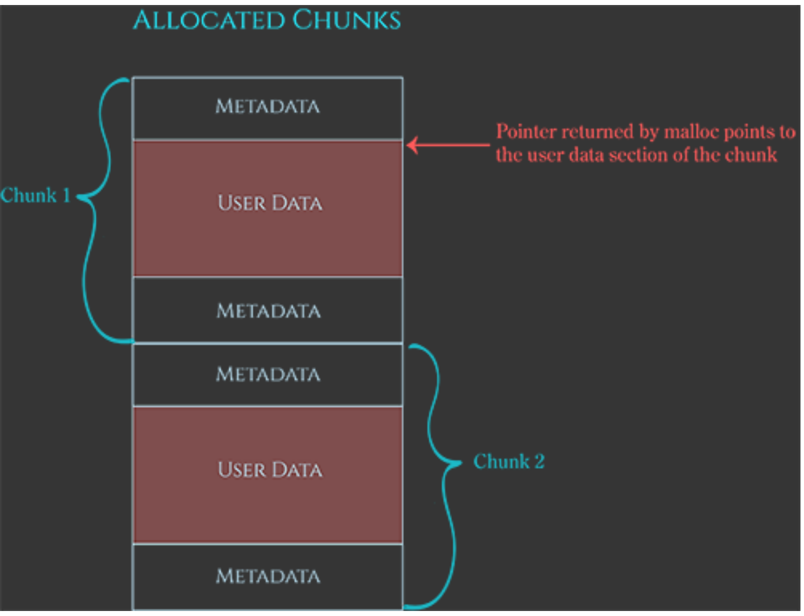
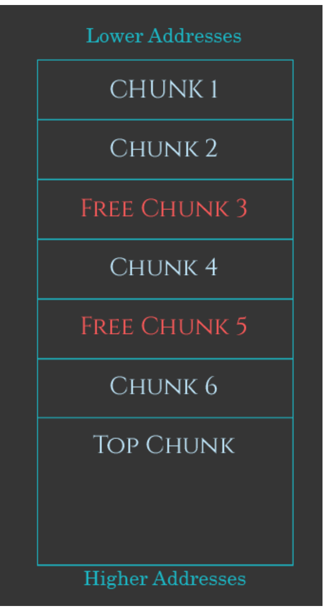
Heap Exploitation Part 2: Understanding the Glibc Heap Implementation

Part 2: Understanding the GLIBC Heap Implementation



Welcome back to this series on understanding and exploiting the *glibc* heap!

In the first part of this series, I explained the basic behavior of *malloc* and *free*. We saw that, under-the-hood, *malloc* handles memory allocation requests by allocating memory *chunks*. Each chunk not only stores the "user data" region returned by malloc that the programmer will interact with, but also metadata associated with that chunk.



We saw how the heap manager's basic chunk-allocation strategy works, and we saw how new chunks get created from the top of the heap when there are no already-freed chunks that can be recycled to service the request.

In this post, I want to talk about how this chunk-recycling strategy works, i.e., how allocations passed back to *free* get saved and eventually recycled to service future *malloc* requests. Lots of heap exploitation techniques rely on exploiting these internal mechanics—we'll look at these in future posts—but for now, let's just look at how these chunks get recycled by *free* when the heap is operating correctly.

How Does Free Work?

When a programmer is finished with an allocation from *malloc* (or a malloc-compatible allocation like *calloc*), the programmer releases it back to the heap manager by passing it to *free*. The C standard defines *free(NULL)* to do nothing, but for all other calls to free, the heap manager's first job is to resolve the pointer back to its corresponding chunk. The heap manager does this by subtracting the size of the chunk metadata from the pointer that was passed to free.

This conversion from pointer to chunk works because the user data region lies inside the chunk, but it is, of course, only valid if the pointer passed to free really is from a live allocation from *malloc*. If some other pointer were passed to free, the heap manager might release or recycle an invalid chunk leading to memory corruption problems that could cause the process to crash or potentially even let hackers remotely takeover the process.

For this reason, *free* first does a couple of basic sanity checks to see if the freed pointer is obviously invalid before attempting to process it. If any of the checks fail, the program aborts. The checks include:

A check that the allocation is aligned on an 8-byte (or 16-byte on 64-bit) boundary, since *malloc* ensures all allocations are aligned.

A check that the chunk's size field isn't impossible–either because it is too small, too large, not an aligned size, or would overlap the end of the process' address space.

A check the chunk lies within the boundaries of the arena.

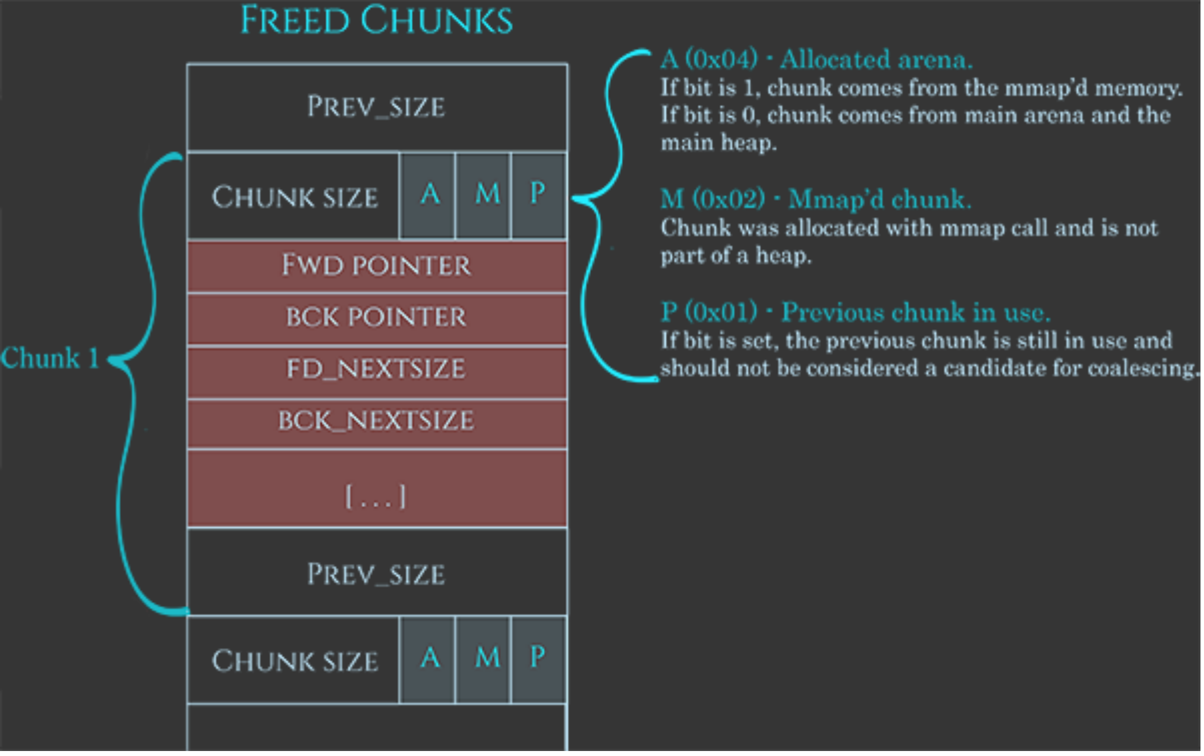
A check that the chunk is not already marked as free by checking the corresponding "P" bit that lies in the metadata at the start of the next chunk.

The heap manager's checks here are not exhaustive; an pointer to data an attacker controls could potentially bypass these sanity checks and still trigger memory corruption in the process. We'll look at this in more detail in later posts.

Free Chunk Metadata

In my last post, I also showed how live chunks store metadata alongside the "user data" region used by the programmer. These live allocations store the "chunk size", and three bits called "A", "M" and "P" in their metadata. Those bits help the heap manager remember if the chunk was allocated from the non-main arena, was allocated off-heap via *mmap*, and whether the previous chunk is free respectively.

Free chunks store metadata too. Like live allocations, they store the "chunk size", "A", and "P" fields, but they do not use the "M" field, since an mmap-ed chunk will always be *munmap-*ed during *free* rather than turned into a free chunk for recycling.


FREED CHUNKS
PREV_SIZE
CHUNK SIZE
A
M
P
FWD POINTER
BCK POINTER
FD_NEXTSIZE
BCK_NEXTSIZE
[...]
PREV_SIZE
CHUNK SIZE
A
M
P
Chunk 1
A (0x04) - Allocated arena.
If bit is 1, chunk comes from the mmap'd memory.
If bit is 0, chunk comes from main arena and the main heap.
M (0x02) - Mmap'd chunk.
Chunk was allocated with mmap call and is not part of a heap.
P (0x01) - Previous chunk in use.
If bit is set, the previous chunk is still in use and should not be considered a candidate for coalescing.

Free chunks also store information *after* the user data region using a technique called "boundary tags". These boundary tags carry size information before and after the chunk. This allows chunks to be traversed starting from any known chunk and in any direction, and thereby enable very fast coalescing of adjacent free chunks.

These freed chunks are stored in corresponding "free bins" that operate as linked lists. This requires each free chunk to also store pointers to other chunks. Since the "user data" in the freed chunk is (by definition) free for use by the heap manager, the heap manager repurposes this "user data" region in freed chunks as the place where this additional metadata lives.

Recycling memory with bins

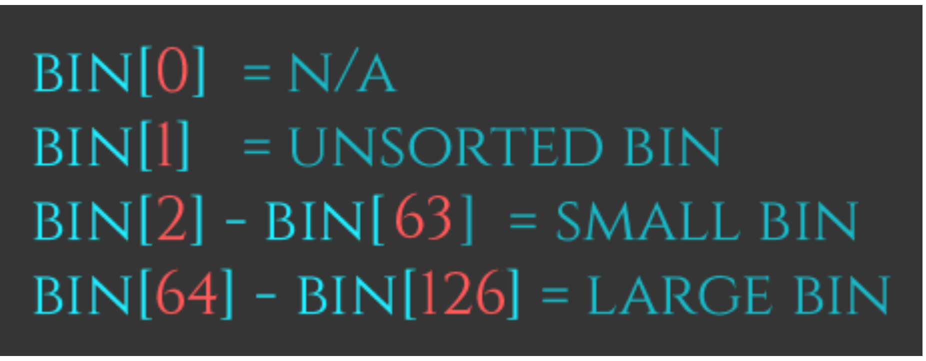
Internally, the heap manager needs to keep track of freed chunks so that malloc can reuse them during allocation requests. In a naive implementation, the heap manager could do this by simply storing all freed chunks together on some enormous linked list. This would work, but it would make *malloc* slow. Since *malloc* is a high-utilization component of most programs, this slowness would have a huge impact on the overall performance of programs running on the system.

To improve performance, the heap manager instead maintains a series of lists called "bins", which are designed to maximize speed of allocations and frees.

There are 5 type of bins: 62 **small bins,** 63 **large bins,** 1 **unsorted bin,** 10 **fast bins** and 64 **tcache bins** per thread.

The small, large, and unsorted bins are the oldest type of bin and are used to implement what I'll refer to here as the *basic* recycling strategy of the heap. The fast bins and tcache bins are optimizations that layer on top of these.

Confusingly, the small, large, and unsorted bins all live together in the same array in the heap manager's source code. Index 0 is unused, 1 is the unsorted bin, bins 2-64 are small bins and bins 65-127 are large bins.



BIN[0] = N/A
BIN[1] = UNSORTED BIN
BIN[2] - BIN[63] = SMALL BIN
BIN[64] - BIN[126] = LARGE BIN

Chunk recycling: the basic strategy

Before getting to *tcache* and *fastbin* optimizations, let’s first look at the basic recycling strategy used by the heap manager.

Recall, the basic algorithm for *free* is as follows:

If the chunk has the *M* bit set in the metadata, the allocation was allocated off-heap and should be *munmap*ed.

Otherwise, if the chunk *before* this one is free, the chunk is merged backwards to create a bigger free chunk.

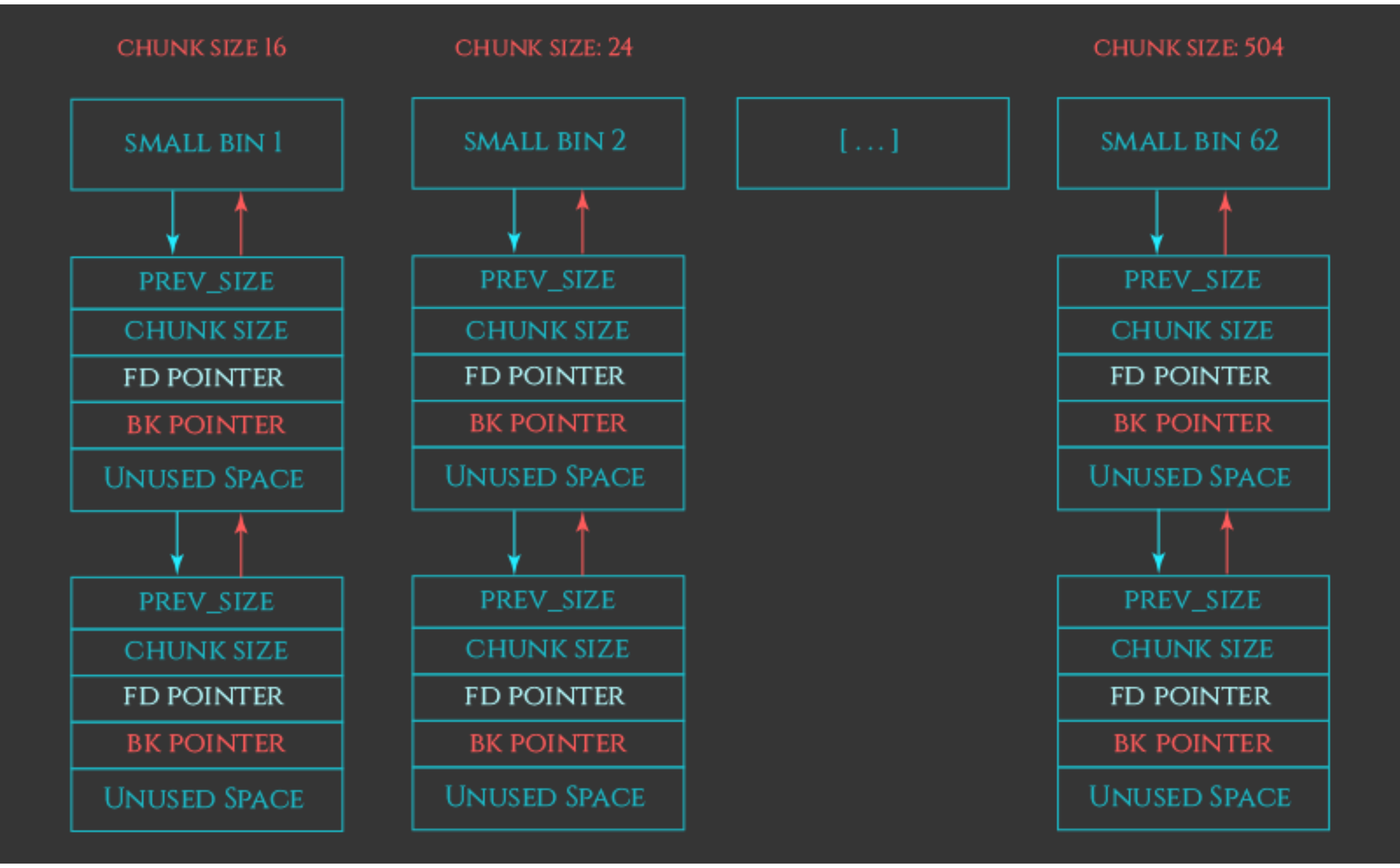
Similarly, if the chunk *after* this one is free, the chunk is merged forwards to create a bigger free chunk.

If this potentially-larger chunk borders the “top” of the heap, the whole chunk is absorbed into the end of the heap, rather than stored in a “bin”.

Otherwise, the chunk is marked as free and placed in an appropriate bin.

Small Bins

Small bins are the easiest basic bin to understand. There are 62 of them, and each small bin stores chunks that are all the same fixed size. Every chunk less than 512 bytes on 32-bit systems (or than 1024 bytes on 64-bit systems) has a corresponding small bin. Since each small bin stores only one size of chunk, they are automatically ordered, so insertion and removal of entries on these lists is incredibly fast.

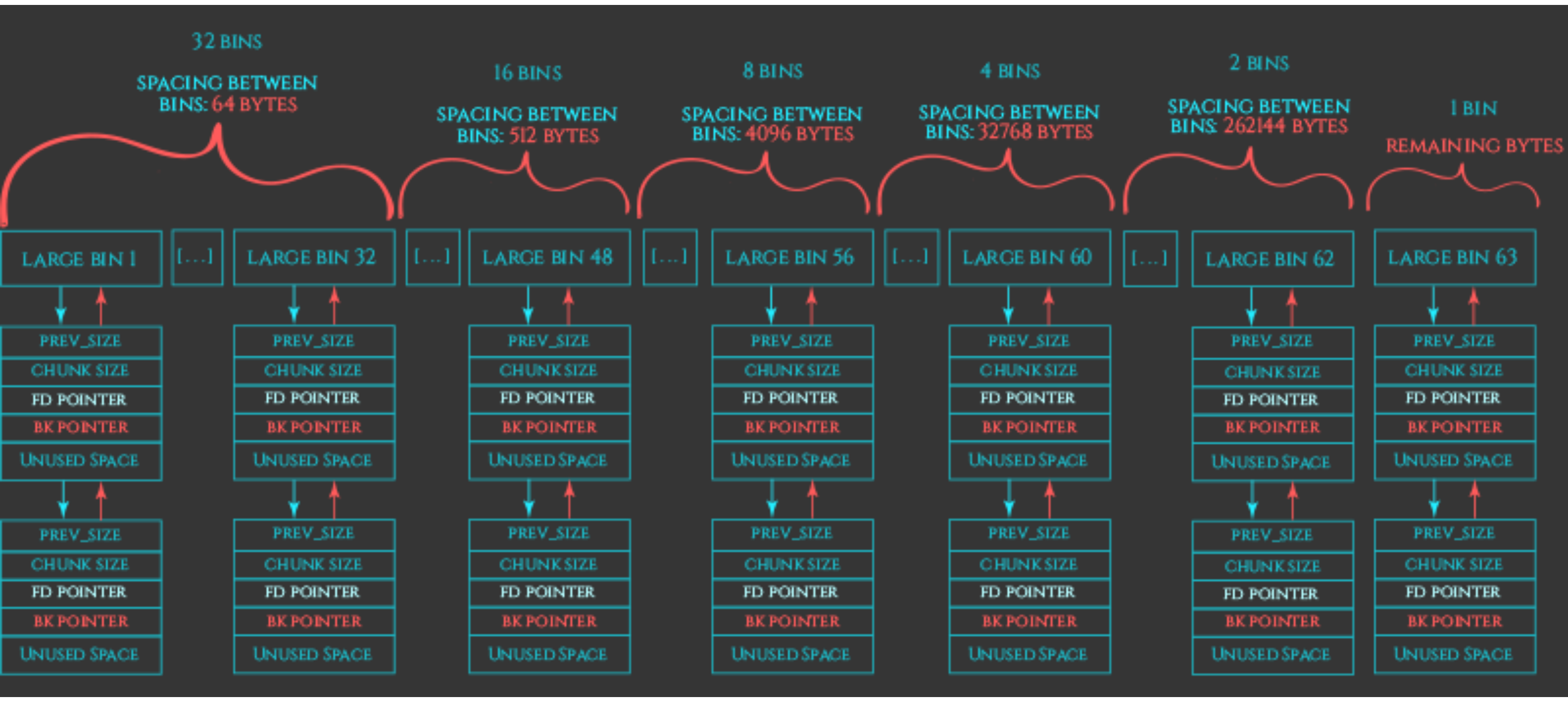


Large Bins

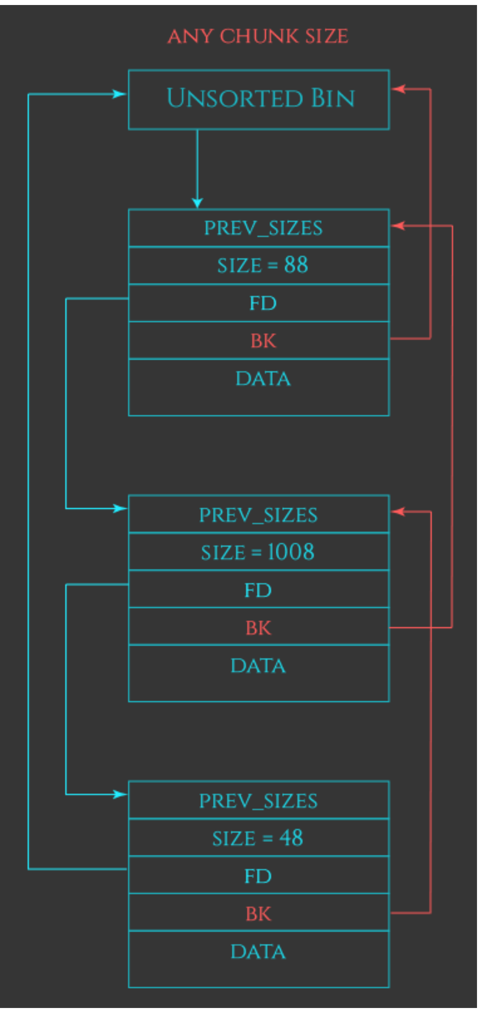
The strategy for small bins is great for small allocations, but we can’t have a bin for every possible chunk size. For chunks over 512 bytes (1024 bytes on 64-bit), the heap manager instead uses “large bins”.

Each of the 63 “large bins” operates mostly the same way as small bins, but instead of storing chunks with a fixed size, they instead store chunks within a size range. Each large bin’s size range is designed to not overlap with either the chunk sizes of small bins or the ranges of other large bins. In other words, given a chunk's size, there is exactly one small bin or large bin that this size corresponds to.

Because large bins store a range of sizes, insertions onto the bin have to be manually sorted, and allocations from the list require traversing the list. This makes large bins inherently slower than their small bin equivalents. Large bins are, however, used less frequently during most programs. This is because programs tend to allocate (and thus release) small allocations at a far higher rate than large allocations on average. For the same reason, "large bin" ranges are clustered towards smaller chunk sizes; the smallest "large bin" covers only the 64-byte range from 512 bytes to 576 bytes, whereas the second largest covers a size range of 256KB. The largest of the large bins covers all freed chunks [above 1MB](#).



Unsorted Bin

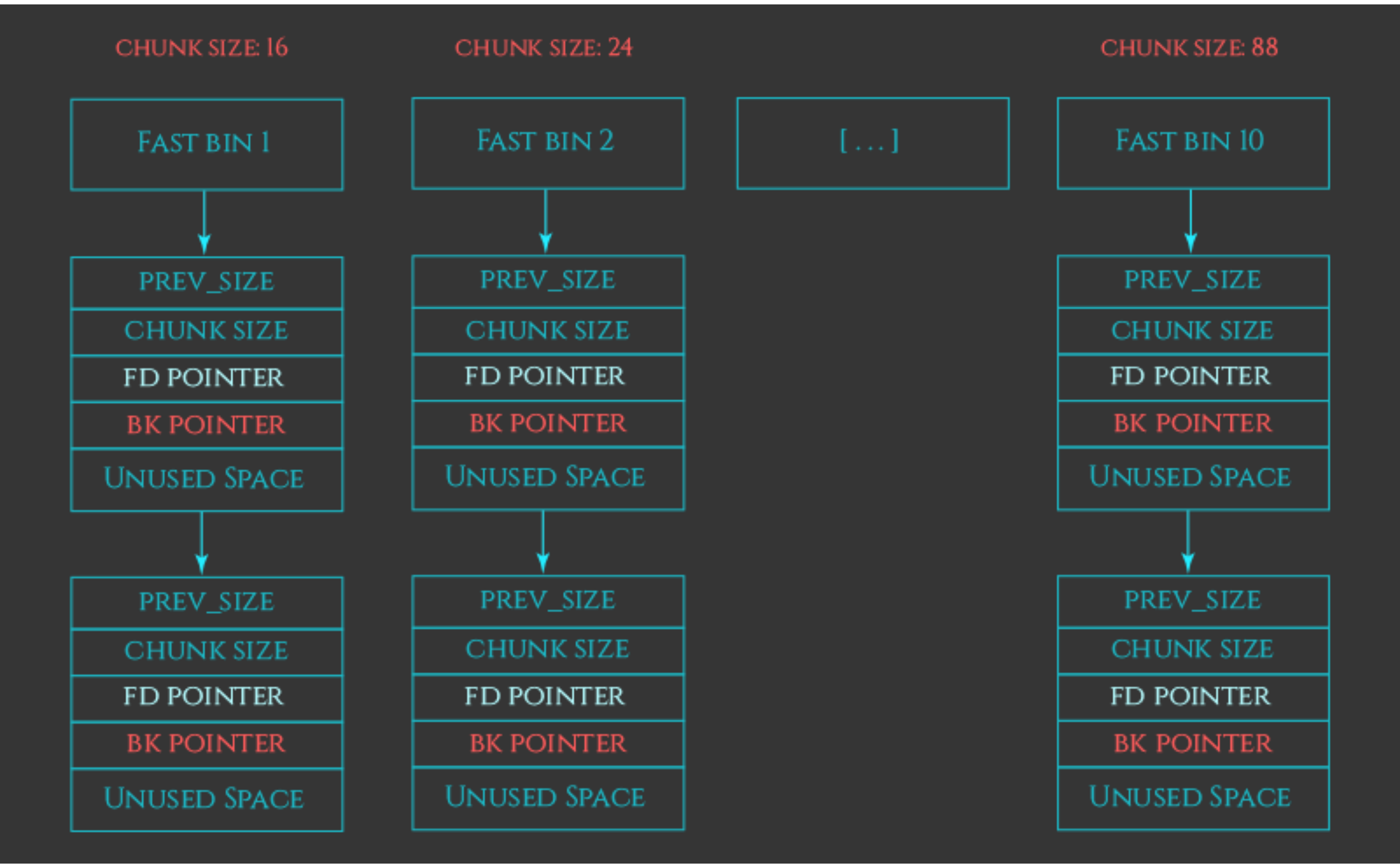


The heap manager improves this basic algorithm one step further using an optimizing cache layer called the “unsorted bin”. This optimization is based on the observation that often frees are clustered together, and frees are often immediately followed by allocations of similarly sized chunks. For example, a program releasing a tree or a list will often release many allocations for every entry all at once, and a program updating an entry in a list might release the previous entry before allocating space for its replacement.

In these cases, merges of these freed chunks before putting the resulting larger chunk away in the correct bin would avoid some overhead, and being able to fast-return a recently freed allocation would similarly speed up the whole process.

The heap manager introduces the *unsorted bin* to take advantage of these observations. Instead of immediately putting newly freed chunks onto the correct bin, the heap manager coalesces it with neighbors, and dumps it onto a general *unsorted* linked list. During *malloc*, each item on the unsorted bin is checked to see if it “fits” the request. If it does, *malloc* can use it immediately. If it does not, malloc then puts the chunk into its corresponding small or large bin.

Fast Bins



Fast bins are a further optimization layered on top of the three basic bins we’ve already seen. These bins essentially keep recently released small chunks on a “fast-turnaround queue”, intentionally keeping the chunk live and not merging the chunk with its neighbors so that it can be immediately repurposed if a malloc request for that chunk size comes in very soon after the chunk is freed.

Like small bins, each fast bin is responsible only for a single fixed chunk size. There are 10 such fast bins, covering chunks of size 16, 24, 32, 40, 48, 56, 64, 72, 80, and 88 bytes plus chunk metadata.

Unlike their small bin cousins, fast bins chunks are never merged with their neighbors. In practice, the way this works is the heap manager doesn’t set the “P” bit at the start of the next chunk. If you like, you can think of this conceptually as the heap manager not “truly” freeing the chunk in the fast-bins.

Like their small bin counterparts, fast bins covers only a single chunk size, are thus automatically sorted, and so insertions and removals are incredibly fast. Moreover, since fast-binned chunks are never merge candidates, they can also be stored in singly-linked lists, rather than needing to be on doubly linked lists so that they can be removed from a list if the chunk gets merged.

The downside of fastbins, of course, is that fastbin chunks are not "truly" freed or merged, and this would eventually cause the memory of the process to fragment and balloon over time. To resolve this, heap manager periodically "consolidates" the heap. This "flushes" each entry in the fast bin by "actually freeing" it, i.e., merging it with adjacent free chunks, and placing the resulting free chunks onto the unsorted bin for malloc to later use.

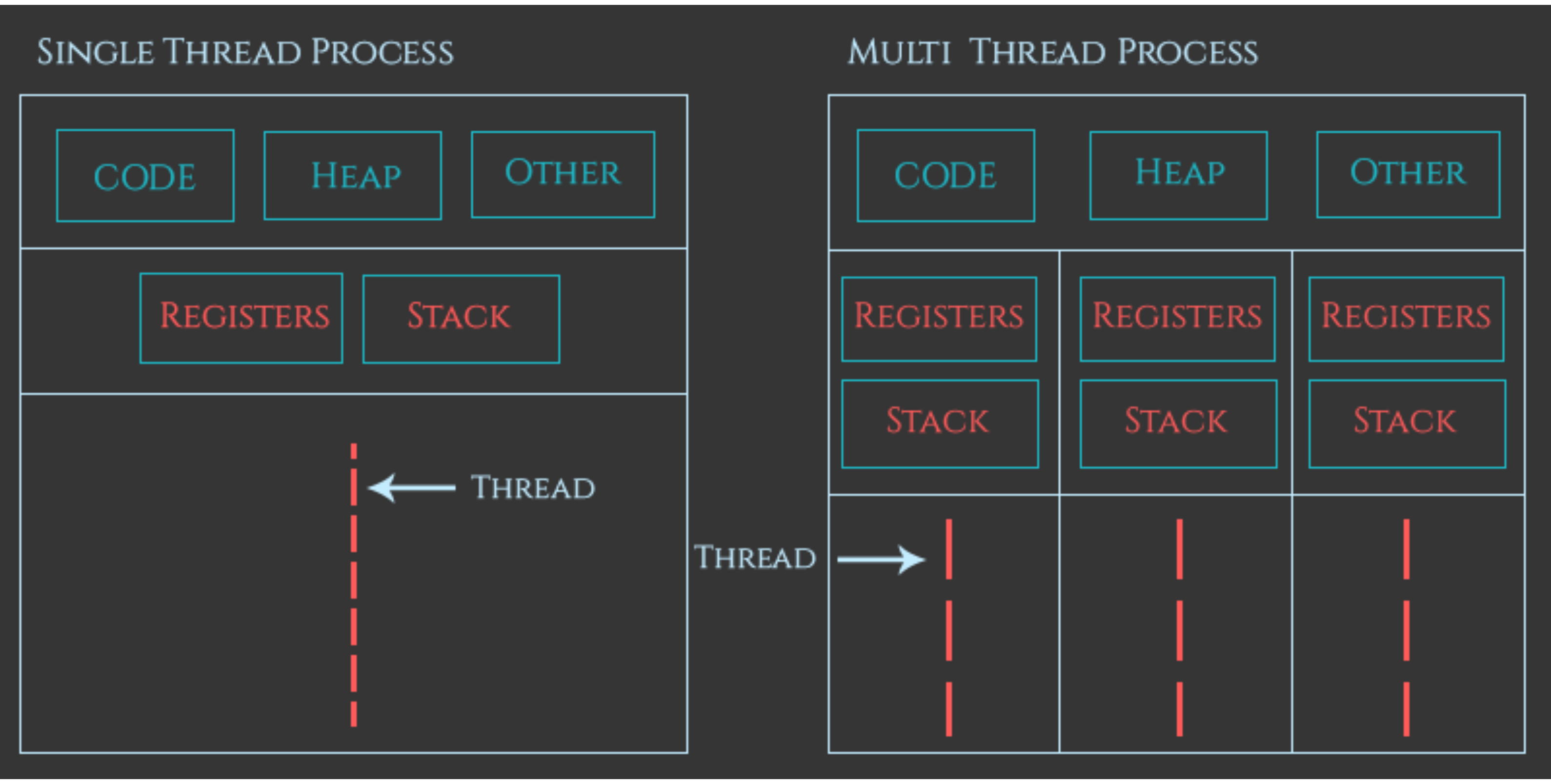
This "consolidation" stage occurs whenever a *malloc* request is made that is larger than a *fastbin* can service (i.e., for chunks over 512 bytes or 1024 bytes on 64-bit), when freeing any chunk over 64KB (where 64KB is a heuristically chosen value), or when malloc_trim or mallopt are called by the program.

Tcache (per-thread cache) bins

The final optimization used by the heap manager to speed up allocations is the per-thread cache, or "*tcache*" allocator. But first let's look at the problem the tcache is trying to solve.

Each process on a given computer system has one or more threads running at the same time. Having multiple threads allows a process to execute multiple concurrent operations. For example, a high-volume web-server might have multiple simultaneous incoming requests, and the web-server might service each incoming request on its own thread, rather than have each request wait in line to be serviced.

Each thread in a given process shares the same address space, which is to say, each thread can see the same code and data in memory. Each thread gets its own registers and stack to store temporary local variables, but resources like global variables and the heap are shared between all of the threads.



Coordinating access to global resources like the heap is a complicated topic, and getting it wrong can lead to a problem known as "*race conditions*", which cause hard-to-debug crashes which are often also exploitable by hackers.

In our example of a website, suppose a web-request being serviced on one thread tries to update a database row, and another concurrent web-request tries to read from that same row. Normally, we'll want to make sure the second thread never sees the row mid-write as it's being overwritten by another thread and thus seeing the row in some partial or corrupted form. Databases solve this problem by making reads and writes appear to operate *atomically*: if two threads try to access the same row at the same time, one operation must complete before the next can begin. A very common way of solving these race-conditions is to force otherwise-simultaneous requests accessing a global resource into a sequential queue by using *locks*.

In general, locks work by one thread "marking" that it has taken ownership of a global resource before using it, then doing its operation, then marking that the resource is no longer in use. If another thread comes along and wants to use the resource and sees some other thread is using it, the thread waits until the other thread is done. This ensures that the global resource is only used by one thread at a time. But it comes with a cost: the thread that is waiting on the resource is stalling and wasting time. This is called "*lock contention*".

For many global variables, this is an acceptable cost. But for the heap which is constantly in use by all threads, this cost quickly translates into the whole program slowing down.

The heap manager mostly solves this problem by using per-thread arenas where each thread gets its own arena until it hits the threshold. Additionally, the *tcache* per-thread cache is designed to reduce the cost of the lock itself because the lock instructions are quite expensive and end up taking a significant portion of the execution time in the fast path. This feature was added to the malloc memory allocation function in glibc 2.26 and is enabled by default.

Per-thread caching speeds up allocations by having per-thread bins of small chunks ready-to-go. That way, when a thread requests a chunk, if the thread has a chunk available on its tcache, it can service the allocation without ever needing to wait on a heap lock.

By default, each thread has 64 singly-linked tcache bins. Each bin contains a maximum of 7 same-size chunks ranging from 24 to 1032 bytes on 64-bit systems and 12 to 516 bytes on 32-bit systems.

How do chunks end up in tcache bins?

When a chunk is freed, the heap manager sees if the chunk will fit into a *tcache* bin corresponding to the chunk size. Like the fast-bin, chunks on the *tcache* bin are considered "in use", and won't be merged with neighboring freed chunks.

If the *tcache* for that chunk size is full (or the chunk is too big for a *tcache* bin), the heap manager reverts to our old slow-path strategy of obtaining the heap lock and then processing the chunk as before.

Corresponding *tcache* allocations are also pretty straightforward. Given a request for a chunk if a chunk is available on an appropriate *tcache* bin, the heap returns that chunk without ever obtaining the heap lock. If the chunk is too big for the *tcache*, we also continue as before.

In the case where we try and make an allocation, there is a corresponding *tcache* bin, but that bin is full, we do a slightly-modified allocation strategy. Rather than just taking the heap lock and finding a *single* chunk, we take the heap lock and opportunistically promote as many chunks as possible at this size to the *tcache* while we still hold the heap lock, up to the *tcache* bin limit of seven, and return the last matching chunk back to the user.

Putting it all together

We now know enough to fully understand the entire behavior of *malloc* and *free* in the *glibc* heap implementation, and why each part of the algorithm exists. Let's recap.

First, every allocation exists as a memory chunk which is aligned and contains metadata as well as the region the programmer wants. When a programmer requests memory from the heap, the heap manager first works out what chunk size the allocation request corresponds to, and then searches for the memory in the following order:

If the size corresponds with a *tcache* bin and there is a *tcache* chunk available, return that immediately.

If the request is enormous allocate a chunk off-heap via *mmap*.

Otherwise we obtain the arena heap lock and then perform the following strategies, in order:

Try the *fastbin/smallbin* recycling strategy

If a corresponding *fast bin* exists, try and find a chunk from there (and also opportunistically prefill the *tcache* with entries from the fast bin).

Otherwise, if a corresponding *small bin* exists, allocate from there (opportunistically prefilling the *tcache* as we go).

Resolve all the deferred frees

Otherwise "truly free" the entries in the fast-bins and move their consolidated chunks to the *unsorted* bin.

Go through each entry in the *unsorted* bin. If it is suitable, stop. Otherwise, put the unsorted entry on its corresponding small/large bin as we go (possibly promoting small entries to the *tcache* as we go).

Default back to the basic recycling strategy

If the chunk size corresponds with a large bin, search the corresponding large bin now.

Create a new chunk from scratch

Otherwise, there are no chunks available, so try and get a chunk from the top of the heap.

If the top of the heap is not big enough, extend it using *sbrk*.

If the top of the heap can't be extended because we ran into something else in the address space, create a discontinuous extension using *mmap* and allocate from there

If all else fails, return NULL.

And the corresponding *free* strategy:

If the pointer is NULL, the C standard defines the behavior as "do nothing".

Otherwise, convert the pointer back to a chunk by subtracting the size of the chunk metadata.

Perform a few sanity checks on the chunk, and abort if the sanity checks fail.

If the chunk fits into a *tcache* bin, store it there.

If the chunk has the *M* bit set, give it back to the operating system via *munmap*.

Otherwise we obtain the arena heap lock and then:

If the chunk fits into a fastbin, put it on the corresponding fastbin, and we're done.

If the chunk is > 64KB, consolidate the fastbins immediately and put the resulting merged chunks on the unsorted bin.

Merge the chunk backwards and forwards with neighboring freed chunks in the small, large, and unsorted bins.

If the resulting chunk lies at the top of the heap, merge it into the top of the heap rather than storing it in a bin.

Otherwise store it in the *unsorted bin*. (*Malloc* will later do the work to put entries from the unsorted bin into the small or large bins).

Looking for a comprehensive training on exploit development?

Sign up for my upcoming Black Hat USA training in Las Vegas, where I will be teaching a 2-day course on “Arm-based IoT Exploit Development”. My 3-day training at Infiltrate is Sold Out. If your company is interested in private trainings and wants to save up to 50% compared to conference trainings, send a message to contact[at]azeria-labs.com.