glibc malloc

Insu Yun

Today’s lecture

- Understand ptmalloc

Heap

- A region for dynamically allocated memory
- Can use with standard library functions: malloc, calloc, free, …

```c
// Dynamically allocate 10 bytes
char *buffer = (char *)malloc(10);

strcpy(buffer, "hello");
printf("%s\n", buffer); // prints "hello"

// Frees/unallocates the dynamic memory allocated earlier
free(buffer);
```

malloc / free

The malloc() function allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized. If size is 0, then malloc() returns either NULL, or a unique pointer value that can later be successfully passed to free().

The free() function frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc(), or realloc(). Otherwise, or if free(ptr) has already been called before, undefined behavior occurs. If ptr is NULL, no operation is performed.

https://man7.org/linux/man-pages/man3/malloc.3.html

glibc malloc (ptmalloc2)

- A default allocator for Linux, which we are working with
  - Many similarties with others (e.g., a Windows allocator or others)

- Derived from ptmalloc, which is derived from dlmalloc (Doug Lea malloc)

- “Heap” style: A larger region of memory (heap) can contain chunks of various sizes
- Allows for multiple heaps for one application

Terminologies

- Arena: A structure that is shared among one or more threads which contains references to **one or more heaps**, as well as linked lists of chunks within those heaps which are "free“. Threads assigned to each arena will allocate memory from that arena's **free list**s.

- Heap: **A contiguous region of memory** that is subdivided into chunks to be allocated. Each heap belongs to exactly one arena.

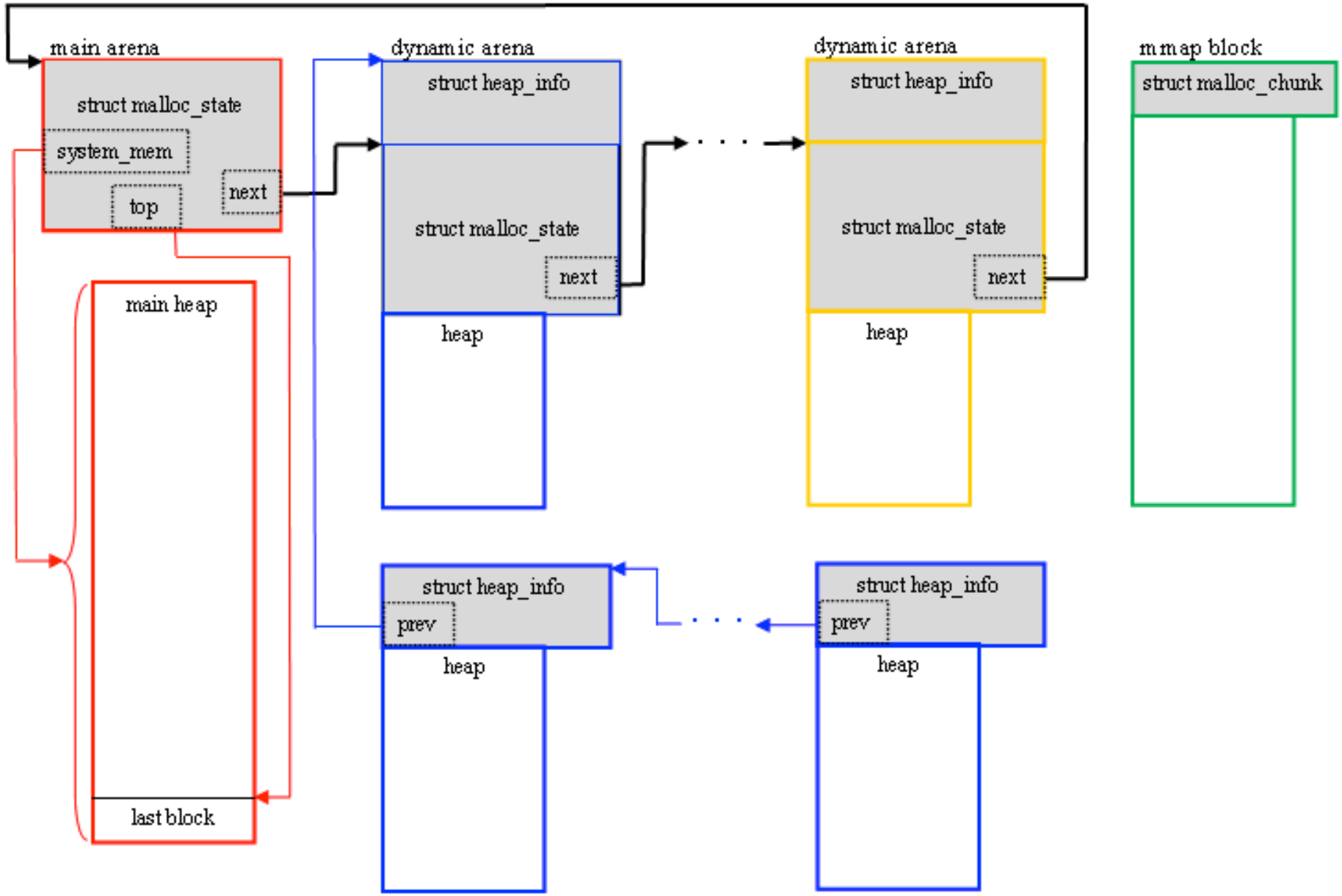
Terminologies

- Chunk: A small range of memory that can be allocated (owned by the application), freed (owned by glibc), or combined with adjacent chunks into larger ranges. Note that a chunk is a wrapper around **the block of memory that is given to the application**. Each chunk exists in one heap and belongs to one arena.

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

Arenas and Heaps

- To efficiently handle multi-threaded applications, glibc's 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
  - These regions of memory are collectively called "arenas“
- main arena: The application's initial heap
  - A static variable in the malloc code
  - Next to link additional arenas
  - For exploit, this is an address in libc
    (i.e., if we leak this memory, we can break ASLR)



http://core-analyzer.sourceforge.net/index_files/Page335.html

malloc_state (glibc 2.31)

```
struct malloc_state
{
  /* Serialize access.  */
  __libc_lock_define (, mutex);
  /* Flags (formerly in max_fast).  */
  int flags;

  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];
  /* Base of the topmost chunk -- not otherwise kept in a bin */
  mchunkptr top;
  /* The remainder from the most recent split of a small request */
  mchunkptr last_remainder;
  /* Normal bins packed as described above */
  mchunkptr bins[NBINS * 2 - 2];

  /* Bitmap of bins */
  unsigned int binmap[BINMAPSIZE];
```

malloc_state (glibc 2.31)

```
  /* Linked list */
  struct malloc_state *next;
  /* Linked list for free arenas.  Access to this field is serialized
     by free_list_lock in arena.c.  */
  struct malloc_state *next_free;
  /* Number of threads attached to this arena.  0 if the arena is on
     the free list.  Access to this field is serialized by
     free_list_lock in arena.c.  */

  INTERNAL_SIZE_T attached_threads;
  /* Memory allocated from the system in this arena.  */
  INTERNAL_SIZE_T system_mem;
  INTERNAL_SIZE_T max_system_mem;
};

typedef struct malloc_state *mstate;
```

Chunks

- Glibc malloc divides a large region of memory (a "heap") into chunks of various sizes.
- Each chunk includes meta-data about how big it is (via a size field in the chunk header), and thus where the adjacent chunks are.
- When a chunk is in use by the application, the only data that's "remembered" 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, such that suitable chunks can quickly be found and re-used when needed.
- Also, the last word in a free'd chunk contains a copy of the chunk size (with the three LSBs set to zeros, vs the three LSBs of the size at the front of the chunk which are used for flags).

```
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, i
ncluding 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;
```

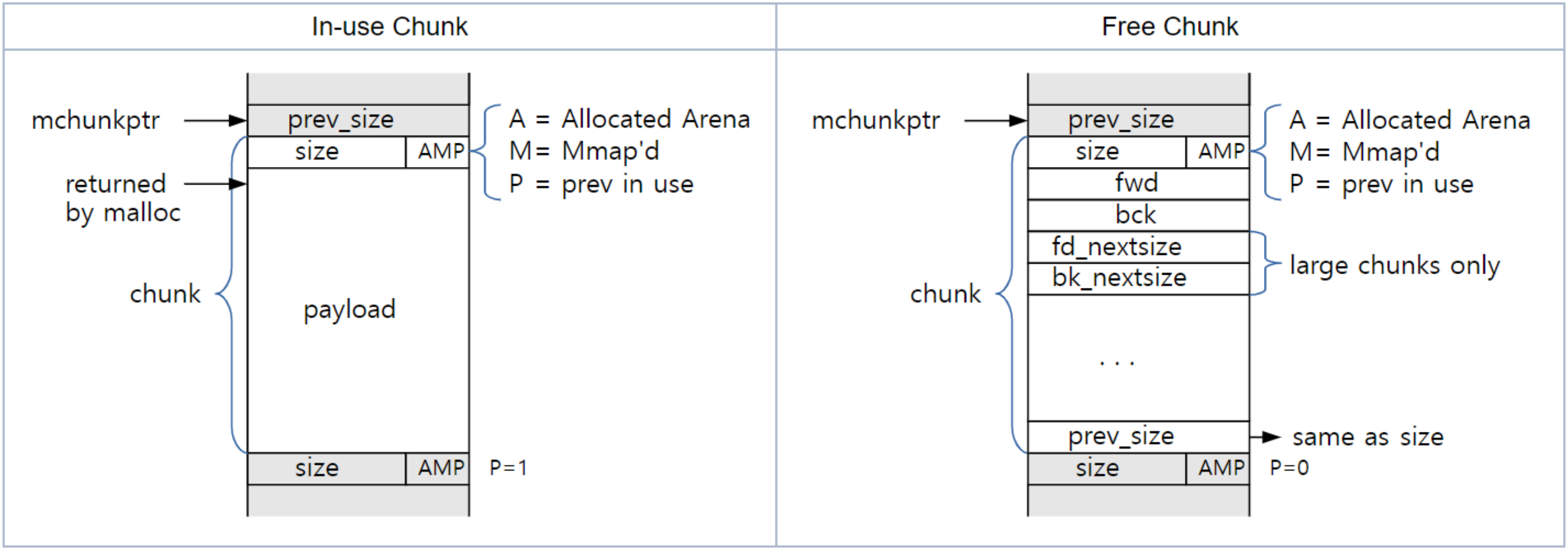
Chunk flags

- Since all chunks are multiples of 8 bytes, the 3 LSBs of the chunk size can be used for flags. These three flags are defined as follows:

- A (0x04): Allocated Arena - the main arena uses the application's heap. Other arenas use mmap'd heaps.

- M (0x02): MMap'd chunk - this chunk was allocated with a single call to mmap and is not part of a heap at all.

- P (0x01): Previous chunk is in use - if set, the previous chunk is still being used by the application, and thus the prev_size field is invalid.


In-use Chunk
mchunkptr
prev_size
size
AMP
returned
by malloc
chunk
payload
size
AMP
P=1
A = Allocated Arena
M= Mmap'd
P = prev in use
Free Chunk
mchunkptr
prev_size
size
AMP
fwd
bck
fd_nextsize
bk_nextsize
chunk
large chunks only
. . .
prev_size
same as size
size
AMP
P=0
A = Allocated Arena
M= Mmap'd
P = prev in use

Bins

- Within each arena, chunks are either in use by the application or they're free (available).
- 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. The lists, called "bins"

Types of bins

- **Fast:** Small chunks are stored in size-specific bins. Chunks added to a fast bin ("fastbin") are not combined with adjacent chunks. Fastbin chunks are stored in an array of singly-linked lists, since they're all the same size and chunks in the middle of the list need never be accessed.

- **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.

Types of bins

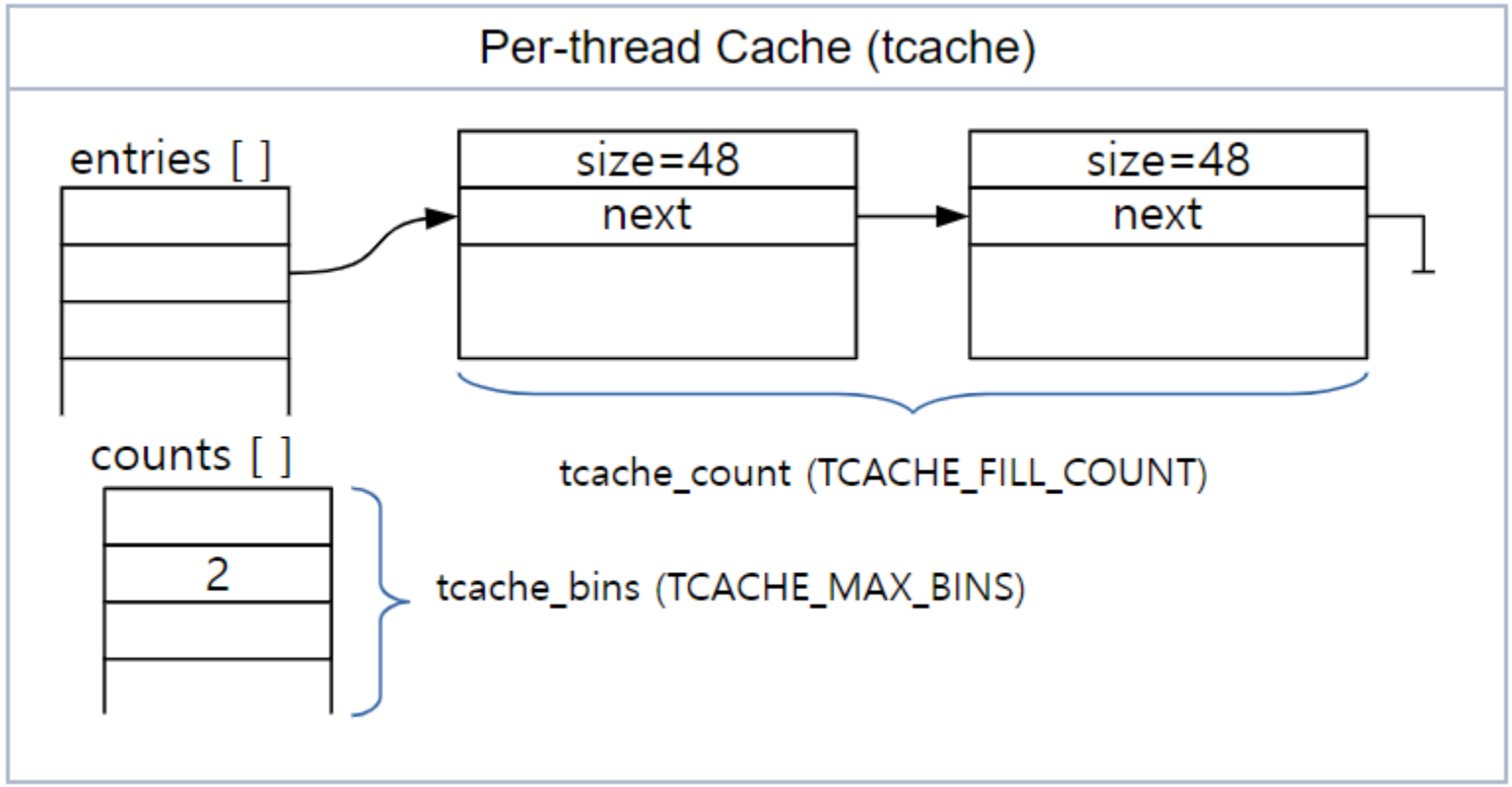
- **Small**: The normal bins are divided into "small" bins, where each chunk is the **same size**. When a chunk is added to these bins, they're first combined with adjacent chunks to **"coalesce"** them into larger chunks. Small and large chunks are **doubly-linked** so that chunks may be removed from the middle.

- **Large**: A chunk is "large" if its bin may contain more than one size. For small bins, you can pick the first chunk and just use it. For large bins, you have to find the "best" chunk, and possibly split it into two chunks (one the size you need, and one for the remainder).

Thread cache (tcache)

- For NUMA architectures, coordinate thread locality, sort threads by core, etc.
- Each thread has a per-thread cache (called the tcache) containing a small collection of chunks which can be accessed without needing to lock an arena.
- These chunks are stored as an array of singly-linked lists
- Each bin contains one size chunk, so the array is indexed (indirectly) by chunk size.
- Unlike fastbins, the tcache is limited in how many chunks are allowed in each bin (tcache_count).
- If the tcache bin is empty for a given requested size, the fallback is to use the normal malloc routines i.e. locking the thread's arena and working from there.

Thread cache (tcache)



Malloc algorithm

- If there is a suitable (exact match only) chunk in the tcache, it is returned to the caller.
- If the request is large enough, mmap() is used to request memory directly from the operating system.
- If the appropriate fastbin has a chunk in it, use that.
- If the appropriate smallbin has a chunk in it, use that.
- … (large)
- Use chunks in unsorted bin (if not, move them to regular bins)
- Split off part of the “top” chunk, possibly enlarging "top" beforehand.

Free algorithm

- If there is room in the tcache, store the chunk there and return.
- If the chunk is small enough, place it in the appropriate fastbin.
- If the chunk was mmap'd, munmap it.
- See if this chunk is adjacent to another free chunk and coalesce if it is.
- Place the chunk in the unsorted list, unless it's now the "top" chunk.
- …

Platform-specific Thresholds and Constants

Parameter	32 bit	i386	64 bit
MALLOC_ALIGNMENT	8	16	16
MIN_CHUNK_SIZE	16	16	32
MAX_FAST_SIZE	80	80	160
MAX_TCACHE_SIZE	516	1,020	1,032
MIN_LARGE_SIZE	512	1,008	1,024
DEFAULT_MMAP_THRESHOLD	131,072	131,072	131,072
DEFAULT_MMAP_THRESHOLD_MAX	524,288	524,288	33,554,432
HEAP_MIN_SIZE	32,768	32,768	32,768
HEAP_MAX_SIZE	1,048,576	1,048,576	67,108,864

Heap vulnerabilities

- Overflow: Writing beyond an object boundary
- Write-after-free: Reusing a freed object
- Invalid free: Freeing an invalid pointer
- Double free: Freeing a reclaimed object

→ Application- or allocator-specific exploitation

Heap overflow

- ptmalloc allocates memory linearly.
- Thus, it would be possible to overflow other object (or even other field in the same object).

- Unlike stack, a heap object has no universal data for hijacking control flow (e.g., return address). Thus, we need to use a other fields for getting control (e.g., data or code pointers)

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

typedef struct {
  char buf[100];
  void (*fp)();
} Packet;

int main() {
  Packet* p1 = calloc(1, sizeof(Packet));
  Packet* p2 = calloc(1, sizeof(Packet));
  read(0, p1->buf, 0x100);

  if (p2->fp != NULL)
    p2->fp();
}
```

```
pwndbg> r <<< $(python -c'print"A"*0x100')
pwndbg> x/i $pc
=> 0x5555555546e8 <main+94>:    call   rdx
pwndbg> x/gx $rdx
0x4141414141414141:     Cannot access memory at address 0x4141414141414141
```

Use-after-Free (UaF)

- Referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code.

- ptmalloc2 makes this exploit easier due to its first-fit strategy
  - If you free a certain object and allocate other one with the same size, the old object is returned for the new request.

Example

```c
#include <stdio.h>
#include <stdlib.h>

struct unicorn_counter { int num; };

int main() {
    struct unicorn_counter* p_unicorn_counter;
    int* run_calc = malloc(sizeof(int));
    *run_calc = 0;
    free(run_calc);
    p_unicorn_counter = malloc(sizeof(struct unicorn_counter));
    p_unicorn_counter->num = 42;
    if (*run_calc) execl("/bin/sh", 0);
}
```

https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html

Double free

- Freeing a resource that is already freed.
- We typically exploit this by changing double free into use-after-free

```
int main(int argc, char **argv) {
  Packet *p1 = malloc(sizeof(Packet));
  free(p1);

  Packet *p2 = malloc(sizeof(Packet));
  free(p1); // Double free

  // using p2 => use-after-free
}
```

Reference

- https://heap-exploitation.dhavalkapil.com/
- https://sourceware.org/glibc/wiki/MallocInternals