

Kernel Memory Corruption Debug

Based On SLAB Implementation

Oliver Yang

Jan 15, 2014

<http://oliveryang.net>



Agenda

- Basic concepts
- Debugging methods
- Case study
- Potential improvements



Agenda

- Basic concepts
- Debugging methods
- Case study
- Potential improvements



Memory corruption causes

- Hardware bugs
 - X86 Machine Check errors
 - CPU
 - DIMM
 - QPI
 - PCIe errors
- Some legacy or low end x86 box's RAS protection had the big gaps
 - Lots of CFDs caused by DIMM UEs and CPU errors
 - Platform SEL logs might not have debug information



Memory corruption causes

- Software bugs
 - Using uninitialized memory
 - Wild pointer
 - Use after free
 - Using un-owned or beyond allocated memory
 - Buffer overflow
 - Stack overflow
 - Race conditions on memory modifications
 - Faulty heap memory management
 - Double free

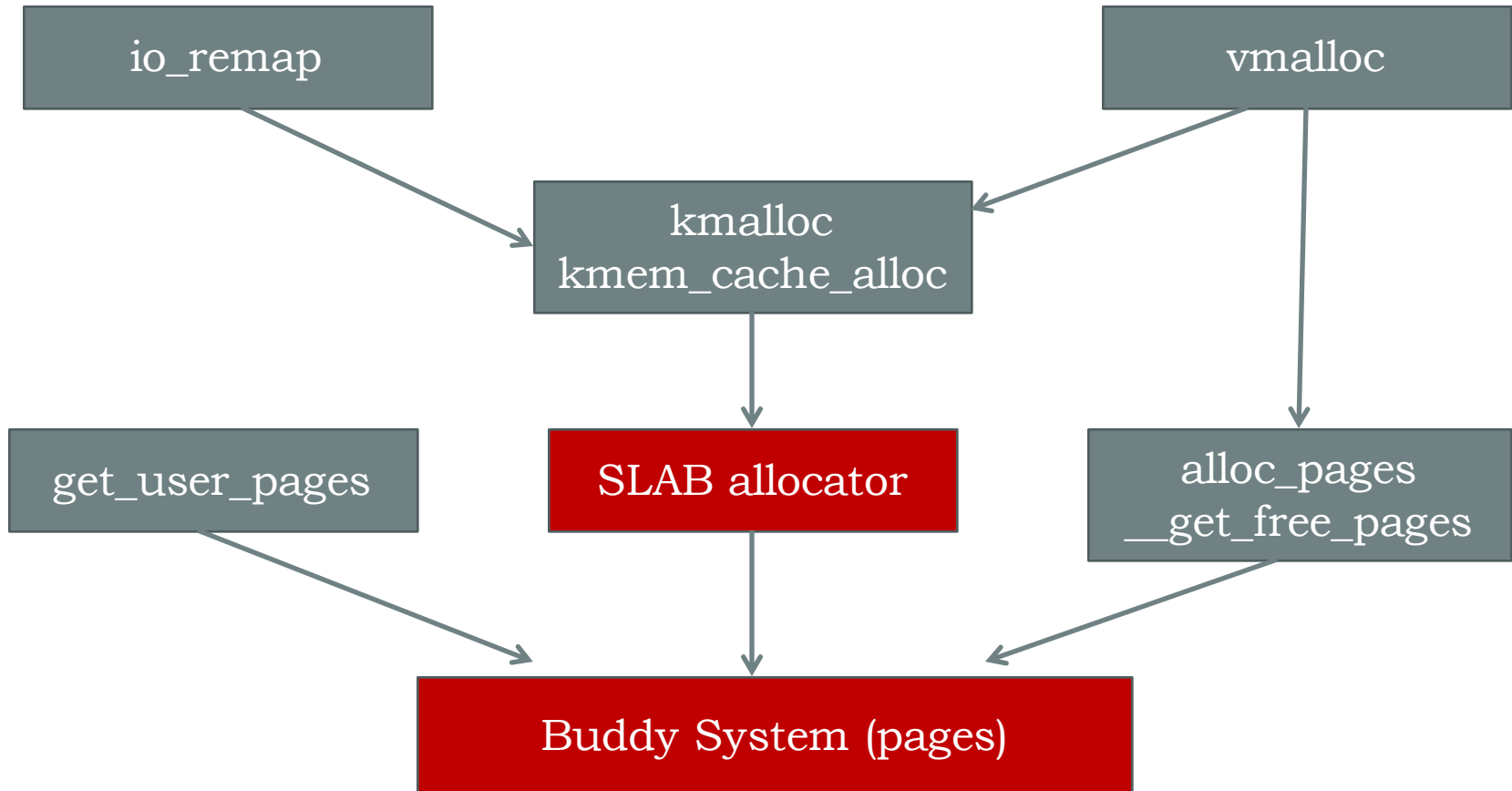


Challenges

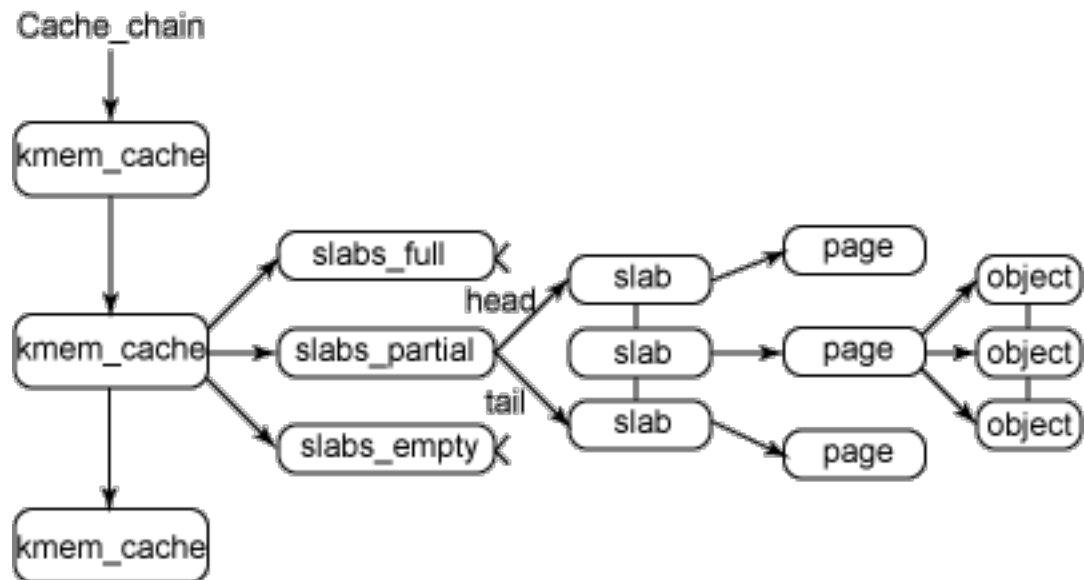
- Memory corruption: one of most intractable issues
 - Hard to get root causes by a core dump file
 - The source of the memory corruption and its manifestation may be far apart, making it hard to correlate the cause and the effect.
 - Difficult to reproduce
 - Symptoms appear under unusual conditions, making it hard to consistently reproduce the error
- Kernel memory debugging is more difficulty
 - Difficult to triage due to cross component boundary
 - Lack of debugging facilities and tools



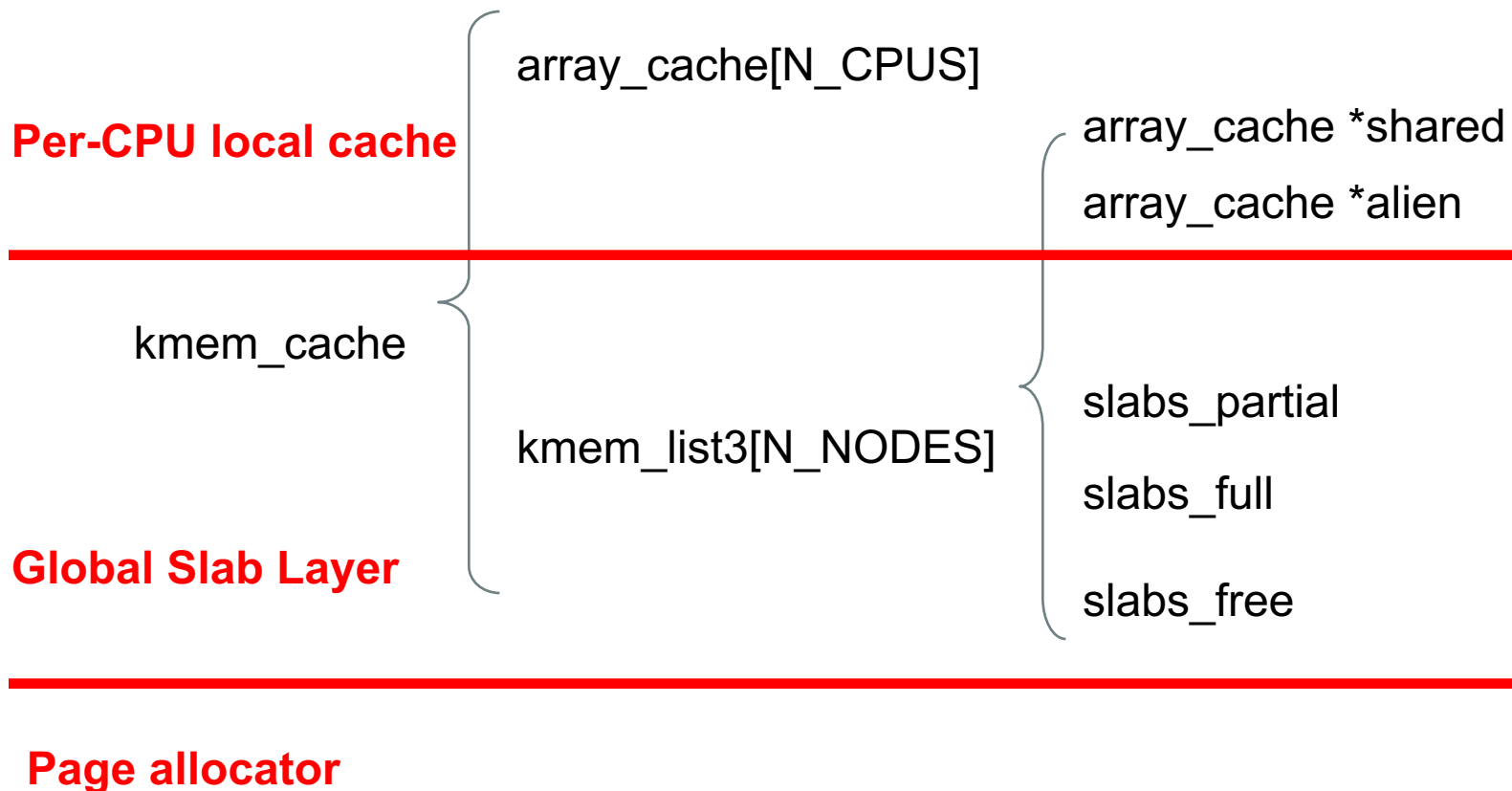
Kernel memory APIs



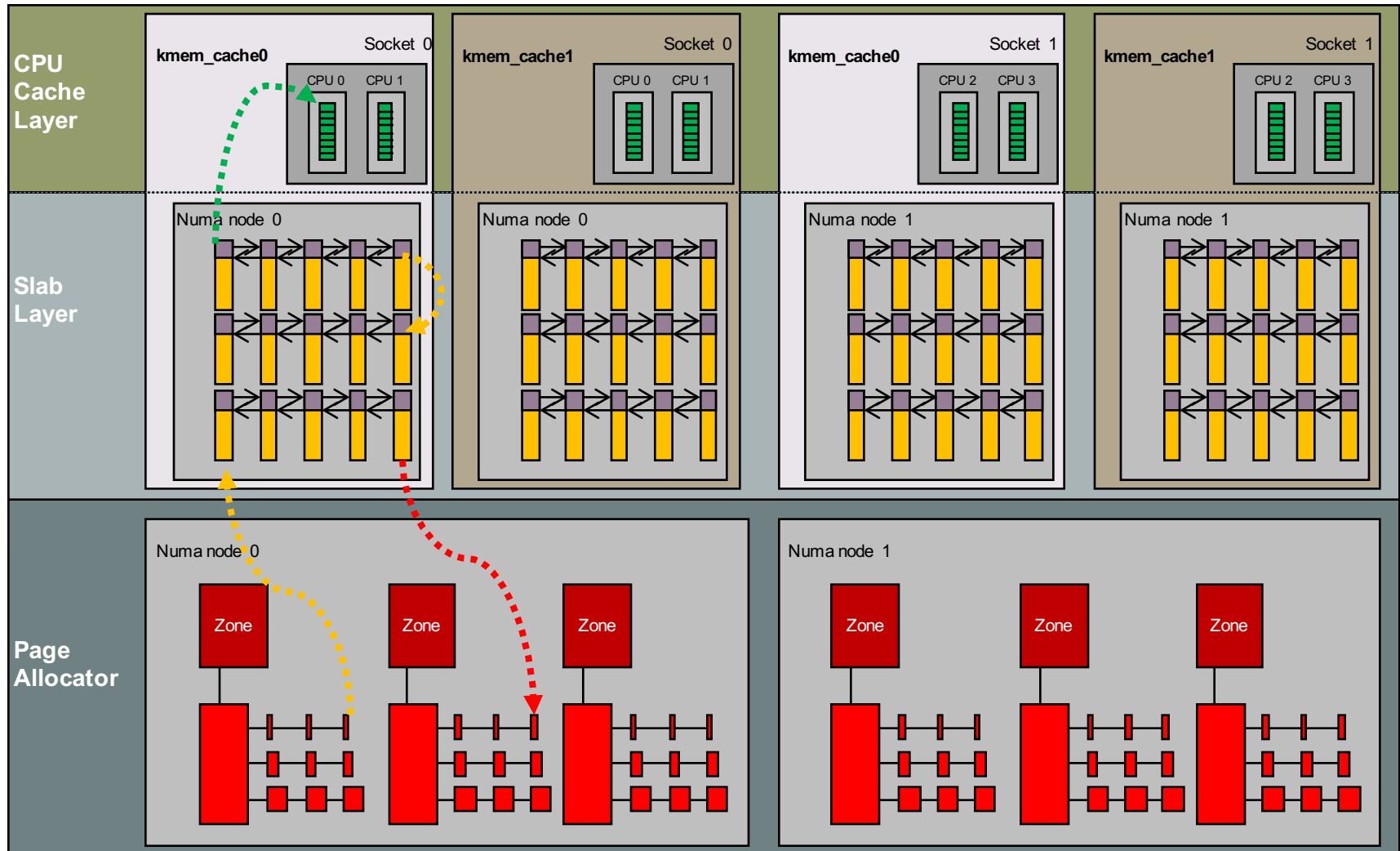
SLAB Intro



Slab Structure



Slab - Layered Design



Slab memory layout

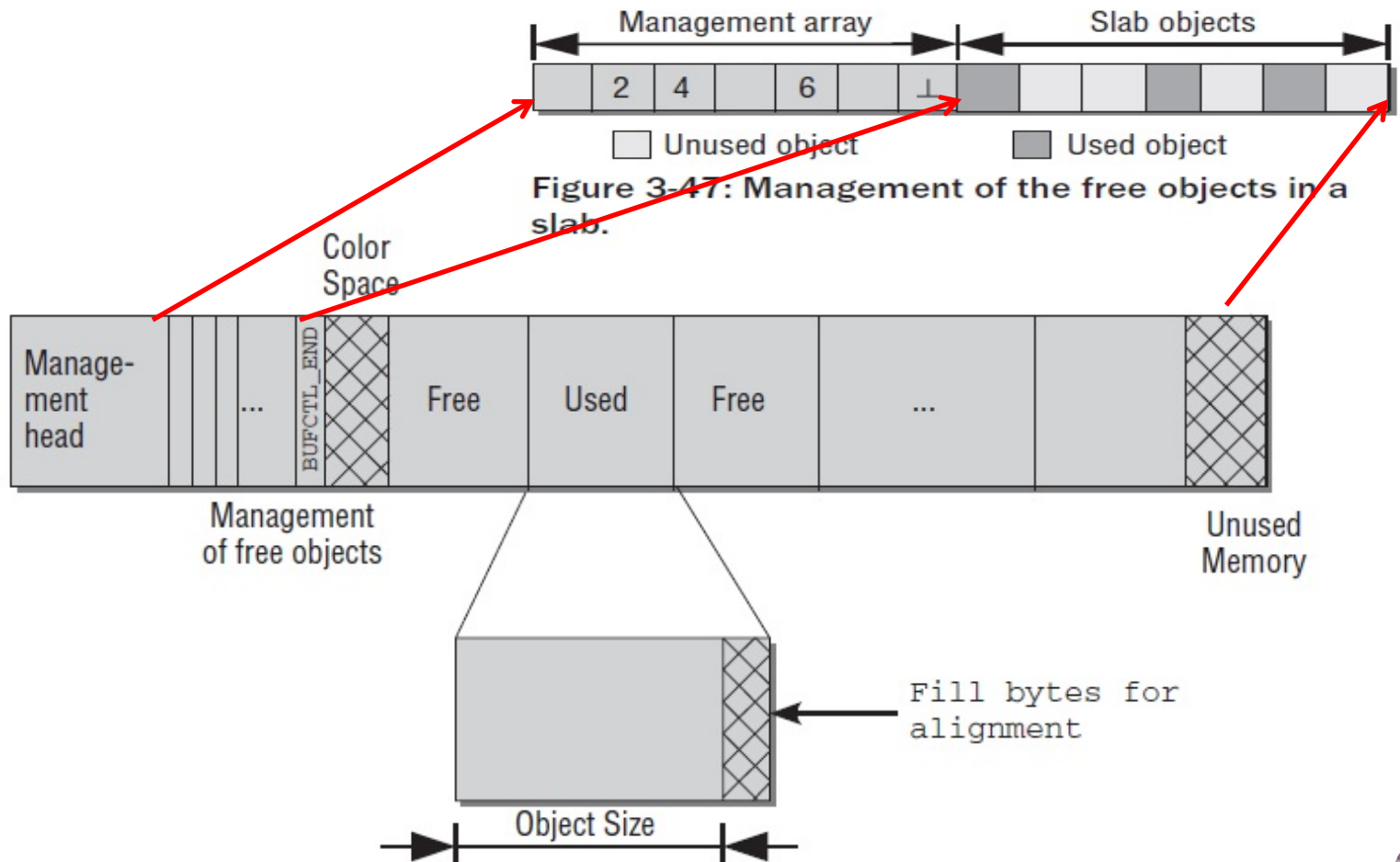


Figure 3-46: Fine structure of a slab.

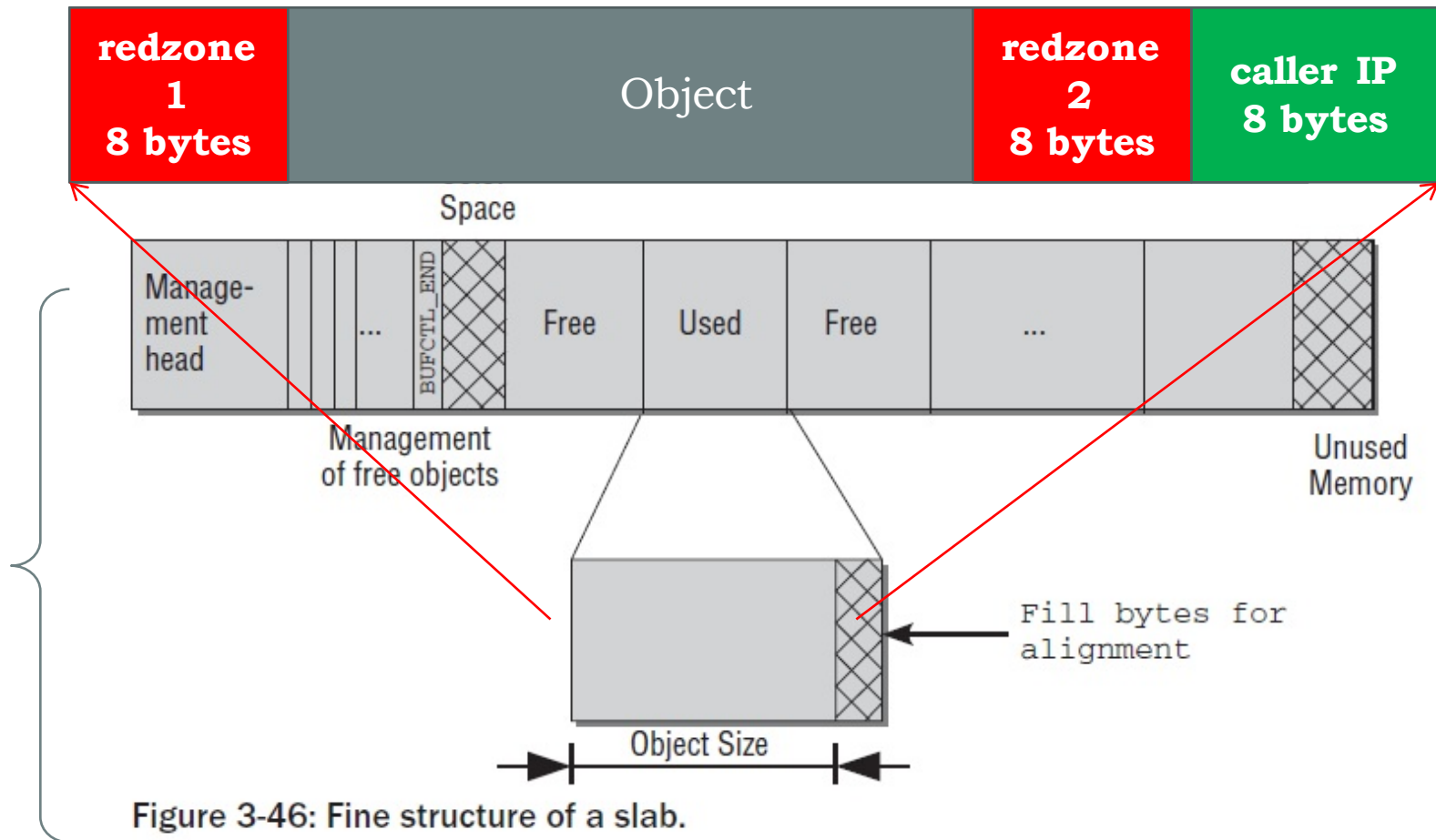


SLAB debugging use cases

- Use-after-free
 - Poison, check at next alloc
 - If off_slab and nPAGEs, record the call stack of last free, then unmap the slab page
- Use-before-initialization
 - Poison, check at debugging time
- Double free, check the memory outside of object
 - Redzone, mark INACTIVE at free time
 - Bufctl, mark FREE at free time
- Buffer overflow
 - Redzone, check redzone at free
- Memory Leak
 - Traverse all the slab, and aggregate the callers (STORE_USER)



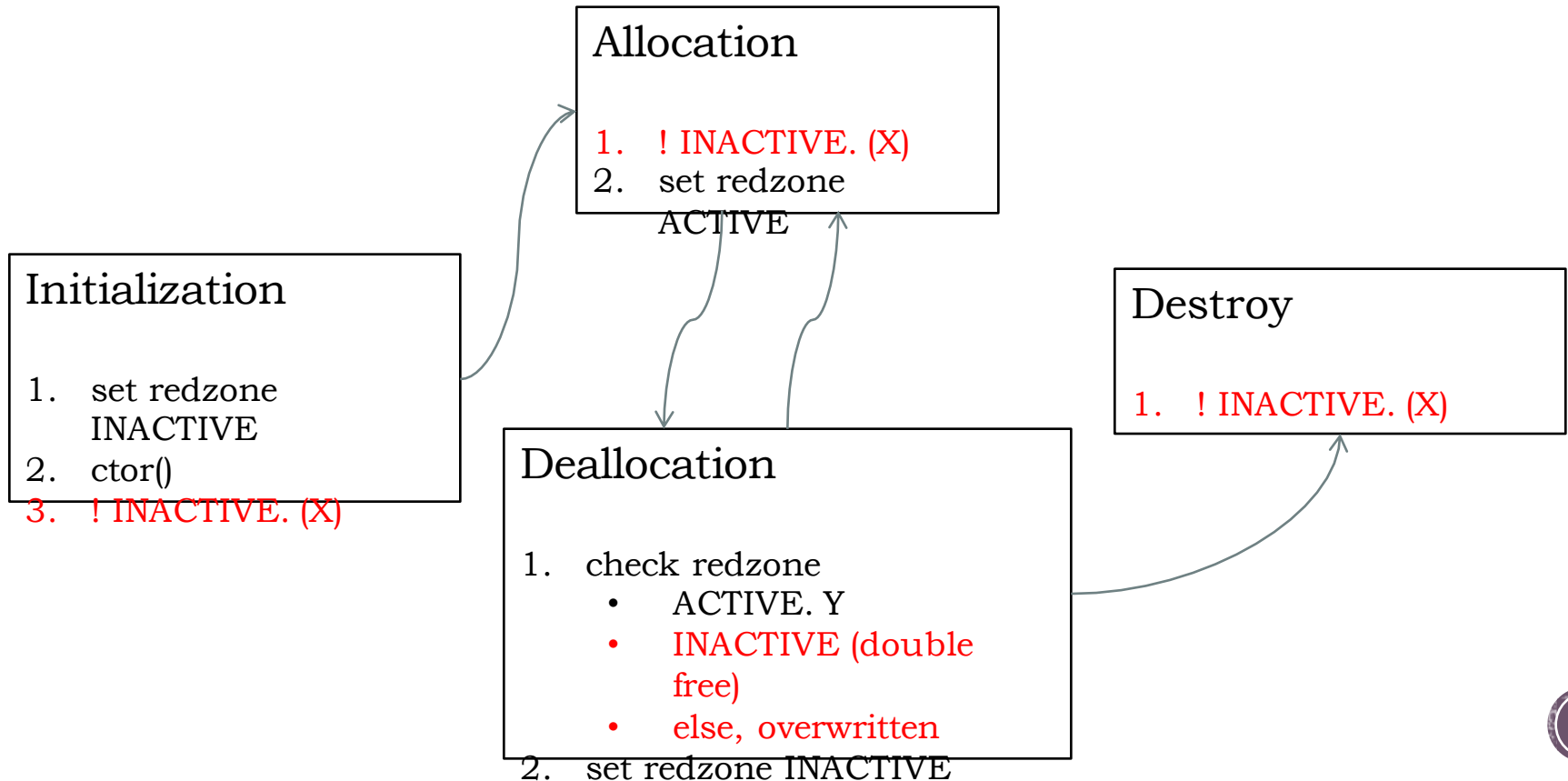
Object layout for debugging



Redzone

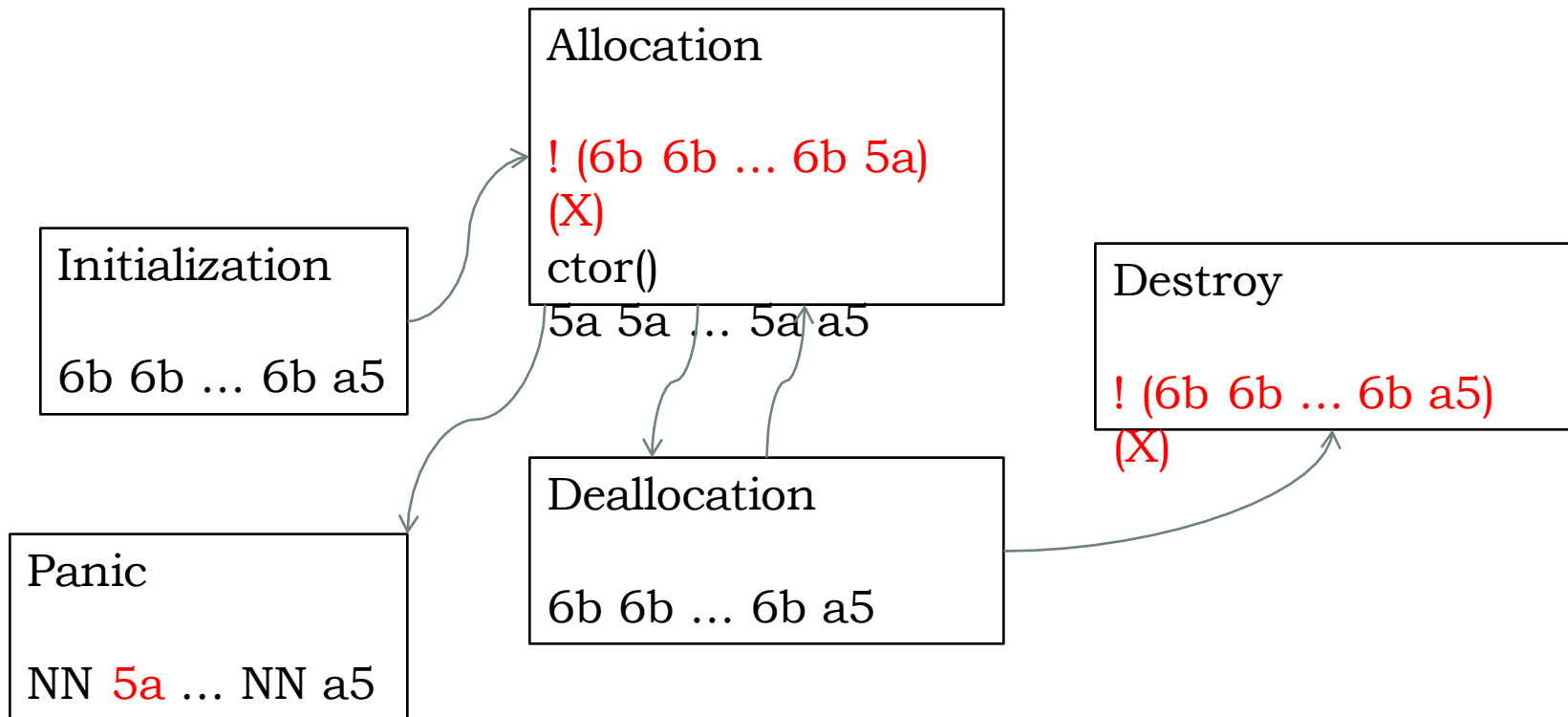
```
#define RED_INACTIVE 0x09F911029D74E35BULL /* when obj is inactive */
```

```
#define RED_ACTIVE 0xD84156C5635688C0ULL /* when obj is active */
```



Poison

```
#define POISON_INUSE 0x5a /* for use-uninitialised poisoning */  
#define POISON_FREE 0x6b /* for use-after-free poisoning */  
#define POISON_END 0xa5 /* end-byte of poisoning */
```



Other Debug Features

- Save caller IP
- State for management array tacking
 - BUFCTL_END
 - BUFCTL_FREE
 - BUFCTL_ACTIVE
 - SLAB_LIMIT



Agenda

- Basic concepts
- Debugging methods
- Case study
- Potential improvements



Kernel Core Analysis

- Identify corruption location
- Confirm corruption pattern
- Search potential culprit
- Nail down issue by code



Identify corruption location

- Understand the scenario
- Study the source code
- Verified the scenario in kernel core files



Confirm Corruption Pattern

- Get to familiar with the corrupted data structure
 - Learn related data structure and source code
- Jump out the box - dump the raw memory pages
 - Basic knowledge of kernel memory allocators
 - Slab/Slub/vmalloc/ioremap/page allocators
- Are they similar with any known corruption patterns?
 - Is it a possible corruption pattern caused by HW error?
 - If yes, confirm from BIOS SEL logs
 - For patterns caused by SW bugs, please refer to page 5.



Search potential culprit

- Search the pointer if culprit owns the pointer
 - Per corruption pattern, determine possible pointer address
 - Run `search -k <pointer address>` to get all references
 - Using `kmem` and `rd` to determine the references owners
- Search the corruption data if culprit owns that pattern
 - Per corruption pattern, determine the basic corrupted data
 - Run `search -k <data pattern>` to get all references
 - Using `kmem` and `rd` to confirm the references owners



Nail down issue by code

- Narrow down the source code in possible culprit
 - Per corruption pattern, determine data structure
 - Per corruption pattern, determine the related memory API
 - Find the memory signature if possible
- Any debug code could be enabled for catching bugs?
 - Run the testing with debug code enabled



Agenda

- Basic concepts
- Debugging methods
- Case study
- Potential improvements



A Slab corruption bug - 1

- Identify the corruption location
 - Get the back trace, and find the panic location
 - `cache_alloc_refill+0x17b`
 - Dump the corrupted memory

```
crash> slab ffff810262bf5040
struct slab {
    list = {
        next = 0x20a00150463, <===== bad pointer
        prev = 0xffff810c0ec002c0
    },
    .....
```
- Understand why the corruption cause the panic
 - Unable to handle kernel paging request at `0000020a0015046b`



A Slab corruption bug - 2

- Confirm the corruption pattern
 - Get to familiar with the corrupted data structure
 - Slab struct is at or close to the page boundary
 - Jump out the box - dump the raw memory pages
 - Not only check the slab, but also dump the adjacent pages
 - Are they similar with any known corruption patterns?
 - Shouldn't be HW bug, as the corruption pattern had the significant pattern
 - It looked like the buffer overflow bug.



A Slab corruption bug - 3

- Correlate corruption with potential culprit
 - Search the pointer if culprit owns the pointer
 - The pointer address might be ffff810262bf4000 because,
 - the corruption pattern seemed to start here.
 - the kmem ffff810262bf4000 indicated it is not allocated by slab
 - Search who reference the pointer?
 - `crash> search -k ffff810262bf4000`
 - `ffff810262f03928: ffff810262bf4000`
 - Who is the owner of ffff810262f03928?
 - `crash> rd ffff810262f03920 -64 128`
 - Found the signature: `qla2xxx_ts_11` and `QLE2562`



Agenda

- Basic concepts
- Debugging methods
- Case study
- Potential improvements



Improvements for memory corruption

- Using debug kernel in...
 - Kernel/Driver unit testing
 - Release testing
- Increase the debuggability for kernel/driver
 - Avoid to use the page allocator if SLUB/SLAB allocation is possible
 - Consider to implement some debug features
 - Create module/driver unique memory signature
 - Introduce the redzone and posion code in module/driver
 - Replace Slab with Slub which has more debug features
 - Production mode debug features
 - Per kmem cache granularity





Q&A

Thank you!

