[阅读型]新版musl libc(1.2.2)堆管理之源码剖析！

原创

easyIyou 于 2021-07-23 01:05:52 发布 1737 收藏 10
分类专栏： 二进制安全 文章标签： 内存管理 安全
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/easy_level1/article/details/118606424
版权

二进制安全 专栏收录该内容
7 篇文章 1 订阅
订阅专栏

目录

前言
关键的数据结构
关键代码
- void* malloc(size_t)
- static int alloc_slot(int, size_t)
- static uint32_t try_avail(struct meta **)
- static struct meta * alloc_group(int, size_t)
- void* enframe(struct meta *, int, size_t, int)
- void free(void *)
- static struct mapinfo nontrivial_free(struct meta *, int)

2021 Defcon Quals mooosl
2021 强网杯初赛 easyheap
额外补充 IO_FILE

前言

文章更新时间 `2021.08.23`

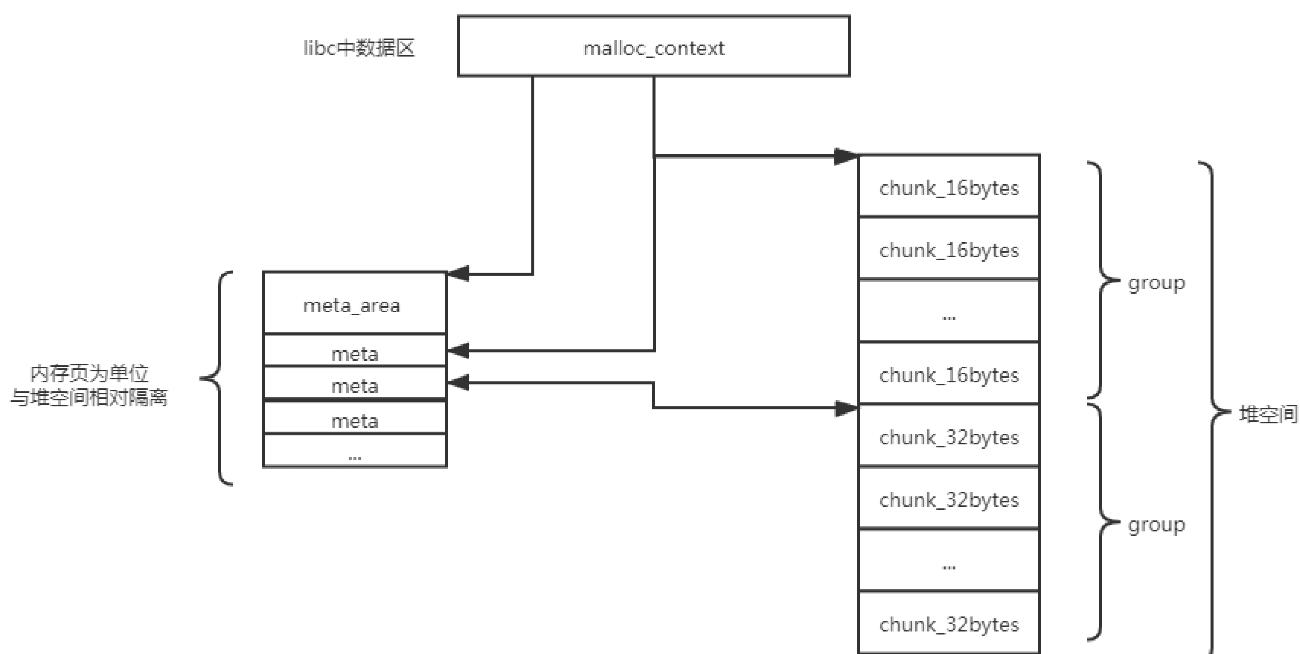
自defcon21q，那道mooosl起，就有些比赛喜欢折腾这新版的musl libc的堆。在当时打defcon的时候，相关资料几乎没有，只能硬磕源码。虽然笔者当时是审出来了 `dequeue()` 会有类似unlink的任意地址写的可能，但是此题还是由大佬亲自出手把它干掉。事后一直懒没有复现，结果强网杯21又出一个musl libc的题。此文章开坑，也是督促自己把这一块源码仔细扣一扣，做一个笔记。下文除了源码剖析，会以defcon21q和强网杯21的真题为例。

注： musl libc在1.2.x以后有新的堆分配算法，与glibc有相当大的不同。尽管如此，本文适合有对glibc堆管理的基本认识的读者。
注：截止目前2021.07，ubuntu20.04默认安装的musl libc `sudo apt install musl`，仍然是老的算法libc1.1.x。老的算法和glibc相当类似，不多赘述。

目前也有一些资料可以参考，比较粗略。
参考资料：
https://www.anquanke.com/post/id/241101
https://f5.pm/go-76812.html

关键的数据结构

下图是一个简单的整体结构图，有四个基本的结构体，分别是`malloc_context`,`meta_area`,`meta`,`group`。与glibc明显不同的是，分配給用户的chunk只是group数组中一个元素。chunk中只有非常少的元数据用作管理，用于管理的数据结构主要在meta里，故musl libc这样的设计就能一定程度防止溢出和UAF对libc堆管理的直接攻击。



```c
//in malloc.c
//定义了一个全局变量，用于管理各个chunk
struct malloc_context ctx = { 0 };

//in glue.h
//在gdb中的符号是 __malloc_context
#define ctx __malloc_context

//in meta.h
__attribute__((__visibility__("hidden")))
extern struct malloc_context ctx;

struct malloc_context {
 uint64_t secret;   //ctx.secret = get_random_secret();
 int init_done;   //if(init_done==0) {init, init_done = 1;}
 unsigned mmap_counter; //使用mmap分配内存的次数
 //---------------------不着急弄明白下面的变量
 struct meta *free_meta_head;
 struct meta *avail_meta; //meta_area中管理的空闲的meta首地址，用avail_meta_count表示数量
 size_t avail_meta_count, avail_meta_area_count, meta_alloc_shift;
 struct meta_area *meta_area_head, *meta_area_tail;
 unsigned char *avail_meta_areas;
 struct meta *active[48]; //缓存可继续分配的meta，数组下标与大小有关。(类似tcache/各种 bins)
 size_t usage_by_class[48]; //对应大小的缓存的所有meta的group所管理的chunk个数。
 uint8_t unmap_seq[32], bounces[32];
 uint8_t seq;
 uintptr_t brk;    //记录目前的brk(0)
};
```

```c
//in meta.h
//单独申请一个内存页，页起始地址为一个struct meta_area结构，该内存页剩下部分就是一个个的meta。
//const struct meta_area *area = (void *)((uintptr_t)meta & -4096);
struct meta_area {
 uint64_t check;   //一个正确的meta_area应有 assert(area->check == ctx.secret);
 struct meta_area *next;

 //管理meta的个数，一般是个固定值
 //ctx.meta_area_tail->nslots = (4096-sizeof(struct meta_area))/sizeof(struct meta);
 int nslots;

 //指向管理的meta的指针。这种写法算是一个trick，意思是结构体后面的内存是meta数组，
 //长度不定(int nslots;)，使用者需要小心计算使用。
 struct meta slots[];
};
```

```c
//in meta.h
//struct meta管理了struct group，group中的storage就是给用户使用的内存。
//group中管理多个"chunk"交由用户使用，可类比glibc中的chunk。
//meta管理的group中chunk的个数由small_cnt_tab数组指定。
//meta管理的group中每个chunk的大小固定，由sizeclass指定。
struct meta {
 struct meta *prev, *next;    //双向链表
 //指向管理的group，一定在另一个内存页，有一定程度的隔离，防止溢出攻击。
 struct group *mem;
 volatile int avail_mask, freed_mask; //掩码的形式，用一个bit表示存在与否。
 uintptr_t last_idx:5;
 uintptr_t freeable:1;
 uintptr_t sizeclass:6;     //管理的group的大小。如果mem是mmap分配，固定为63。
 uintptr_t maplen:8*sizeof(uintptr_t)-12;//如果管理的group是mmap分配的，则为内存页数，否则为0。
};

struct group {
 struct meta *meta;       //指回管理者struct meta
 unsigned char active_idx:5;     //5个bit
 char pad[UNIT - sizeof(struct meta *) - 1]; //手动16字节对齐，这样给用户的storage[]是对齐的
 unsigned char storage[];
};
```

关键代码

void* malloc(size_t)

```c
//in malloc.c
void *malloc(size_t n)
{
 if (size_overflows(n)) return 0;
 ...
 //#define MMAP_THRESHOLD 131052
 if (n >= MMAP_THRESHOLD) {
  //mmap()
  ...
  goto success;
 }

 sc = size_to_class(n);
 g = ctx.active[sc]; //查找缓存，拿到meta* g
 ...
 for (;;) {
  mask = g ? g->avail_mask : 0;
  first = mask&-mask; //找到最低的为1的bit位
  if (!first) break; //没有avail的chunk，break
  ...
  //将avail这一个bit置位0，要分配出去了
  g->avail_mask = mask-first;
  ...
  idx = a_ctz_32(first); //取2的指数，即group中chunk下标
  goto success;
 }

 //如果缓存里没有avail的chunk，进一步申请
 idx = alloc_slot(sc, n);
 ...
 //缓存可能更新了，刷新g
 g = ctx.active[sc];

 success:
 ctr = ctx.mmap_counter;
 return enframe(g, idx, n, ctr);
}
```

static int alloc_slot(int, size_t)

```c
//in malloc.c
//sc = size_to_class(req), req为申请的内存大小
static int alloc_slot(int sc, size_t req)
{
 // malloc()中初略发现ctx.active[sc]没有avail的,
 //详细检查一下缓存(可能有freed mask或者meta.next有可分配的)
 uint32_t first = try_avail(&ctx.active[sc]);
 if (first) return a_ctz_32(first); //分配成功 return

 struct meta *g = alloc_group(sc, req); //进一步申请，申请全新的meta与group
 if (!g) return -1;

 g->avail_mask--;
 queue(&ctx.active[sc], g); //加入缓存，此时ctx.acive[sc]应该是NULL
 return 0;
}
```

static uint32_t try_avail(struct meta **)

```c
//in malloc.c
static uint32_t try_avail(struct meta **pm)
{
 struct meta *m = *pm;
 ...
 uint32_t mask = m->avail_mask;
 if (!mask) { //没有avail了
  if (!m->freed_mask) { //且没有free的chunk，意味着meta的group中所有chunk都分配出去了
   dequeue(pm, m); //meta unlink，UNSAFE UNLINK! 可能的一次任意地址写
   m = *pm; //此时的m为下一个meta, m不为NULL的话肯定有能分配的
   if (!m) return 0;
  } else {
   m = m->next; //优先找下一个，如果没下一个就指回自己(循环链表)
   *pm = m;
  }

  mask = m->freed_mask;

  // skip fully-free group unless it's the only one
  ...

  // activate more slots in a not-fully-active group
  // if needed, but only as a last resort. prefer using
  // any other group with free slots.
  ...

  //把free mask转换成avail mask，cas同步方式
  mask = activate_group(m);
  assert(mask);
  decay_bounces(m->sizeclass); // ???
 }

 //提取出avail的第一个chunk对应的mask，return
 first = mask&-mask;
 m->avail_mask = mask-first;
 return first;
}
```

static struct meta * alloc_group(int, size_t)

```c
//in malloc.c
static struct meta *alloc_group(int sc, size_t req)
{
 size_t size = UNIT*size_classes[sc];
 int i = 0, cnt;
 unsigned char *p;

 //分配全新的meta，不再展开源代码
 //优先查看ctx.free_meta_head的链表，如果没有->
 //再查看ctx管理的meta_area是否有剩的meta，是否有其他的meta_area， 如果都没有->
 //尝试使用brk()以内存页为单位分配堆，会中间多分配一个无读写权限的内存页，作为guard
 //brk()失败会尝试使用mmap()
 struct meta *m = alloc_meta();
 if (!m) return 0;
 size_t usage = ctx.usage_by_class[sc];

 //一通操作，给cnt赋了一个合适的值
 ...
 cnt = xxx; //cnt是这个meta的group中chunk的数量
 ...

 if (size*cnt+UNIT > pagesize/2) {
  ...
  //当需求很大时，尝试mmap()分配
  p = mmap(0, needed, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, -1, 0);
  ...
 } else {
  int j = size_to_class(UNIT+cnt*size-IB);

  //又一次alloc_slot() 这是一个递归过程
  //会尝试向更大的缓存要内存，更大缓存中的chunk会成为这里的group
  int idx = alloc_slot(j, UNIT+cnt*size-IB);
  if (idx < 0) {
   free_meta(m);
   return 0;
  }
  struct meta *g = ctx.active[j];
  p = enframe(g, idx, UNIT*size_classes[j]-IB, ctx.mmap_counter);
  //--------以上代码实质上走了类似malloc()的过程，向更大缓存要了块chunk

  m->maplen = 0;
  p[-3] = (p[-3]&31) | (6<<5);
  for (int i=0; i<=cnt; i++)
   p[UNIT+i*size-4] = 0;
  active_idx = cnt-1;
 }
 //加入缓存的计数
 ctx.usage_by_class[sc] += cnt;
 //给这个全新的meta设置好初始值
 m->avail_mask = (2u<<active_idx)-1;
 m->freed_mask = (2u<<(cnt-1))-1 - m->avail_mask;
 m->mem = (void *)p;
 m->mem->meta = m;
 m->mem->active_idx = active_idx;
 m->last_idx = cnt-1;
 m->freeable = 1;
 m->sizeclass = sc;
 return m;
}
```

void* enframe(struct meta *, int, size_t, int)

```
//in malloc.c
//已经确定要分配出去的内存位置，enframe()这里做一些检查和清理工作
static inline void *enframe(struct meta *g, int idx, size_t n, int ctr)
{

 size_t stride = get_stride(g); //一个chunk的大小
 size_t slack = (stride-IB-n)/UNIT; //chunk多余的空间
 unsigned char *p = g->mem->storage + stride*idx; //返回给用户使用的chunk地址
 unsigned char *end = p+stride-IB;
 // cycle offset within slot to increase interval to address
 // reuse, facilitate trapping double-free.
 //off: 计算出p离第一个chunk的距离，以16byte为单位。
 int off = (p[-3] ? *(uint16_t *)(p-2) + 1 : ctr) & 255;
 ... //check
 if (off) {
  // store offset in unused header at offset zero
  // if enframing at non-zero offset.
  *(uint16_t *)(p-2) = off;
  p[-3] = 7<<5;
  p += UNIT*off;
  // for nonzero offset there is no permanent check
  // byte, so make one.
  p[-4] = 0;
 }
 *(uint16_t *)(p-2) = (size_t)(p-g->mem->storage)/UNIT;
 p[-3] = idx;
 set_size(p, end, n);
 return p;
}
```

void free(void *)

```c
//in free.c
void free(void *p)
{
 if (!p) return;
 struct meta *g = get_meta(p); //p -> struct group -> struct meta,并且检查
 int idx = get_slot_index(p); //p 在group中的下标
 size_t stride = get_stride(g); //chunk的大小

 ... // check

 uint32_t self = 1u<<idx, all = (2u<<g->last_idx)-1;
 ((unsigned char *)p)[-3] = 255; //p[-3]与slot_index和reserve有关，现在置0xff
 *(uint16_t *)((char *)p-2) = 0; //p[-1] = p[-2] = 0;

 ...

 // atomic free without locking if this is neither first or last slot
 for (;;) {
  uint32_t freed = g->freed_mask;
  uint32_t avail = g->avail_mask;
  uint32_t mask = freed | avail;
  assert(!(mask&self)); //防止doubel free

  //如果是full的meta进行free的话，或者是free了后就empty的话，那就break，进行特殊处理。
  //不然就继续，然后return。
  if (!freed || mask+self==all) break;
  if (!MT)
   g->freed_mask = freed+self;

  //无锁的同步方式。成功则return，失败continue
  else if (a_cas(&g->freed_mask, freed, freed+self)!=freed)
   continue;
  return;
 }

 wrlock();
 //特殊处理两个情况，全空或者全满，需要对meta的double link进行处理，可能还伴随着meta的申请与释放。
 struct mapinfo mi = nontrivial_free(g, idx);
 unlock();
 ...
}
```

static struct mapinfo nontrivial_free(struct meta *, int)

```c
static struct mapinfo nontrivial_free(struct meta *g, int i)
{
 uint32_t self = 1u<<i;
 int sc = g->sizeclass;
 uint32_t mask = g->freed_mask | g->avail_mask;

 if (mask+self == (2u<<g->last_idx)-1 && okay_to_free(g)) { //如果是个全空的meta
  // any multi-slot group is necessarily on an active list
  // here, but single-slot groups might or might not be.
  if (g->next) {
   assert(sc < 48);
   int activate_new = (ctx.active[sc]==g);
   dequeue(&ctx.active[sc], g);
   if (activate_new && ctx.active[sc])
    activate_group(ctx.active[sc]);
  }
  //free group，其中会free meta。不再展开源码
  //free meta只是简单的放入ctx.free_meta_head，不会尝试合并
  //free group是递归过程，如果这个group是从更大缓存拿的，那么再走一遍nontrivial_free()
  return free_group(g);
 } else if (!mask) { //如果是个全满的meta
  assert(sc < 48);
  // might still be active if there were no allocations
  // after last available slot was taken.
  if (ctx.active[sc] != g) {
   queue(&ctx.active[sc], g); //取出缓存链表
  }
 }
 a_or(&g->freed_mask, self);
 return (struct mapinfo){ 0 };
}
```

2021 Defcon Quals mooosl

一个典型的菜单堆题。题目中有悬空指针可以用来风水，完成任意地址泄漏和任意地址free。
pwn思路是：

- 泄漏libc地址, heap地址, ctx.secret用来伪造meta_area与meta
- 触发`nontrivial_free()`中的`dequeue()`完成任意地址写
- 任意地址写直接劫持__stdout_used，指向伪造的IO_FILE。musl libc中的IO_FILE结构体内就有函数指针，直接劫持控制流。

to be continued…

2021 强网杯初赛 easyheap

一个典型的CTFer在CTF比赛中给其他CTFer出的一道CTF题，题目中CTF味太浓了。

- 题目开始套了一层aes密码，需要整完才进入堆菜单。
- 堆菜单只是布局，布局完后进入检查环节。
- 检查环节有一个莫名的金手指能力，配合前面的布局完成堆风水。最后也是走到`dequeue()`完成控制流劫持。

to be continued…

额外补充 IO_FILE

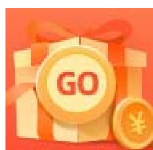
第二届祥云杯刚刚结束。比赛开始第一天有点事，第二天上去发现不知道啥时候出了个musl libc。我这一看不正合我意，不然我写个博客干啥的(欣慰的一笑)，搞一搞就拿了一血。下面分享一下在musl libc的IO_FILE关键代码。

```
// 关键点， _IO_FILE结构体内部自带函数指针
struct _IO_FILE {
 unsigned flags;
 unsigned char *rpos, *rend;
 int (*close)(FILE *);
 unsigned char *wend, *wpos;
 unsigned char *mustbezero_1;
 unsigned char *wbase;
 size_t (*read)(FILE *, unsigned char *, size_t);
 size_t (*write)(FILE *, const unsigned char *, size_t);
 off_t (*seek)(FILE *, off_t, int);
 unsigned char *buf;
 size_t buf_size;
 FILE *prev, *next;
 int fd;
 int pipe_pid;
 long lockcount;
 int mode;
 volatile int lock;
 int lbf;
 void *cookie;
 off_t off;
 char *getln_buf;
 void *mustbezero_2;
 unsigned char *shend;
 off_t shlim, shcnt;
 FILE *prev_locked, *next_locked;
 struct __locale_struct *locale;
};
```

```c
// 思路一：可以想办法写ofl_head，类似glibc中的FSOP，伪造IO_FILE
static FILE *ofl_head;

hidden FILE __stdin_FILE = {
 .buf = buf+UNGET,
 .buf_size = sizeof buf-UNGET,
 .fd = 0,
 .flags = F_PERM | F_NOWR,
 .read = __stdio_read,
 .seek = __stdio_seek,
 .close = __stdio_close,
 .lock = -1,
};
FILE *const stdin = &__stdin_FILE;
FILE *volatile __stdin_used = &__stdin_FILE;

static unsigned char buf[BUFSIZ+UNGET];
hidden FILE __stdout_FILE = {
 .buf = buf+UNGET,
 .buf_size = sizeof buf-UNGET,
 .fd = 1,
 .flags = F_PERM | F_NORD,
 .lbf = '\n',
 .write = __stdout_write,
 .seek = __stdio_seek,
 .close = __stdio_close,
 .lock = -1,
};
FILE *const stdout = &__stdout_FILE;
FILE *volatile __stdout_used = &__stdout_FILE;
```

```c
//思路二 也类似FSOP中走exit()这条路，会依次close_file(),不仅仅是ofl_head链上的
//还包括__stdin_used,__stdout_used。链上的和__stdin_used等可不一定相同。
static void close_file(FILE *f)
{
 if (!f) return;
 FFINALLOCK(f);
 if (f->wpos != f->wbase) f->write(f, 0, 0);
 if (f->rpos != f->rend) f->seek(f, f->rpos-f->rend, SEEK_CUR);
}

void __stdio_exit(void)
{
 FILE *f;
 for (f=*__ofl_lock(); f; f=f->next) close_file(f);
 close_file(__stdin_used);
 close_file(__stdout_used);
 close_file(__stderr_used);
}
```

```c
//使用ropgadget可以发现，musl libc中也有类似glibc的magic，可用作栈迁移。
//0x000000000004bcf3 : mov rsp, qword ptr [rdi + 0x30] ; jmp qword ptr [rdi + 0x38]
```



创作打卡挑战赛

赢取流量/现金/CSDN周边激励大奖