A gentle introduction to exploitation

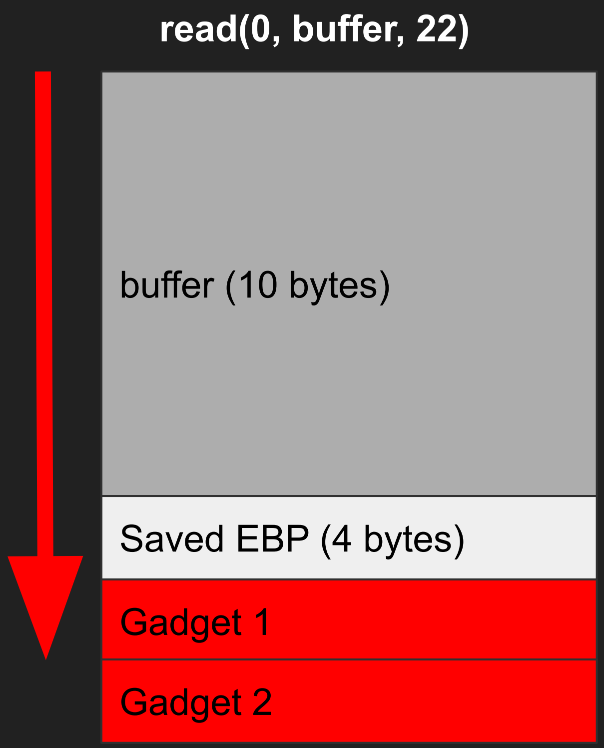
Part 2b : Heap for noobs

by anticlockwise & malweisse

Stack pivoting

Once upon a time, in the magic lands of Exploitation, there was a little hacker that could only insert few bytes in the stack.

He needed a long ropchain, so what he did?



Stack pivoting

read(0, buffer, 22)

buffer (10 bytes)

Saved EBP (4 bytes)

pop esp; ret

0xabadcafe

Filled somewhere in the code before the vulnerable return

0xABADCAFE:

Heap 4 dummies

- `malloc(size)`: gives a chunk of memory of `size` rounded to the nearest 16 multiple
- `free(ptr)`: frees the memory space pointed to by `ptr`
- `calloc(n, size)`: allocates zero initialized memory for an array of `n` elements of `size` bytes each
- `realloc(ptr, size)`: function changes the size of the memory block pointed to by `ptr` to `size` bytes

Heap Golden Rules

- Never read or write to a pointer that has been already freed
- Never use uninitialized heap memory locations
- Never read or write outside `[ptr, ptr + size)`
- Never pass a pointer to `free` more than once
- Never pass a pointer to `free` that was not originated by `malloc`
- Never use a pointer returned by malloc before checking if `ptr == NULL`

Allocators

Several allocators exists and every day you deal with **ptmalloc**, that is the default allocator of the GLIBC on Linux.

Other popular allocators are:

- jemalloc (default for Firefox and Android)
- tcmalloc
- hoard

TrxMalloc

We will use **trxmalloc** that is a didactical allocator build in less the 500 lines of C to teach the basic concepts of heap exploitation.

It’s design is highly inspired on ptmalloc and ptmalloc-based exploits can be easily adapted to this allocator.

It is simplified and totally thread unsafe. Security checks are missing.

malloc() High-Level Overview

1. If there is a previously freed chunk with a compatible size it is served
2. Otherwise, if there is available space at the top of the heap, a new chunk is created and then served (*)
3. Otherwise, the libc asks for more memory to the kernel
4. Otherwise, malloc() returns NULL

(*) Large requests are served using mmap()

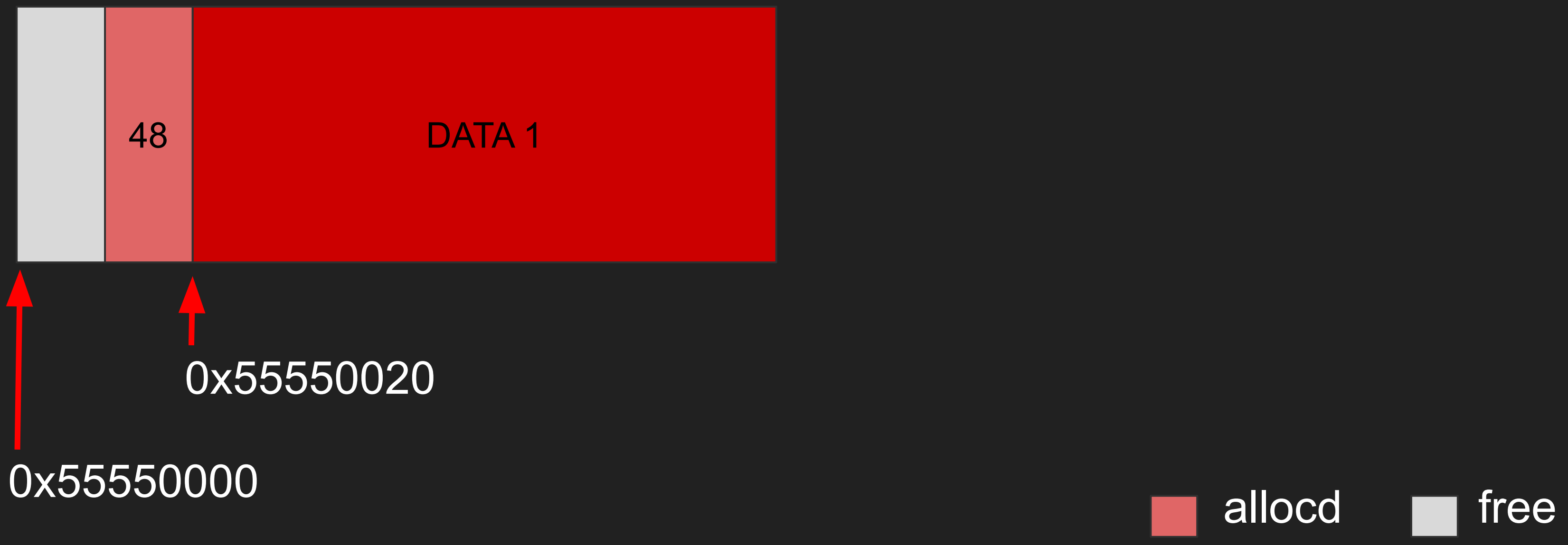
Chunks

```
struct malloc_chunk {

  size_t      prev_size;       /* Size of previous chunk (if free).  */
  size_t      size;            /* Size in bytes, including overhead. */

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

};
```

`size` last 3 bits are used to store additional information (the last 4 bits of a multiple of 0x10 is always 0). If the last bit is 1 the previous chunk (in linear memory) is in use.

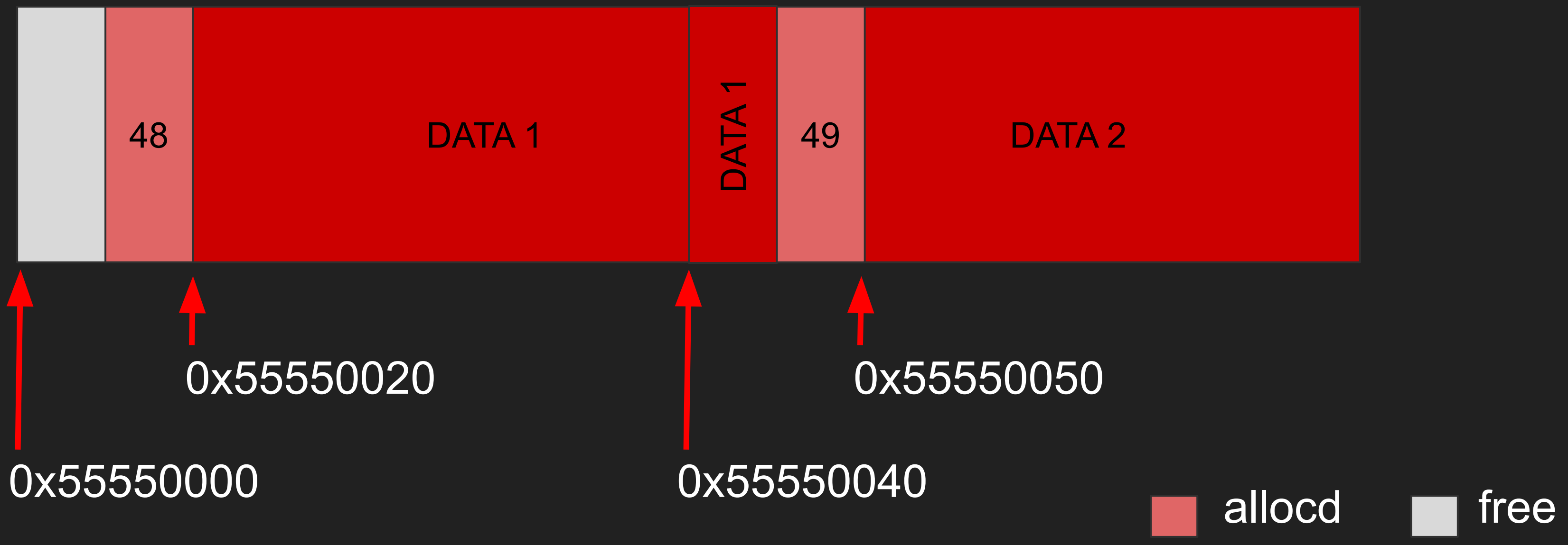
Chunks

malloc(40) -> 0x55550020



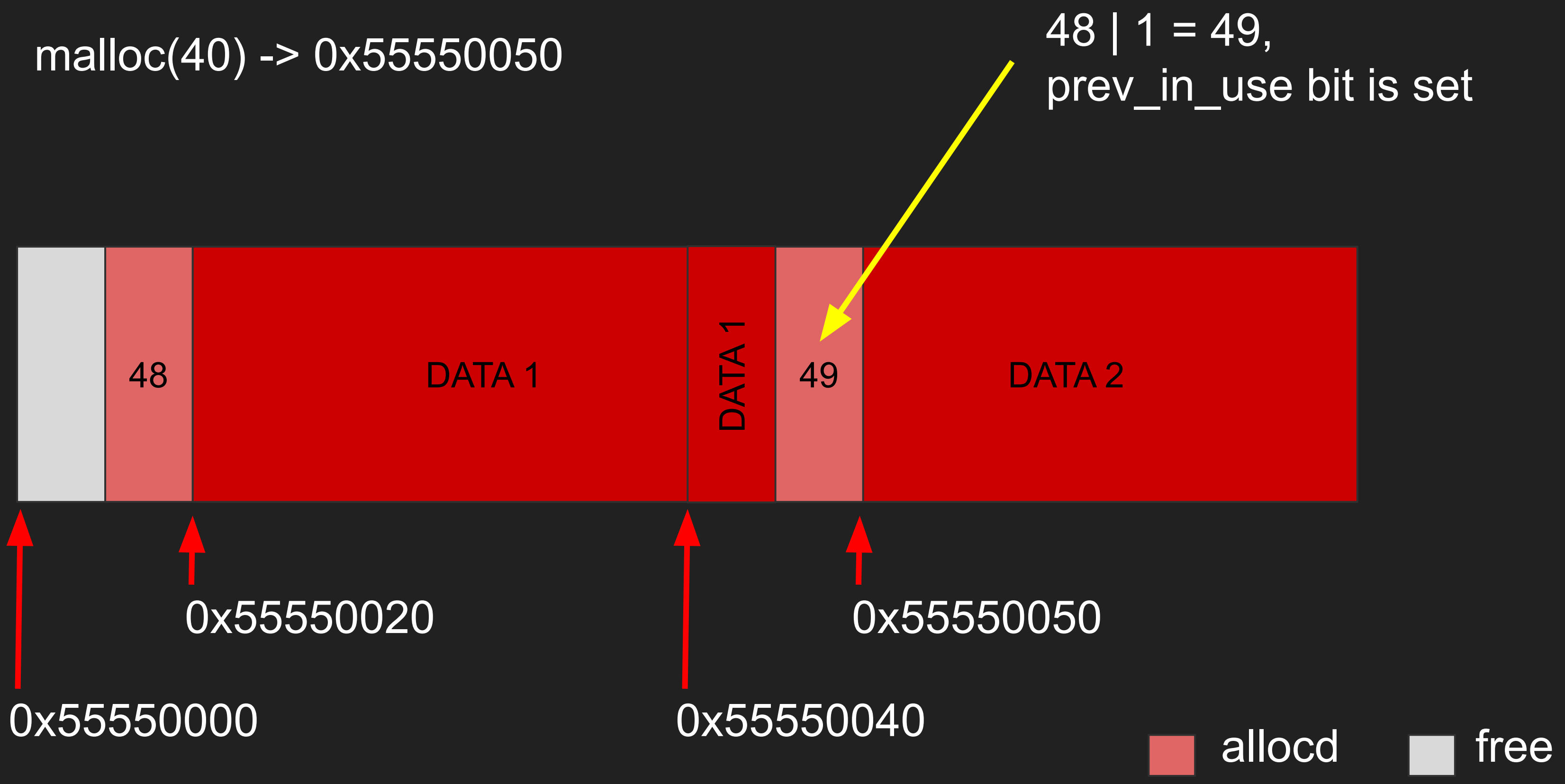
Chunks

malloc(40) -> 0x55550050



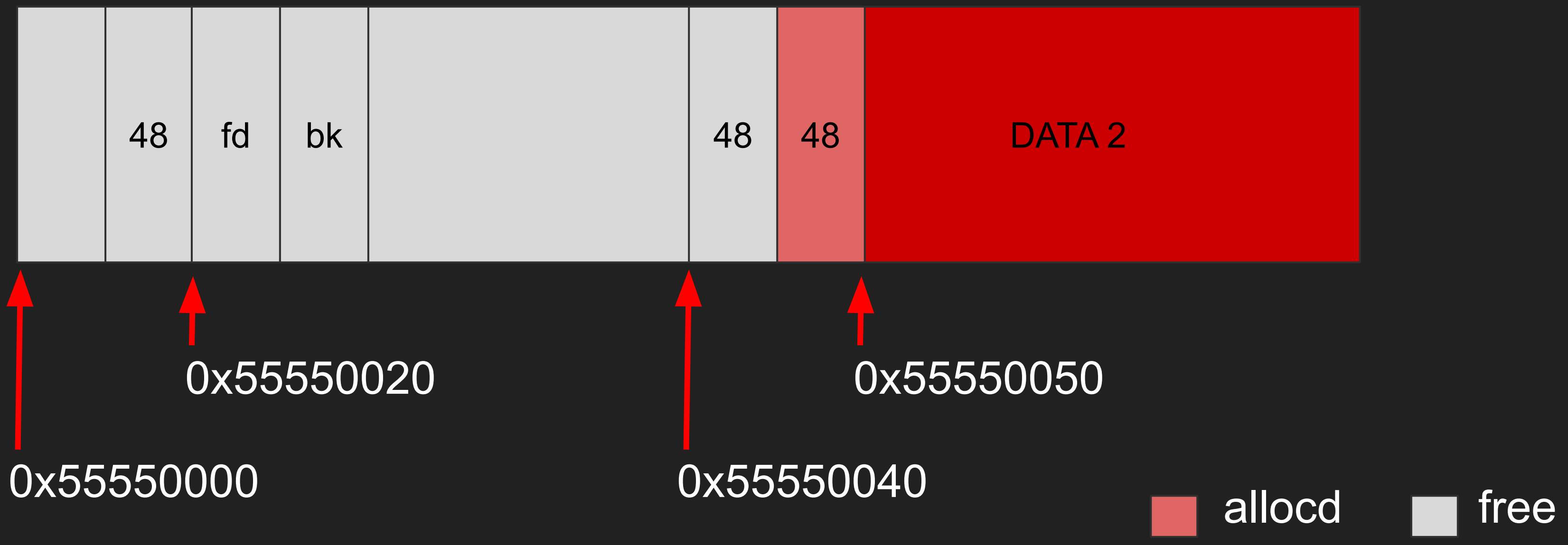
Chunks

malloc(40) -> 0x55550050



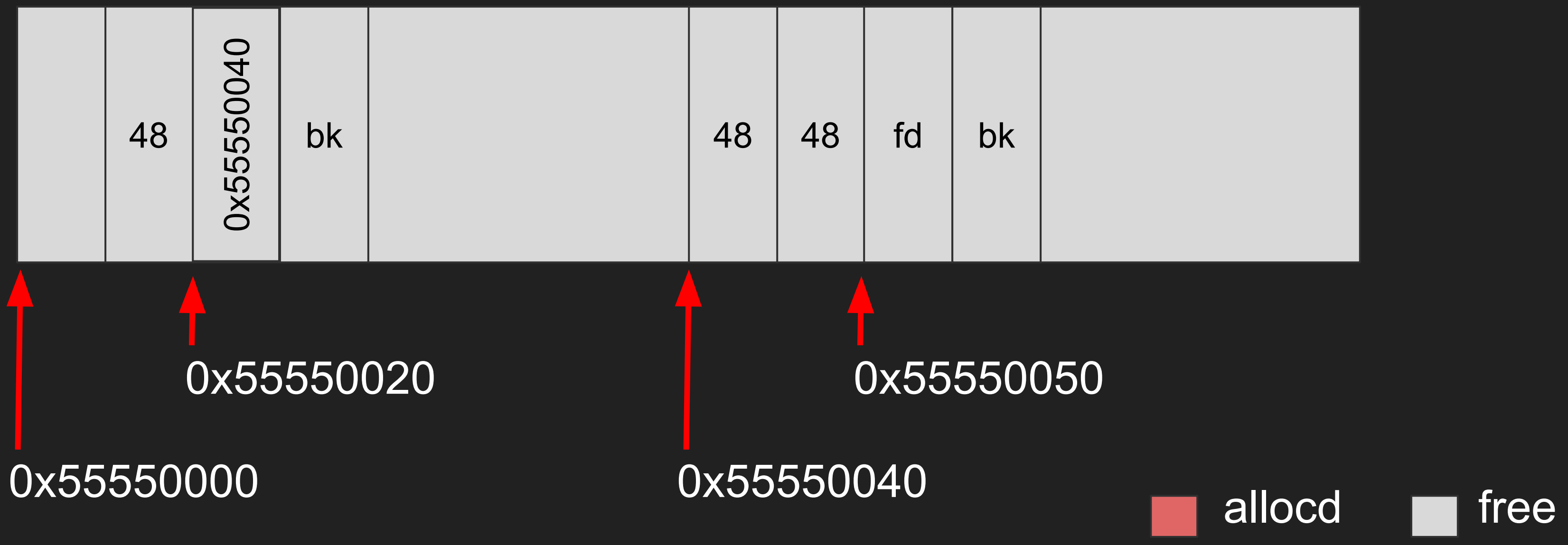
Chunks

free(0x55550020)

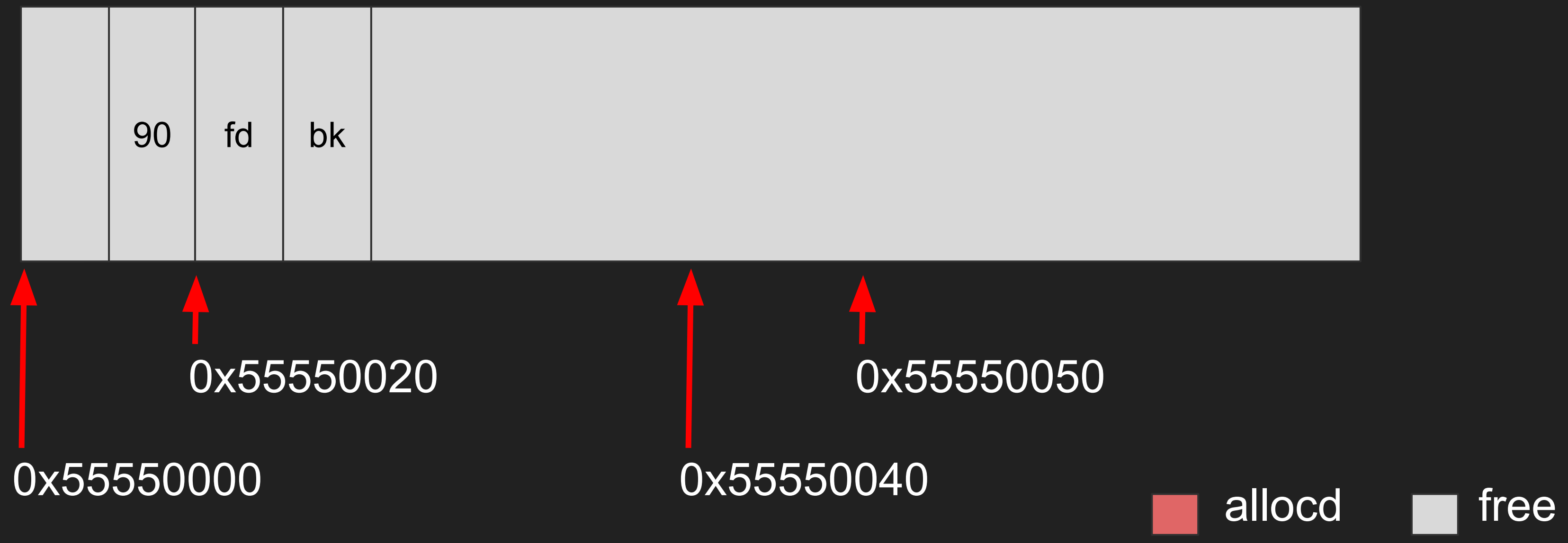


Chunks

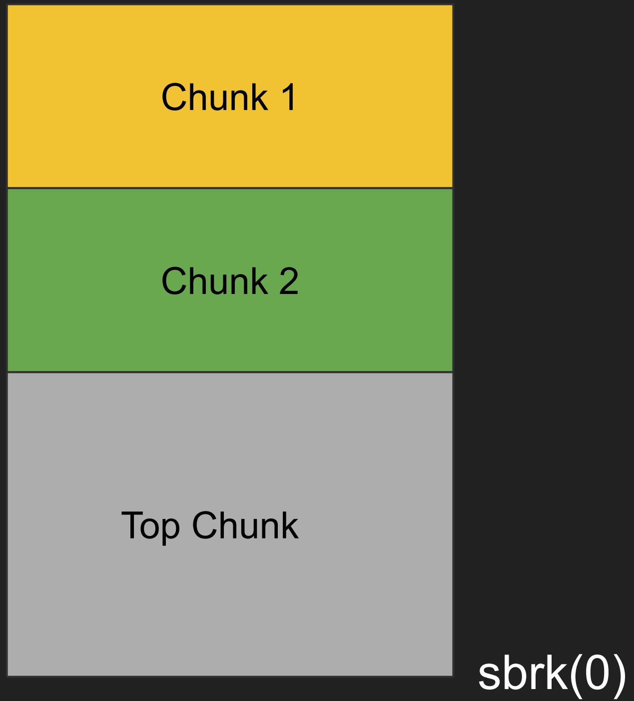
free(0x55550050)



Consolidation



Top Chunk

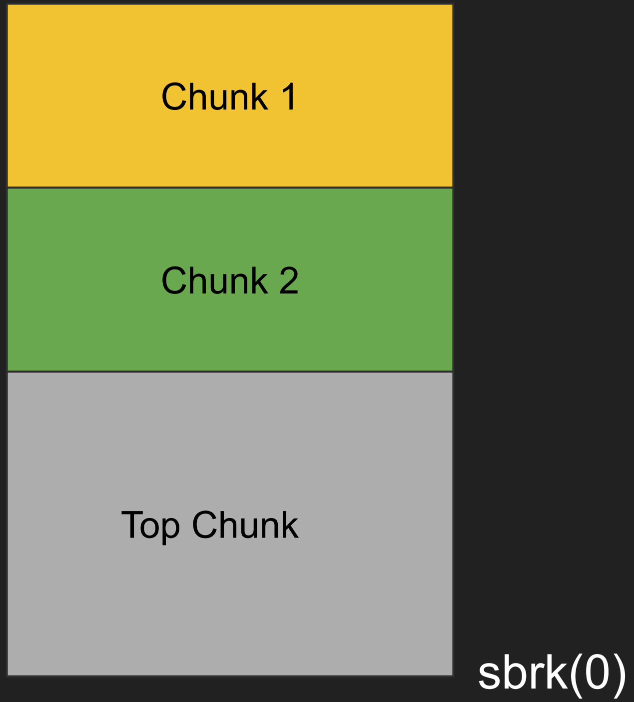


The top chunk is the last chunk of the heap.

It is between the user chunks and the end of the heap.

```
if requested_size < top_chunk_size
    then reduce top_chunk
else ask new memory via brk
```

Consolidation with Top Chunk

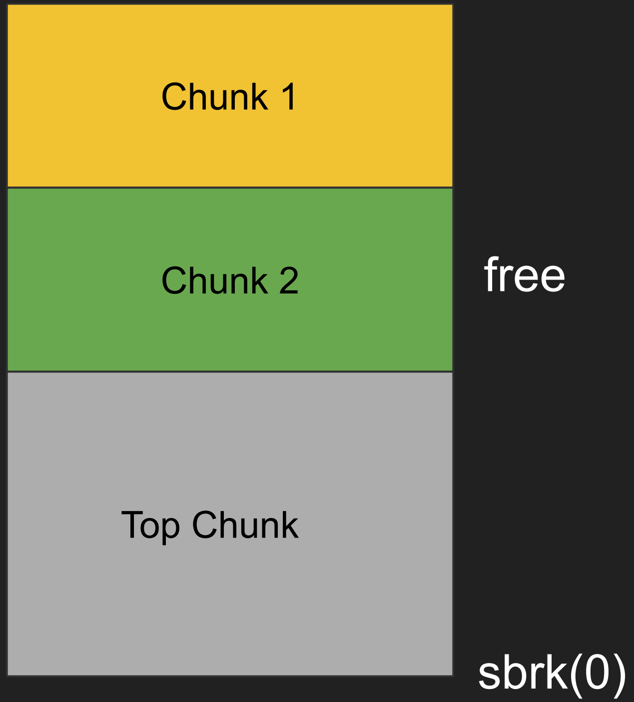


The previous chunk in memory to the top chunk must be always allocated.

This means that the prev_inuse bit of the top chunk is always set.

The previous chunk in memory of the top chunk (2 in this case) is immediately consolidated to the top chunk to achieve this.

Consolidation with Top Chunk

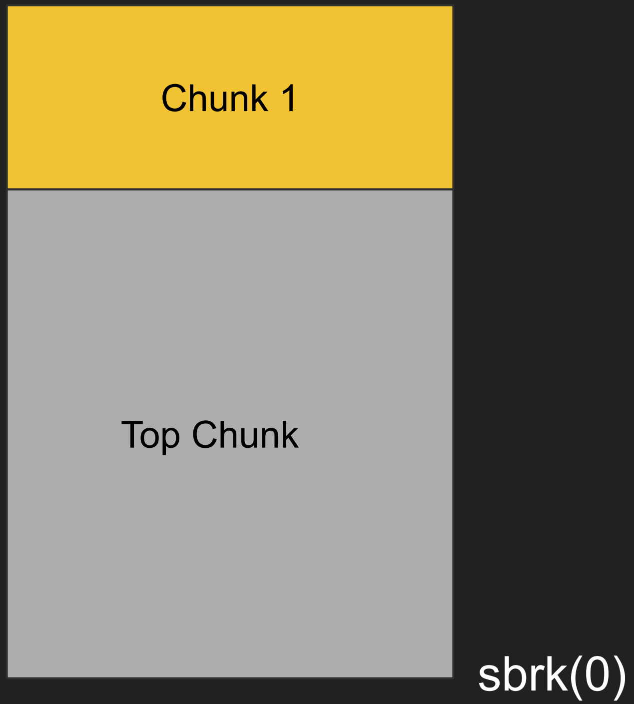


The previous chunk in memory to the top chunk must be always allocated.

This means that the prev_inuse bit of the top chunk is always set.

The previous chunk in memory of the top chunk (2 in this case) is immediately consolidated to the top chunk to achieve this.

Consolidation with Top Chunk



The previous chunk in memory to the top chunk must be always allocated.

This means that the prev_inuse bit of the top chunk is always set.

The previous chunk in memory of the top chunk (2 in this case) is immediately consolidated to the top chunk to achieve this.

Bins

Chunks, when freed, goes to a bin that is a list of free chunks.

Fast bins are single linked lists (NULL terminated) of chunks (bk is ignored) of the same size. In trxmalloc there are 7 fast bins for the sizes 0x20 … 0x80. This list is LIFO.

Small bins are double linked list. Insertion is at head, removal at tail (FIFO).

In trxmalloc a chunk with size > 0x800 is served via mmap(), in ptmalloc2 there are more fastbins and smallbins and there are also the large bins.

Arena

```c
struct malloc_state {

  struct malloc_chunk* fastbins[NFASTBINS];
  struct malloc_chunk* top;
  struct malloc_chunk* bins[NBINS * 2 - 2];
};

struct malloc_state arena;
```

Fastbins, top chunk and bins are maintained in a global variable arena.

Arena

```c
struct malloc_state
{
  __libc_lock_define (, mutex);
  int flags;
  int have_fastchunks;
  mfastbinptr fastbinsY[NFASTBINS];
  mchunkptr top;
  mchunkptr last_remainder;
  mchunkptr bins[NBINS * 2 - 2];
  unsigned int binmap[BINMAPSIZE];
  struct malloc_state *next;
  struct malloc_state *next_free;
  INTERNAL_SIZE_T attached_threads;
  INTERNAL_SIZE_T system_mem;
  INTERNAL_SIZE_T max_system_mem;
};
```

ptmalloc has multiple arenas and they are a magnitude of times complicated over trxmalloc arena.

ptmalloc stardard arena is called

main arena.

So what is the idea?

- Each time `malloc` or `free` are called they perform different actions based on the metadata they have:
  - Choose where is the next chunk to return
  - Merge different chunks together
  - Allocate new space for the heap
- If we corrupt the metadata we can induce the allocator to bad actions
  - usually we achieve Read or Write to arbitrary memory :)

Fastbins 101

- `a = malloc(20); b = malloc(20); c = malloc(20)`
- `free(a)`   `fastbin[0]: a -> NULL`
- `free(b)`   `fastbin[0]: b -> a -> NULL`
- `free(c)`   `fastbin[0]: c -> b -> a -> NULL`
- `assert( c == malloc(20) )`
- `assert( b == malloc(20) )`
- `assert( a == malloc(20) )`

Use After Free

- `a = malloc(20); b = malloc(20); c = malloc(20)`
- `free(a)`   `fastbin[0]: a -> NULL`
- `free(b)`   `fastbin[0]: b -> a -> NULL`
- `free(c)`   `fastbin[0]: c -> b -> a -> NULL`
- `d = malloc(20)`
- `*c = 0xabadcafe`
- `assert( *d == 0xabadcafe )`

Double Free

- `free(a)`
- `free(a)`

What can go wrong?

Double Free

- `a = malloc(20)`
- `free(a)`

```
fastbin[0]: a -> NULL
```

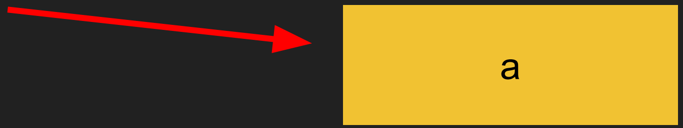
Double Free

- `a = malloc(20)`
- `free(a)`
- `free(a)`

```
fastbin[0]: a -> a -> NULL
```

Double Free

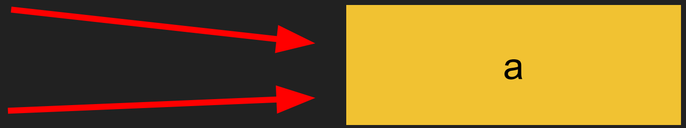
- `a = malloc(20)`
- `free(a)`
- `free(a)`
- `b = malloc(20)`



`fastbin[0]: a -> NULL`

Double Free

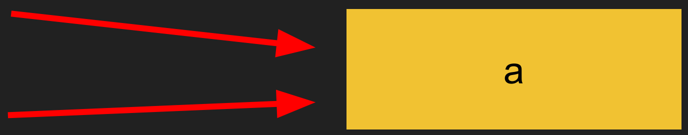
- `a = malloc(20)`
- `free(a)`
- `free(a)`
- `b = malloc(20)`
- `c = malloc(20)`



`fastbin[0]: NULL`

Double Free

- `a = malloc(20)`
- `free(a)`
- `free(a)`
- `b = malloc(20)`
- `c = malloc(20)`
- `*b = 0xabadcafe`
- `assert( *c == 0xabadcafe )`



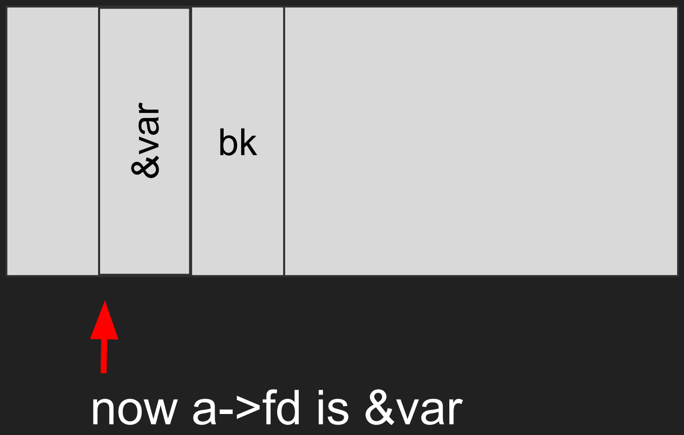
Fastbin Attack (UAF)

- `a = malloc(20)`
- `free(a)`

`fastbin[0]: a -> NULL`

Fastbin Attack (UAF)

- `a = malloc(20)`
- `free(a)`
- `*a = &var`



`fastbin[0]: a -> &var`

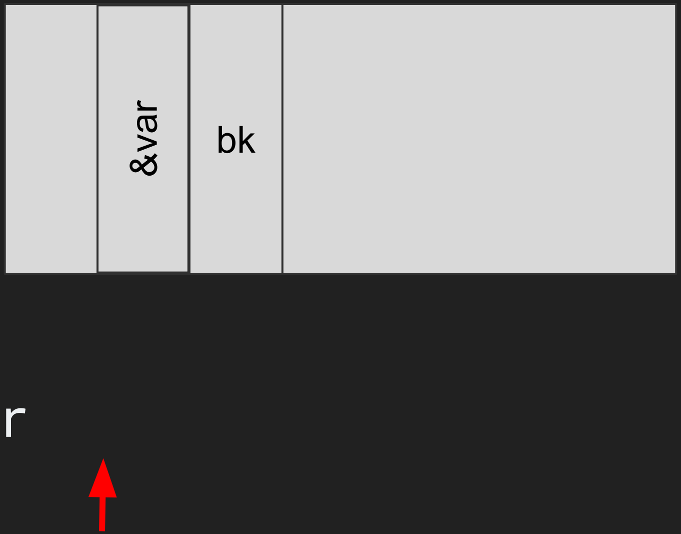
Fastbin Attack (UAF)

- `a = malloc(20)`
- `free(a)`
- `*a = &var`
- `malloc(20) -> a`

`fastbin[0]: &var`

Fastbin Attack (UAF)

- `a = malloc(20)`
- `free(a)`
- `*a = &var -16`
- `malloc(20) -> a`
- `malloc(20) -> &var`



can control the content of *var*

`fastbin[0]: &var`

Small Chunks based Leak

We said that the bins are double linked list and the bin itself is treated as a chunk in the list.

The arena is in the .data of a library (libc in ptmalloc).

Leaking the bk pointer of the last freed chunk in a bin we can leak the address of the bin itself.

Small Chunks based Leak

Look at how small chunks are inserted in the bin when freed:

```
sz = chunksize(ck);

idx = size2bin(sz);

fd = bin_at(idx)->fd;

bin_at(idx)->fd = ck;

ck->bk = bin_at(idx); // HERE the heap of the bin has bk = bin_addr

ck->fd = fd;

fd->bk = ck;
```

Security checks

General purpose allocators implements secuiry checks to make harder the life of an exploit developer.

For example in ptmalloc `free(a) free(a)` can't be done cause the allocator checks of the head of fastbin is the same chunk that we are freeing.

This specific example will crash our program with a `double free or corruption (fasttop)` security abort.

Security checks

To do a double free in ptmalloc you must insert and additional free between the two `free(a)` in order to prevent the abort.

- `free(a)`   `fastbin[0]: a -> NULL`
- `free(b)`   `fastbin[0]: b -> a -> NULL`
- `free(a)`   `fastbin[0]: a -> b -> a -> NULL`

When freing a for the second time the top of the fastbin 0 is b and so this is ok.

Security checks

A partial list of security checks of ptmalloc can be found at

https://heap-exploitation.dhavalkapil.com/diving_into_glibc_heap/security_checks.html

trxmalloc does not perform any security check so you life is easier at the moment dude.

Malloc Hooks

In ptmalloc and trxmalloc there are, for debugging/profiling purposes, function pointers that can be used to substitute malloc/free/realloc/memalign… functions. Maybe an attacker can do something with them?

When those pointer are != NULL, the pointed callback is called in place of the related function.

In trxmalloc search for:

__trx_malloc_hook, __trx_realloc_hook, __trx_free_hook

Others Attacks

- Heap exploitation is huge, and varies depending on the exact library used (2.26, 2.27, …, trxmalloc)
- To learn more:
  - https://heap-exploitation.dhavalkapil.com/ (most things deprecated)
  - https://github.com/shellphish/how2heap (only code but almost at the state of the art)
  - https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/ (very good and recent)
  - CTF Writeups

TODO

If you look inside trx_malloc.c there are many TODOs.

If you want to be a good hacker you must be a good with this kind of stuffs of the operating system so open VIM and implement them!

The most important features to write are:

_trx_malloc_consolidate() and the creation of the remainder chunk in trx_malloc()

TODO (security)

Now comes the leet part.

Today you will develop your exploits on simple vulnerable binaries. The last challenge is, at home, to **implement some security checks** of ptmalloc inside trxmalloc.

Your goal is to **harden** trxmalloc and block our exploits.

```
The spirit of Dennis Ritchie is
with you, young padawan!
```

Debugging tips

- `ltrace -e malloc+free ./binary`
- `pwntrace (ltrace in pwntools)`
  https://github.com/andreafioraldi/pwntrace [unmaintained]

- `libtrxmalloc debug build: make debug`
  trxmalloc has debug symbols by default, just print variables to navigate heap structures (e.g. p arena.fastbins)

- GEF/pwndbg (for ptmalloc): help heap