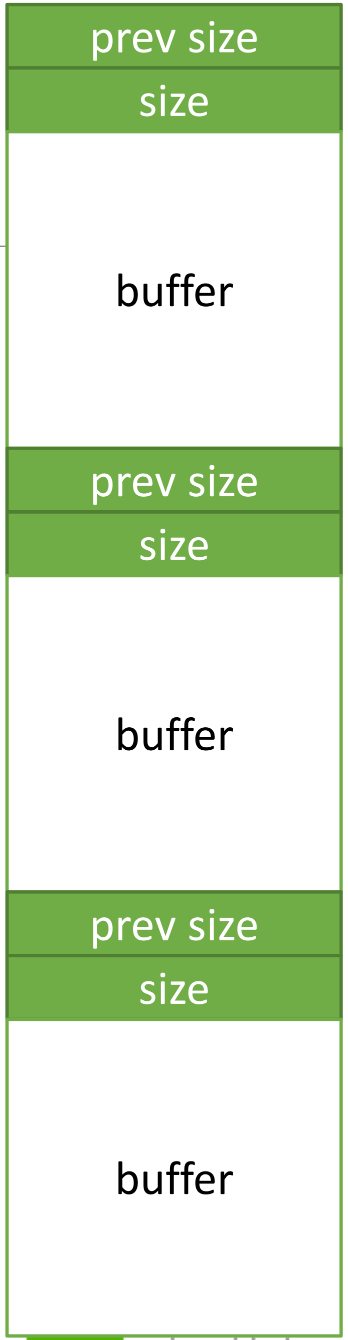
Buffer issues - Heap

JOÃO PAULO BARRACA

Heap Overflow

- **Heap is used to store dynamically allocated variables**
  - Allocation: malloc, calloc and new (C++), release: free or delete (C++)

- **Call reserves a chunk and returns a pointer to the buffer**
  - buffer: (8 + (n / 8)*8 bytes)
    - If chunk is free data will have:
      - Forward Pointer (4 bytes), pointer to next free chunk
      - Backwards Pointer (4 bytes), pointer to previous free chunk
  - Headers used for housekeeping
    - Previous Chunk Size (previous chunk is free), 4 bytes
    - Chunk Size + flags, 4 bytes
      - Flags
        - 0x01 PREV_INUSE – set when previous chunk is in use
        - 0x02 IS_MMAPPED – set if chunk was obtained with mmap()
        - 0x04 NON_MAIN_ARENA – set if chunk belongs to a thread arena

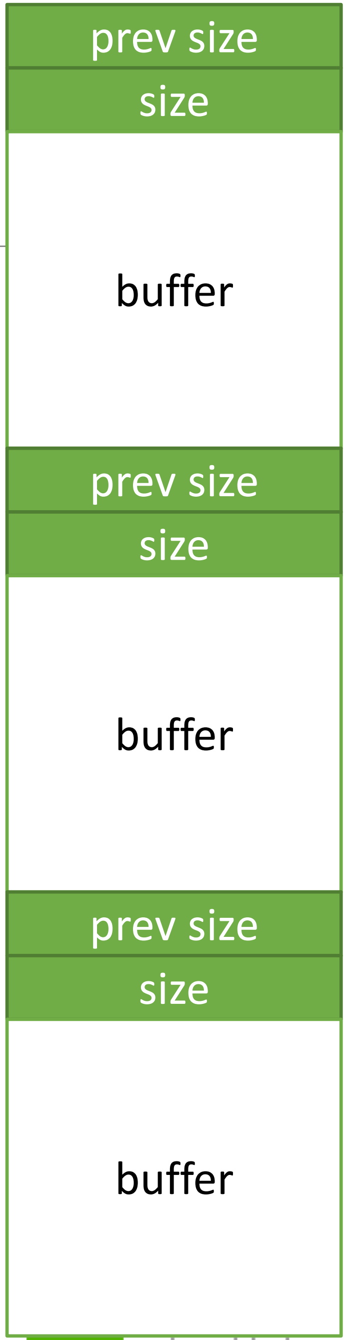


Heap Overflow: overflow.c

- **Overflow/underflow will write/read over control structures and then data**
  - Control structures are implementation specific
  - As well as reuse and actual buffer location

```
int main(int argc, char **argv) {
        char *buf1 = (char *) malloc(BUFSIZE);
        char *buf2 = (char *) malloc(BUFSIZE);
        memset(buf1, 0, BUFSIZE); //Clear data
        memset(buf2, 0, BUFSIZE);

        printf("Buf2: %s\n", buf2); //Should print “Buf2: “
        strcpy(buf1, argv[1]);
        printf("Buf2: %s\n", buf2); //Should print “Buf2: “
}
```



Heap Overflow: dangling.c

- **Dangling references can give access to memory**
  - Both for read and write purposes

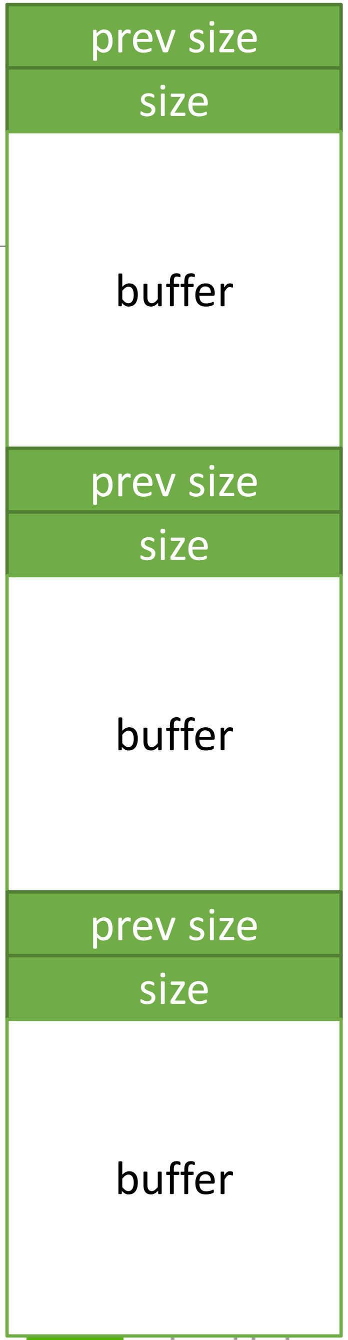
```
char *buf1 = (char *) malloc(BUFSIZE*100); //Allocate buffer
memset(buf1, 'U', BUFSIZE);  //Fill it with 0x55
free(buf1); //Free the memory

char *buf2 = (char *) malloc(BUFSIZE); //Allocate new buffer
memset(buf2, 'A', BUFSIZE);  //Fill it with 0x41

printf("%s\n", buf1); //buf1 was freed
```

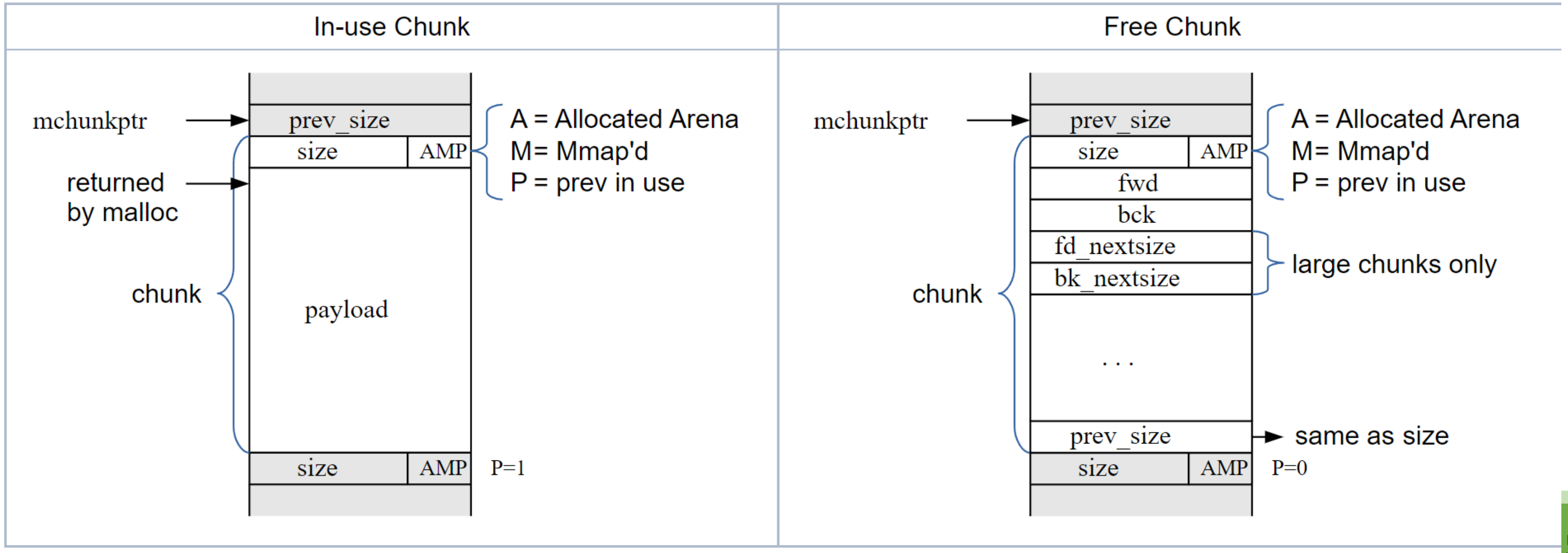
- **Access to buf1 should be denied: it isn't**

- **Access to buf1 should not give access to other ranges: it gives to buf2**



Chunks

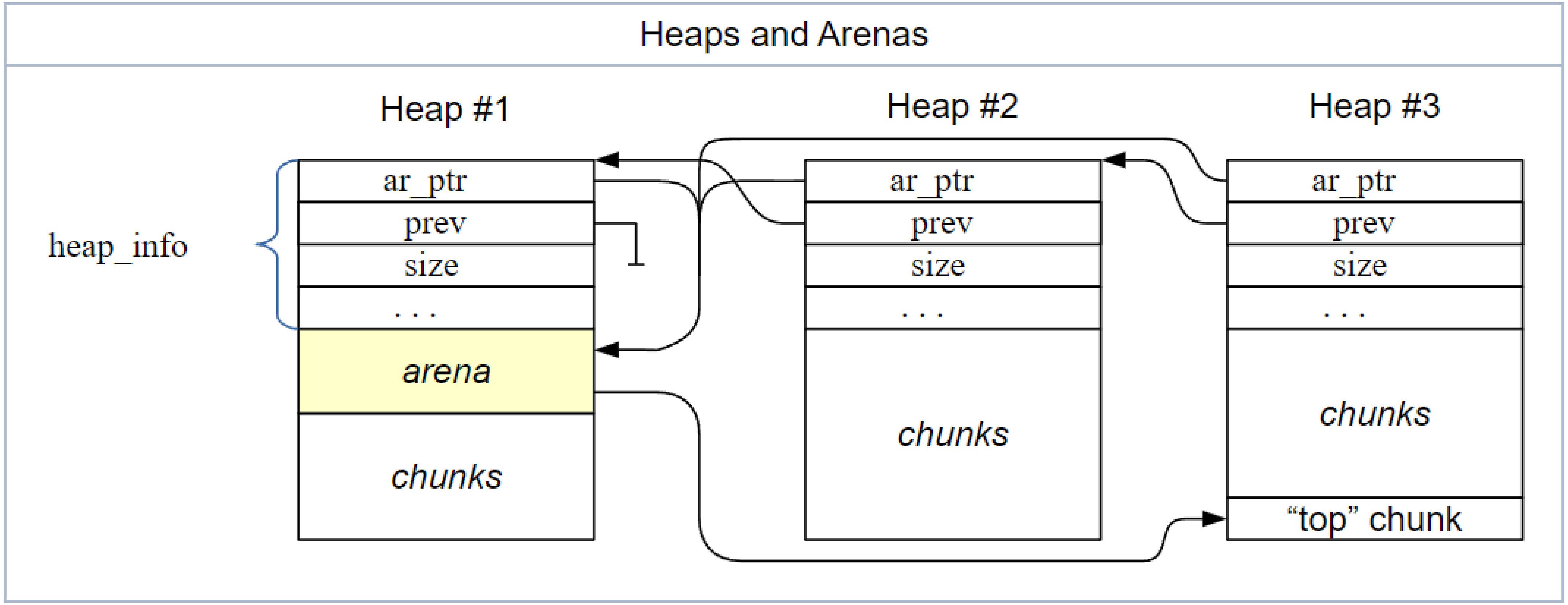
- **Chunks are structures to provide the program with data storage**
  - They are composed by a payload and metadata.
  - The actual content varies if the chunk is free of used
  - Managed by malloc/free



Arenas

- **Malloc allows for more than one region of memory to be active at a time.**
- **Different threads can access different regions of memory without interfering with each other.**
  - Interference may require mutex locks, which is expensive
- **These regions of memory are collectively called "arenas".**
  - Applications start with a “Main area”
  - New threads will use another arenas
    - If too many arenas are created, malloc will reuse existing arenas (max is 8xCPUs)
- **Arenas have heaps where memory is allocated from**
  - Memory is mmaped to the heaps as the program requires more memory

Arenas


Heaps and Arenas
Heap #1
Heap #2
Heap #3
heap_info
ar_ptr
prev
size
. . .
arena
chunks
ar_ptr
prev
size
. . .
chunks
ar_ptr
prev
size
. . .
chunks
“top” chunk

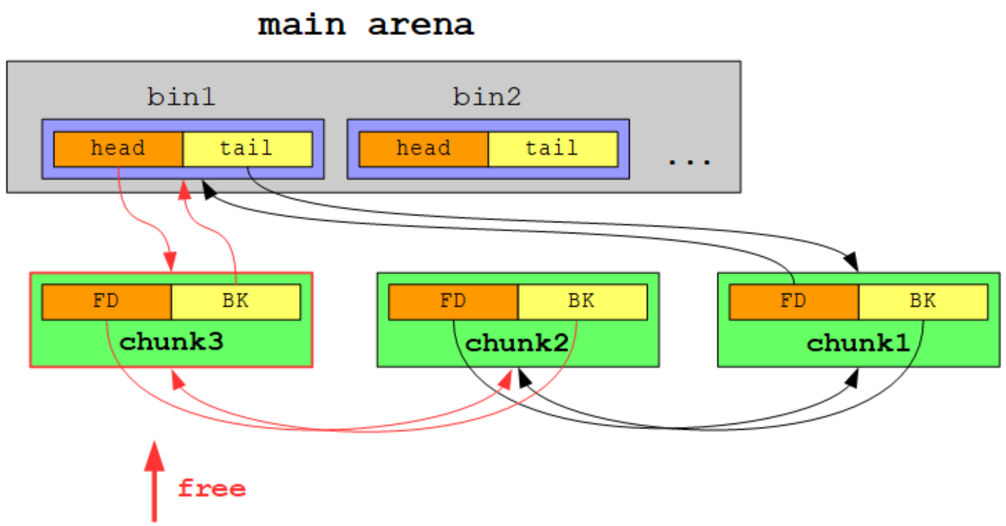
Bins

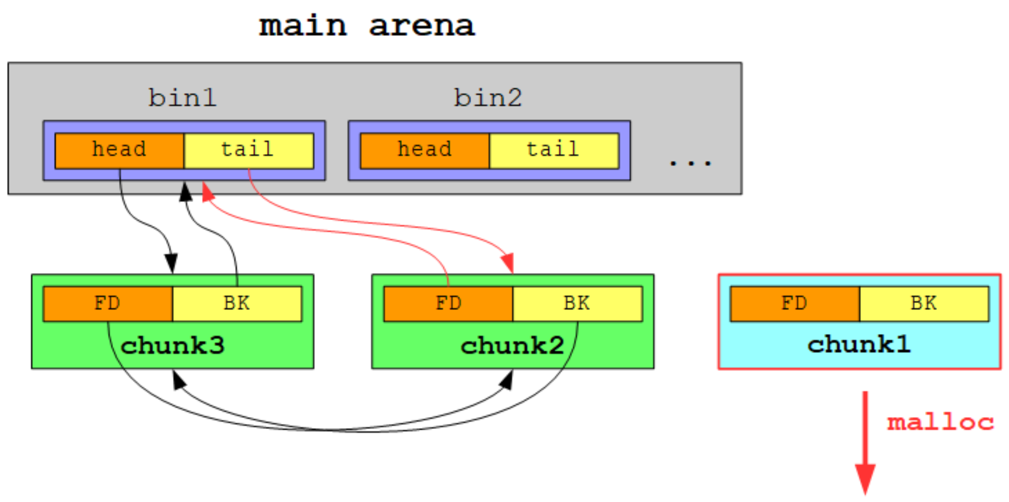
- **Glibc has lists of recently freed chunks**
  - Each list (bin) stores chunks with a specific size
  - Blocks are reused in future allocations if size is compatible
    - Great for performance as the memory is already reserved
    - Horrible for security as dangling pointers will give a view to memory areas
- **Bins are also used to detect double free**
  - We cannot free a chunk that rests at the top of the bin
  - Which is great for security as a double free could corrupt the linked list

Other bins

- **Unsorted: stores chunks rapidly without taking in consideration their size**
  - Malloc will later consolidade these chunks into other bins
- **largebins: stores large chunks from the unsorted bins**
  - Chunks in the unsorted bins are coaleshed into larger chunks and stored here
  - Allocation from largebins requires “finding” the “best suited” chunk for a given allocation
  - Getting a chunk may involve leaving the “remaining” as a new chunk
- **smallbins: stores small chunks**
  - Usually never contiguous as they are the remaining chunks not coaleshed into larger chunks
  - Stored in a ordered manner by fixed size
    - There are 62 small bins for specific sizes

Other bins





https://devel0pment.de/?p=688

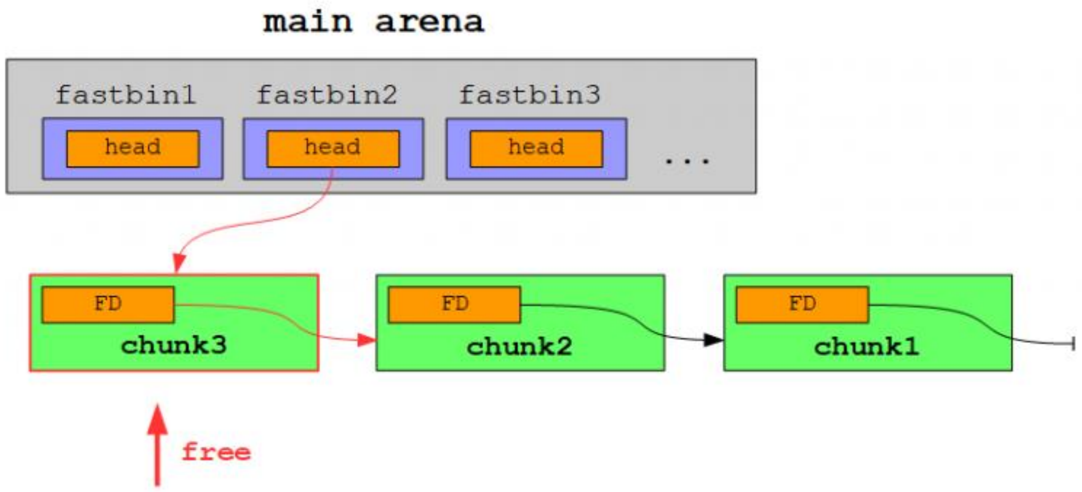
fastbins

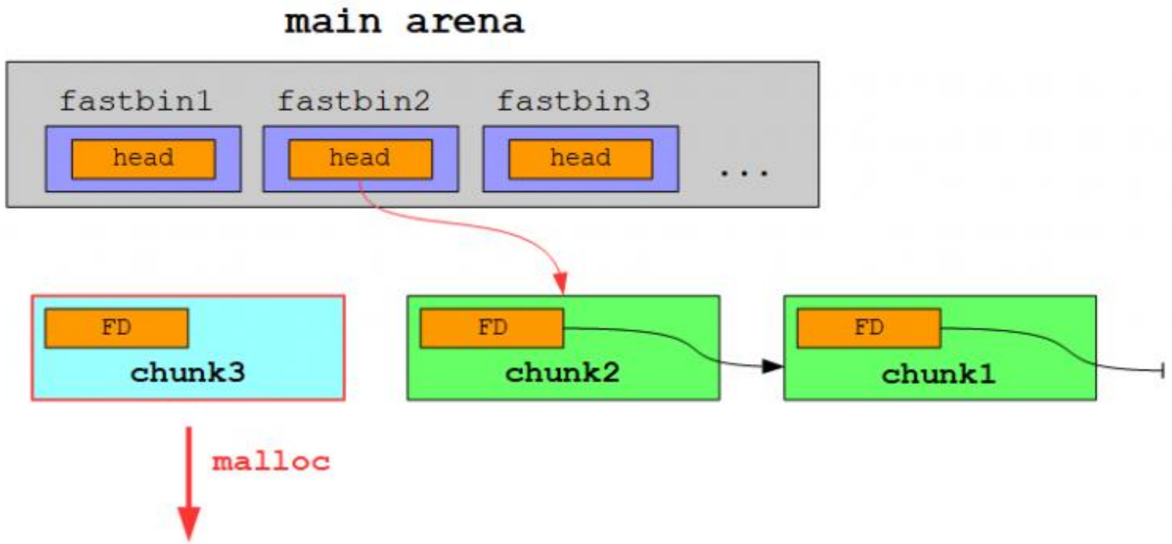
- **A set of single linked lists of free chunks with specific sizes**
  - up to 0xb0 bytes
  - They are consumed from the top, as the logic is minimal.
  - Chunks are first placed here and later consolidated/processed into other bins
  - This is meant for fast access of recent chunks (common objects/chunks)
    - Overlaps with other bins

- **As a linked list, chunks will point to the next free chunk**

- **LIFO Pattern: Last freed will be first allocated**

fastbins





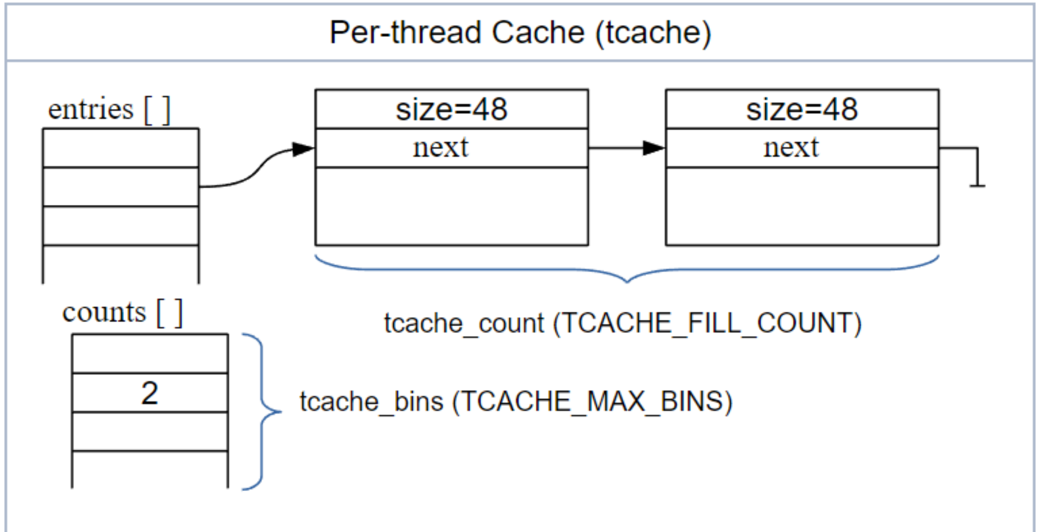
https://devel0pment.de/?p=688

Tcache (Thread Local Cache)

- **In multi-threaded applications, arena contention is expensive, tcache is the optimization**
  - per-thread cache containing a small collection of chunks which can be accessed without needing to lock an arena

- **Singly-linked lists, like fastbins, but with links pointing to the payload (user area) not the chunk header.**

- **Malloc will try to allocate chunks from this**
  - Recent freed chunks, local to the thread
  - Exploits code locality aspects
  - Failure will result in using the normal slow path (lock arena, and search for chunks)

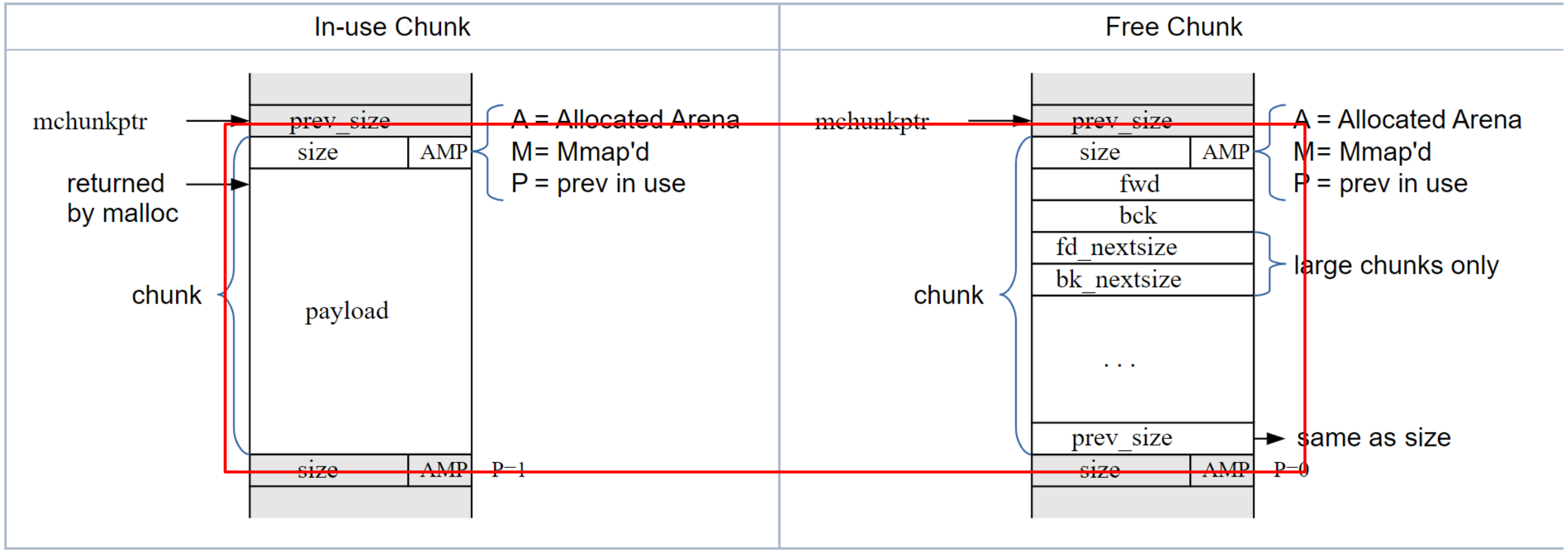


Heap Overflow: fastbin attacks

- **Fast Bin attack explores Bins to get a pointer to an already allocated area**
  - Result is program will have **two pointers to the same memory**
    - Especially useful if memory stores dynamic objects with function, as function pointers can be overwritten
  - The first pointer is legitimate
  - The second is a shadow pointer

- **Attack strategy**
  - Allocate at least three buffers (a, b, c) with the same size
    - To use same bin
  - free(a), then free(b), then free(a) again
    - Double freeing a will ensure that the fast bin will have duplicated entries (a)
    - Bin will have three pointers ready to use: a b a
  - Allocate three buffers again with the same size.
    - Result is a legitimate pointer, another legitimate pointer, and a shadow pointer

fastbin attacks



- **Payload of an used chunk maps to FD and BK pointers of a free chunk**

fastbin attacks

- **Impact: attacker can gain access to memory region**
  - If victim has chunk a with data and leaks
  - Attacker can fill free list and allocate again

```
// Allocating 3 buffers
int *a = calloc(1, 8);
int *b = calloc(1, 8);
int *c = calloc(1, 8);

free(a);
free(b);
free(a); //AGAIN!

//Free list now has: a b a

int *d = calloc(1, 8);
int *e = calloc(1, 8);
int *f = calloc(1, 8);

// d will be equal to f
```

Heap Overflow: overflow.c

- **Exercise: Observe and document the behavior in both programs**
  - dangling.c and overflow.c
  - Use GDB to analyse the addresses
    - x/10gx address
    - heap bins
    - heap chunks

  - What is the impact of writing to a freed pointer?

Countermeasures: ASLR

- **Address Space Layout Randomization (ASLR)**
  - Address are dynamic across process execution
    - Different architectures and configurations apply randomization to different segments
    - Only Stack is randomized, all segments are randomized
  - Not trivial to predict the address to issue a jump or change memory

- **echo $n > /proc/sys/kernel/randomize_va_space**
  - 0 = No randomization
  - 1 = Conservative Randomization: Stack, Heap, Shared Libs
  - 2 = Full Randomization: 1 + memory managed via brk())

Effects of ASLR (WSL1 on Windows 10)

- randomize_va_space =2

```
main: 0x7f80def82189, argc: 0x7fffbfce569c, local: 0x7fffbfce56ac, heap: 0x7fffb8c4b2a0, libc: 0x7f80ded85410
main: 0x7fb811d47189, argc: 0x7fffdbd2928c, local: 0x7fffdbd2929c, heap: 0x7fffd47952a0, libc: 0x7fb811b55410
main: 0x7f95178f0189, argc: 0x7fffee962b7c, local: 0x7fffee962b8c, heap: 0x7fffe67082a0, libc: 0x7f95176f5410
```

- randomize_va_space =1

```
main: 0x7f1672f77189, argc: 0x7fffe5835f0c, local: 0x7fffe5835f1c, heap: 0x7f1672f7b2a0, libc: 0x7f1672d85410
main: 0x7f6f0aed0189, argc: 0x7fffd8eb4e9c, local: 0x7fffd8eb4eac, heap: 0x7f6f0aed42a0, libc: 0x7f6f0acd5410
main: 0x7f8106545189, argc: 0x7ffff8601bdc, local: 0x7ffff8601bec, heap: 0x7f81065492a0, libc: 0x7f8106355410
```

- randomize_va_space=0

```
main: 0x8001189, argc: 0x7ffffffee0ec, local: 0x7ffffffee0fc, heap: 0x80052a0, libc: 0x7fffff5f5410
main: 0x8001189, argc: 0x7ffffffee0ec, local: 0x7ffffffee0fc, heap: 0x80052a0, libc: 0x7fffff5f5410
main: 0x8001189, argc: 0x7ffffffee0ec, local: 0x7ffffffee0fc, heap: 0x80052a0, libc: 0x7fffff5f5410
```

Coutermeasures: PIE

- **Position Independent Executables**
  - Executables compiled such that their base address does not matter, 'position independent code'
- **PIE fully enables ASLR as code can be placed dynamically**
  - Must be enabled at compile time!!
    - gcc –pie –fPIE
- **Breaking ASLR and PIE: Find a reference to some known function**
  - Because while addresses change, the change keeps relative distance
  - e.g.: if we know printf is at 0xbf00332, we will know where is system.

ASLR and relative offsets

```
main: 0x7f80def82189, argc: 0x7fffbfce569c
main: 0x7fb811d47189, argc: 0x7fffdbd2928c
main: 0x7f95178f0189, argc: 0x7fffee962b7c

local: 0x7fffbfce56ac, heap: 0x7fffb8c4b2a0
local: 0x7fffdbd2929c, heap: 0x7fffd47952a0
local: 0x7fffee962b8c, heap: 0x7fffe67082a0

libc: 0x7f80ded85410
libc: 0x7fb811b55410
libc: 0x7f95176f5410
```