Heap Exploitation

讲师 Jr @ Lancet
ID起了好多个自己都忘了

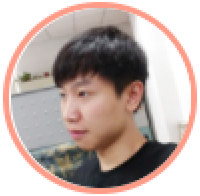
About me

XCTF

课程题目： 《CTF中的堆利用技术》

课程大纲：

1. Linux堆管理基础
2. 堆溢出的基本利用技巧（Unlink、Double Free、Use After Free）
3. 堆溢出进阶（Malloc maleficarum、XXXBin Attack、Off by one）
4. 常见的几种导致堆破坏的场景



讲师： 简容

1. Lancet战队bin选手2. 主攻写段子，虽然没有学医但是也想成为安全老司机 3. 广告位招租

目录 content

目录 content

Heap Overview

- Why use Heap
- 堆的分配和释放可以由用户自由控制
- 堆的空间不一定连续
- 不同的系统有着不同的堆管理机制

```
Start              End                Perm      Name
0x0000555555554000 0x0000555555555000 r-xp      /root/pwn_course/heap
0x0000555555754000 0x0000555555755000 r--p      /root/pwn_course/heap
0x0000555555755000 0x0000555555756000 rw-p      /root/pwn_course/heap
0x0000555555756000 0x0000555555777000 rw-p      [heap]
0x00007ffff7a3b000 0x00007ffff7bd0000 r-xp      /lib/x86_64-linux-gnu/libc-2.24.so
```

Heap Overview

malloc、free、realloc

不一样的实现方法

- dlmalloc – General purpose allocator
- ptmalloc2 – glibc
- jemalloc – Firefox and Android
- tcmalloc – Google Chrome
- libumem – Solaris
- homework – Yourself

Heap Overview

- **Read The Fucking Source Code**
    - http://ftp.gnu.org/gnu/glibc/
- 对于堆管理的实现，比较直观的印象(your homework require)
    - 将一片内存切分成块
    - 使用合理的数据结构来组织（链表、树、等等）
    - 被释放的堆应该能被快速重用
    - 适当减少堆碎片的产生
    - 加上一丢丢的安全机制

Heap Overview

XCTF

- 为了减少系统调用的次数，heap allocator充当了中间层的作用
- 从前没有堆，brk之后就有了堆
- 这一片连续的空间，被称作arena
- 由于是主线程创建的arena，因此被称作main_arena

Heap Chunk

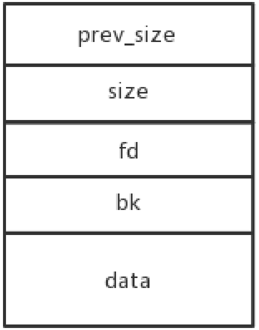
- chunk是堆的基本单位

```c
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;
};
```



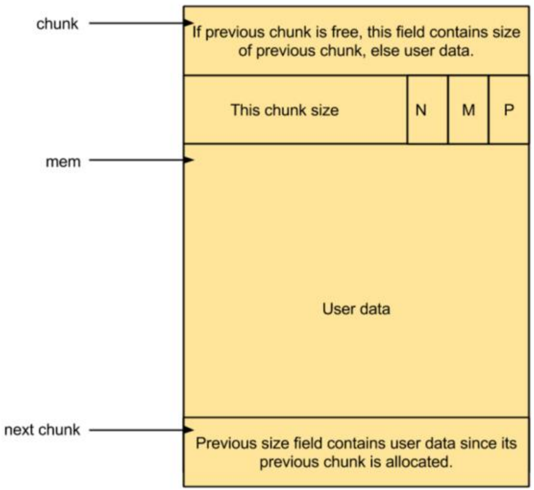
Heap Chunk

- size
  - 堆块的长度
  - 由于size的后3bit恒为0，故将这3位用于标志位（p、m、n）
- prev_size
  - 代表前一个相邻堆块的大小
  - 仅在p标志位为1时才有意义
  - <u>当前一个堆不处于空闲态时</u>，则可以为前一个堆中用户写入的数据
- fd&bk
  - 堆块在allocated状态时，无意义
  - freed状态下，用于形成链表结构

Heap Chunk

- 堆空间被切分为许多个chunk
- 两种状态：allocated、freed

allocated



P：prev inuse

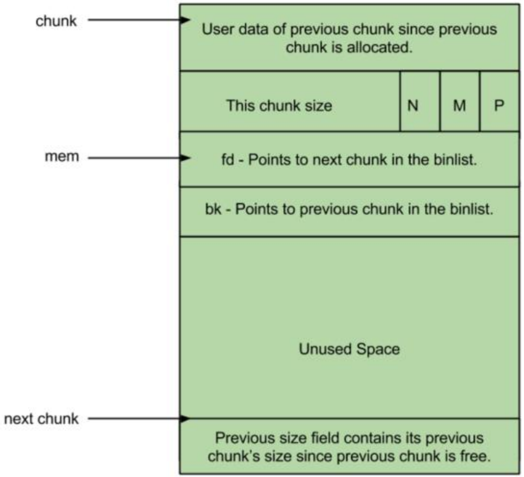
M：is_mmaped

N：non_main_arena

Heap Chunk

- 堆空间被切分为许多个chunk
- 两种状态：allocated、freed

freed



- 双向链表：fd和bk同时被使用
- 单向链表：仅仅使用fd

Heap Chunk

XCTF

- Top chunk
- 在第一次malloc时，heap将会被分成两个部分，第一部分就是被分配出去的堆；剩下的部分就叫做top chunk
- 在之后的分配中，若空间不足，则会从top chunk中切分
- 一个小实验

```
p1 = malloc(0x18);
p2 = malloc(0x88);
p3 = malloc(0x38);
memset(p1, 'a', 0x18);
free(p2);
```

- malloc(0x18) -> 用户得到什么？size字段实际有多大？
- free掉p2后，prev_size和size的标志位有何变化？

Heap Chunk

- p2被free之前

```
addr                prev              size       status
0x555555756000      0x0               0x20       Used
0x555555756020      0x61616161616161610x90       Used
0x5555557560b0      0x0               0x40       Used
```

chunk->	0x555555756000:	0x0000000000000000	0x0000000000000021
user ptr->	0x555555756010:	0x6161616161616161	0x6161616161616161
	0x555555756020:	0x6161616161616161	0x0000000000000091
	0x555555756030:	0x0000000000000000	0x0000000000000000
	0x555555756040:	0x0000000000000000	0x0000000000000000
	0x555555756050:	0x0000000000000000	0x0000000000000000
	0x555555756060:	0x0000000000000000	0x0000000000000000
	0x555555756070:	0x0000000000000000	0x0000000000000000
	0x555555756080:	0x0000000000000000	0x0000000000000000
	0x555555756090:	0x0000000000000000	0x0000000000000000
	0x5555557560a0:	0x0000000000000000	0x0000000000000000
	0x5555557560b0:	0x0000000000000000	0x0000000000000041
	0x5555557560c0:	0x0000000000000000	0x0000000000000000

Heap Chunk

- p2被free之后

addr	prev	size	status	fd	bk
0x555555756000	0x0	0x20	Used	None	None
0x555555756020	0x61616161616161610x90		Freed	0x7ffff7dd3b58	0x7ffff7dd3b58
0x5555557560b0	0x90	0x40	Used	None	None

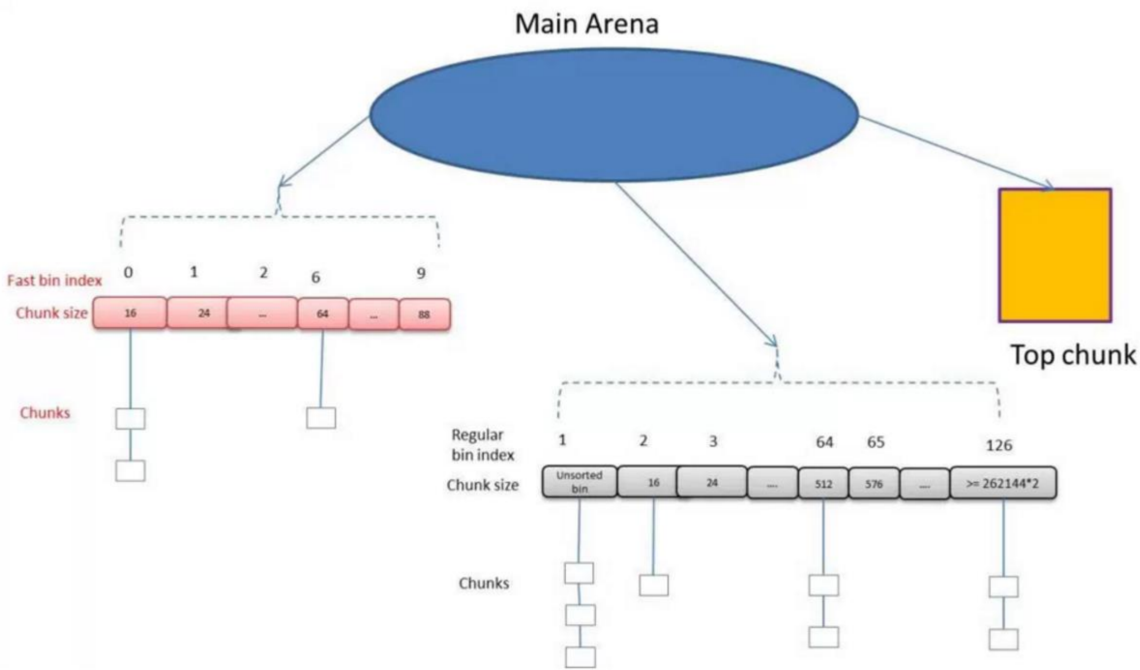
0x555555756000:	0x0000000000000000	0x0000000000000021
0x555555756010:	0x6161616161616161	0x6161616161616161
0x555555756020:	0x6161616161616161	0x0000000000000091
0x555555756030:	0x00007ffff7dd3b58	0x00007ffff7dd3b58
0x555555756040:	0x0000000000000000	0x0000000000000000
0x555555756050:	0x0000000000000000	0x0000000000000000
0x555555756060:	0x0000000000000000	0x0000000000000000
0x555555756070:	0x0000000000000000	0x0000000000000000
0x555555756080:	0x0000000000000000	0x0000000000000000
0x555555756090:	0x0000000000000000	0x0000000000000000
0x5555557560a0:	0x0000000000000000	0x0000000000000000
0x5555557560b0:	0x0000000000000090	0x0000000000000040
0x5555557560c0:	0x0000000000000000	0x0000000000000000

Main Arena

- chunk需要被有效的组织和利用
- main_arena是堆管理中的一个重要结构
- main_arena之大，源码一页贴不下
- 其对应的数据结构为malloc_state

mutex	用于多线程支持
flagd	用于标识该Arena的性质，例如是否连续，是否有fastbin可用等
fastbinsY[NFASTBINS]	fastbin指针数组
top	指向top chunk的指针
bins[NBINS * 2 - 2]	bins数组指针
*next	指向下一个Arena，构成一个循环单向链表
*next_free	指向idle Arena，便于线程快速匹配

Main Arena



Main Arena
Fast bin index
0
1
2
6
9
Chunk size
16
24
...
64
...
88
Chunks
Top chunk
Regular
bin index
1
2
3
64
65
126
Chunk size
Unsorted
bin
16
24
...
512
576
...
>= 262144*2
Chunks

Multi-thread arena

- 产品经理说了，多线程的我们也要handle
- 一个小实验

```
void* threadFunc(void* arg) {
      printf("Before malloc in thread 1\n");
      getchar();
      char* addr = (char*) malloc(1000);
      printf("After malloc and before free in thread 
1\n");
      getchar();
      free(addr);
      printf("After free in thread 1\n");
      getchar();
}
```

Multi-thread arena

- Before thread malloc

```
0x0000555555554000 0x0000555555555000 r-xp     /root/pwn_course/heap
0x0000555555754000 0x0000555555755000 r--p     /root/pwn_course/heap
0x0000555555755000 0x0000555555756000 rw-p     /root/pwn_course/heap
0x0000555555756000 0x0000555555777000 rw-p     [heap]
0x00007ffff781e000 0x00007ffff79b3000 r-xp     /lib/x86_64-linux-gnu/libc-2.24.so
```

- After thread malloc

```
0x0000555555554000 0x0000555555555000 r-xp     /root/pwn_course/heap
0x0000555555754000 0x0000555555755000 r--p     /root/pwn_course/heap
0x0000555555755000 0x0000555555756000 rw-p     /root/pwn_course/heap
0x0000555555756000 0x0000555555777000 rw-p     [heap]
0x00007ffff0000000 0x00007ffff0021000 rw-p     mapped
0x00007ffff0021000 0x00007ffff4000000 ---p     mapped
0x00007ffff701d000 0x00007ffff701e000 ---p     mapped
0x00007ffff701e000 0x00007ffff781e000 rw-p     mapped
```

- 在线程中创建的arena，被称作thread arena
- 虽然我们只请求了1000bytes，但是mmap了0x781e000的大小
- 在新mmap的空间中，只有两个区域被标记位rw

Multi-thread arena

- thread_arena不是想要就能要
- arena最大数量由CPU核心数决定
  - 32bit system ： Number of arena = 2 * Number of cores
  - 64bit system ： Number of arena = 8 * Number of cores
- 当num threads > num arenas
  - 遍历找到可用的arena
  - 使用mutex机制为arena加锁

Mechanism

<u>以32位系统为例</u>

Bins是空闲堆块链表中的基本数据结构，它们被用于保存空闲堆块

- Fast bins
- Unsorted bins
- Small bins
- Large bins

后面三种均在bins[NBINS*2 – 2]中

- Bin 1 – Unsorted bin
- Bin 2 to Bin 63 – Small bin
- Bin 64 to Bin 126 – Large bin
- 注意其双向链表结构

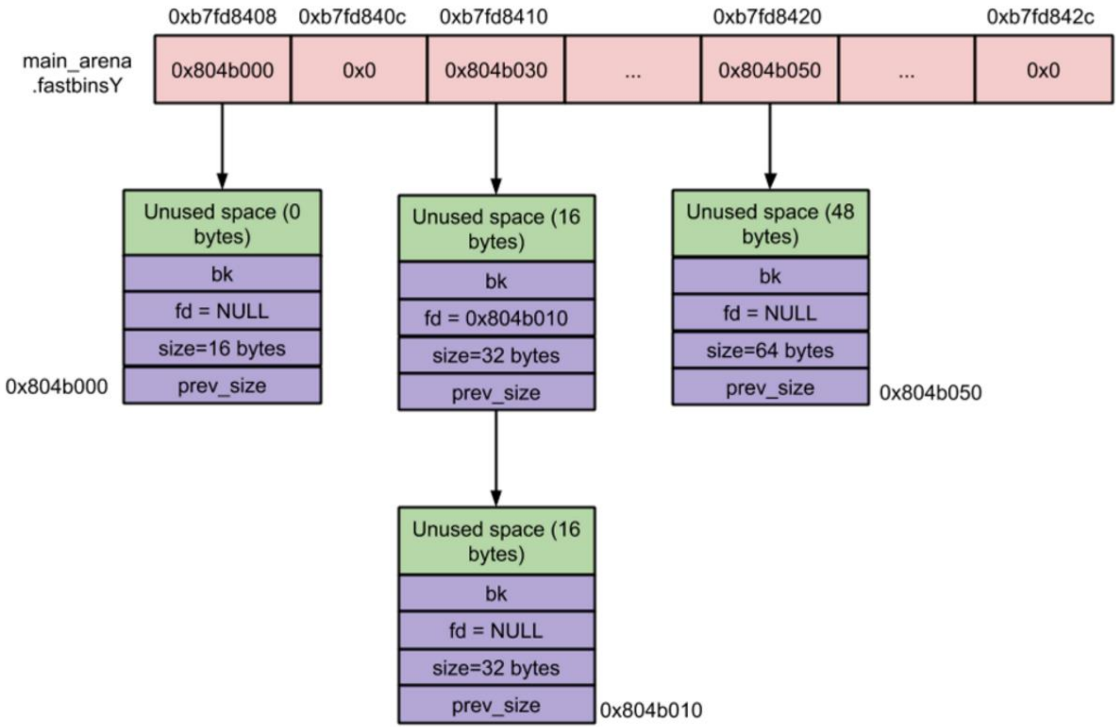
Fast bin

XCTF

fast bin range ： 0x10(16) <= size <= 0x40(64)

- 单向链表 —— FILO
- 每个链表中的size相同，以8bytes为单位递增
- 被free的堆块仍被标记位inuse（防止堆块合并）

Fast bin



0xb7fd8408
0xb7fd840c
0xb7fd8410
0xb7fd8420
0xb7fd842c
main_arena
.fastbinsY
0x804b000
0x0
0x804b030
...
0x804b050
...
0x0
Unused space (0 bytes)
bk
fd = NULL
size=16 bytes
prev_size
0x804b000
Unused space (16 bytes)
bk
fd = 0x804b010
size=32 bytes
prev_size
Unused space (48 bytes)
bk
fd = NULL
size=64 bytes
prev_size
0x804b050
Unused space (16 bytes)
bk
fd = NULL
size=32 bytes
prev_size
0x804b010

Unsorted bin

当一个Small chunk或者Large chunk被free的时候，它们不会被放入对应的bin链表中，而是被加入到Unsorted bin中

- 只有一个，双向循环链表 —— FIFO
- 任何大小的chunk都可以存在于Unsorted bin
- 为了glibc快速重用被释放的堆

Bin 1 – Unsorted bin

Bin 2 to Bin 63 – Small bin

Bin 64 to Bin 126 – Large bin

Small bin

small bin range ： 0x10(16) <= size <= 0x1F8(504)

- 双向循环链表 —— FIFO
- 每个链表中的size相同，以8bytes为单位递增

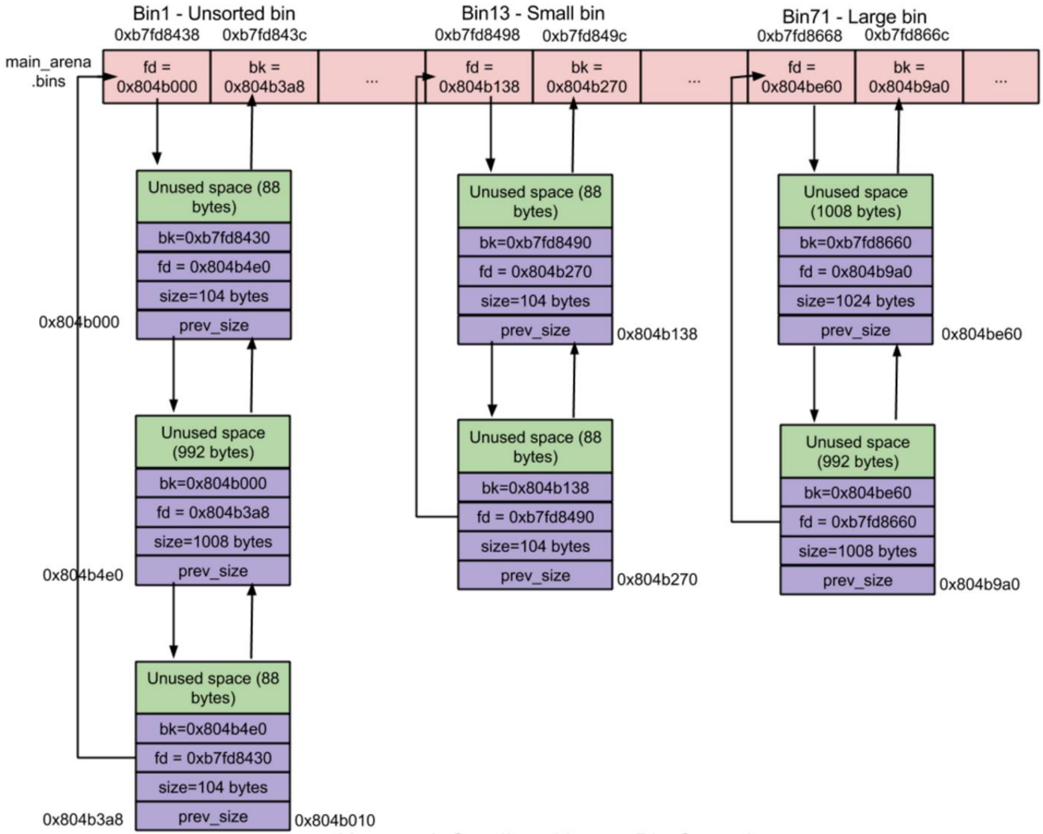
small bin size范围好像和fast bin有重合？

Large bin

Large bin range ： size >= 0x200(512)

- 双向循环链表 —— 可以在任意位置被拆卸
- 每个链表中的size不相同
    - 32个包含以64bytes为单位递增的链表。如第一个Large bin(bin 64)包含了512<=size<576的块
    - 16个包含以512bytes为单位递增的链表
    - 8个包含以4096bytes为单位递增的链表
    - 4包含以32768bytes为单位递增的链表
    - 2包含以262144bytes为单位递增的链表
    - 1包含剩下大小块的链表
- 每个链表中，chunk降序排列

Bins Overview



Unsorted, Small and Large Bin Snapshot

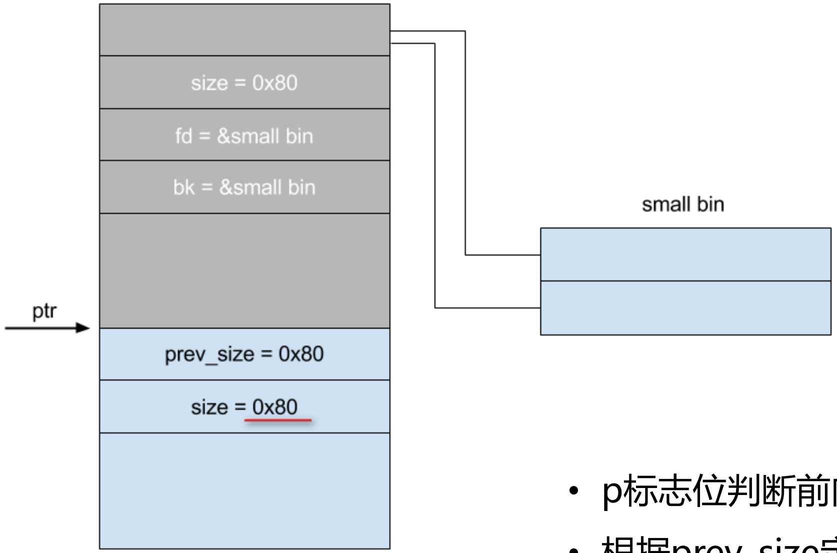
Free Mechanism

XCTF

当一个chunk被free时

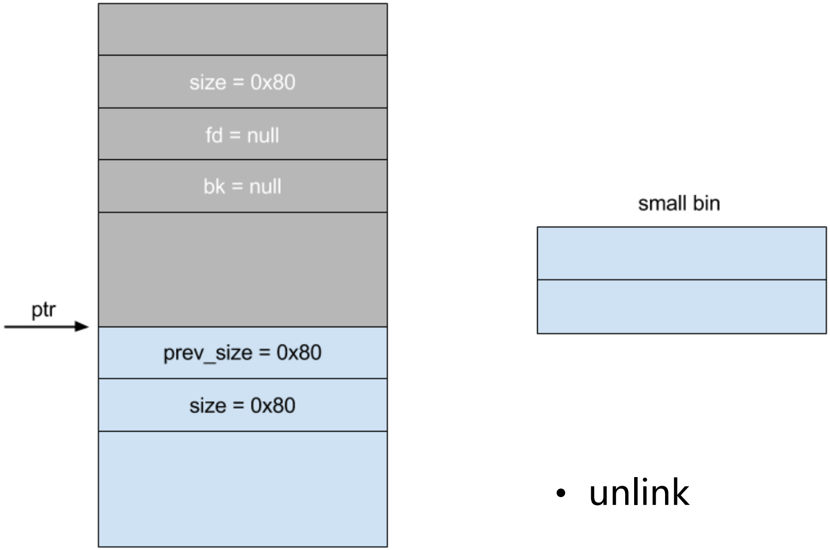
- size是否属于fast bin？
  - 是，插入fast bin 链表，结束
- 是不是mmap分配出去的内存？
  - 是，munmap，结束
- 与当前被free chunk的前一个相邻堆块是不是freed状态？
  - 是，则两个chunk合并
- 与当前被free chunk的后一个相邻堆块是不是top chunk？
  - 是，则与top chunk合并，结束
- 与当前被free chunk的后一个相邻堆块是不是freed状态？
  - 是，则两个chunk合并
- 将该chunk链入Unsorted bin

Free Mechanism



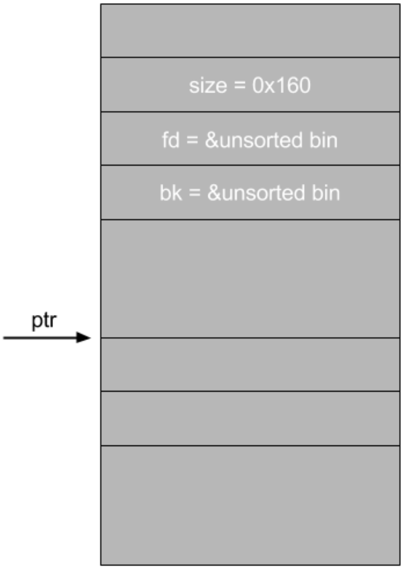
- p标志位判断前向堆块的状态
- 根据prev_size定位

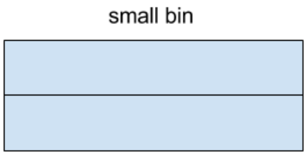
Free Mechanism



- unlink

Free Mechanism





- merge
- 链入Unsorted bin

Malloc Mechanism

当使用malloc分配内存时

- size是否属于fast bin？
    - 是，寻找对应链表
        - 若找到，则拆下对应chunk，结束
- size是否属于small bin？
    - 是，寻找对应链表
        - 若找到，则拆下对应chunk，结束
- 尝试使用Unsorted bin分配
    - 遍历、分割、拆卸

Malloc Mechanism

Unsorted bin 分配机制

- 从链表尾部开始遍历
  - 切分操作（剩下的仍存在于Unsorted bin）

```
in_smallbin_range (nb) &&
bck == unsorted_chunks (av) &&
victim == av->last_remainder &&
(unsigned long) (size) > (unsigned long) (nb + MINSIZE))
```

  - 若不满足切分条件，则进行拆链操作

```
/* remove from unsorted list */
unsorted_chunks (av)->bk = bck;
bck->fd = unsorted_chunks (av);
```

  - 若当前chunk不满足条件，则根据大小放入对应list(Small or Large) —— small bin 和 fast bin size range有重合的原因
- victim即使不满足分配需要，拆链和放入list的操作都会进行
- 遍历下一个chunk

Malloc Mechanism

当Unsorted bin 分配失败时

- size是否属于large bin？
    - 是，寻找对应链表
        - 若找到，则拆下(切分)对应chunk
        - 剩下的加入Unsorted bin，结束
- large bin 仍然不满足要求
    - 再次从small bin 和 large bin中寻找
    - 是否存在size(victim) > size(nb)的chunk
    - 若有，则切分，剩下的加入Unsorted bin
- 还是不行？则使用top chunk
- 还是不行？？江郎才尽，扩展堆空间吧

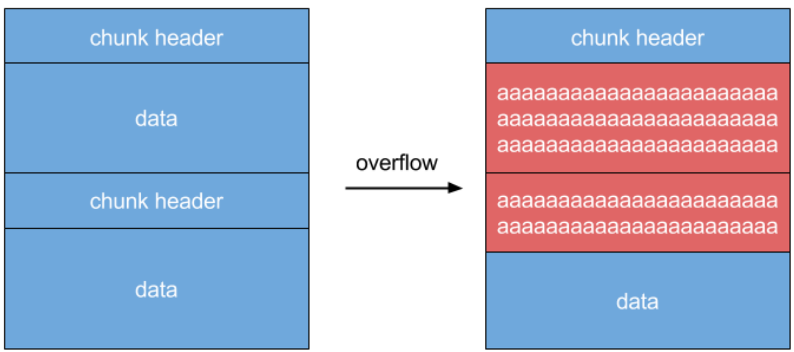
XCTF

目录 content

Vulnerability of Heap

Base on memory corruption

- Heap overflow
- Overwrite important heap metadata
- Overwrite important user data (eg. func_ptr)



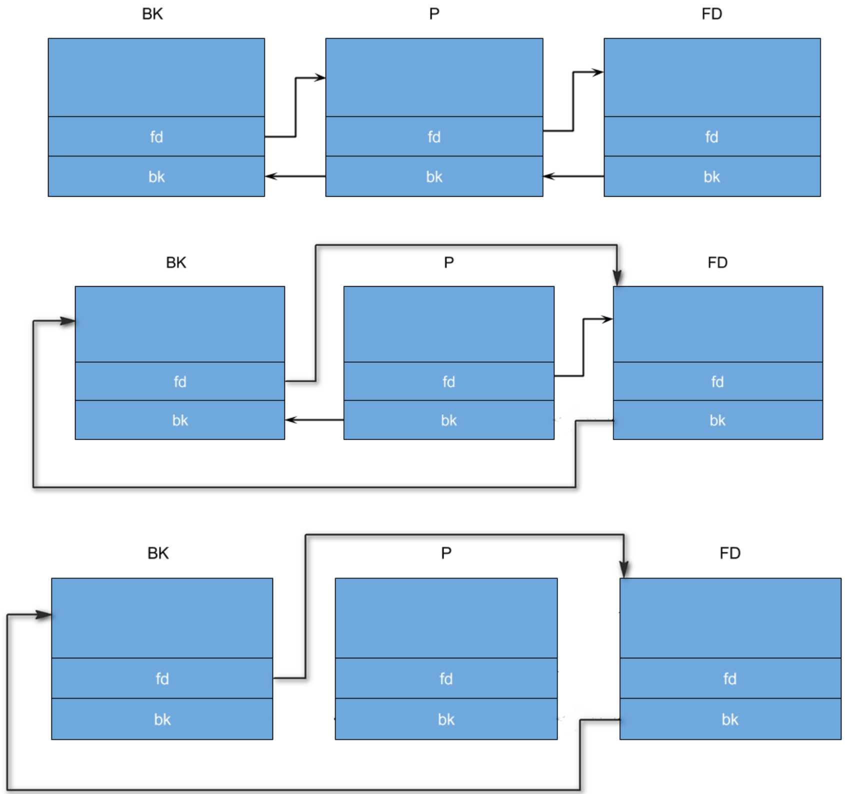
Unlink

链表结构？

- overwrite fd or bk
- trigger unlink
- unlink in ancient times

```
unlink(P, BK, FD) {
        FD = P -> fd;
        BK = P -> bk;
        FD -> bk = BK;
        BK -> fd = FD;
}
```

Unlink




```
unlink(P, BK, FD) {
        FD = P -> fd;
        BK = P -> bk;
        FD -> bk = BK;
        BK -> fd = FD;
}
```

Unlink

what if fd & bk are under control ?

- FD -> bk = BK
  - shellcode + 12 = got_entry – 8
- BK -> fd = FD
  - got_entry = shellcode

P

prev_size = 0
size = 0x81
fd = &shellcode
bk = got_entry - 8

Unlink

一个栗子

P

prev_size = 0
size = 0x81
fd = &shellcode
bk = got_entry - 8

Unlink

unlink in real world

```
if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                \
      malloc_printerr (check_action, "corrupted double-linked list", P);
}
```

- 对指针的有效性做了检查
- 虽然能够bypass，但是可以写入的内容收到了很大限制
- 最终的结果
    - p = &p – 8（x86）
    - p = &p – 16（x64）

Unlink

unlink的利用

- 在_int_free函数中
- free chunk时的堆块前向、后向合并，将使用unlink

```
/* consolidate backward */
if (!prev_inuse(p)) {
  prevsize = p->prev_size;
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  unlink(p, bck, fwd);
}
```

```
    /* consolidate forward */
    if (!nextinuse) {
unlink(nextchunk, bck, fwd);
size += nextsize;
    } else
clear_inuse_bit_at_offset(nextchunk, 0);
```

Unlink

example

Double Free

直接double free？


```
*** Error in `./heap_2': double free or corruption (fasttop): 0x0804b008 ***
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(+0x6737a)[0xf7e6137a]
/lib/i386-linux-gnu/libc.so.6(+0x6dfb7)[0xf7e67fb7]
/lib/i386-linux-gnu/libc.so.6(+0x6e776)[0xf7e68776]
./heap_2[0x80488f3]
./heap_2[0x8048a75]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf6)[0xf7e12276]
./heap_2[0x8048551]
```

glibc中有许多检查是否double free的机制

Double Free

glibc中有许多检查是否double free的机制

```
    if (__builtin_expect (p == av->top, 0))
      {
  errstr = "double free or corruption (top)";
  goto errout;
      }
    /* Or whether the next chunk is beyond the boundaries of
    if (__builtin_expect (contiguous (av)
          && (char *) nextchunk
          >= ((char *) av->top + chunksize(av->top)), 0))
      {
  errstr = "double free or corruption (out)";
  goto errout;
      }
    /* Or whether the block is actually not marked used.  */
    if (__builtin_expect (!prev_inuse(nextchunk), 0))
      {
  errstr = "double free or corruption (!prev)";
```

Double Free

Double Free

```
+-------------------------+-------------------------+-------------------------+
|        Chunk 1          |        Chunk 2          |        Chunk 3          |
+-------------------------+-------------------------+-------------------------+
0x60000                   0x60080                   0x60100


+-------------------------+---------------------------------------------------+
|         Chunk 1         |        Chunk 4                                    |
+-------------------------+---------------------------------------------------+
0x60000                   0x60080                   0x60100


+-------------------------+---------------------------------------------------+
|         Chunk 1         |        Chunk 4                                    |
|   Fake chunk, FD and BK |                         prev_size points to the   |
|   point into the table  |                         fake chunk in Chunk 1     |
+-------------------------+---------------------------------------------------+
0x60000                   0x60080                   0x60100
```

Use after Free

有些堆死了，它还活着

- free完毕之后，并没有做好收尾工作
    - 指针未清0
    - 数据结构未更改
- 通过重用该chunk，可导致不同程度的影响
    - 一个直观栗子

Use after Free

```c
typedef struct _cmdlist {
    void (*sayhello)();
    void (*saygoodbyte)();
}cmdlist;

typedef struct _data {
    int age;
    int score;
}data;

int main() {
    cmdlist *p = (cmdlist *)malloc(sizeof(cmdlist));
    free(p);
    /* do something */
    data *q = (data *)malloc(sizeof(data));
    q -> age = 0x61616161;
    (*(p -> sayhello))();
    return 0;
}
```

Use after Free

- 由于cmdlist和data结构体size相同
  - malloc将重用被free掉的cmdlist
  - `data *q = (data *)malloc(sizeof(data));`
  - p和q此时指向同一chunk
- 通过设置data.age，得以劫持控制流
  - `(*(p -> sayhello))();`

```
gdb-peda$ x/i $pc
=> 0x56555619 <main+89>:        call    eax
gdb-peda$ i r eax
eax            0x61616161       0x61616161
```

Malloc-maleficarum

XCTF

Some stories before house of cards

- 2004年，glibc针对当时常用的一些攻击手法进行了版本更新
    - 包括unlink在内的版本强势英雄被削
- 2005年，Phantasmal Phantasmagoria发表了文章Malloc Maleficarum（ ***Malleus Maleficarum 女巫之锤***）
- 提出了一系列攻击heap机制的新方法
- ~~2013年纸牌屋第一季正式开播~~

Malloc-maleficarum

- ~~House of Prime~~
- ~~House of Mind~~
- House of Force
- ~~House of Lore~~
- House of Spirit

House of Force

在使用Top chunk进行分配的时候：

```
use_top:
   victim = av->top;
   size = chunksize (victim);   //top块的大小
   if ((unsigned long) (size) >= (unsigned long) (nb + MINSIZE))  //nb为需要分配的块大小
    {
     remainder_size = size - nb;
     remainder = chunk_at_offset (victim, nb);
     av->top = remainder;
     set_head (victim, nb | PREV_INUSE |
          (av != &main_arena ? NON_MAIN_ARENA : 0));
     set_head (remainder, remainder_size | PREV_INUSE);

     check_malloced_chunk (av, victim, nb);
     void *p = chunk2mem (victim);
     alloc_perturb (p, bytes);
     return p;
    }
```

House of Force

利用条件：

- 能够覆写top chunk的size字段
- 存在一次malloc，攻击者能控制malloc的size
- 存在另一次malloc，攻击者能向此chunk写入数据

House of Force

example

- 我们想让top chunk指向0x804A058（heap_list - 8）
- 于是下一次分配，我们得以更改heap_list中的指针

- top chunk = 0x804b020
- 0x804A058 - 0x804b020 = -0xFC8 = 0xfffff038
- so we request 0xfffff038 – 4 = 0xfffff034 = -0xfcc

House of Spirit

当chunk被free时，glibc在想什么

- glibc如何确定被free的是一个真正的chunk？
- 如果我们free(0xdeadbeef)呢？

```
if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0)
    || __builtin_expect (misaligned_chunk (p), 0))
  {
    errstr = "free(): invalid pointer";
```

```
if (__builtin_expect (size < MINSIZE || !aligned_OK (size), 0))
  {
    errstr = "free(): invalid size";
```

如果我们想要一个fake chunk成功进入free list，需要哪些条件？

- 以进入fastbin为例

House of Spirit

如果我们想要一个fake chunk成功进入free list，需要哪些条件？

- 以进入fastbin为例

```
  if (__builtin_expect (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ, 0)
|| __builtin_expect (chunksize (chunk_at_offset (p, size))
         >= av->system_mem, 0))
```

```
/* Check that the top of the bin is not the record we are going to add
   (i.e., double free).  */
if (__builtin_expect (old == p, 0))
  {
    errstr = "double free or corruption (fasttop)";
```

House of Spirit

攻击的本质：free anywhere

- 能够控制free的参数
- 能够满足free中对chunk的各项检查
- 触发free，使得fake chunk进入list
- 再次malloc得到该chunk，相当于弱化版的任意地址写

House of Mind

在free函数中

```
ar_ptr = arena_for_chunk (p);
_int_free (ar_ptr, p, 0);

#define arena_for_chunk(ptr) \
(chunk_non_main_arena (ptr) ? heap_for_ptr (ptr)->ar_ptr : &main_arena)

#define heap_for_ptr(ptr) \
((heap_info *) ((unsigned long) (ptr) & ~(HEAP_MAX_SIZE - 1)))
```

- arena_for_chunk 返回当前chunk对应的arena结构
- 若当前chunk不属于main_arena，则计算指针的值
  - p = 0x804a000
  - (0x804a000 & ~(HEAP_MAX_SIZE - 1)) -> ar_ptr = (0x8000000)

    -> ar_ptr

House of Mind

如果我们能控制指针p以及对应的chunk size，那么就相当于将arena指向了我们控制的区域

- 查找一下free中使用到arena的地方

```
    bck = unsorted_chunks(av);
    fwd = bck->fd;

    p->fd = fwd;
    p->bk = bck;
    if (!in_smallbin_range(size))
{
  p->fd_nextsize = NULL;
  p->bk_nextsize = NULL;
}
    bck->fd = p;
    fwd->bk = p;
```

House of Mind

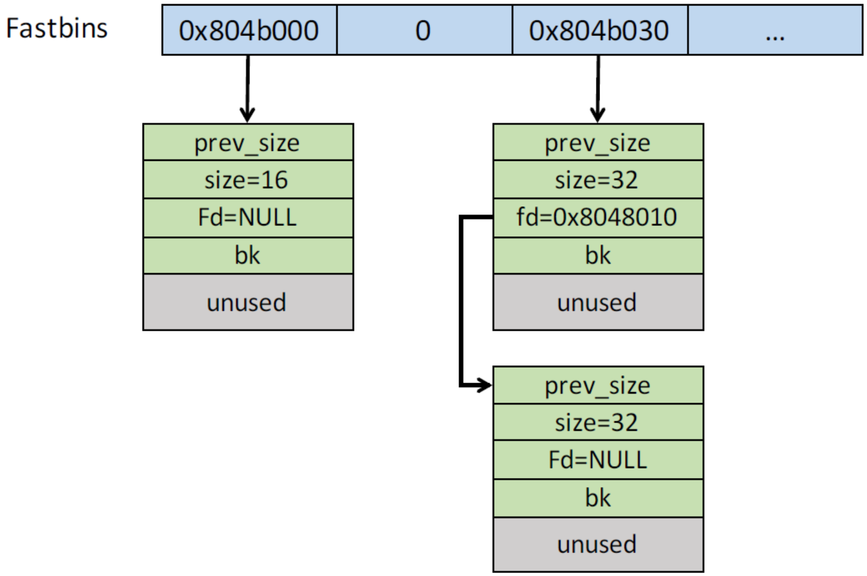
实际上，这个方法已经不能使用了

```c
    bck = unsorted_chunks(av);
    fwd = bck->fd;
    if (__builtin_expect (fwd->bk != bck, 0))
{
  errstr = "free(): corrupted unsorted chunks";
  goto errout;
}
    p->fd = fwd;
    p->bk = bck;
    if (!in_smallbin_range(size))
{
  p->fd_nextsize = NULL;
  p->bk_nextsize = NULL;
}
    bck->fd = p;
    fwd->bk = p;
```

目录 content

Fastbin Attack

fastbin 的单链表结构



Fastbin Attack

XCTF

fastbin检查（in free）

- address < -size && address alignment
- size > MINSIZE(0x20) && multiple of 0x10
- nextchunk size
    - > MINSIZE
    - < system_mem(0x21000 as usually)
- <u>链表中的第一个chunk是否和当前被free的chunk相同</u>

Fastbin Attack

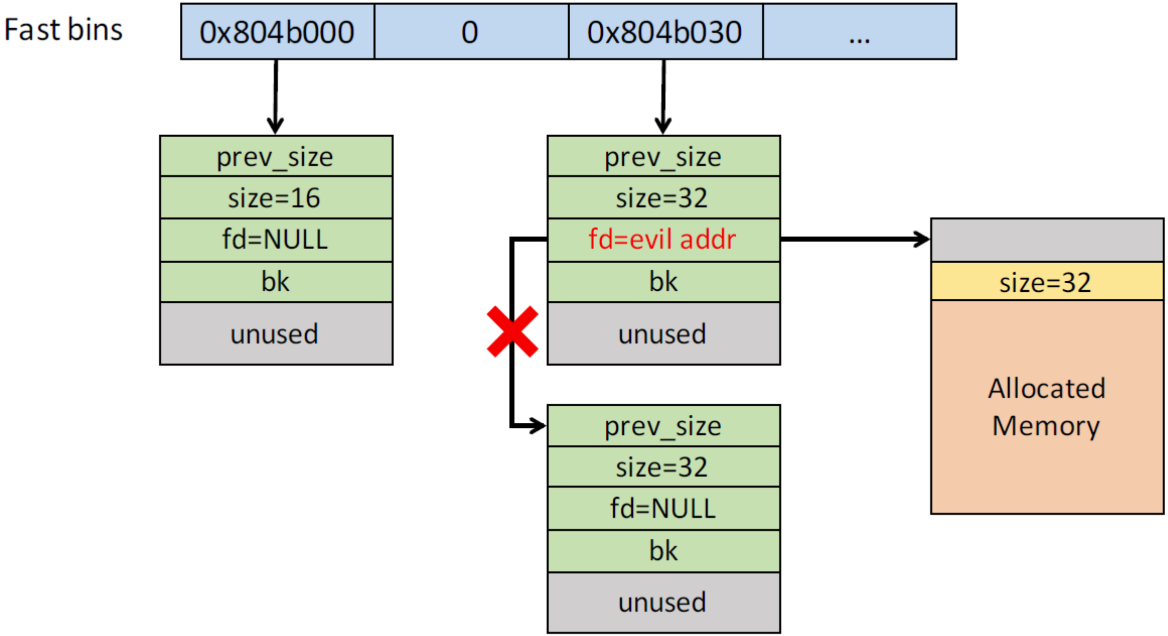
fastbin检查（in malloc）

- chunk size 是否和当前链表匹配

```
if (__builtin_expect (fastbin_index (chunksize (victim)) != idx, 0))
  {
    errstr = "malloc(): memory corruption (fast)";
```

```
#define fastbin_index(sz) \
  ((((unsigned int) (sz)) >> (SIZE_SZ == 8 ? 4 : 3)) - 2)
```

Fastbin Attack



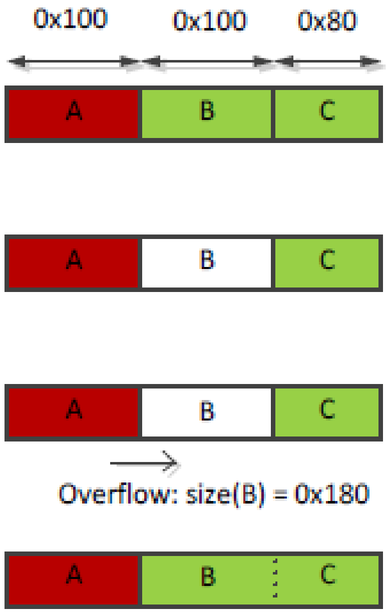
Off by One

XCTF

如果我们仅仅只能够溢出一个字节？

- 这一个字节将影响到下一个chunk的size字段
- Off by One to extend allocated chunk
- Off by Null to shrink free chunk

Extending Allocated Chunk



Initial state

B is freed

Overflow into B

Allocation larger than B’s initial size
C is overlapped

Shrinking Free Chunk

Shrinking Free Chunk

```
if (in_smallbin_range (nb) &&
    bck == unsorted_chunks (av) &&
    victim == av->last_remainder &&
    (unsigned long) (size) > (unsigned long) (nb + MINSIZE))
  {
    /* split and reattach remainder */
    //由于size字段被溢出为一个更小的值，因此导致remainder_size也偏小
    remainder_size = size - nb;
    remainder = chunk_at_offset (victim, nb);
    ...
    set_head (victim, nb | PREV_INUSE |
              (av != &main_arena ? NON_MAIN_ARENA : 0));
    set_head (remainder, remainder_size | PREV_INUSE);
    //由于此处remainder_size小于正常值，因此将不能写入chunk C的prev_size字段
    set_foot (remainder, remainder_size);
    ...
  }
```

Shrinking Free Chunk

demo

shrink.c

Other Things

- 堆溢出的利用可能会同时用到多个技巧
- 将某些漏洞转化成堆溢出
- 例如整数溢出、竞态条件

```
len = read_num();
this -> msg_len = len;
// unsigned overflow
len += 16;
p = new char[len];
this -> msg = p;
```

Other Things

- 竞态条件（Race Condition）
  - 临界区没有正确同步

```
                                       // destination is a global variable
void append(const number_list* source, number_list* destination) {
  size_t new_count = destination->count + source->count;

  if (source->count == 0 || new_count < destination->count)
    return;

  uint64_t* old_numbers = destination->numbers;
  destination->numbers = malloc(new_count * sizeof(uint64_t));

  if (destination->numbers == NULL)
    return;

  if (destination->count) {
    memcpy(destination->numbers, old_numbers,
           destination->count * sizeof(uint64_t));
    free(old_numbers);
  }

  memcpy(destination->numbers + destination->count, source->numbers,
         source->count * sizeof(uint64_t));

  destination->count = new_count;
}
```

Other Things

- 竞态条件（Race Condition）
  - 临界区没有正确同步

What happend	`dest->count`	`dest->numbers`
A malloc(a)	0	buf_a
A finished	a	buf_a
B malloc(a+b)	a	buf_ab
C malloc(a+c)	a	buf_ac
B finished	a+b	buf_ac
C at line 143: `memcpy(buf_ac, buf_ab, a+b)`	a+c	buf_ac

Other Things

XCTF

- Heap Spray
- 0x0c0c0c0c这个地址的优越性
- 当我们能够控制某个虚表指针：
    - mov ebx，[ecx]
    - mov eax，[ebx + 8]
    - call eax
- if eax == 0x41414141?
    - segment fault
- if eax == 0x0c0c0c0c?
    - the same

Other Things

- 堆喷：Unlimited Malloc Works
- eg. 浏览器
- 使得内存空间内充满 nop + shellcode 的堆空间

<table>
<tr><th>Address</th><th colspan="6">Contents</th></tr>
<tr><td></td><td colspan="6"></td></tr>
<tr><td>0c080018</td><td>0x1000 bytes
Nops | shellcode</td><td>0x1000 bytes
Nops | shellcode</td><td>0x1000 bytes
Nops | shellcode</td><td>0x1000 bytes
Nops | shellcode</td><td>...</td><td>0x1000 bytes
Nops | shellcode</td></tr>
<tr><td>0c090018</td><td>0x1000 bytes
Nops | shellcode</td><td>0x1000 bytes
Nops | shellcode</td><td>0x1000 bytes
Nops | shellcode</td><td>0x1000 bytes
Nops | shellcode</td><td>...</td><td>0x1000 bytes
Nops | shellcode</td></tr>
<tr><td>0c0a0018</td><td>0x1000 bytes
Nops | shellcode</td><td>0x1000 bytes
Nops | shellcode</td><td>0x1000 bytes
Nops | shellcode</td><td>0x1000 bytes
Nops | shellcode</td><td>...</td><td>0x1000 bytes
Nops | shellcode</td></tr>
<tr><td>0c0b0018</td><td>0x1000 bytes
Nops | shellcode</td><td>0x1000 bytes
Nops | shellcode</td><td>0x1000 bytes
Nops | shellcode</td><td>0x1000 bytes
Nops | shellcode</td><td>...</td><td>0x1000 bytes
Nops | shellcode</td></tr>
<tr><td>0c0c0018</td><td>0x1000 bytes
Nops | shellcode</td><td>0x1000 bytes
Nops | shellcode</td><td>0x1000 bytes
Nops | shellcode</td><td>0x1000 bytes
Nops | shellcode</td><td>...</td><td>0x1000 bytes
Nops | shellcode</td></tr>
</table>

Other Things

- 堆喷：if 0x0c0c0c0c contains 0x0c0c0c0c
  - mov ebx，[ecx]
  - mov eax，[ebx + 8]   # 0x0c0c0c0c
  - call eax   # 0x0c0c0c0c

- \x0c\x0c -> OR AL， 0C （NOP-like instruction）
- why not 0x0d0d0d0d?
  - \x0d \x0d\x0d\x0d\x0d -> OR EAX， 0x0D0D0D0D
  - 5个字节将可能造成对齐方面的问题

To the End

一道题

werewolf

X-Man

thanks