how2heap 问题汇总(下)

pic4xiu / 2019-10-09 09:40:15 / 浏览数 4209 安全技术 二进制安全 顶(0) 踩(0)

接上篇.上一篇见这里 (虽然两篇联系不大).这篇出现的小 demo ,都可以直接调试,供大家参考

困惑我的 small bin 的源码

做 house_of_lore 时,我在看源码时我遇到了一个很费解的问题(注释来自 ctf-wiki)

```
/*
      If a small request, check regular bin.  Since these "smallbins"
      hold one size each, no searching within bins is necessary.
      (For a large request, we need to wait until unsorted chunks are
      processed to find best fit. But for small ones, fits are exact
      anyway, so we can check now, which is faster.)
    */

   if (in_smallbin_range(nb)) {
       // ■■ small bin ■■■
       idx = smallbin_index(nb);
       // ■■■■ small bin ■■ chunk ■■
       bin = bin_at(av, idx);
       // ■■■ victim= last(bin)■■■ small bin ■■■■■ chunk
       // ■■ victim = bin ■■■■■ bin ■■■
       // ■■■■■■■■■■■■■■
       if ((victim = last(bin)) != bin) {
           // ■■■■■■■small bin ■■■■■■■■
           if (victim == 0) /* initialization check */
               // ■■■■■■■■ fast bins ■■ chunk ■■■■
               malloc_consolidate(av);
           // ■■■■■■■small bin ■■■■■■■ chunk
           else {
               // ■■ small bin ■■■■■■■ chunk ■
               bck = victim->bk;
               // ■■ bck->fd ■■■■ victim■■■■■■
               if (__glibc_unlikely(bck->fd != victim)) {
                   errstr = "malloc(): smallbin double linked list corrupted";
                   goto errout;
               }
               // ■■ victim ■■■■ inuse ■
               set_inuse_bit_at_offset(victim, nb);
               // ■■ small bin ■■■■■ small bin ■■■■■■ chunk ■■■■
               bin->bk = bck;
               bck->fd = bin;
               // ■■■■■ main_arena■■■■■■■■■
               if (av != &main_arena) set_non_main_arena(victim);
               // ■■■■■■
               check_malloced_chunk(av, victim, nb);
               // ■■■■■■ chunk ■■■■■■■ mem ■■
               void *p = chunk2mem(victim);
               // ■■■■■■ perturb_type , ■■■■■■■chunk■■■■■ perturb_type ^ 0xff
               alloc_perturb(p, bytes);
               return p;
           }
       }
   }
```

其中的`bck = victim->bk;`把我整蒙了.ei??既然找到最后一个(victim),那倒数第二个不就应该是 victim->fd 吗??怎么是找 victim->bk 呢.事实上,这就要考虑到 smallbins 的 FIFO (先进先出)原则,在本程序中

```
pwndbg> x/32gx victim-2
0x602000:   0x0000000000000000  0x0000000000000071
0x602010:   0x00007ffff7dd1bd8  0x00007fffffffddf0
...
pwndbg> p stack_buffer_1
$1 = {0x0, 0x0, 0x602000, 0x7fffffffddd0}
```

```
pwndbg> p stack_buffer_2
$2 = {0x0, 0x0, 0x7fffffffddf0}
pwndbg> p &stack_buffer_1
$3 = (intptr_t *(*)[4]) 0x7fffffffddf0
pwndbg> p &stack_buffer_2
$4 = (intptr_t *(*)[3]) 0x7fffffffddd0
```

通过构造情况我们画一下图(表格)

victim_hdr	pre_size	size
victim_ptr	fd	&stack_buffer_1_hdr
...	...	...
stack_buffer_1_hdr	pre_size	size
stack_buffer_1_ptr	&victim_hdr	&stack_buffer_2_hdr
...	...	...
stack_buffer_2_hdr	pre_size	size
stack_buffer_2_ptr	&stack_buffer_1_hdr	bk
...	...	...

所谓的最后一个其实是 victim_hdr (我们伪造的链是从后向前伪造的),关键满足的 bypass 条件便是 `chunk->bk->fd==chunk` 即可,程序第一次 malloc(p3) 的时候是最后一个即 0x602010 要验证的便是 `victim->bk->fd==victim` ,把程序具体变量放进去就是 `stack_buffer_1->fd==victim` .同理,之后的 malloc(p4) 便是 `stack_buffer_2->fd==stack_buffer_1` ,确实成立后自然就 malloc 出来了

关于 poison_null_byte 的思考

做这个的时候我感觉这个 bypass 条件有点过于简单,既然条件为 `prev_size(nextchunk(P))==chunksize(P)` ,只要能够利用溢出修改 chunksize(P) ,我们的 prev_size(nextchunk(P)) 也是由我们控制的,那我在下方任意一处伪造一个 fake_nextchunk 不就行了吗??测试程序如下

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <malloc.h>

int main(void){
   uint8_t* a = (uint8_t*) malloc(1);
   int real_a_size = malloc_usable_size(a);
   void * B = malloc(0x150);
   void * C = malloc(0x150);
   malloc(0x10);
   free(B);
   a[real_a_size] = 0x40;
   *(size_t*)(B+0x130) = 0x140;
   void * D = malloc(0x80);
   void * E = malloc(0x10);
   free(D);
   free(C);
   C = malloc(0x2b0);
   D = malloc(0x90);
}
```

可以看到,我在`a[real_a_size] = 0x40;`和`*(size_t*)(B+0x130) = 0x140;`构造了和 poison_null_byte 一样的条件.断在 malloc(D) 之前,让程序跑起来

```
0x602160:   0x0000000000000140  0x0000000000000000
0x602170:   0x0000000000000000  0x0000000000000000
0x602180:   0x0000000000000160  0x0000000000000160
0x602190:   0x0000000000000000  0x0000000000000000
pwndbg> p C
$3 = (void *) 0x602190
```

之后在 n 单步运行,发现程序果真修改的是我们伪造的 0x602160 处的地址

```
0x602160:   0x00000000000000b0  0x0000000000000000
0x602170:   0x0000000000000000  0x0000000000000000
0x602180:   0x0000000000000160  0x0000000000000160
```

事实上,把 prev_size(nextchunk(P)) 改到 C 的下方也是可以的(修改一下上方程序调试一下).所以结论便是:修改的 0x602160 (prev_size(nextchunk(P)))是通过现在的 B 的 size 找到的,然而在 free(D) 和 free(C) 后 chunk 之间的合并竟然利用的是 C 的 pre_size 来找到的之前的 B (虽然听上去很矛盾,但是 pre_size 设计出来的目的确实是如此,参见此处).所以只要不修改 real_c_presize 无论如何都是能够完成 chunk 的合并的.

其实通过这个 bypass 还可以挖掘一些骚操作,比如通过把 chunksize(P) 改大(要求有溢出漏洞)我们甚至可以通过切割 unsortedbin 的方式获得一个新的 ptr_c ,用来完成一个变向的 UAF 等等

about unsortedbin

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

int main() {
 intptr_t stack_buffer[4] = {0};
 intptr_t stack_buffer_1[4] = {0};
 intptr_t* victim = malloc(0x100);
 intptr_t* p1 = malloc(0x100);
 free(victim);
 stack_buffer[1] = 0x110;
 stack_buffer[3] = (intptr_t)stack_buffer_1;
 stack_buffer_1[1] = 0x100 + 0x10;
 stack_buffer_1[3] = (intptr_t)stack_buffer_1;
 victim[-1] = 32;
 victim[1] = (intptr_t)stack_buffer;
 fprintf(stderr, "malloc(0x100): %p\n", malloc(0x100));
}
```

为了加深理解,我写了一个这样的小 demo ,可以看到,我故意把链加长了,有一个 check 我们需要注意一下,即

```
fprintf(stderr, "Size should be different from the next request size to return fake_chunk and need to pass the check 2*SIZE_SZ
```

大小必须让它和下一个要求的不同,且大于 2*SIZE_SZ ,同时还必须小于已分配内存的大小.好,我们让程序跑起来,断在最后一个 malloc(0x100) 处.尽管 stack_buffer 和 stack_buffer_1 差不多,但是程序在 malloc(0x100) 的时候还是会选择 stack_buffer

```
pwndbg> x/8gx stack_buffer
0x7fffffffddd0: 0x0000000000000000  0x0000000000000110      ----stack_buffer
0x7fffffffdde0: 0x0000000000000000  0x00007fffffffddf0
0x7fffffffddf0: 0x0000000000000000  0x0000000000000110      ----stack_buffer_1
0x7fffffffde00: 0x0000000000000000  0x00007fffffffddf0
pwndbg> n
malloc(0x100): 0x7fffffffdde0
```

然而当把 `stack_buffer[1] = 0x110;` 中的 0x110 改成别的,这时候再 malloc(0x100) 才会使用我们之后构造的 stack_buffer_1

```
11    stack_buffer[1] = 0x100;      --------■■■ 0x100
12    stack_buffer[3] = (intptr_t)stack_buffer_1;
13    stack_buffer_1[1] = 0x110;
14    stack_buffer_1[3] = (intptr_t)stack_buffer_1;

pwndbg> x/8gx stack_buffer
0x7fffffffddd0: 0x0000000000000000  0x0000000000000100
0x7fffffffdde0: 0x0000000000000000  0x00007fffffffddf0
0x7fffffffddf0: 0x0000000000000000  0x0000000000000110
0x7fffffffde00: 0x0000000000000000  0x00007fffffffddf0
pwndbg> n
malloc(0x100): 0x7fffffffde00
```

同时经过遍历的 stack_buffer 和一开始真的 chunk 即 victim 也因为能力不足(上篇中形象的概念),跑到了 smallbins.当然能能不能用另说,2333

```
pwndbg> bins
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all [corrupted]
FD: 0x602000 —■ 0x7ffff7dd1b88 (main_arena+104) ■— 0x602000
BK: 0x7fffffffddf0 ■— 0x7fffffffddf0
smallbins
```

```
0x20: 0x602000 —■ 0x7ffff7dd1b88 (main_arena+104) ■— 0x602000
0x100: 0x7fffffffddd0 —■ 0x7ffff7dd1c68 (main_arena+328) ■— 0x7fffffffddd0
largebins
empty
```

所以 unsortedbin 是从头开始遍历,途中遇到的能力不足的 unsortedbin 都会被安排到对应的 bins
中,而一旦有合适的就停止遍历并使用,为什么说是停止遍历呢??可以调试一下一开始的程序,看程序在最后一个 malloc(0x100) 之后的 bins

```
pwndbg> bins
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all [corrupted]
FD: 0x602000 —■ 0x7ffff7dd1b88 (main_arena+104) ■— 0x602000
BK: 0x7fffffffddf0 ■— 0x7fffffffddf0
smallbins
0x20: 0x602000 —■ 0x7ffff7dd1b88 (main_arena+104) ■— 0x602000
largebins
empty
```

相当于只把一开始的 victim 给并入了 smallbins ,而 stack_buffer_1 还是待在 unsortedbin 中

总结

剩下的一些问题,在网上各位师傅的分析中已经很明了了.本篇的初衷就是扣一些易犯的错和问题,越是遇到难的诸如 `house of orange` 等问题大家就越深入分析,本篇也就不再谈了(而且感觉自己也不能表达的很清楚). how2heap 中的大部分都是欺骗系统的一系列 bypass 和构造,提升的方式就是做题和看源码了

上一篇：PwnThyBytes CTF 2... 下一篇：ThinkPHP5.2.x反序列化利用链

1. 0 条回复
   - 动动手指，沙发就是你的了！