



# Heap Exploitation

## A Brief Introduction

elbee



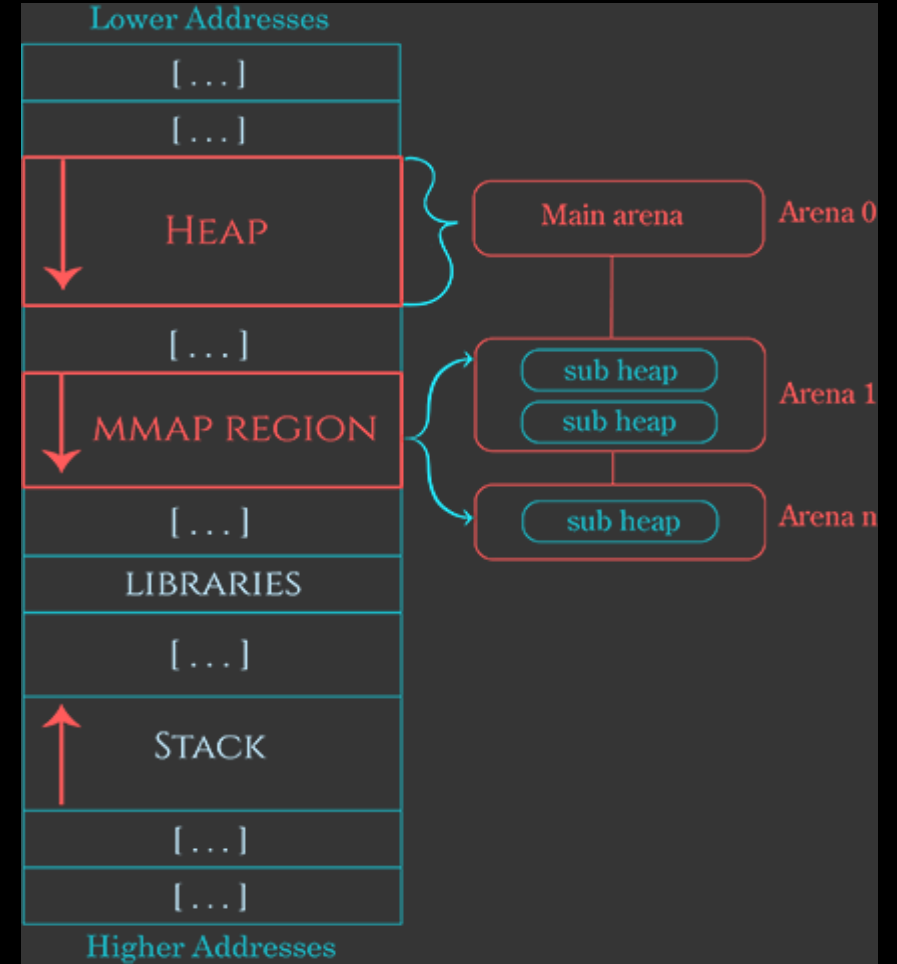
# ● The heap

Glibc is derived from pthreads malloc (ptmalloc)

malloc(size) – Returns a pointer to the newly allocated chunk on the heap.

free(ptr) – Frees a chunk from allocation and returns it to the heap manager for future allocation.

Different threads have different heaps and their own arenas. We're mostly dealing with the main arena. Arenas store heap metadata used by malloc and free to service requests. (e.g. Heads of free lists, maximum serviceable request size, etc) The main arena is located in the libc section.



<https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/>



# ● Chunk anatomy

Chunks are sized to the nearest 8-byte aligned serviceable size. Heap chunks have metadata in them used by malloc and free when servicing future requests. Freeing a chunk does not delete it, it just modifies the chunk to prepare it for future allocation in the event it meets a request.



A = Chunk is not in the main heap & was allocated.  
M = Chunk was mmaped and will be treated directly.  
P = The previous chunk is not free.



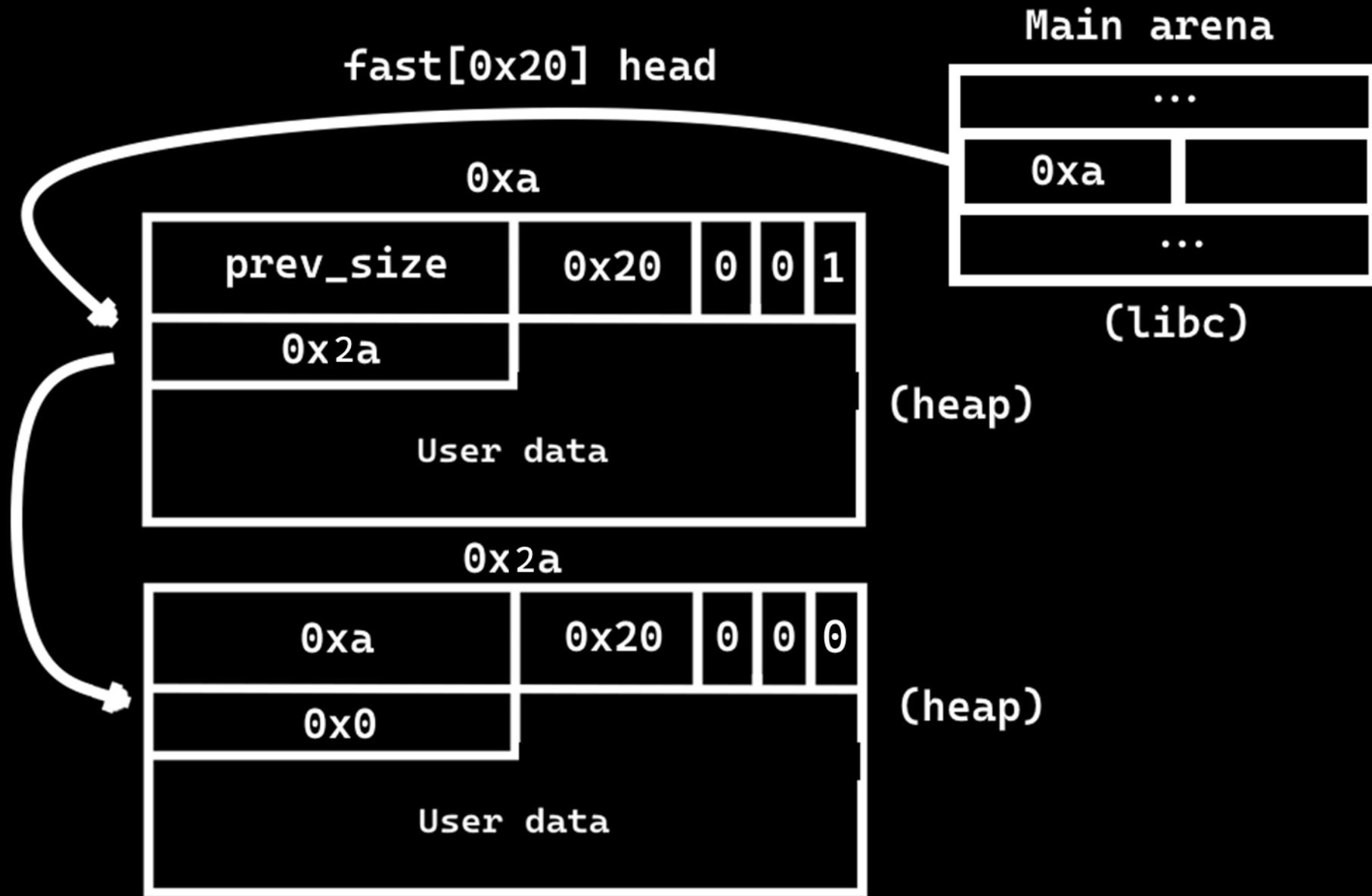
# ● Freed chunk anatomy

Freebins are linked lists of freed chunks. Depending on the chunk, it will be stored in its corresponding list: the tcache, fastbins or unsortedbin (where the chunk is further sorted into the smallbin or largebin by malloc). If a chunk is not a fast or tcache chunk and an adjacent chunk is also free (prev\_inuse) the chunks may be consolidated, this is also true with the top chunk. The head of a bin is stored in the arena and linked members are stored inline in the freed chunk in its first two qwords of user data.



A = Chunk is not in the main heap & was allocated.  
M = Chunk was mmapped and will be treated directly.  
P = The previous chunk is not free.







# Demo

Materials can be found at

[https://faultpoint.com/assets/filebin/malloc\\_demo.c](https://faultpoint.com/assets/filebin/malloc_demo.c)

[https://faultpoint.com/assets/filebin/malloc\\_demo](https://faultpoint.com/assets/filebin/malloc_demo)

<https://faultpoint.com/assets/filebin/template.py>



# ● Fastbins

The fastbins are a singly linked freelist that stores chunks of sizes 0x20 to 0xb0. It is singly linked so only the first qword of user data is utilized as an fd. Malloc has a fast size integrity check when allocating from these bins (more on glibc exploit mitigations later). Chunks that are fast-sized will be put into the corresponding fastbin depending on if the tcache is disabled or the corresponding tcache entry is full.

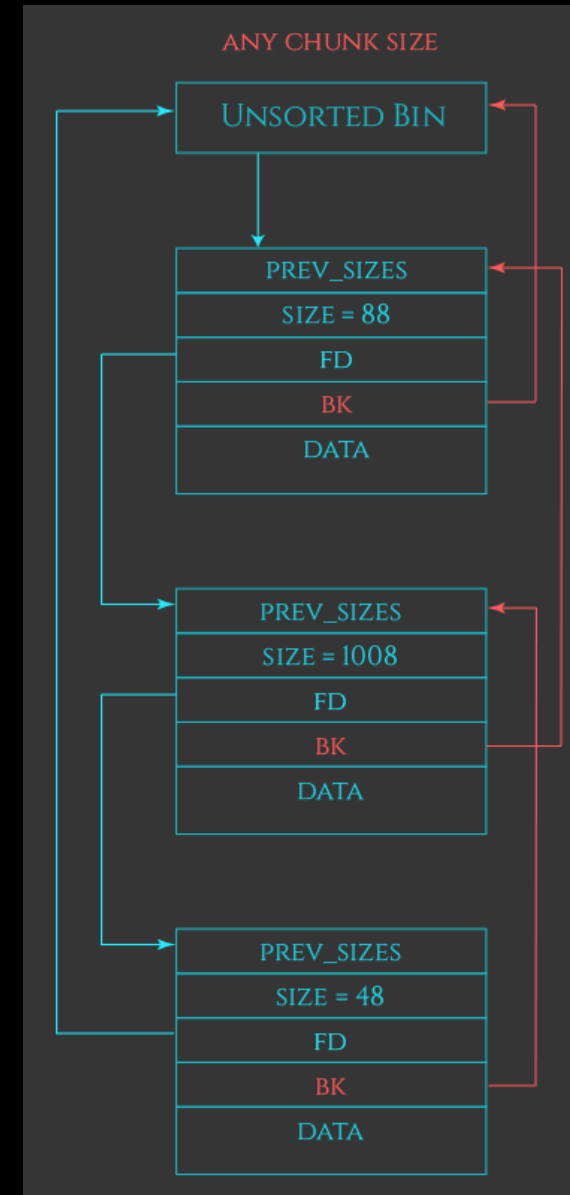
```
0x55555559260 0x0000000000000000 0x0000000000000000 .....
0x55555559270 0x0000000000000000 0x0000000000000000 .....
0x55555559280 0x0000000000000000 0x0000000000000000 .....
0x55555559290 0x0000000000000000 0x0000000000000021 .....!..... <-- fastbins[0x20][5]
0x555555592a0 0x0000000555555559 0x0000000000000000 YUUU..... <-- fastbins[0x20][4]
0x555555592b0 0x0000000000000000 0x0000000000000021 .....!..... <-- fastbins[0x20][3]
0x555555592c0 0x000055500000c7c9 0x0000000000000000 ....PU..... <-- fastbins[0x20][2]
0x555555592d0 0x0000000000000000 0x0000000000000021 .....!..... <-- fastbins[0x20][1]
0x555555592e0 0x000055500000c7e9 0x0000000000000000 ....PU..... <-- fastbins[0x20][0]
0x555555592f0 0x0000000000000000 0x0000000000000021 .....!..... <-- Top chunk
0x55555559300 0x000055500000c789 0x0000000000000000 ....PU.....
0x55555559310 0x0000000000000000 0x0000000000000021 .....!.....
0x55555559320 0x000055500000c7a9 0x0000000000000000 ....PU.....
0x55555559330 0x0000000000000000 0x0000000000000021 .....!.....
0x55555559340 0x000055500000c649 0x0000000000000000 I...PU.....
0x55555559350 0x0000000000000000 0x00000000000020cb1 .....

pwndbg> p/x main_arena.fastbinsY[0]
$3 = 0x55555559330
pwndbg> fastbins
fastbins
0x20: 0x55555559330 -> 0x55555559310 -> 0x555555592f0 -> 0x555555592d0 -> 0x555555592b0 <- ...
pwndbg>
```



# ● Unsortedbin

Non-fast-sized chunks that don't border the top will be placed into the unsortedbin, a doubly-linked circular freelist. Next time malloc is called, the unsortedbin will be searched for a chunk that is serviceable. During iteration it will sort chunks it comes across into the smallbins or largebins, it will only sort up until an allocation is found. Its first two qwords of user data contain an fd and bk respectively. A dummy chunk is also linked into the main\_arena. Since chunks are consistently unlinked from the unsortedbin, unsortedbin metadata can be targeted in unlinking attacks. If no exact sized chunk exists after a search, the last remaining closest fit chunk will be remaindered.





# ● Tcache (>2.26)

- Why the tcache
- What is the tcache
- How does the tcache work
- Security risks

In a lot of scenarios, the presence of the tcache can make exploitation easier!

0x602030	0x0000000000000000	0x0000000000000291	.....	
0x602040	0x0000000000000005	0x0000000000000000	.....	count[0x20]
0x602050	0x0000000000000000	0x0000000000000000	.....	
0x602060	0x0000000000000000	0x0000000000000000	.....	
0x602070	0x0000000000000000	0x0000000000000000	.....	counts
0x602080	0x0000000000000000	0x0000000000000000	.....	
0x602090	0x0000000000000000	0x0000000000000000	.....	
0x6020a0	0x0000000000000000	0x0000000000000000	.....	
0x6020b0	0x0000000000000000	0x0000000000000000	.....	
0x6020c0	0x000000000000602350	0x0000000000000000	P#.....	entries[0x20]
0x6020d0	0x0000000000000000	0x0000000000000000	.....	
0x6020e0	0x0000000000000000	0x0000000000000000	.....	
0x6020f0	0x0000000000000000	0x0000000000000000	.....	
0x602100	0x0000000000000000	0x0000000000000000	.....	
0x602110	0x0000000000000000	0x0000000000000000	.....	
0x602120	0x0000000000000000	0x0000000000000000	.....	
0x602130	0x0000000000000000	0x0000000000000000	.....	
0x602140	0x0000000000000000	0x0000000000000000	.....	
0x602150	0x0000000000000000	0x0000000000000000	.....	
0x602160	0x0000000000000000	0x0000000000000000	.....	
0x602170	0x0000000000000000	0x0000000000000000	.....	entries
0x602180	0x0000000000000000	0x0000000000000000	.....	
0x602190	0x0000000000000000	0x0000000000000000	.....	
0x6021a0	0x0000000000000000	0x0000000000000000	.....	
0x6021b0	0x0000000000000000	0x0000000000000000	.....	
0x6021c0	0x0000000000000000	0x0000000000000000	.....	
0x6021d0	0x0000000000000000	0x0000000000000000	.....	
0x6021e0	0x0000000000000000	0x0000000000000000	.....	
0x6021f0	0x0000000000000000	0x0000000000000000	.....	
0x602200	0x0000000000000000	0x0000000000000000	.....	
0x602210	0x0000000000000000	0x0000000000000000	.....	
0x602220	0x0000000000000000	0x0000000000000000	.....	
0x602230	0x0000000000000000	0x0000000000000000	.....	
0x602240	0x0000000000000000	0x0000000000000000	.....	
0x602250	0x0000000000000000	0x0000000000000000	.....	
0x602260	0x0000000000000000	0x0000000000000000	.....	
0x602270	0x0000000000000000	0x0000000000000000	.....	
0x602280	0x0000000000000000	0x0000000000000000	.....	

# ● Techniques

Some of the first heap exploitation techniques were in the malloc maleficarum. For every libc version, people have been discovering new techniques that utilize corruption on the heap to modify stuff such as metadata or important glibc structures.

<https://github.com/shellphish/how2heap>  
<https://0x434b.dev/overview-of-glibc-heap-exploitation-techniques>

## Exercise

Because of the nature of how ptmalloc stores data, even small, off-by-one vulnerabilities can be leveraged to gain full code execution.

Imagine you have a single null byte overflow into an adjacent chunk, how could you utilize this to achieve an arbitrary write?



# ● Glibc Mitigations

Many of these techniques have been partly or fully mitigated with checks and asserts implemented in different glibc versions. If you encounter a mitigation, the best way to learn what its doing is to read the source.

Other utility functions such as realloc (resize chunks) and calloc (zeros out returned chunks) behave slightly different but all implement int\_malloc the same. Not all mitigations are full proof.

```
0 0x7ffff7c969fc pthread_kill+300
1 0x7ffff7c969fc pthread_kill+300
2 0x7ffff7c969fc pthread_kill+300
3 0x7ffff7c42476 raise+22
4 0x7ffff7c287f3 abort+211
5 0x7ffff7c89676 __libc_message+662
6 0x7ffff7ca0cfc
7 0x7ffff7ca53dc

pwndbg> f 5
#5 0x00007ffff7c89676 in __libc_message (action=action@entry=do_abort, fmt=fmt@entry=0x7ffff7d8bb77 "%s\n") at ../sysdeps/posix/libc_fatal.c: No such file or directory.
155
pwndbg> context stack
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

[ STACK ]
00:0000 rsp 0x7fffffbfa0 -> 0x7ffff7dded20 -- 'malloc(): unaligned tcache chunk detected'
01:0008 -0f8 0x7fffffbfa8 -> 0x29 /* ' */ */
02:0010 -0f0 0x7fffffbfb0 -> 0x7ffff7d8bb79 -- 0x29 /* ' */ */
03:0018 -0e8 0x7fffffbfb8 -> 0x1
04:0020 rbx r12 0x7fffffbfc0 -> 0x27 /* " */ */
05:0028 -0d8 0x7fffffbfc8 -> 0x3e996c997fb29700
06:0030 -0d0 0x7fffffbfd0 -> 0x7ffff7d8bb79 -- 0x27 /* " */ */
07:0038 -0c8 0x7fffffbfd8 -> 0x1

pwndbg> |
```

```
{CODEBROWSER} About Contact Search for a file or
3175 detect a double free. */
3176 e->key = tcache->key;
3177
3178 e->next = PROTECT_PTR (&e->next, tcache->entries[tc_idx]);
3179 tcache->entries[tc_idx] = e;
3180 ++(tcache->counts[tc_idx]);
3181 }
3182
3183 /* Caller must ensure that we know tc_idx is valid and there's
3184 available chunks to remove. */
3185 static __always_inline void *
3186 tcache_get (size_t tc_idx)
3187 {
3188 tcache_entry *e = tcache->entries[tc_idx];
3189 if (__glibc_unlikely (!aligned_OK (e)))
3190 malloc_printerr ("malloc(): unaligned tcache chunk detected");
3191 tcache->entries[tc_idx] = REVEAL_PTR (e->next);
3192 --(tcache->counts[tc_idx]);
3193 e->key = 0;
3194 return (void *) e;
3195 }
```



# Challenge

The earlier provided demo binary as discussed is vulnerable to a read and write after free. The goal of this challenge is to exploit the demo binary using the provided python template to drop a shell. There are 2 ways (and possibly more) you can achieve this.

Hint: If you get stuck, try researching the “safe linking” mitigation introduced in glibc 2.32.

Download challenge binary and template solve  
[https://faultpoint.com/assets/filebin/malloc\\_demo](https://faultpoint.com/assets/filebin/malloc_demo)  
<https://faultpoint.com/assets/filebin/template.py>



# ● Resources

<https://github.com/shellphish/how2heap> (Shellphish how2heap)

<https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/> (Azeria labs)

<https://guyinatuxedo.github.io/25-heap/index.html> (Nightmare)

<https://www.youtube.com/watch?v=6-Et7M7qJJg> (Max Kamper's Introduction to Glibc Heap Exploitation presentation)

<https://www.udemy.com/course/linux-heap-exploitation-part-1/> (HeapLab, not free)

There are also some heap challenges you can practice with on TCTF.

## Questions?

