# HEAPLAB

GLIBC Heap Exploitation Bible

*Max Kamper*

# Table of Contents

# GLIBC

GLIBC is shorthand for the "GNU C Library" and is described on the GLIBC homepage gnu.org/software/libc/ as "[providing] the core libraries for the GNU system and GNU/Linux systems, as well as many other systems that use Linux as the kernel. These libraries provide critical APIs [which include] such foundational facilities as open, read, write, malloc, printf, getaddrinfo, dlopen, pthread_create, crypt, login, exit and more."

The GLIBC project is free software and has been maintained by a dedicated community of developers for over 30 years.

# Malloc

Malloc is the name given to GLIBC's memory allocator. Exploiting malloc has been part of hacker tradition for over 20 years and is still an active field.

Malloc is a collection of functions and metadata that are used to provide a running process with dynamic memory. This metadata consists of arenas, heaps and chunks. Arenas are structures used to administrate heaps. Heaps are large, contiguous blocks of memory which can be broken down into chunks. Malloc's functions use arenas and their heaps to transact chunks of memory with a process.

# Malloc Internals

## Chunks

Chunks are the fundamental unit of memory that malloc deals in, typically they take the form of pieces of heap memory, although they can also be created as a separate entity by a call to mmap(). Chunks consist of a size field followed by user data, as shown below in Figure 1.

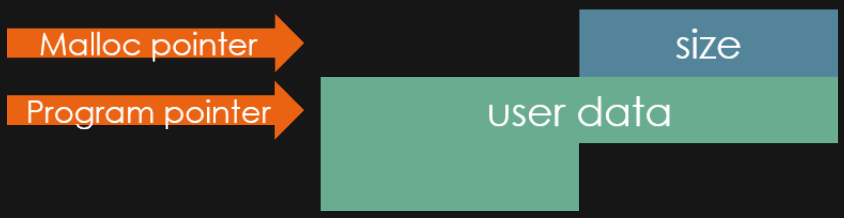



*Figure 1: Chunk layout*

Whilst programs deal with pointers to chunk user data, malloc considers chunks to start 8 bytes before their size field (apart from the tcache, which also uses pointers to user data).

A chunk's size field indicates the amount of user data it has in bytes, plus the number of bytes taken up by the size field itself. A chunk's size field is 8 bytes long, so a chunk with 24 bytes of user data has a size field that holds the value 0x20, or 32. The minimum usable chunk size on x64 architecture is 0x20, although so-called "fencepost chunks" with size 0x10 are used internally by malloc.

On x64 architecture, chunk sizes increase in increments of 16 bytes, so the next size up from a 0x20 chunk is a 0x30 chunk, then a 0x40 chunk etc. This means that the least-significant nybble of a size field is not used to represent chunk size, instead it holds flags that indicate chunk state. These flags are, from least to most significant: PREV_INUSE – when set indicates that the previous chunk is in use, when clear indicates that the previous chunk is free. IS_MMAPPED – when set indicates that this chunk was allocated via mmap(). NON_MAIN_ARENA – when set indicates that this chunk does not belong to the main arena. These flags are shown below in Figure 2.

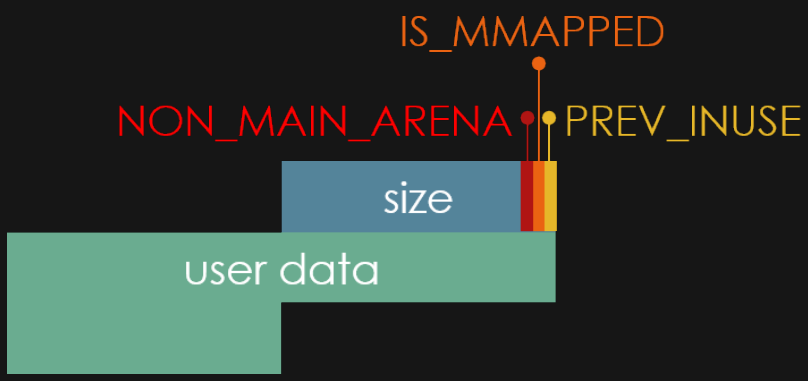



*Figure 2: Size field flags*

A chunk's user data is the memory available to the process that requested it, the address of this user data is returned by malloc's allocation functions.

Chunks are in either of 2 mutually exclusive states: allocated or free. When a chunk is free, up to 5 quadwords of its user data are repurposed as malloc metadata and may even become part of the succeeding chunk. Details on metadata used by each bin is available in the Arenas section.

The 1st quadword of a chunk's user data is repurposed as a forward pointer (fd) when that chunk is freed, all bins use a forward pointer. The 2nd quadword is repurposed as a backward pointer in free chunks linked into any doubly linked list, such as an unsortedbin or smallbin. The 3rd & 4th quadwords are repurposed as fd_nextsize & bk_nextsize pointers, which are used solely by the largebins. The location of this metadata is shown below in Figure 3.

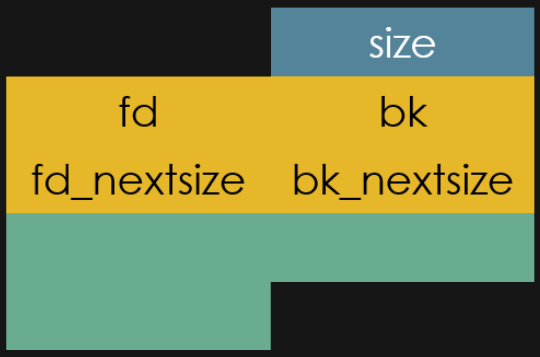



*Figure 3: Inline malloc metadata*

In bins that support consolidation the last quadword of a free chunk's user data is repurposed as a prev_size field, which indicates the size of the freed chunk in the same way as its size field, but without the flags. Malloc considers a prev_size field part of the succeeding chunk and its presence is accompanied by clearing the succeeding chunk's PREV_INUSE flag, as shown in Figure 4.

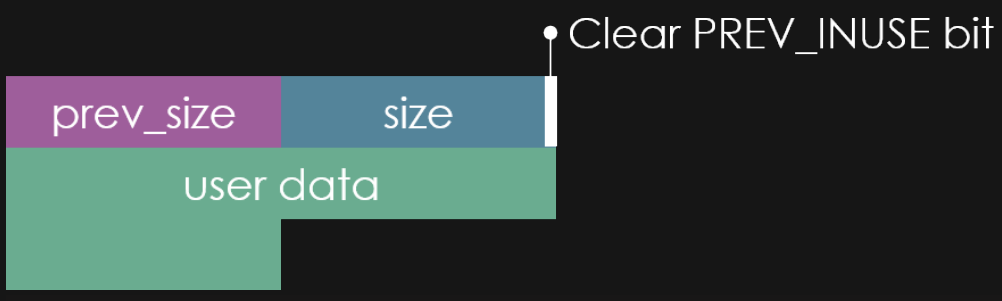



*Figure 4: A prev_size field*

In GLIBC versions >= 2.29 the 2nd quadword of free chunks linked into a tcachebin get repurposed as a “key” field, which is used to detect double-free scenarios. Before GLIBC 2.34 tcache key fields hold the address of their tcache, 2.34 & later use a random value. Figure 5 below shows a free chunk in a tcachebin.

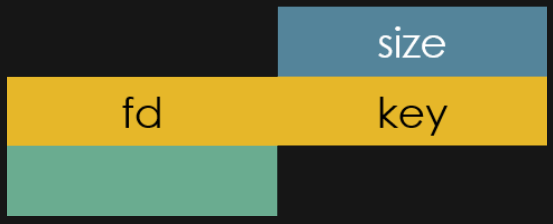



*Figure 5: Tcache metadata*

## Unlinking

During allocation and free operations chunks may need to be unlinked from the free list in which they reside, the chunk being unlinked is often referred to as the “victim” chunk in the malloc source code. See the Arenas section for more information on free lists. There are a variety of ways in which unlinking can occur:

### Fastbin & Tcache Unlink

The fastbins and tcachebins use singly linked, LIFO lists. Unlinking chunks from these lists simply involves copying the victim chunk’s fd into the head of the list. For more information on fastbins see the Arenas section and for more information on the tcache see the Tcache section.

### Partial Unlink

A partial unlink occurs when a chunk is allocated from an unsortedbin or smallbin. The victim chunk’s bk is followed and the address of the head of the bin is copied over the destination chunk’s fd. The victim chunk’s bk is then copied over the bk of the head of the bin. For more information on the unsortedbin and smallbins see the Arenas section.

### Full Unlink

A full unlink occurs when a chunk is consolidated into another free chunk. They also occur when a chunk is allocated from the largebins or via a binmap search. The victim chunk’s fd is followed and the victim bk is copied over the destination bk, then the victim bk is followed and the victim fd is copied over the destination fd.

## Heaps

Heaps are contiguous blocks of memory, chunks of which malloc allocates to a process. They are administrated differently depending on whether they belong to the main arena or not, see the Arenas section for more information on malloc’s arenas.

Heaps can be created, extended, trimmed, or destroyed; a main arena heap is created during the first request for dynamic memory, heaps for other arenas are created via the new_heap() function. Main arena heaps are grown and shrunk via the brk() syscall, which requests more memory from, or returns memory to the kernel. Non-main arena heaps are created with a fixed size and the grow_heap() and shrink_heap() functions map more or less of this space as writable. Non-main arena heaps may also be destroyed by the delete_heap() macro during calls to heap_trim().

## Arenas

Malloc administrates a process's heaps using malloc_state structs, known as arenas. These arenas consist primarily of "bins", used for recycling free chunks of heap memory. A single arena can administrate multiple heaps simultaneously.

New arenas are created via the _int_new_arena() function and initialised with malloc_init_state(). The maximum number of concurrent arenas is based on the number of cores available to a process.

## Arena Layout

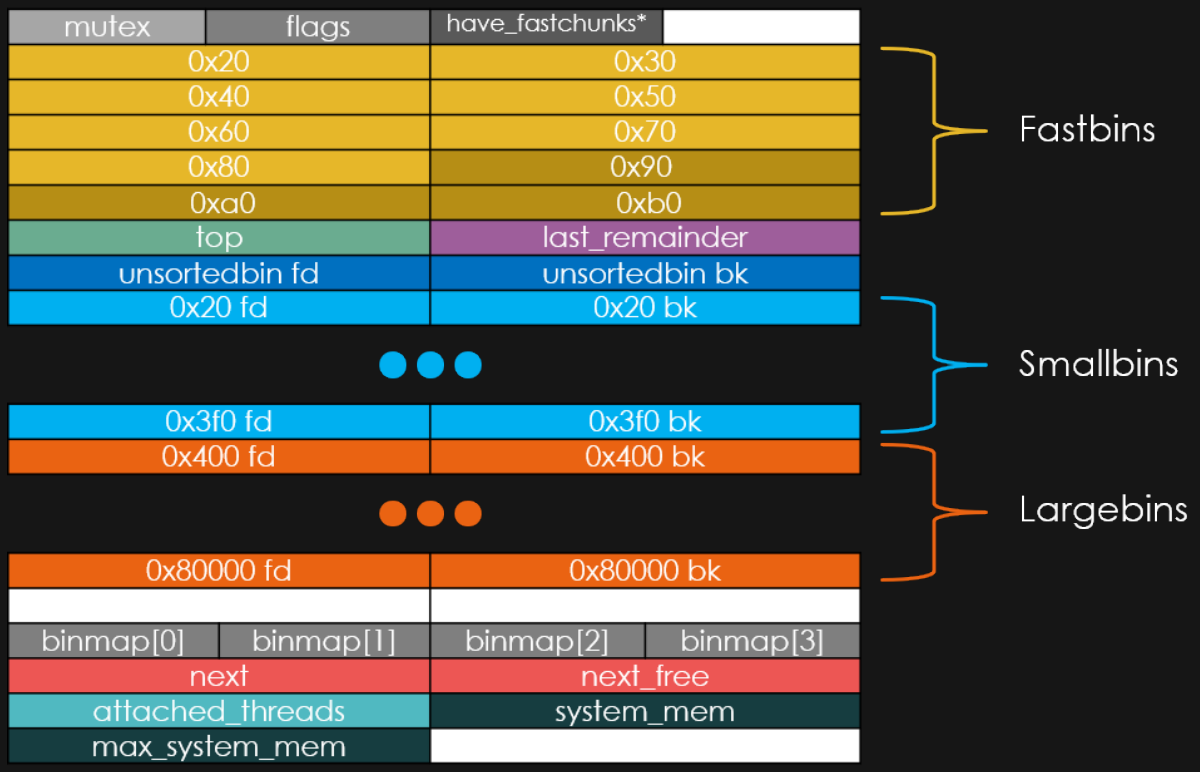



*Figure 6: Arena layout*

### mutex

Serialises access to an arena. Malloc locks an arena's mutex before requesting heap memory from it.

### flags

Holds information such as whether an arena's heap memory is contiguous.

### have_fastchunks

Treated as a bool that indicates the fastbins may not be empty. Set whenever a chunk is linked into a fastbin and cleared by malloc_consolidate().

*N.B. This field and the padding DWORD that follow it are only present in GLIBC versions >= 2.27. In GLIBC versions <= 2.26 it is part of the flags field.*

### Fastbins

The malloc source describes fastbins as "special bins that hold returned chunks without consolidating their spaces". They are a collection of singly linked, non-circular lists that each hold free chunks of a specific size. There are 10 fastbins per arena*, each responsible for holding free chunks with sizes 0x20 through 0xb0. For example, a 0x20 fastbin only holds free chunks with size

0x20, and a 0x30 fastbin only holds free chunks with size 0x30, etc. Although only 7 of these fastbins are available under default conditions, the mallopt() function can be used to change this number by modifying the global_max_fast variable.

The head of each fastbin resides in its arena, although the links between subsequent chunks in that bin are stored inline. The first quadword of a chunk's user data is repurposed as a forward pointer (fd) when it is linked into a fastbin. A null fd indicates the last chunk in a fastbin.

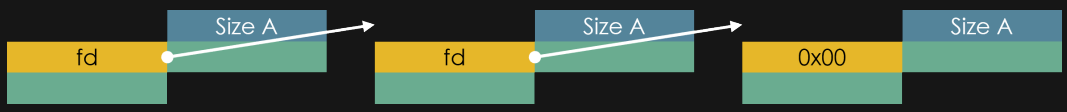



*Figure 7: A fastbin linked list*

Fastbins are last-in, first-out (LIFO) structures, freeing a chunk into a fastbin links it into the head of that fastbin. Likewise, requesting chunks of a size that match a non-empty fastbin will result in allocating the chunk at the head of that fastbin.

Free chunks are linked directly into their corresponding fastbin if their corresponding tcachebin is full. Fastbin searches are conducted after a tcache search and before any other bins are searched, when the request size falls into fastbin range.

*The 0xb0 bin is currently unused, this is because of the disparity between how the MAX_FAST_SIZE constant is treated as a request size versus how the global_max_fast variable is treated as a chunk size.

### Top

From malloc.c: a top chunk is "the topmost available chunk, i.e. the one bordering the end of available memory". After a new arena is initialised, a top chunk always exists and there is only ever one per arena. Requests are only serviced from a top chunk when they can't be serviced from any other bins in the same arena.

When a top chunk is too small to service a request below the mmap threshold, malloc attempts to grow the heap that the top chunk resides on via the sysmalloc() function, then extend the top chunk. If this is unsuccessful, a new heap is allocated and becomes that arena's top chunk, and any remaining memory in the old top chunk is freed. To achieve this, malloc places 2 0x10-sized "fencepost" chunks at the end of the heap to ensure forward consolidation attempts don't result in an out-of-bounds read. The functions used to administrate heaps are listed in the Heaps section.

Malloc keeps track of the remaining memory in a top chunk using its size field, the prev_inuse bit of which is always set. A top chunk always contains enough memory to allocate a minimum-sized chunk and always ends on a page boundary.

### Last_remainder

This field holds the address of the chunk resulting from the previous remainder operation. It is populated by requests that fall into smallbin range that remainder from an unsortedbin (from an existing last_remainder), or from a binmap search.

The last_remainder field is not populated from largebin remainders, nor from unsortedbin or binmap searches when the request size was outside of smallbin range.

To remainder from an unsortedbin, the last remainder chunk must be at the head of the unsortedbin. Learn more about remaindering in the Remaindering section.

## Unsortedbin

An unsortedbin is a doubly linked, circular list that holds free chunks of any size. The head and tail of an unsortedbin reside in its arena, whilst fd & bk links between subsequent chunks in the bin are stored inline on a heap.

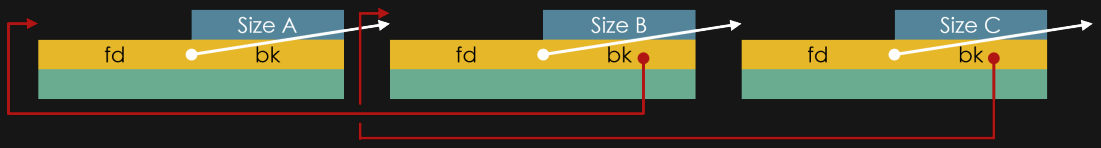



*Figure 8: Unsortedbin doubly linked list*

Free chunks are linked directly into the head of an unsortedbin when their corresponding tcachebin is full, or they are outside tcache size range (0x420 & above under default conditions). In versions of GLIBC compiled without the tcache (GLIBC versions <= 2.25 by default) free chunks are linked directly into the head of an unsortedbin when they are outside fastbin size range (0x90 & above under default conditions).

An unsortedbin is searched after the tcache, fastbins, and smallbins when the request size falls into those ranges, but before the largebins. Unsortedbin searches start from the tail of the bin and work their way towards the head, if a chunk exactly fits the normalized request size it is allocated and the search stops, otherwise it is sorted into its appropriate smallbin or largebin.

If the chunk being checked during an unsortedbin scan isn't an exact fit but is the last remainder and large enough to remainder again, it is remaindered. Chunks that are the result of this remainder operation are linked back into the head of the unsortedbin. For more information on unsortedbin searches see the Unsortedbin Flowchart.

## Smallbins

The smallbins are a collection of doubly linked, circular lists that each hold free chunks of a specific size. There are 62 smallbins per arena, each responsible for holding free chunks with sizes 0x20 through 0x3f0, overlapping the fastbin sizes. For example, a 0x20 smallbin only holds free chunks with size 0x20, and a 0x3c0 smallbin only holds free chunks with size 0x3c0, etc.

The head of each smallbin resides in its arena, although the links between subsequent chunks in that bin are stored inline. Free chunks are only linked into their corresponding smallbin via its arena's unsortedbin, when sorting occurs. When a chunk is linked into a smallbin, the 1$^{st}$ quadword of its user data is repurposed as a forward pointer (fd) and the 2$^{nd}$ quadword is repurposed as a backward pointer (bk).

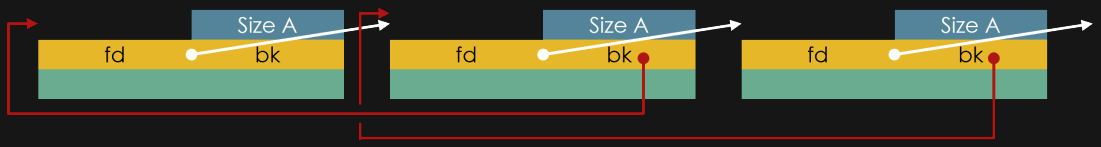



*Figure 9: Smallbin doubly linked list*

Smallbins are first-in, first-out (FIFO) structures, sorting a chunk into a smallbin links it into the head of that smallbin. Likewise, requesting chunks of a size that match a non-empty smallbin will result in allocating a chunk from the tail of that smallbin.

Smallbin searches are conducted after a tcache search, after a fastbin search if the request was in fastbin range, but before any other bins are searched, when the request size falls into smallbin range.

## Largebins

The largebins are a collection of doubly linked, circular lists that each hold free chunks within a range of sizes. There are 63 largebins per arena, each responsible for holding free chunks with sizes 0x400 and up. For example, a 0x400 largebin holds free chunks with sizes between 0x400 – 0x430, whereas a 0x2000 largebin holds chunks with sizes between 0x2000 – 0x21f0.

The head of each largebin resides in its arena, although the links between subsequent chunks in that bin are stored inline. Free chunks are only linked into their corresponding largebin via its arena's unsortedbin, when sorting occurs.

Largebins are maintained in descending size order, with the largest chunk in that bin accessible via the bin's fd pointer, and the smallest chunk accessible via its bk. When a chunk is linked into a largebin, the 1st quadword of its user data is repurposed as a forward pointer (fd) and the 2nd quadword is repurposed as a backward pointer (bk).


*Figure 10: Largebin doubly linked list with skip list*

The first chunk of its size to be linked into a largebin has the 3rd & 4th quadwords of its user data repurposed as skip list pointers, fd_nextsize & bk_nextsize respectively. These nextsize pointers form another doubly linked, circular list which holds the first chunk of each size linked into that bin. Once the first chunk of its size has been linked into a largebin, subsequent chunks of the same size are added after the first chunk of that size to avoid rerouting the skip list.

The largebins are searched during requests for chunks with size 0x400 and above, after an unsortedbin scan, but before a binmap search. During a largebin search, malloc ensures that the appropriate bin holds a chunk that is large enough to support the request; if so, the bin is scanned from back to front for a chunk that is either an exact fit or larger. Malloc will only allocate a chunk that holds skip list pointers if it is the last chunk of its size, otherwise it allocates the chunk of the same size after the skip chunk, this avoids having to reroute the skip list too often.

Any non-exact fitting allocations from a largebin are exhausted or remaindered, but the last_remainder field is not set.

## Binmap

The binmap is bit vector that loosely represents which of an arena's smallbins & largebins are occupied. It is used by malloc to quickly find the next largest, occupied bin when a request couldn't be serviced from its appropriate bins.

Binmap searches occur after an unsuccessful unsortedbin or largebin search, depending on request size. Malloc finds the next largest, occupied bin and exhausts/remainders the last chunk in that bin, in the latter case the remainder is advertised as the last remainder if the request was in smallbin range.

A bin is marked as occupied when a chunk is sorted into it during an unsortedbin scan. A bin is marked as empty when a binmap search finds an empty bin that was marked as occupied.

### next
A singly linked, circular list of all arenas belonging to this process.

### next_free
A singly linked, non-circular list of free arenas (arenas with no threads attached). The head of this list is the free_list symbol.

### attached_threads
The number of threads concurrently using this arena.

### system_mem
The total writable memory currently mapped by this arena.

### max_system_mem
The largest amount of writable memory this arena had mapped at any one time. Used by calloc() to determine if freshly mapped heap memory needs to be zeroed.

## Remaindering
Remaindering is simply the term malloc gives to splitting one free chunk down into two smaller chunks, then allocating the appropriate chunk. The remaining chunk is linked into the unsortedbin belonging to the chunk's arena.

For example, during a request for a 0x100-sized chunk, if the thread's arena only has a 0x300-sized chunk to offer, malloc will unlink the 0x300 chunk from its free list, split a 0x100 chunk off from it, link the so-called remainder (a 0x200-sized chunk) into the head of the unsortedbin and allocate the 0x100 chunk.

Remaindering can occur at one of 3 points in the malloc flowchart: during allocations from the largebins, during a binmap search, and from a last remainder during unsortedbin scanning.

## Exhausting
In the case that a thread requests a 0x80-sized chunk, and its arena only has a 0x90-sized chunk available, malloc will "exhaust" the 0x90 chunk by allocating the whole thing rather than remaindering it. This is because there isn't enough space left over for a minimum sized chunk after taking 0x80 bytes away from a 0x90 chunk.

## Malloc Parameters

The malloc parameters structure holds variables that dictate how malloc operates, it is defined as a malloc_par struct:

```
struct malloc_par {
	/* Tunable parameters. */
	unsigned long trim_threshold;
	INTERNAL_SIZE_T top_pad;
	INTERNAL_SIZE_T mmap_threshold;
	INTERNAL_SIZE_T arena_test;
	INTERNAL_SIZE_T arena_max;
	/* Memory map support. */
	int n_mmaps;
	int n_mmaps_max;
	int max_n_mmaps;
	int no_dyn_threshold;
	/* Statistics. */
	INTERNAL_SIZE_T mmapped_mem;
	INTERNAL_SIZE_T max_mmapped_mem;
	char* sbrk_base;
#if USE_TCACHE
	size_t tcache_bins;
	size_t tcache_max_bytes;
	size_t tcache_count;
	size_t tcache_unsorted_limit;
#endif
};
```

Descriptions of each mp_ field follow, default values are for x64 architecture.

**trim_threshold** – Maximum top chunk size before it's trimmed on free(), default 0x20000.

**top_pad** – How much extra memory to request when enlarging a top chunk, default 0x20000.

**mmap_threshold** – Minimum chunk size serviced by mmap(), default 0x20000.

**arena_test** – Reach this many arenas before testing for arena limit, default 8.

**arena_max** – Maximum number of arenas, default 0.

**n_mmaps** – Current mmapped chunk count, starts at 0.

**n_mmaps_max** – Maximum number of mmapped chunks allowed, default 0x10000.

**max_n_mmaps** – The highest number of concurrent mmapped chunks that have existed so far.

**no_dyn_threshold** – Is dynamic mmap threshold behaviour disabled?

**mmapped_mem** – How much memory is currently mmapped.

**max_mmapped_mem** –The highest amount of mmapped memory that has existed at once.

**sbrk_base** – The 1st address handed out by sbrk

**tcache_bins** – The number of tcache bins to use, default 0x40.

**tcache_max_bytes** – Maximum user data supported by tcache chunks, default 0x408.

**tcache_count** – Maximum number of chunks in each tcachebin, default 7.

**unsorted_limit** – Maximum unsortedbin iterations whilst filling a tcache.

## Tcache

In GLIBC master versions >= 2.26 each thread is allocated its own structure called a tcache, or thread cache. A tcache behaves like an arena, but unlike normal arenas tcaches aren't shared between threads. They are created by allocating space on a heap belonging to their thread's arena and are freed when the thread exits. A tcache's purpose is to relieve thread contention for malloc's resources by giving each thread its own collection of chunks that aren't shared with other threads using the same arena.

A tcache takes the form of a tcache_perthread_struct, shown below in Figure 11, which holds the head of 64 tcachebins preceded by an array of counters which record the number of free chunks in each tcachebin.

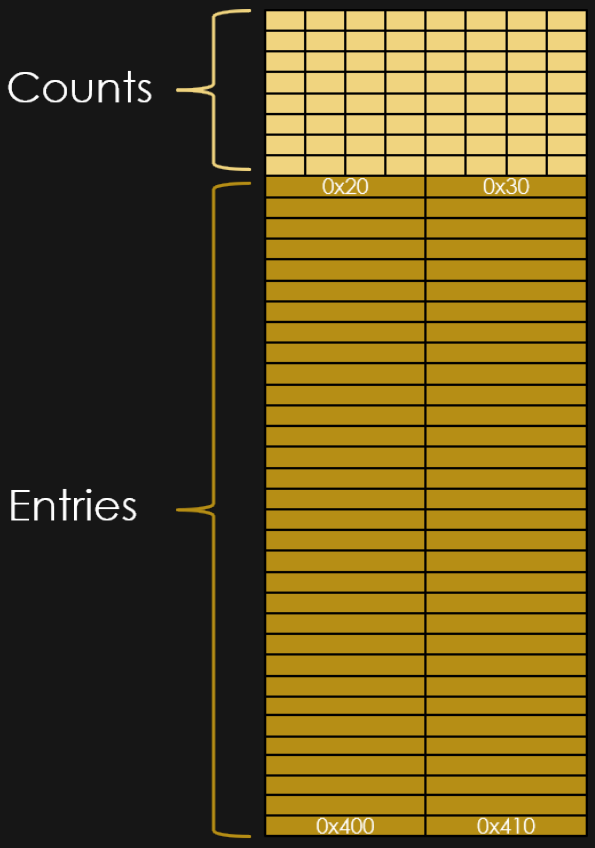



*Figure 11: A tcache_perthread struct*

Note that here "counts" is represented as an array of words, which is only the case in GLIBC versions >= 2.30, prior to this it was an array of chars. Under default conditions a tcache holds chunks with sizes 0x20 – 0x410 inclusive. These tcachebins behave similarly to fastbins, with each acting as the head of a singly linked, non-circular list of free chunks of a specific size. The first entry in the counts array keeps track of the number of free chunks linked into the 0x20 tcachebin, the second entry tracks the 0x30 tcachebin etc.

Under default conditions there is a limit imposed on the number of free chunks a tcachebin can hold, this number is held in the malloc_par struct under the tcache_count field. When a tcachebin's count reaches this limit, free chunks of that bin's size are instead treated as they would be without a tcache present. For example, if the 0x20 tcachebin is full (it holds 7 free chunks) the next 0x20-sized chunk to be freed would be linked into the 0x20 fastbin. Malloc uses a tcache's counts array to determine whether a bin is full.

Allocations from a thread's tcache take priority over its arena, this operation is performed from the __libc_malloc() function and does not enter _int_malloc(). Freed chunks in tcache size range are linked into a thread's tcache unless the target tcachebin is full, in which case the thread's arena is used. Note that tcache entries use pointers to user data rather than chunk metadata.

### Tcache Dumping

In versions of GLIBC compiled with tcache support, chunks in tcache size range are dumped into a tcache when a thread is allocated a chunk from its arena. When a chunk is allocated from the fastbins or smallbins, malloc dumps any remaining free chunks in that bin into their corresponding tcachebin until it is full, as shown in Figure 12 below.

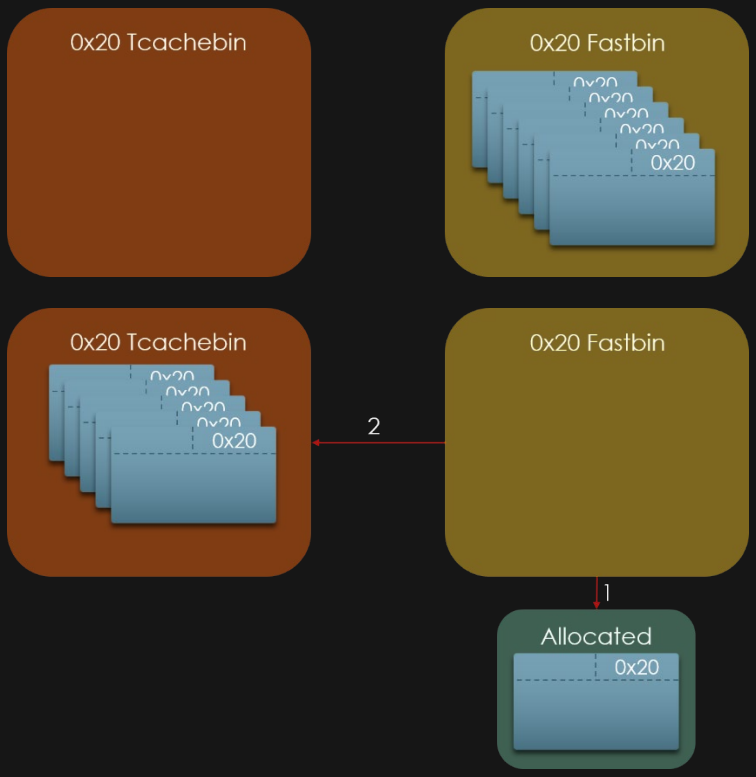



*Figure 12: Tcache dumping from a fastbin*

When an unsortedbin scan occurs, malloc dumps any exact-fitting chunks it finds into their corresponding tcachebin. If the target tcachebin is full and malloc finds another exact-fitting chunk in the unsortedbin, that chunk is allocated. If the unsortedbin scan is completed and one or more chunks were dumped into a tcachebin, a chunk is allocated from that tcachebin. The maximum number of these iterations to perform without allocating a chunk is dictated by the mp_ struct's unsorted_limit field, shown in the Malloc Parameters section.

## Safe Linking

Safe linking is an exploit mitigation introduced in GLIBC version 2.32 by Eyal Itkin. It's designed to force exploit developers to leak a heap address before they can tamper with malloc's singly-linked lists; the fastbins & tcache.

Safe linking is implemented using the PROTECT_PTR and REVEAL_PTR macros, detailed below:

```
#define PROTECT_PTR(pos, ptr) \
((__typeof (ptr)) ((((size_t) pos) >> 12) ^ ((size_t) ptr)))

#define REVEAL_PTR(ptr) PROTECT_PTR (&ptr, ptr)
```

PROTECT_PTR is used to encrypt tcache & fastbin forward pointers using the ASLR entropy of the fd's address. REVEAL_PTR decrypts forward pointers so they can be dereferenced.

Below is an example heap layout that you might see with pwndbg's 'vis' command. Both 0x20-sized chunks shown are linked into the 0x20 tcachebin, which resides just before the yellow chunk. The blue chunk (at a higher address) was freed first, followed by the yellow chunk.

```
          Address
0x55a2e8139290    0x0000000000000000    0x0000000000000021
0x55a2e81392a0    0x000055a7b23d13f9    0x000055a2e8139010
0x55a2e81392b0    0x0000000000000000    0x0000000000000021
0x55a2e81392c0    0x000000055a2e8139    0x000055a2e8139010
0x55a2e81392d0    0x0000000000000000
```

*Figure 13: Safe linking example*

Before safe linking, we'd expect the blue chunk to have a null fd and the yellow chunk's fd to hold the address of the blue chunk's user data. If you shift the address of the yellow chunk's fd (0x55a2e81392a0) right by 12 bits, then xor the result with the same chunk's fd (0x55a7b23d13f9), you'll get the address of the blue chunk's user data (try it in Python or Excel).

Safe linking also imposes alignment checks on chunks allocated from a tcache or fastbin, making it less likely that corrupting the least-significant byte of an encrypted fd by guessing a byte of ASLR entropy will result in a valid encrypted pointer.

One shortcoming of safe linking is that it leaves pointers at the head of fastbins & tcachebins unprotected. Tcaches are particularly vulnerable since they reside on the heap itself, more so in a multithreaded environment where an attacker may have more control over where a thread cache is allocated.

## Malloc Functions

### malloc()

```
void* malloc (size_t bytes)
```

The GLIBC dynamic memory allocation function: takes a request size in bytes as its only argument and returns a pointer to the uninitialized user data region of an appropriately sized chunk of heap memory. The malloc symbol is an alias to __libc_malloc(), which is in turn a wrapper around the _int_malloc() function where the majority of allocation code resides. See the Malloc Flowchart.

### calloc()

```
void* calloc (size_t n, size_t elem_size)
```

Calloc is a wrapper around _int_malloc() that allocates memory for an array of “n” elements of size “size”. The memory returned by calloc() is initialized to zero, calloc() uses some optimizations to ensure this is done efficiently. calloc() does not allocate from the tcache, it is not clear whether this is intended.

### realloc()

```
void* realloc (void* oldmem, size_t bytes)
```

Provides enough dynamic memory to hold “bytes” bytes of data, “oldmem” is a pointer originally provided by one of the memory allocation functions. This may involve allocating a new chunk, copying the data in the “oldmem” chunk, freeing “oldmem” and returning the newly allocated chunk. realloc() uses some optimizations to ensure this is done efficiently, for example by merging forward with a free chunk to avoid the copy operation. When “bytes” is 0 this is an implicit free() operation.

### free()

```
void free (void* mem)
```

The GLIBC dynamic memory recycling function: takes a pointer to a memory region originally provided by one of the memory allocation functions and recycles it. The free symbol is an alias to __libc_free(), which is in turn a wrapper around the _int_free() function where the majority of dynamic memory recycling code resides. See the Free Flowchart.

### malloc_consolidate()

Used internally by malloc to reduce heap fragmentation by consolidating free fast chunks and moving them into their arena’s unsortedbin. malloc_consolidate() can’t be called directly, but is consistently invoked in any of the following circumstances:

1) malloc() receives a request for a large chunk (0x400 and above) that isn’t serviced from the tcache
2) Before a top chunk gets extended
3) When freeing a total space of 65536 (0x10000) bytes, calculated after consolidation
4) When mallopt() is called (only applies to the main arena)

## Malloc Hooks

GLIBC provides hooks for some of malloc's core functionality*. Typical uses for these hooks include monitoring dynamic memory statistics or implementing a different memory allocator altogether. Because they remain writable for the duration of a program's lifecycle, they are a reliable target for heap exploits attempting to gain code execution. GLIBC provides the following hooks related to malloc:

- __after_morecore_hook
- __free_hook
- __malloc_hook
- __malloc_initialize_hook
- __memalign_hook
- __realloc_hook

Some of these hooks are populated with initialisation values which are cleared after the first call to a function. For example __malloc_hook is populated with the address of the malloc_hook_ini() function during GLIBC initialisation, which zeroes __malloc_hook and calls ptmalloc_init():

```
static void* malloc_hook_ini (size_t sz, const void* caller) {
	__malloc_hook = NULL;
	ptmalloc_init();
	return __libc_malloc(sz);
}
```

When a hook is zeroed, calls to its parent function go straight through to that function. When a hook is populated, execution is redirected to the address pointed to by the hook when the parent function is called. For example, the first lines of __libc_malloc() are as follows:

```
void* (*hook) (size_t, const void*) = atomic_forced_read(__malloc_hook);
if(__builtin_expect(hook != NULL, 0))
	return(*hook)(bytes, RETURN_ADDRESS(0));
```

**Note that as of GLIBC version >= 2.34 malloc hook redirection is disabled by default and must be explicitly enabled by the application programmer.*

## mmapped chunks

When a chunk size larger than or equal to mp_.mmap_threshold is requested, it is allocated by mmap() rather than from an arena. These mmapped chunks are slightly different to regular chunks:

They always have a set IS_MMAPPED flag, which is used to indicate to free() how they should be dealt with. When freeing an mmapped chunk it is simply unmapped by munmap(), leaving a hole in the VA space where it used to be.

An mmapped chunk's size field is used to tell munmap() how much memory to unmap. Tampering with an mmapped chunk's size field before freeing it can lead to unmapping more or less memory than the chunk holds.

mmapped chunks also have a prev_size field, which is used when a program requests chunk alignment below page size, e.g. via GLIBC's memalign() function. When an mmapped chunk is freed, its prev_size field determines how much memory preceding the chunk is unmapped. For example, in the figure below, freeing this mmapped chunk would unmap 0x3f0 bytes before the header and 0xc10 bytes in front of it, totalling 0x1000 bytes. Note the IS_MMAPPED flag (2).

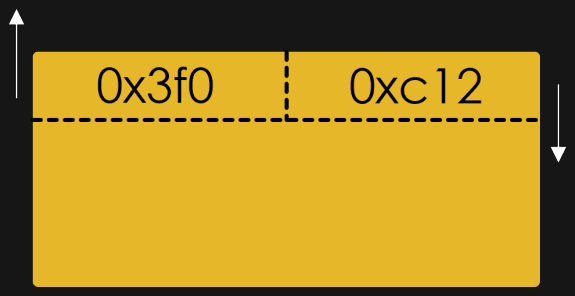



*Figure 14: mmapped chunk metadata*

The only checks performed on mmapped chunks during free() are for page-alignment, since mmap() only allocates in page-sized increments (0x1000 bytes on x64 architecture). The block must start at a page-aligned address and the user data must start at a power of 2 address. The block size must also be a multiple of pages.

mmapped chunks get 1 quadword less user data (on x64 architecture) than their regular chunk counterparts. This is because they are not guaranteed a succeeding chunk's prev_size field to use as user data. Remember to take this into account when requesting mmapped chunks of specific size, a quick way to request a chunk of specific size is to subtract 24 from the desired chunk size.

Freeing an mmapped chunk that is larger than the mmap threshold increases the threshold to the size of the freed chunk. prev_size fields are not factored into this calculation.

The kernel is responsible for where mmap()ed memory gets mapped in the process VA space, so kernel version determines how this happens, but most modern versions take a top-down approach to mapping. This means that when all VA space holes are filled, or you've requested an mmapped chunk larger than any of those holes, your chunk will be mapped just before the library/data at the lowest address in the library ASLR zone. We take advantage of this in the Marauder's mmap lecture to unmap part of the libc binary by tampering with the size field of an mmapped chunk that was mapped right before it.

## Multithreading

Some malloc behaviour is only observed in a multithreaded environment. Calls to malloc() & free() from the main thread act as they would in a single-threaded program, but when a different thread calls malloc() for the 1st time, a new arena may be created. This depends on the mp_ struct's arena_test & arena_max fields. See the Malloc Parameters section for more info on the mp_ struct.

When a thread other than the main threads makes its 1st call to malloc(), the mp_.arena_max field determines whether a new arena will be created. If there are less arenas than arena_max then a new arena will be created for that thread.

If arena_max is zero (its default value) then mp_.arena_test is used to determine a limit on the number of concurrent arenas. arena_test defaults to 8 and when more arenas than this have been created malloc multiplies the number of available CPU cores by a fixed value (8 on x64 architecture) and the result becomes the maximum number of arenas. This limit is kept in a static variable named narenas_limit, populated by the arena_get2() function.

New arenas are created by the _int_new_arena() function, which requests a new heap with new_heap() function, then creates an arena at the start of that heap after its heap_info struct. Non-main arena heaps are mapped at a specific alignment and begin with a heap_info struct. When freeing a chunk with a set NON_MAIN_ARENA flag, the heap's heap_info struct is located by masking off the low-order bytes of the chunk's address. The heap_info struct's ar_ptr field holds the address of the arena administrating that heap.

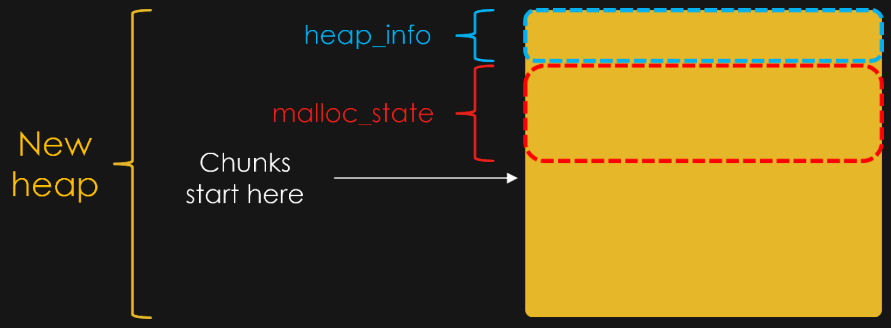



*Figure 15: A new heap & arena*

Non-main heaps aren't extended by sbrk() like the main heap, instead they're mapped as a large space of HEAP_MAX_SIZE with the MAP_NORESERVE flag. Only the amount of space needed to service the 1st request plus mp_.top_pad (or a minimum amount of HEAP_MIN_SIZE) is mapped as read/write. The rest of the heap memory is mapped without read/write permissions, this way it may not take up any physical memory.

When a non-main heap must be extended, the required space is simply given read/write permissions by the mprotect() function, and heaps may be shrunk in the same way. Sometimes a heap may be deleted entirely if it contains neither an arena nor allocated chunks. When the process arena limit is reached (either via mp_.arena_max or arena_test) arenas are reused. Arenas that have no threads attached (e.g. due to their thread exiting) are given priority. These so-called "free" arenas are added to a singly-linked, non-circular list that begins at the free_list symbol.

If no free arenas exist, the thread calling malloc() will be attached to an occupied arena which is found by the reused_arena() function. Arenas are reused in *roughly* the same order they were created, but if an arena is locked at the moment another thread tries to attach, it will be ignored and the search will continue to the next arena. Each new search begins where the last one left off, recorded in the static next_to_use variable.

## Mitigations

List of exploit mitigations introduced into GLIBC malloc.

Note that GLIBC release branches often backport these mitigations, so you may find them present in libc binaries labelled with an earlier version than you see in this table.

| Commit date | Published in GLIBC version | Author | Description | Diff |
|---|---|---|---|---|
| 19/08/2003 | 2.3.3 | Ulrich Drepper | Ensure chunks don't wrap around memory on free(). | diff |
| 21/08/2004 | 2.3.4 | Ulrich Drepper | Safe unlinking checks. | diff |
| 09/09/2004 | 2.3.4 | Ulrich Drepper | Check that the chunk being freed is not the top chunk. Check the next chunk on free is not beyond the bounds of the heap. Check that the next chunk has its prev_inuse bit set before free. | diff |
| 19/11/2004 | 2.3.4 | Ulrich Drepper | Check next chunk's size sanity on free(). | diff |
| 20/11/2004 | 2.3.4 | Ulrich Drepper | Check chunk about to be returned from fastbin is the correct size. Check that the chunk about to be returned from the unsorted bin has a sane size. | diff |
| 22/12/2004 | 2.3.4 | Ulrich Drepper | Ensure a chunk is aligned on free(). | diff |
| 13/10/2005 | 2.4 | Ulrich Drepper | Check chunk is at least MINSIZE bytes on free(). | diff |
| 30/04/2007 | 2.6 | Ulrich Drepper | Unsafe unlink checks for largebins. | diff |
| 19/06/2009 | 2.11 | Ulrich Drepper | Check if bck->fd != victim when allocating from a smallbin. Check if fwd->bk != bck before adding a chunk to the unsorted bin whilst remaindering an allocation from a large bin. Check if fwd->bk != bck before adding a chunk to the unsorted bin whilst remaindering an allocation from a binmap search. Check if fwd->bk != bck when freeing a chunk directly into the unsorted bin. | diff |
| 03/04/2010 | 2.12 | Ulrich Drepper | When freeing a chunk directly into a fastbin, check that the chunk at the top of the fastbin is the correct size for that bin. | diff |
| 17/03/2017 | 2.26 | DJ Delorie | Size vs prev_size check in unlink macro. | diff |

| | | | | |
|---|---|---|---|---|
| 30/08/2017 | 2.27 | Florian Weimer | Don't backtrace on abort anymore. | diff |
| 30/11/2017 | 2.27 | Arjun Shankar | Fix integer overflow when allocating from the tcache. | diff |
| 12/01/2018 | 2.27 | Istvan Kurucsai | Fastbin size check in malloc_consolidate. | diff |
| 14/04/2018 | 2.28 | DJ Delorie | Check if bck->fd != victim when removing a chunk from the unsorted bin during unsorted bin iteration. | diff |
| 16/08/2018 | 2.29 | Pochang Chen | Check top chunk size field sanity in use_top. | diff |
| 17/08/2018 | 2.29 | Moritz Eckert | Proper size vs prev_size check before unlink() in backward consolidation via free. Same check in malloc_consolidate(). | diff |
| 17/08/2018 | 2.29 | Istvan Kurucsai | When iterating unsorted bin check: size sanity of next chunk on heap to removed chunk, next chunk on heap prev_size matches size of chunk being removed, check bck->fd != victim and victim->fd != unsorted_chunks (av) for chunk being removed, check prev_inuse is not set on next chunk on heap to chunk being removed. | diff |
| 20/11/2018 | 2.29 | DJ Delorie | Tcache double-free check. | diff |
| 26/11/2018 | 2.29 | Florian Weimer | Validate tc_idx before checking for tcache double-frees. | diff |
| 14/03/2019 | 2.30 | Adam Maris | Check for largebin list corruption when sorting into a largebin. | diff |
| 18/04/2019 | 2.30 | Adhemerval Zanella | Request sizes cannot exceed PTRDIFF_MAX (0x7fffffffffffffff) | diff |
| 29/03/2020 | 2.32 | Eyal Itkin | Tcache & fastbins safe linking introduced. | diff |
| 06/04/2020 | 2.32 | DJ Delorie | Ensure set_max_fast doesn’t store zero. | diff |
| 21/12/2020 | 2.33 | Richard Earnshaw | Support for memory tagging added. | diff |
| 07/07/2021 | 2.34 | Siddhesh Poyarekar | Replace tcache ‘key’ field with randomly generated value. | diff |
| 22/07/2021 | 2.34 | Siddhesh Poyarekar | Malloc hooks disabled by default. | diff |

## Mitigation error messages

GLIBC malloc error messages and their corresponding mitigations. Not an exhaustive list, but the most frequently triggered mitigations with regards to exploit development.

| Error message | Triggered mitigation |
|---|---|
| "double free or corruption (fasttop)" | Fastbins double-free check. |
| "malloc(): memory corruption (fast)" | Size field check during allocation from fastbins. |
| "corrupted double-linked list" | Safe unlinking check in the unlink macro/function. |
| "corrupted size vs. prev_size" | Size vs. prev_size check in the unlink macro/function. |
| "corrupted size vs. prev_size while consolidating" | Size vs. prev_size check (pre-unlink). |
| "corrupted double-linked list (not small)" | Largebins nextsize safe unlinking check in the unlink macro/function. |
| "malloc(): smallbin double linked list corrupted" | Link integrity check during allocation from smallbins. |
| "malloc(): invalid size (unsorted)" | Size field integrity check during allocation from unsortedbin. |
| "malloc(): invalid next size (unsorted)" | Next size check during allocation from unsortedbin. |
| "malloc(): mismatching next->prev_size (unsorted)" | prev_size check during allocation from unsortedbin. |
| "malloc(): unsorted double linked list corrupted" | fd link integrity check during allocation from unsortedbin. |
| "malloc(): invalid next->prev_inuse (unsorted)" | prev_inuse check during allocation from unsortedbin. |
| "malloc(): corrupted unsorted chunks 3" | bk link integrity check during allocation from unsortedbin. |
| "malloc(): largebin double linked list corrupted (nextsize)" | Nextsize link integrity check during allocation from largebins. |
| "malloc(): largebin double linked list corrupted (bk)" | bk link integrity check during allocation from largebins. |
| "malloc(): corrupted unsorted chunks" | Link integrity check during remaindering from largebins. |
| "malloc(): corrupted unsorted chunks 2" | Link integrity check during remaindering from a binmap search. |
| "malloc(): corrupted top size" | Top chunk size integrity check during allocation from top chunk. |

| | |
|---|---|
| "free(): invalid pointer" | Wraparound & alignment check during free. |
| "free(): invalid size" | Set 4$^{th}$ bit check during free. |
| "free(): double free detected in tcache 2" | Tcache double-free check. |
| "free(): invalid next size (fast)" | Nextsize check during free (fast chunks only). |
| "double free or corruption (top)" | Top chunk free check (non-fast sizes). |
| "double free or corruption (out)" | Heap boundary check (non-fast sizes). |
| "double free or corruption (!prev)" | Non-fast double-free check. |
| "free(): invalid next size (normal)" | Nextsize check during free (non-fast sizes). |
| "free(): corrupted unsorted chunks" | Unsortedbin link integrity check during free. |
| "munmap_chunk(): invalid pointer" | Misaligned mmapped chunk during free(). |
| "malloc(): unaligned tcache chunk detected" | Safe linking during allocation from tcache. |
| "malloc(): unaligned fastbin chunk detected" | Safe linking during allocation from fastbins. |

## Malloc Flowchart

Below is a simplified flowchart depicting the code paths malloc() can take. Blocks coloured yellow are only applicable to versions of GLIBC compiled with tcache support, purple blocks contain a series of operations which are depicted in their own flowcharts. The green blocks indicate a successful allocation and return from malloc(). Entry point is the top-left block.

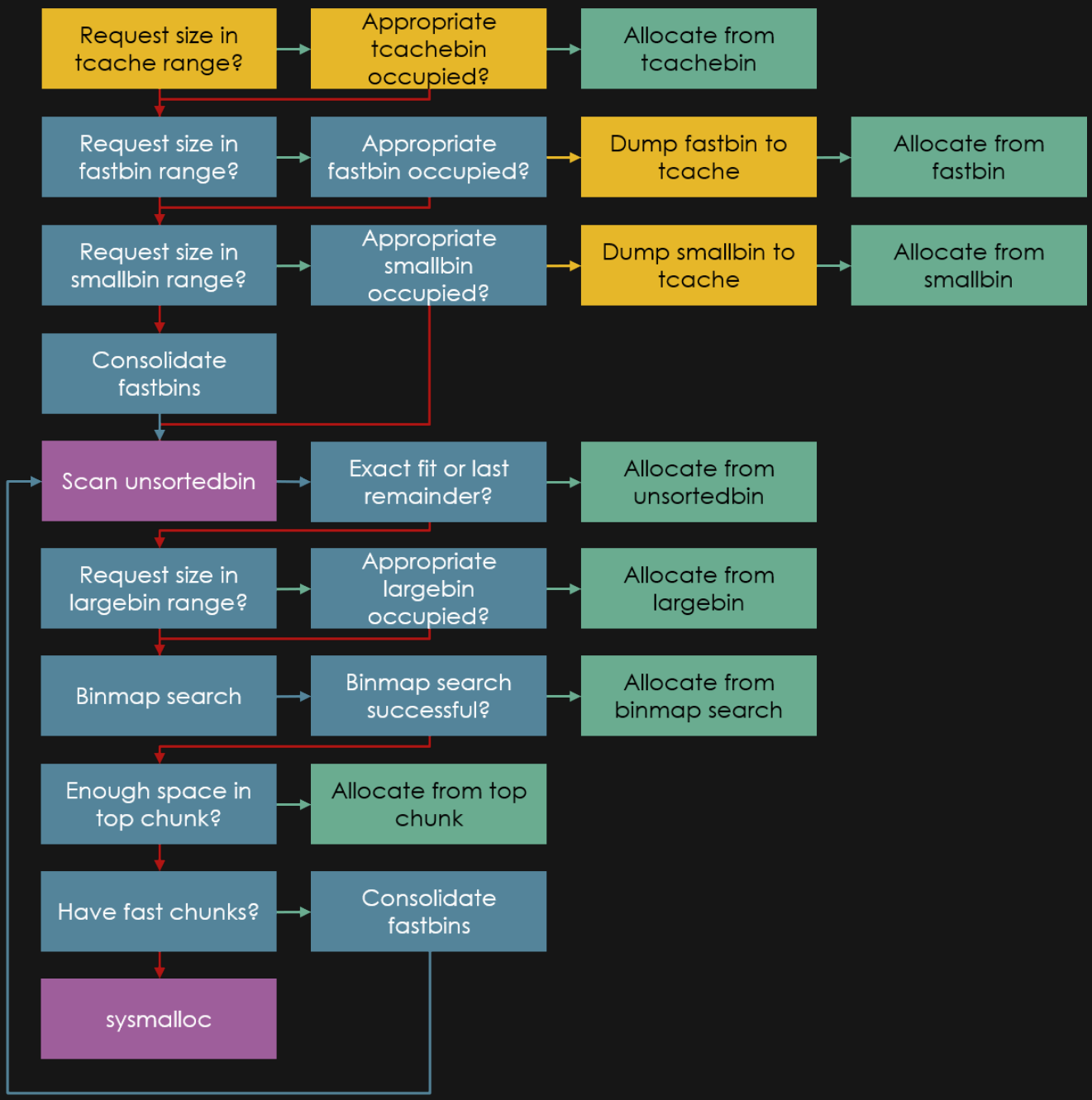



*Figure 16: Malloc flowchart*

## Unsortedbin Flowchart

Below is a flowchart depicting the code paths that an unsortedbin scan can take. Blocks coloured yellow are only applicable to versions of GLIBC compiled with tcache support, green blocks indicate a successful allocation and return from malloc(). If execution reaches the red block malloc() continues on from the unsortedbin scan, as shown in the malloc flowchart. Entry point is the top-left block.

Note that the unsortedbin deals with tcache dumping differently to the fastbins & smallbins; exact-fitting chunks are immediately stored in the tcache. Unless the target tcachebin is full, or the victim chunk is outside the tcache size range, a chunk is allocated from the target tcachebin once the unsortedbin is empty.

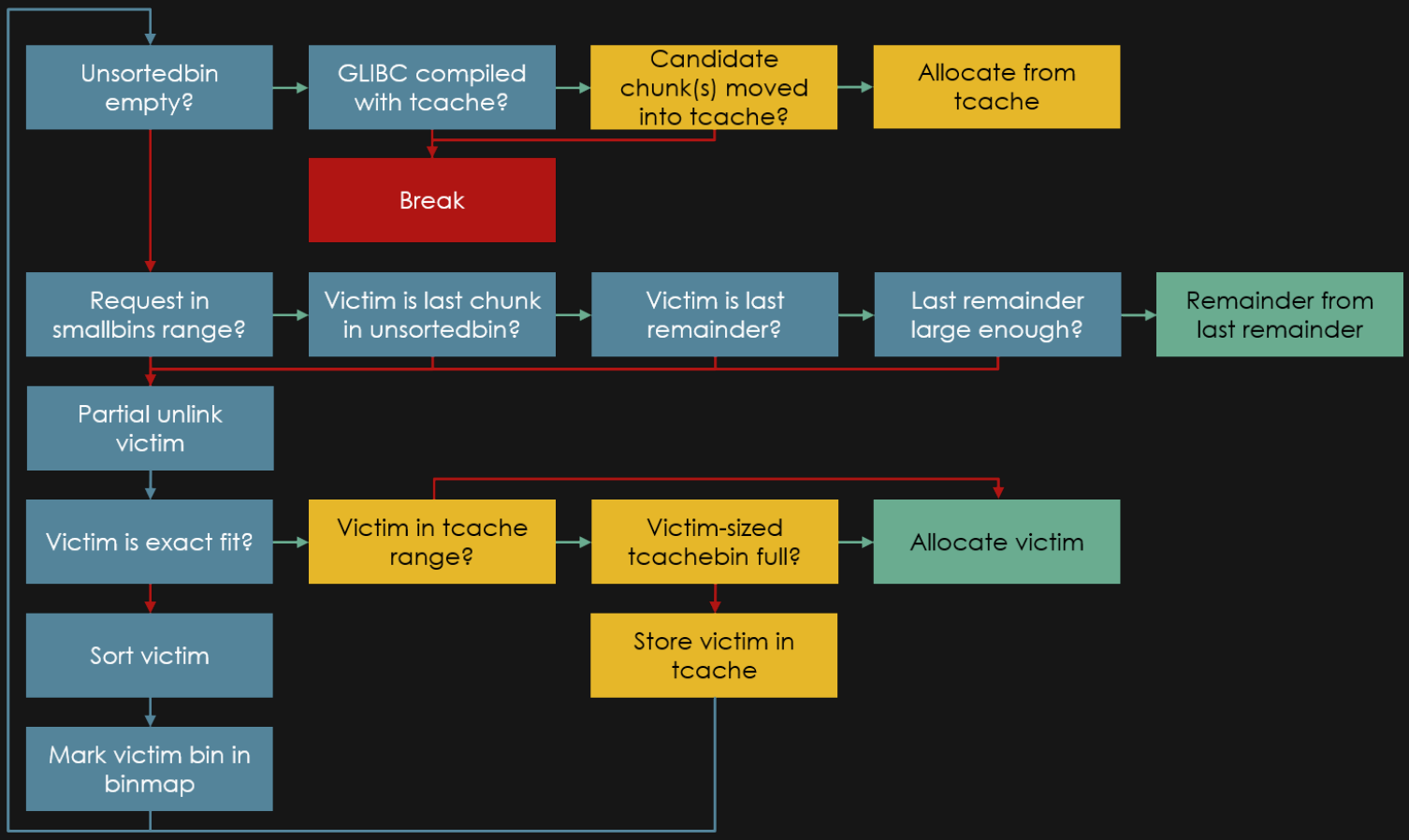



*Figure 17: Unsortedbin flowchart*

## Sysmalloc Flowchart

Below is a flowchart showing the code paths that sysmalloc() can take. Blocks coloured green indicate a successful allocation and return from sysmalloc(). If execution reaches the red block, the ENOMEM error number is set and sysmalloc() returns 0. Entry point is the top-left block.

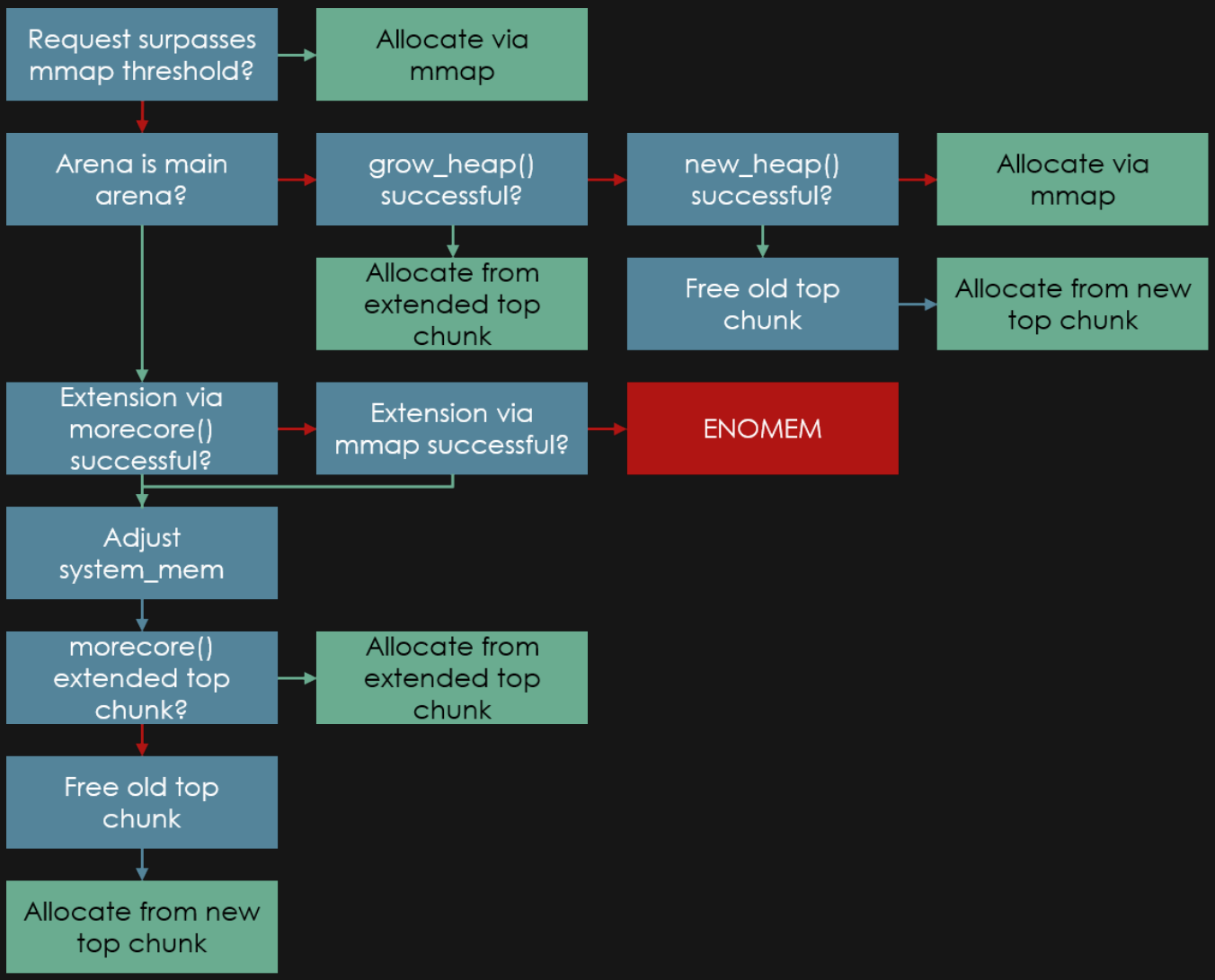



*Figure 18: Sysmalloc flowchart*

## Free Flowchart

Below is a flowchart showing the code paths that free() can take. Blocks coloured green indicate a return path, yellow blocks are only applicable to versions of GLIBC compiled with tcache support.

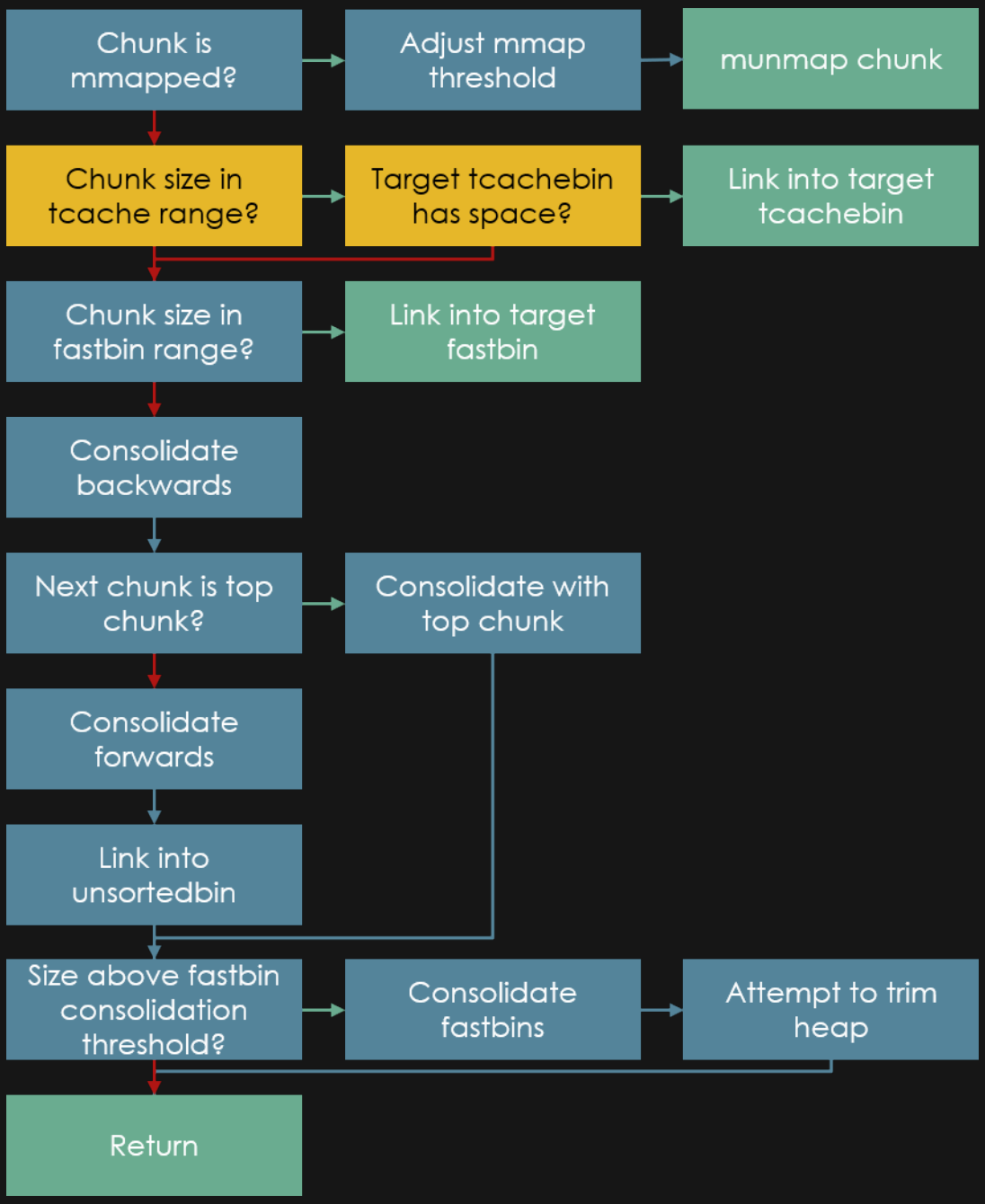



*Figure 19: Free flowchart*

# Exploitation Techniques

This section serves as an extension to the HeapLAB training course, it is not meant as an exhaustive explanation of each heap exploitation technique.

## House of Force

### Overview

Overwrite a top chunk size field with a large value, then request enough memory to bridge the gap between the top chunk and target data. Allocations made in this way can wrap around the VA space, allowing this technique to target memory at a lower address than the heap.

### Detail

In GLIBC versions < 2.29, top chunk size fields are not subject to any integrity checks during allocations. If a top chunk size field is overwritten using e.g. an overflow and replaced with a large value, subsequent allocations from that top chunk can overlap in-use memory. Very large allocations from a corrupted top chunk can wrap around the VA space in GLIBC versions < 2.30.

For example, a top chunk starts at address 0x405000 and target data residing at address 0x404000 in the program's data section must be overwritten. Overwrite the top chunk size field using a bug, replacing it with the value 0xffffffffffffffff1. Next, calculate the number of bytes needed to move the top chunk to an address just before the target. The total is 0xffffffffffffffff - 0x405000 bytes to reach the end of the VA space, then 0x404000 - 0x20 more bytes to stop just short of the target address.

After this request has been serviced from the top chunk, the next chunk to be serviced from it will overlap the target data.

### Further use

If the target resides on the same heap as the corrupt top chunk, leaking a heap address is not required, the allocation can wrap around the VA space back onto the same heap to an address relative to the top chunk.

The malloc hook is a viable target for this technique because passing arbitrarily large requests to malloc() is a prerequisite of the House of Force. Overwriting the malloc hook with the address of system(), then passing the address of a "/bin/sh" string to malloc masquerading as the request size becomes the equivalent of system("/bin/sh").

### Limitations

GLIBC version 2.29 introduced a top chunk size field sanity check, which ensures that the top chunk size does not exceed its arena's system_mem value.

GLIBC version 2.30 introduced a maximum allocation size check, which limits the size of the gap the House of Force can bridge.

## Fastbin Dup

### Overview

Leverage a double-free bug to coerce malloc into returning the same chunk twice, without freeing it in between. This technique is typically capitalised upon by corrupting fastbin metadata to link a fake chunk into a fastbin. This fake chunk can be allocated, then program functionality could be used to read from or write to an arbitrary memory location.

### Detail

The fastbin double-free check only ensures that a chunk being freed into a fastbin is not already the first chunk in that bin, if a different chunk of the same size is freed between the double-free then the check passes.

For example, request chunks A & B, both of which are the same size and qualify for the fastbins when freed, then free chunk A. If chunk A is freed again immediately, the fastbin double-free check will fail because chunk A is already the first chunk in that fastbin. Instead, free chunk B, then free chunk A again. This way chunk B is the first chunk in that fastbin when chunk A is freed for the second time. Now request three chunks of the same size as A & B, malloc will return chunk A, then chunk B, then chunk A again.

This may yield an opportunity to read from or write to a chunk that is allocated for another purpose. Alternatively, it could be used to tamper with fastbin metadata, specifically the forward pointer (fd) of the double-freed chunk. This may allow a fake chunk to be linked into the fastbin which can be allocated, then used to read from or write to an arbitrary location. Fake chunks allocated in this way must pass a size field check which ensures their size field value matches the chunk size of the fastbin they are being allocated from.

Watch out for incompatible flags in fake size fields, a set NON_MAIN_ARENA flag with a clear CHUNK_IS_MMAPPED flag can cause a segfault as malloc attempts to locate a non-existent arena.

### Further use

The malloc hook is a good target for this technique, the 3 most-significant bytes of the _IO_wide_data_0 vtable pointer can be used in conjunction with part of the succeeding padding quadword to form a reliable 0x7f size field. This works because allocations are subject neither to alignment checks nor to flag corruption checks.

Fastbin metadata may instead be tampered with using an overflow or write-after-free bug.

### Limitations

The fastbin size field check during allocation limits candidates for fake chunks.

## Unsafe Unlink

### Overview

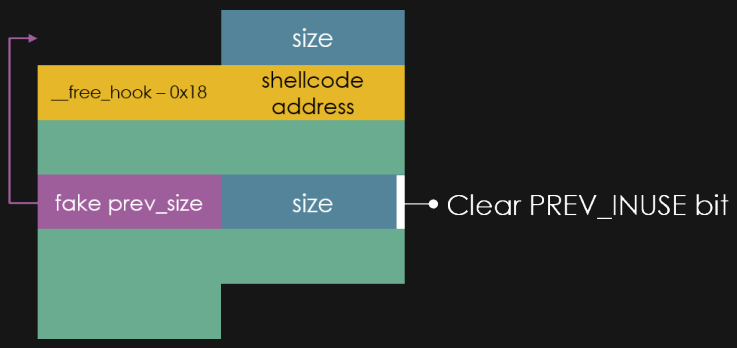
Force the unlink macro to process designer-controlled fd/bk pointers, leading to a reflected write.

### Detail

During chunk consolidation the chunk already linked into a free list is unlinked from that list via the unlink macro. The unlinking process is a reflected write using the chunk's forward (fd) and backward (bk) pointers; the victim bk is copied over the bk of the chunk pointed to by the victim fd and the victim fd is written over the fd of the chunk pointed to by the victim bk. If a chunk with designer-controlled fd & bk pointers is unlinked, this write can be manipulated.

One way to achieve this is via an overflow into a chunk's size field, which is used to clear its prev_inuse bit. When the chunk with the clear prev_inuse bit is freed, malloc will attempt to consolidate it backwards. A designer-supplied prev_size field can aim this consolidation attempt at an allocated chunk where counterfeit fd & bk pointers reside.

For example, request chunks A & B, chunk A overflows into chunk B's size field and chunk B is outside fastbin size range. Prepare counterfeit fd & bk pointers within chunk A, the fd points at the free hook – 0x18 and the bk points to shellcode prepared elsewhere. Prepare a prev_size field for chunk B that would cause a backward consolidation attempt to operate on the counterfeit fd & bk. Leverage the overflow to clear chunk B's prev_inuse bit.




When chunk B is freed the clear prev_inuse bit in its size field causes malloc to read chunk B's prev_size field and unlink the chunk that many bytes behind it. When the unlink macro operates on the counterfeit fd & bk pointers, it writes the address of the shellcode to the free hook and the address of the free hook – 0x18 into the 3rd quadword of the shellcode. The shellcode can use a jump instruction to skip the bytes corrupted by the fd.

Triggering a call to free() executes the shellcode.

### Further use

It is possible to use a prev_size field of 0 and craft the counterfeit fd & bk pointers within chunk B in the above example. The same technique can be applied to forward consolidation but requires stricter heap control.

### Limitations

This technique can only be leveraged against GLIBC versions <= 2.3.3, safe unlinking was introduced in GLIBC version 2.3.4 in 2004 and GLIBC versions that old are not common. This technique was originally leveraged against platforms without NX/DEP and is described as such here. In 2003 AMD introduced hardware NX support to their consumer desktop processors, followed by Intel in 2004, systems without this protection are not common.

## Safe Unlink

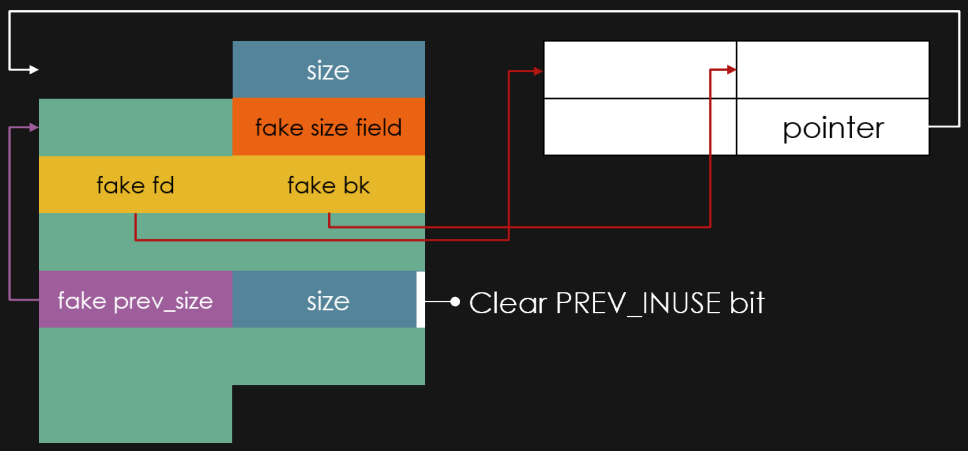
### Overview

The modern equivalent of the Unsafe Unlink technique. Force the unlink macro to process designer-controlled fd/bk pointers, leading to a reflected write. The safe unlinking checks are satisfied by aiming the reflected write at a pointer to an in-use chunk. Program functionality may then be used to overwrite this pointer again, which may in turn be used to read from or write to an arbitrary address.

### Detail

The Safe Unlink technique is similar to the Unsafe Unlink, but accounts for safe unlinking checks introduced in GLIBC version 2.3.4. The safe unlinking checks ensure that a chunk is part of a doubly linked list before unlinking it. The checks pass if the bk of the chunk pointed to by the victim chunk's fd points back to the victim chunk, and the fd of the chunk pointed to by the victim's bk also points back to the victim chunk.

Forge a fake chunk starting at the first quadword of a legitimate chunk's user data, point its fd & bk 0x18 and 0x10 bytes respectively before a user data pointer to the chunk in which they reside. Craft a prev_size field for the succeeding chunk that is 0x10 bytes less than the actual size of the previous chunk. Leverage an overflow bug to clear the succeeding chunk's prev_inuse bit, when this chunk is freed malloc will attempt to consolidate it backwards with the fake chunk.




The bk of the chunk pointed to by the fake chunk's fd points back to the fake chunk, and the fd of the chunk pointed to by the fake chunk's bk also points back to the fake chunk, satisfying the safe unlinking checks. The result of the unlinking process is that the pointer to the fake chunk (a pointer to a legitimate chunk's user data) is overwritten with the address of itself minus 0x18.

If this pointer is used to write data, it may be used to overwrite itself a second time with the address of sensitive data, then be used to tamper with that data.

### Further use

By forging a very large prev_size field the consolidation attempt may wrap around the VA space and operate on a fake chunk within the freed chunk.

### Limitations

A size vs prev_size check introduced in GLIBC version 2.26 requires the fake chunk's size field to pass a simple check; the value at the fake chunk + size field must equal the size field, setting the fake size field to 8 will always pass this check. A 2$^{nd}$ size vs prev_size check introduced in GLIBC version 2.29 requires the fake chunk's size field to match the forged prev_size field.

## Unsortedbin Attack

### Overview

The unsortedbin attack yields a primitive that seems benign but is used to great effect when coupled with other heap exploitation techniques; it writes the address of an arena's unsortedbin to an arbitrary memory location.

### Detail

When a chunk is allocated or sorted from an unsortedbin, it is subject to what malloc refers to as a "partial unlink". This is the process of removing a chunk from the tail end of a doubly linked, circular list, and one part of this process involves writing the address of the unsortedbin over the fd pointer of the chunk pointed to by the victim chunk's bk.

Tampering with the bk pointer of a chunk linked into an unsortedbin via an overflow or write-after-free bug, then allocating the chunk from the unsortedbin results in the address of the unsortedbin being written to the designer-supplied address + 0x10 bytes.

### Further use

Novel uses of the unsortedbin attack include leaking libc by, for example, writing the main arena's unsortedbin address into an output buffer. It can be used to disable the libio vtable integrity check in GLIBC versions 2.24 – 2.26 by targeting the _dl_open_hook symbol. It can also be used to corrupt the global_max_fast variable to leverage a House of Prime primitive. The House of Orange targets the _IO_list_all symbol with an unsortedbin attack as part of a file stream exploitation attempt.

### Limitations

Various unsortedbin integrity checks were introduced in GLIBC version 2.29, one of which ensures the chunk at victim.bk->fd is the victim chunk, mitigating the unsortedbin attack.

## House of Orange

### Overview

The House of Orange leverages an overflow into the main arena's top chunk to drop a shell, it makes use of file stream exploitation to gain code execution and involves the novel application of an unsortedbin attack.

### Detail

The House of Orange can be broken down into 3 phases: top chunk extension, the unsortedbin attack and file stream exploitation. The 1st phase takes advantage of how the main arena deals with top chunk extension. When the main arena's top chunk is exhausted, more memory is requested from the kernel via the brk() syscall, if this memory is not contiguous to the top chunk it is marked as the top chunk and the old top chunk is freed. The House of Orange takes advantage of this by leveraging an overflow into the top chunk to write a small, page-aligned value over the top chunk's size field, then making a request too large to be serviced by the shrunken top chunk. This has the effect of generating a free chunk linked into the unsortedbin in the path of the overflow.

The 2nd phase involves leveraging the overflow bug a 2nd time to overwrite the bk pointer belonging to the old top chunk, which resides in the main arena's unsortedbin. This is used to aim an unsortedbin attack at the _IO_list_all pointer, which will be used by the _IO_flush_all_lockp() function to flush all open file streams.

While overwriting the old top chunk's bk, a fake file stream is crafted on the heap via the overflow such that its _flags field overlaps the old top chunk's prev_size field. The fake file stream's _mode field is set to <= 0, and its _IO_write_ptr field is set to a larger value than its _IO_write_base field. The fake file stream's vtable pointer is populated with the address of a fake vtable, at any location the designer can create one, with the address of system() overlapping the vtable's __overflow entry. The string "/bin/sh\0" is written into the fake file stream's _flags field and the old top chunk's size field (which overlaps the _IO_read_ptr field) is set to 0x61.

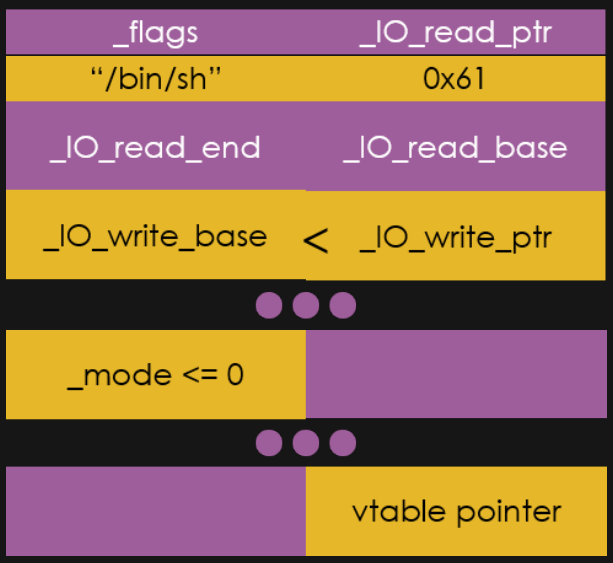



*Figure 20: Fake file stream example*

Figure 17 above shows an example fake file stream, with fields that must be set in yellow and the remaining file stream members in purple.

Next, a chunk with size other than 0x60 is requested. The old top chunk is sorted into the 0x60 smallbin during unsortedbin scanning, triggering the unsortedbin attack which overwrites the _IO_list_all pointer with the address of the unsortedbin. The unsortedbin scan continues and the “chunk” overlapping the _IO_list_all pointer fails a size sanity check, triggering the abort() function.

The abort() function in turn flushes all file streams via _IO_flush_all_lockp(), which dereferences _IO_list_all to find the first file stream. The fake file stream overlapping the main arena isn’t flushed and its _chain pointer overlapping the 0x60 smallbin bk is followed to the fake file stream on the heap. The _mode, _IO_write_base & _IO_write_ptr fields of the fake file stream ensure that the __overflow entry in the fake file stream’s vtable is called, with the address of the fake file stream as the first argument.

This results in a call to system(“/bin/sh”).

### Further use

Phase 1 can be skipped if the designer can free chunks and leverage an unsortedbin attack by other means.

In the case that only 0x60-sized chunks can be requested in phase 3, setting the old top chunk size to 0x69 can also work because this identifies the chunk as non-exact-fitting, but it is still sorted into the 0x60 smallbin. Alternatively, a size field of 0xb1 will also suffice if the 0x60 smallbin is empty, which results in it being followed back into the main arena a second time when treated as a _chain pointer, where the 0xb0 smallbin now overlaps this file stream’s _chain pointer.

### Limitations

When determining if the fake file stream overlapping the main arena should be flushed, the 0xb0 smallbin's fd becomes fp->_wide_data->_IO_write_ptr, and the 0xa0 smallbin's bk becomes fp->_wide_data->_IO_write_base. When these bins are empty this will always result in fp->_wide_data->_IO_write_base being smaller than fp->_wide_data->_IO_write_ptr, which will trigger a call to the file stream's __overflow() member function if fp->_mode > 0. When this happens, fp->_vtable overlaps the bk of the 0xd0 smallbin which results in the __overflow() vtable entry also overlapping the 0xd0 smallbin's bk in an empty main arena. If the __overflow() member of vtable is called, this results in a segfault.

If the fp->_mode field of the file stream in the main arena, which overlaps the 0xc0 smallbin's fd, is greater than zero then the __overflow() vtable entry is called and the program segfaults trying to execute data marked as non-executable. Thus, this technique will only work when the low order dword of the 0xc0 smallbin fd can be interpreted as a negative integer. The bit that determines this state is subject to ASLR, therefore this technique will only succeed approximately 50% of the time, depending on the libc load address.

## House of Spirit

### Overview

Pass an arbitrary pointer to the free() function, linking a fake chunk into a bin which can be allocated later.

### Detail

The House of Spirit is the only technique that does not rely on one of the conventional heap-related bugs, instead it takes advantage of a scenario that allows a designer to corrupt a pointer that is subsequently passed to free().

By passing a pointer to a fake chunk to free(), the fake chunk can be allocated and used to overwrite sensitive data. The fake chunk must have an appropriate size field and in the case of a fast chunk, must have a succeeding size field that satisfies size sanity checks, meaning that a designer must control at least 2 quadwords that straddle the target data.

In the case of a small chunk, there must be 2 trailing size fields to ensure forward consolidation is not attempted, fencepost chunks will work. Because of this a designer must control at least 3 quadwords that straddle the target data.

### Further use

When combined with a heap leak, the House of Spirit can be used to coerce a double-free which can provide a more powerful primitive.

### Limitations

if an arena's contiguity flag is set, fake small chunks must reside at a lower address than their thread's heap, this does not apply to fake fast chunks. Fake chunks must pass an alignment check, which not only ensures that they are 16-byte aligned but mitigates the presence of a set $4^{th}$-least-significant bit in the size field.

Fake chunks must avoid having set NON_MAIN_ARENA and IS_MMAPED bits, in the former case the free() function will search for a non-existent arena and will most likely segfault whilst doing so, and in the latter case the fake chunk is unmapped rather than freed.

# House of Lore

## Overview

Link a fake chunk into the unsortedbin, smallbins or largebins by tampering with inline malloc metadata.

## Detail

Linking a fake chunk into an unsortedbin is equivalent to aiming an unsortedbin attack at a fake chunk by overwriting the unsorted chunk's bk with the address of the fake chunk. The fake chunk must have a bk which points to a writable address. The fake chunk can be allocated directly from the unsortedbin, although its size field must match the request size and differ from the chunk with the corrupt bk.

Linking a fake chunk into a smallbin requires overwriting the bk of a chunk linked into a smallbin with the address of the fake chunk and ensuring the victim->bk->fd == victim check passes by writing the address of the victim small chunk into the fake chunk's fd pointer before the small chunk is allocated. Once the small chunk is allocated, the fake chunk must pass the victim->bk->fd == victim check too, this can be achieved by pointing both its fd & bk at itself. In scenarios where the fake chunk cannot be changed after the victim small chunk is allocated, it's possible to use a 2$^{nd}$ fake chunk, although only 1 quadword is required to hold a fake fd. Pointing this 2$^{nd}$ fake chunk's fd at the primary fake chunk, and the primary fake chunk's bk at the 2$^{nd}$ fake chunk will satisfy the check. The size of the fake chunk is irrelevant as it is not checked.

The easiest way to link a fake chunk into a largebin involves overwriting a skip chunk's fd with the address of a fake chunk and preparing the fake chunk's fd & bk to satisfy the safe unlinking checks. The fake chunk must have the same size field as the skip chunk and the skip chunk must have another same-sized or smaller chunk in the same bin, as malloc will not check the skip chunk's fd for a viable chunk if the skip chunk is the last in the bin. The fake chunk's fd & bk can be prepared to satisfy the safe unlinking checks by pointing them both at the fake chunk.

## Limitations

The amount and precise location of controlled memory required to construct fake small and large chunks for the House of Lore can make it difficult to implement against these bins.

## House of Einherjar

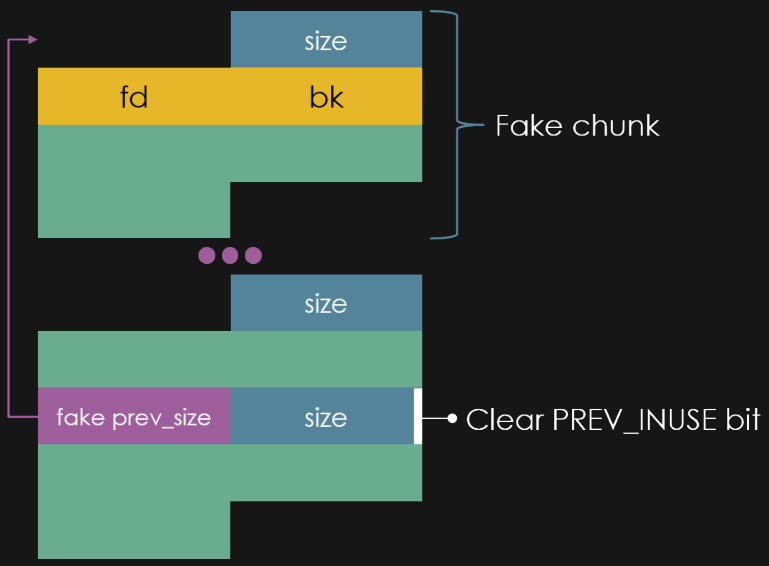
### Overview

Clear a chunk's prev_inuse bit and consolidate it backwards with a fake chunk or an existing free chunk, creating overlapping chunks.

### Detail

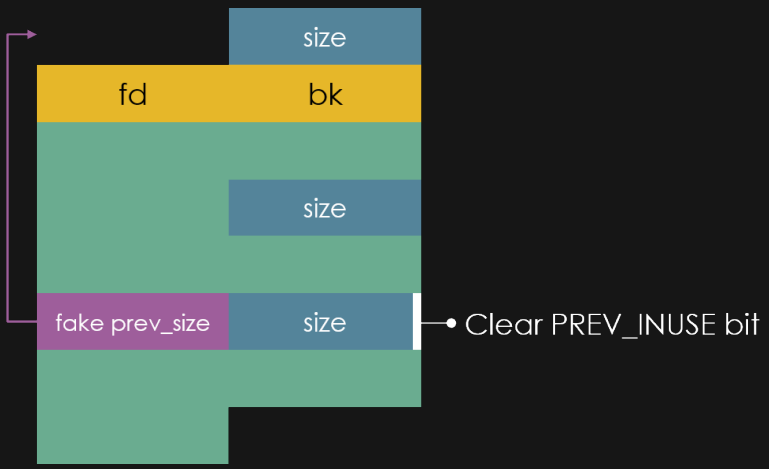
The House of Einherjar was originally presented as a single null-byte overflow technique, but this is not its most realistic application. It assumes an overflow that can clear a victim chunk's prev_inuse bit whilst having control of the victim chunk's prev_size field.

The victim's prev_size field is populated such that when the victim chunk is freed it consolidates backwards with a fake chunk on the heap or elsewhere. In this case, arbitrary allocations can be made from the fake chunk which could be used to read from or write to sensitive data.




### Further use

It is also possible to consolidate with legitimate free chunks on the heap, creating overlapping chunks which can be used to build a stronger primitive.




### Limitations

The size vs prev_size check introduced in GLIBC version 2.26 requires a designer to set an appropriate size field in their fake chunk.

# House of Rabbit

## Overview

Forge a House of Force-like primitive by linking a fake chunk into the largest largebin and setting its size field to an exceptionally large value.

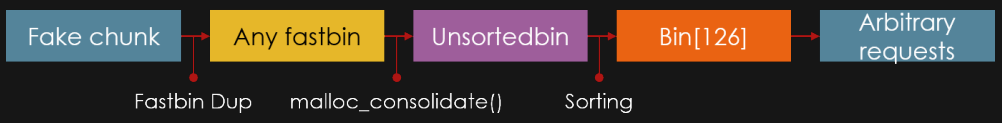
## Detail

Leverage a heap bug to link a fake chunk into a fastbin, the fake chunk consists of 2 size fields: one belongs to the fake chunk itself and the other belongs to the succeeding chunk's size field. However, the succeeding chunk's size field is placed 0x10 bytes before the fake chunk's size field. The fake chunk wraps around the VA space with a size field of 0xfffffffffffffff1 and the succeeding chunk's size is set to 0x11 (a so-called fencepost chunk). An alternative with a smaller memory footprint is to use a fake size field of 0x01, this way the chunk becomes its own forward consolidation guard.


Once the fake chunk is linked into a fastbin it is consolidated into the unsortedbin via malloc_consolidate(). malloc_consolidate() cannot be triggered via malloc() because this results in the fake chunk being sorted which triggers an abort() call when it fails a size sanity check. Instead the fake chunk is sorted by freeing a chunk that exceeds the FASTBIN_CONSOLIDATION_THRESHOLD (0x10000 by default), this can be achieved by freeing a normal chunk that borders the top chunk because _int_free() considers the entire consolidated space to be the size of the freed chunk.

Modify the fake chunk size so that it can be sorted into the largest largebin (bin[126]), malloc only searches this bin for very large requests. To qualify for this bin the fake chunk size must be 0x80001 or larger. Sort the fake chunk into bin[126] by requesting a larger chunk. If the arena's system_mem variable is less than 0x80000, which it will be under default conditions when this heap has not been extended, it is required to artificially increase system_mem by requesting a large chunk, freeing it and requesting it again.

Now that the fake chunk is linked into the largest largebin, it is safe to return its size to 0xfffffffffffffff1. Note that such a large size may not be appropriate when attempting to overwrite stack variables as the fake chunk size may be larger than av->system_mem after the allocation. This will fail a size sanity check during subsequent allocation from the unsortedbin.




This provides a House of Force-like primitive where a large request can be made from the fake chunk that spans the gap between the fake chunk and target memory.

## Further use

An alternative to using a large fake chunk that wraps around the VA space during the initial link into a fastbin is to use a fake fast chunk with trailing fencepost chunks. This requires more controlled memory but bypasses the size sanity checks in malloc_consolidate() as of GLIBC 2.27.

## Limitations

The size vs prev_size check introduced in GLIBC 2.26 means a designer must manually populate the prev_size field of the fake fencepost chunk.

## Poison Null Byte

### Overview

Leverage a single null-byte overflow to create overlapping chunks without having to provide a fake prev_size field.

### Detail

This technique focuses on a single null-byte overflow in a realistic scenario of incorrect null-termination of a string. In such cases it is unusual to be able to provide a fake prev_size field since the quadword before the succeeding chunk's size field is most like holding a string, which is hard to coerce into a sensible prev_size field without using null bytes.

The overflow must be directed at a free chunk with a size field of 0x110 or larger, when this happens the least-significant byte of the size field is cleared, scrubbing 0x10 or more bytes from the affected chunk's size from malloc's perspective. When the victim chunk is allocated again, the succeeding prev_size field isn't updated, this can be leveraged as follows:

Request 4 chunks, A through D: chunk A is used to overflow into chunk B, which is a 0x110 or larger sized chunk, chunk C is normal sized. Chunk D is used to avoid consolidation with the top chunk which is not vital but can make practicing the technique clearer. Free chunk B into its arena's unsortedbin and trigger the single null-byte overflow into chunk B, clearing the least-significant byte of its size field.

Request 2 chunks that will be allocated in the space left by chunk B, designated chunks B1 & B2, chunk B1 must be normal-sized. Free chunk B1, then chunk C. Chunk C is consolidated backwards with chunk B1, overlapping B2, because its prev_size field has not been updated since chunk B was freed. Request memory from the chunk overlapping the still-allocated chunk B2.

### Further use

In GLIBC versions >= 2.26 the size vs prev_size check in the unlink macro/function must be satisfied when chunk B is unlinked during remaindering.

### Limitations

In GLIBC versions >= 2.29 the size vs prev_size check before the unlink function fails when chunk C is freed because chunk B1's size field is incorrect and can't be tampered with.

## House of Corrosion

### Overview

The House of Corrosion leverages a small write-after-free to drop a shell against position independent binaries that don't leak any addresses.

### Detail

The House of Corrosion makes use of a House of Prime primitive in conjunction with a write-after-free bug to drop a shell via file stream exploitation. The write-after-free bug in the example is of varied length, meaning that a designer may overwrite one or more bytes of free metadata as they choose. This primitive can also be coerced by lining up the same write-after-free with different fields of free metadata, or by writing to different fields of the freed object. The technique is leveraged against GLIBC version 2.27, but a different approach using the same tools is effective against GLIBC version 2.29.

Using the write-after-free bug to execute an unsortedbin attack against the global_max_fast variable results in a House of Prime primitive in which freeing large chunks writes their address to an arbitrary location at an offset from the heap's arena's fastbins. In the case the heap belongs to the main arena, this primitive can be used to tamper with the libc writable segment. The unsortedbin attack is made possible by overwriting the 2 least-significant bytes of an unsorted chunk's bk pointer with the write-after-free bug, doing so involves guessing 4 bits of libc load address entropy.

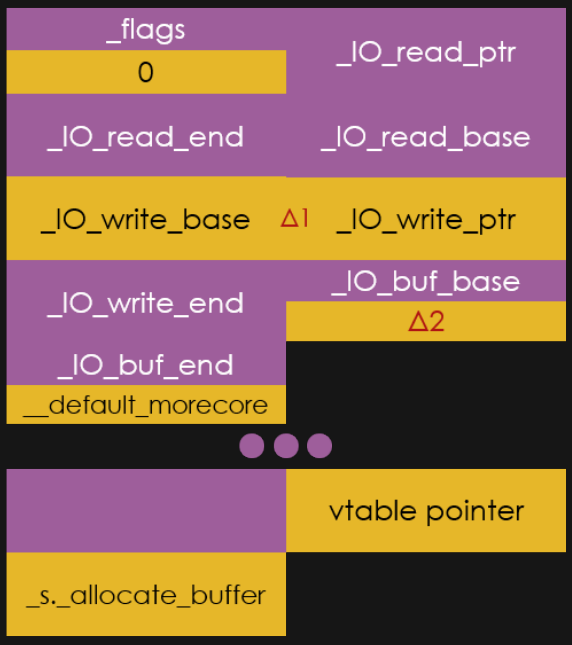
The House of Prime primitive is improved upon by combining it with the write-after-free bug to tamper with memory in a more controlled manner. Once a large chunk has been freed after tampering with the global_max_fast variable, the value at the target address is written into the first quadword of the free chunk's user data as a fastbin fd pointer. Using the write-after-free to tamper with this value then requesting the same sized chunk will write the tampered value back into its original location in memory. This allows a designer to modify the least-significant bytes of an address in the libc writable segment or replace a value entirely.

One further adjustment can be made to provide a primitive which can transplant values between writable memory locations. Changing the size of a chunk between freeing and allocating it allows a value to be read onto the heap from one address, then written to a second address after being changed. This requires a designer to emulate a double-free bug using their write-after-free, which is achieved by requesting 2 nearby chunks, freeing them, then modifying the least-significant byte of the chunk's fd that points to the other chunk to point back to itself instead. When this victim chunk is allocated, a pointer to it remains in the fastbin slot overlapping the transplant destination. Now the victim chunk size can be changed via a write-after-free aligned with its size field, then it is freed again to copy the transplant source data into its fd on the heap. At this point the designer can modify the data with the write-after-free. Lastly, the victim chunk size is changed again, and the chunk allocated, writing the target data to its destination fastbin slot.

To ensure the above primitive is successful a designer must write "safety" values to the heap where the fastbin next size checks are applied against the victim chunk. This can be done by requesting large chunks to write the top chunk size field into the appropriate location beforehand.

These primitives are combined into a file stream exploit by tampering with various stderr fields then triggering a failed assert. The libio vtable integrity check is bypassed by modifying the stderr vtable pointer such that a function that uses a function pointer is called when the assert fails. The location

of this function pointer is overwritten with the address of a call rax gadget, at a point when the rax register contains a designer-controlled offset into the libc DSO.




Δ1 _IO_write_ptr - _IO_write_base > _IO_buf_end - _IO_buf_base

Δ2 _IO_buf_end - _IO_buf_base == call rax gadget

### Further use

The House of Corrosion can be leveraged against GLIBC version 2.29 by disabling the libio vtable integrity check entirely. This requires using the House of Corrosion primitives to tamper with the libc linkmap in such a way that the check interprets the libc DSO as being in a different linker namespace.

### Limitations

The House of Corrosion uses gadgets in the libc DSO, therefore a designer must develop their exploit on a GLIBC build-specific basis.

# Tcache Dup

## Overview

Leverage a double-free bug to coerce malloc into returning the same chunk twice, without freeing it in between. This technique is typically capitalised upon by corrupting tcache metadata to link a fake chunk into a tcachebin. This fake chunk can be allocated, then program functionality could be used to read from or write to an arbitrary memory location.

## Detail

The Tcache Dup technique operates in a similar manner to the Fastbin Dup, the primary difference being that in GLIBC versions < 2.29 there is no tcache double-free mitigation. Simply free the same chunk twice without allocating it to link it into the same tcachebin twice. The Tcache Dup is a very powerful primitive because there is no chunk size integrity check on allocations from a tcachebin, making it very easy to overlap a fake tcache chunk with memory at an arbitrary address.

## Further use

In GLIBC version 2.29 a tcache double-free check was introduced: when a chunk is linked into a tcachebin, the address of that thread's tcache is written into the slot usually reserved for a free chunk's bk pointer, which is relabelled as a "key" field. When chunks are freed, their key field is checked and if it matches the address of the tcache then the appropriate tcachebin is searched for the freed chunk. If the chunk is found to be already in the tcache then abort() is called.

This check can be bypassed by filling the target tcachebin to free a victim chunk into the same sized fastbin, emptying the tcachebin then freeing the victim chunk a 2$^{nd}$ time. Next, the victim chunk is allocated from the tcachebin at which point a designer can tamper with its fastbin fd pointer. When the victim chunk is allocated from its fastbin, the remaining chunks in the same fastbin are dumped into the tcache, including the fake chunk, tcache dumping does not include a double-free check. Note that the fake chunk's fd must be null for this to succeed.

Since the tcache itself resides on the heap it can be subject to corruption after a heap leak.

## Marauder's mmap

My wife's unofficial title for the technique used by Qualys to exploit the qmail server. You may also have seen it referred to as the House of Muney.

### Overview

Tampering with mmapped chunks' size fields, then freeing them can lead to unmapping memory that wasn't supposed to get unmapped. Remapping this same memory as chunks that you control can sometimes be used to hijack the flow of execution. This concept has been around for a while but has recently seen use against production software.

### Detail

Mmapped chunks are subject to none of the usual malloc exploit mitigations we're used to dealing with, they must only pass an alignment check. Increasing an mmapped chunk's size field (e.g. via an overflow), then freeing it allows you to unmap the memory succeeding that chunk in page-sized increments.

The modern Linux kernel mmaps memory in a predictable manner, which can be taken advantage of to map chunks before data in the library ASLR zone, e.g. the libc DSO. Tampering with the size field of such a chunk, then freeing it can lead to the 1$^{st}$ few pages of libc (or any library/data) getting unmapped. We take advantage of this in the 'marauders mmap' lecture to remap libc's .gnu.hash and .dynsym sections, just like Qualys did in their exploit, by requesting chunks larger than the mmap threshold.

Remapping these sections allows us to redirect the flow of execution when a symbol from the affected library is resolved by the program. Specifically, we must repair a bloom filter, hash bucket and hash value in the library's .gnu.hash section and restore the target symbol's entry in the symbol table (the .dynsym section) along with a modified st_value field.

The biggest advantages of this technique are that we only have to deal with a single mitigation, the mmapped chunk alignment check, and that it requires no infoleak to work, yet is very reliable. If you want to learn the specifics and find links to the Qualys blog that I learned this technique from, check out the 'marauders mmap' lecture

# Appendix A: Quick Reference

## Pwndbg/GDB

### Arenas

Show information on **all arenas**:
```
pwndbg> arenas
```

Show information on **a specific arena** (defaults to main_arena):
```
pwndbg> arena [arena address]
```

### Bins

Show information on **all bins** within an arena (defaults to main_arena and current thread):
```
pwndbg> bins [arena address] [tcache address]
```

Show an arena's **fastbins** contents (defaults to main_arena):
```
pwndbg> fastbins [arena address]
```

Show an arena's **smallbins** contents (defaults to main_arena):
```
pwndbg> smallbins [arena address]
```

Show an arena's **largebins** contents (defaults to main_arena):
```
pwndbg> largebins [arena address]
```

Show an arena's **unsortedbin** contents (defaults to main_arena):
```
pwndbg> unsortedbin [arena address]
```

Show a thread's **tcachebins** contents (defaults to the current thread):
```
pwndbg> tcachebins [tcache address]
```

### Chunks

Show information about **chunks on a heap** (defaults to the main_arena heap):
```
pwndbg> heap [heap address]
```

Show information about an arena's **top chunk** (defaults to the main_arena):
```
pwndbg> top_chunk [arena address]
```

Show information about **a single chunk**:
```
pwndbg> malloc_chunk [chunk address]
```

**Visualize** a heap (defaults to the main_arena heap):
```
pwndbg> vis_heap_chunks [count] [heap address]
```

### Miscellaneous

Show information on a **thread cache** (defaults to the current thread):
```
pwndbg> tcache [tcache address]
```

Show the **mp_ struct** members:
```
pwndbg> mp
```

Set search path for libthread-db & add it to safe auto-load list:
```
pwndbg> set libthread-db-search-path <path to glibc directory>
pwndbg> add-auto-load-safe-path <path to glibc directory>
```

## Pwntools

Where **`elf`** is an ELF object representing the binary, **`libc`** is an ELF object representing GLIBC and **`io`** is the process object.

## Symbols

To access the “main” symbol of the binary:

```
elf.sym.main
```

To access the “system” symbol of GLIBC:

```
libc.sym.system
```

## Packing

Pack a 64-bit value into a string:

```
p64(libc.sym.main_arena)
```

Pack a 32-bit value into a string:

```
p32(0xdeadbeef)
```

Pack an 8-bit value into a string:

```
p8(0)
```

Unpack an 8-character string into an integer:

```
u64(b“\xef\xbe\xad\xde\xff\x7f\x00\x00”)
```

## Interacting

To interact with the binary manually:

```
io.interactive()
```

## Template scripts

To start a pwntools script and attach GDB to the process:

```
$ ./script.py GDB
```

To start a pwntools script with ASLR disabled (can be combined with GDB):

```
$ ./script.py NOASLR
```

To start a pwntools script and see debugging output:

```
$ ./script.py DEBUG
```

## One-Gadget

Search for & print any one-gadgets in the GLIBC binary that <target program> was linked against:

```
$ one_gadget $(ldd <target program> | grep libc.so | cut -d’ ’ -f3)
```

It can be useful to add a shell function that makes finding a binary’s libc path easier:

```
$ libc() {ldd $1 | grep libc.so | cut -d’ ‘ -f3}
```

## Appendix B: File stream exploitation

Not strictly heap exploitation but included for reference. Below is an example _IO_FILE struct, comments are from the GLIBC source, except the last 3.

```
struct _IO_FILE {
      int _flags;  /* High-order word is _IO_MAGIC; rest is flags. */

      /* The following pointers correspond to the C++ streambuf protocol. */
      char* _IO_read_ptr;       /* Current read pointer */
      char* _IO_read_end;       /* End of get area. */
      char* _IO_read_base;      /* Start of putback+get area. */
      char* _IO_write_base;     /* Start of put area. */
      char* _IO_write_ptr;      /* Current put pointer. */
      char* _IO_write_end;      /* End of put area. */
      char* _IO_buf_base;       /* Start of reserve area. */
      char* _IO_buf_end;        /* End of reserve area. */

      /* The following fields are used to support backing up and undo. */
      char* _IO_save_base;      /* Pointer to start of non-current get area. */
      char* _IO_backup_base;    /* Pointer to 1st valid char of backup area */
      char* _IO_save_end;       /* Pointer to end of non-current get area. */

      struct _IO_marker* _markers;
      struct _IO_FILE* _chain;
      int _fileno;
      int _flags2;
      __off_t _old_offset; /* This used to be _offset but it's too small. */

      /* 1+column number of pbase(); 0 is unknown. */
      unsigned short _cur_column;
      signed char _vtable_offset;
      char _shortbuf[1];
      _IO_lock_t* _lock;
      __off64_t _offset;

      /* Present in GLIBC < 2.33 */
      struct _IO_codecvt* _codecvt;
      struct _IO_wide_data* _wide_data;
      struct _IO_FILE* _freeres_list;
      void* _freeres_buf;

      /* Present in GLIBC >= 2.33 */
      void* __pad1;
      void* __pad2;
      void* __pad3;
      void* __pad4;

      size_t __pad5;
      int _mode;
      char _unused2[20];

      /* vtable pointer appended here when treated as an _IO_FILE_plus struct. */
      const struct _IO_jump_t* vtable;
};
```

## Arbitrary reads with file streams

If you're able to corrupt an active file stream, you may be able to leak information to a file descriptor. A good example of this is the 'optimize' challenge, in which we leak a libc address to the terminal.

When preparing a file stream for reading, clobber the _flags field and ensure the following flags are set/clear:

| Flag | State | Reason |
|---|---|---|
| _IO_NO_WRITES | Clear | This flag indicates the file stream shouldn't be written to. Since we want it to write data to the file descriptor, ensure this flag is cleared. |
| _IO_CURRENTLY_PUTTING | Set | If this flag is clear the file stream's _IO_write_base field will be clobbered. We need to keep that field intact so set this flag. |
| _IO_IS_APPENDING | Set | Avoids an early return if set. An alternative is to ensure _IO_read_end & _IO_write_base match. |
| _IO_USER_LOCK | Set | If this flag isn't set the file stream must be unlocked to write, which means pointing the _lock field at a null quadword. |

You can safely ignore the 0xfbad magic bytes.

Having set the appropriate flags, the next time this file stream is written to it will dump all data between the addresses pointed to by its _IO_write_base & _IO_write_ptr fields to the file descriptor in its _fileno field. Setting _IO_write_base & _ptr accordingly may allow you to leak arbitrary data.

One way to leak a libc address to the terminal is to modify just the least-significant byte of stdout's _IO_write_base field. Where _IO_write_base points depends on whether stdout is buffered, which it is by default. You may be able to get away with clobbering the 2 least-significant bytes, then sifting the output for any libc pointers. See the 'optimize' challenge lecture for one way to deal with an unbuffered stdout scenario.

It's supposedly possible to make arbitrary writes via file streams too, but since I've never tried it I'm not confident writing about it here.