```
___________________________.
|;;|                     |;;||
|[]|---------------------|[]||
|;;|                     |;;||
|;;|   -------------     |;;||
|;;|   | COMP6447 |      |;;||
|;;|   -------------     |;;||
|;;|      malloc(4)      |;;||
|;;|                     |;;||
|;;|_____________________|;;||
|;;;;;;;;;;;;;;;;;;;;;;;;;;;||
|;;;;;;_______________ ;;;;;||
|;;;;;|  ___          |;;;;;||
|;;;;;| |;;;|         |;;;;;||
|;;;;;| |;;;|         |;;;;;||
|;;;;;| |;;;|         |;;;;;||
|;;;;;| |;;;|         |;;;;;||
|;;;;;| |___|         |;;;;;||
\_____|_______________|_____||
 ~~~~~^^^^^^^^^^^^^^^^^~~~~~~
```

# Heap

- What is the heap?
- Overview of ptmalloc2 dynamic allocator
  - Malloc
  - Free
  - Bins?
- Glibc Heap exploitation in 32bit linux systems
  - Use after free
  - Double free
  - Chunk Forging
- Heap spraying

# why is heap overflow different to the stack

- What we have done so far is exploit bugs in certain programs
- Goal is often to control the stack ret addr or the GOT
- We are at attacking the implementation of heaps rather than shit program code
- Because we are attacking the implementation of heaps, what might work on your linux program, might not work on a different program on the same computer, since different implementations exist between programs
- **Heap is hard**
- **You need to actually understand the program to do any heap challenge**
- **You can’t just guess and check like previous weeks**
- **If you can’t wrap your head around how the heap looks (use pen and paper) you won’t succeed**

# Heap exploitation is linked to your libc version

- If you’re using a computer with a more uptodate version or older version of libc, the solution to this weeks challenges might not work.
- I recommend using a docker container to run the binaries / your scripts in

**Example cmd (docker has pwntools / pwndbg / etc)**

cd wargames

docker run -d --rm -h banana --name banana -v $(pwd):/ctf/work --cap-add=SYS_PTRACE skysider/pwndocker

docker exec -it banan /bin/bash

/ctf/work/challengename

# To understand the heap you must be the heap

To understand heap exploitation

**You must understand how the heap works**

**Most of this lecture will be explaining how the heap works**

Heap exploitation methods are **trivial** if you understand the heap

# What is malloc

**dlmalloc** - General purpose allocation

**ptmalloc2** - glibc

- **Fast** for multi threaded applications
- **Fast** for really small allocations

**jemalloc** - Firefox

**tcmalloc** - chrome

- is faster when threads are created/destructed…
- Uses a shittonne of memory

The heap memory segment is an area of memory used for dynamic allocations

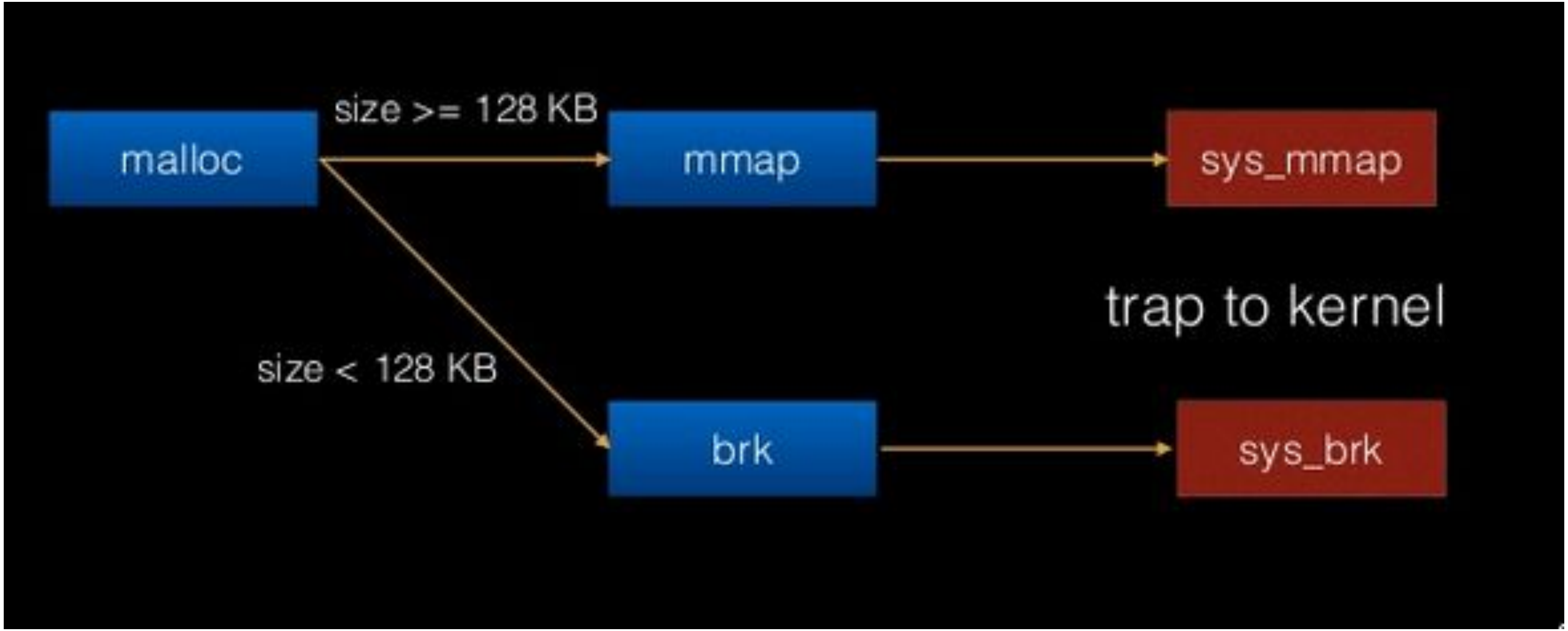
meaning that blocks of memory can be allocated and freed in an arbitrary order and accessed multiple times (as opposed to the stack, which is Last-In-First-Out).

Unlike memory in the stack, memory allocated to the heap must be explicitly de-allocated when the data is no longer needed

Higher level languages abstract deallocation/freeing away from the developer through a garbage collector

Majority of memory usage in large programs come from this region

# This photo is a lie



# This isnt

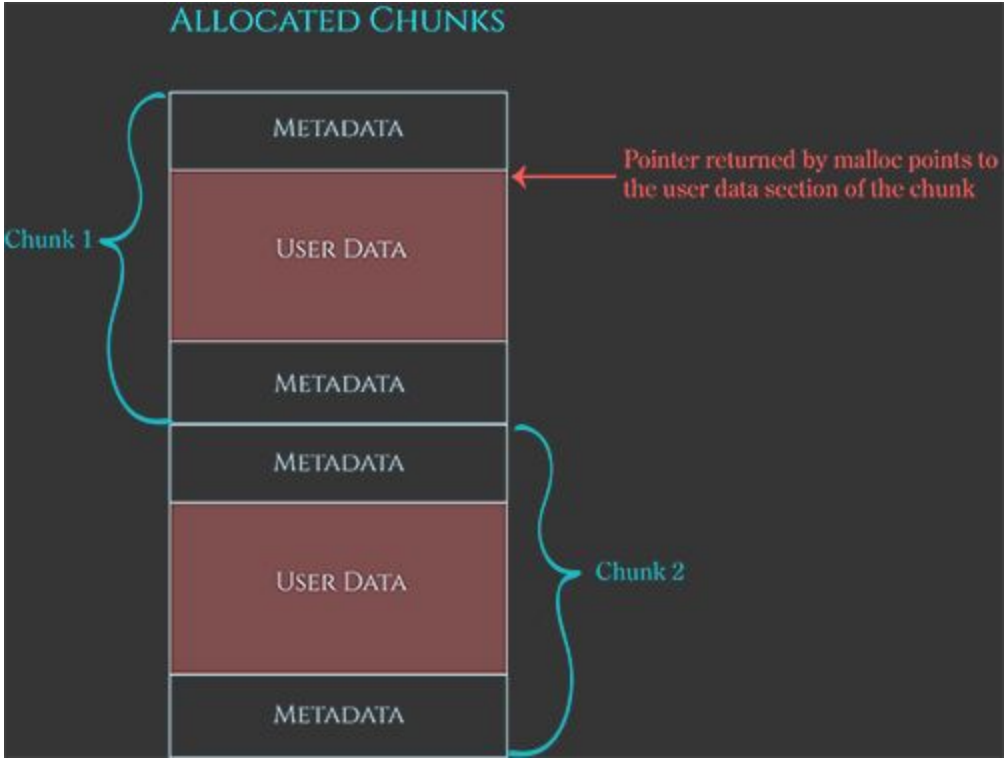
Usually a first-fit allocator

First searches recently free chunks,

then new chunks

Stores metadata before/after data

Split into free and not free chunks




Mixing data + important stuff is bad

Glibc's malloc is chunk-oriented.

It divides a large region of memory into chunks of various sizes.

Each chunk includes **metadata about how big it is**

and **thus where the adjacent chunks are**

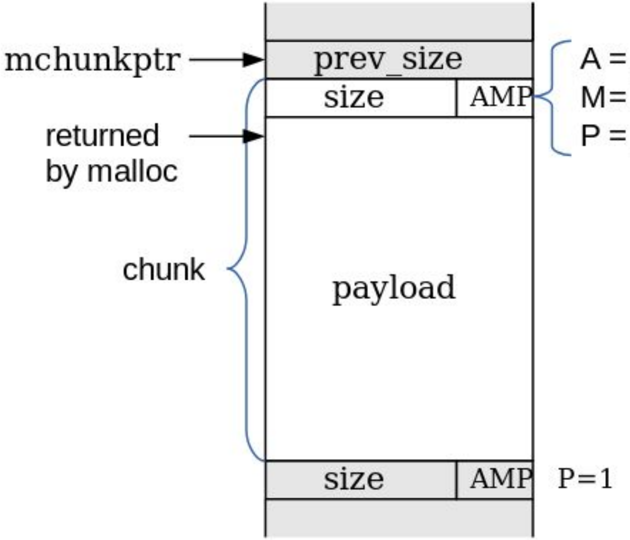
When a chunk **is in use** by the application, **the only metadata stored is the size of the chunk.**

When the chunk **is free'd**, the memory that used to be application data is **re-purposed** for additional arena-related information, such as **pointers within linked lists**

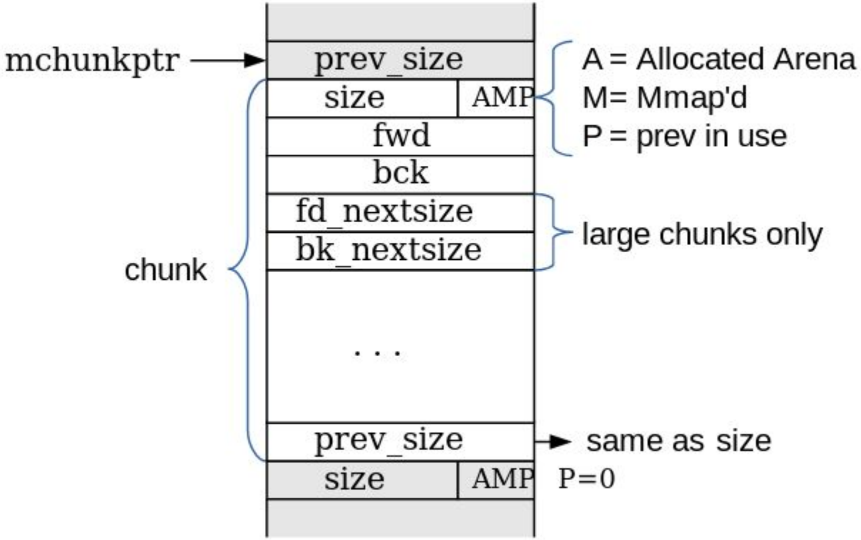
```c
struct malloc_chunk {
  INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if 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 free. */

 struct malloc_chunk* bk_nextsize;
};
```

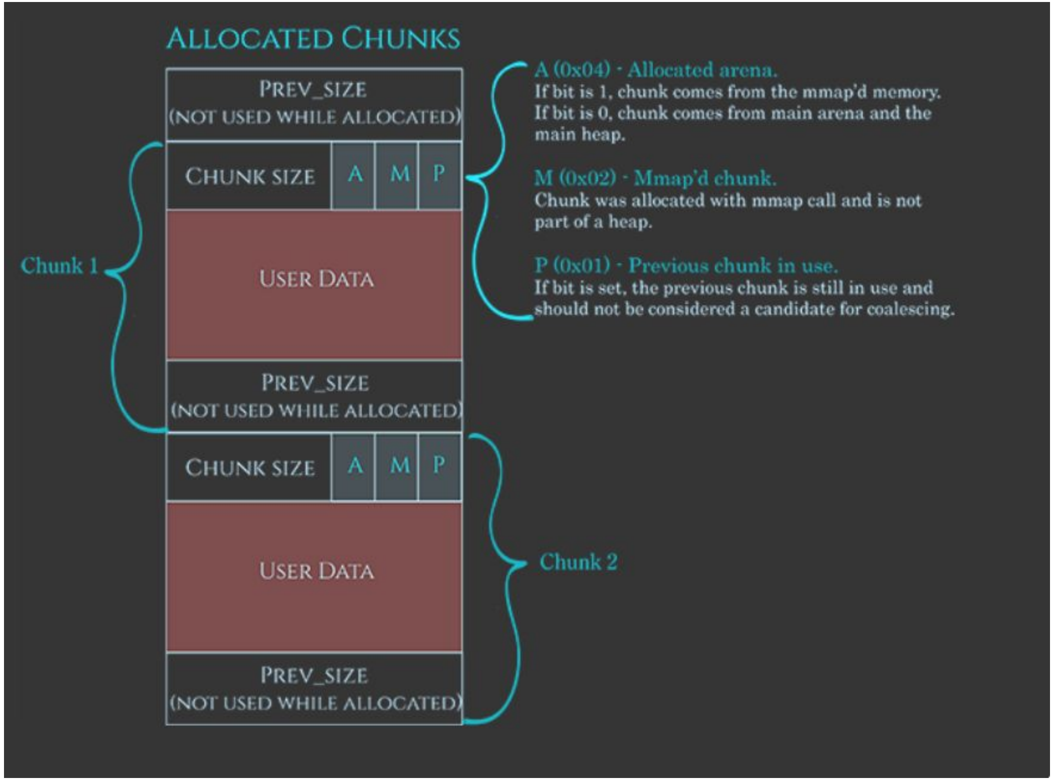
- In order to ensure that a chunk's payload area is large enough to hold the overhead needed by malloc, the minimum size of a chunk is 4*sizeof(void*)
- In 32 bit this means the minimum chunk size is 0x10
- All chunk sizes are aligned to 8 byte boundaries
- Valid chunk sizes are 0x10, 0x18, 0x20, 0x28, 0x30, etc
- Since sizes are 8 byte aligned, the last 3 bits of size are unused
- These are used to store a bitmap of information on the chunks
    - Bit 1 - 1 if chunk is in main arena
    - Bit 2 - 1 if chunk is mmap’d and not part of a heap
    - Bit 3 - 1 if previous chunk is in use

In-use Chunk




Free Chunk




ALLOCATED CHUNKS
PREV_SIZE
(NOT USED WHILE ALLOCATED)
CHUNK SIZE
A
M
P
USER DATA
PREV_SIZE
(NOT USED WHILE ALLOCATED)
CHUNK SIZE
A
M
P
USER DATA
PREV_SIZE
(NOT USED WHILE ALLOCATED)
Chunk 1
Chunk 2
A (0x04) - Allocated arena.
If bit is 1, chunk comes from the mmap'd memory.
If bit is 0, chunk comes from main arena and the main heap.
M (0x02) - Mmap'd chunk.
Chunk was allocated with mmap call and is not part of a heap.
P (0x01) - Previous chunk in use.
If bit is set, the previous chunk is still in use and should not be considered a candidate for coalescing.

Within each **arena**, chunks are either **in use** by the application or **they're free**.

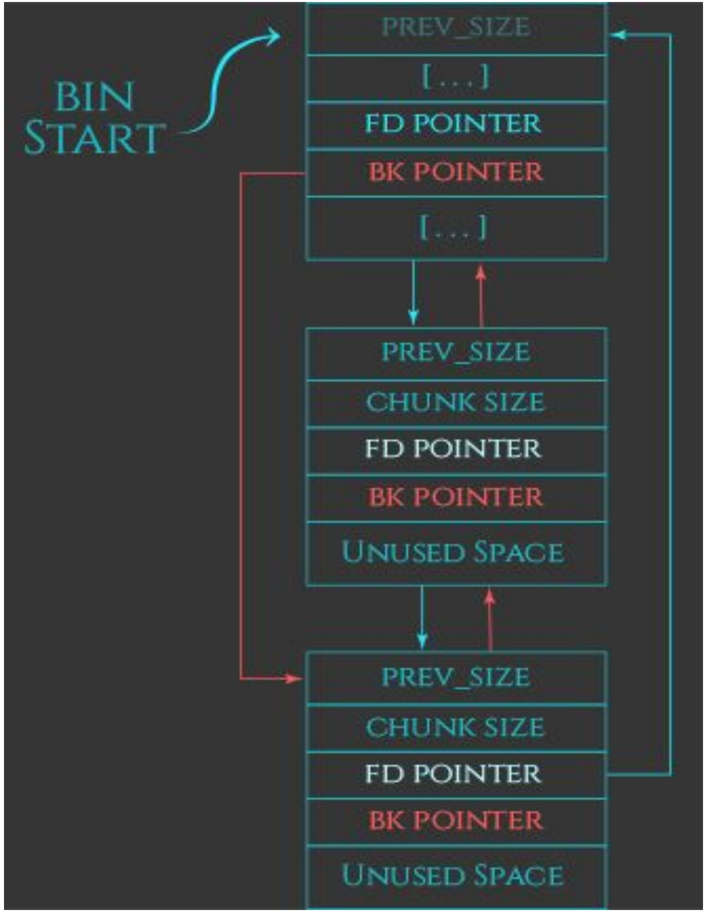
In-use chunks **are not tracked** by the arena.

Free chunks are stored in various lists based on size and history, so that the library can quickly find suitable chunks to satisfy allocation requests.

# Free'ing chunks

Free needs to be **fast**.

- Implemented using **different sized** bins
- Smaller the chunk, less secure the bin is (means its faster to allocate/deallocate)
- All chunks have these pointers, whether or not they're used depends on
    - If the chunks free
    - Size of chunk
- Bins are just arrays of linked lists of chunks
- The nodes in the linked lists are **old chunks**

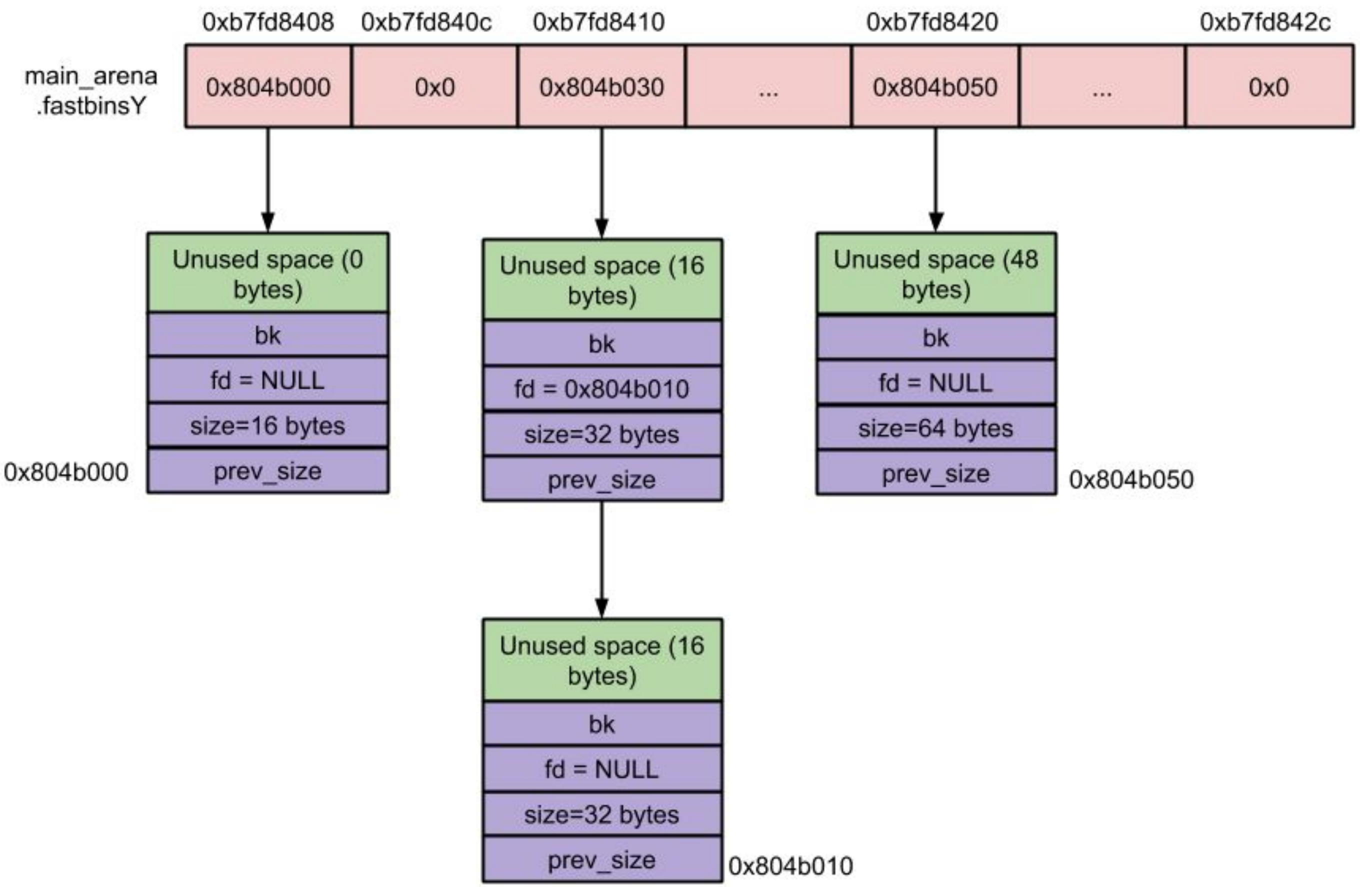

BIN
START
PREV_SIZE
[ . . . ]
FD POINTER
BK POINTER
[ . . . ]
PREV_SIZE
CHUNK SIZE
FD POINTER
BK POINTER
UNUSED SPACE
PREV_SIZE
CHUNK SIZE
FD POINTER
BK POINTER
UNUSED SPACE


Lower Addresses
CHUNK 1
CHUNK 2
FREE CHUNK 3
CHUNK 4
FREE CHUNK 5
CHUNK 6
CHUNK 7
TOP CHUNK
Higher Addresses

# Fast bins

Fast

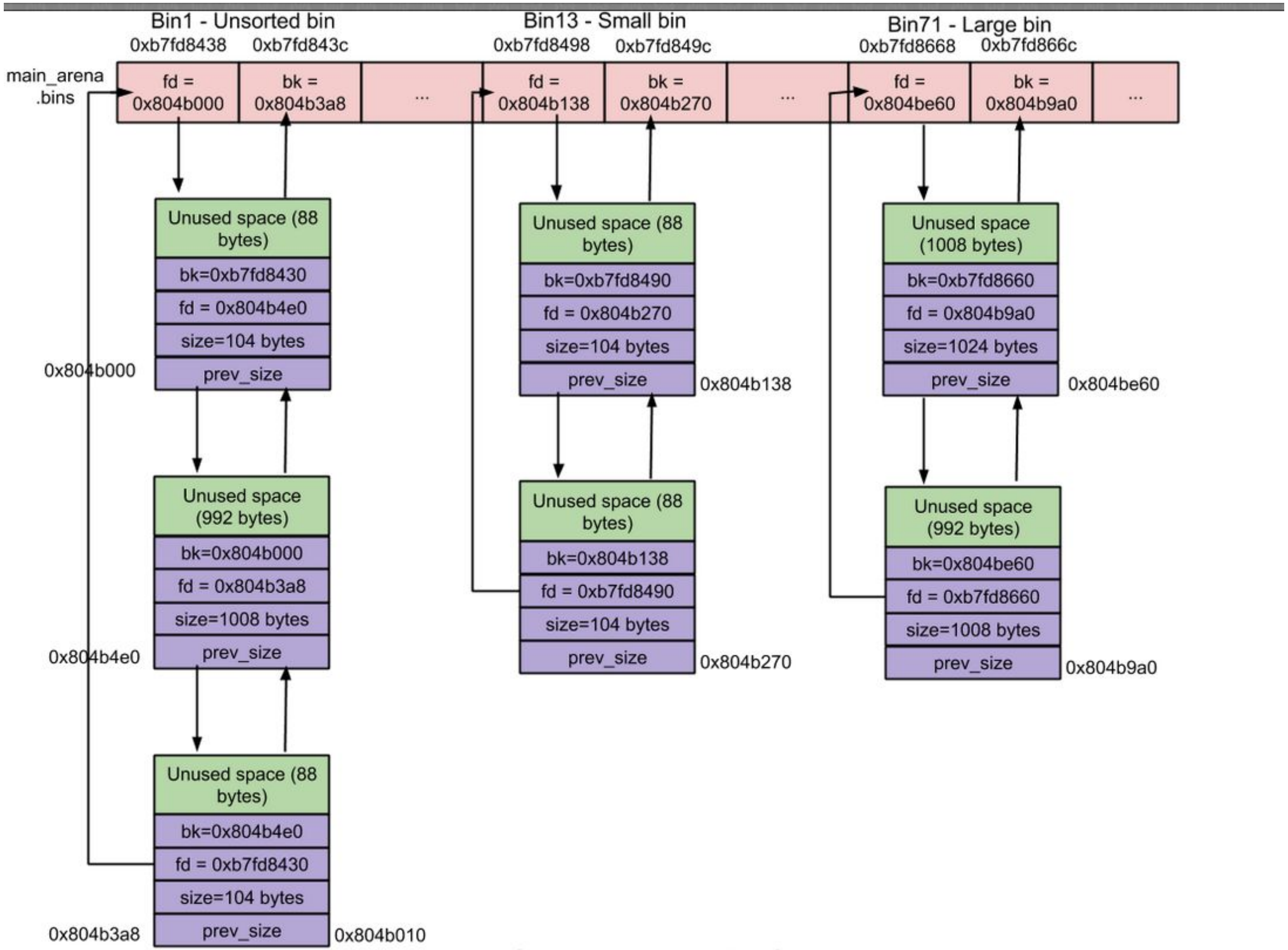
- Small chunks are stored in **size-specific** bins.
- Chunks added to a fastbin **are not combined** with adjacent chunks
- Fastbin chunks are stored in a **single linked list**, since they're all the same size and chunks in the middle of the list need never be accessed
- There are **10** fastbins, size 16,24,32,etc




Fast Bin Snapshot

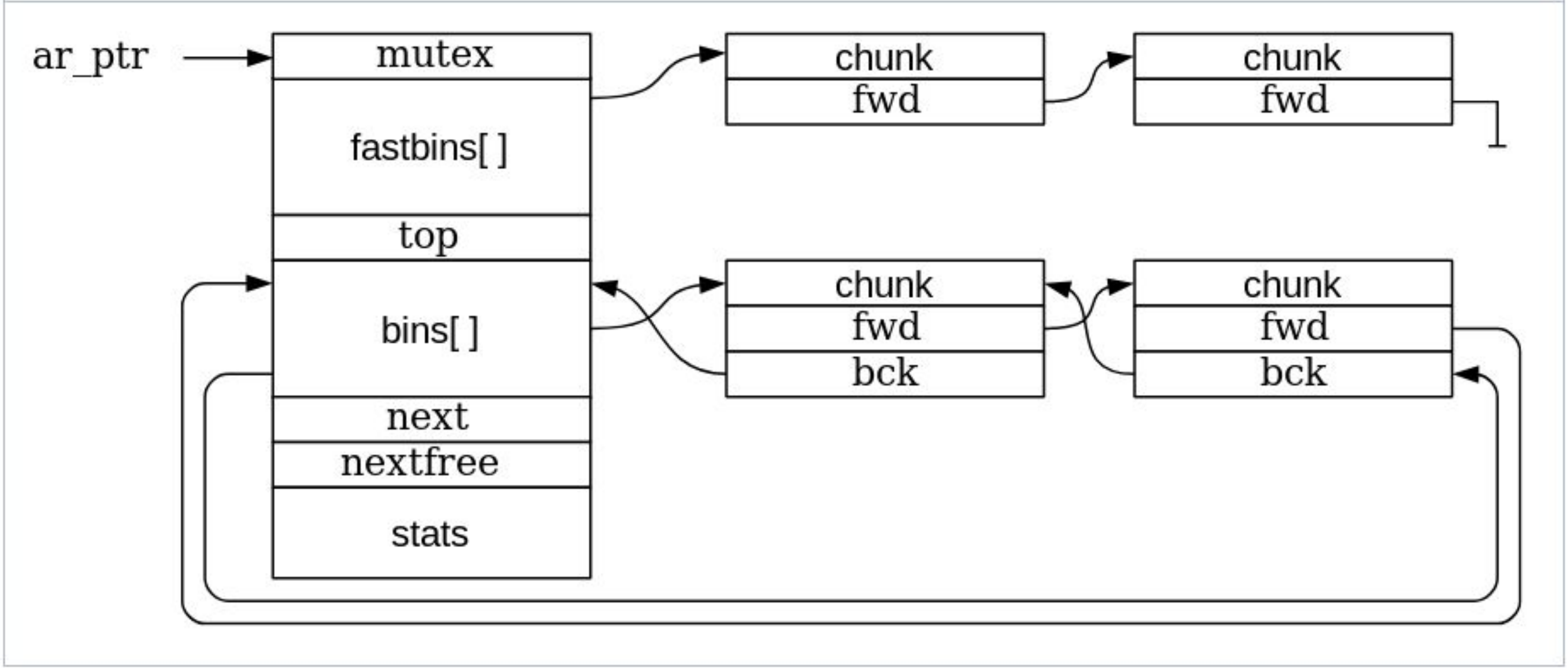
# Non fast

- Unsorted
    - When chunks are free'd they're initially stored in a single bin.
    - They're sorted later, in malloc, in order to give them one chance to be quickly re-used.
- Others
    - The normal bins are divided into 62 "**small**" bins, where each chunk is the same size, and "**large**" bins
    - When a chunk is added to these bins, they're first combined with adjacent chunks to "coalesce" them into larger chunks.
    - **Small** and **large** chunks are **doubly-linked** so that chunks may be removed from the middle (**to be merged with nearby chunks**)
        - Large chunks may be split into smaller chunks on a malloc




Unsorted, Small and Large Bin Snapshot

## Free, Unsorted, Small, and Large Bin Chains



# Its 2017

- In 2017 GLIBC was updated with a new bin type
    - Tcache
- very similar to fastbins
    - Single linked list
- Less checks
- Easy to hack
- Double free are easy
- No checks on header

In 2019

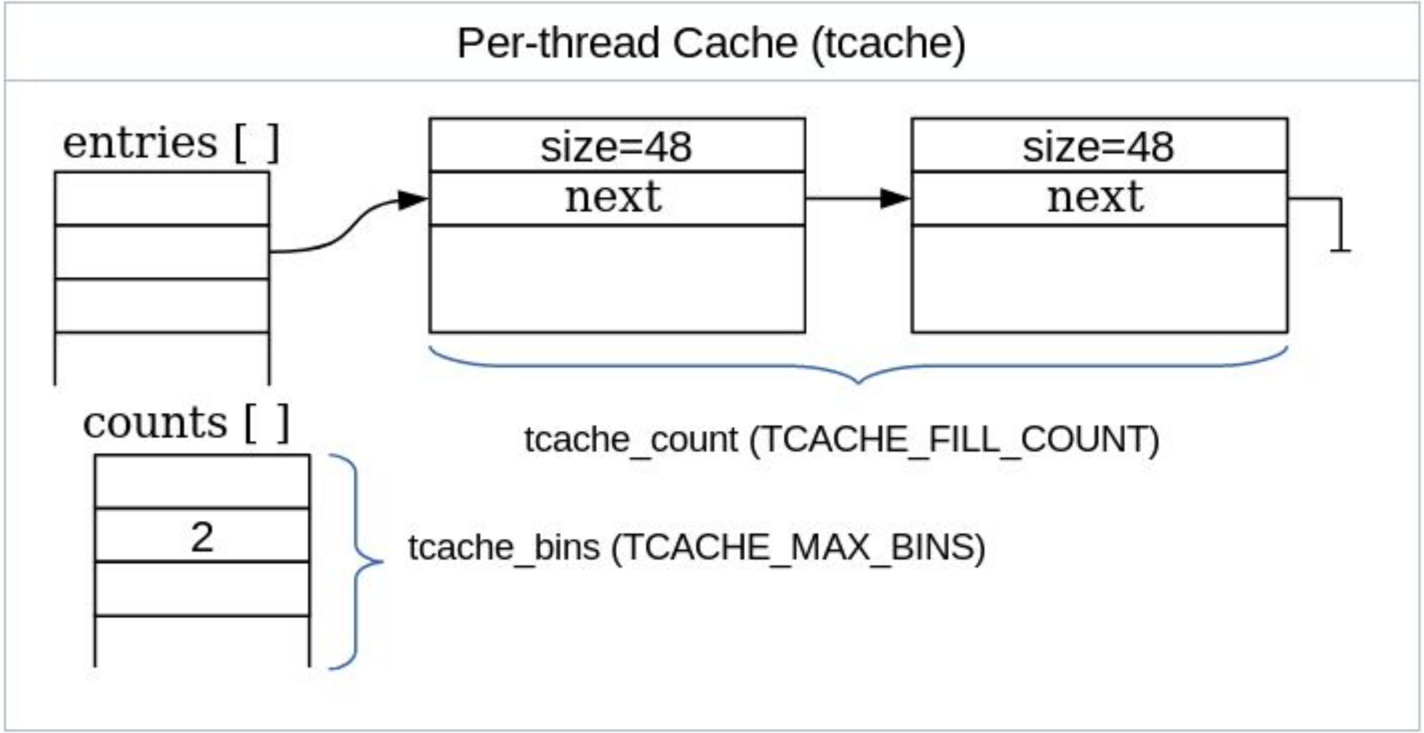
- In latest **GLIBC**, 2019, they added checks for double frees… oh no

- Unlike fastbins, the tcache is limited in how many chunks are allowed in each bin (I think 7 by default).
- If the tcache bin is empty for a given requested size, the normal malloc routine is used (other bins)
- Tcache are interesting from a exploitation perspective
    - They’re new
- Up until recently, there were very little (none) mitigations

# Example tcache code from last year

```c
/* Caller must ensure that we know tc_idx is valid and there's

   for more chunks.  */

static __always_inline void

tcache_put (mchunkptr chunk, size_t tc_idx)

{

  tcache_entry *e = (tcache_entry *) chunk2mem (chunk);

  assert (tc_idx < TCACHE_MAX_BINS);

  e->next = tcache->entries[tc_idx];

  tcache->entries[tc_idx] = e;

  ++(tcache->counts[tc_idx]);
}
```

```c
/* Caller must ensure that we know tc_idx is valid and there's

   available chunks to remove.  */

static __always_inline void *

tcache_get (size_t tc_idx)

{

  tcache_entry *e = tcache->entries[tc_idx];

  assert (tc_idx < TCACHE_MAX_BINS);

  assert (tcache->entries[tc_idx] > 0);

  tcache->entries[tc_idx] = e->next;

--(tcache-&gt;counts[tc_idx]); // Get a chunk, counts one less
  return (void *) e;

}
```



Per-thread Cache (tcache)
entries [ ]
size=48
next
size=48
next
tcache_count (TCACHE_FILL_COUNT)
counts [ ]
2
tcache_bins (TCACHE_MAX_BINS)

# Malloc is a **first fit** allocator

- Practical example of understanding malloc/free implementation

```
char *a = malloc(300);
char *b = malloc(250);


free(a);


a = malloc(250);
```

'a' **freed**.

head -> **a** -> tail

'malloc' **request**.

head -> **a2** -> tail [ 'a1' is returned ]

a chunk is split into two chunks 'a1' and 'a2'

- 'a' freed.
  - head -> a -> tail
- 'b' freed.
  - head -> b -> a -> tail
- 'c' freed.
  - head -> c -> b -> a -> tail
- 'd' freed.
  - head -> d -> c -> b -> a -> tail
- 'malloc' request.
  - head -> c -> b -> a -> tail [ 'd' is returned ]

```
char *a = malloc(20);
char *b = malloc(20);
char *c = malloc(20);
char *d = malloc(20);

free(a);
free(b);
free(c);
free(d);

a = malloc(20);
b = malloc(20);
c = malloc(20);
d = malloc(20);
```

# Demo of first fit + bins

+ questions

# Finally exploitation

- There are many heap exploitation techniques
  - Double free
  - Forging chunks
  - Unlink
  - Shrinking free chunks
  - House of spirit
  - House of love
  - House of Force
  - House of …
- For most the goal is to make malloc return a arbitrary pointer
  - This talk will cover attack methods for glibc 2.26
- All methods have certain
  - dependencies of what you control
  - Specific control you get after an attack

**Alone, these aren’t useful, but when used together...**

Use after free

Double Free

Leaking with small chunks

Forging chunks (with overflows)

Heap spray

# Goals

- Why do we want to exploit stuff on the heap
  - What is our goal?
    - Run our own code?
- How?
  - Change variables to give us more permissions? (think sudo or bash)
- What are our tools?
  - Buffer overflows?
  - Logic bugs
- **Let's start by saying our goal is to get two heap chunks to overlap**
  - This means if we control one of the chunks (ie: name variable)
  - We can overwrite the second chunk which might have more sensitive data
    - like program metadata like global function pointers
      - Global Offset Table
      - C++ Vtable
      - etc

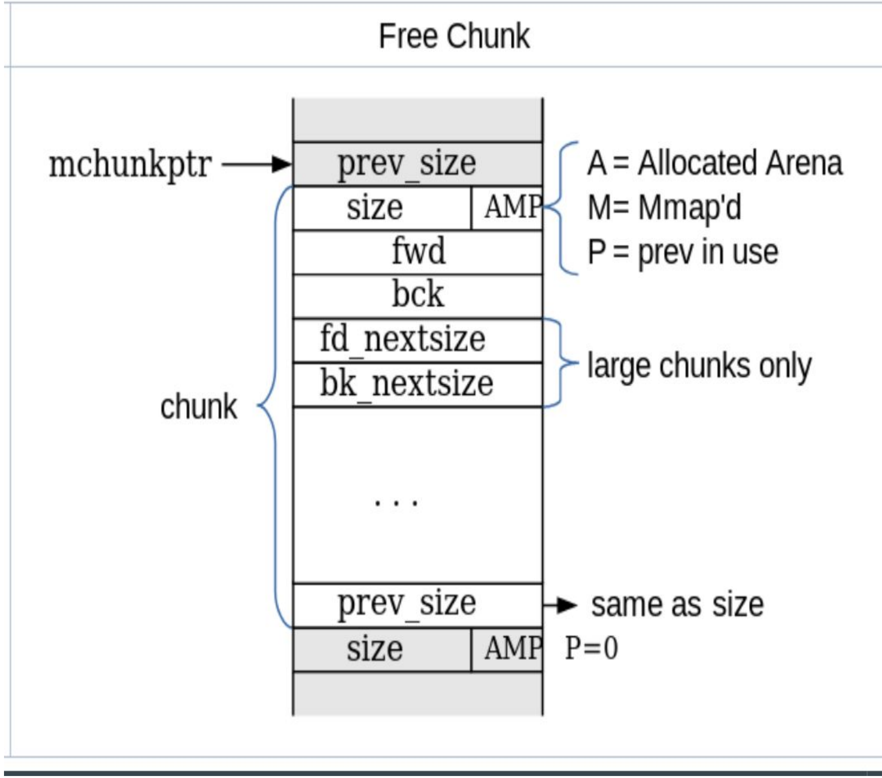
# Use after free

You free something

- Then you use it

**How can we leverage this to our advantage?**

- We can corrupt the free linked list structure
- We can change the free structures to allow us to allocate our own memory

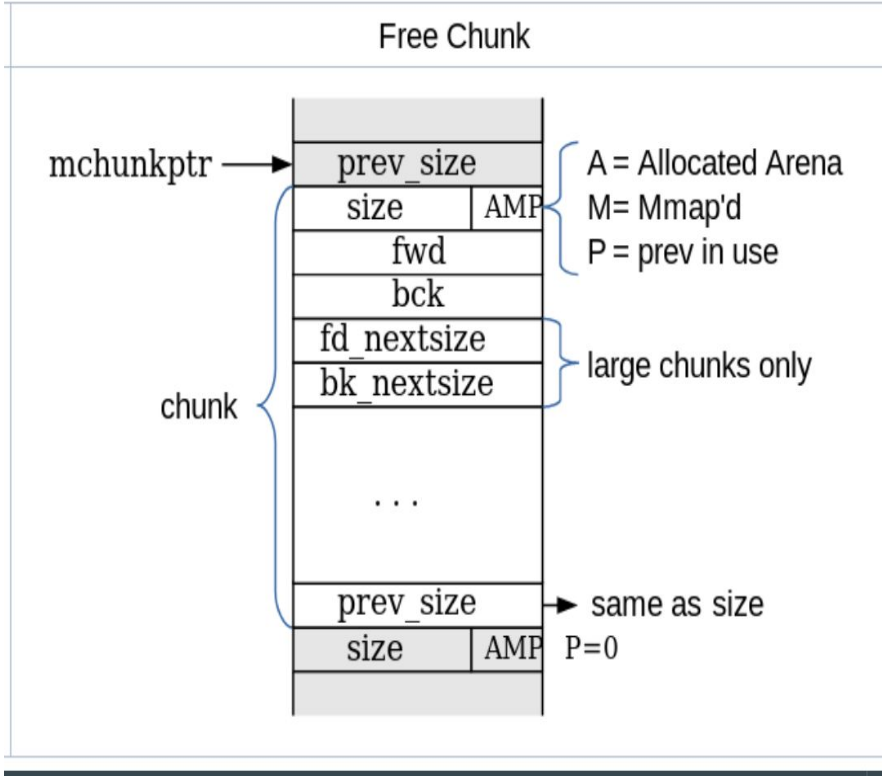


# Use after free

- After free a chunk is repurposed
- Can control list pointers
    - Can craft arbitrary chunks

Demo

- So what?
- Next allocated chunk you control
- Can point it to anywhere
    - GOT
    - malloc_hook, free_hook
    - vtable



# Sometimes UAF isn't easy to spot

- Might have been used

  by another thread?

```
void gc(void* obj) {
    if (obj != NULL) {
        free(obj);
    }

    ... 1000 lines of code

    memcpy(obj[0], obj, 8);
}
```

# Double free

Demo time

**How can we leverage this?**

```
struct important_struct {
    char* name;
    int is_admin;
}
```

# Tcache bad

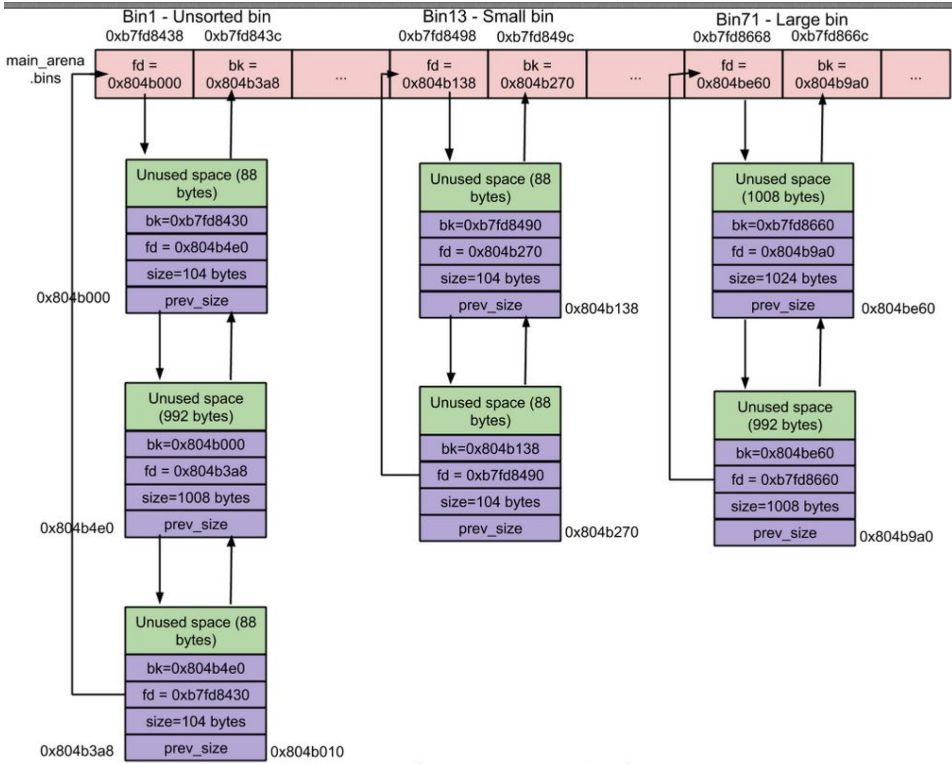
- Can’t free something right after itself
- Fastbin/Tcache Freelist check immediate double free
    - free(a);
    - free(a);
- **Solution???**
    - free(a);
    - free(b);
    - free(a);
- Allows you to
    - Manipulate free list
    - Control other data structures not normally editable

# Leaking memory with a small chunk

Remember a small chunck is a doubly linked list.

The first element points **back** into libc.

If you can somehow leak the first 4 bytes of a free small chunk, you have leaked an address to libc!

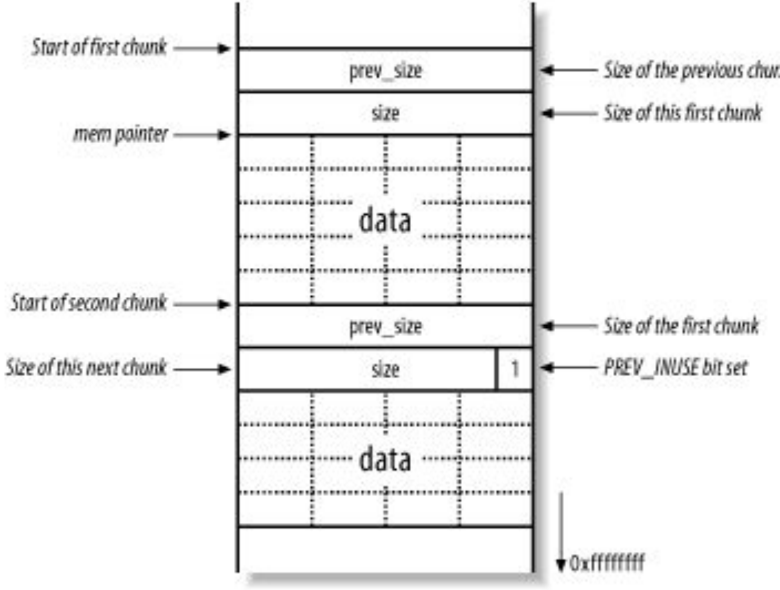


Unsorted, Small and Large Bin Snapshot

# Forging chunks

## Setup

- 2 chunks next to each other
  - The first chunk has a buffer overflow
- Two cases
  - If second chunk is free
    - Can overflow into freelist metadata
      - Cause new chunks to be allocated
  - If second chunk is in use
    - Can overflow into chunk metadata
      - Modify size, cause chunk to be put into a different sized bin
      - Can be used to convert fastbin into smallbin, and then leak an address with it



# Ok so now I have a corrupted chunk

Example

1. We create a fake chunk pointing to some function pointer
2. We can overwrite the function pointer with a pointer to our own code
3. Where do we put the code?
4. Any controlled data we have. Is code :)
5. Remember how different programs can use different version of malloc?
    a. How does that work?

**Gives us a read/write primitive similar to format strings**

# Malloc Hook functions

Literally a global function pointer in every program

- Calling `malloc` calls `__malloc_hook`
  - If you control __malloc_hook
    - Everytime someone calls malloc, theyre calling your code

Pop shellz not pills

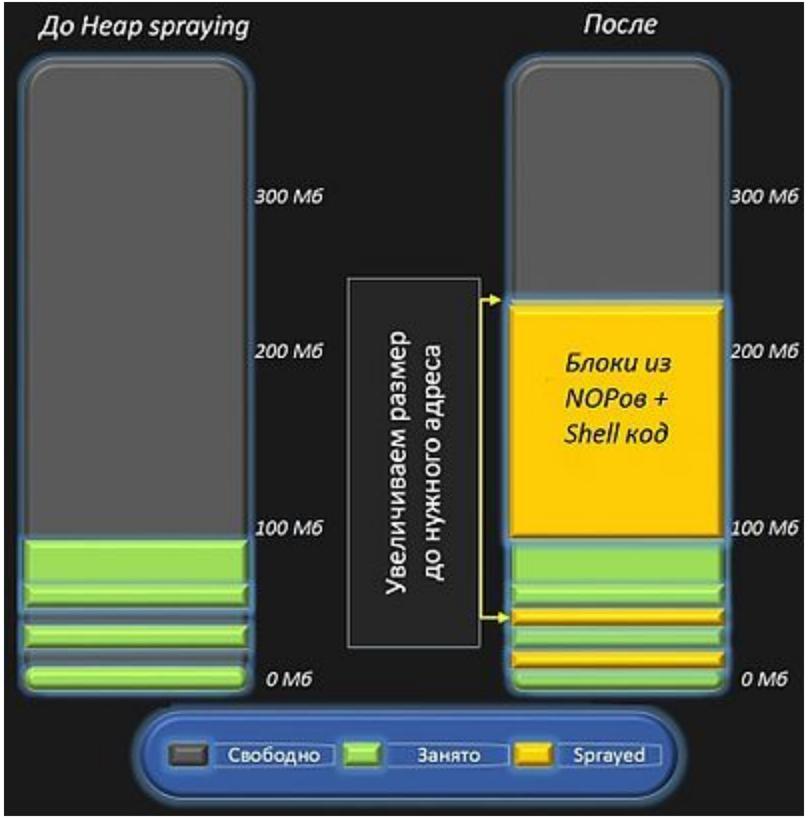
# Heap spraying

- Most of these exploits need us to be able to create 2 or more chunks sequentially
- This isnt always possible
  - Remember we aren’t calling malloc
    - We are the user
      - We are doing things like
        - Uploading images to software
        - Setting our name variable
        - Deleting posts
          - These are our *malloc* and *free*

**How do we ensure the heap looks nice and groomed ( like me :-) )**


HEAP SPRAYING
MAKES BABY
JESUS CRY
memegenerator.es

- We want to reliably know where things are in memory
- Use a malloc primitive to allocate a bunch of tiny chunks
  - Fill in all the gaps
  - Since there are no *free bins*
  - Malloc will just allocate on the top of the heap
  - We can reliably know where our chunks are
  - Really helpful for putting our **shellcode**
    - So we know where it is
- Reliability is a big thing
  - Jailbreaks are almost impossible without heapspray



# More modern / complex heap overflows

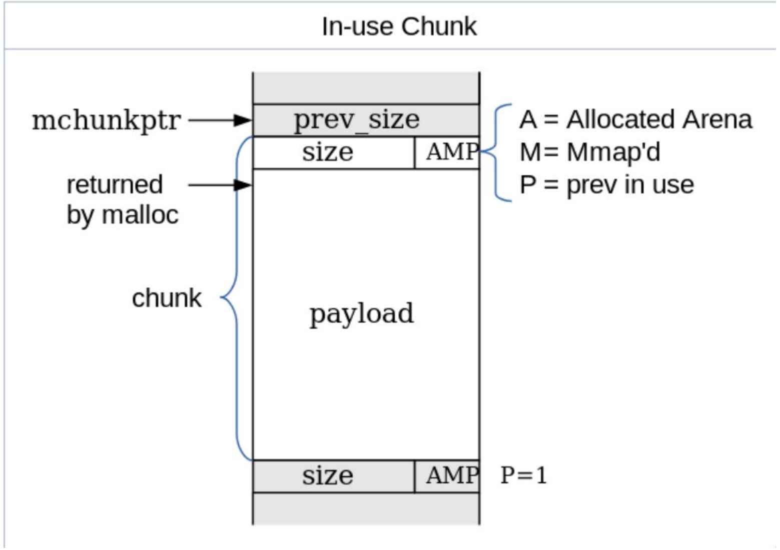
House of **Einherjar**

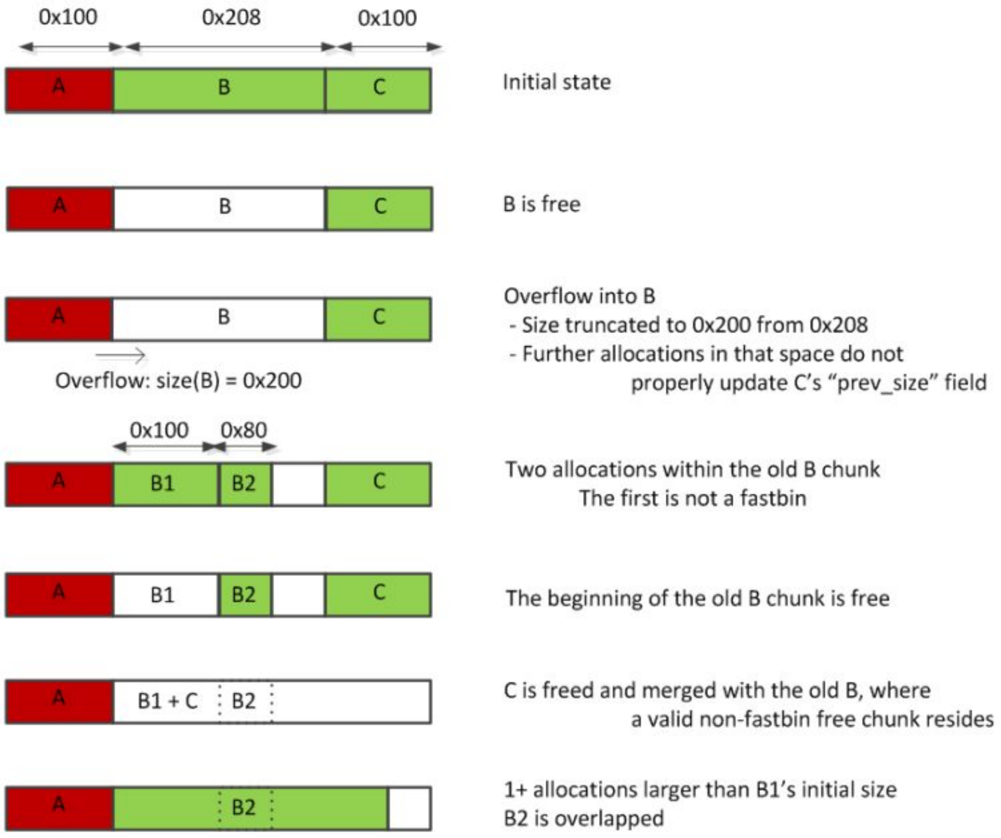
Typically useful in **off by one bugs**

Requires a **1 byte overflow**

# More on forging

- **House of Einherjar**
- Typically useful in off by one bugs
- Requires a 1 byte overflow



- Tutorials will go over some **simple heap** examples
- Most of this weeks wargames are up to you to learn the content