堆漏洞挖掘（十）

安全+ 安全加 2019-11-10

免责声明：本公众号发布的文章均转载自互联网或经作者投稿授权的原创，文末已注明出处，其内容和图片版权归原网站或作者本人所有，并不代表安全+的观点，若有无意侵权或转载不当之处请联系我们处理，谢谢合作！

malloc一个chunk的检测机制

一、unlink的概念

- 双向链表中移除/添加一个chunk时，会发生断链的操作，这个断链的过程就叫做unlink
- 注意事项：unlink不发生在fastbin和smallbin中，所以fastbin和smallbin容易产生漏洞

二、可能发生的场景

malloc时：

- 从恰好大小合适的largebin中获取chunk，发生unlink
- 从比malloc要求大的largebin中取chunk，发生unlink

free时：

- free之后，与前后空闲的chunk进行合并

malloc consolidate时：

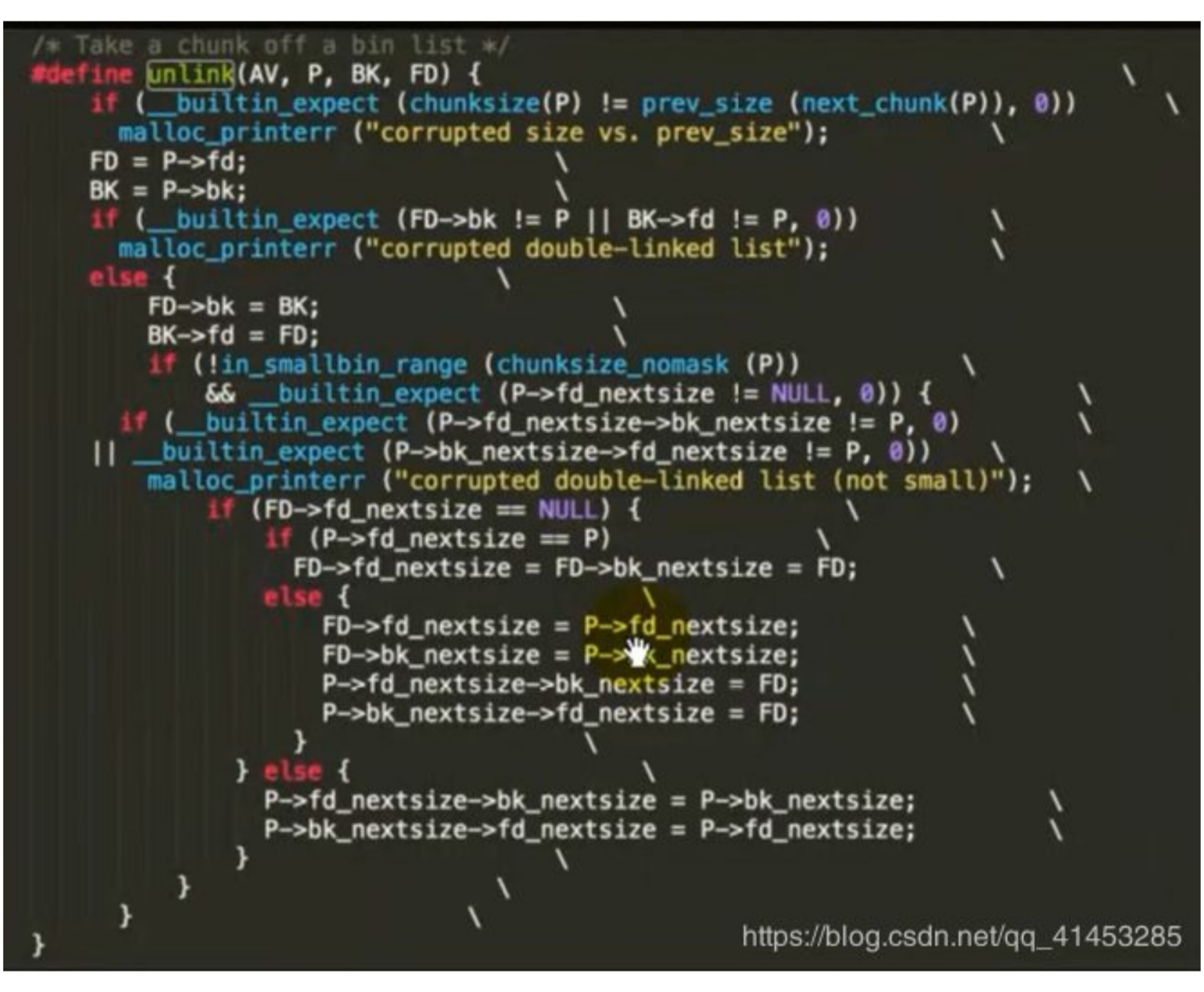
- consolidate时，chunk之间的unlink

realloc时：

- 向前扩展，合并物理相邻高地址空闲chunk

三、unlink宏

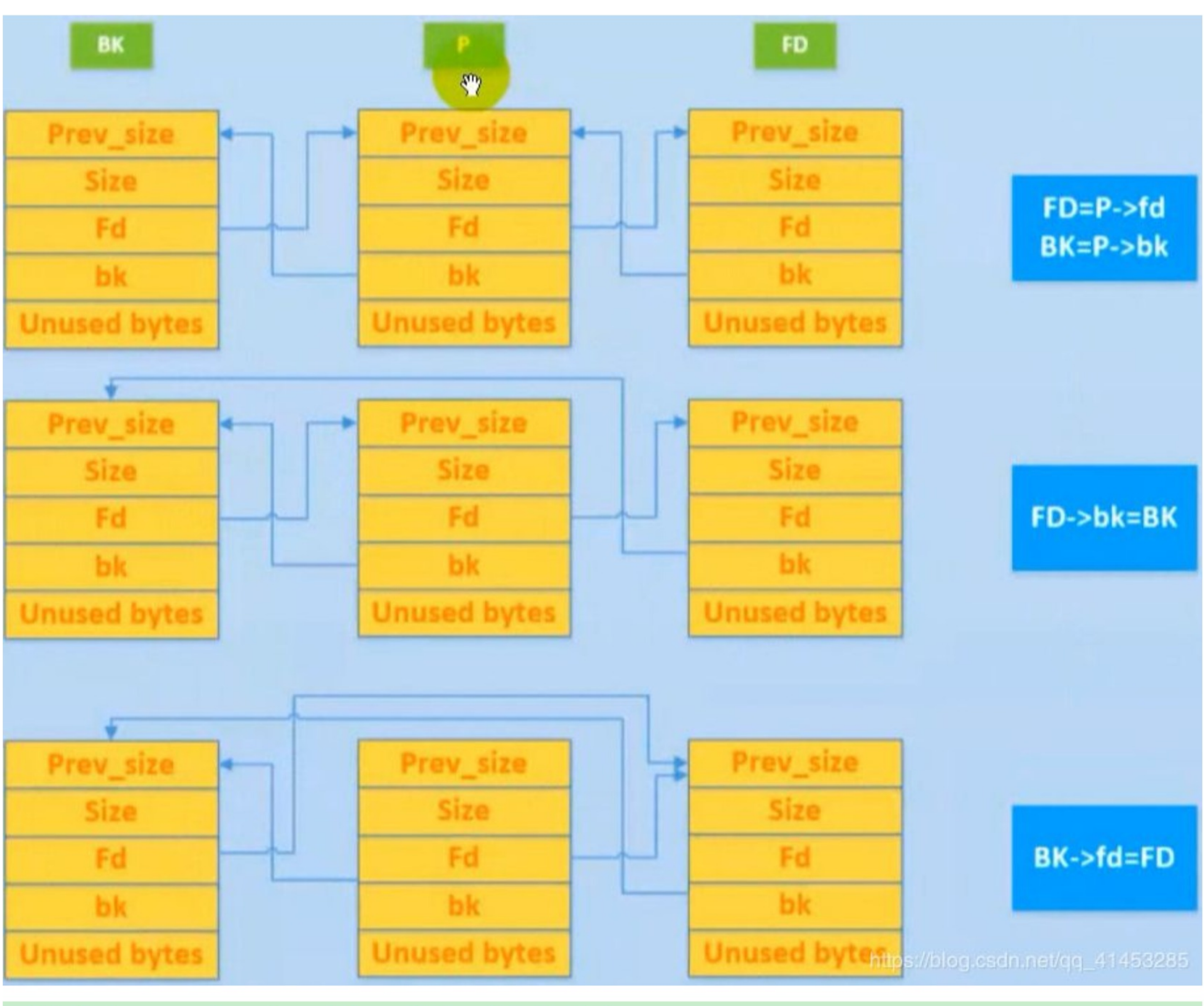
- unlink使用很频繁，在glibc中被定义为一个宏

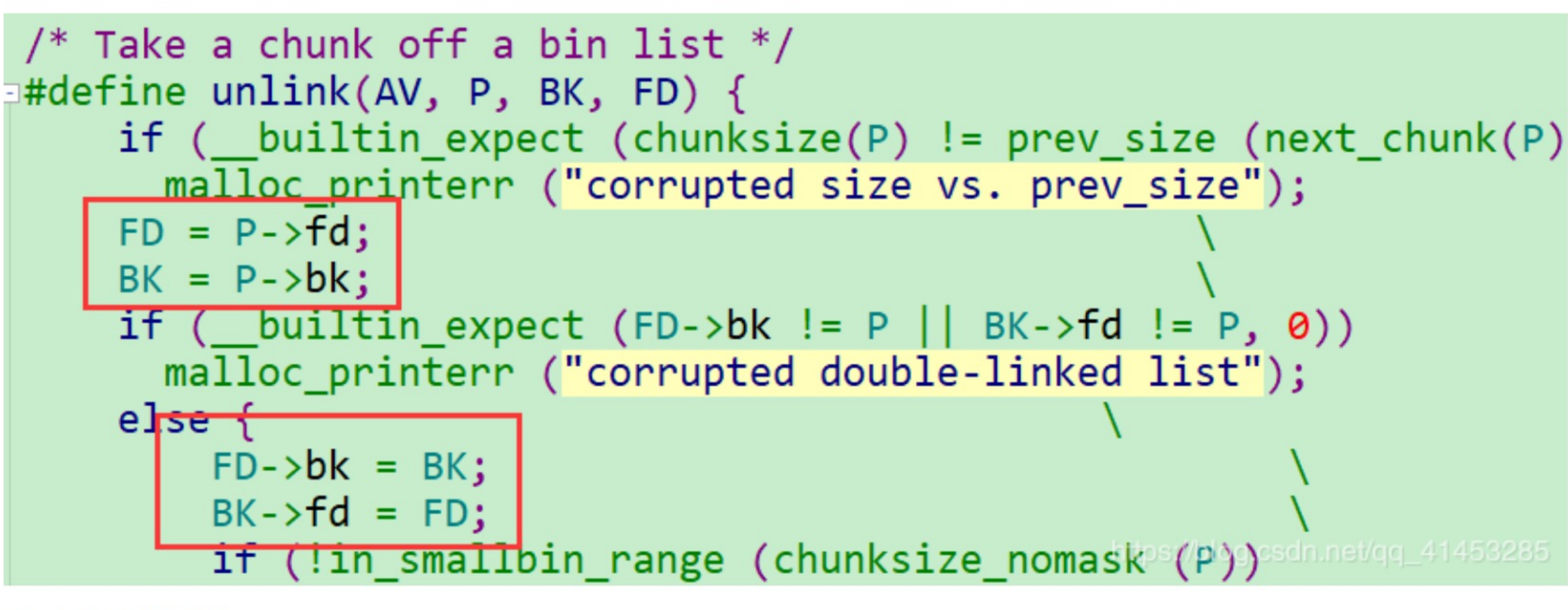


```
/* Take a chunk off a bin list */
#define unlink(AV, P, BK, FD) {                                            \
    if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))      \
      malloc_printerr ("corrupted size vs. prev_size");                      \
    FD = P->fd;                                                           \
    BK = P->bk;                                                           \
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                  \
      malloc_printerr ("corrupted double-linked list");                      \
    else {                                                                \
        FD->bk = BK;                                                      \
        BK->fd = FD;                                                      \
        if (!in_smallbin_range (chunksize_nomask (P))                     \
            && __builtin_expect (P->fd_nextsize != NULL, 0)) {           \
      if (__builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)          \
    || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0))            \
        malloc_printerr ("corrupted double-linked list (not small)");      \
            if (FD->fd_nextsize == NULL) {                                \
                if (P->fd_nextsize == P)                                  \
                  FD->fd_nextsize = FD->bk_nextsize = FD;                 \
                else {                                                    \
                    FD->fd_nextsize = P->fd_nextsize;                     \
                    FD->bk_nextsize = P->bk_nextsize;                     \
                    P->fd_nextsize->bk_nextsize = FD;                     \
                    P->bk_nextsize->fd_nextsize = FD;                     \
                  }                                                       \
              } else {                                                    \
                P->fd_nextsize->bk_nextsize = P->bk_nextsize;             \
                P->bk_nextsize->fd_nextsize = P->fd_nextsize;             \
              }                                                           \
          }                                                               \
      }                                                                   \
}
```

四、unlink的过程

- P是我们要操作的堆块。那么所发生的操作如下



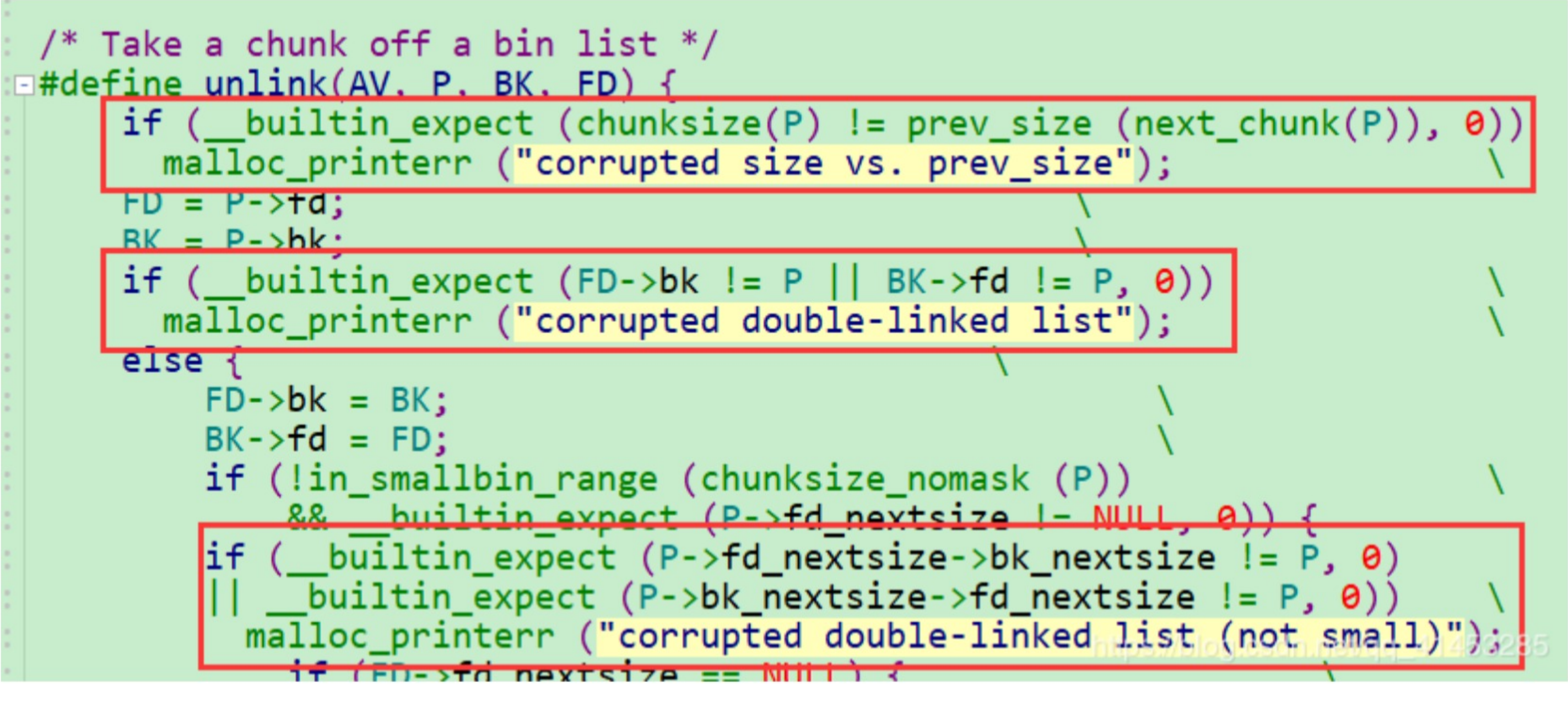


```
/* Take a chunk off a bin list */
#define unlink(AV, P, BK, FD) {
    if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)
      malloc_printerr ("corrupted size vs. prev_size");
    FD = P->fd;                                         \
    BK = P->bk;                                         \
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
      malloc_printerr ("corrupted double-linked list");
    else {                                            \
        FD->bk = BK;                                      \
        BK->fd = FD;                                      \
        if (!in_smallbin_range (chunksize_nomask (P))
```

五、unlink的检测机制

从unlink宏中可以看到，unlink的时候会发生3种检测，如果检测不合格，就会unlink失败

- **检测1：**如果要释放的P的后面chunk的prev_size成员数值不等于P本身的大小，出错退出
- **检测2：**如果P的前chunk的bk指针不等于P，P的后chunk的fd指针不等于P，出错退出
- **检测3：**



```
 /* Take a chunk off a bin list */
#define unlink(AV, P, BK, FD) {
    if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))
      malloc_printerr ("corrupted size vs. prev_size");                \
    FD = P->fd;                                    \
    BK = P->bk;                                    \
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))               \
      malloc_printerr ("corrupted double-linked list");                 \
    else {                                    \
        FD->bk = BK;                                    \
        BK->fd = FD;                                    \
        if (!in_smallbin_range (chunksize_nomask (P))                   \
            && __builtin_expect (P->fd_nextsize != NULL, 0)) {
        if (__builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)
        || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0))       \
          malloc_printerr ("corrupted double-linked list (not small)");
            if (FD->fd_nextsize == NULL) {                               \
```

▲▲▲▲▲

_lib_malloc函数源码详解

一、_lib_malloc函数介绍

- 当我们在应用层调用malloc申请堆的时候，在glibc中实际上调用的是_lib_malloc函数，但是_lib_malloc函数只是用来简单的封装_int_malloc函数的，_int_malloc函数才是申请堆的核心函数

二、__malloc_hook全局变量

- **函数介绍：**_lib_malloc首先通过__malloc_hook全局变量获取一个函数指针，然后判断这个函数是否为空，这个函数代表用户自定义的堆分配函数，主要为了方便用户快速修改该函数并进行测试
- **_malloc_hook漏洞：**如果__malloc_hook被修改，那么就会执行被修改后的函数（one_gadget），以后文章介绍

从下面的源码可以看到，先读取__malloc_hook全局变量，然后判断是否有用户自定义的堆分配函数，如果有就执行，不再进行系统的堆分配了(_int_malloc)

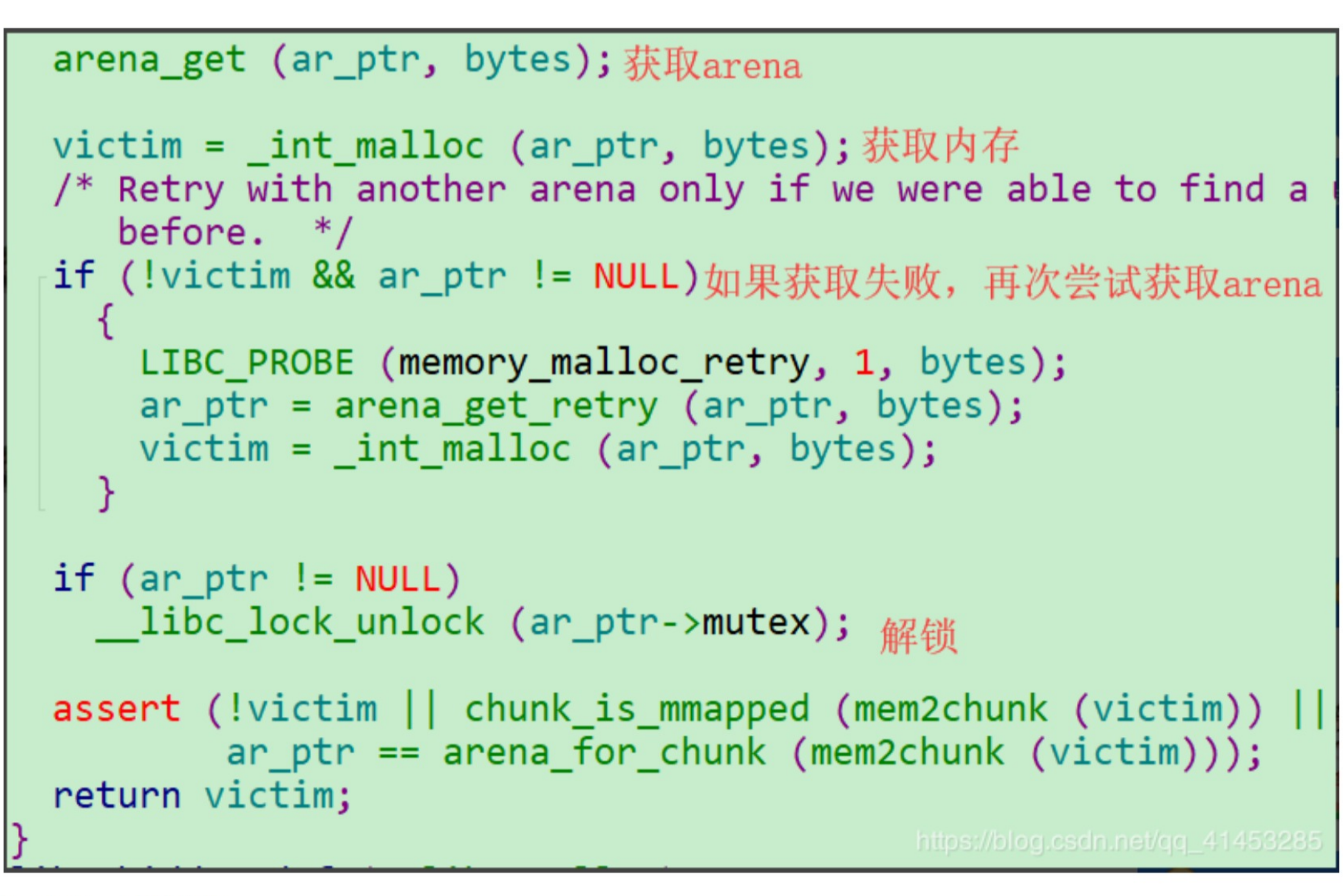
```
void *
__libc_malloc (size_t bytes)   //bytes:要malloc的大小
{
  mstate ar_ptr;
  void *victim;

  //1.先去读取__malloc_hook全局变量，然后返回一个函数指针
  void *(*hook) (size_t, const void *)
    = atomic_forced_read (__malloc_hook);

  //2.__builtin_expect内置函数来判断hook函数指针是否为空，不为空直接执行hook函数然后返回了
  if (__builtin_expect (hook != NULL, 0))
    return (*hook)(bytes, RETURN_ADDRESS (0));
```

三、寻找arena并执行_int_malloc函数函数

- 解析_lib_malloc函数回去寻找一个arena来试图分配内存（arena_get），然后调用_int_malloc函数取申请内存
- 如果分配失败（注：_int_malloc函数中会再次判断是否有arena可用），ptmalloc会尝试再去寻找一个可用的arena，并再次调用_int_malloc函数取申请内存
- 如果申请到了arena，那么在使用完arena之后，函数退出之前还需要解锁（_lib_lock_lock）



```
  arena_get (ar_ptr, bytes); 获取arena

  victim = _int_malloc (ar_ptr, bytes); 获取内存
  /* Retry with another arena only if we were able to find a
     before.  */
  if (!victim && ar_ptr != NULL) 如果获取失败，再次尝试获取arena
    {
      LIBC_PROBE (memory_malloc_retry, 1, bytes);
      ar_ptr = arena_get_retry (ar_ptr, bytes);
      victim = _int_malloc (ar_ptr, bytes);
    }

  if (ar_ptr != NULL)
    __libc_lock_unlock (ar_ptr->mutex);  解锁

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

四、_lib_malloc函数总结

_lib_malloc函数就是做以下事情：

- 要么没有申请到内存
- 要么通过mmap/brk申请内存
- 要么在其arena中分配内存
- （或者出错退出）
- 最终返回内存（return）
- 核心：调用_int_malloc函数

▲▲▲▲▲

本文作者 江南、董少
本文转载自CSDN
原文链接：
https://blog.csdn.net/qq_41453285/article/details/98850842
https://blog.csdn.net/qq_41453285/article/details/99048619

“安全+”征稿活动火热开启

一、参与方式
请将文章发送到以下邮箱
jacky.qin@anquanjia.net.cn
投稿者请填写真实姓名及联系方式，并标明文章内容

二、征稿内容
云安全、物联网安全、数据安全、企业安全建设、AI安全、漏洞、内网安全、安全管理、安全趋势、挖洞经验、安全研究、企业安全实践及所有安全相关的内容等。

三、征稿须知
所有文章都要保证本人原创，且从未在任何平台发表过。

四、征稿奖励
经审核通过后，
1. 作为演讲嘉宾，在EISS、"安全+"沙龙、白帽子技术沙龙等活动做主题分享;
2.优秀文章前10名，在EISS峰会接受颁奖;
3.作为VIP，免费参加"安全+"线下所有活动;
4.可获得"安全+"所有活动授权公开分享的PPT。

联系人：Jacky
T:+86 18116005259
E：jacky.qin@anquanjia.net.cn

安全+

安全+ 专注于安全行业，通过互联网平台、线下沙龙、培训、峰会、人才招聘等多种形式，致力于创建亚太地区最好的甲乙双方交流、学习的平台，培养安全人才，提升行业整体素质，助推安全生态圈的健康发展。

官方网站：www.anquanjia.net.cn

堆漏洞挖掘（九）
堆漏洞挖掘（八）
堆漏洞挖掘（七）
堆漏洞挖掘（六）
堆漏洞挖掘（五）
堆漏洞挖掘（四）
堆漏洞挖掘（三）
堆漏洞挖掘（二）
堆漏洞挖掘（一）