[http://www.phrack.org/issues/57/8.html?fbclid=IwAR1dj8VwiMSSBXCCVLzyauXlU0TKL4Na5P9JefKHvh4GUd8RlFht-viWmMg#article](http://www.phrack.org/issues/57/8.html?fbclid=IwAR1dj8VwiMSSBXCCVLzyauXlU0TKL4Na5P9JefKHvh4GUd8RlFht-viWmMg" \l "article)

[3.2 – chunk of memory]

* không có 2 chunk free liên tiếp nhau, 2 chunk này sẽ được gộp lại với nhau.

[ 3.2.1 - Synopsis of public routines ]

* malloc(size\_t n): trả về 1 con trỏ tới 1 vùng vừa được cấp phát ít nhất n bytes, null nếu không thể đáp ứng ( không có vùng nhớ nào đáp ứng yêu cầu)
* free(void\_t \*p): giải phóng vùng nhớ được trỏ tới bởi con trỏ p,hoặc không làm gì nếu p = null
* realloc(void\_t \*p, size\_t n): trả về 1 con trỏ đến 1 chunk có size n, có nội dung giống với chunk p tới [min(n, p’s size)] bytes. Nếu size=0, tương tự như free(\*p)
* calloc(size\_t unit, size\_t quantity) : trả về con trỏ tới 1 chunk có (unit\*quantity) size, với tất cả phần tử = 0.

[ 3.2.2 - Vital statistics ]

* khi gọi malloc, size của chunk được trả về luôn lớn hơn size được yêu cầu, đó là do boundary tag và 8 bytes alignment.
* Alignment: vì size của 1 chunk luôn là bội số của 8 và chunk đầu tiên luôn alignment cho 8 nên địa chỉ của 1 chunk luôn là địa chỉ chia hết cho 8
* Chi phí tối thiểu cho mỗi chunk được malloc: ít nhất là 4 bytes – boundary tag liên kết với từng chunk, lưu giữ size và trạng thái của chunk.
* Kích thước chunk tối thiểu: 16 bytes

[ 3.2.3 - Available chunks ]: khối có sẵn

* Bins: lưu trữ những free chunk.
* Top-most chunk: luôn có sẵn(để malloc) và không thuộc bins nào.
* Phần còn lại của 1 chunk gần đây nhất bị cắt ra (bởi 1 yêu cầu malloc nào đó) .

[ 3.3 - Boundary tags ]

[ 3.3.1 – Structure:cấu trúc ]

* #define INTERNAL\_SIZE\_T size\_t
* struct malloc\_chunk {
* INTERNAL\_SIZE\_T prev\_size;
* INTERNAL\_SIZE\_T size;
* struct malloc\_chunk \* fd;
* struct malloc\_chunk \* bk;
* };
* Cấu trúc trên được sử dụng cho boundary tag, tuy nhiên từng field của nó được sử dụng như thế nào tuỳ vào trạng thái (freed or allocated) của chunk và chunk trước đó.
* Allocated chunk:
  + chunk -> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  + | prev\_size:kích thước của chunk trước đó,in bytes(được sử|
  + |dụng bởi dlmalloc chỉ khi chunk trước đó được free) |
  + +---------------------------------------------------------+
  + | size: size of the chunk (the number of bytes between |
  + | "chunk" and "nextchunk") and 2 bits status information |
  + mem -> +---------------------------------------------------------+
  + | fd: not used by dlmalloc because "chunk" is allocated |
  + | (user data therefore starts here) |
  + + - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
  + | bk: not used by dlmalloc because "chunk" is allocated |
  + | (there may be user data here) |
  + + - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
  + | .
  + . .
  + . user data (may be 0 bytes long) .
  + . .
  + . |
  + nextchunk -> + + + + + + ++ + + + + + + + + + + + + + + + + + + + + +
  + | prev\_size: not used by dlmalloc because "chunk" is |
  + | allocated (may hold user data, to decrease wastage) |
  + +---------------------------------------------------------+
  + Sự chuyển đổi từ “chunk” (dlmalloc header) sang “mem”(users pointer) và ngược lại, được thực hiện bởi 2 hàm, chunk2mem() and mem2chunk() [ Đặt tên dễ hiểu phết] .
  + Mem = chunk + sizeof(prev\_size) + sizeof(size)
  + Chunk = mem - sizeof(prev\_size) - sizeof(size)
  + #define Void\_t void
  + #define SIZE\_SZ sizeof(INTERNAL\_SIZE\_T)
  + typedef struct malloc\_chunk \* mchunkptr;
  + #define chunk2mem( p ) \
  + ( (Void\_t \*)((char \*)(p) + 2\*SIZE\_SZ) )
  + #define mem2chunk( mem ) \
  + ( (mchunkptr)((char \*)(mem) - 2\*SIZE\_SZ) )
  + Thực tế, user có thể lưu dữ liệu bắt đầu từ “mem” đến hết “prev\_size” của nextchunk
  + = (“nextchunk”+4 – mem) = (nextchunk+4 – (chunk-8)) = (nextchunk – chunk – 4 ) = size-4
* Freed chunk:
  + chunk -> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  + | prev\_size: may hold user data (indeed, since "chunk" is |
  + | free, the previous chunk is necessarily allocated) |
  + +---------------------------------------------------------+
  + | size: size of the chunk (the number of bytes between |
  + | "chunk" and "nextchunk") and 2 bits status information |
  + +---------------------------------------------------------+
  + | fd: forward pointer to the next chunk in the circular |
  + | doubly-linked list (not to the next \_physical\_ chunk) |
  + +---------------------------------------------------------+
  + | bk: back pointer to the previous chunk in the circular |
  + | doubly-linked list (not the previous \_physical\_ chunk) |
  + +---------------------------------------------------------+
  + | .
  + . .
  + . unused space (may be 0 bytes long) .
  + . .
  + . |
  + nextchunk -> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  + | prev\_size: size of "chunk", in bytes (used by dlmalloc |
  + | because this previous chunk is free) |
  + +---------------------------------------------------------+
* [ 3.3.2 - Size of a chunk ]
  + Khi user yêu cầu 1 chunk có size là req size, dlmalloc sẽ gọi hàm request2size() để chuyển req size sang size thực tế( effective size of allocated chunk of memory) tức là bao gồm cả alignment và boundary tag.
* #define MINSIZE sizeof(struct malloc\_chunk)
* #define MALLOC\_ALIGNMENT ( SIZE\_SZ + SIZE\_SZ )
* #define MALLOC\_ALIGN\_MASK ( MALLOC\_ALIGNMENT - 1 )
* size\_t request2size( size\_t req )
* {
* size\_t nb;
* nb = req + ( SIZE\_SZ + MALLOC\_ALIGN\_MASK );
* if ( nb < (MINSIZE + MALLOC\_ALIGN\_MASK) ) {
* nb = MINSIZE;
* } else {
* nb &= ~MALLOC\_ALIGN\_MASK;
* }
* return( nb );
* }
* Finally, the request2size() macro implemented in Doug Lea's Malloc works
* likewise but adds an integer overflow detection:
* #define request2size(req, nb) \
* ((nb = (req) + (SIZE\_SZ + MALLOC\_ALIGN\_MASK)),\
* ((long)nb <= 0 || nb < (INTERNAL\_SIZE\_T) (req) \
* ? (\_\_set\_errno (ENOMEM), 1) \
* : ((nb < (MINSIZE + MALLOC\_ALIGN\_MASK) \
* ? (nb = MINSIZE) : (nb &= ~MALLOC\_ALIGN\_MASK)), 0)))
  + Dựng lại hàm bằng python :
  + size\_sz = 4
  + minsize = 8
  + malloc\_alignment = 8
  + malloc\_align\_mask = malloc\_alignment - 1
  + def request2size(req):
  + nb = 0
  + nb = req + size\_sz + malloc\_align\_mask
  + if(nb <(minsize + malloc\_align\_mask)):
  + nb = minsize
  + else:
  + print('nb:'+bin(nb))
  + print('~malloc\_align\_mask:'+bin(~malloc\_align\_mask))
  + nb &= ~malloc\_align\_mask
  + print('nb:'+bin(nb))
  + return nb
  + print( request2size(22))
* [ 3.3.3 - prev\_size field ]
  + Nếu chunk đang xét là chunk p, chunk đứng trước nó đag được sự dụng thì prev\_size field của p sẽ được sử dụng để chứa dữ liệu, giảm tình trạng lãng phí.
  + Nếu chunk trước nó đã được freed, prev\_size field sẽ được sử dụng bởi dlmalloc và chứa size của chunk trước nó. Địa chỉ của chunk đứng trước có thể được tính bởi macro prev\_chunk():
    - #define prev\_chunk( p ) \
    - ( (mchunkptr)(((char \*)(p)) - ((p)->prev\_size)) )
* [ 3.3.4 - size field ]
  + Nói gọn lại là field này chứa 2 size của chunk và thêm vào 2 thông tin trạng thái của chunk bằng cách đánh dấu 2 bits thấp nhất, vì giá trị size này luôn là bội của 8 nên 3 bits cuối luôn = 0. Bit thấp hơn giữ giá trị của bit PRE\_INUSE và biết thấp thứ 2 giữ giá trị của IS\_MAPPED bit.
  + Để lấy được chunksize của p, ta phải dùng macro chunksize(p):
    - #define SIZE\_BITS ( PREV\_INUSE | IS\_MMAPPED )
    - #define chunksize( p ) \
    - ( (p)->size & ~(SIZE\_BITS) )
    - #define PREV\_INUSE 0x1
    - #define IS\_MMAPPED 0x2
  + Để kiểm tra chunk p có được cấp phát bởi mmap không, chương trình dùng macro
    - #define chunk\_is\_mmapped( p ) \
    - ( (p)->size & IS\_MMAPPED )
  + Để kiểm tra chunk đứng trước chunk p có đang được cấp phát hay free, chương trình dùng
    - #define prev\_inuse( p ) \
    - ( (p)->size & PREV\_INUSE )
  + Để kiểm tra chunk p đang được cấp phát hay free, chương trình dùng:
    - #define inuse( p ) \

(((mchunkptr)((char\*)(p)+((p)->size&~PREV\_INUSE)))->size&PREV\_INUSE)

* [ 3.4 - Bins ]
  + Tất cả những chunk sẵn có(để malloc) được giữ trong bins, được nhóm lại theo size, trừ 2 trường hợp là : phần còn lại của những chunk bị cắt và top-most chunk( wilderness chunk) . 2 ngoại lệ này được xử lí đặc biệt và không thuộc bin nào.
* [ 3.4.1 - Indexing into bins ]
  + Có tất cả 128 bins, được group dựa vào size của chunk.
  + Để có thể tìm được index của bin, dlmalloc sẽ gọi 2 macro là smallbin\_index() và bin\_index().
    - #define smallbin\_index( sz ) \
    - ( ((unsigned long)(sz)) >> 3 )
  + Dlmalloc coi size < 512 bytes là small chunk, chứa trong 62 small\_bin. Index của small chunk = size/8
  + Đối với chunk có size > 512, dlmalloc sẽ gọi macro bin\_index()
    - #define bin\_index(sz) \
    - ((((unsigned long)(sz) >> 9) == 0) ? ((unsigned long)(sz) >> 3):\
    - (((unsigned long)(sz) >> 9) <= 4) ? 56 + ((unsigned long)(sz) >> 6):\
    - (((unsigned long)(sz) >> 9) <= 20) ? 91 + ((unsigned long)(sz) >> 9):\
    - (((unsigned long)(sz) >> 9) <= 84) ? 110 + ((unsigned long)(sz) >> 12):\
    - (((unsigned long)(sz) >> 9) <= 340) ? 119 + ((unsigned long)(sz) >> 15):\
    - (((unsigned long)(sz) >> 9) <= 1364) ? 124 + ((unsigned long)(sz) >> 18):\
* 126)
  + - Mình build lại bằng python cho dễ hiểu:
* def bin\_index(size):
* index = 0
* tmp = size>>9
* if(tmp == 0):
* index = size>>3
* elif(tmp<=4):
* index = int(size>>6)+56
* elif(tmp<=20):
* index = int(size>>9) + 91
* elif(tmp<=84):
* index = int(size>>12) + 110
* elif(tmp<=340):
* index = int(size>>15) + 119
* elif(tmp<=1364):
* index = int(size>>18) + 124
* else:
* index = 126
* return index
* [ 3.4.2 - Linkin Park^H^H^H^H^Hg chunks in bin lists ]
  + Những chunk được free chứa trong 1 danh sách liên kết đôi, và thực chất thì 1 bin là 1 cặp pointer: forward pointer và backward pointer
  + Mỗi chunk trong từng bin được sắp xếp theo chiều giảm dần về kích thước, không áp dụng cho small bin.
  + Con trỏ forward của bin trỏ tới chunk đầu tiên( chunk lớn nhất) trong danh sáchhoặc trỏ tới chính nó nếu danh sách đang rỗng, cho đến khi nó trỏ lại 1 lần nữa về bin.
  + Con trỏ backward trỏ ngược lại đến chunk trỏ tới nó cho đến khi trỏ lại về bin.
  + Để lấy 1 chunk ra khỏi danh sách liên kết, dlmalloc gọi macro:
    - #define unlink( P, BK, FD ) { \
    - BK = P->bk; \
    - FD = P->fd; \
    - FD->bk = BK; \
    - BK->fd = FD; \
    - }
  + Để bỏ 1 freed chunk có size S vào trong bin,sắp xếp theo size. Dlmalloc gọi macro frontlink(). Frontlink() gọi 2 macro là smallbin\_index() và bin\_index() để tìm index IDX phù hợp với S.
    - #define frontlink( A, P, S, IDX, BK, FD ) { \
    - if ( S < MAX\_SMALLBIN\_SIZE ) { \
    - IDX = smallbin\_index( S ); \
    - mark\_binblock( A, IDX );đánh dấu rằng bin đó không rỗng
    - BK = bin\_at( A, IDX );lấy địa chỉ vật lý của bin \
    - FD = BK->fd; \
    - P->bk = BK; \
    - P->fd = FD; \
    - FD->bk = BK->fd = P; \
    - } else { \
    - IDX = bin\_index( S ); \
    - BK = bin\_at( A, IDX ); \
    - FD = BK->fd; \
    - if ( FD == BK ) { \
    - mark\_binblock(A, IDX); \
    - } else { \
    - while ( FD != BK && S < chunksize(FD) ) { \
    - FD = FD->fd; \
    - } \
    - BK = FD->bk; \
    - } \
    - P->bk = BK; \
    - P->fd = FD; \
    - FD->bk = BK->fd = P; \
    - } \
    - }
* [ 3.5 - Main public routines ]
  + Mục đích cuối cùng của attacker là thực thi code tuỳ ý, chúng ta có thể thực hiện điều đó nhờ vào 2 macro: unlink() và frontlink()
* [ 3.5.1 - The malloc(3) algorithm ]
  + Malloc có tên là \_\_libc\_malloc() trong GNU libc, và mALLOc() trong file malloc.c

1. Được thực thi ở đầu đoạn code được trỏ tới bởi \_\_malloc\_hook pointer nếu (this debugging hook) nó không trỏ tới null. Tiếp theo nó gọi request2size() để chuyển đổi req size, tiếp theo nó sẽ gọi hàm chunk\_alloc(). Tiếp theo nó sẽ dò trong bin tương ứng với req size, nếu nó gặp 1 chunk có “kích thước chính xác” thì chunk đó sẽ được lấy ra.   
   chunk có kích thước chính xác được xác định bởi 0<(sizeofchunk – req\_size)<minsize
   1. Nếu trường hợp size được request nằm trong small bin:
      1. Nếu danh sách liên kết không rỗng thì chunk\_alloc() sẽ chọn chunk cuối cùng của list.
      2. Nếu danh sách này rỗng, và danh sách kế tiếp( next bin) không rỗng thì chunk\_alloc() sẽ chọn chunk cuối cùng của danh sách đó.
      3. Cuối cùng, nếu 1 free chunk được chọn, chunk\_alloc() gọi unlink() để lấy chunk đó ra khỏi danh sách liên kết. Trong trường hợp không có chunk nào được tìm thấy thì sẽ thực hiện bước 2.
   2. Nếu trường hợp size được request > small bin, bin tương ứng sẽ được duyệt từ chunk cuối cùng(chunk nhỏ nhất):
      1. Nếu trong quá trình duyệt gặp chunk quá lớn, thì quá trình duyệt sẽ được dừng vì chunk tiếp theo cũng sẽ lớn hơn chunk đang duyệt. Sau đó nhảy qua bước 2.
      2. Giống bước 1.1.3
2. Đoạn còn lại gần đây nhất(recently remainder) được sử dụng nếu nó đủ lớn. nhưng không phải lúc nào chunk này cũng tồn tại: dlmalloc sẽ gắn [the last\_remainder label] cho 1 free chunk bằng macro link\_last\_remainder() và xoá nó đi bằng macro clear\_last\_remainder(). Vậy nếu 1 chunk được đánh dấu bằng label last\_remainder:
   1. Nếu quá lớn: sizeofchunk – req size>=minsize ,nó sẽ được chia thành 2 phần (), 1 phần có size=req size, sẽ được đưa đến cho mALLOc() và phần còn lại sẽ trở thành last\_remainder mới( thông qua hàm link\_last\_remainder()).
   2. Nếu sizeofchunk-req size < minsize, dlmalloc sẽ gọi macro clear\_last\_remainder() và sau đó:
      1. Nếu sizeofchunk-req size>0, nó sẽ được trả về cho hàm mALLOc().
      2. Nếu sizeofchunk-req size < 0, nó sẽ được đưa vào danh sách liên kết của bin phù hợp ( bằng cách gọi frontlink()) và qua bước 3.
3. Duyệt tất cả các bin còn lại theo thứ tự tăng dần của size, cho đến khi gặp được 1 chunk đủ lớn:
   1. Giống 2.1
   2. Nhưng nếu chunk đó có size vừa đủ chính xác để đáp ứng req size, unlink() sẽ được gọi để lấy chunk đó ra khỏi danh sách liên kết đôi và trả về cho hàm mALLOc() , nếu không có chunk nào phù hợp, gọi bước 4.
4. So sánh wilderness chunk, nếu sizeofchunk-req size>=minsize, wilderness chunk sẽ được chia thành 2 phần, phần đầu tiên có size đáp ứng req size sẽ được trả về cho hàm mALLOc() , phần còn lại sẽ trở thành wilderness chunk mới. Nếu wilderness chunk không đủ đáp ứng, gọi bước 5.
5. Nếu hệ thống hỗ trợ mmap+có 1 vài vùng hiện tại được cấp phát bởi mmap+ gọi mmaped thành công => đáp ứng nhu cầu request = 1 vùng cấp phát bởi mmap. Nếu những điều kiện trên( ví dụ gọi mmap không thành công) thì gọi bước 6
6. Wilderness chunk sẽ được mở rộng bởi hệ thống ( thường thông qua hàm sbrk()) , sau khi cấp phát thành công thì như bước 4.

* [ 3.5.2 - The free(3) algorithm ]
  + Free(3) có tên là \_\_libc\_free() trong GNU C lib, và fREe() trong malloc.c, thực thi bởi \_\_free\_hook nếu nó khác null.

1. Free(0): không thực thi gì.
2. Nếu nó được cấp phát bởi mmap, nó sẽ được giải phóng bởi munmap(). Free() phân biệt chunk có được cấp phát bởi mmap hay không bằng macro chunk\_is\_mmapped(), nếu đúng, gọi munmap\_chunk(), nếu không đúng, gọi chunk\_free().
3. Nếu chunk được freed tiếp nằm ngay trước top-most chunk, nó sẽ được gộp lại với nhau tạo thành top-most chunk mơí.
4. 4.1+ 4.2. Nếu chunk nằm trước/sau của chunk được freed cũng là 1 freed chunk thì chúng sẽ được lấy ra khỏi danh sách liên kết bằng unlink() và được gộp vào chunk được freed.
   1. Chunk được gộp lại được đặt vào danh sách liên kết bằng macro frontlink() hoặc trở thành last\_remainder chunk mới nếu chunk được freed và last\_remainder cũ gộp lại với nhau.

* [ 3.5.3 - The realloc(3) algorithm ]
  + Realloc(3) function được đặt tên là \_\_libc\_realloc() trong GNU C libc and rEALLOc( ) trong malloc.c . Thực thi bởi \_\_realloc\_hook nếu nó khác null.

1. Khi size = 0, nếu #define REALLOC\_ZERO\_BYTES\_FREES được set thì nó sẽ gọi free() thay thế, còn nếu không được set thì nó sẽ cấp phát 1 vùng nhớ có size = minsize. Size # 0 gọi bước 2.
2. Nếu pointer argument = null, dlmalloc sẽ gọi mALLOc() để cấp phát 1 chunk mới.
3. Nếu chunk được cấp phát bởi mmap(),nó sẽ được xử lí khác, còn nếu không, gọi chunk\_realloc().
4. Nếu size mới được yêu cầu < size cũ của chunk:
   1. Nếu oldsize-newsize >= minsize. Nó sẽ được chia thành 2 phần, phần đầu tiên sẽ được trả về cho rEALLOc(), phần còn lại sẽ được giải phóng thông qua chunk\_free().
   2. Nếu 0<=oldsize-newsize< minsize, nó không bị chia mà chỉ đơn giải trả về cho rEALLOc().
5. Nếu size mới > size cũ:
   1. Mở rộng về 1 phía trước:

Nếu chunk liền kề phía sau là 1 chunk free, nó sẽ thực hiện 2 bước sau(trước khi qua 5.2):

* + 1. Nếu chunk đó là top-most chunk(wilderness chunk) và size của top-most chunk + size cũ>req size, thì top-most chunk sẽ được chia thành 2 phần, phần 1 sẽ được gộp chung với chunk cần realloc và trả về cho hàm rEALLOc(), phần còn lại sẽ trở thành top-most chunk mới.
    2. Nếu chunk đó là 1 chunk bình thường, thì nó sẽ được lấy ra khỏi danh sách liên kết bằng unlink() và gộp lại với chunk cần realloc và thực hiện như bước 4.
  1. Mở rộng về 2 phía, thực hiện 2 bước sau( trước khi qua 5.3):
     1. Nếu chunk phía sau của chunk bị realloc là top-most chunk và size của top-most + size của chunk bị realloc+size của chunk trước chunk bị realloc > req size, thì 3 chunk này sẽ được gộp lại với nhau, previous chunk sẽ được unlink() trước khi gộp. Sau đó data của chunk cũ sẽ được copy qua chunk mới, cuối cùng, chunk mới sẽ được chia thành 2 phần, 1 phần trả về rEALLOc() ( có size đáp ứng req) , phần còn lại sẽ trở thành top-most chunk mới.
     2. Nếu chunk phía sau là 1 chunk bình thường, và  
        sizeOfReallocChunk+sizeOfPrevChunk+sizeOfNextChunk > reqsize  
        thì next và previous chunk sẽ được unlink() sau đó gộp thành 1 chunk mới, dữ liệu của realloc chunk sẽ được copy qua chunk mới ,rồi quay lại y bước 4.
  2. Mở rộng về phía sau:  
     nếu chunk phía trước chunk được realloc là 1 free chunk và  
     sizeofpreviousChunk+sizeofReallocChunk > req size, thì unlink(previous\_chunk) sau đó 2 chunk này sẽ được gộp lại thành 1 chunk mới , data của realloc chunk sẽ được copy qua chunk mới, sau đó thực hiện như bước 4.
  3. Nếu chunk bị reallocated không thể mở rộng được, thì chunk\_alloc() sẽ cấp phát 1 chunk mới đúng với req size:
     1. Nếu chunk mới được cấp phát bởi chunk\_alloc() nằm ngay phía sau chunk được reallocated, nó sẽ được gộp lại với nhau và thực hiện thứ bước 4.
     2. Nếu không thì chunk bị reallocate được giải phóng thông qua chunk\_free() , nhưng trước đó data của nó sẽ được copy qua chunk mới được trả về bởi chunk\_alloc(). Cuối cùng, chunk mới đó sẽ được trả về cho hàm rEALLOc().
* [ 3.6 - Execution of arbitrary code ]
* ---[ 3.6.1 - The unlink() technique ]
* ------[ 3.6.1.1 - Concept ]
  + Nếu kẻ tấn công lừa dlmalloc để unlink() 1 fake chunk, họ có thể ghi đè bất kì giá trị vào bộ nhớ mà họ muốn, và thậm chí có thể thực thi lệnh tuỳ ý.

#define unlink( P, BK, FD ) { \

[1] BK = P->bk; \

[2] FD = P->fd; \

[3] FD->bk = BK; \

[4] BK->fd = FD; \

}

* + Nếu attacker có thể tạo ra 1 fake chunk p mà:
    - FD of p= a function Pointer X – 12
    - BK of p = shellcode of address
  + Khi gọi unlink() để lấy chunk ra khỏi danh sách liên kết , nó sẽ ghi đè giá trị BK vào FD+12 ( vì 12 là offset của BK field trong boundary tag)[bước 3]. Tức là khi đó X pointer -> shellcode. Nếu X là 1 địa chỉ trong GOT entry hoặc 1 trong những debugging hook( \_\_malloc\_hook,\_\_free\_hook,etc) , khi chúng được gọi, nó sẽ nhảy tới địa chỉ chúng trỏ tới, và nếu đó là shellcode hợp lệ thì shellcode sẽ được thực thi.
  + Nhưng vì unlink() cũng sẽ ghi đè giá trị FD vào BK+8( 8 là offset của BK trong boundary tag), tức là ghi vào giữa shellcode, vậy nên lệnh đầu tiên của shellcode nên nhảy qua khỏi giá trị bị ghi đè -> shellcode.
* ------[ 3.6.1.2 - Proof of concept ]
  + Thực hành đi 😊