Heapple Pie

The macOS/iOS default heap

Date 14/09/2018

At Sthack 2018

By Eloi Benoist-Vanderbeken

Whoami

- **Eloi Benoist-Vanderbeken**
- **@elvanderb on twitter**

- **Working for Synacktiv:**
  - Offensive security company (pentest, red team, vuln hunting, exploitation, tool dev, etc.)

- **Reverse engineering team coordinator:**
  - 14 reversers / 36 ninjas
  - Focus on low level dev, reverse, vuln research/exploitation
  - If there is software in it, we can own it :)
  - We are recruiting!

Introduction

Why this presentation?

- **Growing interest in macOS/iOS**
  - JailBreak scene → fame^3 - money^0
  - Lots of pwn competitions → fame^2 - money^1

    (mobile) Pwn2Own

    PWNFEST

    GeekPwn

    XPwn...
  - Vulnerability brokers → fame^0 - money^3
  - Apple Bug Bounty → fame^2 - money^2

    If you manage to get paid…

- **But almost no documentation on the macOS/iOS user default heap from an exploiter point of view**

Why so little love?

- **Safari exploits → WebKit heap**
  - lots of good resources
  - kudos to saelo
- **Kernel exploits → kernel heap**
  - lots of good resources
  - kudos to Stefan Esser
- **Services exploits**
  - lots of logic bugs
- **But…**
  - All the Obj-C framework and almost all the other lib / exe are based on the default heap

Previous work

- **OS X Heap Exploitation Techniques – 2005 – Nemo**
  - Not a lot of details on heap internals
  - Outdated (64bits kills the exploitation technique)
- **Mac OS Xploitation (and others) – 2009 – Dino A. Dai Zovi**
  - Outdated (new checksums)
- **In the Zone: OS X Heap Exploitation – 2016 – Tyler Bohan**
  - Good description of the heap
  - LLDB scripts released
  - Describes some exploitation techniques as how to transform a heap overflow into a use-after-free (more on this later…)

How does malloc works

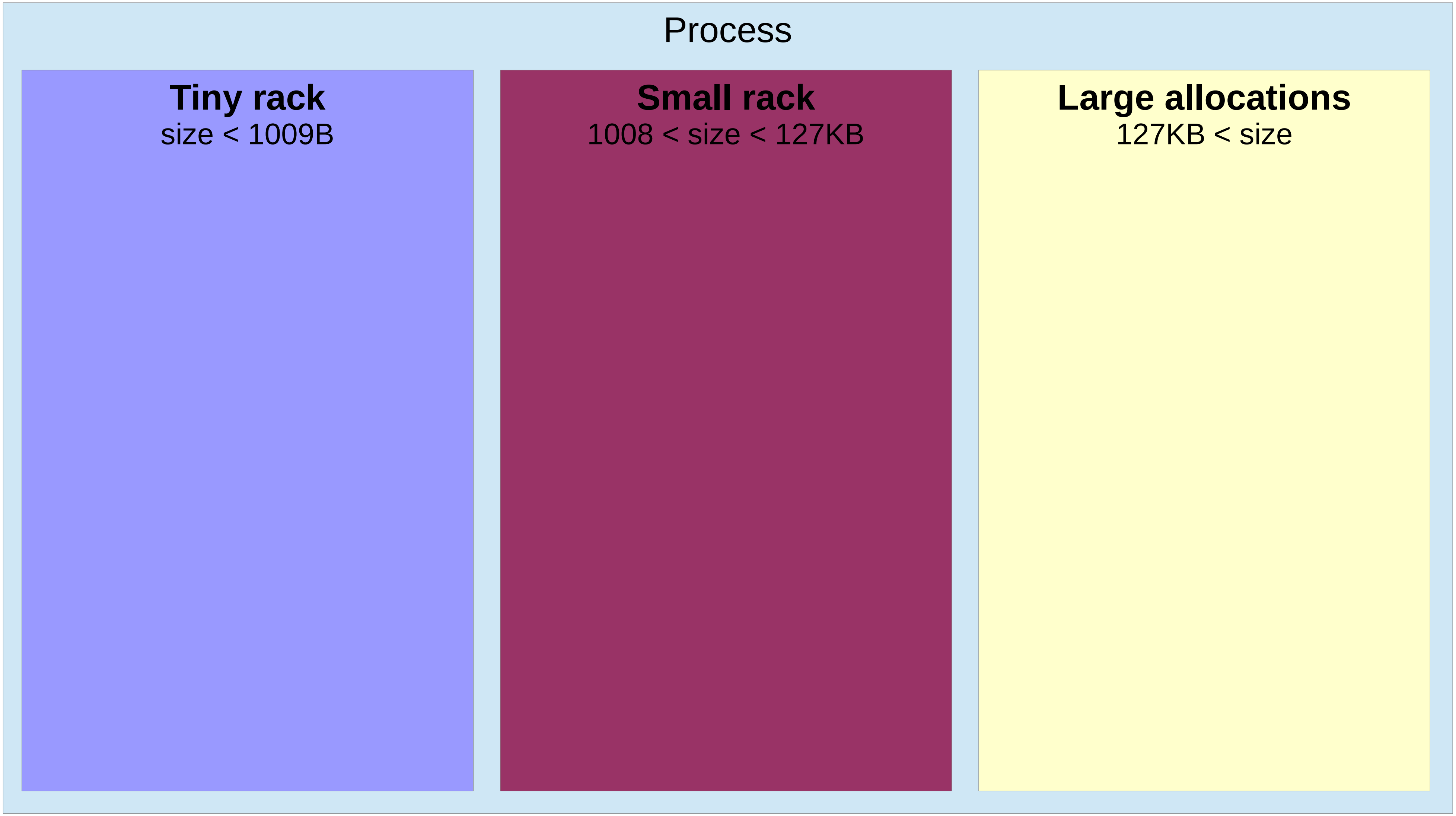
malloc zones

- ***malloc* is actually just a wrapper on *malloc_zone_malloc***
  - called with the default zone which is a scalable zone
  - we will focus on this zone
- **Other zones can be registered**
  - WebKit Malloc
  - GFXMallocZone
  - QuartzCore
  - etc.
- **malloc_zone_{malloc/free/realloc/...} functions are just wrappers that call zone functions**
  - zone functions handle the allocation
  - malloc_zone_* functions handle the generic stuff
    - find the zone associated with the passed pointer
    - log / trace / periodically check the zone / etc.
- **malloc will always allocate from the default heap but realloc/free/malloc_size can be called with pointers belonging to other zones**

How does the scalable zone works

- **Each process has two racks**
  - tiny
    - ≤ 1008 bytes on a 64bits machine
    - ≤ 496 bytes on a 32bits machine
  - small
    - ≤ 15 KB on machine with less than 1GB of memory
    - ≤ 127 KB else
  - from now on, we will only consider the 64bits and +1GB case
- **If an allocation doesn’t fit in the small rack then the large allocator is used**
  - directly allocates pages
  - we won’t talk about this allocator
    - not often encountered and not really interesting from an exploitation point of view
- **There is an other allocator, the nano allocator, but it is not activated by default**
  - used for allocations < 256 B
  - activated with a special posix_spawn undocumented flag (_POSIX_SPAWN_NANO_ALLOCATOR) or with the MallocNanoZone environment variable set to 1.
  - quite interesting but that’s an other story…

How does the scalable zone works



Process

Tiny rack
size < 1009B

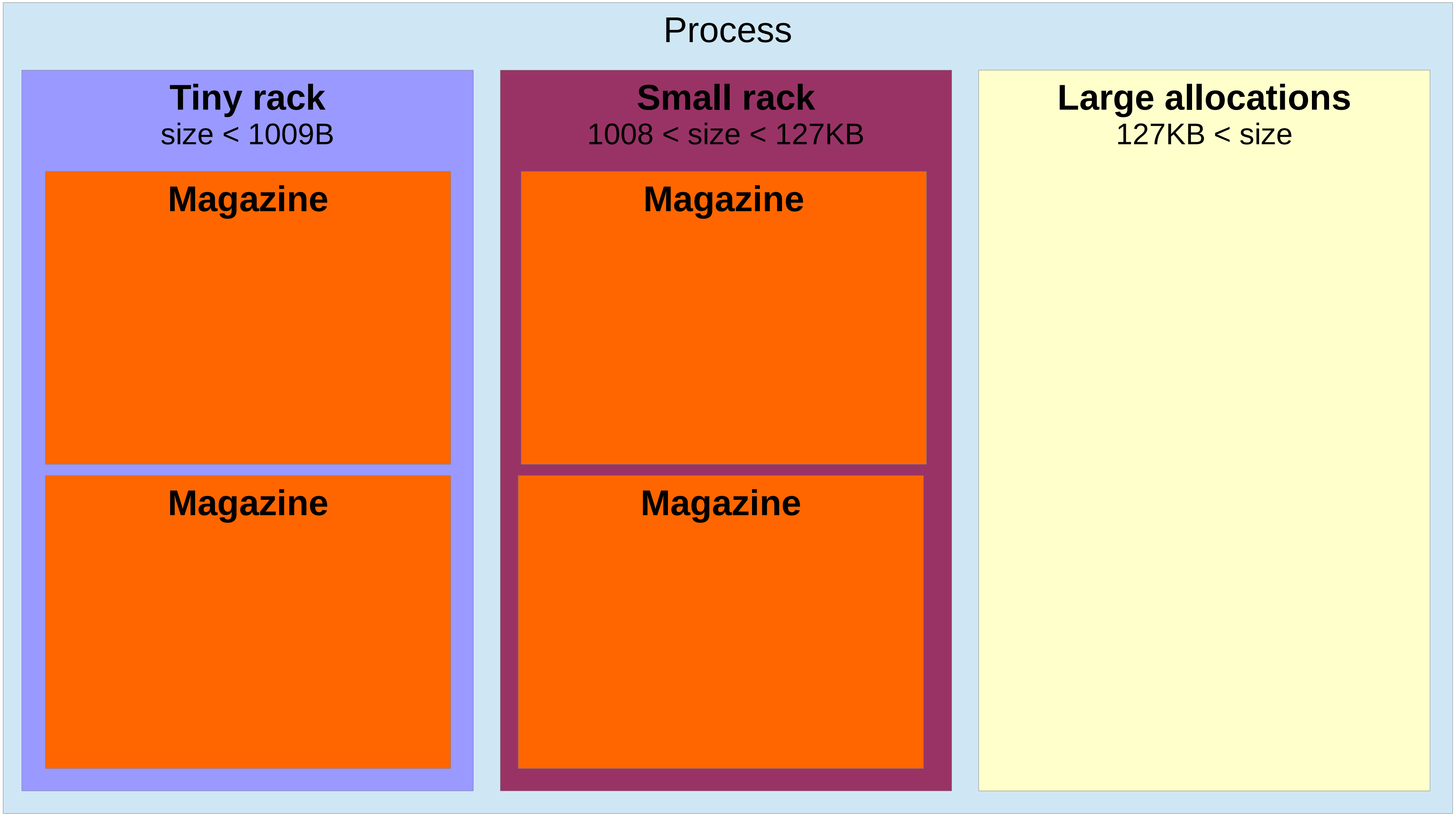
Small rack
1008 < size < 127KB

Large allocations
127KB < size

How does the scalable zone works

- **Each rack has one magazine per physical core**
  - optimize the processor caches accesses
  - reduce the risk of concurrent access (less locks)

How does the scalable zone works



Process

Tiny rack
size < 1009B

Magazine

Magazine

Small rack
1008 < size < 127KB

Magazine

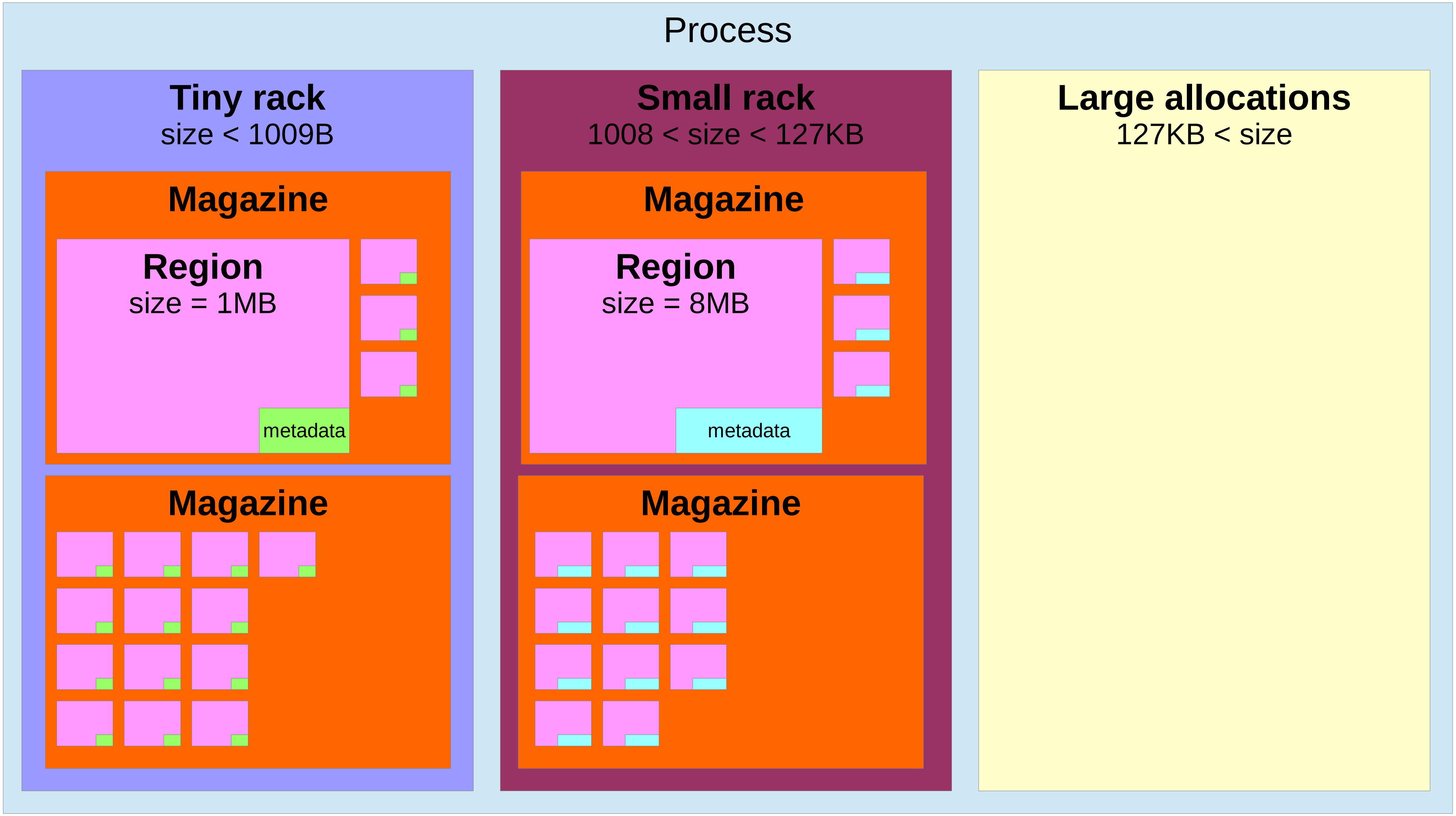
Magazine

Large allocations
127KB < size

How does the scalable zone works

- **Each rack has one magazine per physical core**
  - optimize the processor caches accesses
  - reduce the risk of concurrent access (less locks)
- **Each magazine has multiple regions**
  - 1MB for tiny allocations
  - 8MB for small ones
  - metadata (rack specific) is at the end of the region

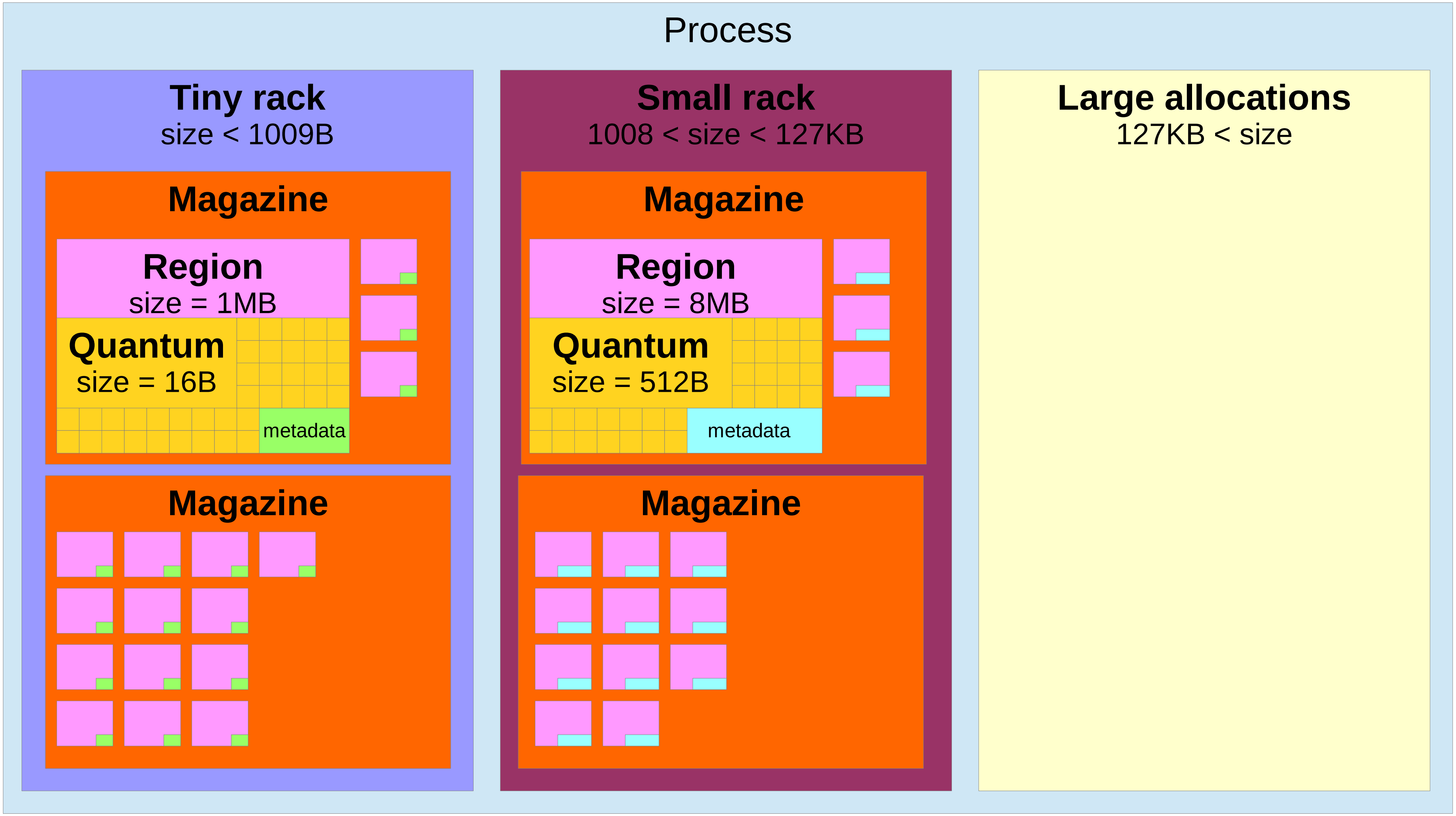
How does the scalable zone works



How does the scalable zone works

- **Each rack has one magazine per physical core**
  - optimize the processor caches accesses
  - reduce the risk of concurrent access (less locks)
- **Each magazine has multiple regions**
  - 1MB for tiny allocations
  - 8MB for small ones
  - metadata (rack specific) is at the end of the region
- **Each region is divided in quantum**
  - 16B for tiny allocations (64520 quantums / region)
  - 512B for small ones (16319 quantums / region)

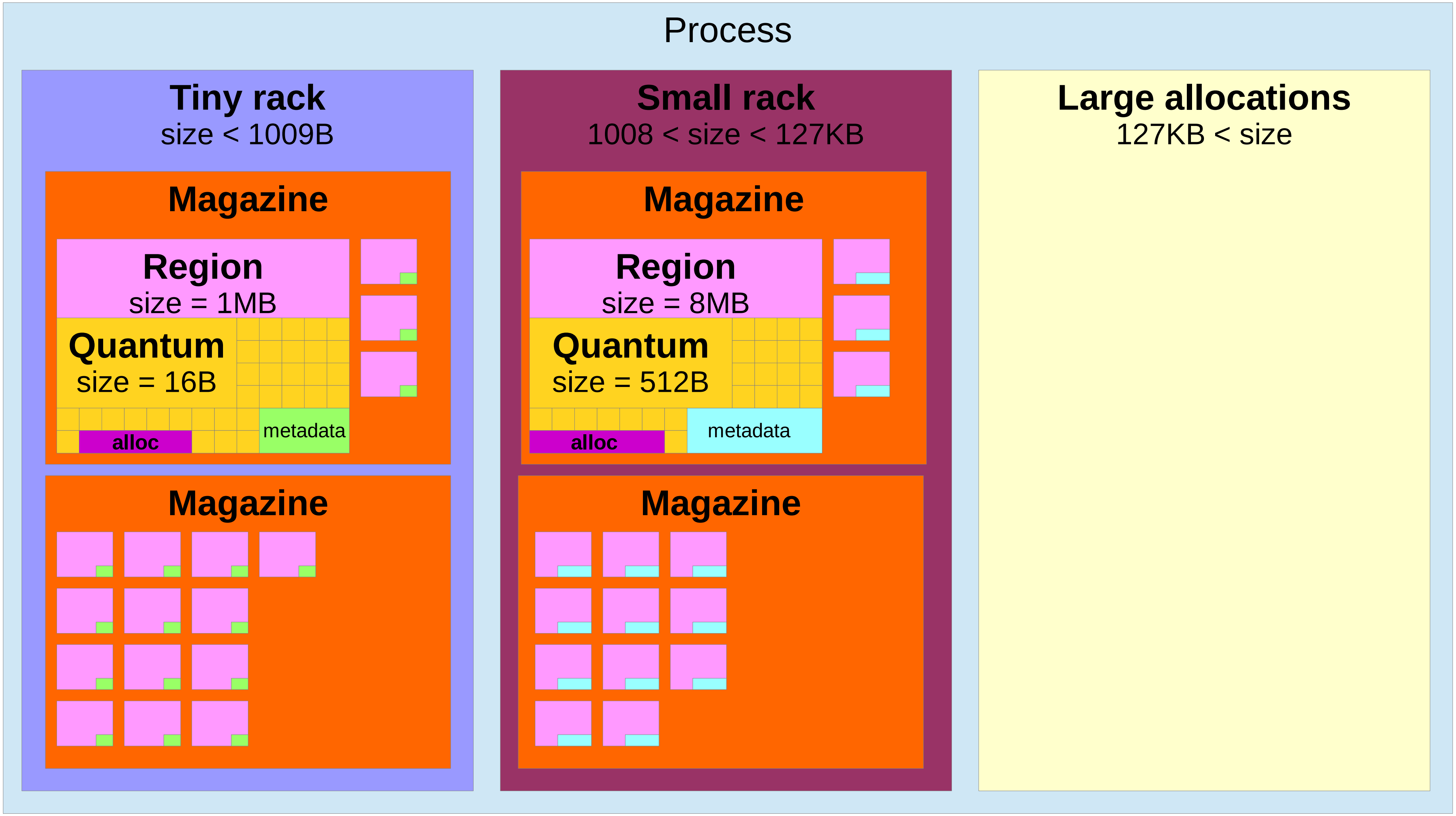
How does the scalable zone works



How does the scalable zone works

- **Each rack has one magazine per physical core**
  - optimize the processor caches accesses
  - reduce the risk of concurrent access (less locks)
- **Each magazine has multiple regions**
  - 1MB for tiny allocations
  - 8MB for small ones
  - metadata (rack specific) is at the end of the region
- **Each region is divided in quantum**
  - 16B for tiny allocations (64520 quantums / region)
  - 512B for small ones (16319 quantums / region)
- **An allocation is a block made of *n* quantums**
  - 31/63 max for tiny allocations depending on the arch (32bits/64bits)
  - 60/508 max for small allocations depending on the machine (less/more than 1GB of memory)

How does the scalable zone works

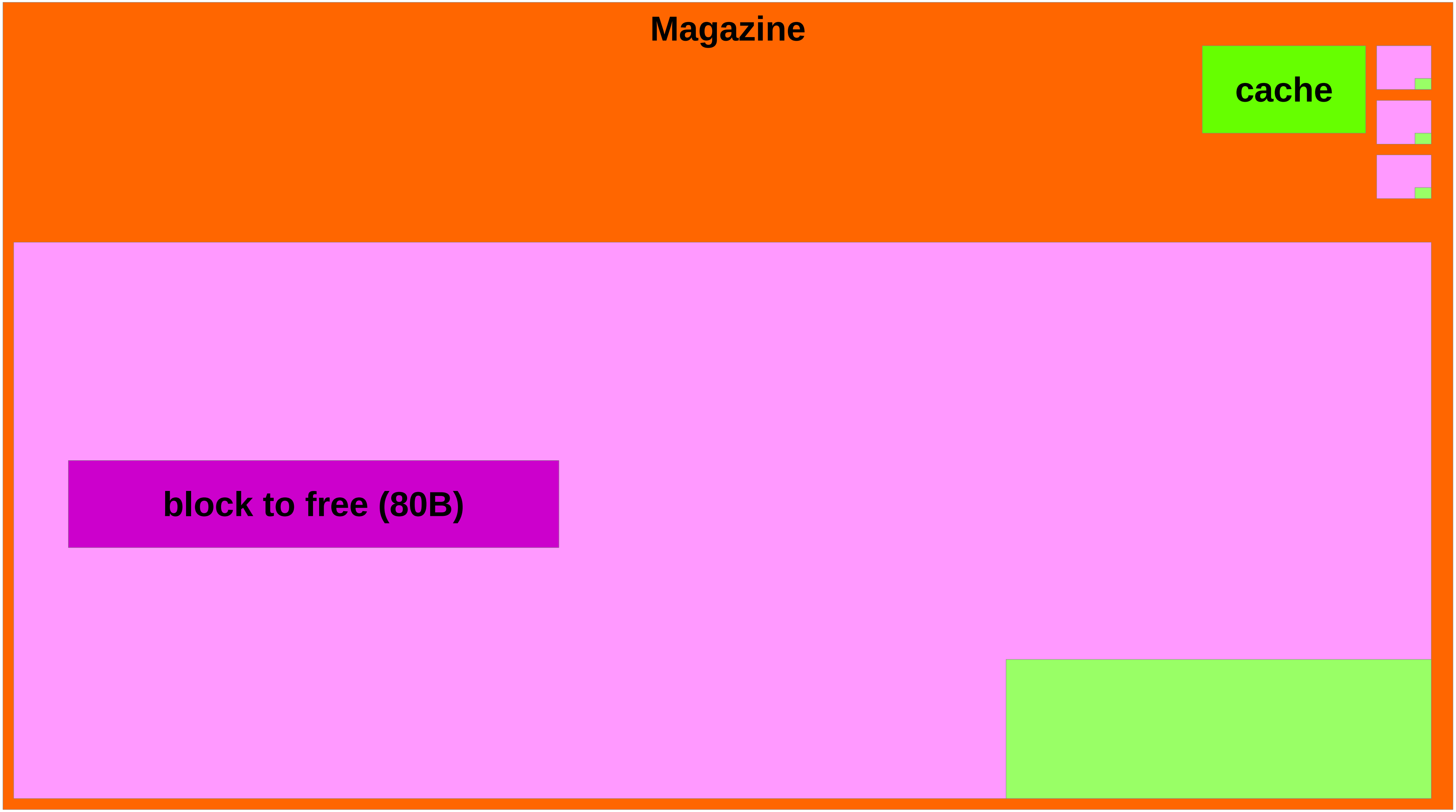


How does the scalable zone works

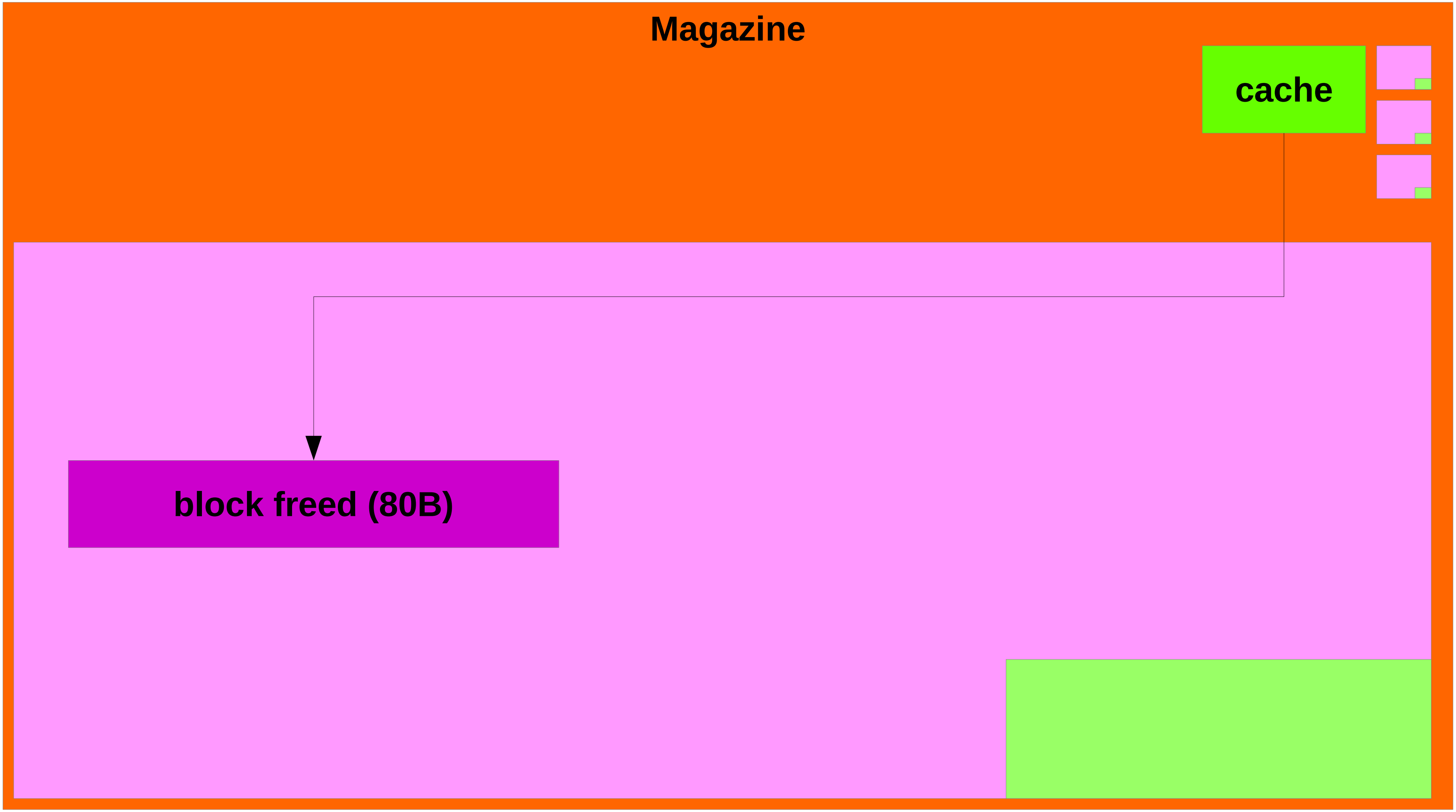
- **When an alloc is freed, the block is cached in the magazine**
  - for the tiny track, only if the block is not too big
  - because the number of quantums has to fit in 4 bits

    ⇨ size < 256
  - otherwise, we directly go to the next step...

How does the scalable zone works



How does the scalable zone works

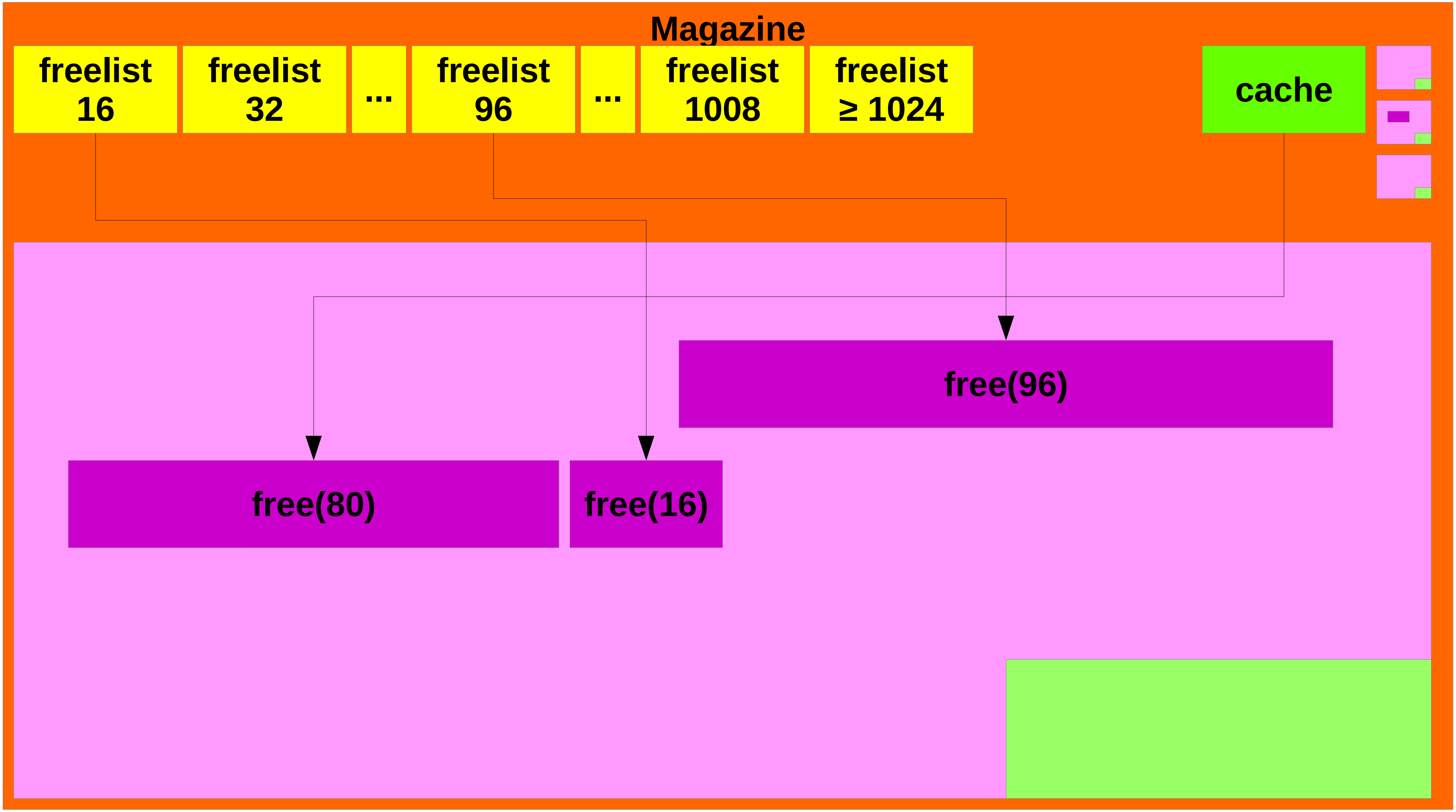


How does the scalable zone works

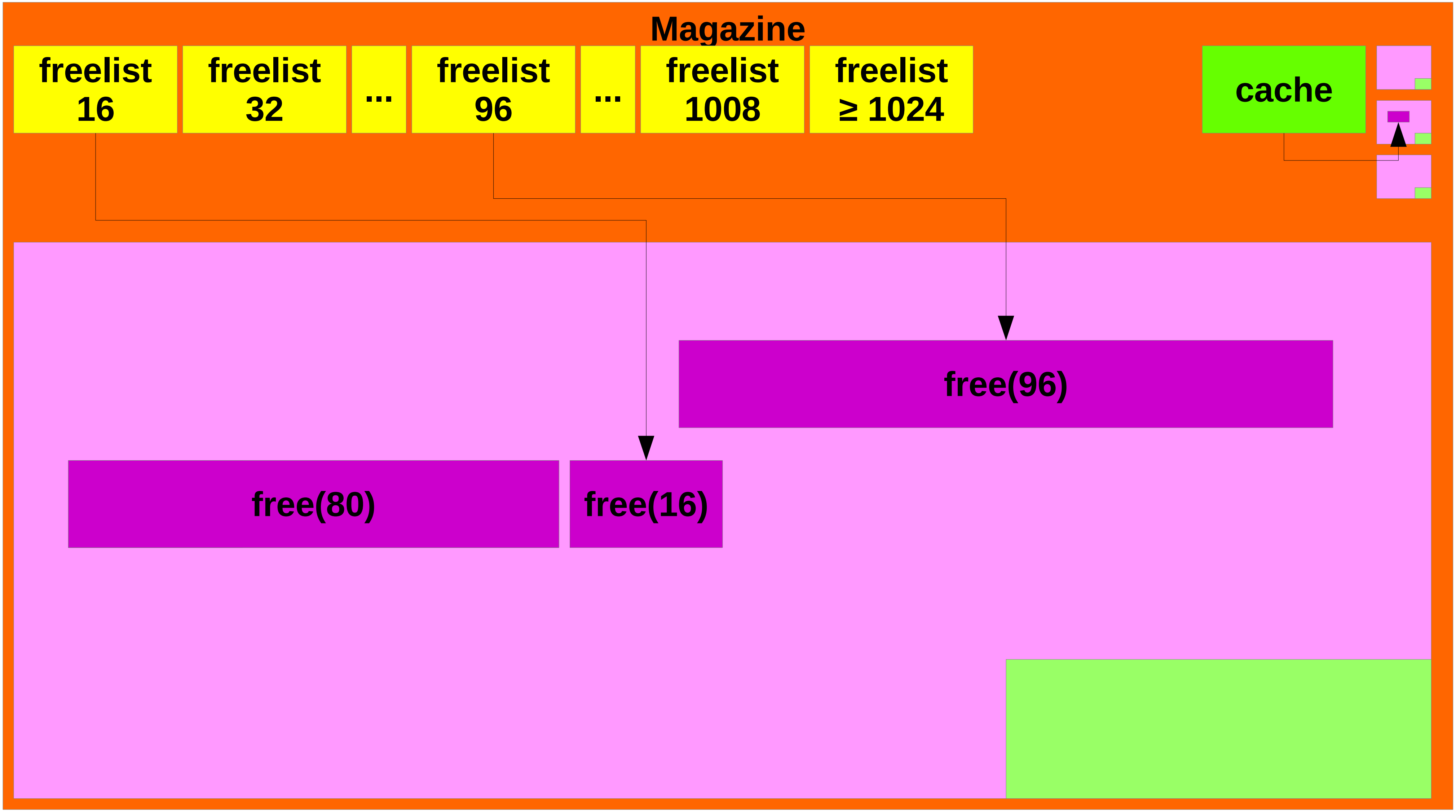
- **When an alloc is freed, the block is cached in the magazine**
  - for the tiny track, only if the block is not too big
  - because the number of quantums has to fit in 4 bits

    ⇨ size < 256
  - otherwise, we directly go to the next step...
- **The old cached one, if any, is freed**
  - It is first coalesced with adjacent free blocks if any

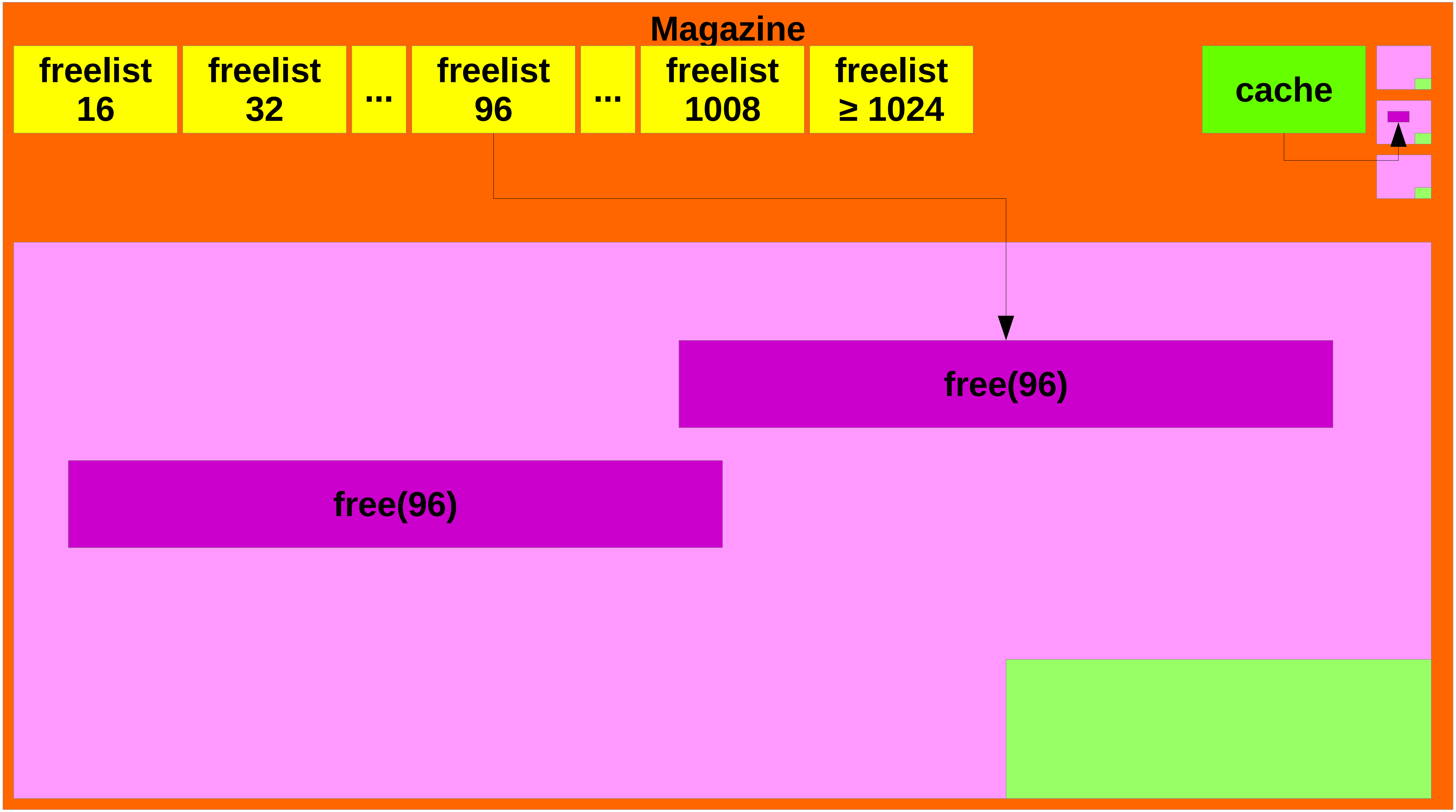
How does the scalable zone works



How does the scalable zone works



How does the scalable zone works

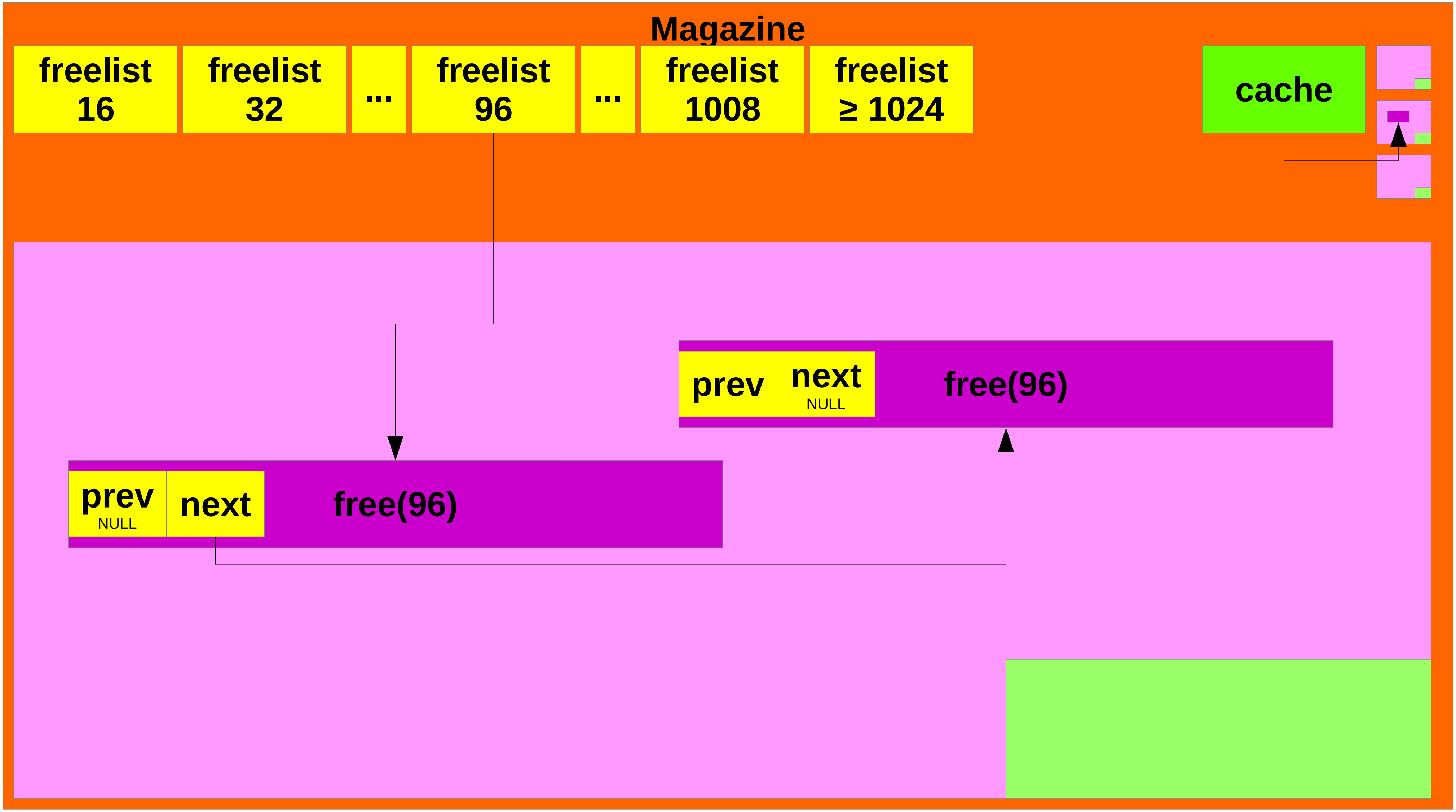


How does the scalable zone works

- **When an alloc is freed, the block is cached in the magazine**
  - for the tiny track, only if the block is not too big
  - because the number of quantums has to fit in 4 bits

    ⇨ size < 256
  - otherwise, we directly go to the next step...
- **The old cached one, if any, is freed**
  - It is first coalesced with adjacent free blocks if any
  - It is then put int the free list
  - Pointers are protected with a 4bit randomized checksum

How does the scalable zone works



How does the scalable zone works

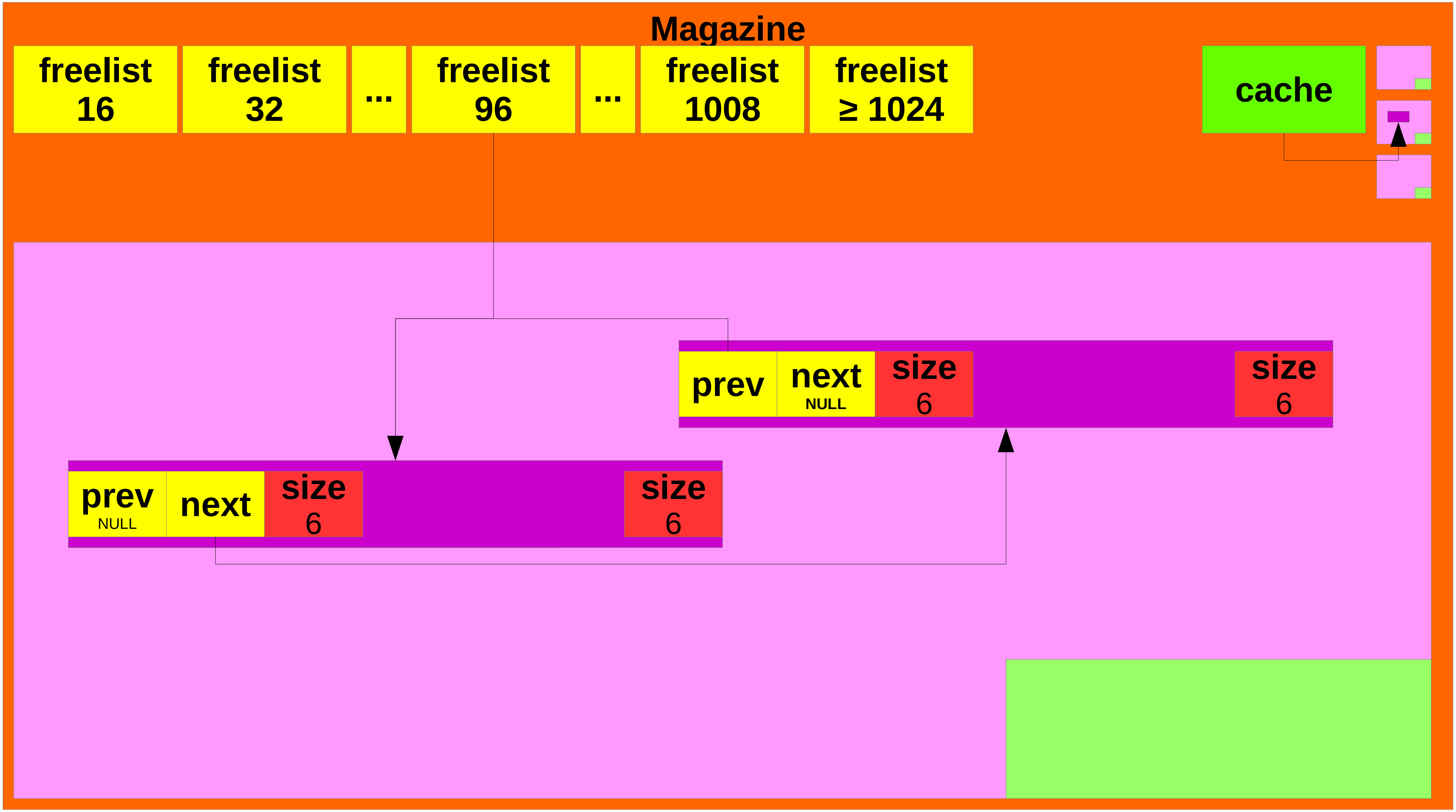
- **When an alloc is freed, the block is cached in the magazine**
  - for the tiny track, only if the block is not too big
  - because the number of quantums has to fit in 4 bits

    ⇨ size < 256
  - otherwise, the block is directly free
- **The old cached one, if any, is free**
  - It is first coalesced with adjacent free blocks if any
  - It is then put int the free list
  - Pointers are protected with a 4bit randomized checksum
  - For the tiny track, if it is big enough (≥ 16B), it also contains its size

    after the pointers and at the end of the allocation

    for the small track, the block size is stored in the metadata

How does the scalable zone works



How does the scalable zone works

- **When a block is allocated, malloc will try to:**
  - use the cache if the size matches
  - use a block in freelists[size]
  - use a larger block in freelists[size+n]

    the leftover is put in the freelist
  - use the end of the region

    which is not already allocated
  - allocate a new region
- **If everything fails, it returns NULL**

Important things to remember

1/2

- **One magazine per core**
  - Important when you massage/spray a multi thread process or when your exploit takes time…
- **To fill all the holes in the heap, just make a lot of tiny allocations**
- **Allocations are contiguous**
- **Allocations are not randomized**
  - Useful for massaging
- **Allocations of different sizes are in the same region**
  - Even if your UAF/overflow can only be triggered on a fixed size block you can hit a lot of different objects

Important things to remember
2/2

- **Last freed chunk is cached**
  - so not instantly coalesced!
- **Metadata in freed chunks is protected**
  - next and previous pointers are aligned on 16 bytes
  - *malloc* uses the 4 less significant bits to store a (randomized) checksum
  - rotate the result to place the checksum in the most significant bits

    unclear why… to protect against a partial overwrite?
- **If you want to know more, it's open-source**
  - https://opensource.apple.com/source/libmalloc/

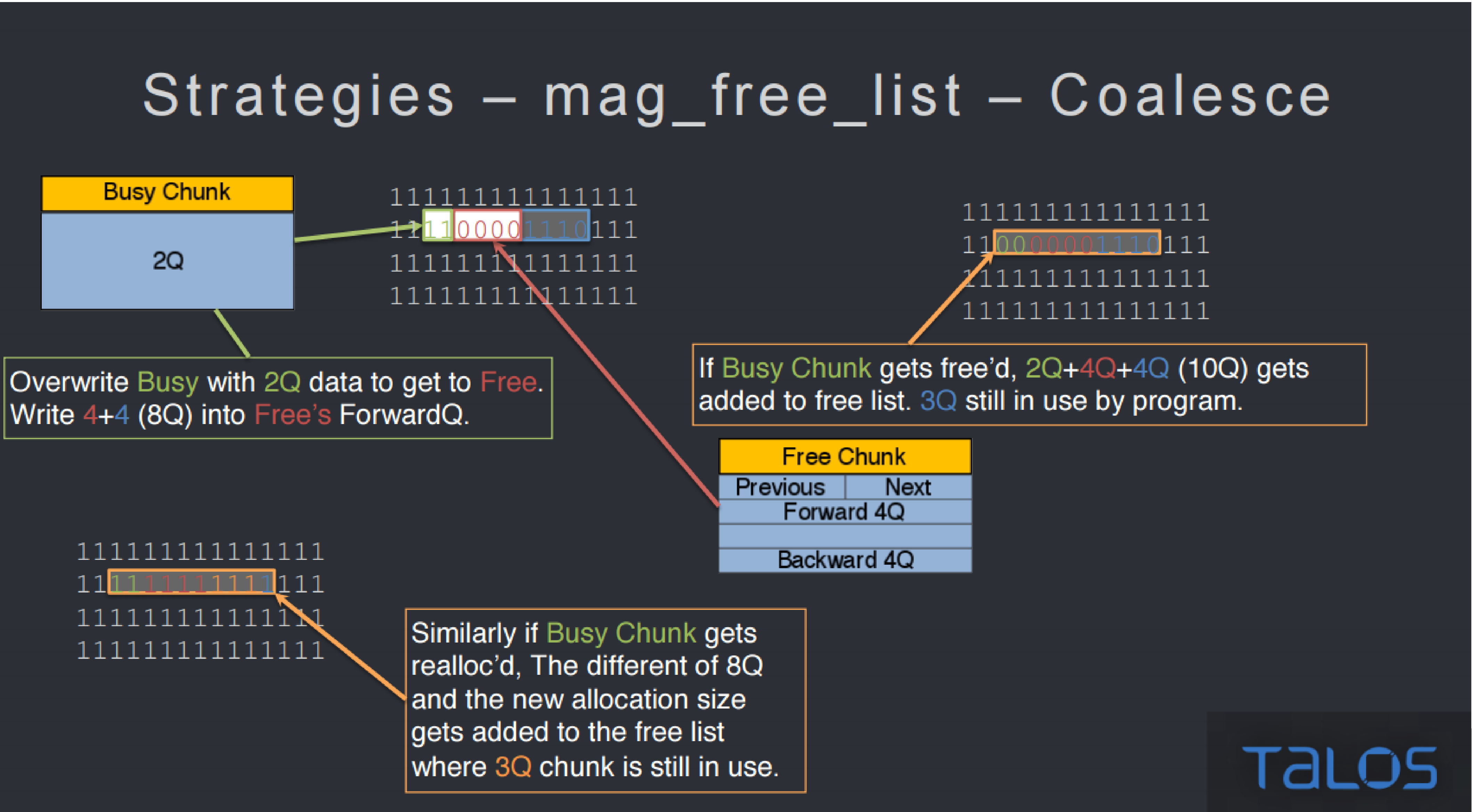
Exploit!

“In the Zone: OS X Heap Exploitation” techniques

- **Tries to transform a linear heap overflow in the tiny heap into a use-after-free alike primitive**
  - By overwriting freed blocks size
  - Couldn’t work in the small heap as sizes are in the metadata
- **Useful to leak pointers for example**

“In the Zone: OS X Heap Exploitation” techniques

src: PacSec 2016 – Tyler Bohan – https://pacsec.jp/psj16/PSJ2016_Bohan_PacSec_2016.pdf



Strategies – mag_free_list – Coalesce

```
111111111111111
111100001110111
111111111111111
111111111111111
```

Overwrite Busy with 2Q data to get to Free. Write 4+4 (8Q) into Free’s ForwardQ.

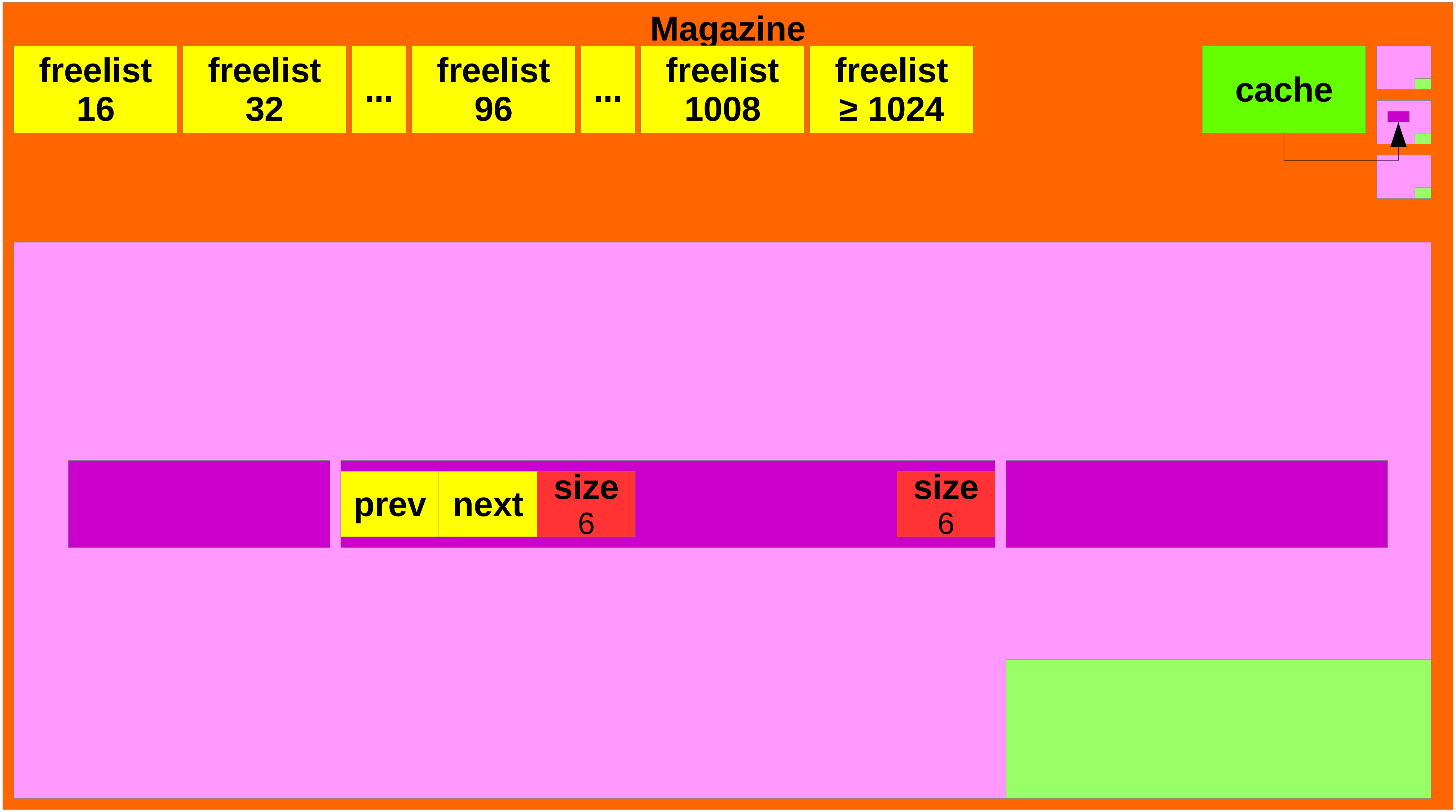
```
111111111111111
110000001110111
111111111111111
111111111111111
```

If Busy Chunk gets free’d, 2Q+4Q+4Q (10Q) gets added to free list. 3Q still in use by program.

```
111111111111111
111111111111111
111111111111111
111111111111111
```

Similarly if Busy Chunk gets realloc’d, The different of 8Q and the new allocation size gets added to the free list where 3Q chunk is still in use.

How does the scalable zone works



Magazine

freelist 16

freelist 32

...

freelist 96

...

freelist 1008

freelist ≥ 1024

cache

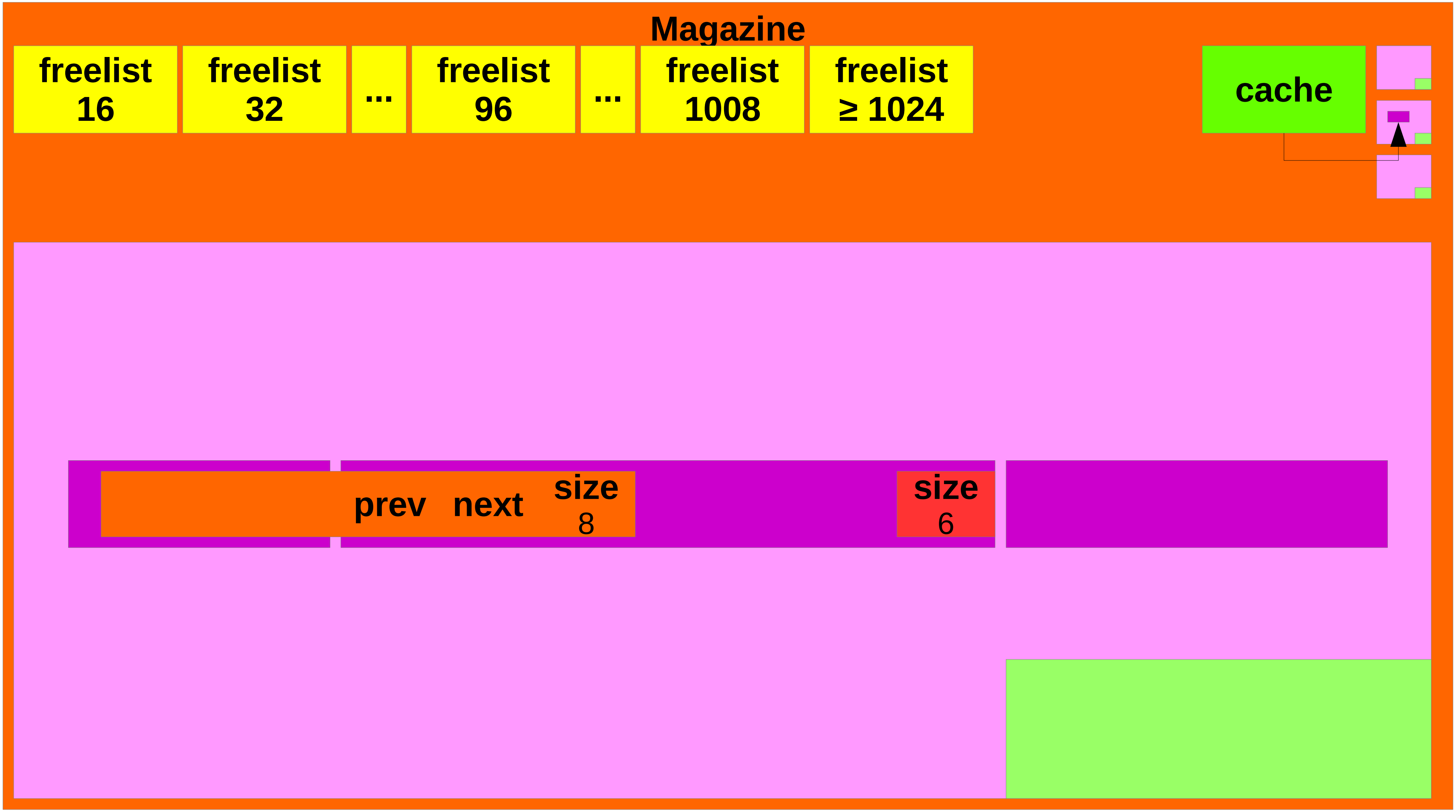
prev

next

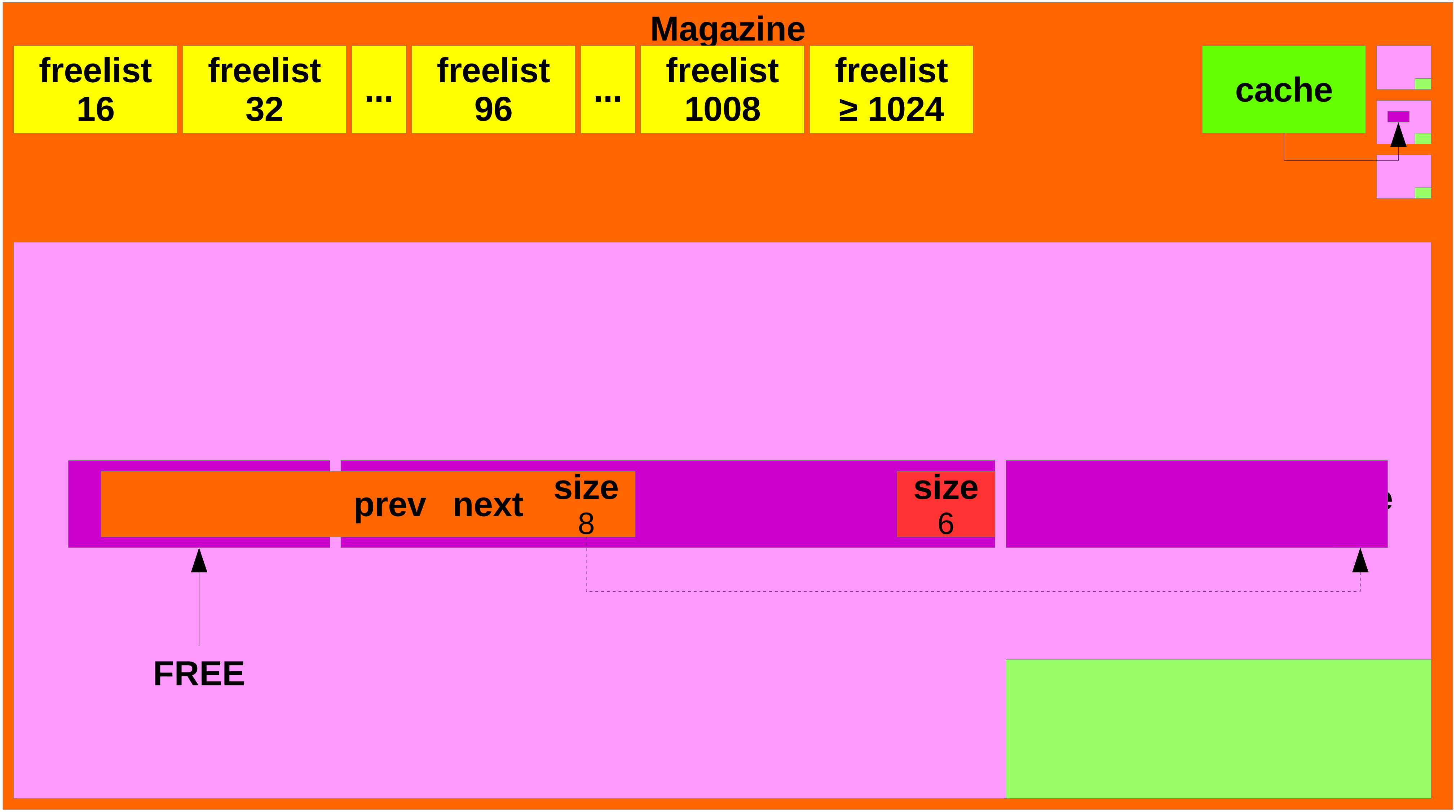
size 6

size 6

How does the scalable zone works



How does the scalable zone works



Magazine

freelist 16

freelist 32

...

freelist 96

...

freelist 1008

freelist ≥ 1024

cache

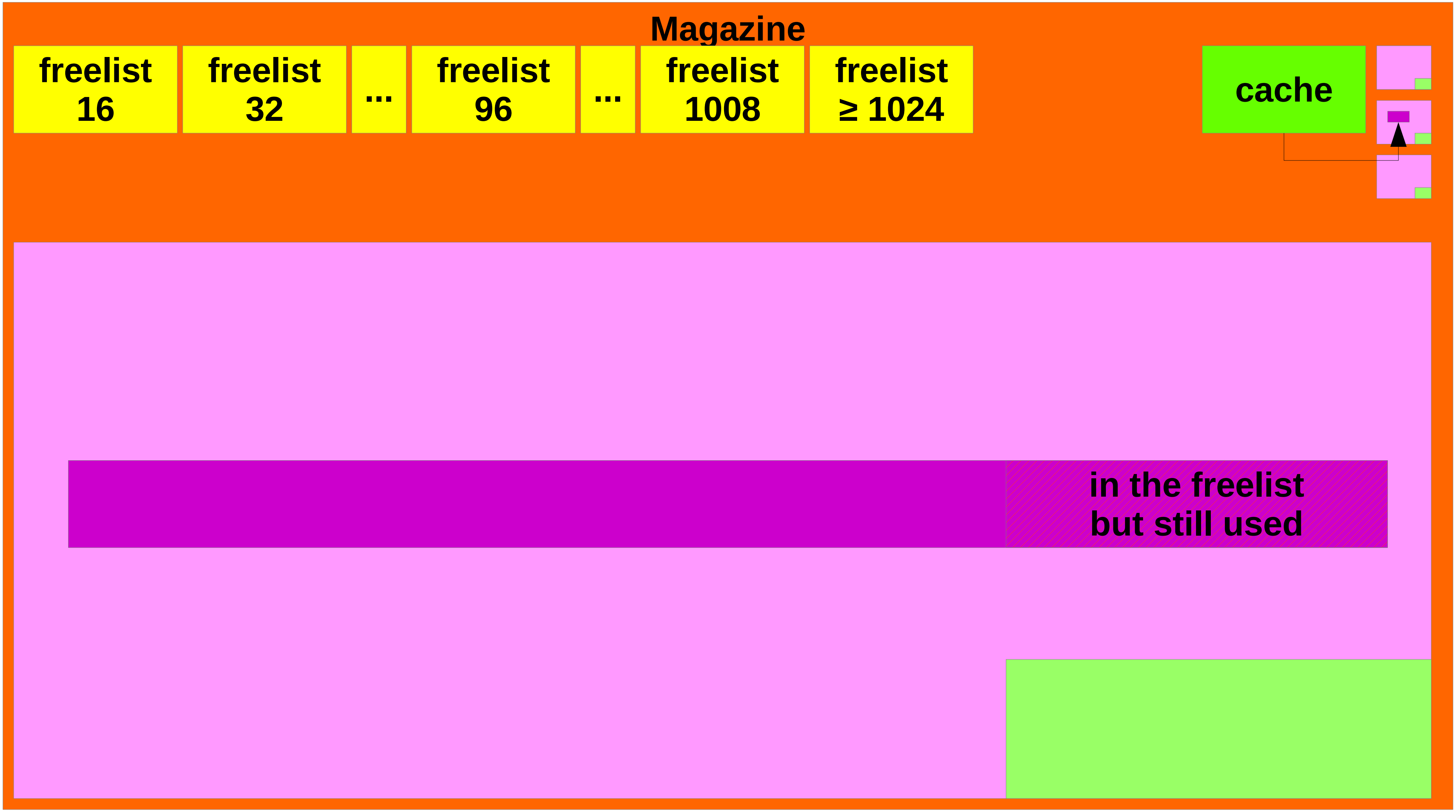
prev next

size 8

size 6

FREE

How does the scalable zone works



“In the Zone: OS X Heap Exploitation” techniques

- **Actually never worked**
  - You cannot overflow the size of a chunk without overflowing its pointers
  - Pointers are checked during coalescing

    when the coalesced block is removed from its previous free list

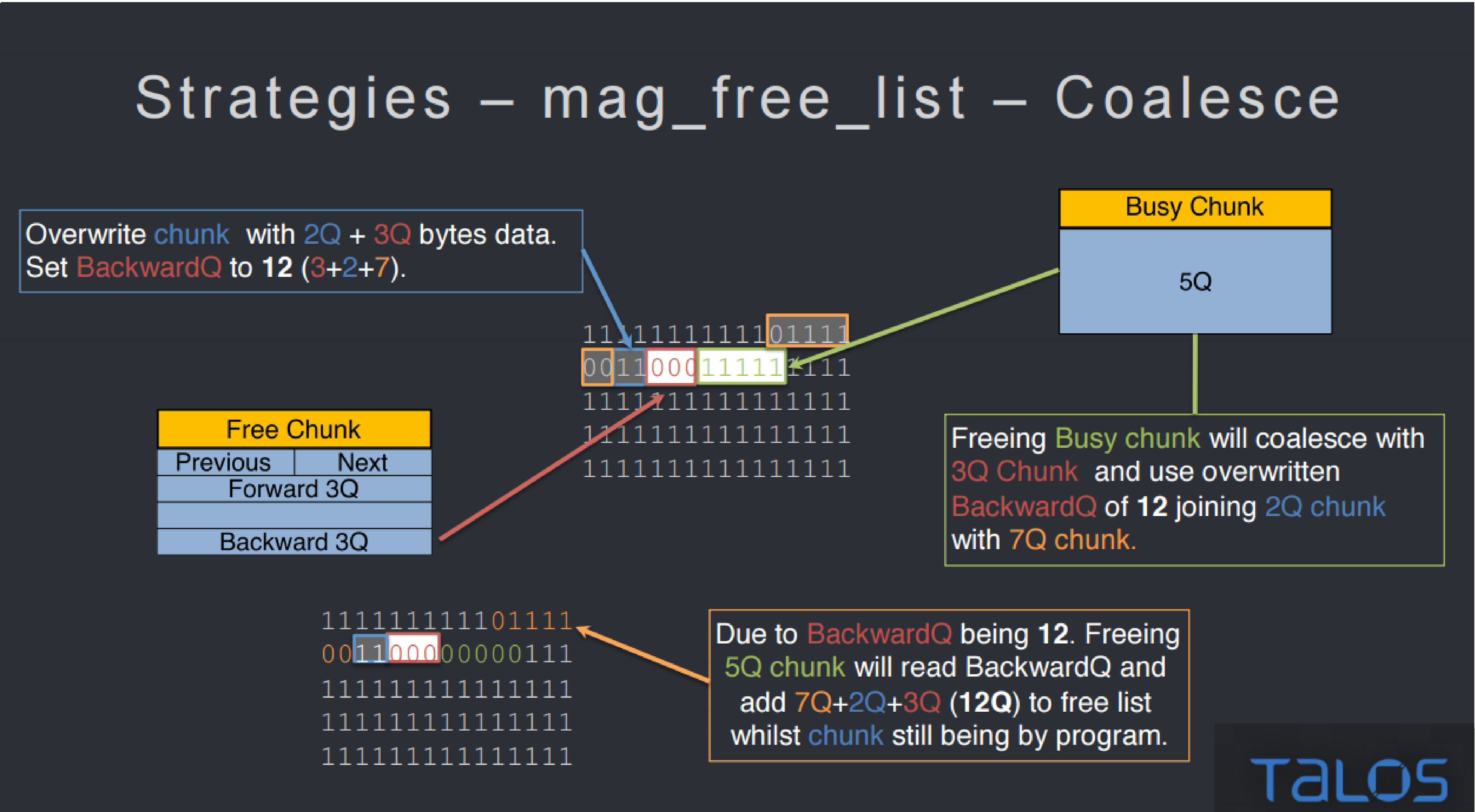
    see *tiny_free_list_remove_ptr* and *free_list_unchecksum_ptr* in *tiny_free_no_lock*
  - Without a leak (or a lot of luck) you are toasted
- **Trick applicable only if you have a non-linear OOB write**
  - So you can overwrite size without overwriting the pointers
  - For example an indexed write with an attacker chosen index
- **Fortunately, another technique is proposed…**

“In the Zone: OS X Heap Exploitation” techniques

- **You may think that you can trick the allocator by using backward coalescing**
  - the heap will then use the unmodified pointers of another preceding allocation
  - checksum bypassed!

“In the Zone: OS X Heap Exploitation” techniques

src: PacSec 2016 – Tyler Bohan – https://pacsec.jp/psj16/PSJ2016_Bohan_PacSec_2016.pdf



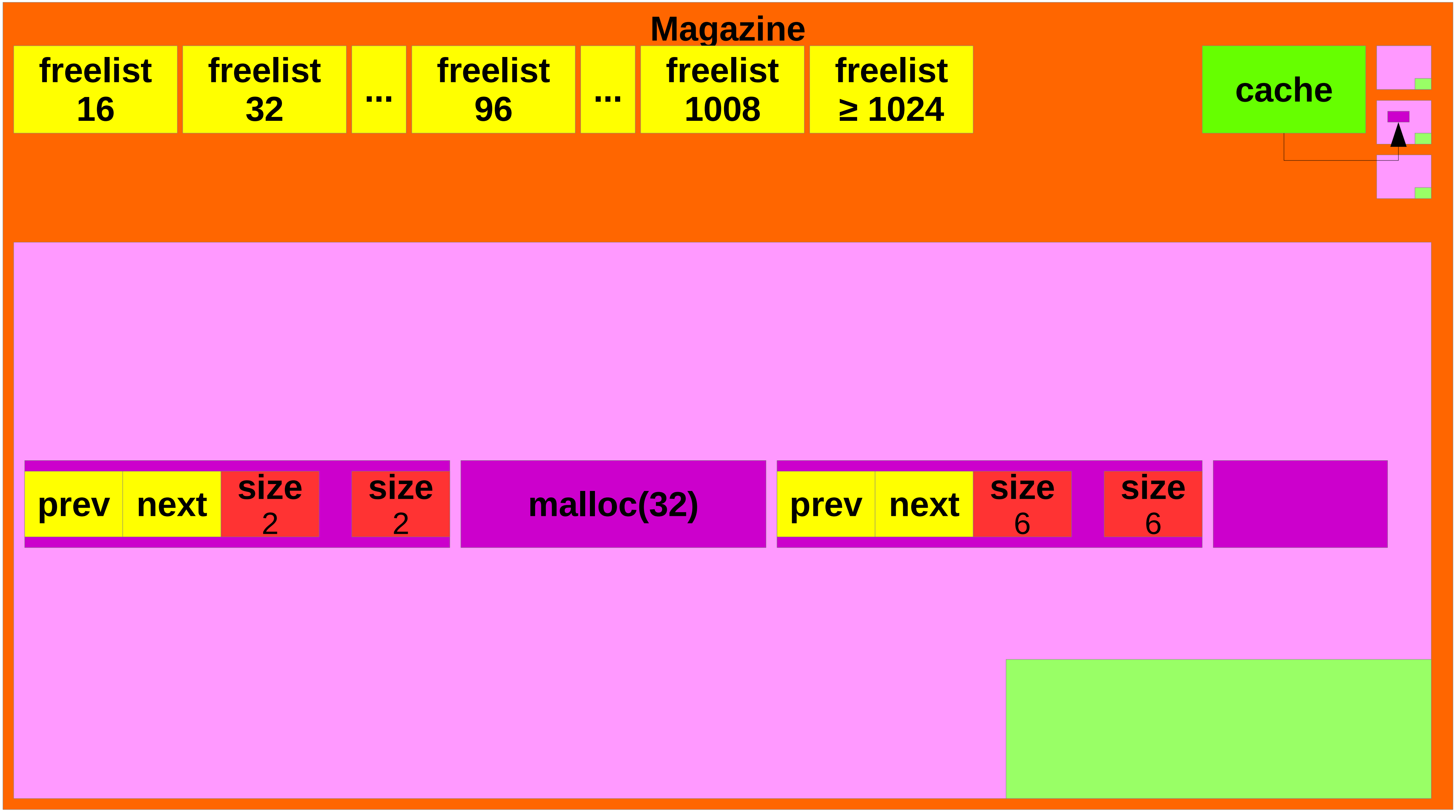
Strategies – mag_free_list – Coalesce

Overwrite chunk with 2Q + 3Q bytes data.
Set BackwardQ to **12** (3+2+7).

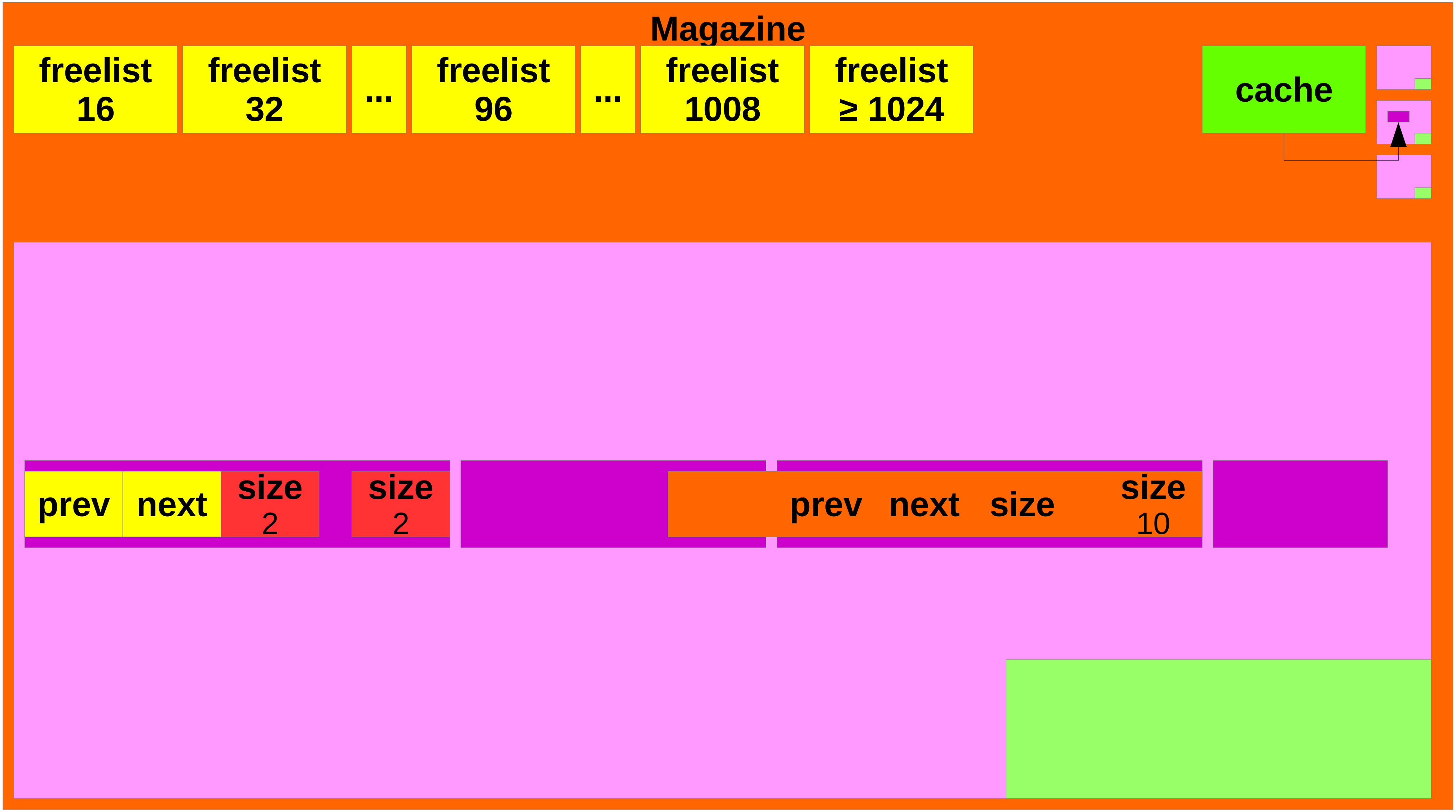
Freeing Busy chunk will coalesce with 3Q Chunk and use overwritten BackwardQ of **12** joining 2Q chunk with 7Q chunk.

Due to BackwardQ being **12**. Freeing 5Q chunk will read BackwardQ and add 7Q+2Q+3Q (**12Q**) to free list whilst chunk still being by program.

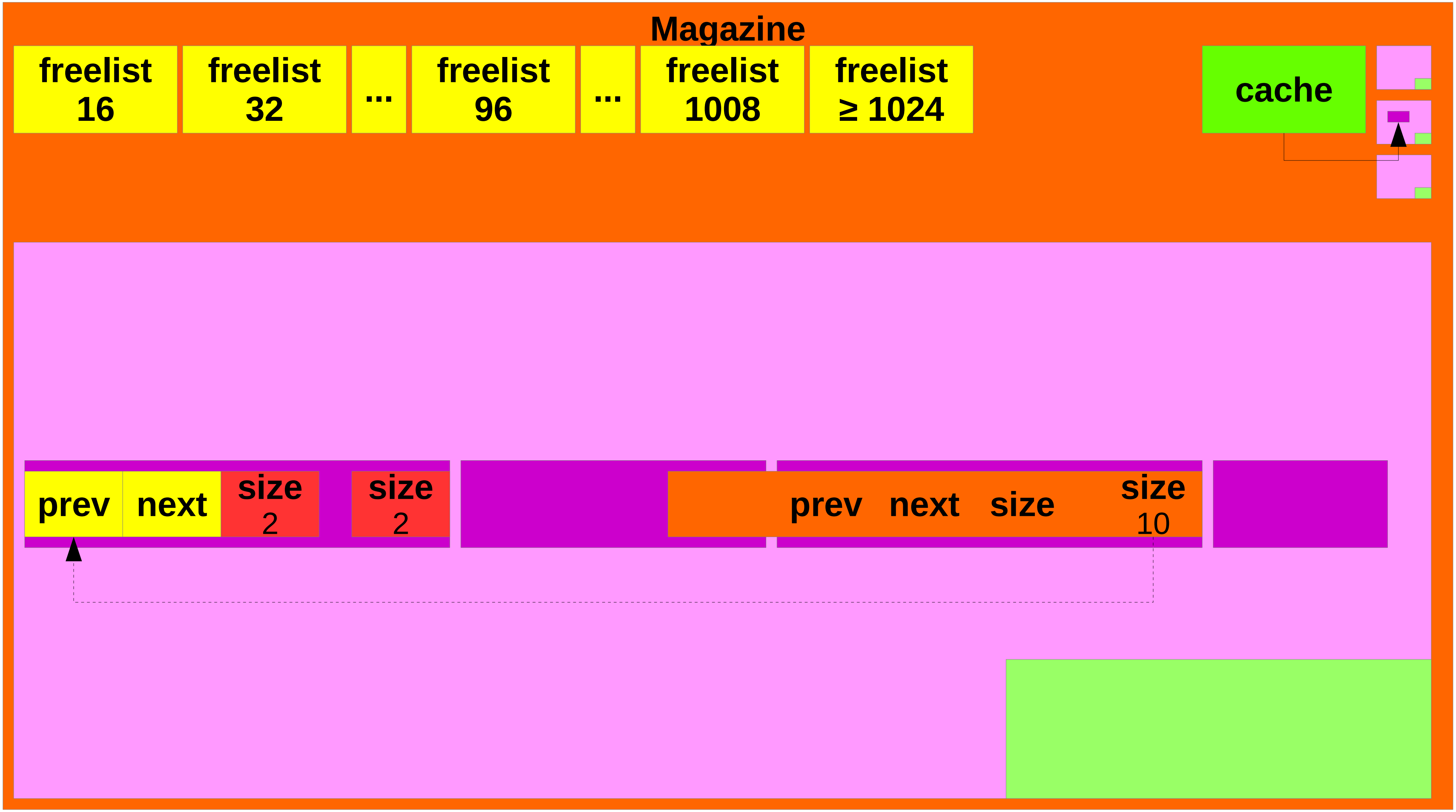
How does the scalable zone works



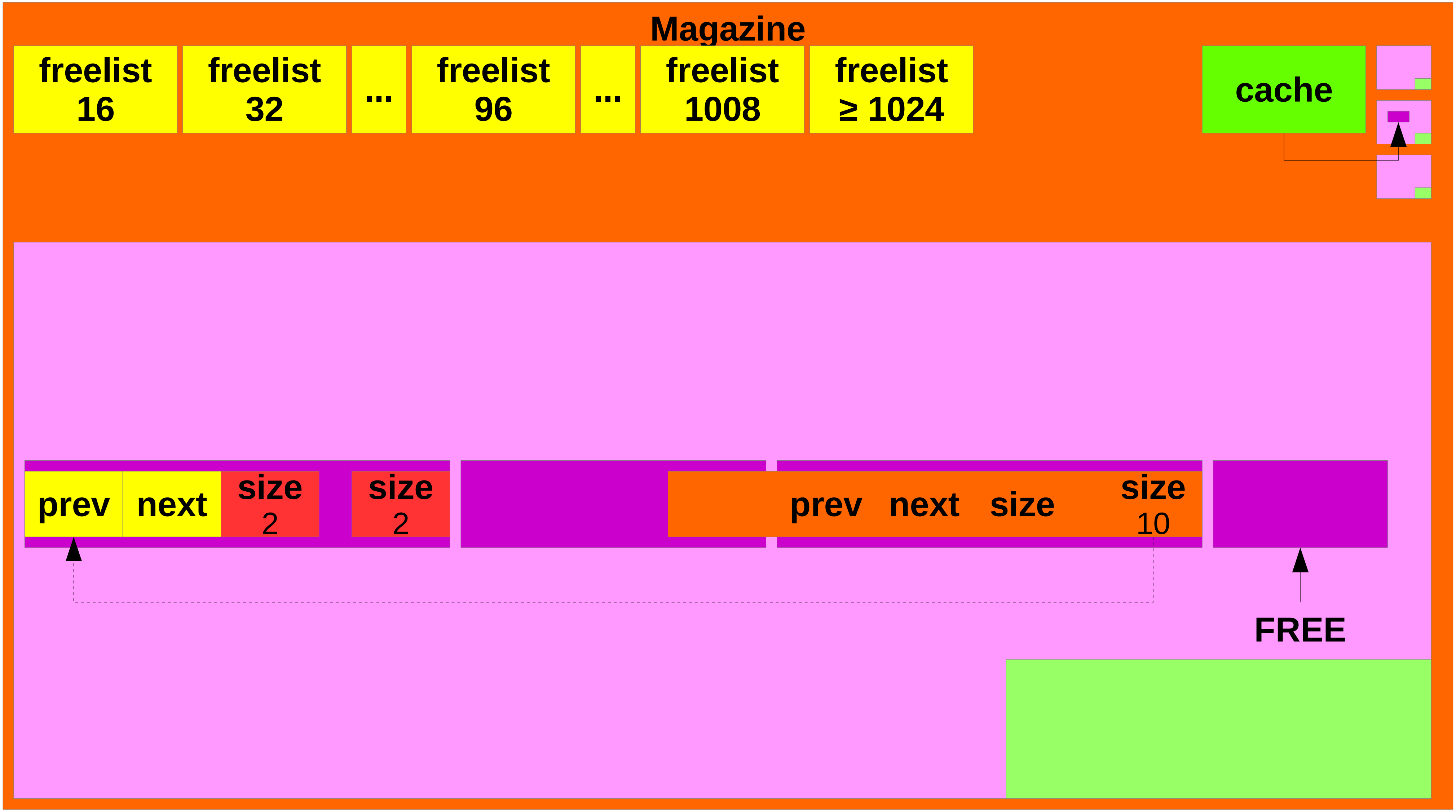
How does the scalable zone works



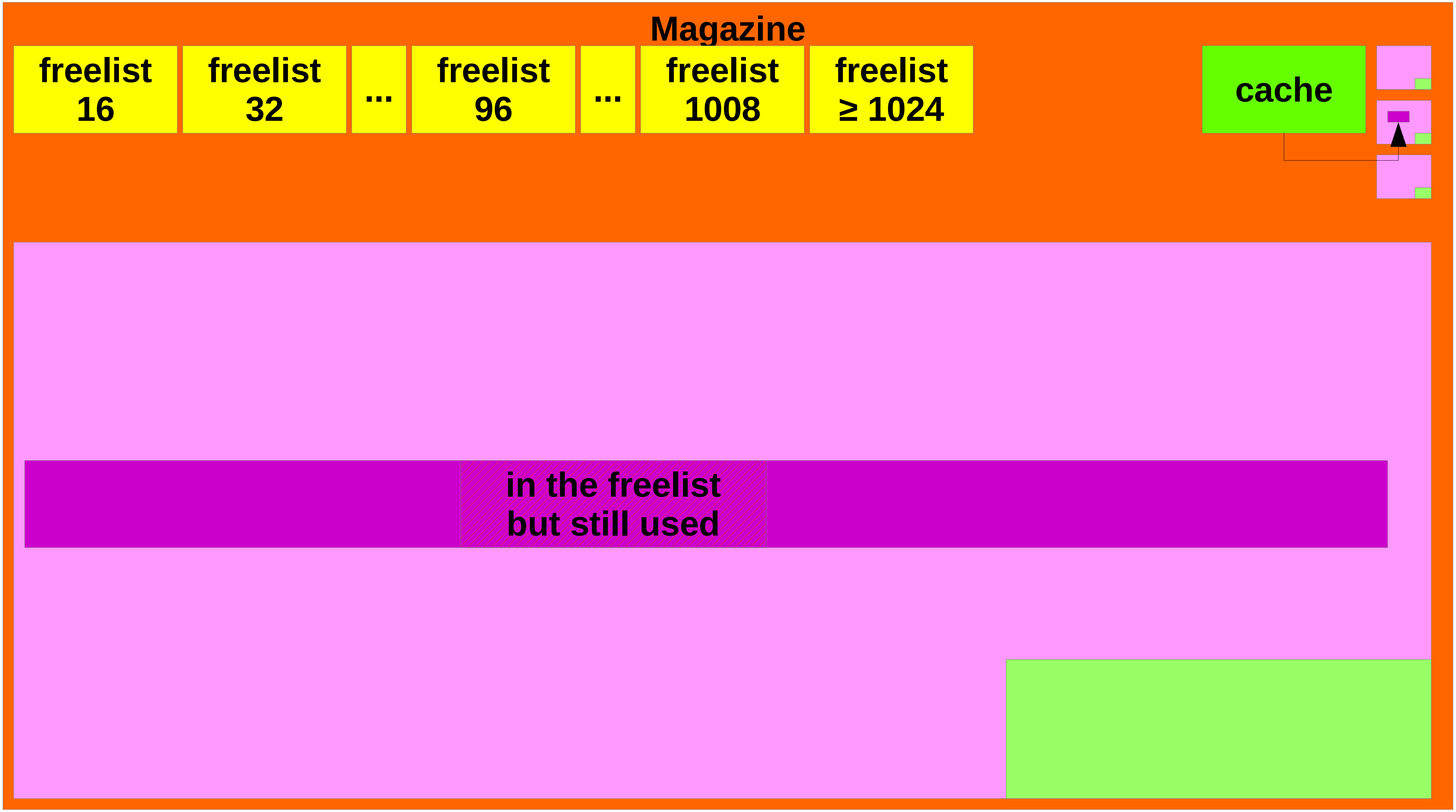
How does the scalable zone works



How does the scalable zone works



How does the scalable zone works



Magazine

freelist 16

freelist 32

...

freelist 96

...

freelist 1008

freelist ≥ 1024

cache

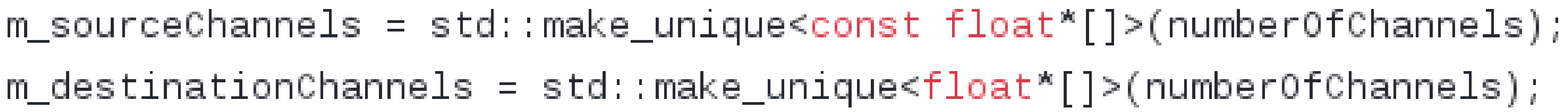
in the freelist but still used

“In the Zone: OS X Heap Exploitation” techniques

- **You may think that you can trick the allocator by using backward coalescing**
  - the heap will then use the unmodified pointers of another preceding allocation
  - checksum bypassed!
  - but...
- **If the size stored at the beginning and the end of the freed block doesn’t match then no coalescing is done**
  - actually not a security check
  - the allocator first assumes that the preceding block is freed because it cannot directly check if it’s freed
  - then it checks if it is effectively freed
  - see *tiny_previous_preceding_free* in *tiny_free_no_lock*
- **This check exists since the first magazine malloc version**
  - both techniques never worked

“In the Zone: OS X Heap Exploitation” techniques

- **Use the Web Audio API in WebKit to massage the default heap**
  - in *WebCore/Modules/webaudio/AudioBufferSourceNode.cpp*:

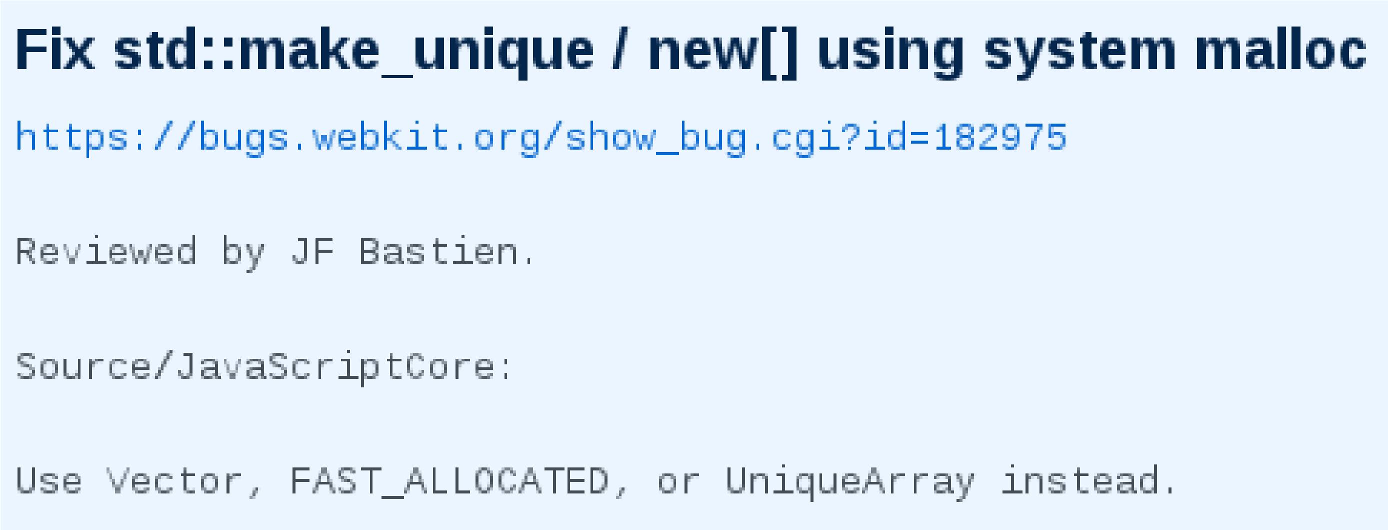


```
m_sourceChannels = std::make_unique<const float*[]>(numberOfChannels);
m_destinationChannels = std::make_unique<float*[]>(numberOfChannels);
```

- **std → allocate in the default heap**
- ***numberOfChannels* is controlled**
  - 1 to 32 channels
- **previous buffers are freed**
- **(almost) perfect to massage the heap!**
  - you cannot free a block without allocating another one
  - needs some gymnastic to make it works
  - but no garbage collection problems!

“In the Zone: OS X Heap Exploitation” techniques

- **Until commit 1d211e1fc1cf4801da64b6881d07bda01f643cf3…**
  - March 2018



Fix std::make_unique / new[] using system malloc

```
https://bugs.webkit.org/show_bug.cgi?id=182975

Reviewed by JF Bastien.

Source/JavaScriptCore:

Use Vector, FAST_ALLOCATED, or UniqueArray instead.
```

- **Removes almost all references to the default heap in WebKit**
  - technique is dead

What’s left?

- **Not much :)**
- **You may try to attack metadata at the end of a region**
  - but that’s another story…
- **You may try to attack adjacent allocations**
  - to overflow pointers, lengths, vtables…
  - or Objective-C objects

    see *Modern Objective-C Exploitation Techniques* in Phrack #69 by nemo
- **Heap layout makes this relatively easy**
  - remember: objects of different size are all allocated in the same region / page

How to debug the heap?

- **Apple gives us powerful tools**
- **Environment variables (extract of the malloc man)**
  - *MallocGuardEdges*

    to add 2 guard pages for each large block
  - *MallocStackLogging*

    to record all stacks.
  - *MallocScribble*

    to detect writing on free blocks and missing initializers: 0x55 is written upon free and 0xaa is written on allocation
  - *MallocCheckHeapStart <n>*

    to start checking the heap after <*n*> operations
  - *MallocCheckHeapEach <s>*

    to repeat the checking of the heap after <*s*> operations
  - *MallocTracing*

    to emit kdebug trace points on malloc entry points

How to debug the heap? – cont’d

- **heap**
  - displays all the allocations of a given process
  - able to recognize Obj-C and C++ objects

    ex: `heap --addresses '(WebKit::WebFormClient|CFString)' Safari`
- **malloc_history**
  - displays the information gathered via the *MallocStackLogging* environment variable
- **leak**
  - used to discover leaks…
  - not really interesting from an exploitation point of view

Anything more visual?

- ***malloc_history* is great to get information on specific addresses**
  - useful for bug triage / debug
- **But it doesn’t give you an overview of the heap**
  - hard to test or validate heap massaging techniques
- **Moreover *MallocStackLogging* is quite slow…**

➔ **We need to go deeper!**

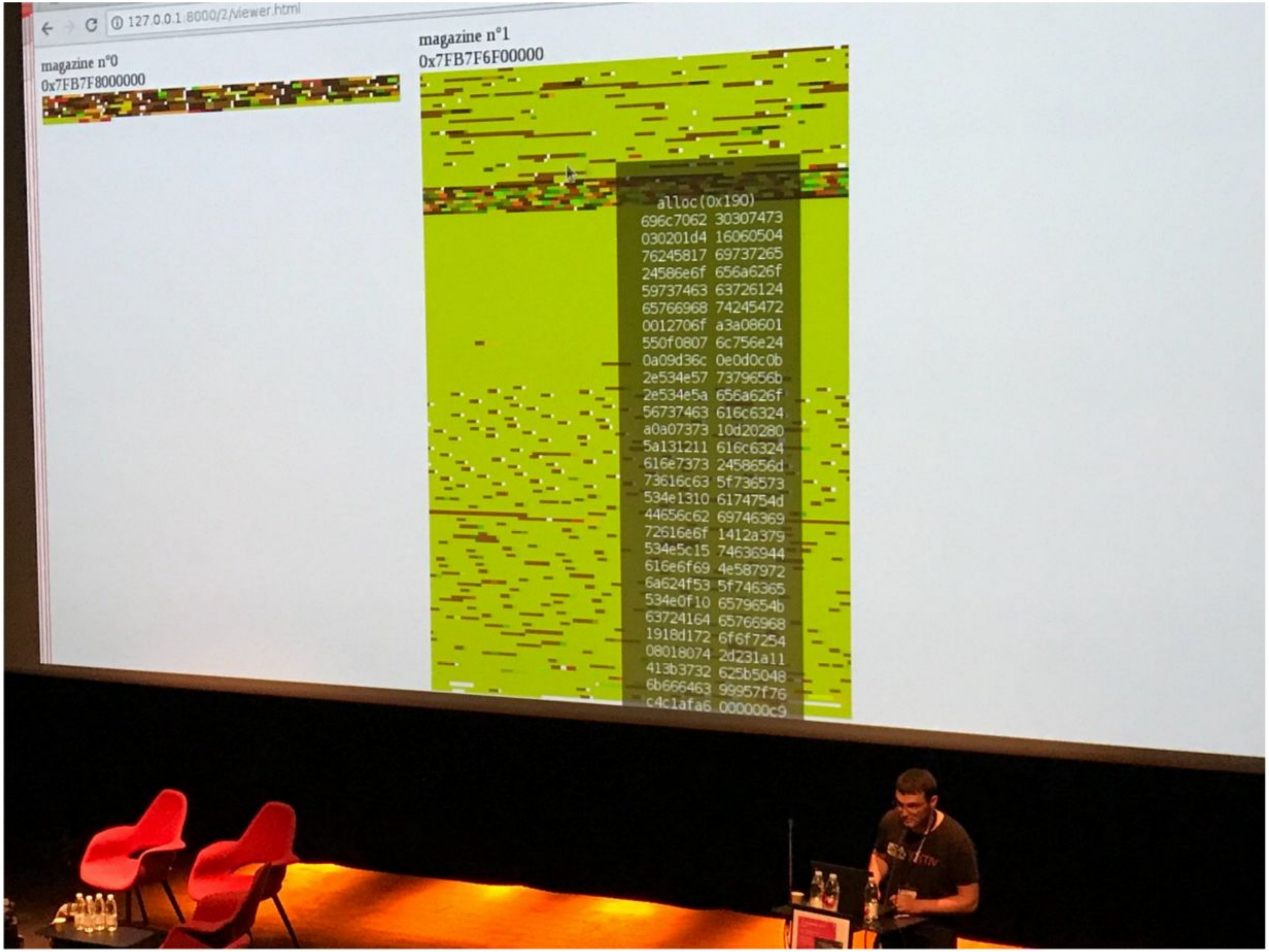
Remember the zones?

- **Zones must expose some functions**
  - see the definition of *malloc_zone_t* in *malloc/malloc.h*
- **Including introspection functions**
  - see *struct malloc_introspection_t*
- **Can be used to list both your own and other processes allocations**
  - functions take a pointer to a *reader* function
- **Not all zones implement it correctly…**
  - but the default zone does!

Visualizing

- **Blocks that start with the same *qword* have the same color**
  - Obj-C and C++ instances of a given object will have the same color
- **Do not use PIL and other Python imaging libraries**
  - try to do smart things like scaling your rectangles
  - rounding problems so not pixel perfect…
  - very slow
- **We developed a minimal python PNG lib**
  - based on *lodepng* (simple PNG C library, 1 file)
  - can only draw rectangles
  - but do it well and fast!
- **Interaction with HTML/JS**
  - Displays the PNG
  - Displays the data on click
  - Simple but efficient

Démo



```
alloc(0x190)
696c7062 30307473
030201d4 16060504
76245817 69737265
24586e6f 656a626f
59737463 63726124
65766968 74245472
0012706f a3a08601
550f0807 6c756e24
0a09d36c 0e0d0c0b
2e534e57 7379656b
2e534e5a 656a626f
56737463 616c6324
a0a07373 10d20280
5a131211 616c6324
616e7373 2458656d
73616c63 5f736573
534e1310 6174754d
44656c62 69746369
72616e6f 1412a379
534e5c15 74636944
616e6f69 4e587972
6a624f53 5f746365
534e0f10 6579654b
63724164 65766968
1918d172 6f6f7254
08018074 2d231a11
413b3732 625b5048
6b666463 99957f76
c4c1afa6 000000c9
```

Conclusion

- **No generic method**
  - sorry :)
- **But an attacker-friendly heap**
  - adjacent allocations
  - easy to massage
  - different sizes in the same region
  - no randomization
- **And a great introspection API**

Thank you!

- **Sthack for the amazing event**
  - can’t wait for tonight ;)
- **Synacktiv for the cool missions :)**
  - Did I say that we are recruiting?
- **SzLam for the presentation title idea**
  - ❤❤❤
- **You for your attention!**

Do you have any questions?

THANK YOU FOR YOUR ATTENTION

SYNACKTIV
DIGITAL SECURITY