Debugging the heap
@morbith-dqtz at r2con 2018

Debugging the heap

- Introducing myself
- Introducing the heap
- Meeting the heap
- Dating with the heap
- Exploiting the heap

Introducing myself

- VoIP professional
- Passionate of learning
- Insane curious
- First time in an event such that, so please … be patient with me :P

Introducing the heap

- What is the heap memory ?

  - Dynamic memory allocated and released by a program
  - Intends to limit kernel interactions
  - Many implementations to manage heap memory

- What composes the heap ?

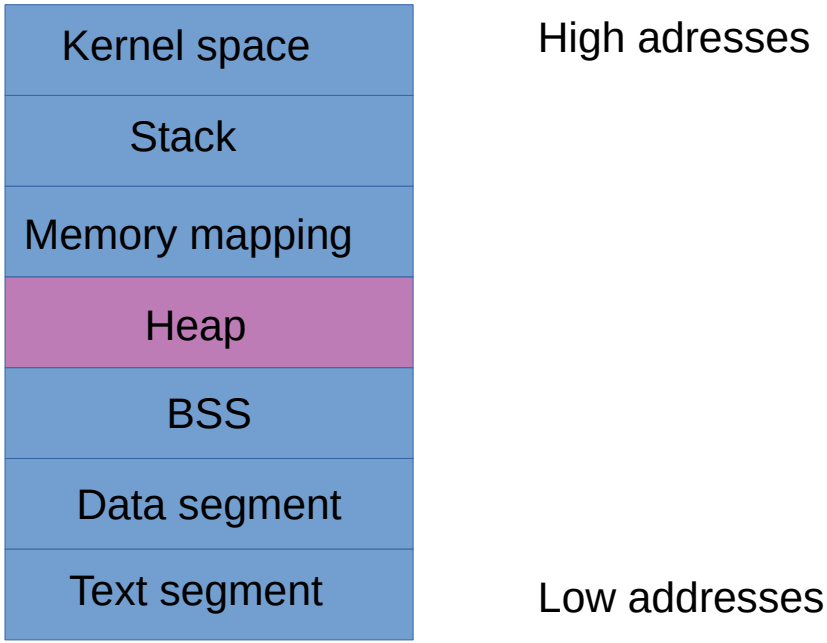
  - Main arena
  - Mmaped arenas
  - Heap section
  - Heap chunk

Introducing the heap

- Where is the heap ?

  After the first malloc()

  the memory takes that form



Introducing the heap

- There are many implementations

dlmalloc, jemalloc, ptmalloc, OpenBSD's malloc, Hoard's malloc, tcmalloc and more….

All of them have their own properties but shares many of the core concepts.

glibc is the most common adopted standard libc and it's a derivative of ptmalloc.

Because of that extensive use is out debug target.

Introducing the heap

Radare2 is awesome and have a plenty of functions to analyze the heap

```
[0x7f56eeb61295]> dmh?
|Usage:  dmh # Memory map heap
| dmh                                   List chunks in heap segment
| dmh [malloc_state]                    List heap chunks of a particular arena
| dmha                                  List all malloc_state instances in application
| dmhb                                  Display all parsed Double linked list of main_arena's bins instance
| dmhb [bin_num|bin_num:malloc_state]   Display parsed double linked list of bins instance from a particular arena
| dmhbg [bin_num]                       Display double linked list graph of main_arena's bin [Under developemnt]
| dmhc @[chunk_addr]                    Display malloc_chunk struct for a given malloc chunk
| dmhf                                  Display all parsed fastbins of main_arena's fastbinY instance
| dmhf [fastbin_num|fastbin_num:malloc_state]  Display parsed single linked list in fastbinY instance from a particular arena
| dmhg                                  Display heap graph of heap segment
| dmhg [malloc_state]                   Display heap graph of a particular arena
| dmhi @[malloc_state]                  Display heap_info structure/structures for a given arena
| dmhm                                  List all elements of struct malloc_state of main thread (main_arena)
| dmhm [malloc_state]                   List all malloc_state instance of a particular arena
| dmht                                  Display all parsed thead cache bins of main_arena's tcache instance
| dmh?                                  Show map heap help
```

But … let’s figure out how debug without them!

Meeting the heap

- Main Arena

- Main malloc_state structure, its placed in glibc data segment as a global variable

- Contains references to one “next heap” or itself if there is none. Its referenced by “next” by last heap allocated

- Contains linked lists of chunks that are "free".

Threads assigned to each arena will allocate memory from that arena's reserved lists.

Meeting the heap

Let’s find the main_arena !

Load debug version of the library in memory

```
[0x7f78adfc3fc0]> oba 0x0 /usr/lib/debug/lib64/libc-2.27.so.debug
```

Search the main_arena symbol inside

```
[0x7f78adfc3fc0]> is~main_arena
943 0x001cac00 0x003cac00  LOCAL    OBJ 2200 main_arena
961 0x001cc8f8 0x003cc8f8  LOCAL    OBJ    8 dumped_main_arena_start
962 0x001cc8f0 0x003cc8f0  LOCAL    OBJ    8 dumped_main_arena_end
```

Its symbol is at the offset 0x3cac00 from the top of the libc

Meeting the heap

Let’s find the main_arena !

Listing opened file descriptors

```
[0x7f78adfc3fc0]> ob
  1  x86-64 at:0x00000000 sz:15072 fd:3 /tmp/heap_play
  3  x86-64 at:0x00000000 sz:4791248 fd:5 /usr/lib/debug/lib64/libc-2.27.so.debug
```

Returning to debugging process file descriptor

```
[0x7f78adfc3fc0]> ob 3
```

Continue with the debug until main is reached

```
[0x5559286b38c0]> dcu main
Continue until 0x5559286b39d1 using 1 bpsize
hit breakpoint at: 5559286b39d1
```

Meeting the heap

Let’s find the main_arena !

Check where libc is loaded

```
[0x5559286b39d1]> dm~libc
0x00007f78adbf3000 - 0x00007f78addb9000 - usr   1.8M s r-x /lib64/libc-2.27.so /lib64/libc-2.27.so
0x00007f78addb9000 - 0x00007f78adfb9000 - usr     2M s --- /lib64/libc-2.27.so /lib64/libc-2.27.so
0x00007f78adfb9000 - 0x00007f78adfbd000 - usr    16K s r-- /lib64/libc-2.27.so /lib64/libc-2.27.so
0x00007f78adfbd000 - 0x00007f78adfbf000 - usr     8K s rw- /lib64/libc-2.27.so /lib64/libc-2.27.so
```

Pointing the main_arena

```
[0x5559286b39d1]> ?v 0x00007f78adbf3000+0x003cac00
0x7f78adfbdc00
```

Meeting the heap

Let’s find the main_arena !

This is how an uninitialized main_arena looks like

```
[0x5559286b39d1]> pxq @0x7f78adfbdc00
0x7f78adfbdc00  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdc10  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdc20  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdc30  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdc40  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdc50  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdc60  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdc70  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdc80  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdc90  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdca0  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdcb0  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdcc0  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdcd0  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdce0  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdcf0  0x0000000000000000  0x0000000000000000  ................
```

Meeting the heap

Let’s find the main_arena !

That is distribution and architecture dependent, so we can not relay on that

Gentoo and Kali 2018 rolling : 0x0
On debian 9.5 x86_64 SID : 0x192000
On debian 9.5 i686 : 0x1bb000

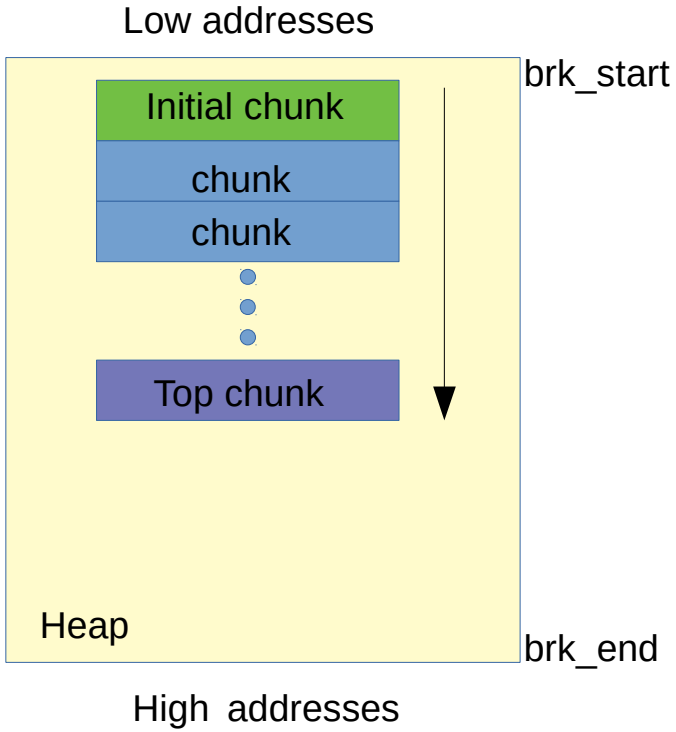
e dbg.glibc.ma_offset = 0x0

```
[0x7f93f80cea45]> dm~libc
0x00007f93f7cee000 - 0x00007f93f7eb2000 - usr   1.8M s r-x /lib64/libc-2.27.so /lib64/libc-2.27.so
0x00007f93f7eb2000 - 0x00007f93f80b2000 - usr     2M s --- /lib64/libc-2.27.so /lib64/libc-2.27.so
0x00007f93f80b2000 - 0x00007f93f80b6000 - usr    16K s r-- /lib64/libc-2.27.so /lib64/libc-2.27.so
0x00007f93f80b6000 - 0x00007f93f80b8000 - usr     8K s rw- /lib64/libc-2.27.so /lib64/libc-2.27.so
```

0x00007f93f80b600 + ~c00

Meeting the heap

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



Meeting the heap

Let’s find the heap !

- As previously seen, main_arena stores pointers to the heaps

- Until the first malloc isn’t realized, main_arena remains uninitialized.

- We will use out test program to create a new heap
  (requesting some space via malloc())

- At the end of this malloc() a heap and a chunk will be created.

Meeting the heap

Let’s find the heap !

```
[0x5559286b39d1]> dc
*.... Heap playground ....*

1.Alloc chunk
2.Free chunk
3.Write to chunk
4.Print chunk
5.Print ptrs array
6.Write to tmp
7.Print tmp
>1
How many space ?: 100
Allocated chunk at 0x555928e84260
        idx 0, addr: 0x555928e84260
```

We created a 0x100 bytes chunk

https://github.com/morbith-dqtz/r2con2018/blob/master/source_examples/heap_play_stdio.c

Meeting the heap

Printing main_arena now is revealing some interesting data.

What it should be ? →

```
[0x7f78adce7295]> pxq @0x7f78adfbdc00
0x7f78adfbdc00  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdc10  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdc20  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdc30  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdc40  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdc50  0x0000000000000000  0x0000000000000000  ................
0x7f78adfbdc60  0x0000555928e84360  0x0000000000000000  `C.(YU..........
0x7f78adfbdc70  0x00007f78adfbdc60  0x00007f78adfbdc60  `...x...`...x...
0x7f78adfbdc80  0x00007f78adfbdc70  0x00007f78adfbdc70  p...x...p...x...
0x7f78adfbdc90  0x00007f78adfbdc80  0x00007f78adfbdc80  ....x.......x...
0x7f78adfbdca0  0x00007f78adfbdc90  0x00007f78adfbdc90  ....x.......x...
0x7f78adfbdcb0  0x00007f78adfbdca0  0x00007f78adfbdca0  ....x.......x...
0x7f78adfbdcc0  0x00007f78adfbdcb0  0x00007f78adfbdcb0  ....x.......x...
0x7f78adfbdcd0  0x00007f78adfbdcc0  0x00007f78adfbdcc0  ....x.......x...
0x7f78adfbdce0  0x00007f78adfbdcd0  0x00007f78adfbdcd0  ....x.......x...
0x7f78adfbdcf0  0x00007f78adfbdce0  0x00007f78adfbdce0  ....x.......x...
[0x7f78adce7295]>
```

Meeting the heap

Part of the struct that stores the malloc state of the heap, extracted from glibc

```
/* Serialize access.  */
 __libc_lock_define (, mutex);

 /* Flags (formerly in max_fast).  */
 int flags;

#only glibc > 2.25 :
 /* Set if the fastbin chunks contain recently inserted free blocks.  */
 /* Note this is a bool but not all targets support atomics on booleans.  */
 int have_fastchunks;
#end only

 /* Fastbins */
 mfastbinptr fastbinsY[NFASTBINS];

 /* Base of the topmost chunk -- not otherwise kept in a bin */
 mchunkptr top;
```

Here we have the ptr to top chunk

Meeting the heap

Let’s do some math

int to serialize acces	(4 bytes 32 / 64)
int to flags	(4 bytes 32 / 64)
#IF glibc with tcache :	
int to check is there are fast chunks	(4 bytes 32 / 64)
#ENDIF	
array of [NFASTBINS] pointers	(4 bytes * 11 on 32 bits / 8 bytes * 10 on 64 bits)

Total 32 bits on a glibc > 2.25: 14 * 4 bytes
Total 64 bits on a glibc > 2.25: 4*4 + 0x8*10

Each pointer on 64 bits occupy 8 bytes, because of this we add an additional 4 bytes
They came from the 3th int declaration, this 3 ints will occupy 16 bytes inside a structure, not 12 bytes.

Meeting the heap

Let’s explain the 16 bytes occupied by those 3 ints

```
typedef struct test {
        int a;
        int b;
        int c;
        char *d;
}test;
```

If we declare a struct like the above and initialize the ints to 1, the memory take this from

0x0000000100000001 0xZZZZZZZZ00000001

Each int occupies 4 bytes, but it we declare a pointer after them, we need 8 bytes to store their full contents, leaving ZZZZZZZZ (4 bytes) unused to ensure the alignment and accessibility.

0x0000000100000001 0xZZZZZZZZ00000001
0x0000000000000000

Meeting the heap

According with our calculations the pointer to top chunk is at

```
[0x7f78adce7295]> pxq 0x8 @0x7f78adfbdc00+0x4*4+0x8*10
0x7f78adfbdc60  0x0000555928e84360                      `C.(YU..
```

Let's inspect it

```
[0x7f78adce7295]> pxq 0x10 @0x0000555928e84360
0x555928e84360  0x0000000000000000  0x0000000000020ca1  ................
```

Top Chunk :

Special chunk that stores the remaining free space in the heap
So heap end (or brk_end) is at :

```
[0x7f78adce7295]> ?v 0x555928e84360+0x0000000000020ca0
0x555928ea5000
```

That bite is a flag,
meaning less to calculate offsets

Meeting the heap

In order to ease this kind of offset calculations, we are going to introduce the pf (print format) command from radare2.

Given a know structure :

```
typedef struct car {
        char *model;
        char *plate;
        char *owner_name;
        int penalty;
}car;
```

We can just define it as format characters, this way we can ask for a field easily

Meeting the heap

```
[0x560d03f7e66d]> pf.car SSSSi brand model plate owner penalty
[0x560d03f7e66d]> pf.car @rax
   brand : 0x560d04a57260 = 0x560d04a57260 -> 0x560d03f7e714 grate one
   model : 0x560d04a57268 = 0x560d04a57268 -> 0x560d03f7e71e the best
   plate : 0x560d04a57270 = 0x560d04a57270 -> 0x560d03f7e727 AAAAAAAAA
   owner : 0x560d04a57278 = 0x560d04a57278 -> 0x560d03f7e731 Arthur
 penalty : 0x560d04a57280 = 100000
[0x560d03f7e66d]> pf.car.owner @rax
   owner : 0x560d04a57278 = 0x560d04a57278 -> 0x560d03f7e731 Arthur
[0x560d03f7e66d]>
```

Glibc format types could be found here :

https://github.com/morbith-dqtz/r2con2018/blob/master/pf_formats/glib_heap

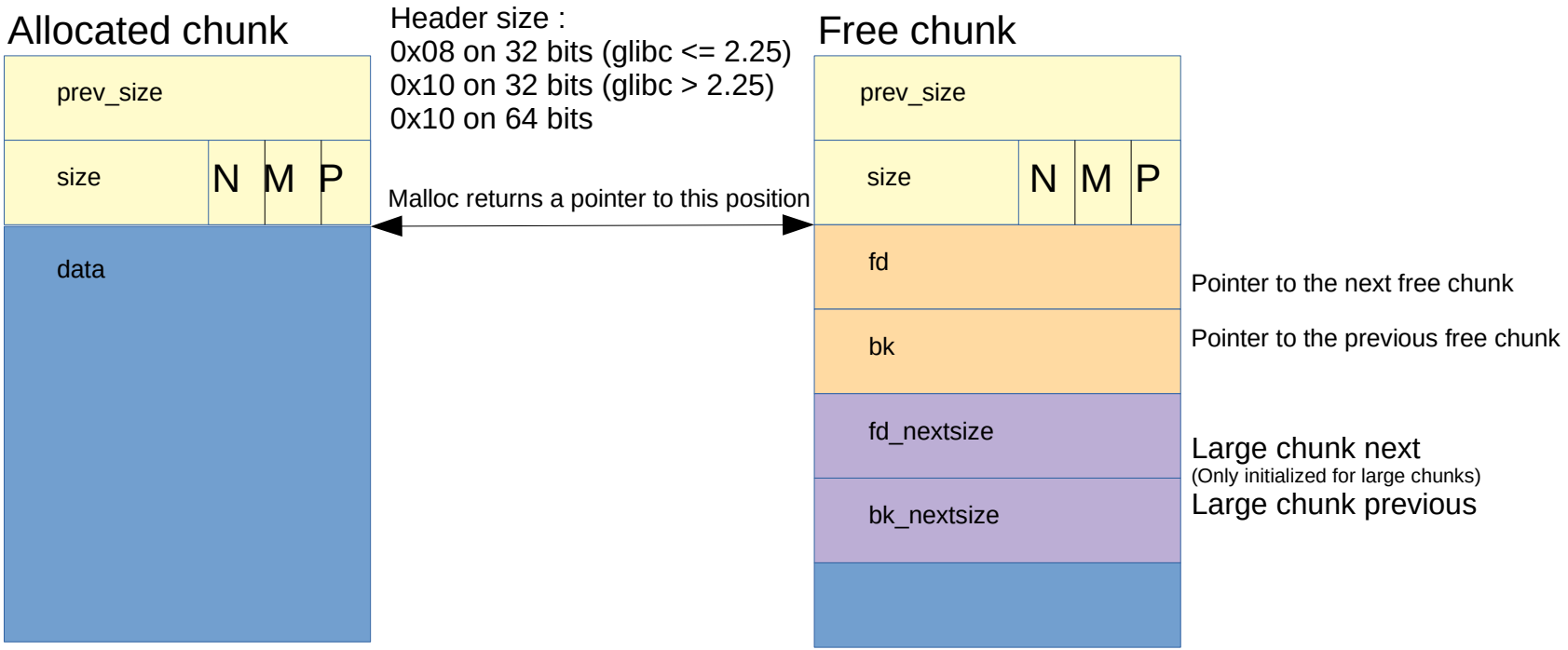
Meeting the heap

- 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.

```
[0x7f384a6dc295]> pf.r_malloc_chunk_64 @0x563abcd79250
   prev_size : 0x563abcd79250 = (qword)0x0000000000000000
        size : 0x563abcd79258 = (qword)0x0000000000000111
          fd : 0x563abcd79260 = (qword)0x0000000000000000
          bk : 0x563abcd79268 = (qword)0x0000000000000000
 fd_nextsize : 0x563abcd79270 = (qword)0x0000000000000000
 bk_nextsize : 0x563abcd79278 = (qword)0x0000000000000000
```

Meeting the heap



3 LSB of size field represent chunk flags, these are:

- PREV_INUSE (P) : bit set when previous chunk in heap is allocated.
- IS_MMAPPED (M) : bit set when chunk is being mmap'd.
- NON_MAIN_ARENA (N) : bit set when chunk does not belong to heap segment

Meeting the heap

Top Chunk is at
0x555928e84360.

We subtract
0x100 of the data request
0x010 of the header

Here we are our first chunk

```
[0x7f78adce7295]> pxq 0x110 @0x555928e84360-0x100-0x10
0x555928e84250  0x0000000000000000  0x0000000000000111  ................
0x555928e84260  0x0000000000000000  0x0000000000000000  ................
0x555928e84270  0x0000000000000000  0x0000000000000000  ................
0x555928e84280  0x0000000000000000  0x0000000000000000  ................
0x555928e84290  0x0000000000000000  0x0000000000000000  ................
0x555928e842a0  0x0000000000000000  0x0000000000000000  ................
0x555928e842b0  0x0000000000000000  0x0000000000000000  ................
0x555928e842c0  0x0000000000000000  0x0000000000000000  ................
0x555928e842d0  0x0000000000000000  0x0000000000000000  ................
0x555928e842e0  0x0000000000000000  0x0000000000000000  ................
0x555928e842f0  0x0000000000000000  0x0000000000000000  ................
0x555928e84300  0x0000000000000000  0x0000000000000000  ................
0x555928e84310  0x0000000000000000  0x0000000000000000  ................
0x555928e84320  0x0000000000000000  0x0000000000000000  ................
0x555928e84330  0x0000000000000000  0x0000000000000000  ................
0x555928e84340  0x0000000000000000  0x0000000000000000  ................
0x555928e84350  0x0000000000000000  0x0000000000000000  ................
```

Meeting the heap

We are running a glibc-2.27, so we have to keep in mind tcache

As seen before, it adds a couple of ints in main_state structure, but also include this new structure.

```
typedef struct tcache_perthread_struct
{
  char counts[TCACHE_MAX_BINS];
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;
```

- TCACHE_MAX_BINS is defined to 64
- Chars occupy 1 bytes on 32 and 64 bites
- entries are pointers, 4 bytes on 32 and 8 bytes in 64 bits
- tcache_perthread structure is similar to a chunk, initialized by _int_malloc() it so has a chunk header
- The whole tcache structure occupy :
    32 bits : 0x1*64 + 0x4*64 + 0x10 = 0x150
    64 bits : 0x1*64 + 0x8*64 + 0x10 = 0x250

Meeting the heap

First Chunk is at
0x555928e84250.

We subtract
0x250 of the tcache

Here we are our heap start

```
[0x7f78adce7295]> ?v 0x555928e84250-0x250
0x555928e84000
[0x7f78adce7295]> pxq @0x555928e84250-0x250
0x555928e84000  0x0000000000000000  0x0000000000000251  ........Q.......
0x555928e84010  0x0000000000000000  0x0000000000000000  ................
0x555928e84020  0x0000000000000000  0x0000000000000000  ................
0x555928e84030  0x0000000000000000  0x0000000000000000  ................
0x555928e84040  0x0000000000000000  0x0000000000000000  ................
0x555928e84050  0x0000000000000000  0x0000000000000000  ................
0x555928e84060  0x0000000000000000  0x0000000000000000  ................
0x555928e84070  0x0000000000000000  0x0000000000000000  ................
0x555928e84080  0x0000000000000000  0x0000000000000000  ................
0x555928e84090  0x0000000000000000  0x0000000000000000  ................
0x555928e840a0  0x0000000000000000  0x0000000000000000  ................
0x555928e840b0  0x0000000000000000  0x0000000000000000  ................
0x555928e840c0  0x0000000000000000  0x0000000000000000  ................
0x555928e840d0  0x0000000000000000  0x0000000000000000  ................
0x555928e840e0  0x0000000000000000  0x0000000000000000  ................
0x555928e840f0  0x0000000000000000  0x0000000000000000  ................
[0x7f78adce7295]>
```

Meeting the heap

Our trip to heap start is unnecessary
It was used as a form to illustrate the basic structures of the heap memory: main_arena, heap and chunk

The heap addresses are populated by the running program via procfs

```
% cat /proc/20716/maps | grep “\[heap”
555928e84000-555928ea5000 rw-p 00000000 00:00 0                  [heap]
```

Radare2 offers us the command dm that can retrieve the same information

```
[0x7f78adce7295]> dm~[heap
0x0000555928e84000 - 0x0000555928ea5000 - usr   132K s rw- [heap] [heap]
```

Meeting the heap

Since we have defined all the heap structures as format types, once we know where is located, for example, the heap, we can just ask for what we need

```
[0x7fcfd88ac295]> dm=~[heap
map 324K - 0x0000555a79a5b000 |-----------------------------------------------
[0x7fcfd88ac295]> pf.r_tcache_perthread_struct_header.size @0x0000555a79a5b000
      size : 0x555a79a5b008 = (qword)0x0000000000000251
[0x7fcfd88ac295]> pf.r_malloc_chunk_64 @0x0000555a79a5b000+0x0000000000000250
   prev_size : 0x555a79a5b250 = (qword)0x0000000000000000
        size : 0x555a79a5b258 = (qword)0x0000000000000111
          fd : 0x555a79a5b260 = (qword)0x0000000000000000
          bk : 0x555a79a5b268 = (qword)0x0000000000000000
 fd_nextsize : 0x555a79a5b270 = (qword)0x0000000000000000
 bk_nextsize : 0x555a79a5b278 = (qword)0x0000000000000000
[0x7fcfd88ac295]>
```

Dating the heap

Let’s start debugging a very simple program

```c
#include <stdlib.h>

void main (void){
        char *a, *b, *c, *d, *e, *f;
        a = malloc (0x16);
        b = malloc (16);
        c = malloc (32);
        d = malloc (0);
        e = malloc (80);
        f = malloc (0x100);
}
```

```
r_config_set: variable 'asm.cmtright' not found
Process with PID 6703 started...
= attach 6703 6703
bin.baddr 0x556ecbdef000
Using 0x556ecbdef000
asm.bits 64
 -- This is fine.
[0x7f8ad6954fc0]> dcu main
Continue until 0x556ecbdef62a using 1 bpsize
hit breakpoint at: 556ecbdef62a
[0x556ecbdef62a]> Vpp
```

Dating the heap

```
;-- main:
0x556ecbdef62a      55              push rbp
0x556ecbdef62b      4889e5          rbp = rsp
0x556ecbdef62e      4883ec30        rsp -= 0x30
0x556ecbdef632      bf16000000      edi = 0x16                  ; 22
0x556ecbdef637      e8c4fefffff     sym.imp.malloc ()           ;[1]
;-- rip:
0x556ecbdef63c      488945d0        qword [rbp - 0x30] = rax
0x556ecbdef640      bf10000000      edi = 0x10                  ; 16
0x556ecbdef645      e8b6fefffff     sym.imp.malloc ()           ;[1]
0x556ecbdef64a      488945d8        qword [rbp - 0x28] = rax
0x556ecbdef64e      bf20000000      edi = 0x20                  ; 32
0x556ecbdef653      e8a8fefffff     sym.imp.malloc ()           ;[1]
0x556ecbdef658      488945e0        qword [rbp - 0x20] = rax
0x556ecbdef65c      bf00000000      edi = 0
0x556ecbdef661      e89afefffff     sym.imp.malloc ()           ;[1]
0x556ecbdef666      488945e8        qword [rbp - 0x18] = rax
0x556ecbdef66a      bf50000000      edi = 0x50                  ; 'P' ; 80
0x556ecbdef66f      e88cfefffff     sym.imp.malloc ()           ;[1]
0x556ecbdef674      488945f0        qword [rbp - 0x10] = rax
0x556ecbdef678      bf00010000      edi = 0x100                 ; 256
0x556ecbdef67d      e87efefffff     sym.imp.malloc ()           ;[1]
0x556ecbdef682      488945f8        qword [rbp - 8] = rax
0x556ecbdef686      90
0x556ecbdef687      c9              leave
0x556ecbdef688      c3              return
```

```
[0x556ecbdef62a]> dr
rax = 0x556ecc19f260
rbx = 0x00000000
rcx = 0x556ecc19f260
rdx = 0x556ecc19f260
r8 = 0x00000003
r9 = 0x0000000c
r10 = 0xfffffffffffff000
r11 = 0x556ecc19f010
r12 = 0x556ecbdef520
r13 = 0x7ffc61df4270
r14 = 0x00000000
r15 = 0x00000000
rsi = 0x00000000
rdi = 0x00000000
rsp = 0x7ffc61df4160
rbp = 0x7ffc61df4190
rip = 0x556ecbdef63c
rflags = 0x00000206
orax = 0xffffffffffffffff
[0x556ecbdef62a]>
```

Dating the heap

```
[0x556ecbdef62a]> pxq 0x10+0x16 @rax-0x10
0x556ecc19f250  0x0000000000000000  0x0000000000000021  ........!.......
0x556ecc19f260  0x0000000000000000  0x0000000000000000  ................
```

malloc(0x16)

```
[0x556ecbdef62a]> pxq 0x10+0x10 @rax-0x10
0x556ecc19f270  0x0000000000000000  0x0000000000000021  ........!.......
0x556ecc19f280  0x0000000000000000  0x0000000000000000  ................
```

malloc(16)

```
[0x556ecbdef62a]> pxq 0x10+0x20 @rax-0x10
0x556ecc19f290  0x0000000000000000  0x0000000000000031  ........1.......
0x556ecc19f2a0  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f2b0  0x0000000000000000  0x0000000000000000  ................
```

malloc(32)

```
[0x556ecbdef62a]> pxq 0x10+0x10 @rax-0x10
0x556ecc19f2c0  0x0000000000000000  0x0000000000000021  ........!.......
0x556ecc19f2d0  0x0000000000000000  0x0000000000000000  ................
```

malloc(0)

Dating the heap

```
[0x556ecbdef62a]> pxq 0x10+0x16 @rax-0x10
0x556ecc19f250  0x0000000000000000  0x0000000000000021  ........!.......
0x556ecc19f260  0x0000000000000000  0x0000000000000000  ................
```

0x16 + 0x10 = 0x26
0x21 ? What ?
prev_size my friend

```
[0x556ecbdef62a]> pxq 0x10+0x10 @rax-0x10
0x556ecc19f270  0x0000000000000000  0x0000000000000021  ........!.......
0x556ecc19f280  0x0000000000000000  0x0000000000000000  ................
```

16 → 0x10 + 0x10

```
[0x556ecbdef62a]> pxq 0x10+0x20 @rax-0x10
0x556ecc19f290  0x0000000000000000  0x0000000000000031  ........1.......
0x556ecc19f2a0  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f2b0  0x0000000000000000  0x0000000000000000  ................
```

32 → 0x20 + 0x10

```
[0x556ecbdef62a]> pxq 0x10+0x10 @rax-0x10
0x556ecc19f2c0  0x0000000000000000  0x0000000000000021  ........!.......
0x556ecc19f2d0  0x0000000000000000  0x0000000000000000  ................
```

0 → 0 + 0x10
But the smallest space
for a chunk takes (2 ptrs)

Dating the heap

```
[0x556ecbdef62b]> pxq 0x10+0x50 @rax-0x10
0x556ecc19f2e0  0x0000000000000000  0x0000000000000061  ........a.......
0x556ecc19f2f0  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f300  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f310  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f320  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f330  0x0000000000000000  0x0000000000000000  ................
```

malloc(80)
80 → 0x50 + 0x10

Dating the heap

```
[0x556ecbdef678]> pxq 0x10+0x100 @rax-0x10
0x556ecc19f340  0x0000000000000000  0x0000000000000111  ................
0x556ecc19f350  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f360  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f370  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f380  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f390  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f3a0  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f3b0  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f3c0  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f3d0  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f3e0  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f3f0  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f400  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f410  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f420  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f430  0x0000000000000000  0x0000000000000000  ................
0x556ecc19f440  0x0000000000000000  0x0000000000000000  ................
```

malloc(0x100)
0x100 + 0x10

Dating the heap

Using the heap_play program, we allocate 5 chunks with an arbitrary size :
100 (0x64), 90 (0x5a) , 80 (0x50) , 70 (0x46) , 60 (0x3c)

We will fill them with some data and inspect each one.

```
[0x7f5cb1d7c295]> pxq 0x64+0x10 @0x55706a5e2a80-0x10
0x55706a5e2a70  0x0000000000000000  0x0000000000000071   ........q.......
0x55706a5e2a80  0x2065772065726548  0x72754f2021657261   Here we are! Our
0x55706a5e2a90  0x6320747372696620  0x746164206b6e7568    first chunk dat
0x55706a5e2aa0  0x616c702073692061  0x6572656820646563   a is placed here
0x55706a5e2ab0  0x0000000000000a2e  0x0000000000000000   ................
0x55706a5e2ac0  0x0000000000000000  0x0000000000000000   ................
0x55706a5e2ad0  0x0000000000000000  0x0000000000000000   ................
```

Dating the heap

```
[0x7f5cb1d7c295]> pxq 0x5a+0x10 @0x55706a5e2af0-0x10
0x55706a5e2ae0  0x0000000000000000  0x0000000000000071  ........q.......
0x55706a5e2af0  0x6320646e6f636553  0x746164206b6e7568  Second chunk dat
0x55706a5e2b00  0x7265682073692061  0x0000000000000a65  a is here.......
0x55706a5e2b10  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2b20  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2b30  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2b40  0x0000000000000000                      ........
```

```
[0x7f5cb1d7c295]> pxq 0x50+0x10 @0x55706a5e2b60-0x10
0x55706a5e2b50  0x0000000000000000  0x0000000000000061  ........a.......
0x55706a5e2b60  0x6863206472696854  0x61746164206b6e75  Third chunk data
0x55706a5e2b70  0x6572656820736920  0x000000000000000a   is here........
0x55706a5e2b80  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2b90  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2ba0  0x0000000000000000  0x0000000000000000  ................
```

Dating the heap

```
[0x7f5cb1d7c295]> pxq 0x46+0x10 @0x55706a5e2bc0-0x10
0x55706a5e2bb0  0x0000000000000000  0x0000000000000051  ........Q.......
0x55706a5e2bc0  0x6320687472756f46  0x746164206b6e7568  Fourth chunk dat
0x55706a5e2bd0  0x7265682073692061  0x0000000000000a65  a is here.......
0x55706a5e2be0  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2bf0  0x0000000000000000  0x0000000000000000  ................
```

```
[0x7f5cb1d7c295]> pxq 0x50+0x10 @0x55706a5e2c10-0x10
0x55706a5e2c00  0x0000000000000000  0x0000000000000051  ........Q.......
0x55706a5e2c10  0x6863206874666946  0x61746164206b6e75  Fifth chunk data
0x55706a5e2c20  0x6572656820736920  0x000000000000000a   is here.......
0x55706a5e2c30  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2c40  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2c50  0x0000000000000000  0x00000000000203b1  ................
```

Top Chunk

Dating the heap

```
[0x7f5cb1d7c295]> pxq  100 + 90 + 80 + 70 + 60 + 0x10*5 + 0x10 @0x55706a5e2a70
0x55706a5e2a70  0x0000000000000000  0x0000000000000071  ........q.......
0x55706a5e2a80  0x2065772065726548  0x72754f2021657261  Here we are! Our
0x55706a5e2a90  0x6320747372696620  0x746164206b6e7568   first chunk dat
0x55706a5e2aa0  0x616c702073692061  0x6572656820646563  a is placed here
0x55706a5e2ab0  0x0000000000000a2e  0x0000000000000000  ................
0x55706a5e2ac0  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2ad0  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2ae0  0x0000000000000000  0x0000000000000071  ........q.......
0x55706a5e2af0  0x6320646e6f636553  0x746164206b6e7568  Second chunk dat
0x55706a5e2b00  0x7265682073692061  0x0000000000000a65  a is here.......
0x55706a5e2b10  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2b20  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2b30  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2b40  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2b50  0x0000000000000000  0x0000000000000061  ........a.......
0x55706a5e2b60  0x6863206472696854  0x61746164206b6e75  Third chunk data
0x55706a5e2b70  0x6572656820736920  0x000000000000000a   is here........
0x55706a5e2b80  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2b90  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2ba0  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2bb0  0x0000000000000000  0x0000000000000051  ........Q.......
0x55706a5e2bc0  0x6320687472756f46  0x746164206b6e7568  Fourth chunk dat
0x55706a5e2bd0  0x7265682073692061  0x0000000000000a65  a is here.......
0x55706a5e2be0  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2bf0  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2c00  0x0000000000000000  0x0000000000000051  ........Q.......
0x55706a5e2c10  0x6863206874666946  0x61746164206b6e75  Fifth chunk data
0x55706a5e2c20  0x6572656820736920  0x000000000000000a   is here........
0x55706a5e2c30  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2c40  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2c50  0x0000000000000000  0x00000000000203b1  ................
```

Dating the heap

We are going to free second and forth but we didn’t see big changes in our chunks.

Only, the first memory register, corresponding the pointers to the data, are set to zero.

The reason is that we are debugging a tcache system, otherwise both chunks headers would have changed respective headers.

In our case, we need inspect how the tcache perthread struct works.

Dating the heap

```
[0x7f5cb1d7c295]> pxq  100 + 90 + 80 + 70 + 60 + 0x10*5 + 0x10 @0x55706a5e2a70
0x55706a5e2a70  0x0000000000000000  0x0000000000000071  ........q.......
0x55706a5e2a80  0x2065772065726548  0x72754f2021657261  Here we are! Our
0x55706a5e2a90  0x6320747372696620  0x746164206b6e7568   first chunk dat
0x55706a5e2aa0  0x616c702073692061  0x6572656820646563  a is placed here
0x55706a5e2ab0  0x0000000000000a2e  0x0000000000000000  ................
0x55706a5e2ac0  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2ad0  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2ae0  0x0000000000000000  0x0000000000000071  ........q.......
0x55706a5e2af0  0x0000000000000000  0x746164206b6e7568  ........hunk dat
0x55706a5e2b00  0x7265682073692061  0x0000000000000a65  a is here.......
0x55706a5e2b10  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2b20  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2b30  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2b40  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2b50  0x0000000000000000  0x0000000000000061  ........a.......
0x55706a5e2b60  0x6863206472696854  0x6174164206b6e75  Third chunk data
0x55706a5e2b70  0x6572656820736920  0x0000000000000a65   is here.......
0x55706a5e2b80  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2b90  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2ba0  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2bb0  0x0000000000000000  0x0000000000000051  ........Q.......
0x55706a5e2bc0  0x0000000000000000  0x746164206b6e7568  ........hunk dat
0x55706a5e2bd0  0x7265682073692061  0x0000000000000a65  a is here.......
0x55706a5e2be0  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2bf0  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2c00  0x0000000000000000  0x0000000000000051  ........Q.......
0x55706a5e2c10  0x6863206874666946  0x6174164206b6e75  Fifth chunk data
0x55706a5e2c20  0x6572656820736920  0x0000000000000a65   is here.......
0x55706a5e2c30  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2c40  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2c50  0x0000000000000000  0x00000000000203b1  ................
```

Dating the heap

```
[0x7f5cb1d7c295]> pf.r_tcache_perthread_struct_header @0x55706a5e2000
 prev_size : 0x55706a5e2000 = (qword)0x0000000000000000
      size : 0x55706a5e2008 = (qword)0x0000000000000251
    counts : 0x55706a5e2010 = [ 0x00, 0x00, 0x00, 0x01, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x
00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00 ]
   entries : 0x55706a5e2050 = (qword)[ 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x000055706a5e2bc0, 0x0000000000000000, 0x000055706a5e2af0, 0x0000000000000000, 0x000000000
0000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0
x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x000000000
0000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0
x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x000000000
0000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0
x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000 ]
```

The entries in this structure now have two addresses, pointing to their respective “data” pointers.

Each entry represents a chunk size. Are spaced by 8 bytes in 32 bits and 16 in 64 bits
That’s means, if we allocate 70 and 60 their sizes will match. Also when freed will share tcache entry list.

If we free some chunks with the same size, we only see on entry in this list, but we found the “zeroed” pointer of the just freed one, filled with the address of the previous one.
Its a LIFO list.

Dating the heap

Tcache perthread struct

Two items

```
[0x7fae9b0fe295]> pxq @0x562188f85000
0x562188f85000  0x0000000000000000  0x0000000000000251  ........Q.......
0x562188f85010  0x0000000020000000  0x0000000000000000  ................
0x562188f85020  0x0000000000000000  0x0000000000000000  ................
0x562188f85030  0x0000000000000000  0x0000000000000000  ................
0x562188f85040  0x0000000000000000  0x0000000000000000  ................
0x562188f85050  0x0000000000000000  0x0000000000000000  ................
0x562188f85060  0x0000000000000000  0x0000562188f85b40  ........@[..!V..
```

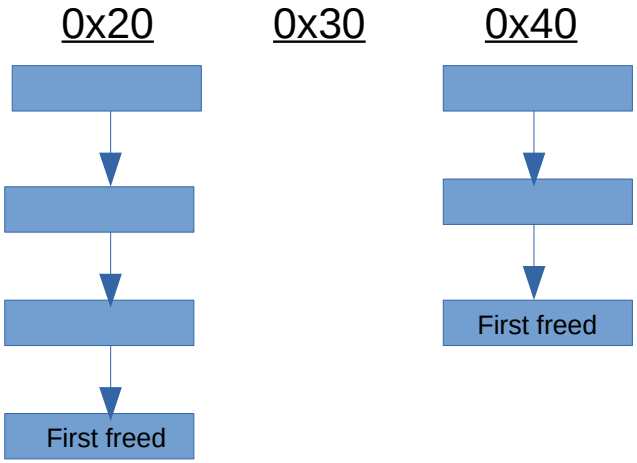
Last freed chunk

```
[0x7fae9b0fe295]> pxq @0x0000562188f85b30
0x562188f85b30  0x0000000000000000  0x0000000000000051  ........Q.......
0x562188f85b40  0x0000562188f85af0  0x0000000000000000  .Z..!V..........
0x562188f85b50  0x0000000000000000  0x0000000000000000  ................
0x562188f85b60  0x0000000000000000  0x0000000000000000  ................
0x562188f85b70  0x0000000000000000  0x0000000000000000  ................
```

First freed chunk

```
[0x7fae9b0fe295]> pxq @0x0000562188f85ae0
0x562188f85ae0  0x0000000000000000  0x0000000000000051  ........Q.......
0x562188f85af0  0x0000000000000000  0x0000000000000000  ................
0x562188f85b00  0x0000000000000000  0x0000000000000000  ................
0x562188f85b10  0x0000000000000000  0x0000000000000000  ................
0x562188f85b20  0x0000000000000000  0x0000000000000000  ................
```

Sample of a simple linked list



Dating the heap

We allocate 2 new chunks, 50 and 40.

Previously we freed a 90’s chunk

Will them fit there ?

Dating the heap

We allocate 2 new chunks, 50 and 40.

Previously we freed a 90’s chunk

Will them fit there ?

(no because the header of the second one ?)

Dating the heap

We allocate 2 new chunks, 50 and 40.

Previously we freed a 90's chunk

Will them fit there ?

(no because the header of the second one ?)

Again, tcache, changes the way it works. Since didn't have any entry for :

50 → 0x32 → 0x40
40 → 0x28 → 0x30

They will be allocated under the last one we created

Dating the heap

```
0x55706a5e2c00  0x0000000000000000  0x0000000000000051  ........Q.......
0x55706a5e2c10  0x6863206874666946  0x61746164206b6e75  Fifth chunk data
0x55706a5e2c20  0x6572656820736920  0x000000000000000a   is here........
0x55706a5e2c30  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2c40  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2c50  0x0000000000000000  0x0000000000000041  ........A.......
0x55706a5e2c60  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2c70  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2c80  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2c90  0x0000000000000000  0x0000000000000031  ........1.......
0x55706a5e2ca0  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2cb0  0x0000000000000000  0x0000000000000000  ................
0x55706a5e2cc0  0x0000000000000000  0x0000000000020341  ........A.......
```

Dating the heap

We have two entries lists 0x50 and 0x70 at tcache
Let’s create a new chunk of size 65 → 0x41 + 0x10 = 0x51
We have 8 bytes (64 bits) of next chunks prev_size , will fit at size 0x50

```
[0x7f5cb1d7c295]> pf.r_tcache_perthread_struct_header @0x55706a5e2000
 prev_size : 0x55706a5e2000 = (qword)0x0000000000000000
      size : 0x55706a5e2008 = (qword)0x0000000000000251
    counts : 0x55706a5e2010 = [ 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x
00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 ]
   entries : 0x55706a5e2050 = (qword)[ 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0
x000055706a5e2af0, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0
x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0
x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0
x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0
x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0
x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0
x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0
x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0
x0000000000000000, 0x0000000000000000, 0x0000000000000000 ]
```

Dating the heap

If we exceed 7 bin per entrie on tcache, bin lists recover their normal behave.

The next freed chunk is stored at their corresponding bin list

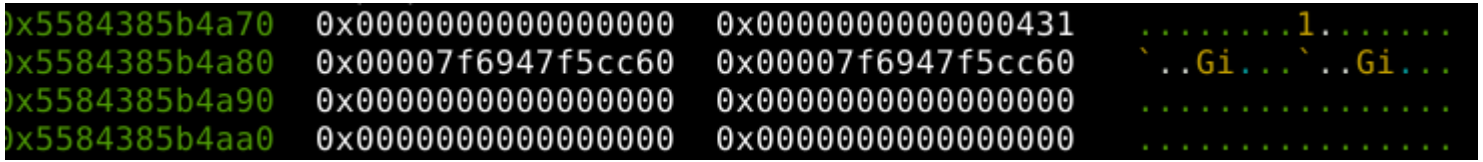
fastbinY (16 to 80 bytes at 32 bits, 32 to 160 bytes at 64 bits manages 10 fast bins)
bins (grater than fastbins)

fastinY : is a simple linked list and behaves like explained for the tcache
bins : is a double linked list constructed by the fd and bk chunk pointers

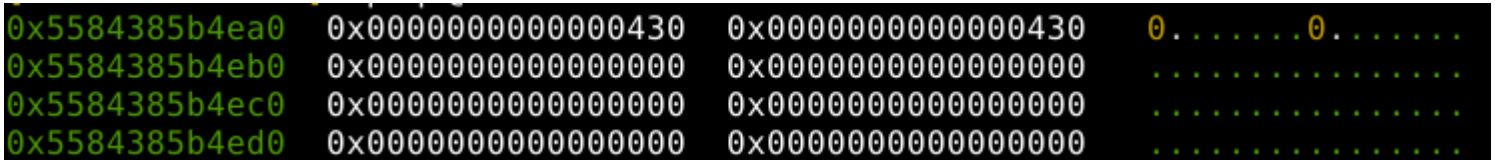
Dating the heap

Tcache only applies for chunk sizes from 24 to 1032 bytes, so we can use this to observe how the “normal” chunk behavior works.

Let’s allocate two 1050 bytes chunks and then free the first one.



Freed chunk now have
fd & bk set
At this stage is unsorted at
the bin table of the arena



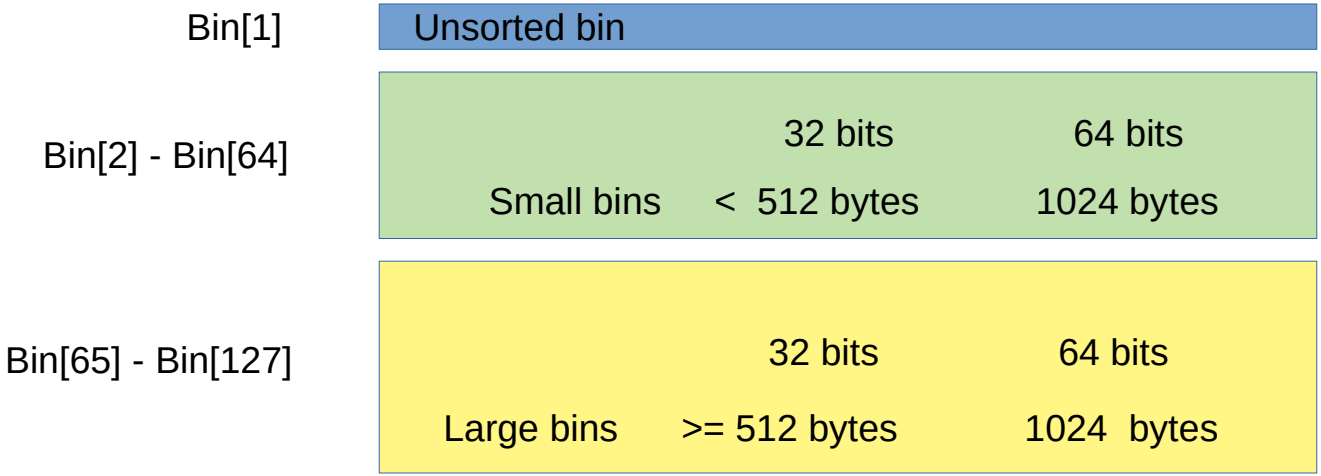
Next adjacent chunk
have set pv_chunk size set

prev_inuse flag is gone
from its size

Dating the heap

Now we allocate a big chunk that not fit on the freed area (grater than 1050)

we force the chunk to move from unsorted bin the corresponding place on the bins table



Dating the heap

```
[0x7f010d72c415]> pf.r_malloc_state_tcache_64 @0x7f010da00c40
           mutex : 0x7f010da00c40 = 0
           flags : 0x7f010da00c44 = 0
 have_fast_chunks : 0x7f010da00c48 = 0
        fastbinsY : 0x7f010da00c4c = (qword)[ 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x000000000000
0000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000 ]
             top : 0x7f010da00ca0 = (qword)0x000055ae214912d0
   last_remainder : 0x7f010da00ca8 = (qword)0x0000000000000000
            bins : 0x7f010da00cb0 = (qword)[ 0x000055ae21490a70, 0x000055ae21490a70, 0x00007f010da00cb0, 0x00007f010da00cb0, 0x00007f010da0
0cc0, 0x00007f010da00cc0, 0x00007f010da00cd0, 0x00007f010da00cd0, 0x00007f010da00ce0, 0x00007f010da00ce0, 0x00007f010da00cf0, 0x00007f010da0
0cf0, 0x00007f010da00d00, 0x00007f010da00d00, 0x00007f010da00d10, 0x00007f010da00d10, 0x00007f010da00d20, 0x00007f010da00d20, 0x00007f010da0
0d30, 0x00007f010da00d30, 0x00007f010da00d40, 0x00007f010da00d40, 0x00007f010da00d50, 0x00007f010da00d50, 0x00007f010da00d60, 0x00007f010da0
```

The recently freed chunk is set as unsorted bin

Dating the heap

```
[0x7f010d72c415]> pf.r_malloc_state_tcache_64 @0x7f010da00c40
           mutex : 0x7f010da00c40 = 0
           flags : 0x7f010da00c44 = 0
 have_fast_chunks : 0x7f010da00c48 = 0
       fastbinsY : 0x7f010da00c4c = (qword)[ 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000 ]
            top : 0x7f010da00ca0 = (qword)0x000055ae21491ab0
   last_remainder : 0x7f010da00ca8 = (qword)0x0000000000000000
           bins : 0x7f010da00cb0 = (qword)[ 0x00007f010da00ca0, 0x00007f010da00ca0, 0x00007f010da00cb0, 0x00007f010da00cb0, 0x00007f010da00cc0, 0x00007f010da00cc0, 0x00007f010da00cd0, 0x00007f010da00cd0, 0x00007f010da00ce0, 0x00007f010da00ce0, 0x00007f010da00cf0, 0x00007f010da00cf0, 0x00007f010da00d00, 0x00007f010da00d00, 0x00007f010da00d10, 0x00007f010da00d10, 0x00007f010da00d20, 0x00007f010da00d20, 0x00007f010da00d30, 0x00007f010da00d30, 0x00007f010da00d40, 0x00007f010da00d40, 0x00007f010da00d50, 0x00007f010da00d50, 0x00007f010da00d60, 0x00007f010da00d60, 0x00007f010da00d70, 0x00007f010da00d70, 0x00007f010da00d80, 0x00007f010da00d80, 0x00007f010da00d90, 0x00007f010da00d90, 0x00007f010da00da0, 0x00007f010da00da0, 0x00007f010da00db0, 0x00007f010da00db0, 0x00007f010da00dc0, 0x00007f010da00dc0, 0x00007f010da00dd0, 0x00007f010da00dd0, 0x00007f010da00de0, 0x00007f010da00de0, 0x00007f010da00df0, 0x00007f010da00df0, 0x00007f010da00e00, 0x00007f010da00e00, 0x00007f010da00e10, 0x00007f010da00e10, 0x00007f010da00e20, 0x00007f010da00e20, 0x00007f010da00e30, 0x00007f010da00e30, 0x00007f010da00e40, 0x00007f010da00e40, 0x00007f010da00e50, 0x00007f010da00e50, 0x00007f010da00e60, 0x00007f010da00e60, 0x00007f010da00e70, 0x00007f010da00e70, 0x00007f010da00e80, 0x00007f010da00e80, 0x00007f010da00e90, 0x00007f010da00e90, 0x00007f010da00ea0, 0x00007f010da00ea0, 0x00007f010da00eb0, 0x00007f010da00eb0, 0x00007f010da00ec0, 0x00007f010da00ec0, 0x00007f010da00ed0, 0x00007f010da00ed0, 0x00007f010da00ee0, 0x00007f010da00ee0, 0x00007f010da00ef0, 0x00007f010da00ef0, 0x00007f010da00f00, 0x00007f010da00f00, 0x00007f010da00f10, 0x00007f010da00f10, 0x00007f010da00f20, 0x00007f010da00f20, 0x00007f010da00f30, 0x00007f010da00f30, 0x00007f010da00f40, 0x00007f010da00f40, 0x00007f010da00f50, 0x00007f010da00f50, 0x00007f010da00f60, 0x00007f010da00f60, 0x00007f010da00f70, 0x00007f010da00f70, 0x00007f010da00f80, 0x00007f010da00f80, 0x00007f010da00f90, 0x00007f010da00f90, 0x00007f010da00fa0, 0x00007f010da00fa0, 0x00007f010da00fb0, 0x00007f010da00fb0, 0x00007f010da00fc0, 0x00007f010da00fc0, 0x00007f010da00fd0, 0x00007f010da00fd0, 0x00007f010da00fe0, 0x00007f010da00fe0, 0x00007f010da00ff0, 0x00007f010da00ff0, 0x00007f010da01000, 0x00007f010da01000, 0x00007f010da01010, 0x00007f010da01010, 0x00007f010da01020, 0x00007f010da01020, 0x00007f010da01030, 0x00007f010da01030, 0x00007f010da01040, 0x00007f010da01040, 0x00007f010da01050, 0x00007f010da01050, 0x00007f010da01060, 0x00007f010da01060, 0x00007f010da01070, 0x00007f010da01070, 0x00007f010da01080, 0x00007f010da01080, 0x000055ae21490a70, 0x000055ae21490a70, 0x00007f010da010a0, 0x00007f010da010a0, 0x00007f010da010b0, 0x00007f010da010b0 ]
```

Bin is freed at its place

Dating the heap

And this is the freed chunk pointing to its final place at bins

```
[0x7f010d72c415]> pxq @0x55ae21490a70
0x55ae21490a70  0x0000000000000000  0x0000000000000431  ........1.......
0x55ae21490a80  0x00007f010da01090  0x00007f010da01090  ................
0x55ae21490a90  0x000055ae21490a70  0x000055ae21490a70  p.I!.U..p.I!.U..
0x55ae21490aa0  0x0000000000000000  0x0000000000000000  ................
```

If we allocate a new chunk that fits a freed space in bins,
it will be allocated, and its excess will be stored as unsorted bin,
behaving normally.

Dating the heap

To illustrate how a double linked list works we could :

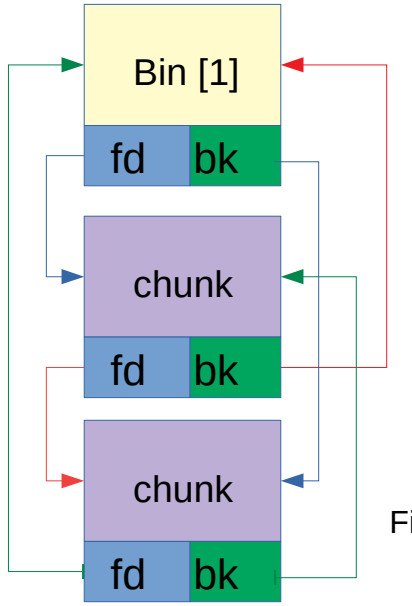
allocate 1050 - 0x558c75178a70
allocate 1050 - 0x558c75179250
allocate 1050 - 0x558c75179a30
allocate 1050 - 0x558c7517a210
free 1 - 0x558c75178a70
free 3 - 0x558c75179a30

```
bins : 0x7f8be18c7cb0 = (qword)[ 0x0000558c75179a30, 0x0000558c75178a70,

0x558c75179a40  0x0000558c75178a70  0x00007f8be18c7ca0

0x558c75178a80  0x00007f8be18c7ca0  0x0000558c75179a30
```

First freed

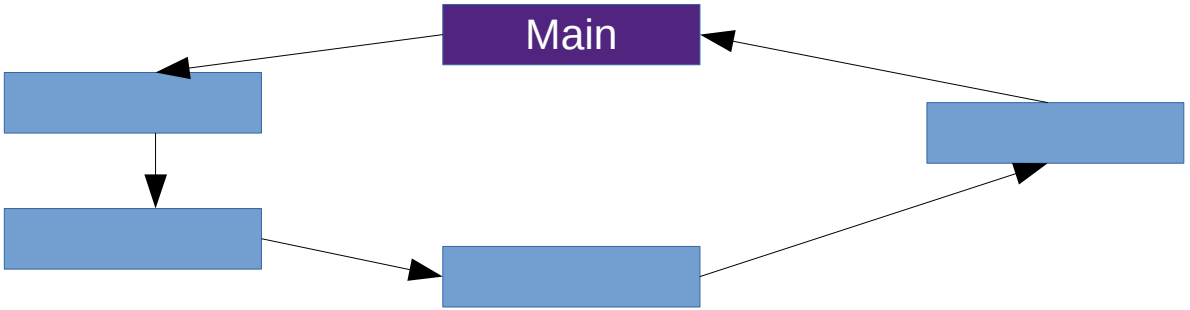


First freed

Dating the heap

When running a multi-thread program, each new thread maps a new arena.

This arenas are used as normal until the thread, then is freed
Arenas assemble a simple list linked by pointers



Dating the heap

The arenas linked link could be fallowed via field next at malloc_state

```
main_arena @ 0x7f26b88a7c40
thread arena @ 0x7f26a8000020
thread arena @ 0x7f26b0000020
[0x7f26b88bfa93]> pf.r_malloc_state_tcache_64.next @0x7f26b88a7c40
             next : 0x7f26b88a84b0 = (qword)0x00007f26a8000020
[0x7f26b88bfa93]> pf.r_malloc_state_tcache_64.next @0x00007f26a8000020
             next : 0x7f26a8000890 = (qword)0x00007f26b0000020
[0x7f26b88bfa93]> pf.r_malloc_state_tcache_64.next @0x00007f26b0000020
             next : 0x7f26b0000890 = (qword)0x00007f26b88a7c40
[0x7f26b88bfa93]>
```

Exploiting the heap

Time to show how tcache could be exploited.

Here you can found a friendly exploiters program that permit overflow one chunk over the next one without any restrictions

https://github.com/morbith-dqtz/r2con2018/blob/master/source_examples/heap_play_net_threaded.c

The solved exercises introduces :

Overlap chunks
Tcache poisoning
House of spirit

Exploiting the heap

Overlap the chunks

Consist into overflow a chunk and alter the size at the header of the next one.

If the altered chunk was freed, then when you ask malloc the crafted size,
chunk will be placed at the same place was befor free, but assuming its length is the new one

If previous size < edited one, it will overlaps the next chunk (or chunks).

Exploiting the heap

Poisoning tcache

Consists into overflow a chunk and alter the forward pointer at the header of the next one.

If the altered chunk was freed, then if you ask malloc again the edited chunk size,
it reallocates the edited chunk, but our crafted fd pointer do the trick.

If we malloc again, glib take as valid the fd pointer and returns its address

Exploiting the heap

House of spirit

Consists into alter the address of a pointer, and redirect it to some place we can write to.

In that place we write a entirely fake chunk.

When the program frees the hijacked pointer, our fake chunk passes to free bins

When malloc is ask for the size of our fake chunk, it will returns the address that we control