sploitF-U-N

Understanding glibc malloc

Posted on February 10, 2015July 6, 2015 by sploitfun

I always got fascinated by heap memory. Questions such as

> How heap memory is obtained from kernel?
> How efficiently memory is managed?
> Is it managed by kernel or by library or by application itself?
> Can heap memory be exploited?

were in my mind for quite some time. But only recently I got time to understand about it. So here I would like to share my fascination turned knowledge!! Out there in the wild, many memory allocators are available:

- dlmalloc – General purpose allocator
- ptmalloc2 – glibc
- jemalloc – FreeBSD and Firefox
- tcmalloc – Google
- libumem – Solaris
- …

Every memory allocator claims they are fast, scalable and memory efficient!! But not all allocators can be suited well for our application. Memory hungry application's performance largely depends on memory allocator performance. In this post, I will only talk about 'glibc malloc' memory allocator. In future, I am hoping to cover up other memory allocators. Throughout this post, for better understanding of 'glibc malloc', I will link its recent (https://sourceware.org/ml/libc-alpha/2014-09/msg00088.html) source code. So buckle up, lets get started with glibc malloc!!

History: ptmalloc2 (http://www.malloc.de/en/) was forked from dlmalloc (http://g.oswego.edu/dl/html/malloc.html). After fork, threading support was added to it and got released in 2006. After its official release, ptmalloc2 got integrated into glibc source code. Once its integration, code changes were made directly to glibc malloc source code itself. Hence there could be lot of changes between ptmalloc2 and glibc's malloc implementation.

System Calls: As seen in this (https://sploitfun.wordpress.com/2015/02/11/syscalls-used-by-malloc/) post malloc internally invokes either brk (http://man7.org/linux/man-pages/man2/sbrk.2.html) or mmap (http://man7.org/linux/man-pages/man2/mmap.2.html) syscall.

Threading: During early days of linux, dlmalloc was used as the default memory allocator. But later due to ptmalloc2's threading support, it became the default memory allocator for linux. Threading support helps in

improving memory allocator performance and hence application performance. In dlmalloc when two threads call malloc at the same time ONLY one thread can enter the critical section, since freelist data structure is shared among all the available threads. Hence memory allocation takes time in multi threaded applications, resulting in performance degradation. While in ptmalloc2, when two threads call malloc at the same time memory is allocated immediately since each thread maintains a separate heap segment and hence freelist data structures maintaining those heaps are also separate. This act of maintaining separate heap and freelist data structures for each thread is called **per thread arena**.

Example:

```
/* Per thread arena example. */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>

void* threadFunc(void* arg) {
        printf("Before malloc in thread 1\n");
        getchar();
        char* addr = (char*) malloc(1000);
        printf("After malloc and before free in thread 1\n");
        getchar();
        free(addr);
        printf("After free in thread 1\n");
        getchar();
}

int main() {
        pthread_t t1;
        void* s;
        int ret;
        char* addr;

        printf("Welcome to per thread arena example::%d\n",getpid());
        printf("Before malloc in main thread\n");
        getchar();
        addr = (char*) malloc(1000);
        printf("After malloc and before free in main thread\n");
        getchar();
        free(addr);
        printf("After free in main thread\n");
        getchar();
        ret = pthread_create(&t1, NULL, threadFunc, NULL);
        if(ret)
        {
                printf("Thread creation error\n");
                return -1;
        }
        ret = pthread_join(t1, &s);
        if(ret)
        {
                printf("Thread join error\n");
                return -1;
        }
        return 0;
}
```

Output Analysis:

Before malloc in main thread: In the below output we can see that there is NO heap segment yet and no per thread stack too since thread1 is not yet created.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
b7e05000-b7e07000 rw-p 00000000 00:00 0
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

After malloc in main thread: In the below output we can see that heap segment is created and its lies just above the data segment (0804b000-0806c000), this shows heap memory is created by increasing program break location (ie) using **brk** (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L2458) syscall). Also do note that eventhough user requested only 1000 bytes, heap memory of size 132 KB (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L2429) is created. This contiguous region of heap memory is called **arena**. Since this arena is created by main thread its called **main arena** (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1740). Further allocation requests keeps using this arena until it runs out of free space. When arena runs out of free space (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3788), it can grow (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L2458) by increasing program break location (After growing top chunk's size (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L2521) is adjusted to include the extra space). Similarly arena can also shrink (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L4044) when there is lot of free space on top chunk.

NOTE: Top chunk is the top most chunk of an arena. For further details about it, see "Top Chunk" section below.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
...
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/ptmalloc.ppt/mthread$ cat /proc/
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7e05000-b7e07000 rw-p 00000000 00:00 0
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

After free in main thread: In the below output we can see that when allocated memory region is freed, memory behind it doesnt get released to the operating system immediately. Allocated memory region (of size 1000 bytes) is released (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L4017) only to 'glibc malloc' library, which adds this freed block to main arenas bin (In glibc malloc, freelist datastructures are referred as bins). Later when user requests memory, 'glibc malloc' doesnt get new heap memory from kernel, instead it will try to find (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3376) a free block in bin. And only when no free block exists, it obtains memory from kernel.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
...
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/ptmalloc.ppt/mthread$ cat /proc/
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7e05000-b7e07000 rw-p 00000000 00:00 0
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

Before malloc in thread1: In the below output we can see that there is NO thread1 heap segment but now thread1's per thread stack is created.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
Before malloc in thread 1
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7604000-b7605000 ---p 00000000 00:00 0
b7605000-b7e07000 rw-p 00000000 00:00 0          [stack:6594]
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

After malloc in thread1: In the below output we can see that thread1's heap segment is created. And its lies in memory mapping segment region (b7500000-b7521000 whose size is 132 KB) and hence this shows heap memory is created using **mmap** (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/arena.c#L546https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/arena.c#L546) syscall unlike main thread (which uses sbrk). Again here, eventhough user requested only 1000 bytes, heap memory of size 1 MB (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/arena.c#L546) is mapped to process address space. Out of these 1 MB, only for 132KB (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/arena.c#L573) read-write permission is set and this becomes the heap memory for this thread. This contiguous region of memory (132 KB) is called **thread arena** (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/arena.c#L736).

NOTE: When user request size is more than 128 KB (lets say malloc(132*1024)) and when there is not enough space in an arena to satisfy user request, memory is allocated using mmap syscall (and NOT using sbrk) irrespective of whether a request is made from main arena or thread arena.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
Before malloc in thread 1
After malloc and before free in thread 1
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7500000-b7521000 rw-p 00000000 00:00 0
b7521000-b7600000 ---p 00000000 00:00 0
b7604000-b7605000 ---p 00000000 00:00 0
b7605000-b7e07000 rw-p 00000000 00:00 0          [stack:6594]
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

After free in thread1: In the below output we can see that freeing allocated memory region doesnt release heap memory to the operating system. Instead allocated memory region (of size 1000 bytes) is released (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L4017) to 'glibc malloc', which adds this freed block to its thread arenas bin.

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
Before malloc in thread 1
After malloc and before free in thread 1
After free in thread 1
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun/ptmalloc.ppt/
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7500000-b7521000 rw-p 00000000 00:00 0
b7521000-b7600000 ---p 00000000 00:00 0
b7604000-b7605000 ---p 00000000 00:00 0
b7605000-b7e07000 rw-p 00000000 00:00 0          [stack:6594]
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

Arena:

Number of arena's: In above example, we saw main thread contains main arena and thread 1 contains its own thread arena. So can there be a one to one mapping between threads and arena, irrespective of number of threads? Certainly not. An insane application can contain more number of threads (than number of cores), in such a case, having one arena per thread becomes bit expensive and useless. Hence for this reason, application's arena limit is based on number of cores (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/arena.c#L847) present in the system.

```
For 32 bit (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/mal
     Number of arena = 2 * number of cores.
For 64 bit (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/mal
     Number of arena = 8 * number of cores.
```

Multiple Arena:

Example: Lets say a multithreaded application (4 threads – Main thread + 3 user threads) runs on a 32 bit system which contains 1 core. Here no of threads (4) > 2*no of cores (2). Hence in such a case, 'glibc malloc' makes sure that multiple arenas are shared among all available threads. But how its shared?

- When main thread, calls malloc for the first time already created main arena (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1740) is used without any contention.
- When thread 1 and thread 2 calls malloc for the first time, a new arena is created (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/arena.c#L870) for them and its used without any contention. Until this point threads and arena have one-to-one mapping.
- When thread 3 calls malloc for the first time, number of arena limit is calculated (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/arena.c#L847). Here arena limit is crossed, hence try reusing (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/arena.c#L875) existing arena's (Main arena or Arena 1 or Arena 2)
- Reuse:
  - Once loop over (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/arena.c#L810)the available arenas, while looping try (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/arena.c#L805) to lock that arena.
  - If locked successfully (lets say main arena is locked successfully), return (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/arena.c#L826) that arena to the user.
  - If no arena is found free, block (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/arena.c#L819) for the arena next in line.
- Now when thread 3 calls malloc (second time), malloc will try to use last accessed arena (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L2884) (main arena). If main arena is free its used else thread3 is blocked until main arena gets freed. Thus now main arena is shared among main thread and thread 3.

Multiple Heaps:

Primarily below three datastructures are found in 'glibc malloc' source code:

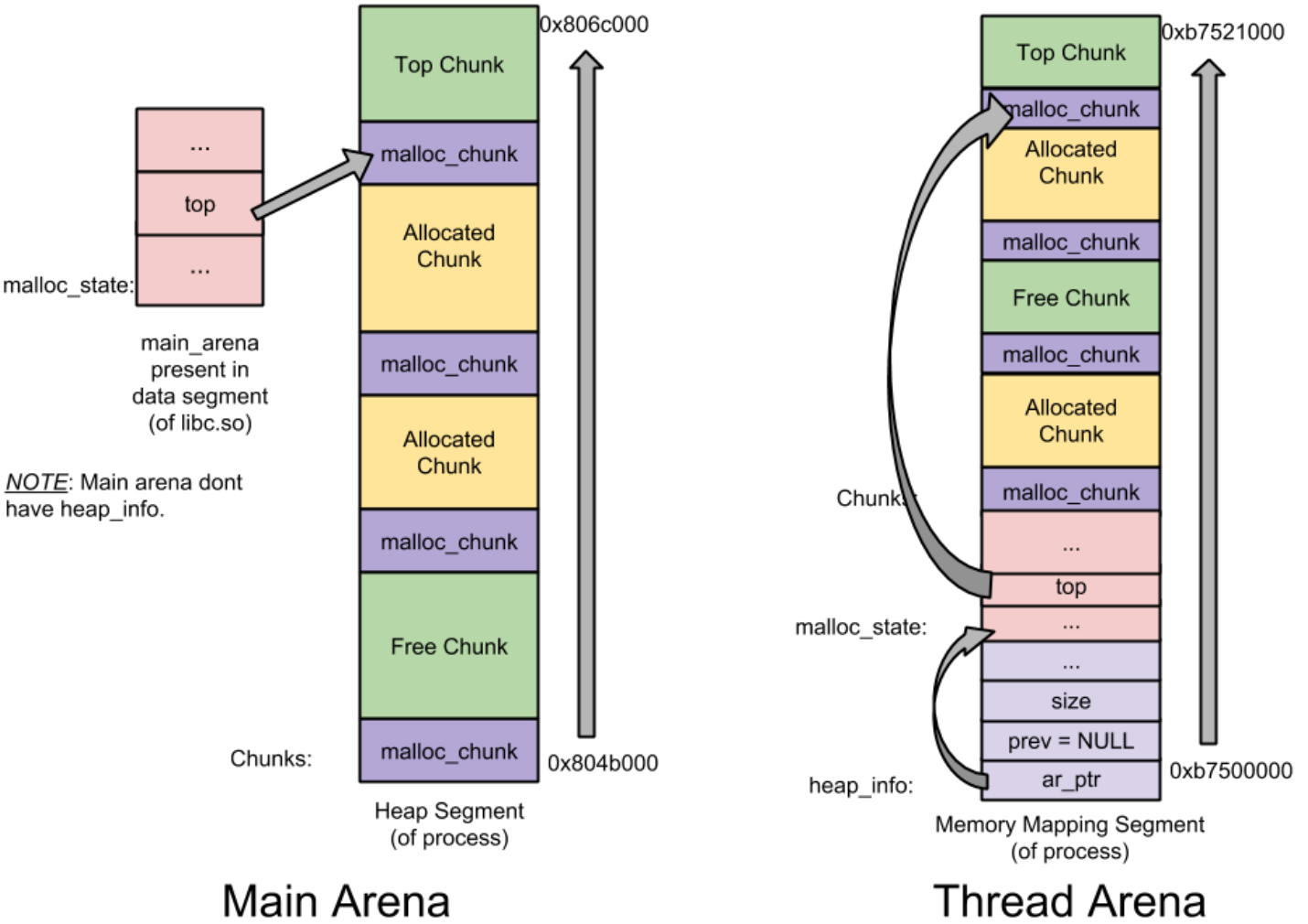
heap_info (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/arena.c#L59) – Heap Header – A single thread arena can have multiple heaps. Each heap has its own header. Why multiple heaps needed? To begin with every thread arena contains ONLY one heap, but when this heap segment runs out of

space, new heap (non contiguous region) gets mmap'd to this arena.
malloc_state (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1671) – Arena Header – A single thread arena can have multiple heaps, but for all those heaps only a single arena header exists. Arena header contains information about bins, top chunk, last remainder chunk…
malloc_chunk (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1108) – Chunk Header – A heap is divided into many chunks based on user requests. Each of those chunks has its own chunk header.

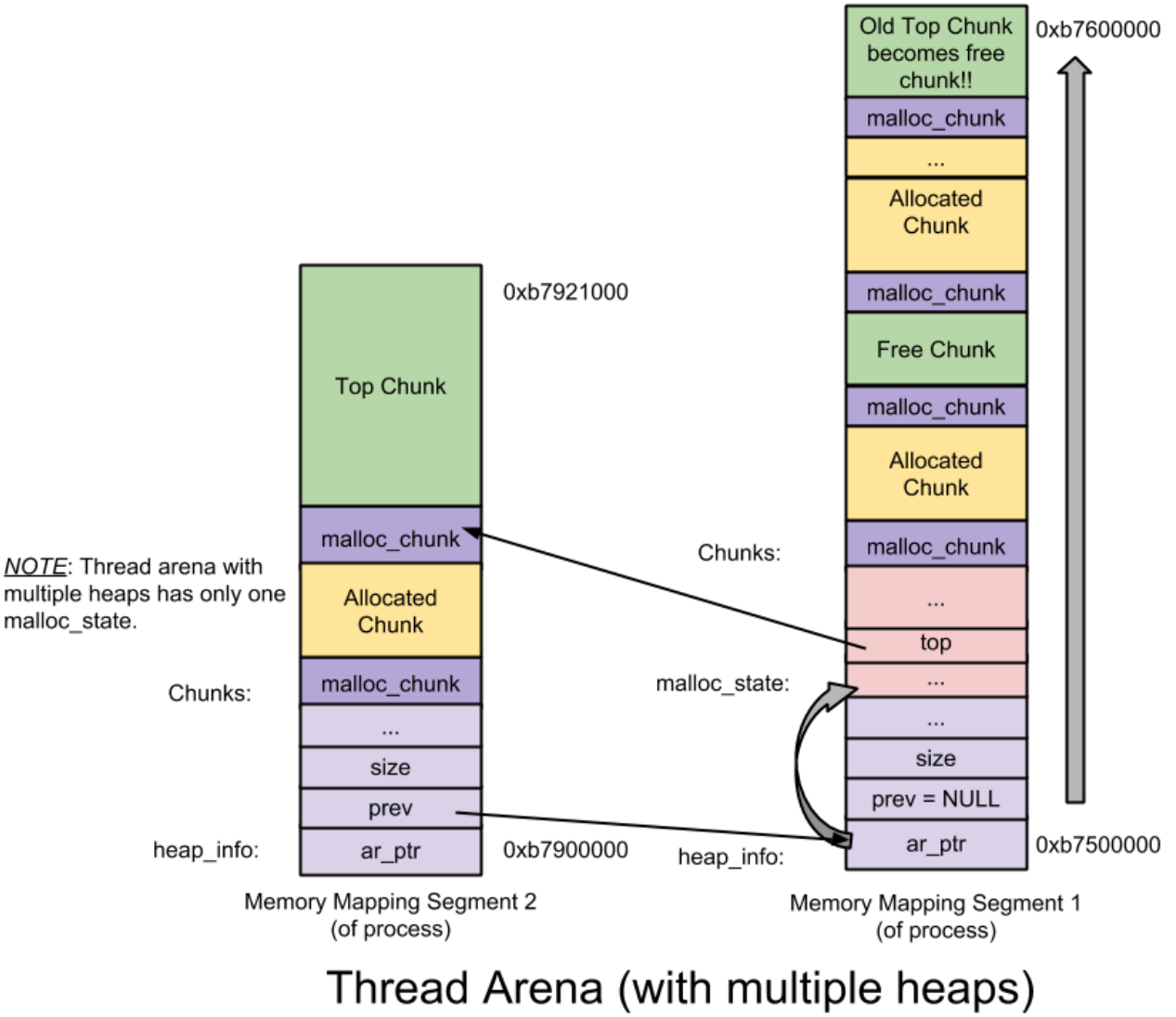
NOTE:

- Main arena dont have multiple heaps and hence no heap_info structure. When main arena runs out of space, sbrk'd heap segment is extended (contiguous region) until it bumps into memory mapping segment.
- Unlike thread arena, main arena's arena header isnt part of sbrk'd heap segment. Its a global variable (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1740) and hence its found in libc.so's data segment.

Pictorial view of main arena and thread arena (single heap segment):



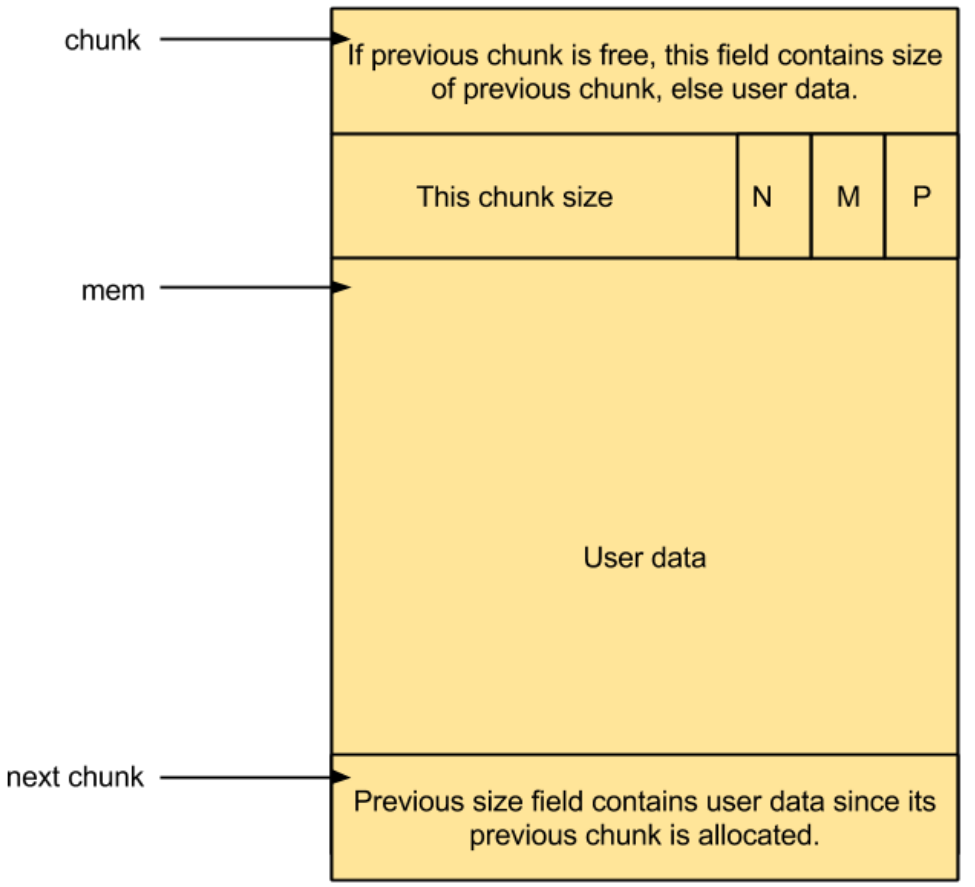
Pictorial view of thread arena (multiple heap segment's):



Chunk: A chunk found inside a heap segment can be one of the below types:

- Allocated chunk
- Free chunk
- Top chunk
- Last Remainder chunk

Allocated chunk:



Allocated Chunk

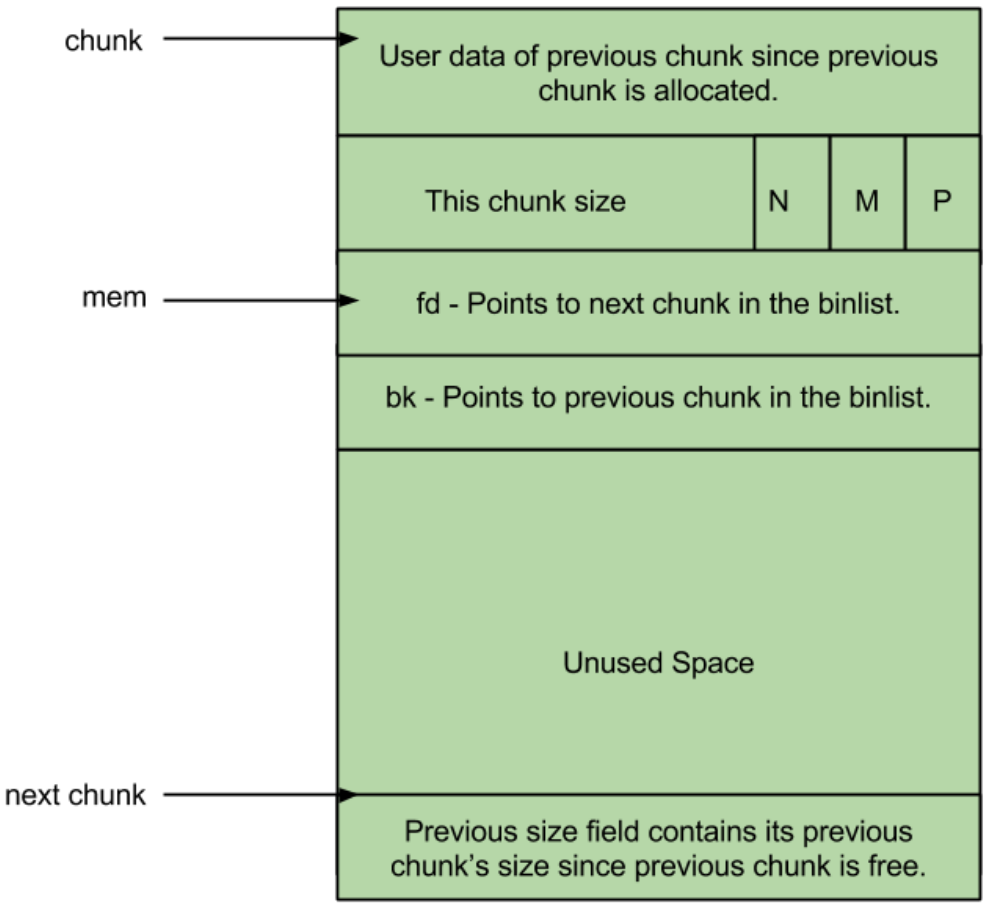
prev_size (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1110): If the previous chunk is free, this field contains the size of previous chunk. Else if previous chunk is allocated, this field contains previous chunk's user data.
size (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1111): This field contains the size of this allocated chunk. Last 3 bits of this field contains flag information.

- PREV_INUSE (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1267) (P) – This bit is set when previous chunk is allocated.
- IS_MMAPPED (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1274) (M) – This bit is set when chunk is mmap'd.
- NON_MAIN_ARENA (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1283) (N) – This bit is set when this chunk belongs to a thread arena.

NOTE:

- Other fields of malloc_chunk (like fd, bk) is NOT used for allocated chunk. Hence in place of these fields user data is stored.
- User requested size is converted (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1254) into usable size (internal representation size) since some extra space is needed for storing malloc_chunk and also for alignment purposes. Conversion takes place in such a way that last 3 bits of usable size is never set and hence its used for storing flag information.

Free Chunk:



Free Chunk

prev_size (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1110): No two free chunks can be adjacent together. When both the chunks are free, its gets combined into one single free chunk. Hence always previous chunk to this freed chunk would be allocated and therefore prev_size contains previous chunk's user data.
size (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1111): This field contains the size of this free chunk.
fd (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1113): Forward pointer – Points to next chunk in the same bin (and NOT to the next chunk present in physical memory).
bk (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1114): Backward pointer – Points to previous chunk in the same bin (and NOT to the previous chunk present in physical memory).

Bins: Bins are the freelist datastructures. They are used to hold free chunks. Based on chunk sizes, different bins are available:

- Fast bin
- Unsorted bin
- Small bin
- Large bin

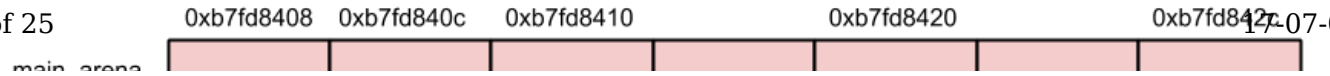
Datastructures used to hold these bins are:

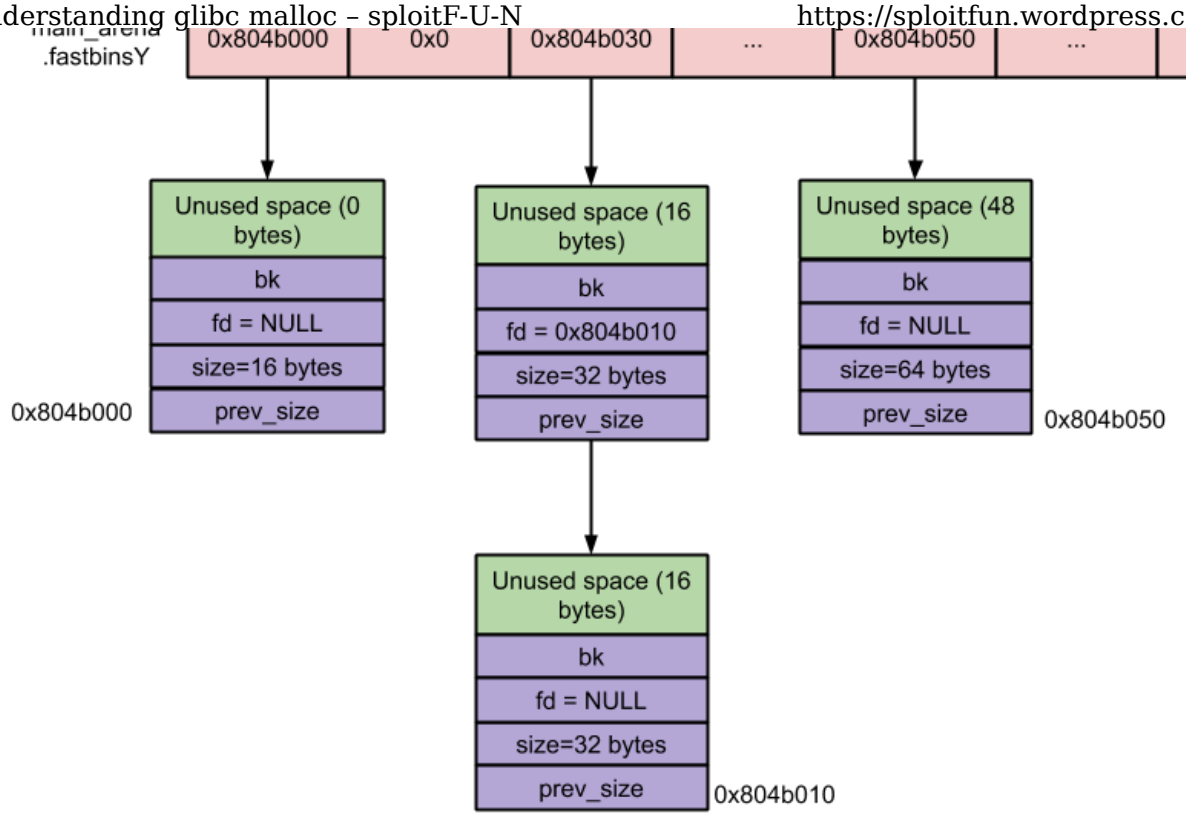
fastbinsY (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1680): This array hold fast bins.
bins (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1689): This array hold unsorted, small and large bins. Totally there are 126 bins and its divided as follows:

- Bin 1 – Unsorted bin
- Bin 2 to Bin 63 – Small bin
- Bin 64 to Bin 126 – Large bin

Fast Bin: Chunks of size 16 (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1249) to 80 (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1600) bytes is called a fast chunk. Bins holding fast chunks are called fast bins. Among all the bins, fast bins are faster in memory allocation and deallocation.

- Number of bins – 10 (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1680)
  - Each fast bin contains a single linked list (a.k.a binlist) of free chunks. Single linked list is used since in fast bins chunks are not removed from the middle of the list. Both addition and deletion happens at the front end of the list – LIFO.
- Chunk size – 8 bytes apart (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1595)
  - Fast bins contain a binlist of chunks whose sizes are 8 bytes apart. ie) First fast bin (index 0) contains binlist of chunks of size 16 bytes, second fast bin (index 1) contains binlist of chunks of size 24 bytes and so on…
  - Chunks inside a particular fast bin are of same sizes.
- During malloc initialization (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1778), maximum fast bin size is set (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1795) to 64 (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L794) (!80) bytes. Hence by default chunks of size 16 to 64 is categorized as fast chunks.
- No Coalescing – Two chunks which are free can be adjacent to each other, it doesnt get combined into single free chunk. No coalescing could result in external fragmentation but it speeds up free!!
- malloc(fast chunk) –
  - Initially fast bin max size (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1765) and fast bin indices (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1680) would be empty and hence eventhough user requested a fast chunk, instead of fast bin code (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3330), small bin code (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3367) tries to service it.
  - Later when its not empty, fast bin index is calculated (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3332) to retrieve its corresponding binlist.
  - First chunk from the above retrieved binlist is removed (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3341) and returned (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3355) to the user.
- free(fast chunk) –
  - Fast bin index is calculated (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3887) to retrieve its corresponding binlist.
  - This free chunk gets added at the front position (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3908) of the above retrieved binlist.

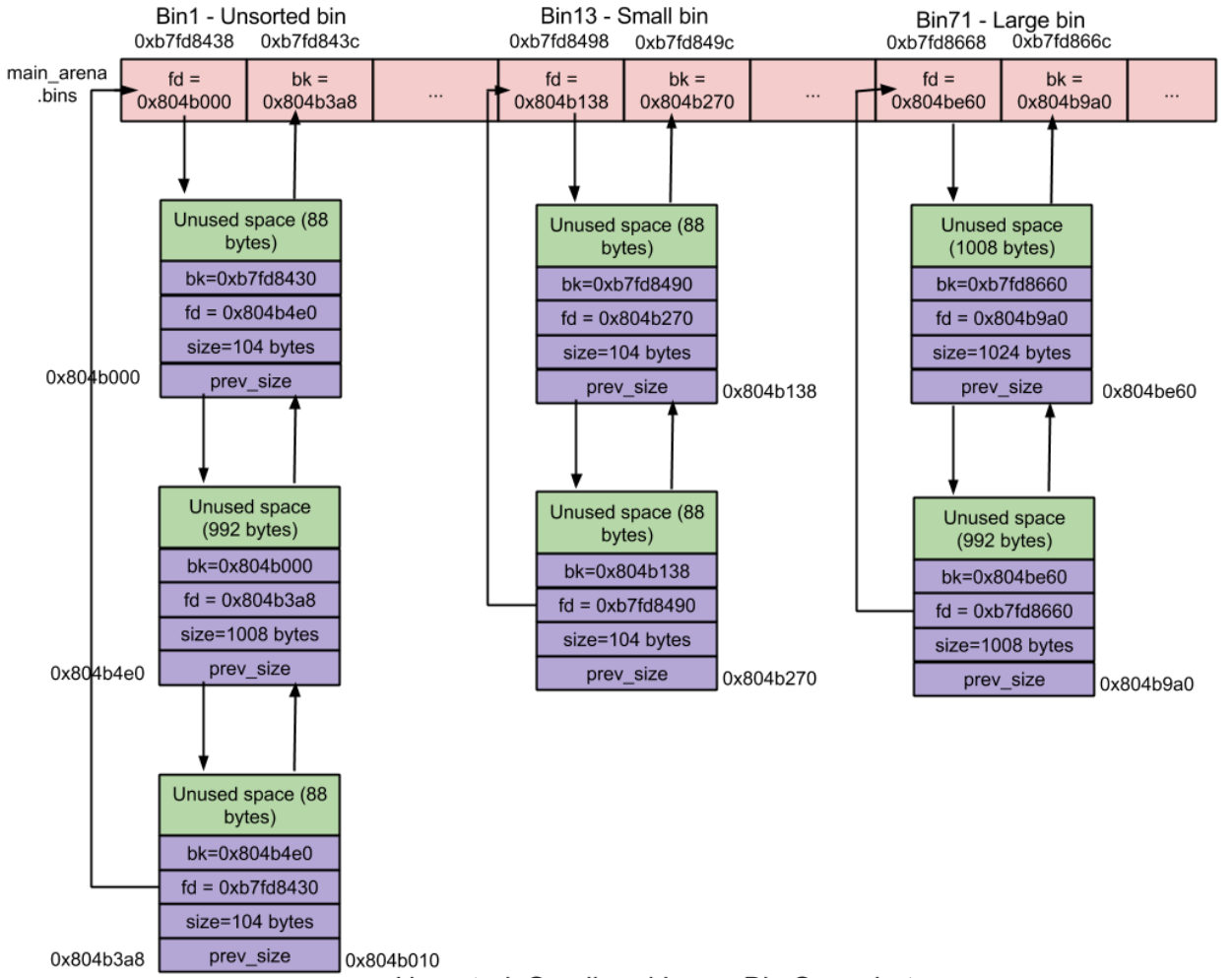




Fast Bin Snapshot

Unsorted Bin: When small or large chunk gets freed instead of adding them in to their respective bins, its gets added (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3988) into unsorted bin. This approach gives 'glibc malloc' a second chance to reuse (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3432) the recently freed chunks. Hence memory allocation and deallocation speeds up a bit (because of unsorted bin) since time taken to look for appropriate bin is eliminated.

- Number of bins – 1
  - Unsorted bin contains a circular double linked list (a.k.a binlist) of free chunks.
- Chunk size – There is no size restriction, chunks of *any* size belongs to this bin.



Unsorted, Small and Large Bin Snapshot

Small Bin: Chunks of size less than 512 (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1470) bytes is called as small chunk. Bins holding small chunks are called small bins. Small bins are faster than large bins (but slower than fast bins) in memory allocation and deallocation.

- Number of bins – 62
  - Each small bin contains a circular double linked list (a.k.a binlist) of free chunks. Double linked list is used since in small bins chunks are unlinked from the middle of the list. Here addition happens at the front end and deletion happens at the rear end of the list – FIFO.
- Chunk Size – 8 bytes apart (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1473)
  - Small bin contains a binlist of chunks whose sizes are 8 bytes apart. ie) First Small bin (Bin 2) contains binlist of chunks of size 16 bytes, second small bin (Bin 3) contains binlist of chunks of size 24 bytes and so on…
  - Chunks inside a small bin are of same sizes and hence it doesnt need to be sorted.
- Coalescing – Two chunks which are free cant be adjacent to each other, it gets combined (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3964) into single free chunk. Coalescing eliminates external fragmentation but it slows up free!!
- malloc(small chunk) –
  - Initially all small bins would be NULL and hence eventhough user requested a small chunk, instead of small bin code (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3367), unsorted bin code (https://github.com/sploitfun/lsploits/blob/master/glibc

/malloc/malloc.c#L3432) tries to service it.
    - Also during the first call to malloc, small bin and large bin datastructures (bins (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1689)) found in malloc_state is initialized ie) bins would point to itself (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3372)signifying they are empty.
    - Later when small bin is non empty, last chunk from its corresponding binlist is removed (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3372) and returned (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3393) to the user.
- free(small chunk) –
    - While freeing this chunk, check if its previous (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3964) or next (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3976) chunk is free, if so coalesce ie) unlink those chunks from their respective linked lists and then add the new consolidated chunk into the beginning of unsorted bin's (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3995)linked list.

Large Bin: Chunks of size greater than equal to 512 is called a large chunk. Bins holding large chunks are called large bins. Large bins are slower than small bins in memory allocation and deallocation.

- Number of bins – 63
    - Each large bin contains a circular double linked list (a.k.a binlist) of free chunks. Double linked list is used since in large bins chunks are added and removed at any position (front or middle or rear).
    - Out of these 63 bins:
        - 32 bins contain binlist of chunks of size which are 64 bytes apart (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1478). ie) First large bin (Bin 65) contains binlist of chunks of size 512 bytes to 568 bytes, second large bin (Bin 66) contains binlist of chunks of size 576 bytes to 632 bytes and so on…
        - 16 bins contain binlist of chunks of size which are 512 bytes apart (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1479).
        - 8 bins contain binlist of chunks of size which are 4096 bytes apart (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1480).
        - 4 bins contain binlist of chunks of size which are 32768 bytes apart (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1481).
        - 2 bins contain binlist of chunks of size which are 262144 bytes apart (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1482).
        - 1 bin contains a chunk of remaining (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1483) size.
    - Unlike small bin, chunks inside a large bin are NOT of same size. Hence they are stored in decreasing order. Largest chunk is stored in the front end while the smallest chunk is stored in the rear end of its binlist.
- Coalescing – Two chunks which are free cant be adjacent to each other, it gets combined (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3964) into single free chunk.
- malloc(large chunk) –
    - Initially (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3571) all large bins would be NULL and hence eventhough user requested a large chunk, instead of large bin code (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3566), next largest bin code (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3639) tries to service it.
    - Also during the first call to malloc, small bin and large bin datastructures (bins) found in malloc_state is initialized (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3411) ie) bins would point to itself signifying they are empty.

- Later when large bin is non empty, if the largest chunk size (in its binlist) is greater than user requested size (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3571), binlist is walked from rear end to front end (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3575), to find a suitable chunk whose size is near/equal to user requested size. Once found, that chunk is split (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3584) into two chunks
  - User chunk (of user requested size) – returned to user.
  - Remainder chunk (of remaining size) – added to unsorted bin.
- If largest chunk size (in its binlist) is lesser than user requested size (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3571), try to service user request by using the next largest (non empty) bin. Next largest bin code scans (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3648) the binmaps to find the next largest bin which is non empty, if any such bin found (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3682), a suitable chunk from that binlist is retrieved, split (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3687) and returned to the user. If not found, try serving user request using top chunk (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3653).

- free(large chunk) – Its procedure is similar to free(small chunk).

Top Chunk: Chunk which is at the top border of an arena is called top chunk (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1683). It doesnt belong to any bin. Top chunk is used to service user request when there is NO free blocks (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3739), in any of the bins. If top chunk size is greater than user requested size (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3760) top chunk is split into two:

- User chunk (of user requested size)
- Remainder chunk (of remaining size)

The remainder chunk becomes the new top (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3762). If top chunk size is lesser than user requested size, top chunk is extended (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3790) using sbrk (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L2458) (main arena) or mmap (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L2390) (thread arena) syscall.

Last Remainder Chunk: Remainder from the most recent split of a small request. Last remainder chunk helps to improve locality of reference ie) consecutive malloc request of small chunks might end up being allocated close to each other.

But out of the many chunks available in an arena, which chunk qualifies to be last remainder chunk?

When a user request of small chunk, cannot be served by a small bin and unsorted bin, binmaps are scanned to find next largest (non empty) bin. As said earlier, on finding the next largest (non empty) bin, its split into two, user chunk gets returned to the user and remainder chunk gets added to the unsorted bin. In addition to it, it becomes the new last remainder chunk (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3721).

How locality of reference is achieved?

Now when user subsequently request's a small chunk and if the last remainder chunk is the only chunk in unsorted bin (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3449), last remainder chunk is split (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc

/malloc.c#L3455) into two, user chunk gets returned to the user and remainder chunk gets added to the unsorted bin. In addition to it, it becomes the new last remainder chunk (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3458). Thus subsequent memory allocations end up being next to each other.

Tagged glibc, heap, malloc, ptmalloc

45 thoughts on “Understanding glibc malloc”

1. **Heap overflow using unlink | sploitF-U-N** says:
   February 26, 2015 at 2:19 pm
   […] Understanding glibc malloc […]

   Reply
2. **Heap overflow using Malloc Maleficarum | sploitF-U-N** says:
   March 4, 2015 at 11:06 am
   […] Understanding glibc malloc […]

   Reply
3. **rurban** says:
   April 6, 2015 at 1:52 pm
   You forgot to mention the improved version, called ptmalloc3.

   Reply
   - **sploitfun** says:
     April 6, 2015 at 2:41 pm
     Yup im yet to look into ptmalloc3, so just avoided it here. Thanks for reading my post!!

     Reply
     - **Avinash Hanwate** says:
       February 6, 2017 at 4:23 am
       can you provide a block diagram of ptmalloc version 3 with data structures used in it?
4. **chris** says:
   April 6, 2015 at 8:08 pm
   Great artical

   Reply
   - **sploitfun** says:
     April 7, 2015 at 11:21 am
     Thanks Chris!!

Reply

5. **ibivmfmvihit** says:
   April 6, 2015 at 11:39 pm
   Hrm. Unless something has changed you're wrong in a few places. You call a section of memory allocated with a mmap call a per thread arena, which is not correct. From memory once you exceed a certain size you always get mmap, which you correctly note, however no additional Metadata is stored for that allocation and when it is freed it just gets passed to munmap.

   Unless something had changed you don't automatically get a new heap per thread.

   Eyeroll. Baltimore.

   Reply

   - **sploitfun** says:
     April 7, 2015 at 11:51 am
     Hmm I don't think so per thread arenas do use mmap!! AFAIK mmap is used in below two places

     1. When non main thread call malloc, mmap is used and this mapped region contains metadata like heap_info and mstate. This space doesn't get released to OS immediately when user does free since even though the space is obtained using mmap syscall, munmap isn't invoked since IS_MMAPPED bit is not set.
     2. When user requested size is greater than threshold size, mmap is used and this mmapped region doesn't contain any metadata (as you rightly pointed out). This space is released to OS immediately when user does free i.e.) munmap gets invoked since IS_MMAPED bit is set.

     I'm no expert in glibc malloc, this is my first attempt reading its source code, so I might be wrong at times, but in this context it looks I'm correct!! If not do comment back, I am happy to rectify it!!

     Reply

6. **S. Rahul** says:
   April 7, 2015 at 2:47 am
   A very well written article. Helps a lot in understanding and visualizing how things work.
   I went through glibc malloc code lots of times while working on some issues at my work place but your explanation made me understand certain things in a much better way. Thanks a lot.

   Reply

   - **sploitfun** says:
     April 7, 2015 at 11:44 am
     Thanks a lot Rahul!! I envy you for looking into glibc malloc code as part of your work, I never got such an opp 😛

     I'm a fan of Gustavo Duarte's blog posts, so just mimicked his style 🙂

     Reply

7. **Guy Fawkes** says:
   April 7, 2015 at 11:57 am
   When talking about the case where there are more threads (4 – main + 3 threads) than 2 * cores (2), I'm not sure I understood your explanation. Why are there 3 arenas (Main Arena, Arena 1, Arena 2) when we only have 1 core?
   I think it should go like this:
   – main thread uses main arena
   – thread 1 uses arena 1

– thread 2 cannot create a new arena and so loops over existing arenas and tries to lock an arena; let's say he locks main arena; now main thread and thread 2 share main arena
– thread 3 cannot create a new arena and so loops over existing arenas and tries to lock an arena; let's say he locks arena 1; now thread 1 and thread 3 share main arena
Am I right?

Also, when talking about large bins, you say there are 2 bins with chunk sizes 256KB apart and one bin with a chunk of the remaining size. How is the size of this largest chunk determined? Initially I thought it was the top chunk, but then I read that the top chunk isn't part of any bin. So what's the deal with this chunk in the largest bin?

Anyway, great article. Thanks for writing it.

Reply

- **sploitfun** says:
  April 8, 2015 at 3:03 pm
  1. Initially while skimming the source code, I was also under the impression that for 32 bit systems, no of arenas = 2 * no of cores but looking closer I found that no of arenas = 2* no of cores **+ 1**. One extra arena gets added because of this condition in arena_get2. Frankly I also dont know the reason behind that one extra arena 🙂

  2. These bins are bit tricky, so to avoid confusions I haven't talked about bin_at and binmaps in the blog post. But to just answer your question, each bin has its own capacity. For instance bin 124 contains chunks of size ranging from 163840 to 262136, while bin 125 contains chunks of size ranging from 262144 to 524280. The last bin 126 contains chunks of size >= 524288.

  For example when we have a chunk of size 162KB it gets stored in bin 124, while chunk of size 260KB gets stored in bin 125. And when we have a chunk of size 515KB it cant be found in bin 2 to 125 since this size exceeds every bin's capacity. Thus such chunks gets stored in the last bin – Bin 126!!

  Incase if you are not very clear about "how chunks of size >= 128KB can get into large bins when they can only be mmapped and NOT brk'd!! See this example.

  Thanks for reading it 🙂

  Reply

8. **Anon** says:
   April 9, 2015 at 8:45 pm
   Hi, great article this is my fourth reading (since m a slow learner and the last part about the different bins is pretty hardcore at least to me =/), its pretty great the only thing that bothers me is the style of the explanations since you first like to explain and make references to future part (like the output analysis) and i got pretty lost on the first try, since i was always thinking "wtf, is this guy talking about… oooh its explaining whats below :facepalm:".

   Kindest Regards

   Reply

   - **sploitfun** says:
     April 10, 2015 at 8:08 am
     Yes there is a lot, I need to improve upon in blogging. Its my early days, so do bear with me 🙂 In future posts I will try to make a definite structure!! Thanks for reading!!

     Reply

9. **In C++, can new in one thread allocate the memory deleted by other thread? - BlogoSfera** says:
   July 4, 2015 at 11:44 pm
   […] In glibc, malloc is implemented with arenas. So, for example, it is possible that the memory first allocated by malloc and later freed in thread A can not be used by another call of malloc in thread B, since thread A and B may be in different arenas, and different arenas maintain different heaps and free lists of memory. […]

   Reply
10. **someone** says:
    September 26, 2015 at 2:01 am
    Thanks for this great article!

    Reply
11. **ryzn** says:
    October 11, 2015 at 1:19 pm
    Great article, help me for understanding glibc malloc very well. Thanks a lot.

    Reply
12. **muhe** says:
    November 19, 2015 at 2:50 pm
    Wonderful! That's help me understand glibc well~ Thx!

    Reply
13. **mathboy7** says:
    January 29, 2016 at 4:33 am
    Hello! Thanks for great article… it helps my study so much.. thank you! 🙂
    I'm a student in Korea. There is no documents about explaining glibc heap structure… so I wanna translate this article into Korean and upload in my blog.
    may I translate this article…? (P.S I'm sorry about my English is not good…:( )

    Reply
    - **sploitfun** says:
      February 1, 2016 at 3:46 am
      Sure go ahead!! Thanks for reading!!!

      Reply
      - **Phenix** says:
        August 9, 2016 at 4:05 am
        It is really a good article. I also want to translate this article into Chinese.

14. **wanchouchou** says:
    March 18, 2016 at 5:30 pm
    Great article! Thank you for share!!!

    Reply
    - **wanchouchou** says:
      March 23, 2016 at 8:13 am
      Great article! I hava learned lots of knowledge about glibc malloc by this article, thank you!
      And I'm a student too, from China, I also want to translate this article into Chinese and display in my blog . May I translate it D:)?

      In addition, I have a question about the range of chunk size of each large bin. In your description, you

said that “First large bin (Bin 65) contains binlist of chunks of size 512 bytes to 568 bytes”. I know why it is start from 512 bytes, by can’t understand why it is end up to 568 bytes instead of 575? I find the ‘glibc malloc’ how to calculate a large bin’s index by requested size, from the source code in malloc.c line 1477, it use this code:
#define largebin_index_32(sz) \
(((((unsigned long) (sz)) >> 6) > 6) 😐
((((unsigned long) (sz)) >> 9) > 9) 😐
((((unsigned long) (sz)) >> 12) > 12) 😐
((((unsigned long) (sz)) >> 15) > 15) 😐
((((unsigned long) (sz)) >> 18) > 18) 😐
126)

It means that if the size is between 512 and 575, the index of large bin was 64, and if the size is between 576 and 639, the index was 65, and so on.. So why the end up size of each large bin, in your description, is 8 bytes less than what I calculated? Is there something else I have not noticed? Thank you.

Reply

15. **Unlink technique | BabyPhD CTF Team** says:
April 8, 2016 at 12:43 pm
[…] REFERENCES: https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/ […]

Reply

16. **Linux堆内存管理深入分析（上） | 信息安全海外资讯** says:
May 13, 2016 at 12:10 am
[…] 前段时间偶然学习了这篇文章：Understanding glibc malloc […]

Reply

17. **tkay** says:
May 26, 2016 at 7:36 am
It is one of the best article about glibc malloc I’ve ever read so far!
Many thanks for the great article!

Reply

18. **SigmaPhi** says:
June 27, 2016 at 1:13 pm
Nowadays,I read glibc malloc about version 2.17,but I dont know when the main arena to be initialized. Before the user first call malloc, what happened? Can you help me,please?

Reply

19. **조힘찬빛** says:
July 3, 2016 at 6:57 am
Hi.
Can I (translate (Article *it, language Korean) && post (Article *it, Article *on_my_blog)) it ?

Reply

- **sploitfun** says:
July 4, 2016 at 3:13 am
Sure.. All yours!!

Reply

20. **The macabre danse of memory chunks | This is Security :: by Stormshield** says:

July 27, 2016 at 1:51 pm
[…] further, we strongly encourage the reader to go through glibc malloc internals. The post made by sploitfun is probably the best documentation on glibc allocator (ptmalloc2). Here we just the recap, the […]

Reply

21. **The macabre dance of memory chunks | This is Security :: by Stormshield** says:
September 16, 2016 at 1:01 pm
[…] further, we strongly encourage the reader to go through glibc malloc internals. The post made by sploitfun is probably the best documentation on glibc allocator (ptmalloc2). Here we just recap the structure […]

Reply

22. **Realtime Memory Allocation研究 | Rebuild Yourself** says:
September 24, 2016 at 12:00 pm
[…] jemalloc (2006) A Scalable Concurrent malloc(3) Implementation for FreeBSD ptmalloc (2006) Understanding glibc malloc Hoard (2000) Hoard: A Scalable Memory Allocator for Multithreaded Applications […]

Reply

23. **laogong** says:
September 26, 2016 at 5:32 am
"When thread 1 and thread 2 calls malloc for the first time, a new arena is created for them and its used without any contention. Until this point threads and arena have one-to-one mapping."
thread 1 and thread 2 will not create two thread arena??

Reply

24. **Savan Patel** says:
November 2, 2016 at 3:41 am
Thanks for wonderful explanation. It helped me to optimize my own implementation
One question I have is,
Why is it 2* number of cores and 8 * number of cores.

What is the reasoning behind choosing 2 and 8 as factor for 32 and 64 bit architecture respectively.

it would be good of you could throw some light over it.

Reply

25. **skysider** says:
November 15, 2016 at 1:18 am
freed small chunk will be always added to unsorted bin? so when a freed chunk will be added to small bin list?

Reply

26. **Kush** says:
November 28, 2016 at 6:53 pm
Very good article Indeed.
I have a query here. How calloc differ from malloc in allocation process ? I mean , Is there any subtle difference between these two except its no. of arguments and zero initialization ?

Reply

27. **S. Rahul** says:

February 13, 2017 at 10:42 am
Atleast in the case of glibc calloc functionality there would not be any difference as it just calls relevant “malloc” routine to allocate memory and does memset.

Reply

27. **Jack Cluts** says:
January 8, 2017 at 10:04 am
Super post! Thanks for the info , saved me a lot of time looking around the web.

Reply

28. **kim** says:
February 4, 2017 at 12:55 pm
glibc how to know heap memory or process vmap which is kernel started process by load_elf_binary() in linux

Reply

29. **Haijun Wang** says:
March 21, 2017 at 3:37 pm
I have two questions about the unsorted, small and large Bin snapshot:
1. At the block which has the address 0x804b3a8, we can obtain fd=0xb7fd8430, but why is its linked address is 0xb7fd8438?
2. The type of Bins is an array of pointers of malloc_chunk, but the type of fd is the pointer of malloc_chunk. why can fd point to an element of Bins?

Reply

30. **PicoCTF 2017 – Enter The Matrix – Nawhack** says:
April 17, 2017 at 7:49 am
[…] https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/ […]

Reply

31. **BCTF – 2017 – Babyuse – Nawhack** says:
April 25, 2017 at 7:11 pm
[…] https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/ […]

Reply

32. **PWN之堆内存管理 | 黑阔资讯 ' Blog** says:
April 26, 2017 at 1:34 am
[…] #很多人引用了这篇文章, 关于堆布局的图都是采用这篇文章里的 Understanding glibc malloc #Phrack #这篇文章很值的读, 虽然里面的一些技术不再适用, […]

Reply

33. **shahril96** says:
May 14, 2017 at 10:12 am
Hi, thanks for the explanation, it really helps!

Can I know what is the meaning of `consolidate` in the context of Glibc malloc? I can’t understand it while you’re explaining it and I usually mixed up
the term with coelesce.

From my understanding, when there are few free chunks in the unsorted bin, the malloc internal code will consolidate (combine) the free chunks and will
add it to the next small or large bins. But this is a coalesce process right?

definitions of:

1) coalesce = combine (elements) in a mass or whole.

2) consolidate = combine (a number of things) into a single more effective or coherent whole.

Both of these have the “combine” word there, which confused me a lot!

Reply