THE UNIVERSITY of NEW ORLEANS

CSCI 4621/5621 Intro to CyberSecurity

09: GLIBC HEAP EXPLOITS

Vassil Roussev

vassil@cs.uno.edu

code/4621-heap.zip

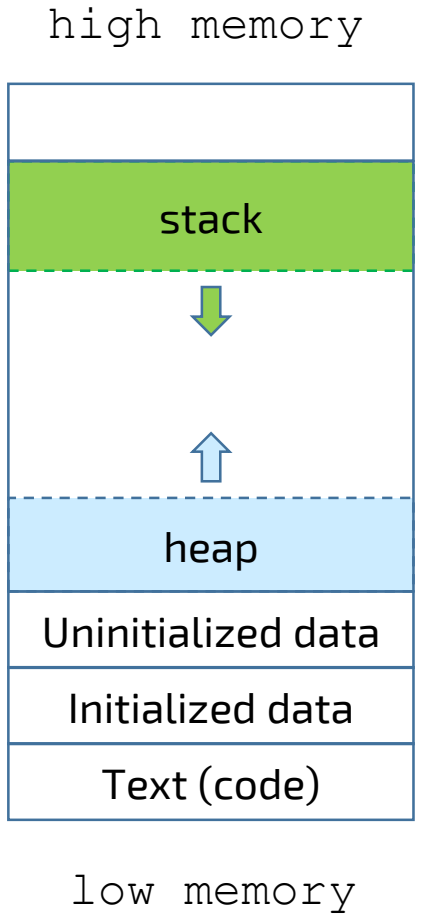
REF

- https://github.com/shellphish/how2heap
- https://azeria-labs.com/heap-exploitation-part-2-glibc-heap-free-bins/
- https://heap-exploitation.dhavalkapil.com/

GLIBC HEAP OPERATION

THE HEAP

- Dynamically allocated/deallocated memory
  - » *on request*
- Allocations managed by **glibc** (Linux)
  - » *fast performance + efficiency is critical*
  - » *memory allocator maintains metadata*
  - » *overall, a non-trivial management algorithm*
- API
  - » **malloc** / calloc (realloc / reallocarray)
  - » **free**

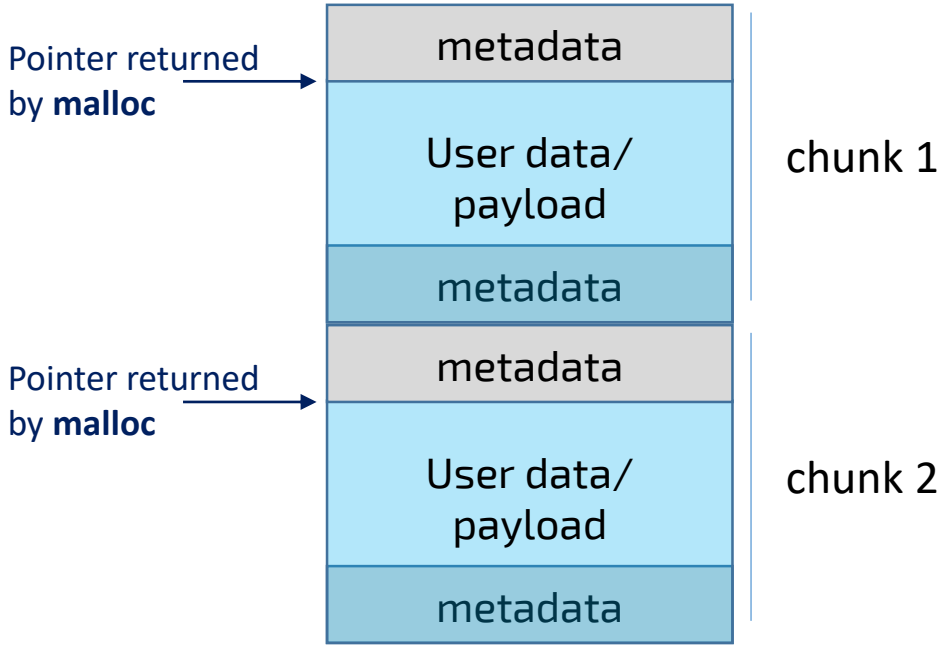


MEMORY CHUNKS

```
struct malloc_chunk {
  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk, if it is free. */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead. */
  struct malloc_chunk* fd;                /* double links -- used only if this chunk is free.
  struct malloc_chunk* bk;
  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if this chunk is free. */
  struct malloc_chunk* bk_nextsize;
};

typedef struct malloc_chunk* mchunkptr;
```

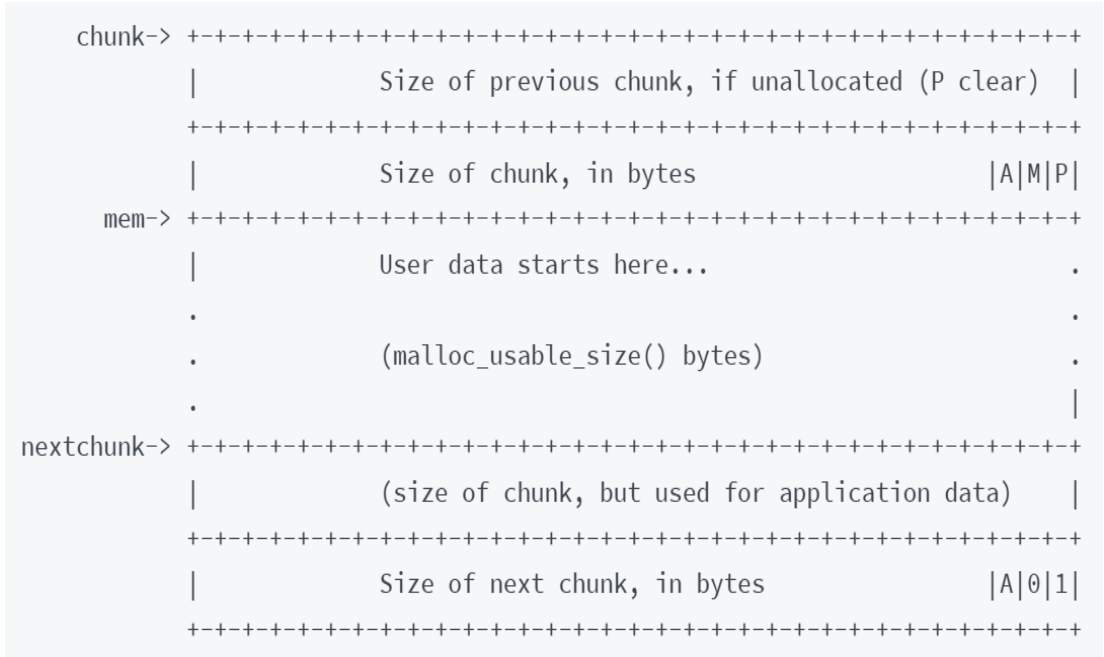
- Allocated chunks are
  - » *preceded by metadata header*
  - » *aligned on 8-byte (32-bit) / 16-byte (64-bit) boundaries*
- Free'd chunks are organized into **bins** (linked lists)
  - » 10x **FAST BINS** (fixed size): 16, 24, …, 88 bytes + header
  - » 1x **UNSORTED BIN**
  - » 62x **SMALL BINS**: 16, 24, …504
    - neighbors may be coalesced & placed in the **unsorted**
  - » 63x **LARGE BINS**
    - 32x `size` 64
    - 16x `size` 512
    - 8x `size` 4096
    - 4x `size` 32768
    - 2x `size` 262144
    - 1x `everything else`



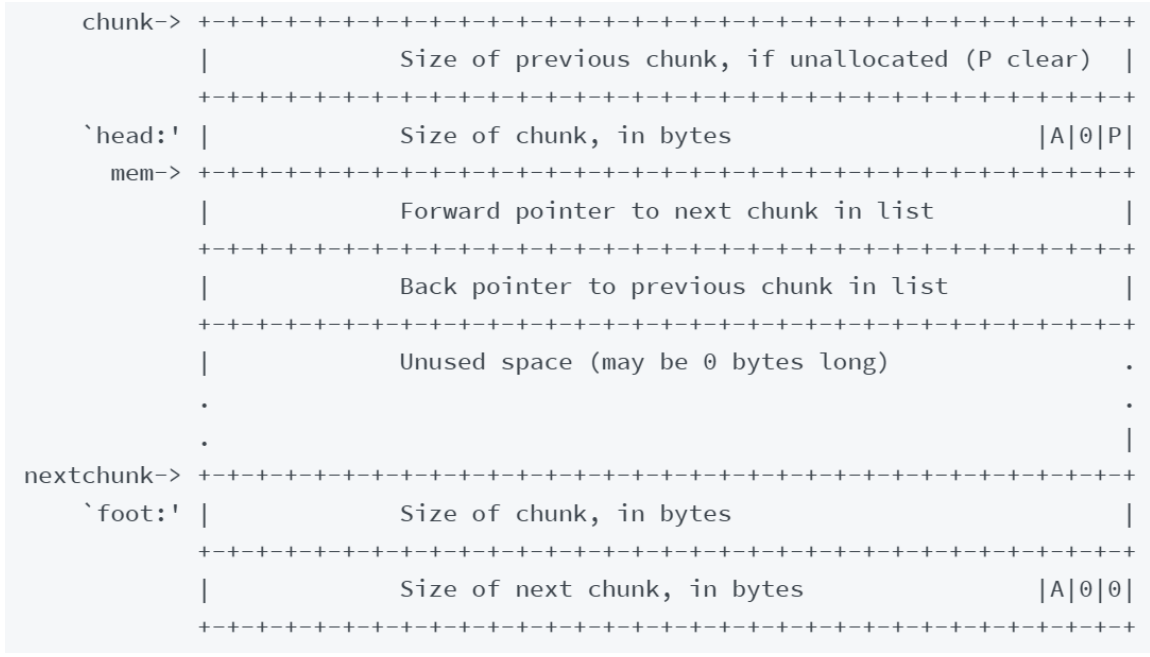
CHUNK LAYOUT

```
struct malloc_chunk {
  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk, if it is free. */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead. */
  struct malloc_chunk* fd;                /* double links -- used only if this chunk is free.
  struct malloc_chunk* bk;
  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if this chunk is free. */
  struct malloc_chunk* bk_nextsize;
};

typedef struct malloc_chunk* mchunkptr;
```



allocated



free

HEAP RULES / VIOLATIONS

- **Do not** read/write to a **malloc**-ed pointer after it has been **free**-d
  - ***Use after free*** *vulnerability*
- **Do not** use or leak uninitialized info in a heap allocation
  - ***Information leaks*** *&* ***uninitialized data*** *vulnerabilities*
- **Do not** read/write bytes after the end of an allocation
  - ***Heap overflow*** *&* ***read beyond bounds*** *vulnerabilities*
- **Do not** pass a pointer that originated from **malloc** to **free** more than once
  - ***Double free*** *vulnerability*
- **Do not** read/write bytes before the beginning of an allocation
  - ***Heap underflow*** *vulnerability*
- **Do not** pass a point that did not originate from **malloc** to **free**
  - ***Invalid free*** *vulnerability*
- **Do not** use a pointer returned by **malloc** before checking against **NULL**
  - ***Null-dereference bug*** *& occasional* ***arbitrary write vulnerability***

HOW DOES **FREE** WORK? (SIMPLE VERSION)

- free(NULL) → a valid no-op
- Sanity checks
  - *address alignment*
  - *is size realistic (not huge, not too small)*
  - *is address within **arena** boundaries*
  - *is chunk not already free*
  - *…*
- Bottom line
  - *these are sanity check that an attacker can readily bypass*

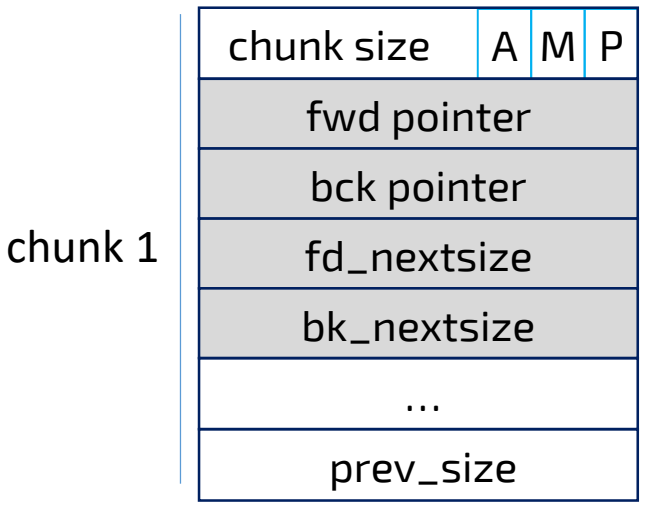
FREED CHUNKS

```
struct malloc_chunk {
  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk, if it is free. */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead. */
  struct malloc_chunk* fd;                /* double links -- used only if this chunk is free.
  struct malloc_chunk* bk;
  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if this chunk is free. */
  struct malloc_chunk* bk_nextsize;
};

typedef struct malloc_chunk* mchunkptr;
```

- Freed chunks are
  - » *added to free lists*
  - » *used to store metadata about the lists*
    - like **boundary tags**

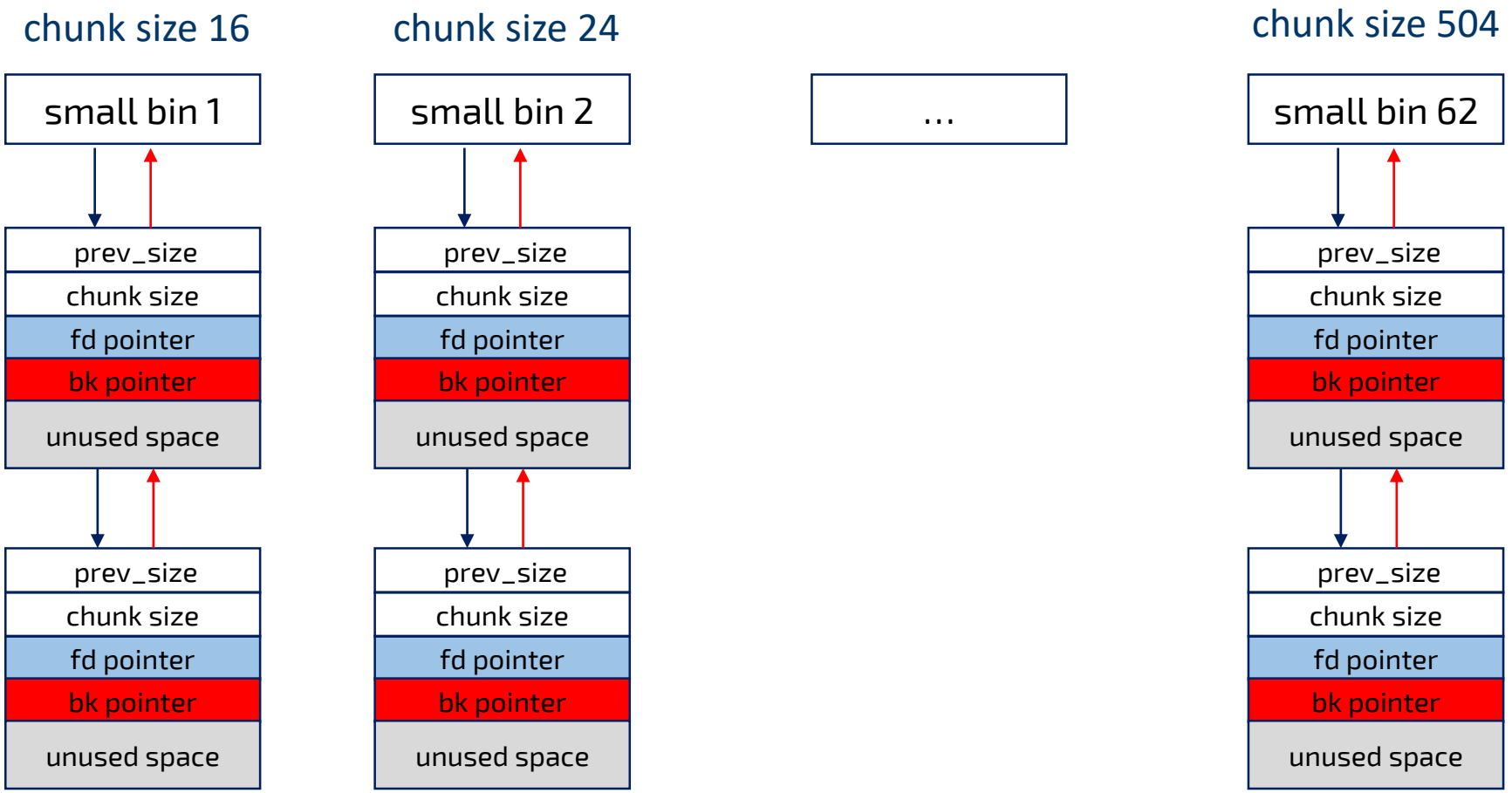
```
A (0x04) Allocated arena
M (0x02) Mmap'd chunk
P (0x01) Prev chunk in use
```



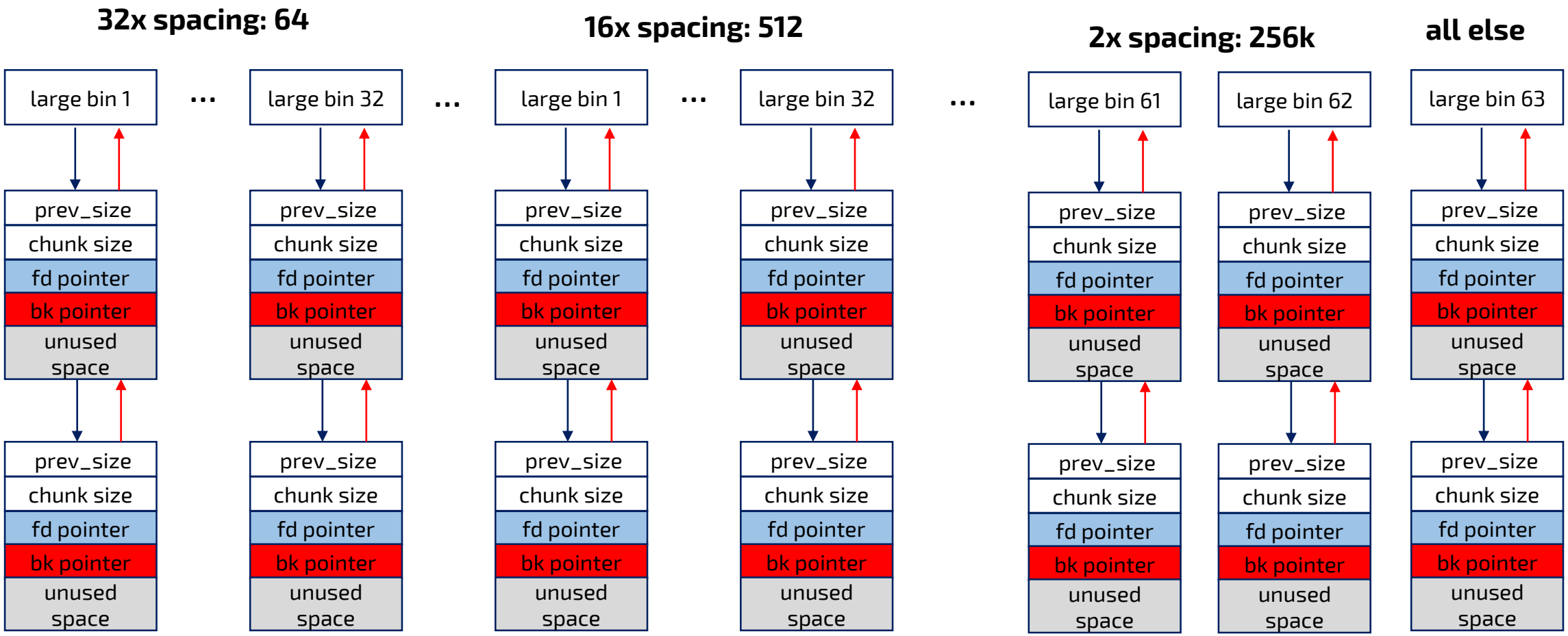
RECYCLING MEMORY W/ BINS

- Bins == lists
  - BIN[0] → N/A
  - BIN[1] → UNSORTED
  - BIN[2] – BIN[64] → SMALL BINS
  - BIN[65] – BIN[127] → LARGE BINS
- Baseline **free** algorithm
  - *If the chunk has the M bit set*
    - allocation was allocated off-heap and should be **munmap**ed.
  - *Otherwise*
    - if the chunk before this one is free, the chunk is merged backwards to create a bigger free chunk.
  - *Similarly*
    - if the chunk after this one is free, the chunk is merged forwards to create a bigger free chunk.
  - *If this potentially-larger chunk borders the "top" of the heap*
    - the whole chunk is absorbed into the end of the heap, rather than stored in a "bin".
  - *Otherwise*
    - the chunk is marked as free and placed in an appropriate bin.

SMALL BINS (< 512 / 1024 BYTES ON 32-/64-BIT)

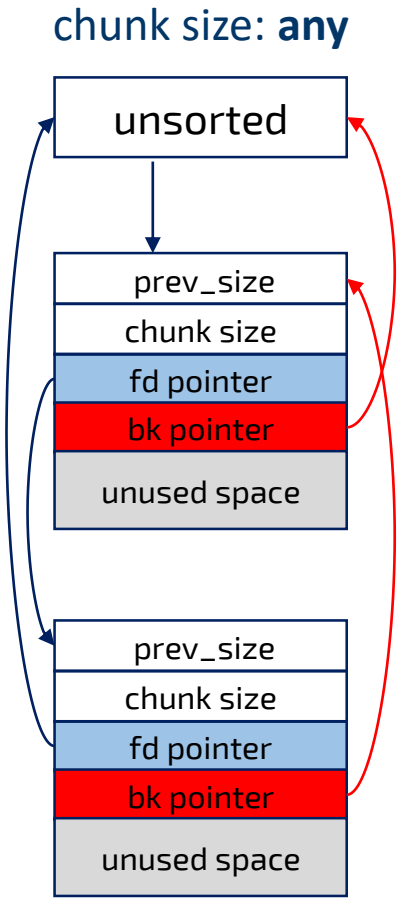


LARGE BINS


32x spacing: 64
16x spacing: 512
2x spacing: 256k
all else
large bin 1
large bin 32
large bin 1
large bin 32
large bin 61
large bin 62
large bin 63
prev_size
chunk size
fd pointer
bk pointer
unused space

UNSORTED BIN

- Rationale
  - » *often **free**d objects are clustered together*
- Strategy
  - » *merge them before placing them in the correct bin*
- Heap manager merges neighboring **de**allocations
  - » *and places them in the unsorted bin*
- **malloc**
  - » *bins checked for good fit:*
  - » *if yes*
    - allocate it
  - » *otherwise*
    - place in correct small/large bin

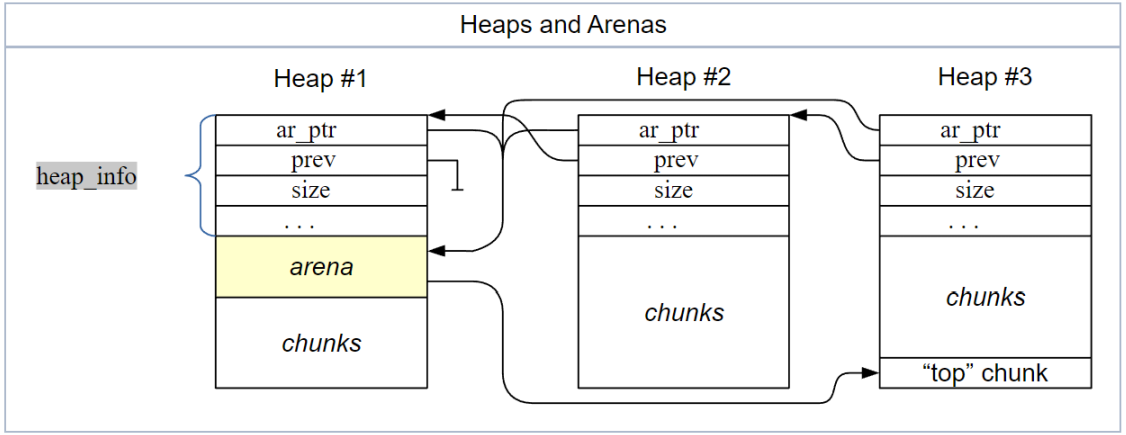


FAST BINS

- Keep small, recently released chunks in a “fast” queue
  - » *sizes: 16, 24, 32, 40, 48, 56, 64, 72, 80, 88*
  - » *no neighbor merges*
    - P bit remains 0
- Periodically, heap mgr consolidates the heap
- Triggered by
  - » *a large **malloc** request*
  - » *freeing chunks over 64KiB*
  - » ***malloc_trim** / **mallopt** invocations*

TCACHE (PER-THREAD CACHE) BINS

- Problem
  - » *lock contention in multi-threaded apps over shared resources*
  - » *cost too high for heap allocations*
- Solution
  - » *per-thread **arenas***
  - » *each thread has 64 tcache bins*
    - up to 7 same-size chunks per bin
    - 12-516 / 24-1032 bytes
- Operation
  - » *similar to fast bin*
  - » *with slow-path default (acquire arena lock)*



THE FULL **MALLOC**

1. If **tcache** chunk available → return it
2. If request is very large → mmap it
3. Otherwise, acquire arena lock and
    1. *Try fast bin/small bin recycling*
        - If corresponding fast bin exists, try to find chunk there; pre-fill tcache opportunistically
        - Otherwise, if corresponding fast bin allocate from there
    2. *Resolve all deferred **free**s*
        - "truly free" fastbin entries and consolidate to the unsorted
        - search unsorted for suitable; if found, stop. otherwise put entry in small/large/tcache
    3. *Default to basic recycling*
        - If the chunk size corresponds with a large bin, search the corresponding large bin now
    4. *Create new chunk from scratch*
        - request heap extension if not enough space
    5. *If all else fails, return NULL*

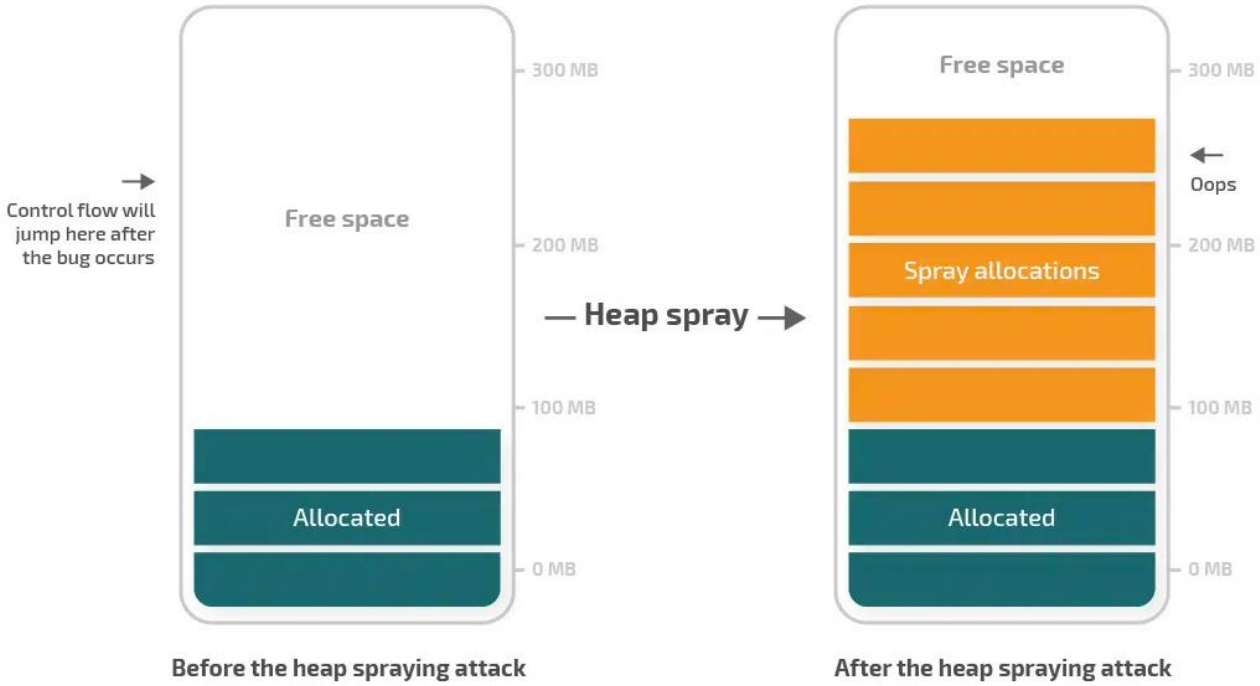
THE FULL **FREE**

1. If ptr == NULL → do nothing
2. Convert ptr to chunk
3. Perform sanity checks; abort on failure
4. If chunk fits in **tcache**, store it there
5. If chunk has **M** bit set, use **munmap**
6. Otherwise, obtain arena heap lock and
    1. *If chunk fits into a **fastbin** → place it there and return*
    2. *If chunk > 64K, consolidate **fastbins** and place merged chunks on the **unsorted** bin*
    3. *Merge the chunk backwards + forwards in **small**, **large** and **unsorted** bins*
    4. *If chunk at the top of the heap → merge into the **top***
    5. *Otherwise, store it in the **unsorted** bin*

REVISITING HEAP RULES / VIOLATIONS

- **Do not** read/write to a **malloc**-ed pointer after it has been **free**-d
  - » ***Use after free*** *vulnerability*
- **Do not** use or leak uninitialized info in a heap allocation
  - » ***Information leaks*** *&* ***uninitialized data*** *vulnerabilities*
- **Do not** read/write bytes after the end of an allocation
  - » ***Heap overflow*** *&* ***read beyond bounds*** *vulnerabilities*
- **Do not** pass a pointer that originated from **malloc** to **free** more than once
  - » ***Double free*** *vulnerability*
- **Do not** read/write bytes before the beginning of an allocation
  - » ***Heap underflow*** *vulnerability*
- **Do not** pass a point that did not originate from **malloc** to **free**
  - » ***Invalid free*** *vulnerability*
- **Do not** use a pointer returned by **malloc** before checking against **NULL**
  - » ***Null-dereference bug*** *& occasional* ***arbitrary write vulnerability***

HEAP SPRAYING




```
nops = unescape('%u9090%u9090');
s = shellcode.length + 50;

while (nops.length < s)
    nops += nops;
f = nops.substring(0, s);
block = nops.substring(0, nops.length - s);

while (block.length + s < 0x40000)
    block = block + block + f;

memory = new Array();
for (counter = 0; counter < 250; counter++)
    memory[counter] = block + shellcode;

ret = '';
for (counter = 0; counter <= 1000; counter++)
    ret += unescape("%0a%0a%0a%0a");
```

HANDS-ON: HOW2HEAP

USING HOW2HEAP EXAMPLES BY SHELLPHISH

```
git clone https://github.com/shellphish/how2heap.git
```

- Narrative

Attack	Target	Technique
First Fit	This is not an attack, it just demonstrates the nature of glibc's allocator	---
Double Free	Making `malloc` return an already allocated fastchunk	Disrupt the fastbin by freeing a chunk twice
Forging chunks	Making `malloc` return a nearly arbitrary pointer	Disrupting fastbin link structure
Unlink Exploit	Getting (nearly)arbitrary write access	Freeing a corrupted chunk and exploiting `unlink`
Shrinking Free Chunks	Making `malloc` return a chunk overlapping with an already allocated chunk	Corrupting a free chunk by decreasing its size
House of Spirit	Making `malloc` return a nearly arbitrary pointer	Forcing freeing of a crafted fake chunk
House of Lore	Making `malloc` return a nearly arbitrary pointer	Disrupting smallbin link structure
House of Force	Making `malloc` return a nearly arbitrary pointer	Overflowing into top chunk's header
House of Einherjar	Making `malloc` return a nearly arbitrary pointer	Overflowing a single byte into the next chunk