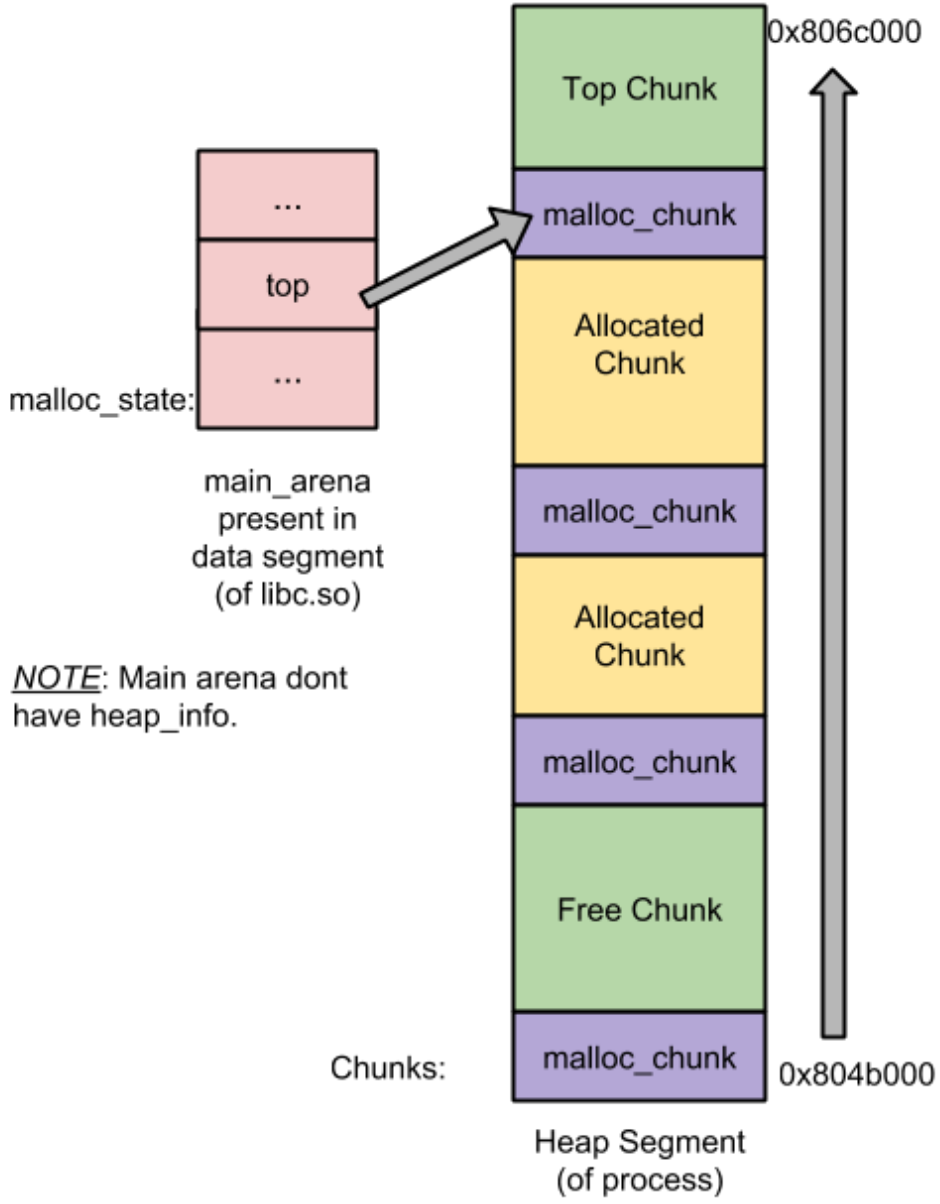


arena

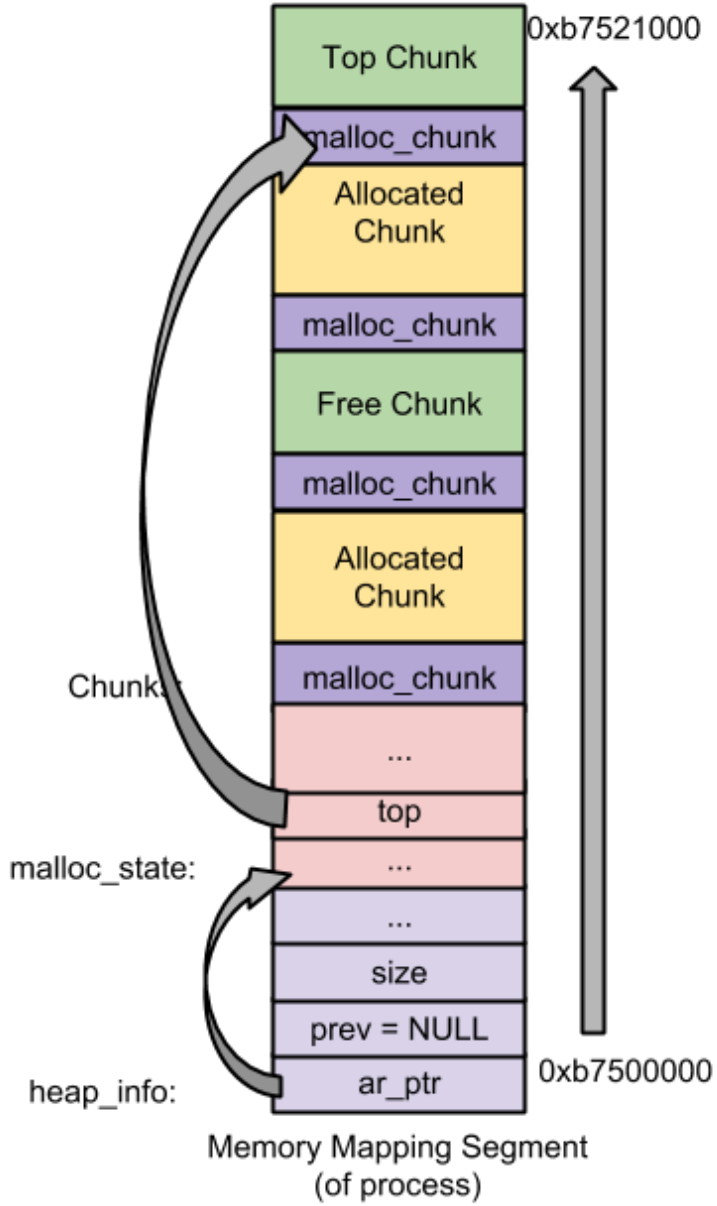
内存分配区，可以理解为堆管理器所持有的内存池

```
操作系统 --> 堆管理器 --> 用户
物理内存 --> a r e n a  --> 可用内存
```

堆管理器与用户的内存交易发生于arena中，可以理解为堆管理器向操作系统批发来的有冗余的内存库存



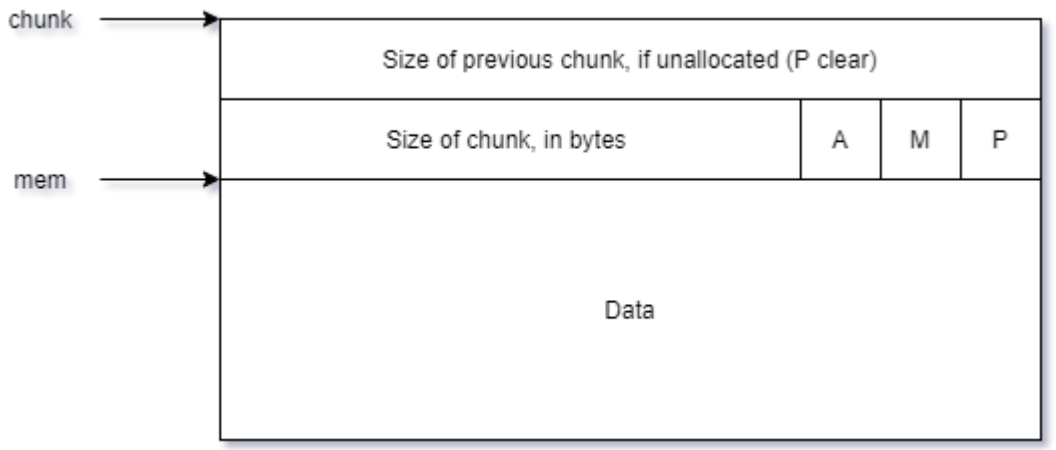
Main Arena



Thread Arena

chunk

用户申请内存的单位，也是堆管理器管理内存的基本单位
malloc()返回的指针指向一个chunk的数据区域



chunk 的具体实现

```
struct malloc_chunk {

  INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

chunk 的分类

按状态

- malloced
- free

按大小

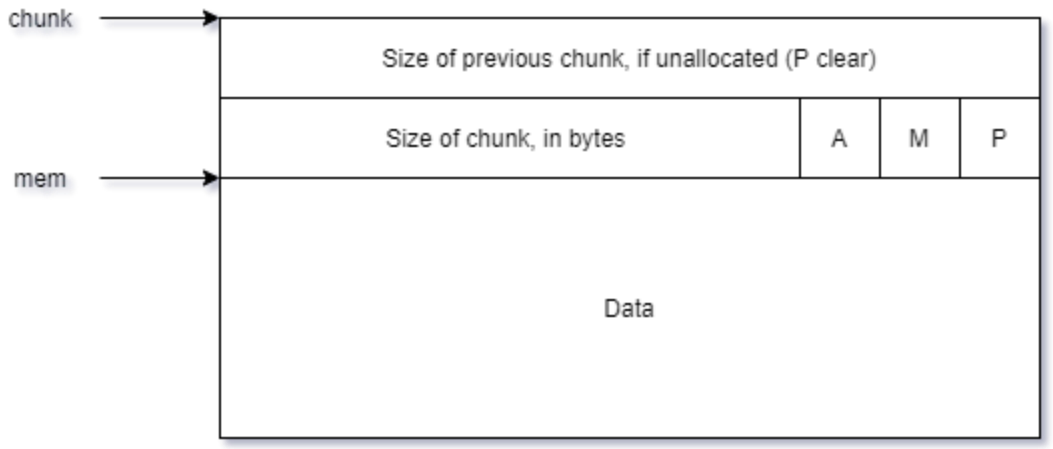
- fast
- small
- targe
- tcache

按特定功能

- top chunk
- last remainder chunk

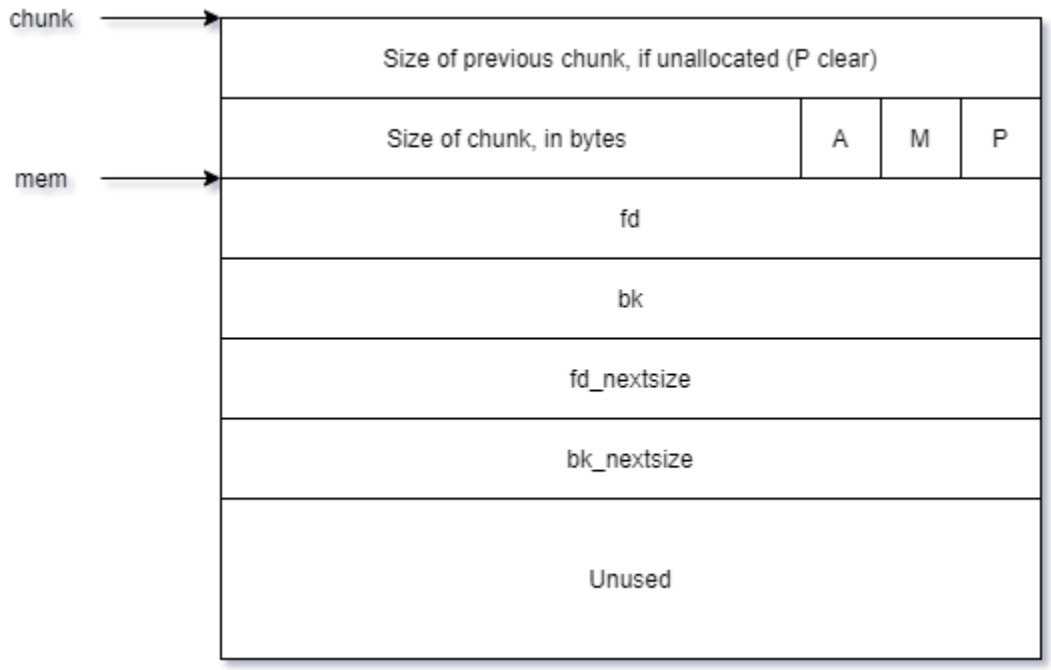
malloced chunk

已被分配且填写了相应数据的chunk



free chunk

被释放掉的malloced chunk
成为free chunk

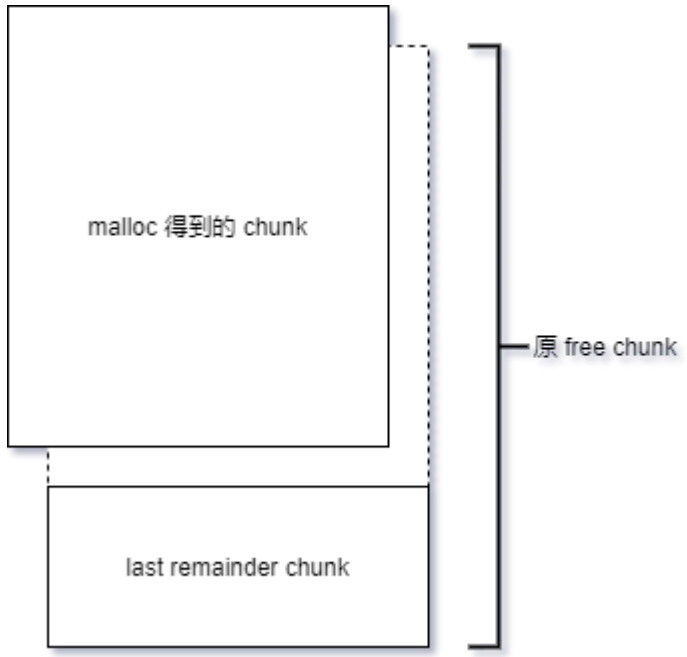


top chunk

arena中从未被使用过的内存区域

last remainder chunk

malloc分割原chunk后剩余的部分



Chunk 的微观结构

- prev_size — 仅当前一个chunk为free chunk时生效
- size
  - size
  - A
  - M
  - P

  占据size域的低3bits
- fd
- bk — 仅为处于双向链表bin中的free chunk时生效
- fd_nextsize
- bk_nextsize

  仅为large free chunk时生效

仅为free chunk时生效

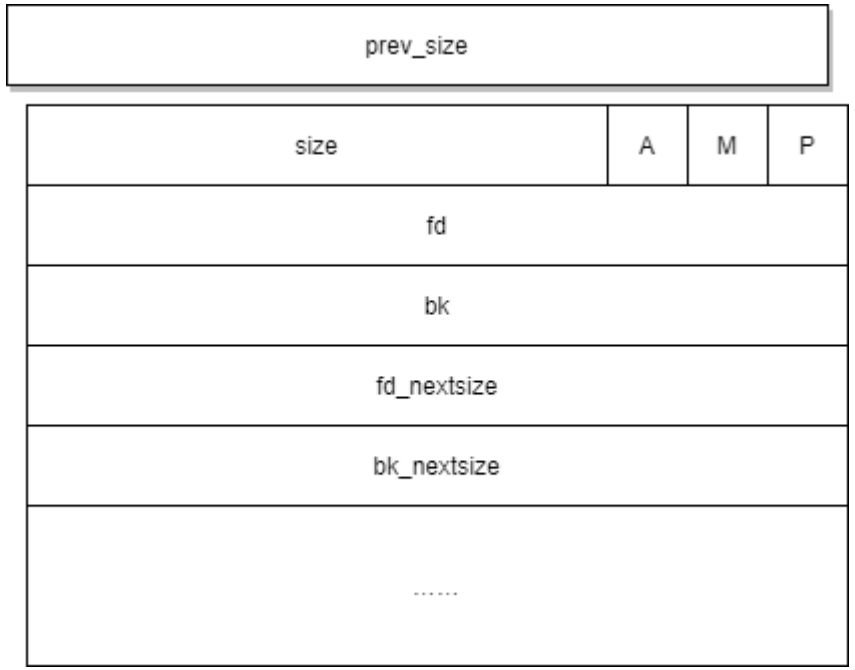
bin

管理 arena 中空闲 chunk 的结构，以数组的形式存在，数组元素为相应大小的 chunk 链表的链表头，存在于 arena 的 malloc_state 中

- unsorted bin
- fast bins
- small bins
- large bins
- (tcache)

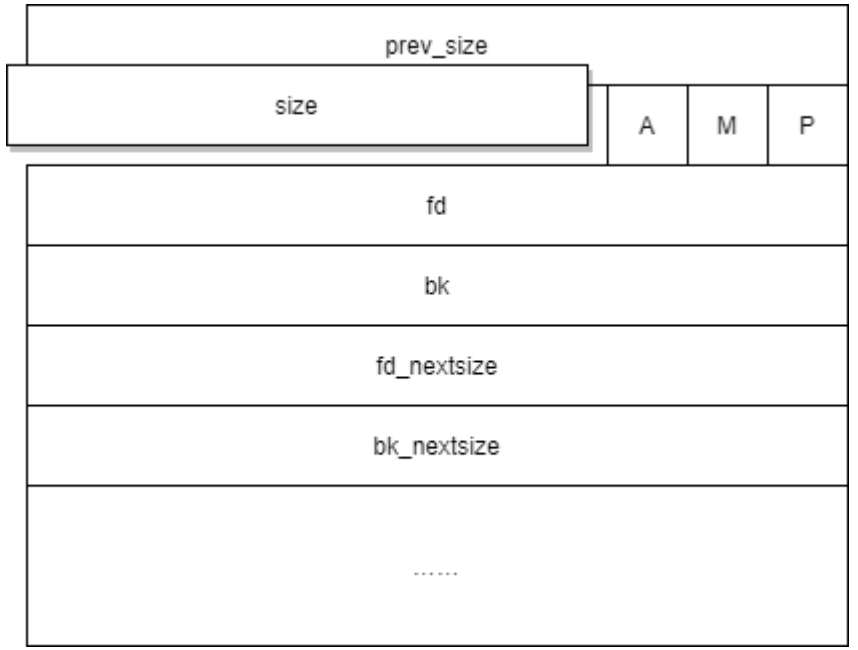
prev_size

若前一个物理相邻的chunk是free chunk，则表示其大小。否则用于存储前一个chunk的数据



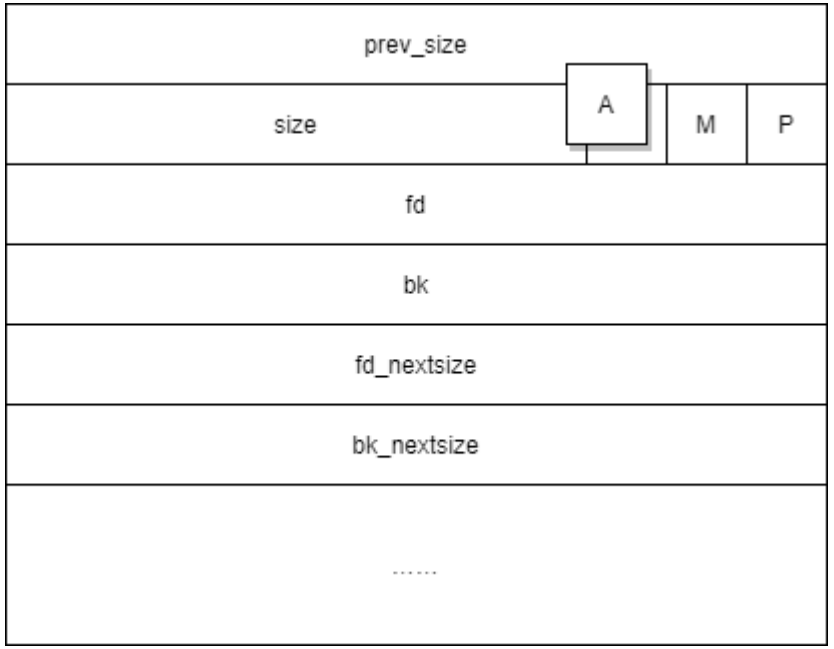
size

占据一字长的低3bits以后的地址，用于表示当前chunk的大小（整个chunk的大小，包括chunk头）



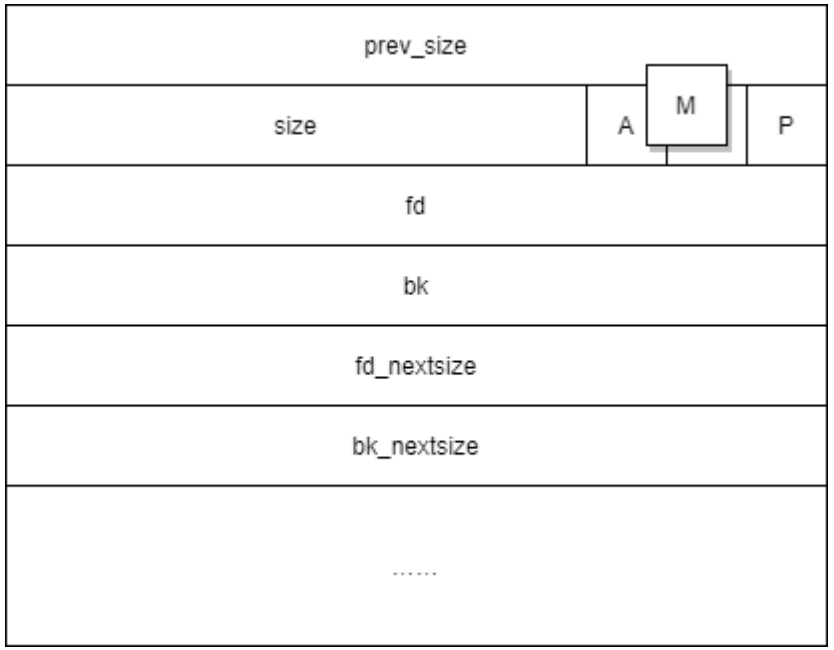
A flag

NON_MAIN_ARENA，记录当前chunk是否不属于主线程，1 表示不属于，0 表示属于。



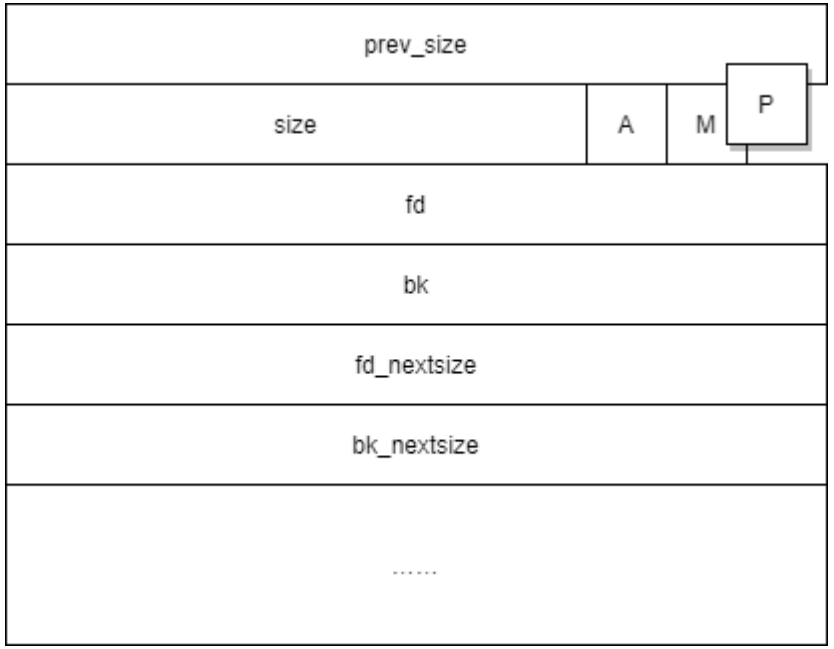
M flag

IS_MAPPED，记录当前 chunk 是否是由 mmap 分配的。



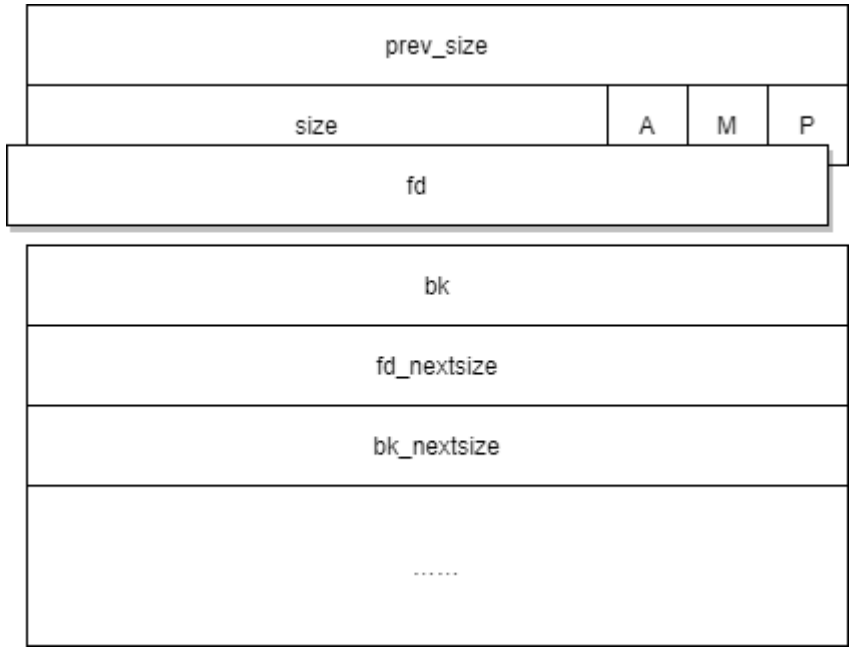
P flag

PREV_INUSE，记录前一个 chunk 块是否被分配。一般来说，堆中第一个被分配的内存块的 size 字段的 P 位都会被设置为 1，以便于防止访问前面的非法内存。当一个 chunk 的 size 的 P 位为 0 时，我们能通过 prev_size 字段来获取上一个 chunk 的大小以及地址。这也方便进行空闲 chunk 之间的合并。



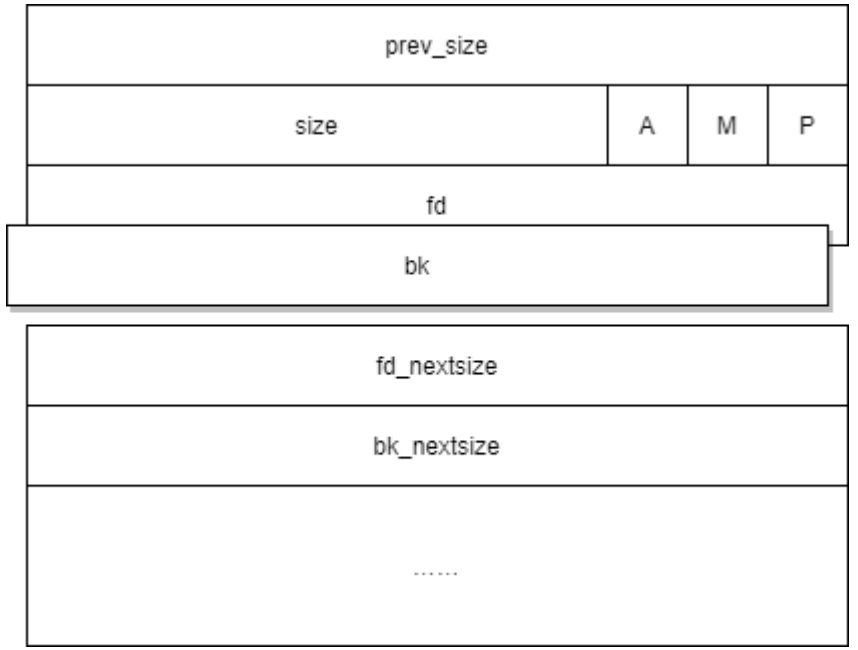
fd pointer

在bin中指向下一个（非物理相邻）空闲的 chunk



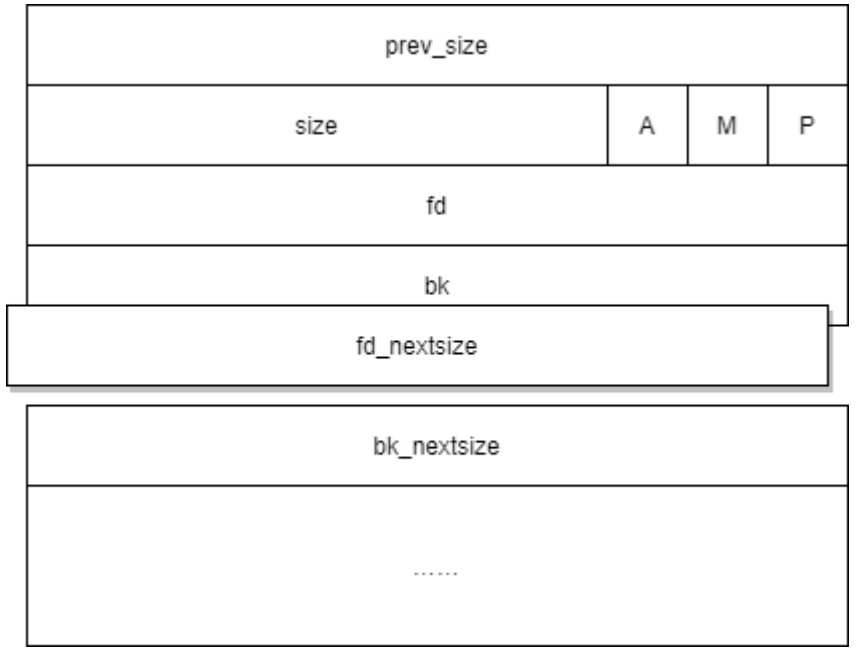
bk pointer

在bin中指向上一个（非物理相邻）空闲的 chunk



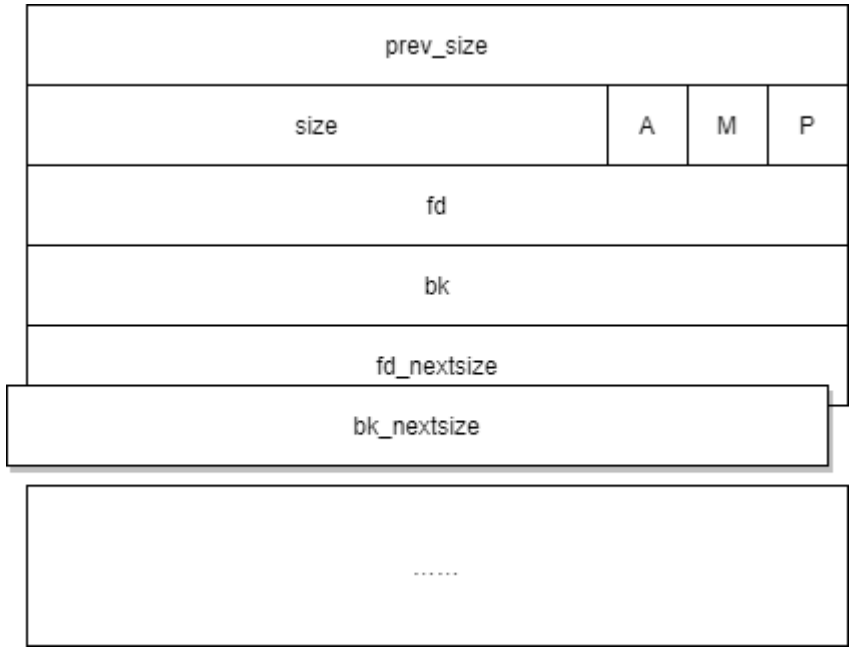
fd_nextsize

在large bin中指向前一个与当前chunk 大小不同的第一个空闲块，不包含 bin 的头指针



bk_nextsize

在large bin中指向后一个与当前chunk 大小不同的第一个空闲块，不包含 bin 的头指针



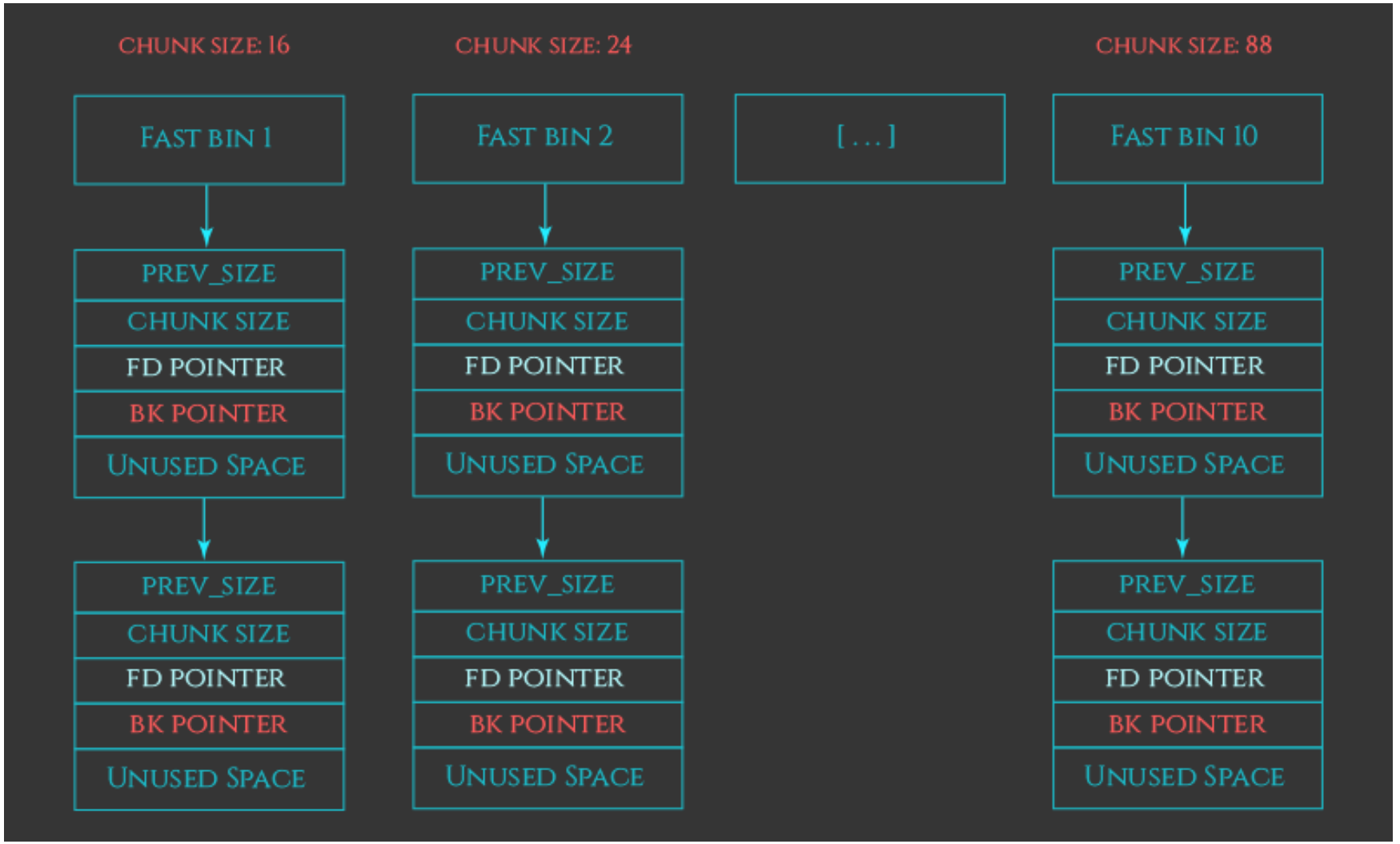
一般空闲的 large chunk 在 fd 的遍历顺序中，按照由大到小的顺序排列。这样做可以避免在寻找合适 chunk 时挨个遍历

main arena 的 malloc_state 并不是 heap segment 的一部分，而是一个全局变量，存储在 libc.so 的数据段

```
struct malloc_state {
    /* Serialize access.  */
    __libc_lock_define(, mutex);
    /* Flags (formerly in max_fast).  */
    int flags;
    /* Fastbins */
    mfastbinptr fastbinsY[ NFASTBINS ];
    /* Base of the topmost chunk -- not otherwise kept in a bin */
    mchunkptr top;
    /* The remainder from the most recent split of a small request */
    mchunkptr last_remainder;
    /* Normal bins packed as described above */
    mchunkptr bins[ NBINS * 2 - 2 ];
    /* Bitmap of bins, help to speed up the process of determinating if a given bin is definitely empty.*/
    unsigned int binmap[ BINMAPSIZE ];
    /* Linked list, points to the next arena */
    struct malloc_state *next;
    /* Linked list for free arenas.  Access to this field is serialized
       by free_list_lock in arena.c.  */
    struct malloc_state *next_free;
    /* Number of threads attached to this arena.  0 if the arena is on
       the free list.  Access to this field is serialized by
       free_list_lock in arena.c.  */
    INTERNAL_SIZE_T attached_threads;
    /* Memory allocated from the system in this arena.  */
    INTERNAL_SIZE_T system_mem;
    INTERNAL_SIZE_T max_system_mem;
};
```

fast bins

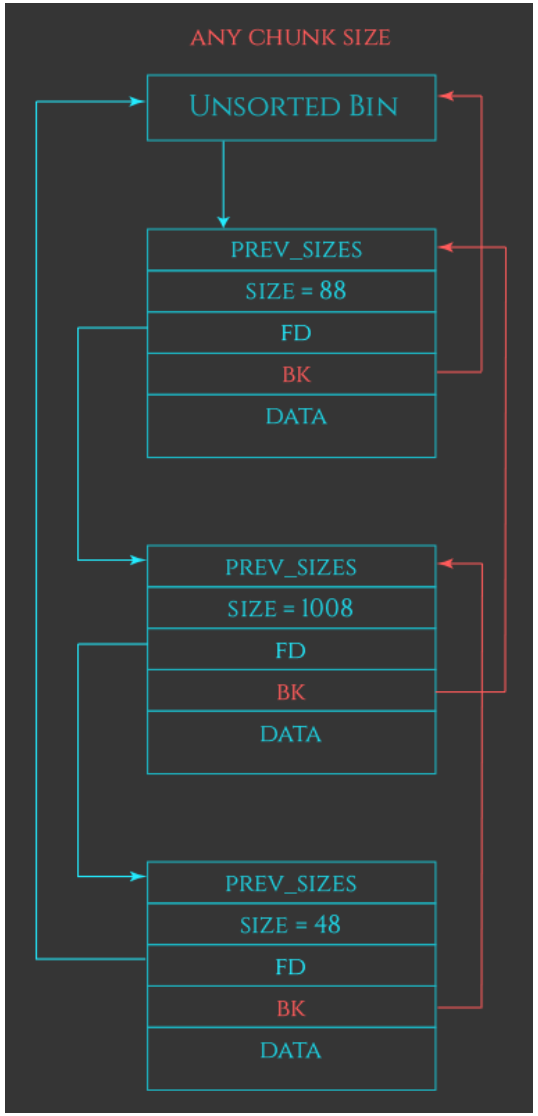
- fastbinsY[]
- 单向列表
- LIFO
- 管理 16、24、32、40、48、56、64 Bytes 的 free chunks（32位下默认）
- 其中的 chunk 的 in_use 位（下一个物理相邻的 chunk 的 P 位）总为1


CHUNK SIZE: 16
CHUNK SIZE: 24
CHUNK SIZE: 88
FAST BIN 1
FAST BIN 2
[. . .]
FAST BIN 10
PREV_SIZE
CHUNK SIZE
FD POINTER
BK POINTER
UNUSED SPACE
PREV_SIZE
CHUNK SIZE
FD POINTER
BK POINTER
UNUSED SPACE
PREV_SIZE
CHUNK SIZE
FD POINTER
BK POINTER
UNUSED SPACE

unsorted bin

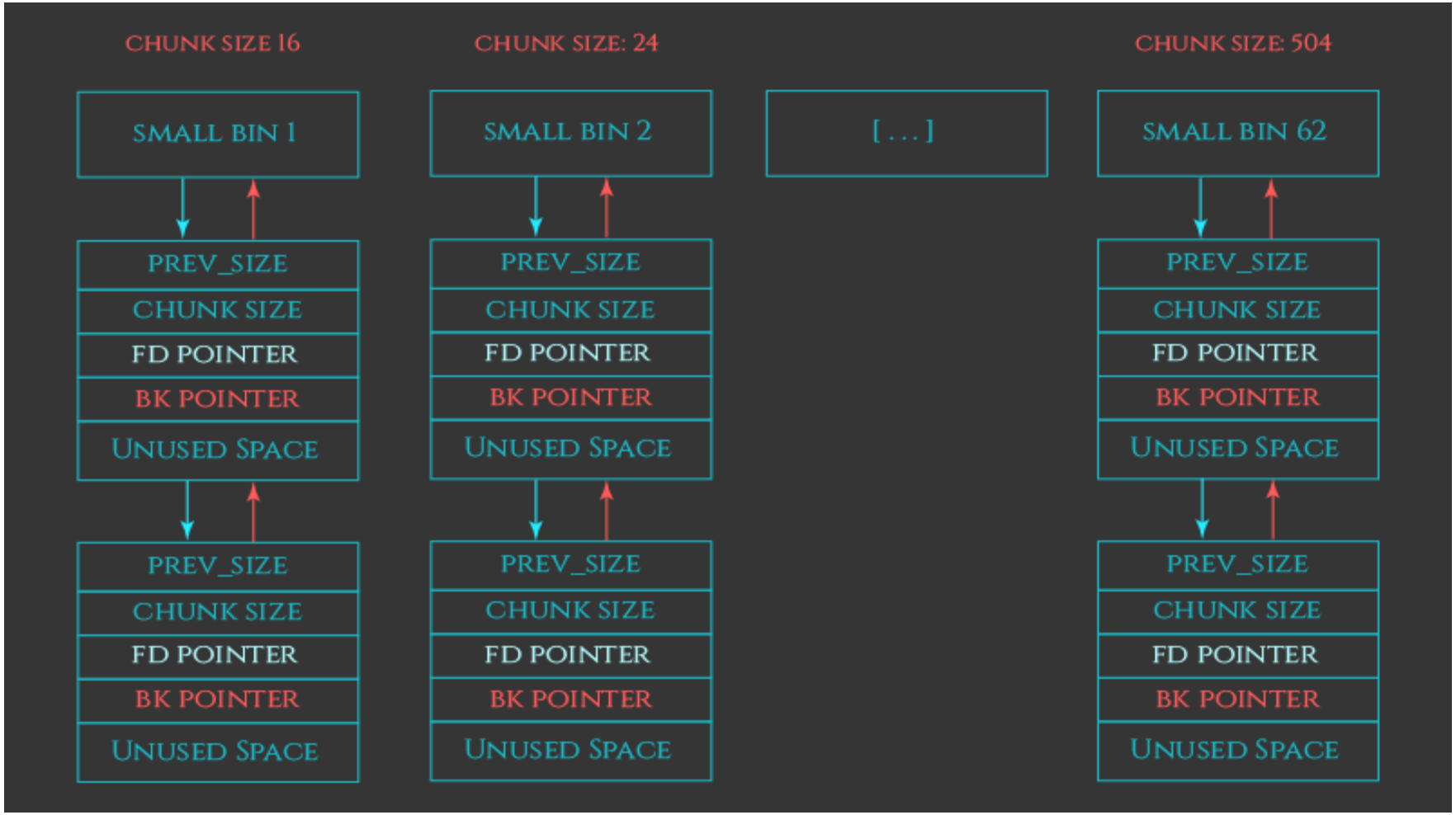
- bins[1]
- 管理刚刚释放还为分类的 chunk
- 可以视为空闲 chunk 回归其所属 bin 之前的缓冲区

堆利用 | bin


ANY CHUNK SIZE
UNSORTED BIN
PREV_SIZES
SIZE = 88
FD
BK
DATA
PREV_SIZES
SIZE = 1008
FD
BK
DATA
PREV_SIZES
SIZE = 48
FD
BK
DATA

small bins

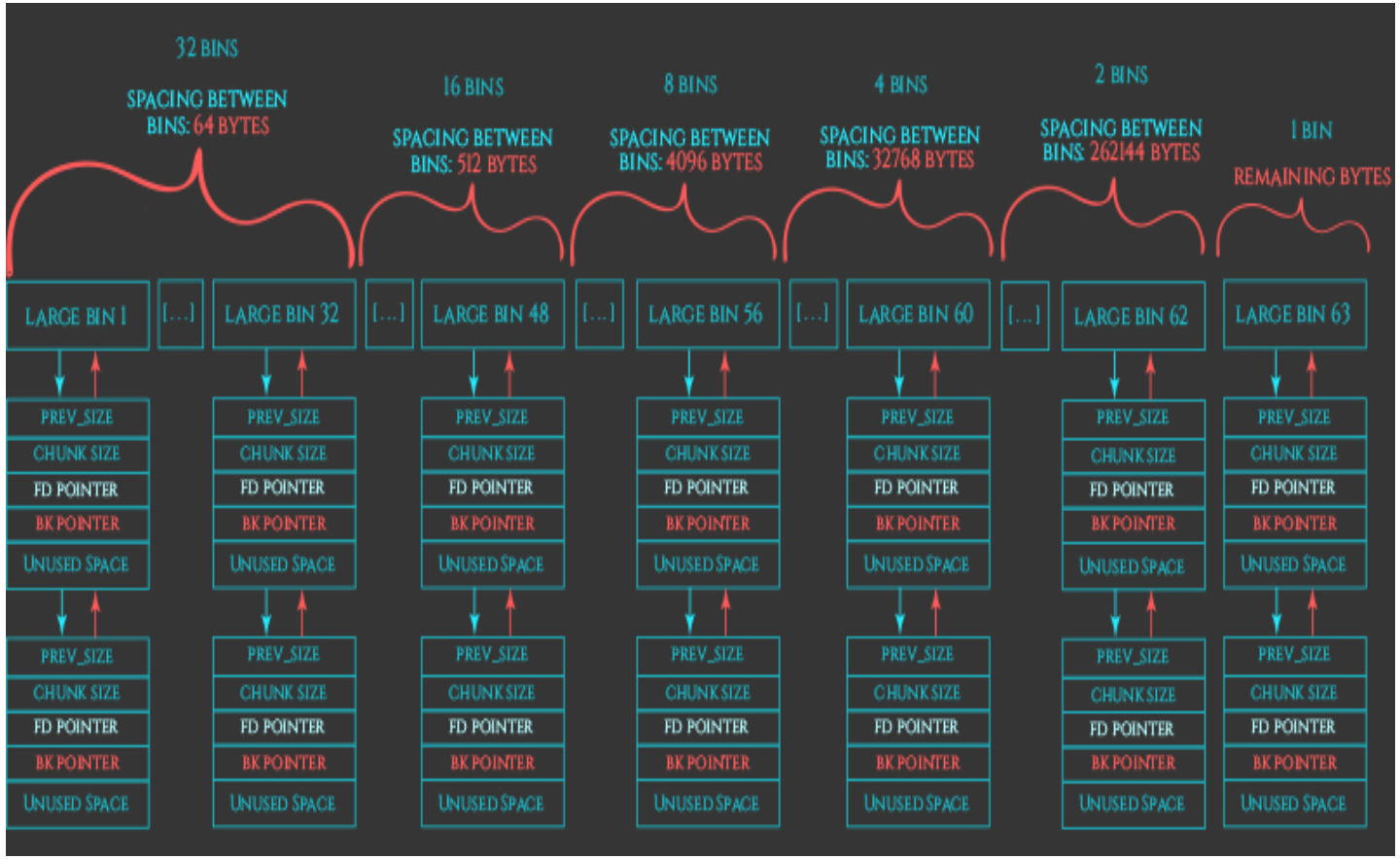
- bins[2] ~ bins[63]
- 62 个循环双向链表
- FIFO
- 管理 16、24、32、40、 …… 、504 Bytes 的 free chunks（32位下）
- 每个链表中存储的 chunk 大小都一致


CHUNK SIZE 16
CHUNK SIZE: 24
CHUNK SIZE: 504
SMALL BIN 1
SMALL BIN 2
[. . .]
SMALL BIN 62
PREV_SIZE
CHUNK SIZE
FD POINTER
BK POINTER
UNUSED SPACE
PREV_SIZE
CHUNK SIZE
FD POINTER
BK POINTER
UNUSED SPACE
PREV_SIZE
CHUNK SIZE
FD POINTER
BK POINTER
UNUSED SPACE

large bins

- bins[64] ~ bins[126]
- 63 个循环双向链表
- FIFO
- 管理大于 504 Bytes 的 free chunks（32位下）

堆利用 | bin


32 BINS
SPACING BETWEEN BINS: 64 BYTES
16 BINS
SPACING BETWEEN BINS: 512 BYTES
8 BINS
SPACING BETWEEN BINS: 4096 BYTES
4 BINS
SPACING BETWEEN BINS: 32768 BYTES
2 BINS
SPACING BETWEEN BINS: 262144 BYTES
1 BIN
REMAINING BYTES
LARGE BIN 1
[...]
LARGE BIN 32
[...]
LARGE BIN 48
[...]
LARGE BIN 56
[...]
LARGE BIN 60
[...]
LARGE BIN 62
LARGE BIN 63
PREV_SIZE
CHUNK SIZE
FD POINTER
BK POINTER
UNUSED SPACE

malloc

- 它根据用户申请的内存块大小以及相应大小 chunk 通常使用的频度（fastbin chunk, small chunk, large chunk），依次实现了不同的分配方法。
- 它由小到大依次检查不同的 bin 中是否有相应的空闲块可以满足用户请求的内存。
- 当所有的空闲 chunk 都无法满足时，它会考虑 top chunk。
- 当 top chunk 也无法满足时，堆分配器才会进行内存块申请。

free

- 它将用户暂且不用的chunk回收给堆管理器，适当的时候还会归还给操作系统。
- 它依据chunk大小来优先试图将free chunk链入tcache或者是fast bin。不满足则链入usorted bin中。
- 在条件满足时free函数遍历usorted bin并将其中的物理相邻的free chunk合并，将相应大小的chunk分类放入small bin或large bin中。
- 除了tcache chunk与fast bin chunk，其它chunk在free时会与其物理相邻的free chunk合并

Part6 AWD

- 堆管理器
  - 堆概述
  - Areana
  - Chunk
  - Bin
- 堆漏洞与其利用

•分析流量的时候才发现平台提供的流量只有进来的流量, 没有从服务器返回的流量, 所以看不到完整的交互, 所以根据流量生成exp的脚本几乎没怎么用到
•打别人的时候没有加混淆, 打了3, 4轮就打不到了, 都被patch了, 而且还看到了别人用我们的流量打我们, 不得不说有些队复用效率太高了
打别的队的时候一定要加混淆
如果是awd题目的话出题的时候可以多留几个后门, 要不然也不会像这次一样打个3, 4轮就打不了了. 而且也可以加一些反调手段, 争取可以多打几轮
较新颖的姿势一般不会出现在awd

PartEx1 内存保护措施

- ASLR
- PIE
- the NX bits
- Canary
- RELRO

ASLR (Address Space Layout Randomization)

- 系统的防护措施，程序装载时生效
  - /proc/sys/kernel/randomize_va_space = 0：没有随机化。即关闭 ASLR
  - /proc/sys/kernel/randomize_va_space = 1：保留的随机化。共享库、栈、mmap() 以及 VDSO 将被随机化
  - /proc/sys/kernel/randomize_va_space = 2：完全的随机化。在randomize_va_space = 1的基础上，通过 brk() 分配的内存空间也将被随机化

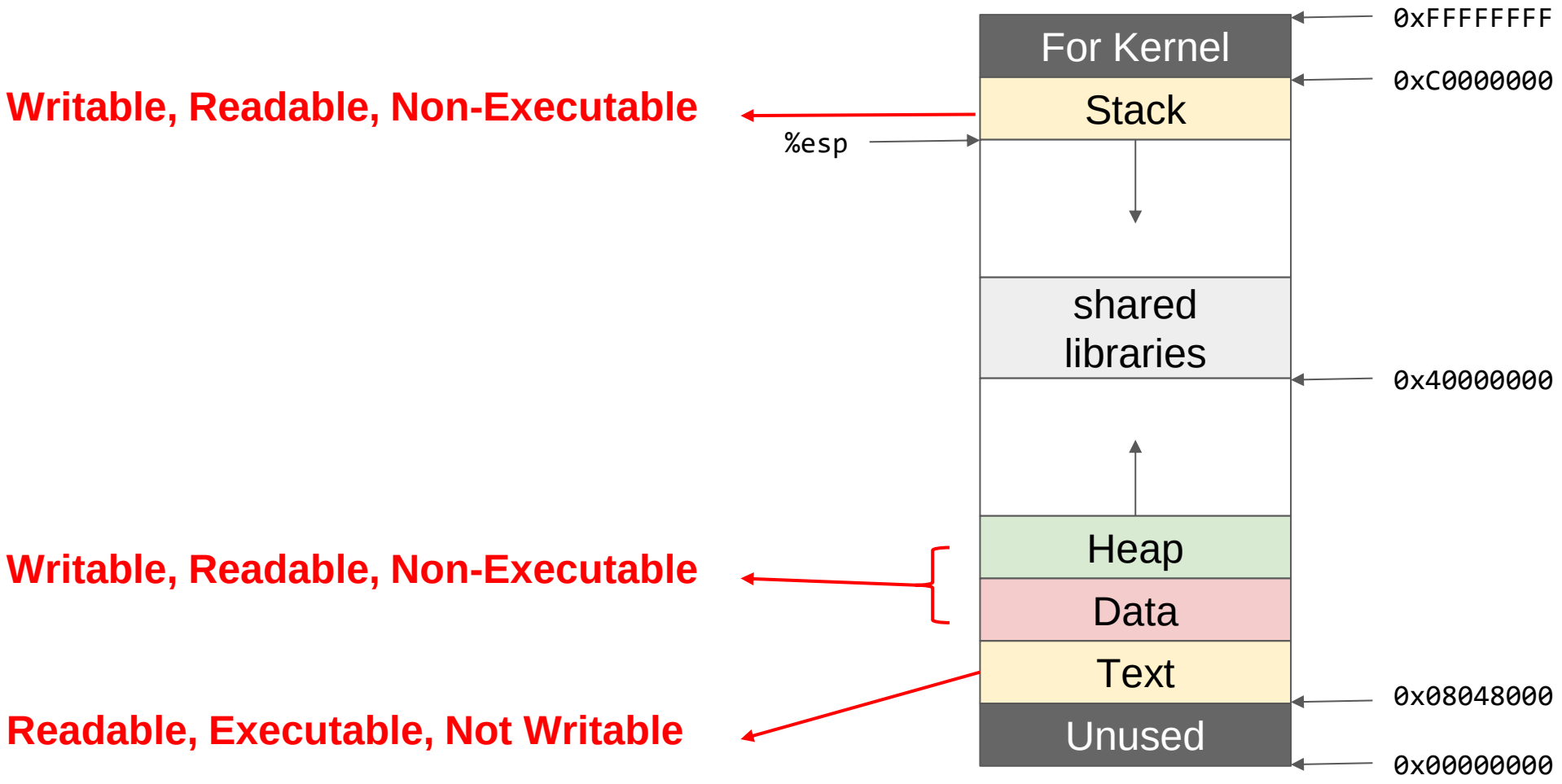
PIE (Position-Independent Executable)

- 程序的防护措施，编译时生效
- 随机化ELF文件的映射地址
- 开启 ASLR 之后，PIE 才会生效

The NX bits (the No-eXecute bits)

- 程序与操作系统的防护措施，编译时决定是否生效，由操作系统实现
- 通过在内存页的标识中增加“执行”位，可以表示该内存页是否可以执行，若程序代码的 EIP 执行至不可运行的内存页，则 CPU 将直接拒绝执行“指令”造成程序崩溃

The NX bits (the No-eXecute bits)



Canary

- 程序的防护措施，编译时生效
- 在刚进入函数时，在栈上放置一个标志canary，在函数返回时检测其是否被改变。以达到防护栈溢出的目的
- canary长度为1字长，其位置不一定与ebp/rbp存储的位置相邻，具体得看程序的汇编操作

RELRO (RELocate Read-Only)

- 程序的防护措施，编译时生效
  - 部分 RELRO: 在程序装入后, 将其中一些段(如.dynamic)标记为只读, 防止程序的一些重定位信息被修改
  - 完全 RELRO: 在部分 RELRO 的基础上, 在程序装入时, 直接解析完所有符号并填入对应的值, 此时所有的 GOT 表项都已初始化, 且不装入link_map与_dl_runtime_resolve的地址

PartEx2 PWN Tools

- IDA Pro
- pwntools
- pwndbg
- checksec
- ROPgadget
- one_gadget
- LibcSearcher
- main_arena_offset

IDA Pro

pwntools

pwndbg

checksec

ROPgadget

One_gadget