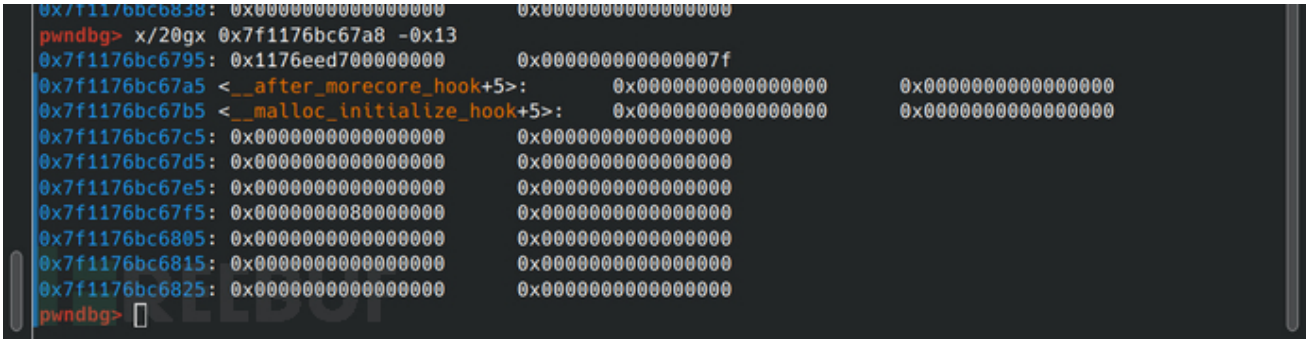
0x00 写在前面

发freebuf了：https://www.freebuf.com/articles/endpoint/373258.html

本次阅读源码是本人第一次，算是一个全新的开始。本次看源码是为了调试roarctf的babyheap那道题目，wp写在独奏者2 序章那篇的0x04（在我个人博客里），为了看看为什么free_hook-0x13不能分配堆。

让我们先看看是怎么回事。



此时在fastbin构建了这样一个堆块，按理来说可以分配到的，但是——

这就不得不深挖一下了。

0x01 阅读前言和别名

搜索alias

```
1  weak_alias (__malloc_info, malloc_info)
2
3  strong_alias (__libc_calloc, __calloc) weak_alias (__libc_calloc, calloc)
4
5  strong_alias (__libc_free, __cfree) weak_alias (__libc_free, cfree)
6
7  strong_alias (__libc_free, __free) strong_alias (__libc_free, free)
8
9  strong_alias (__libc_malloc, __malloc) strong_alias (__libc_malloc, malloc)
10
11 strong_alias (__libc_memalign, __memalign)
12
13 weak_alias (__libc_memalign, memalign)
14
15 strong_alias (__libc_realloc, __realloc) strong_alias (__libc_realloc, realloc)
16
17 strong_alias (__libc_valloc, __valloc) weak_alias (__libc_valloc, valloc)
18
```

```
strong_alias (__libc_pvalloc, __pvalloc) weak_alias (__libc_pvalloc, pvalloc)

strong_alias (__libc_mallinfo, __mallinfo)

weak_alias (__libc_mallinfo, mallinfo)

strong_alias (__libc_mallopt, __mallopt) weak_alias (__libc_mallopt, mallopt)

weak_alias (__malloc_stats, malloc_stats)

weak_alias (__malloc_usable_size, malloc_usable_size)

weak_alias (__malloc_trim, malloc_trim)

weak_alias (__malloc_get_state, malloc_get_state)

weak_alias (__malloc_set_state, malloc_set_state)
```

发现__libc_malloc和__malloc是一个东西。

阅读前言（学英语了）

```
/* Malloc implementation（分配器/实施方案） for multiple（多种） threads without lock
contention.

Copyright (C) 1996-2016 Free Software Foundation, Inc.

This file is part of the GNU C Library.

Contributed by Wolfram Gloger <wg@malloc.de>

and Doug Lea <dl@cs.oswego.edu>, 2001.

（多种没有锁链接的一种malloc的实施方法）



The GNU C Library is free software; you can redistribute（重新分配） it and/or

modify（修改） it under the terms(关系) of the GNU Lesser General Public License as

published by the Free Software Foundation; either version 2.1 of the

License, or (at your option) any later version.

（我们可以瞎改）

The GNU C Library is distributed in the hope that it will be useful,（希望有用）

but WITHOUT ANY WARRANTY（但是没有任何依据）; without even the implied warranty of
（甚至没有默示担保）

MERCHANTABILITY（适销性） or FITNESS FOR A PARTICULAR PURPOSE. See the GNU

Lesser General Public License for more details.

（有用但是没啥担保）
```

```
34
35
36
37  You should have received a copy of the GNU Lesser General Public
38
39  License along with the GNU C Library; see the file COPYING.LIB. If
40
41  not, see <http://www.gnu.org/licenses/>. */
42
43  //（应该收到了一个凭证，收不收都行对我来说）
44
45  /*
46
47  This is a version (aka ptmalloc2) of malloc/free/realloc written by
48
49  Doug Lea and adapted to multiple threads/arenas by Wolfram Gloger.
50
51  （这是ptmalloc2 的一个版本，这俩人写的）
52
53  There have been substantial（大量的，基本的，重要的） changes made after the
    integration（整合） into
54
55  glibc in all parts of the code. Do not look for much commonality
56
57  with the ptmalloc2 version.
58
59  （因为整合到glibc里所以改了很多，所以找不到太多和原本ptmalloc2版本共性）
60
61
62
63  * Version ptmalloc2-20011215
64
65  based on:
66
67  VERSION 2.7.0 Sun Mar 11 14:14:06 2001 Doug Lea (dl at gee)
68
69
70
71  （版本信息，2001年，够老了哈）
72
73  * Quickstart
74
75  （正片开始？草）
76
77
78
79  In order to compile（整合） this implementation（分配器）, a Makefile is provided
    with
80
81  the ptmalloc2 distribution, which has pre-defined targets for some
82
83  popular systems (e.g. "make posix" for Posix threads).
84
85  （为了整合给了个makefile，匹配很多常用版本的系统）
86
87
88
```

```
89  All that is typically required with regard (认为) to compiler flags is the
    selection of
90
91  the thread (贯穿主线) package via defining one out of USE_PTHREADS, USE_THR or
92
93  USE_SPROC. Check the thread-m.h file for what effects this has.
94
95  Many/most systems will additionally require USE_TSD_DATA_HACK to be
96
97  defined, so this is the default for "make posix".
98
99   (解释了一下配置文件的方法)
100
101
102
103 * Why use this malloc?
104
105
106
107 This is not the fastest, most space-conserving (节省空间的), most portable (最轻便
    的, 可移植的), or
108
109 most tunable (最和谐的) malloc ever written. However it is among the fastest
110
111 while also being among the most space-conserving, portable and tunable.
112
113  (他不是最节省空间, 最可移植, 嘴和写的, 但是他是其中之一? 离谱)
114
115 Consistent balance across these factors (因素) results in a good general-purpose
116
117 allocator for malloc-intensive programs.
118
119  (这些因素的持续的平衡, 最终造就了这是一个给malloc深入细致的程序的通用性很好的分配方式。)
120
121
122
123 The main properties of the algorithms are:
124
125  (算法主要的性能)
126
127 * For large (>= 512 bytes) requests, it is a pure best-fit allocator,
128
129 with ties normally decided via FIFO (i.e. least recently used).
130
131  (对大的来说最合适不过了! )
132
133 * For small (<= 64 bytes by default) requests, it is a caching (缓冲)
134
135 allocator, that maintains (保持, 维持) pools of quickly recycled chunks.
136
137  (对小的来说也不错! 可以作为一个缓冲, 循环利用堆块! )
138
139 * In between, and for combinations of large and small requests, it does
140
141 the best it can trying to meet both goals at once.
142
143  (在二者之间的也不错! 因为他会尝试两个尽可能都达到)
144
```

```
* For very large requests (>= 128KB by default), it relies on system

memory mapping facilities, if supported.

 (太大的话要看系统支不支持了呜呜)



For a longer but slightly out of date high-level description, see

http://gee.cs.oswego.edu/dl/html/malloc.html



You may already by default be using a C library containing a malloc

that is based on some version of this malloc (for example in

linux). You might still want to use the one in this file in order to

customize (定制) settings or to avoid overheads (额外开支) associated with library

versions.

 (你可能是默认用的，也可能是想定制，也可能是图个免费。)

* Contents, described in more detail in "description of public routines (常规) "
below.



Standard (ANSI/SVID/...) functions:

 (标准函数来了)

malloc(size_t n);

calloc(size_t n_elements, size_t element_size);

free(void* p);

realloc(void* p, size_t n);

memalign(size_t alignment, size_t n);

valloc(size_t n);

mallinfo()

mallopt(int parameter_number, int parameter_value)



Additional functions:

(扩展的有这些)

independent_calloc(size_t n_elements, size_t size, void* chunks[]);
```

```
202
203 independent_comalloc(size_t n_elements, size_t sizes[], void* chunks[]);
204
205 pvalloc(size_t n);
206
207 cfree(void* p);
208
209 malloc_trim(size_t pad);
210
211 malloc_usable_size(void* p);
212
213 malloc_stats();
214
215
216
217 * Vital statistics:
218
219  (至关重要的统计数字)
220
221
222
223 Supported pointer representation: 4 or 8 bytes
224
225 Supported size_t representation: 4 or 8 bytes
226
227 Note that size_t is allowed to be 4 bytes even if pointers are 8.
228
229 You can adjust this by defining INTERNAL_SIZE_T
230
231
232
233 Alignment: 2 * sizeof(size_t) (default)
234
235 (i.e., 8 byte alignment with 4byte size_t). This suffices for
236
237 nearly all current machines and C compilers. However, you can
238
239 define MALLOC_ALIGNMENT to be wider than this if necessary.
240
241
242
243 Minimum overhead per allocated chunk: 4 or 8 bytes
244
245 Each malloced chunk has a hidden word of overhead holding size
246
247 and status information.
248
249
250
251 Minimum allocated size: 4-byte ptrs: 16 bytes (including 4 overhead)
252
253 8-byte ptrs: 24/32 bytes (including, 4/8 overhead)
254
255
256
257 When a chunk is freed, 12 (for 4byte ptrs) or 20 (for 8 byte
258
259 ptrs but 4 byte size) or 24 (for 8/8) additional bytes are
```

```
260
261 needed; 4 (8) for a trailing size field and 8 (16) bytes for
262
263 free list pointers. Thus, the minimum allocatable size is
264
265 16/24/32 bytes.
266
267
268
269 Even a request for zero bytes (i.e., malloc(0)) returns a
270
271 pointer to something of the minimum allocatable size.
272
273
274
275 The maximum overhead wastage (i.e., number of extra bytes
276
277 allocated than were requested in malloc) is less than or equal
278
279 to the minimum size, except for requests >= mmap_threshold that
280
281 are serviced via mmap(), where the worst case wastage is 2 *
282
283 sizeof(size_t) bytes plus the remainder from a system page (the
284
285 minimal mmap unit); typically 4096 or 8192 bytes.
286
287
288
289 Maximum allocated size: 4-byte size_t: 2^32 minus about two pages
290
291 8-byte size_t: 2^64 minus about two pages
292
293
294
295 It is assumed that (possibly signed) size_t values suffice to
296
297 represent chunk sizes. `Possibly signed' is due to the fact
298
299 that `size_t' may be defined on a system as either a signed or
300
301 an unsigned type. The ISO C standard says that it must be
302
303 unsigned, but a few systems are known not to adhere to this.
304
305 Additionally, even when size_t is unsigned, sbrk (which is by
306
307 default used to obtain memory from system) accepts signed
308
309 arguments, and may not be able to handle size_t-wide arguments
310
311 with negative sign bit. Generally, values that would
312
313 appear as negative after accounting for overhead and alignment
314
315 are supported only via mmap(), which does not have this
316
317 limitation.
```

```
318
319
320
321 Requests for sizes outside the allowed range will perform an optional
322
323 failure action and then return null. (Requests may also
324
325 also fail because a system is out of memory.)
326
327
328
329 Thread-safety: thread-safe
330
331
332
333 Compliance: I believe it is compliant with the 1997 Single Unix Specification
334
335 Also SVID/XPG, ANSI C, and probably others as well.
336
337
338
339 * Synopsis of compile-time options:
340
341
342
343 People have reported using previous versions of this malloc on all
344
345 versions of Unix, sometimes by tweaking some of the defines
346
347 below. It has been tested most extensively on Solaris and Linux.
348
349 People also report using it in stand-alone embedded systems.
350
351
352
353 The implementation is in straight, hand-tuned ANSI C. It is not
354
355 at all modular. (Sorry!) It uses a lot of macros. To be at all
356
357 usable, this code should be compiled using an optimizing compiler
358
359 (for example gcc -O3) that can simplify expressions and control
360
361 paths. (FAQ: some macros import variables as arguments rather than
362
363 declare locals because people reported that some debuggers
364
365 otherwise get confused.)
366
367
368
369 OPTION DEFAULT VALUE
370
371
372
373 Compilation Environment options:
374
375
```

```
376
377 HAVE_MREMAP 0
378
379
380
381 Changing default word sizes:
382
383
384
385 INTERNAL_SIZE_T size_t
386
387 MALLOC_ALIGNMENT MAX (2 * sizeof(INTERNAL_SIZE_T),
388
389 __alignof__ (long double))
390
391
392
393 Configuration and functionality options:
394
395
396
397 USE_PUBLIC_MALLOC_WRAPPERS NOT defined
398
399 USE_MALLOC_LOCK NOT defined
400
401 MALLOC_DEBUG NOT defined
402
403 REALLOC_ZERO_BYTES_FREES 1
404
405 TRIM_FASTBINS 0
406
407
408
409 Options for customizing MORECORE:
410
411
412
413 MORECORE sbrk
414
415 MORECORE_FAILURE -1
416
417 MORECORE_CONTIGUOUS 1
418
419 MORECORE_CANNOT_TRIM NOT defined
420
421 MORECORE_CLEARS 1
422
423 MMAP_AS_MORECORE_SIZE (1024 * 1024)
424
425
426
427 Tuning options that are also dynamically changeable via mallopt:
428
429
430
431 DEFAULT_MXFAST 64 (for 32bit), 128 (for 64bit)
432
433 DEFAULT_TRIM_THRESHOLD 128 * 1024
```

```
 DEFAULT_TOP_PAD 0

 DEFAULT_MMAP_THRESHOLD 128 * 1024

 DEFAULT_MMAP_MAX 65536



 There are several other #defined constants and macros that you

 probably don't want to touch unless you are extending or adapting malloc. */



 /*

 void* is the pointer type that malloc should say it returns

 */

 //到这上面就是一堆标准了。
```

0x02 找到malloc

```
void *__libc_malloc (size_t bytes)

{

mstate ar_ptr;

void *victim;



void *(*hook) (size_t, const void *)

= atomic_forced_read (__malloc_hook);

if (__builtin_expect (hook != NULL, 0))

return (*hook)(bytes, RETURN_ADDRESS (0));

//这里就是先看看hook有没有，有的话执行



arena_get (ar_ptr, bytes);

//搞一下arena的指针吧



victim = _int_malloc (ar_ptr, bytes);//arena的指针和申请大小，执行intmalloc函数，重点看看

```

```
31
32
33  /* Retry with another arena only if we were able to find a usable arena
34
35  before. */
36
37  if (!victim && ar_ptr != NULL)
38
39  {
40
41  LIBC_PROBE (memory_malloc_retry, 1, bytes);
42
43  ar_ptr = arena_get_retry (ar_ptr, bytes);
44
45  victim = _int_malloc (ar_ptr, bytes);
46
47  }
48
49
50
51  if (ar_ptr != NULL)
52
53  (void) mutex_unlock (&ar_ptr->mutex);
54
55
56
57  assert (!victim || chunk_is_mmapped (mem2chunk (victim)) ||
58
59  ar_ptr == arena_for_chunk (mem2chunk (victim)));
60
61  return victim;
62  }
63
64  libc_hidden_def (__libc_malloc)//这个玩意是延迟绑定用的，用到才绑定地址，节约资源
```

问题可能在intmalloc，或者chunk_is_mmapped (mem2chunk (victim))和ar_ptr == arena_for_chunk (mem2chunk (victim)));的检查。所以仔细过去看看。先去吃饭了等一下。

0x03 看看在哪报的错

```
1  b"*** Error in `./pwn': malloc(): memory corruption (fast): 0x00007fafbedc67a5
   ***\n"
2  *** Error in `./pwn': malloc(): memory corruption (fast): 0x00007fafbedc67a5 ***
```

然后去源码找一下相关的。

```
1  static void *
2
3  _int_malloc (mstate av, size_t bytes)//从vscode上面可以找到
4
5  {
6
7  INTERNAL_SIZE_T nb; /* normalized request size （标准请求的大小）*/
8
9  unsigned int idx; /* associated bin index （bin的索引）*/
```

```
mbinptr bin; /* associated bin （分配的bin）*/

其大小

mchunkptr victim; /* inspected/selected chunk （收集的堆）*/

INTERNAL_SIZE_T size; /* its size （收集堆的大小）*/

int victim_index; /* its bin index （堆的索引？？）*/

//上面是收集堆块

mchunkptr remainder; /* remainder from a split （分裂剩余的部分）*/

unsigned long remainder_size; /* its size （和其大小）*/


unsigned int block; /* bit map traverser */

unsigned int bit; /* bit map traverser */

unsigned int map; /* current word of binmap （一个当前一个横跨）*/


mchunkptr fwd; /* misc temp for linking */

mchunkptr bck; /* misc temp for linking （fd和bk）*/


const char *errstr = NULL;


/*

Convert request size to internal form by adding SIZE_SZ bytes

overhead plus possibly more to obtain necessary alignment and/or

to obtain a size of at least MINSIZE, the smallest allocatable

size. Also, checked_request2size traps (returning 0) request sizes

that are so large that they wrap around zero when padded and

aligned.

*/


checked_request2size (bytes, nb);
```

这个是函数前面的一点定义。

```
if ((unsigned long) (nb) <= (unsigned long) (get_max_fast ()))

{

idx = fastbin_index (nb);//获取fastbin的索引

mfastbinptr *fb = &fastbin (av, idx);

mchunkptr pp = *fb;

do

{

victim = pp;

if (victim == NULL)

break;

}

while ((pp = catomic_compare_and_exchange_val_acq (fb, victim->fd, victim))

!= victim);

if (victim != 0)//如果有分配

{

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

/*如果__builtin_expect (fastbin_index (chunksize (victim)) != idx, 0)为真则报错

找到堆的大小，然后找bin的索引，和0一起传入了__builtin_expect。

所以去看看__builtin_expect


loading……


这个玩意是用来优化代码的，一会会贴一下网址，是在gcc里

如果第二个参数是1，则通常执行if的内容，是0的话通常执行else。

 (第二个参数其实就是期望的意思，函数是建立时的期望)

这样可以节约jmp指令，提升效率。


他真的我哭死，好相信我们，认为我们都会正常分配去执行else
```

```
可是我们在利用他的信任……也就是说报错说明第一个参数返回1了。

也就是说fastbin_index (chunksize (victim)) 和idx不相等。那我们去看看咋回事吧!



总结来说，就是我这个大小的堆的索引和我想申请的不一样。

也就是bin应该存的大小和实际的大小不一样。

*/

{

errstr = "malloc(): memory corruption (fast)";

errout:

malloc_printerr (check_action, errstr, chunk2mem (victim), av);

return NULL;

}

check_remalloced_chunk (av, victim, nb);//否则正常分配

void *p = chunk2mem (victim);

alloc_perturb (p, bytes);

return p;

}

}
```

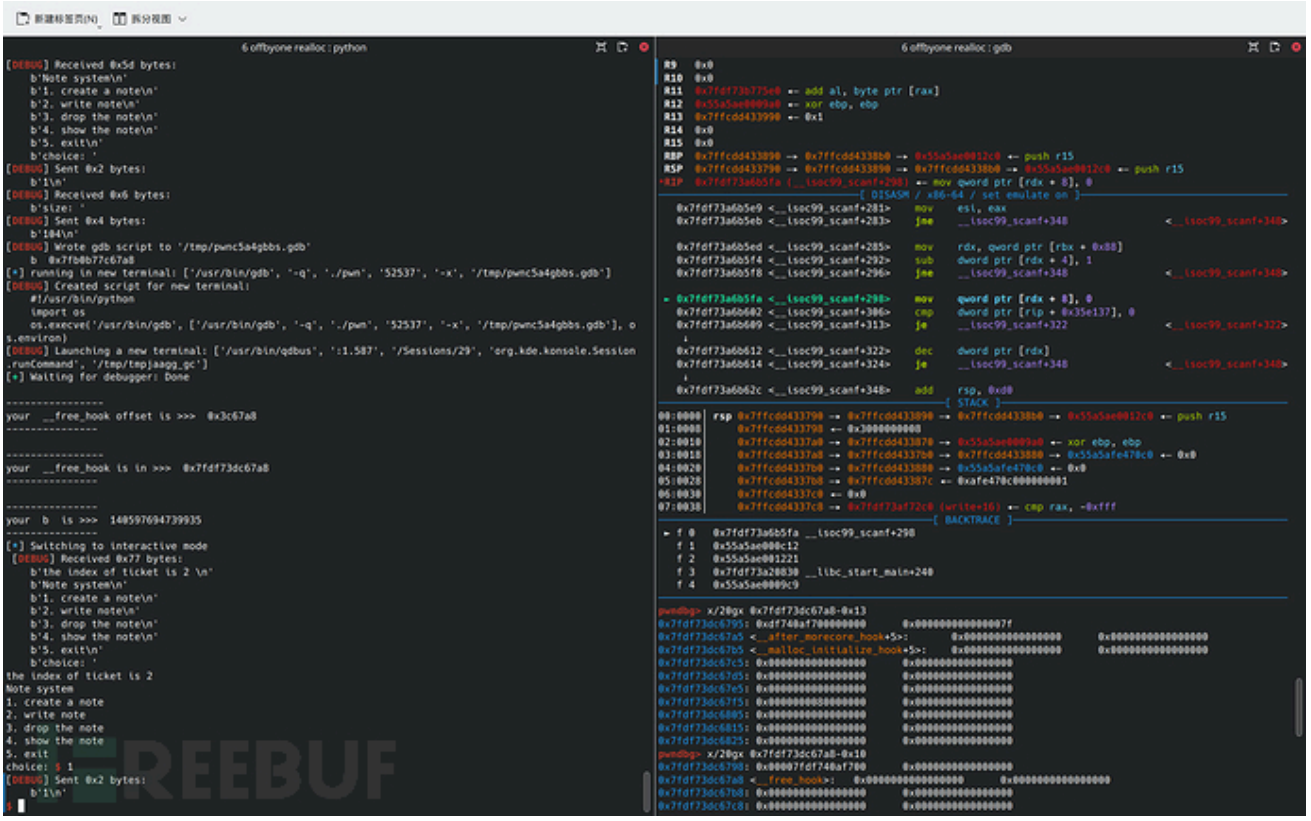
经过一波分析，发现报错是因为，bin应该存的大小和实际的大小不一样。但是显然这不是我们的问题，因为伪造0x7f已经很熟悉了，申请0x68，结果发现再去gdb调试是在申请的时候那段内存突然变成0了，所以才会报错。

0x04 为什么变成0了呢?

我们打断点看看。

```
pwndbg> watch *0x7f97f2fc67a8 -0x13
Hardware watchpoint 2: *0x7f97f2fc67a8 -0x13
```

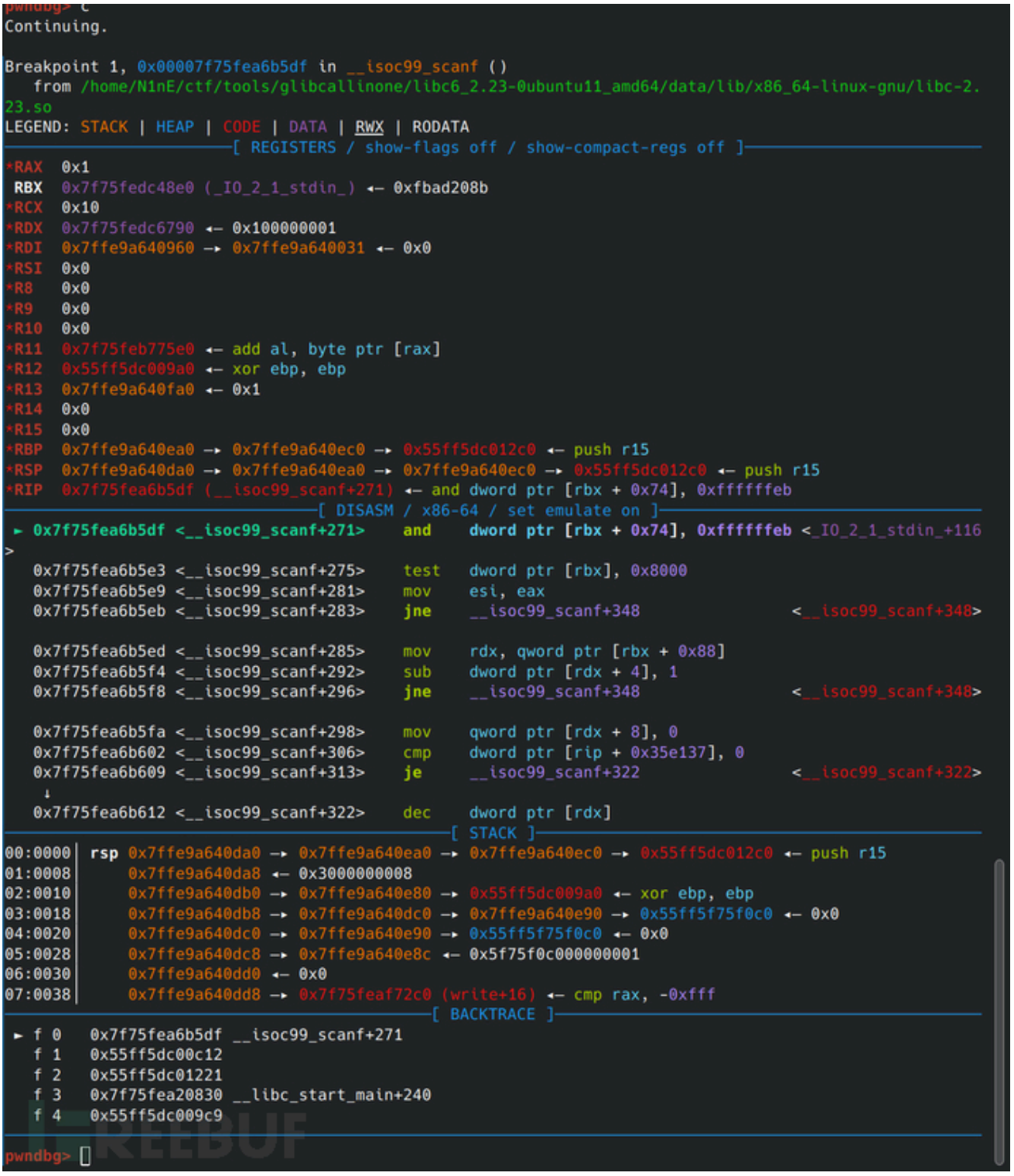
这么久终于用到watch命令了，观察这个地址数据有没有变化。(没啥用，求大佬教教，这个当一个疑问吧)



调了很久终于找到是这里搞的了。

看看rdx怎么来的。

我再看看为什么malloc_hook不受影响。



注意看rdx，不管是fastbin attack到mallochook还是freehook都是一样的。

结果发现这个地方就是用来存参数用的。我们再看看存的是什么。然后发现不管输入什么这里都会变成……1。最后用0覆盖掉。

再运行之后发现每次调用__isoc99_scanf那个地方会被恢复，也就是说这个地址刚好是存有相关参数的地方。

运行的过程中0x0000000100000001 变成0x0000000100000000，执行完__isoc99_scanf后，
0x7fe64dfc6790: 0x0000000100000001 0x00007fe64e40f700(这里是freehook的地址减去0x18)
全部变为0，我认为这是因为存取了某些参数。

0x05 探秘__isoc99_scanf

```
1  int
```

```
attribute_compat_text_section

__nldbl___isoc99_scanf (const char *fmt, ...)

{

va_list arg;

int done;



va_start (arg, fmt);

done = __nldbl___isoc99_vfscanf (stdin, fmt, arg);

va_end (arg);



return done;

}
```

这是__isoc99_scanf的源码，发现没有什么设置参数的东西，但是调用了__nldbl___isoc99_vfscanf。

```
int

attribute_compat_text_section

__nldbl__IO_vfscanf (FILE *s, const char *fmt, _IO_va_list ap,

int *errp)

{

int res;

set_no_long_double ();

res = _IO_vfscanf (s, fmt, ap, errp);

clear_no_long_double ();

return res;

}
```

这是 __nldbl___isoc99_vfscanf的源码，发现可疑函数两个，set_no和clear_no。

然后发现了惊天秘密！

```
#define set_no_long_double() \

__libc_cleanup_push (__nldbl_cleanup, NULL); __no_long_double = 1

#define clear_no_long_double() \

__no_long_double = 0; __libc_cleanup_pop (0)



/* Compatibility with IEEE double as long double.

IEEE quad long double is used by default for most programs, so

we don't need to split this into one file per function for the

sake of statically linked programs. */
```

其实大致就是前面是让他为1并且设置一个函数，后面是让他为0，并且清理掉。破案了破案了。

问问chat哥：

> 感谢提供更多代码片段，让我们来解释一下这些宏和相关的代码：
>
> 1. `set_no_long_double()` 宏： 这个宏用于设置一个名为 `__no_long_double` 的标志，将其值设置为1，并且注册了一个清理函数 `__nldbl_cleanup` 。在设置 `__no_long_double` 为1的同时，会调用 `__libc_cleanup_push` 函数，将 `__nldbl_cleanup` 添加到清理函数栈中。这意味着当函数返回或执行 `__libc_cleanup_pop` 时，会调用 `__nldbl_cleanup` 来清理资源。
> 2. `clear_no_long_double()` 宏： 这个宏用于将 `__no_long_double` 标志的值设置为0，并且通过调用 `__libc_cleanup_pop` 函数从清理函数栈中弹出清理函数。这样做是为了确保在函数返回或退出之前，能够正确地重置 `__no_long_double` 的值。
> 3. 注释部分： 这部分注释提到与 IEEE 双精度浮点数作为长双精度浮点数的兼容性。它指出，对于大多数程序，默认情况下使用 IEEE 四倍精度浮点数作为长双精度浮点数。并且提到为了静态链接的程序的缘故，不需要为每个函数拆分为单独的文件。
>
> 综合以上信息，可以推断这段代码在处理长双精度浮点数时进行了一些处理，可能是为了兼容性或特定环境下的优化。 `__no_long_double` 标志可能用于控制是否使用长双精度浮点数，而 `__libc_cleanup_push` 和 `__libc_cleanup_pop` 则用于确保在函数返回时正确地处理相关资源。

0x06 总结

总结就是free_hook改不了，因为调用scanf的时候会进行一系列操作把我们的fakechunk的头部（尤其是size的位置）清零，直到下一次调用scanf的时候又恢复有清理函数，这也是为什么我们能看到fakechunk——因为我们是按c到那里停住，而停住的时候正是用scanf读取数字的过程，这个时候清理函数是存在的……

第一次读源码，太牛了，有点震撼背后的原理，真有意思。一直从早上干到下午六点，也就是现在。

还有点感冒，中午没睡好，一会去拿某人买的药，嘿嘿。

0x07 尾声

（此处应有流浪地球丫丫的bgm）

> 这里可能存在一些涉及长双精度浮点数的处理逻辑，但没有足够的上下文信息来进一步解释代码的用途和目的。如果您有完整的代码，可以进一步查看相关代码和调用处，以理解更多细节。同时，您还可以参考相关的文档或联系代码的原作者或维护人员，以获取更准确的解释和用途说明。

posted @ 2023-07-27 17:55 .N1nEmAn 阅读(208) 评论(0)