How to use KASAN to debug memory corruption in OpenStack environment?

Gavin Guo
gavin.guo@canonical.com
Software Engineer
Cloud Dev Ops - Support Technical Services

Liang Chen
liang.chen@canonical.com
Software Engineer
Cloud Dev Ops - Support Technical Services

Agenda

- Background
- What is KASan?
- Why KASan?
- Backport KASan to the v3.13 kernel.
- How to deal with memory corruption bug?
  - Standard flow
  - KASan's limitation and how to handle?
- KASan's To be done in Ubuntu kernel LTS v3.13
- How KASan is applied to catch a bug in OpenStack environment.
- Solve the bug and submit patch to upstream.

Background

Ubuntu OpenStack solution

Ubuntu Server LTS

+

openstack

What is KASan?

Introduction

- KASan is the kernel space implementation of ASan(Address Sanitizer)[1] and can be used to detect the use-after-free and out-of-bound memory access for both read/write.
- KASan was introduced in v4.0-rc1 kernel in Feb 2015 by Andrey Ryabinin.
- Support x86_64/ARM64[2] and SLUB/SLAB[3] allocator.
- GCC >= 5.0 with “-fsanitize=kernel-address” compile option.
- It consumes about ⅛ of available memory and brings about ~x3 performance slowdown.[4]

[1]. AddressSanitizer: A Fast Address Sanity Checker by Google research.

https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[2]. Andrey Ryabinin in Oct 2015(39d114ddc682 arm64: add KASAN support).

[3]. Alexander Potapenko(Google) in Mar 2016(7ed2f9e66385 mm, kasan: SLAB support).

[4]. lib/Kconfig.kasan

What is KASan?

High level view

1. KASan allocates shadow memory to track all of kernel virtual memory on an 8-byte granularity.
2. KASan instruments all memory accesses in the compiled code.
3. Memory alloc / free operations update the state in the shadow memory for the corresponding virtual address.
4. At runtime, each memory access instrumentation checks the shadow memory state to determine if it is valid.

What is KASan?

Compiler instrumentation

Compiler adds ***__asan_{load,store}**** before corresponding memory access.
And the hook functions are implemented in Kernel.

```
ffffffff81002f10:    mov    -0x208(%rbp),%rbx
ffffffff81002f17:    mov    %rbx,%rdi
ffffffff81002f1a:    callq  ffffffff812fc460 <__asan_store8>
ffffffff81002f1f:    movq   $0x0,(%rbx)
ffffffff81002f26:    addl   $0x1,-0x200(%rbp)
ffffffff81002f2d:    addq   $0x8,-0x208(%rbp)
```

The "call __asan_store8" instruction is added by compiler before the store operation.

This is the store operation which writes the value 0 to the address pointed by %rbx register.

Compiler added hook functions

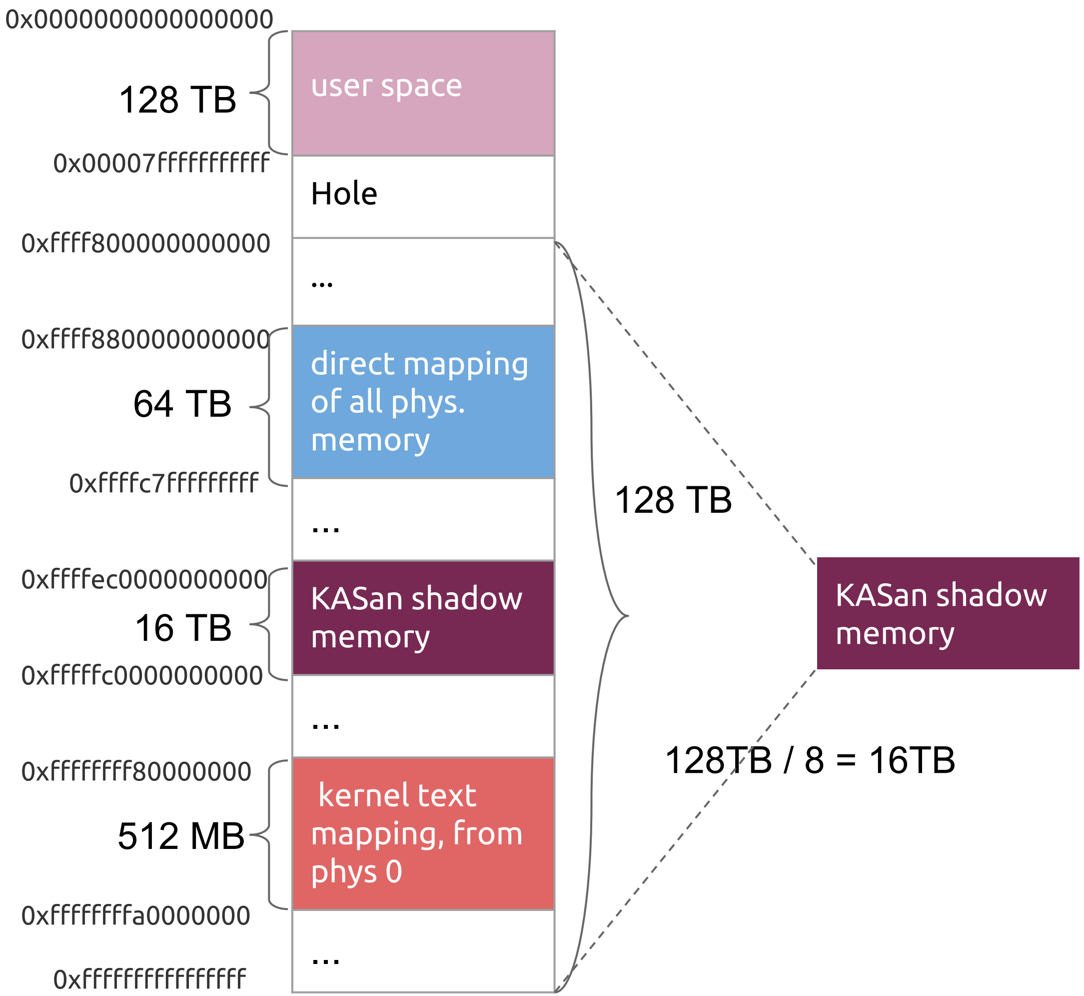
Memory access instruction

Functions like ***memset/memcpy/memmove*** cannot be instrumented by compiler as these functions are implemented in assembly and need manual modification. ***Kernel modules*** also need to be recompiled with GCC 5+ to support KASan.

What is KASan?

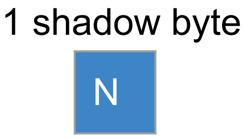
X86_64 KASan shadow memory

- shadow_addr = (addr >> 3) + KASAN_SHADOW_OFFSET
- User address access can also be detected when the kernel accesses user space memory without using special API (copy_to_user / copy_from_user)
-

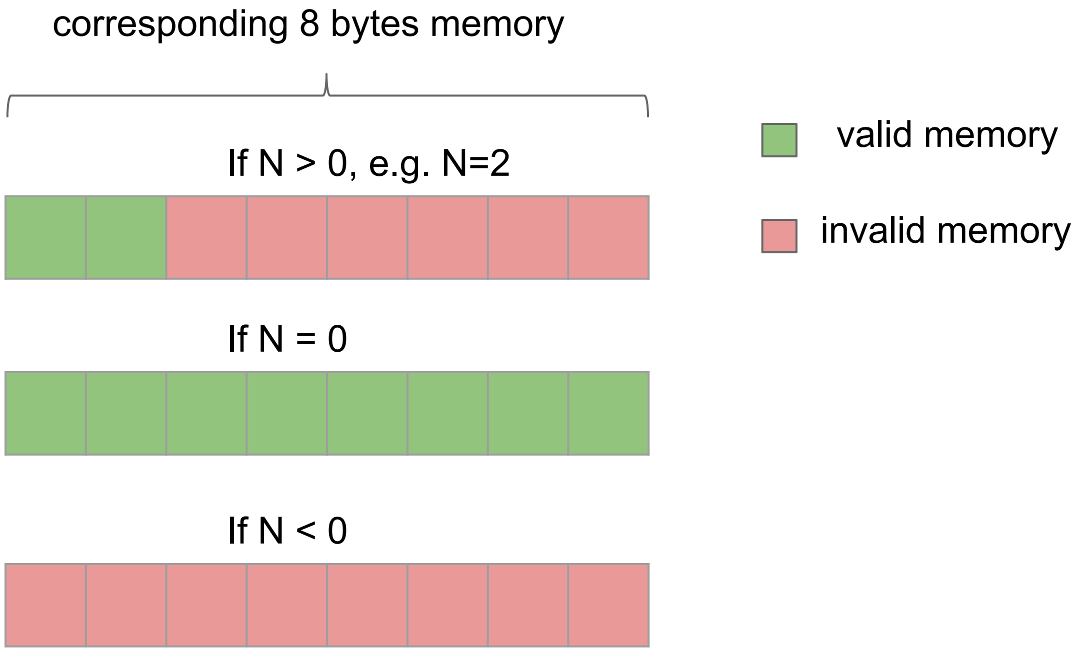


What is KASan?

KASan shadow memory mechanism



Address sanitizer uses 1/8 of the memory addressable in kernel for shadow



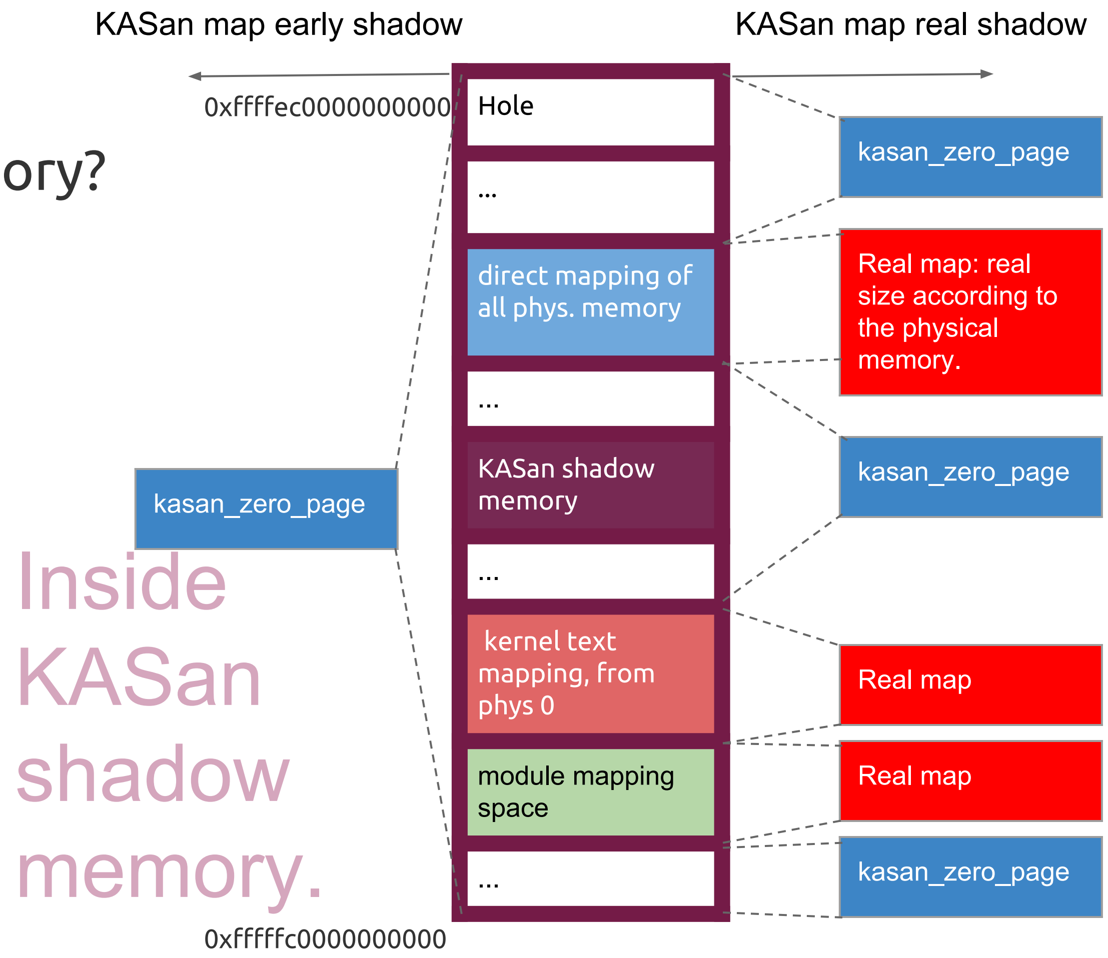

```
#define KASAN_FREE_PAGE         0xFF  /* page was freed */
#define KASAN_PAGE_REDZONE      0xFE  /* redzone for kmalloc_large allocations */
#define KASAN_KMALLOC_REDZONE   0xFC  /* redzone inside slub object */
#define KASAN_KMALLOC_FREE      0xFB  /* object was freed (kmem_cache_free/kfree)
*/
#define KASAN_GLOBAL_REDZONE    0xFA  /* redzone for global variable */
```

What is KASan?

When to map shadow memory?

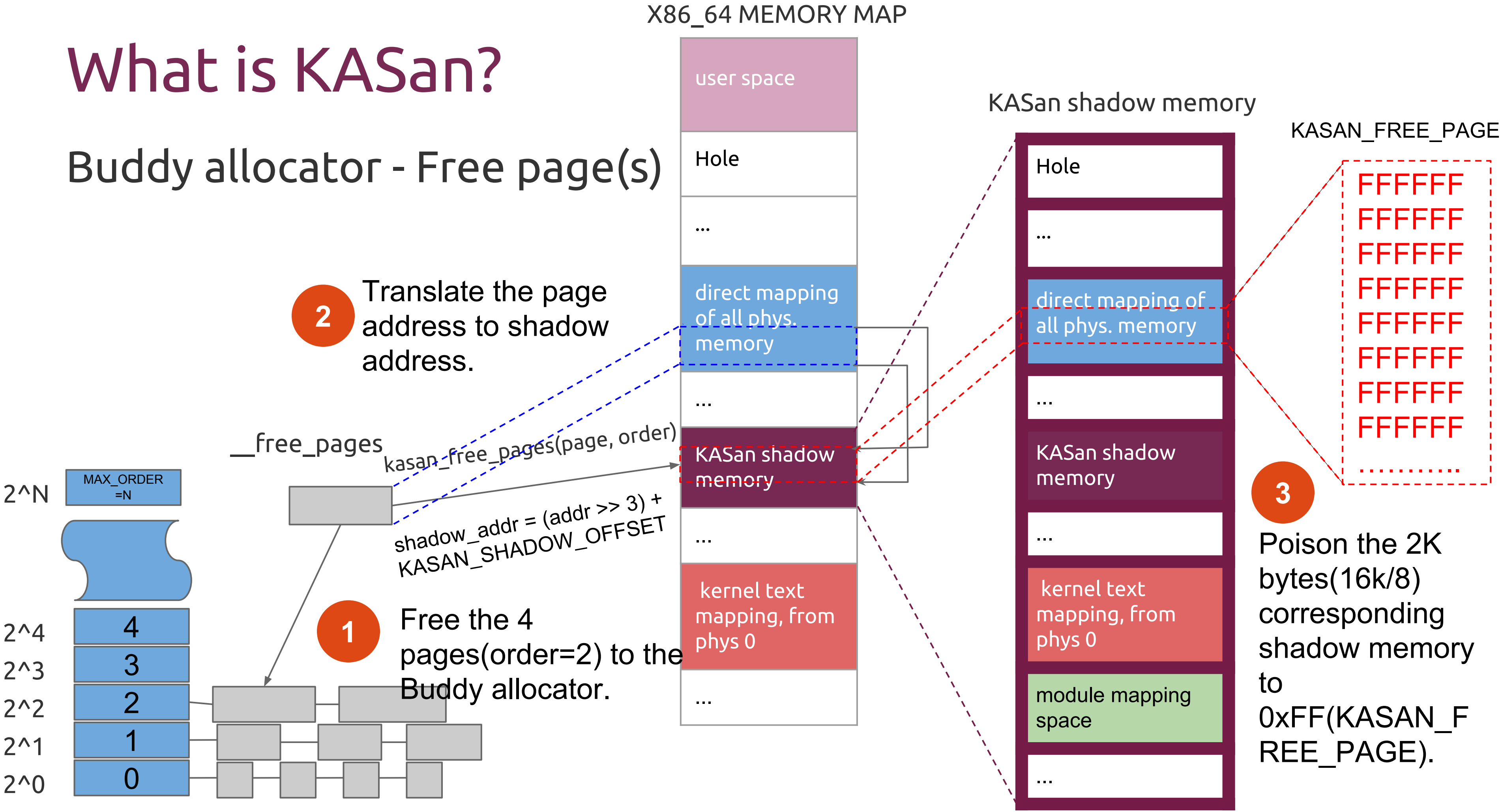
1. During the early boot process, the zero page was mapped to the KASan shadow address space. When the virtual address is mapped, the KASan shadow address space was mapped by vmemalloc.

2. About the module mapping space, the real shadow space is allocated when the module is loaded.



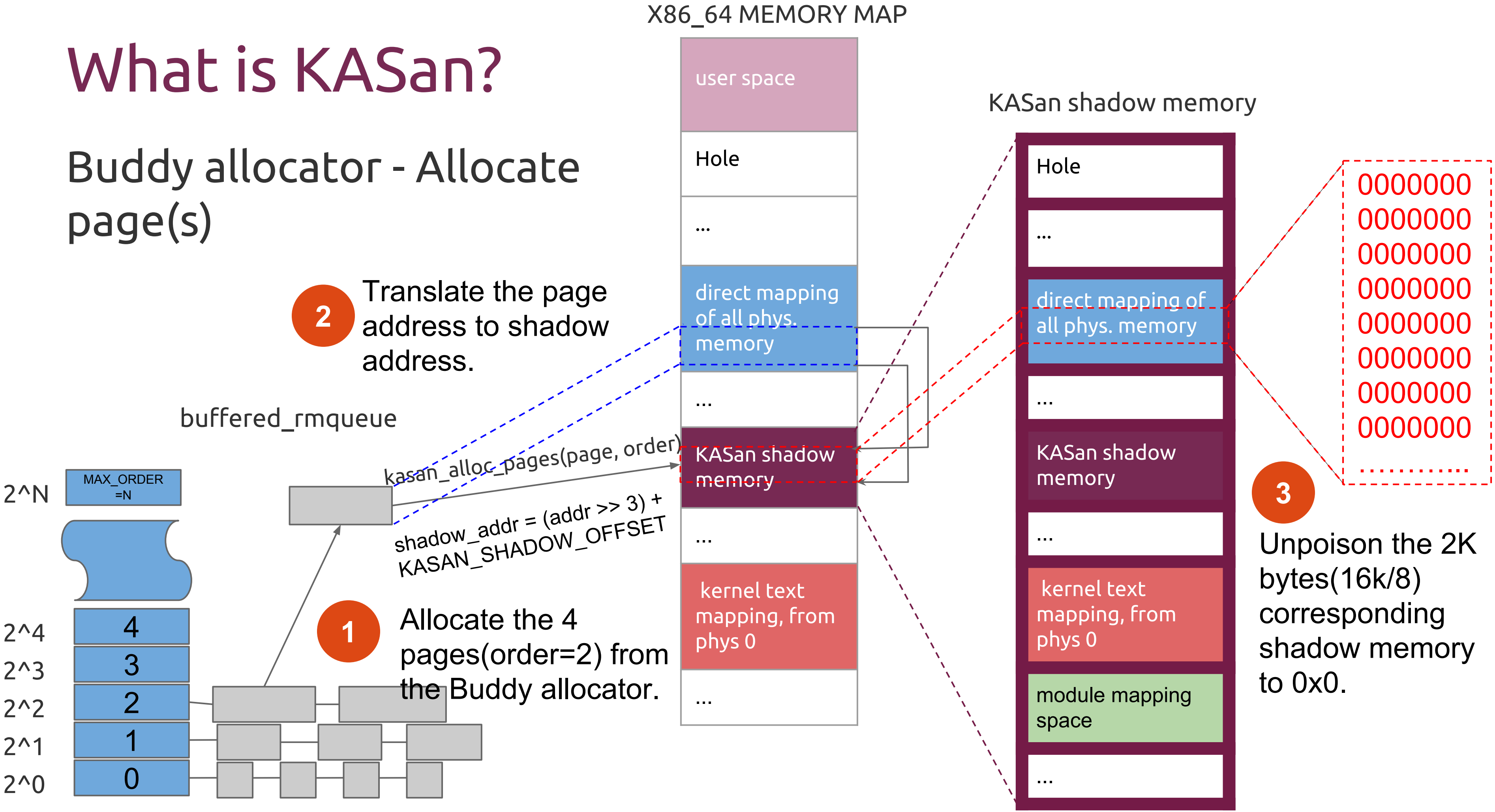
What is KASan?

Buddy allocator - Free page(s)

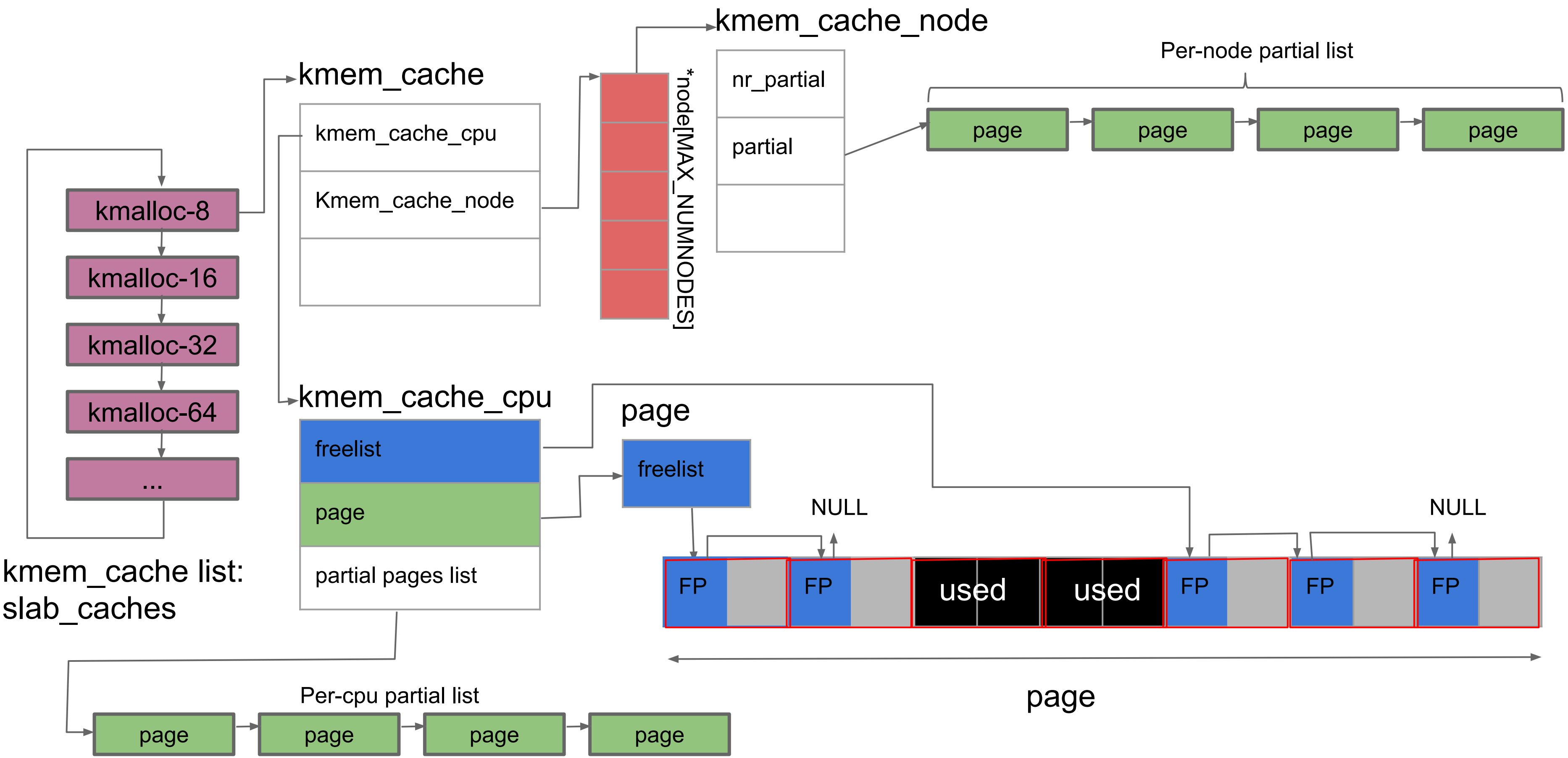


What is KASan?

Buddy allocator - Allocate page(s)



What is KASan? SLUB allocator: Object allocation

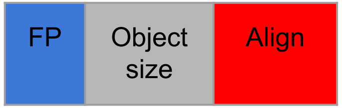


What is KASan?

SLUB allocator structure: Object format

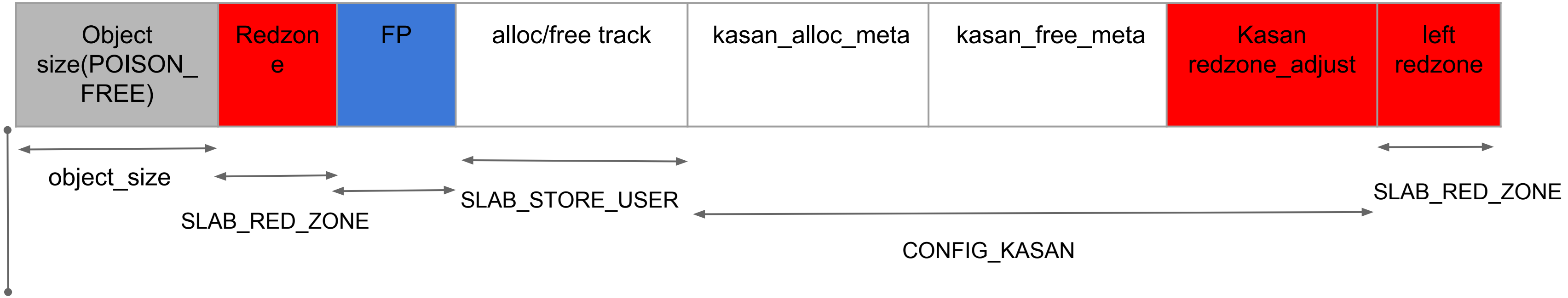
FP: Freelist pointer points to next available freelist pointer.
SLAB_STORE_USER: "slub_debug=**U**" is used to save the allocation/free call path.
SLAB_RED_ZONE: "slub_debug=**Z**" is to fill RED_ZONE flag to detect the OOB memory corruption.
SLAB_POISON: "slub_debug=**P**" is to fill with the POISON_FREE flag to detect the UAF memory corruption.
kasan_alloc_meta: Save the allocation/free path by stack depot.
kasan_free_meta: Save the quarantine information.

Normal Format



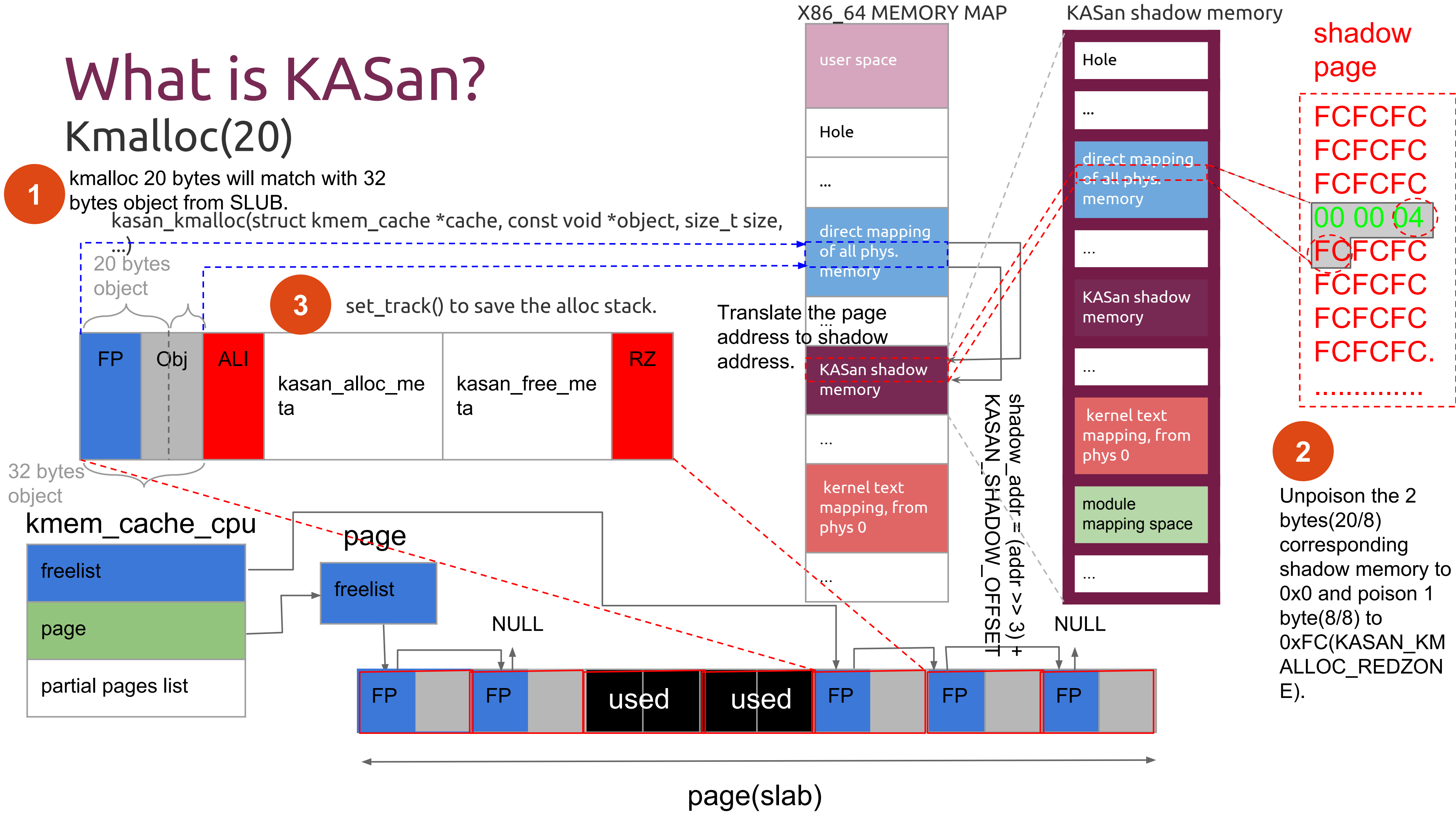
slub_debug=PZU

slub_debug=PZU with KASan enabled

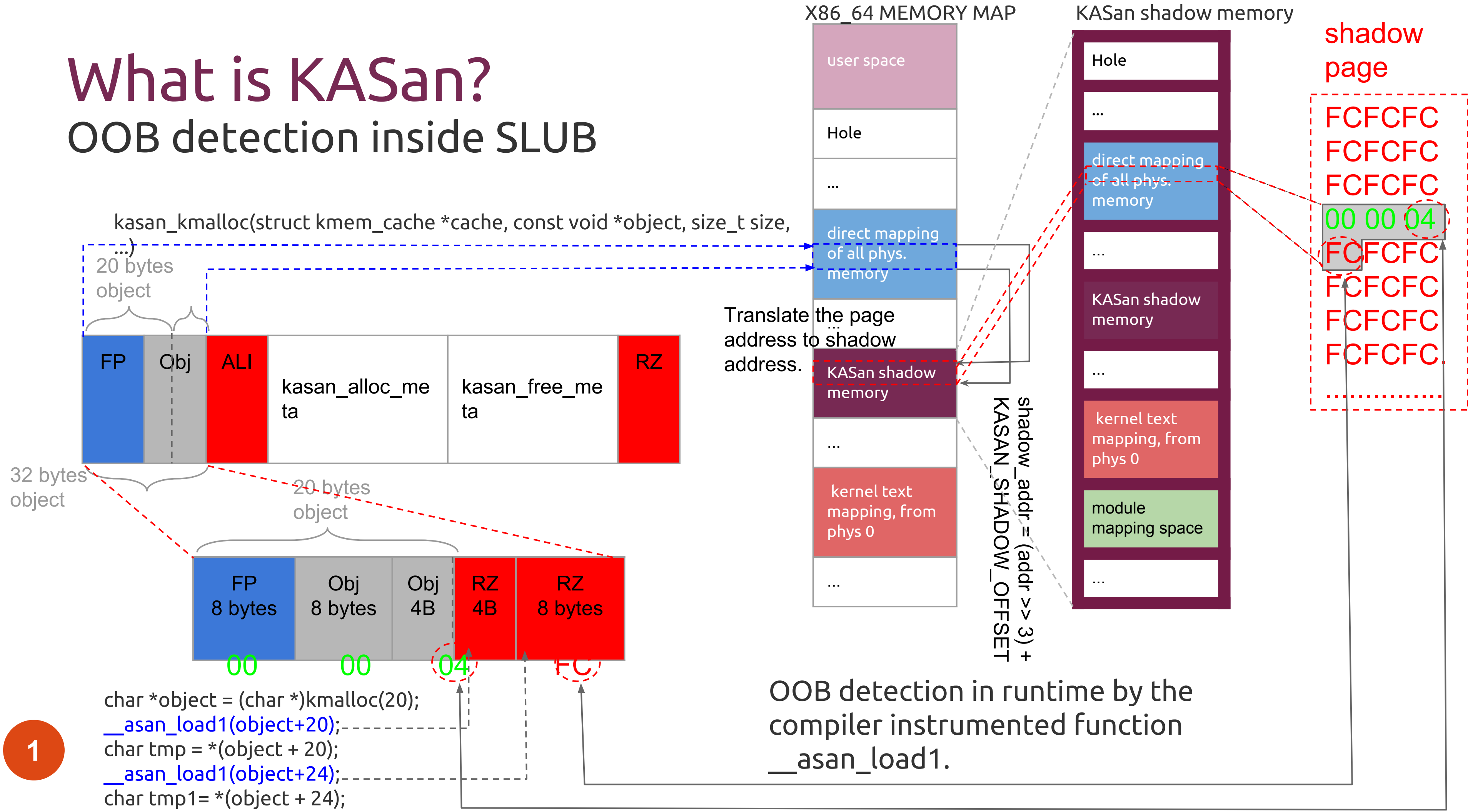


What is KASan?

Kmalloc(20)



What is KASan?
OOB detection inside SLUB

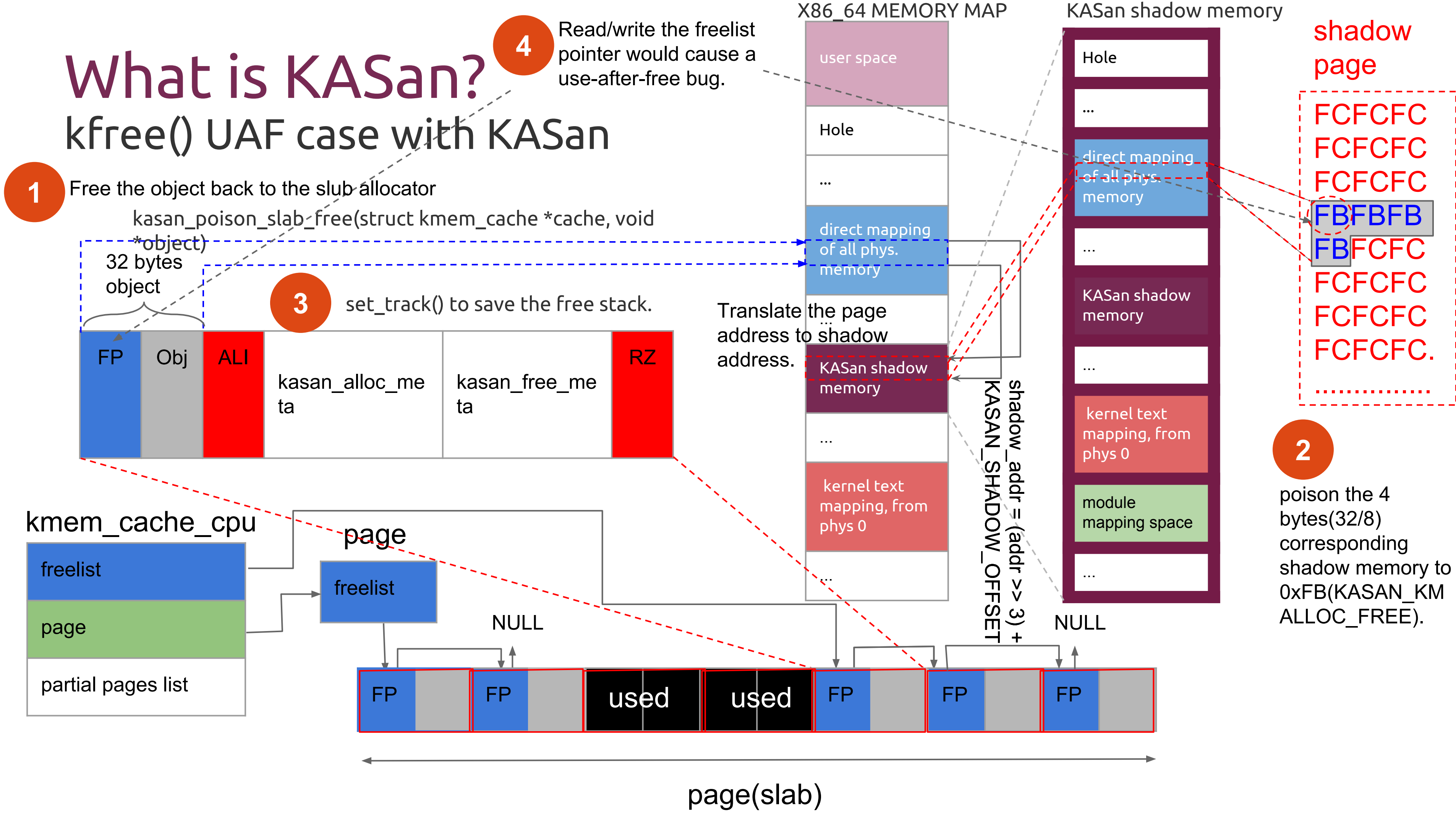



```
char *object = (char *)kmalloc(20);
__asan_load1(object+20);
char tmp = *(object + 20);
__asan_load1(object+24);
char tmp1= *(object + 24);
```

OOB detection in runtime by the compiler instrumented function __asan_load1.

What is KASan?

kfree() UAF case with KASan



What is KASan?

Global variables redzone detection

- Compiler would build each global variable initialization function into the init_array section.[1]
- Kernel will initialize the redzone flag of each global variable in the boot up process/module allocation[2].

3 The __asan_register_globals will fill the shadow memory with the redzone flag after every struct members inside the struct.

```
struct uts_namespace init_uts_ns = {
        .kref = {
                .refcount       = ATOMIC_INIT(2),
        },
        .name = {
                .sysname        = UTS_SYSNAME,
                .nodename       = UTS_NODENAME,
                .release        = UTS_RELEASE,
                .version        = UTS_VERSION,
                .machine        = UTS_MACHINE,
                .domainname     = UTS_DOMAINNAME,
        },
        .user_ns = &init_user_ns,
        .ns.inum = PROC_UTS_INIT_INO,
#ifdef CONFIG_UTS_NS
        .ns.ops = &utsns_operations,
#endif
};
```

1 Boot process proceed to the do_ctors() function.

kernel_init_freeable → __ctors_start

do_basic_setup

do_ctors()

.init_array.*

__ctors_end

```
for (; fn < (ctor_fn_t *) __ctors_end; fn++)
      (*fn)();
```

2 The do_ctors() will invoke the callback function for every variable provided by compiler.

```
ffffffff81d26790 <_GLOBAL__sub_I_65535_1_init_uts_ns>:
_GLOBAL__sub_I_65535_1_init_uts_ns():
/home/gavin/ubuntu-xenial/init/version.c:57
ffffffff81d26790:       55                      push   %rbp
ffffffff81d26791:       be 03 00 00 00          mov    $0x3,%esi
ffffffff81d26796:       48 c7 c7 80 64 61 82    mov    $0xffffffff82616480,%rdi
ffffffff81d2679d:       48 89 e5                mov    %rsp,%rbp
ffffffff81d267a0:       e8 0b 62 5d ff          callq  ffffffff812fc9b0 <__asan_register_globals>
ffffffff81d267a5:       5d                      pop    %rbp
ffffffff81d267a6:       c3                      retq
ffffffff81d267a7:       66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
ffffffff81d267ae:       00 00
```


[1]. 9ddf82521c86 kernel: add support for .init_array.* constructors

[2]. bebf56a1b176 kasan: enable instrumentation of global variables

What is KASan?

Stack redzone OOB detection

Stack instrumentation allows to detect out of bounds memory accesses for variables allocated on stack. Compiler adds redzones around every variable on stack and poisons redzones in function's prologue. Such approach significantly increases stack usage, so all in-kernel stacks size were doubled.

Why KASan?

Comparing to existed debugging mechanisms.

1. slub_debug=PZU can detect the memory corruption, however, cannot find the exact point of the invalid memory access.
2. KGDB, qemu, or ICE-liked instrumentation tool cannot be used in the production environment. And it's also hard to watch dynamically allocated memory.
3. PAGE_POISON can just detect the memory corruption on page allocation.
4. Kdump can find the corpse of the memory corruption victim, however, the killer is gone. Have no clues!! In the next slide, there is a use-after-free example explaining the difficulty of finding the culprit.

Why KASan?

Example of use-after-free condition inside the SLUB allocator.

1. The object 3 is freed(e.g. kfree), connected to the freelist and served as the head.

Current freelist: 3

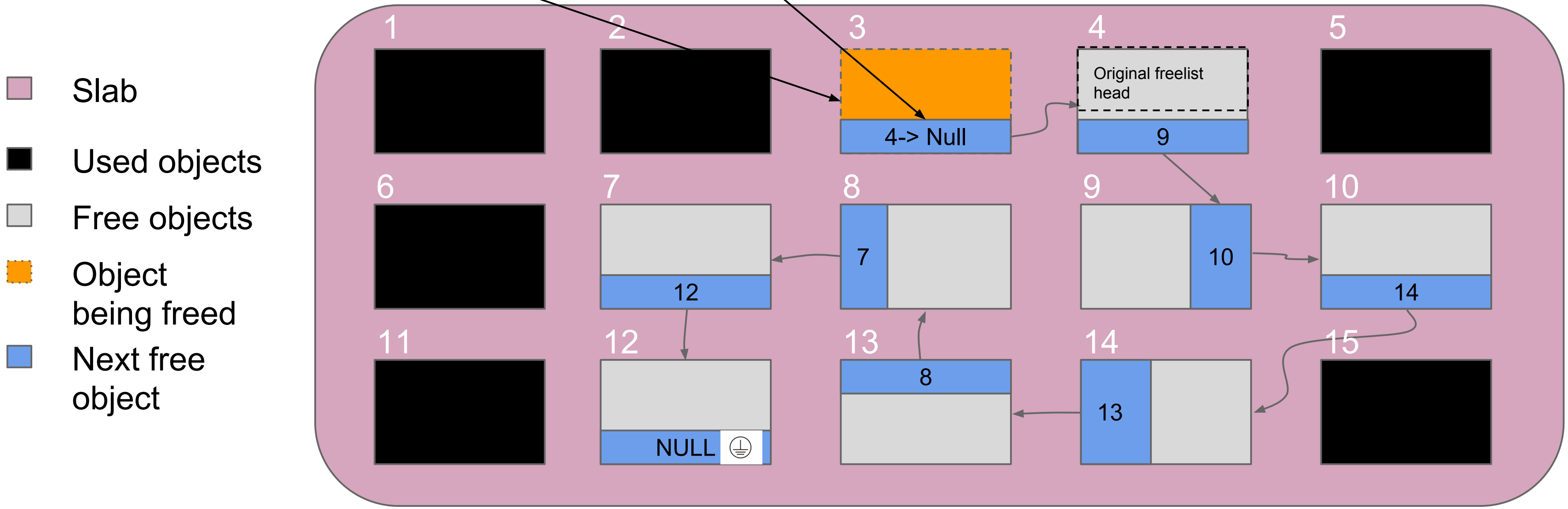
2. The object 3 is overwritten to unknown address, e.g. NULL, and next pointer is corrupted.

3. The object 3 is allocated again(e.g. kmalloc) and current freelist is set to unknown value NULL.

Current freelist: NULL ⏚

4. The kmalloc is called again and because of the corrupted freelist. It triggers the NULL pointer access corruption.

Current freelist: NULL ⏚



Why KASan?

Comparing to existed debugging mechanisms.

5 DEBUG_PAGE_ALLOC

- The debug mechanism would unmap the page from the PTE when the page is freed to *Buddy system*.
- An use-after-free memory corruption would trigger the page fault since the mapping is not there.
- The granularity is too big since it works in the page size. The bug happens in the SLUB allocator cannot be detected.

Free page(s) path:
__free_pages-> ... -> free_pages_prepare -> kernel_map_pages(0) to unmap the page

Allocate page(s) path:
__alloc_pages_nodemask->...->prep_new_page->kernel_map_pages(1) to map the page again

6 kmemcheck is highly costed and the performance degradation cannot be accepted by production environment. ([PATCH] mm: kill kmemcheck[1]: "As discussed on LSF/MM, kill kmemcheck").

Reference:
[1]. https://lkml.org/lkml/2015/3/11/242

Backport KASan to the v3.13 kernel

- In Feb 2015, KASAN was just merged into the upstream kernel v4.0-rc1.
- We encountered CVE-2015-1805 and NUMA performance issue.

1. **Basic KASan infrastructure.**
   a. ef7f0d6a6ca8 x86_64: add KASan support
   b. 786a8959912e kasan: disable memory hotplug
   c. 0b24becc810d kasan: add kernel address sanitizer infrastructure
2. **SLUB allocator and Page allocator support**
   a. 0316bec22ec9 mm: slub: add kernel address sanitizer support for slub allocator
   b. a79316c6178c mm: slub: introduce metadata_access_enable()/metadata_access_disable()
   c. b8c73fc2493d mm: page_alloc: add kasan hooks on alloc and free paths
3. **Instrumentation of global variables**
   a. bebf56a1b176 kasan: enable instrumentation of global variables
   b. 6301939d97d0 module: fix types of device tables aliases
   c. 9ddf82521c86 kernel: add support for .init_array.* constructors
4. **Instrumentation of stack variables**
   a. c420f167db8c kasan: enable stack instrumentation
5. **MISC**
   a. git log --grep=kasan -i --oneline; git log --author="Andrey Ryabinin" --oneline; git log --author="Alexander Potapenko" --oneline; git log --author="Dmitry Vyukov" --oneline; git log --author="Andrey Konovalov" --oneline

Grep authors of ASAN

How to deal with memory corruption bugs?

Standard flow

1. How to observe the memory corruption?
   1.1. Suspicious error messages. e.g. CVE-2015-1805

$ addr2line 0xffffffff811a31a0 -e usr/lib/debug/boot/vmlinux-3.13.0-48-generic -f -i

```
get_freepointer
/build/buildd/linux-3.13.0/mm/slub.c:260
get_freepointer_safe
/build/buildd/linux-3.13.0/mm/slub.c:275
slab_alloc_node
/build/buildd/linux-3.13.0/mm/slub.c:2416
slab_alloc
/build/buildd/linux-3.13.0/mm/slub.c:2455
kmem_cache_alloc_trace
/build/buildd/linux-3.13.0/mm/slub.c:2472
```

```
[186453.619488] CPU: 3 PID: 736944 Comm: nova-compute Not tainted 3.13.0-48-generic #80-Ubuntu
[186453.621751] Hardware name: Supermicro X8QB6/X8QB6, BIOS 2.0c    06/11/2
[186453.623561] task: ffff8816feb3c800 ti: ffff8816feb16000 task.ti: ffff8816feb16000
[186453.625608] RIP: 0010:[<ffffffff811a31a0>]  [<ffffffff811a31a0>] kmem_cache_alloc_trace+0x80/0x1f0
[186453.628097] RSP: 0018:ffff8816feb17e80  EFLAGS: 00010286
[186453.629550] RAX: 0000000000000000 RBX: ffff880763c71c80 RCX: 00000000000f682c
[186453.631503] RDX: 00000000000f682b RSI: 00000000000080d0 RDI: ffff88085f403500
[186453.682800] RBP: ffff8816feb17eb8 R08: 0000000000016320 R09: ffff88085f403500
[186453.786074] R10: ffffffff811c6f8e R11: 0000000000000246 R12: 3c72657475706d6f
[186453.890372] R13: 00000000000080d0 R14: 0000000000000280 R15: ffff88085f403500
[186453.994735] FS:  00007ff4dca50740(0000) GS:ffff88085f8c0000(0000) knlGS:0000000000000000
[186454.099126] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[186454.151258] CR2: 00007ff4dc99acb0 CR3: 000000184e521000 CR4: 00000000000027e0
[186454.253634] Stack:
[186454.303079]  ffff88085f403500 ffffffff811c6f8e ffff880763c71c80 ffff8816feb17f60
[186454.402456]  0000000000000000 ffff8816feb17f58 00007ff4d8a622fd ffff8816feb17ed0
[186454.501618]  ffffffff811c6f8e ffff8807a5908490 ffff8816feb17f10 ffffffff811c74a6
[186454.601087] Call Trace:
[186454.649358]  [<ffffffff811c6f8e>] ? alloc_pipe_info+0x3e/0xb0
[186454.698214]  [<ffffffff811c6f8e>] alloc_pipe_info+0x3e/0xb0
[186454.746217]  [<ffffffff811c74a6>] create_pipe_files+0x46/0x200
[186454.793277]  [<ffffffff811c7694>] __do_pipe_flags+0x34/0xf0
[186454.839382]  [<ffffffff811c7860>] SyS_pipe+0x20/0xa0
[186454.884356]  [<ffffffff81731fbd>] system_call_fastpath+0x1a/0x1f
[186454.929200] Code: cd 00 00 49 8b 50 08 4d 8b 20 49 8b 40 10 4d 85 e4 0f 84 14 01 00 00 48 85 c0 0f 84 0b 01 00 00 49 63 47 20 48 8d 4a 01
4d 8b 07 <49> 8b 1c 04 4c 89 e0 65 49 0f c7 08 0f 94 c0 84 c0 74 b9 49 63
[186455.065173] RIP  [<ffffffff811a31a0>] kmem_cache_alloc_trace+0x80/0x1f0
[186455.109610]  RSP <ffff8816feb17e80>
```

How to deal with memory corruption bugs?

Standard flow

   1.1.
   1.2. Kdump.
2. Try to identify what the kmem_cache is and the memory corruption type. e.g. use-after-free or out-of-bound access.
   2.1. Kmem_cache can be observed inside the kdump or enabling the "slub_debug=PUZ" to capture that.
   2.2. Need to enable the slub_nomerge to avoid the kmem_cache merging with the same object size.
3. In the enterprise production environment, please enable, for example: "slub_debug=P, kmalloc-32" in the kernel command line to alleviate (with slub_debug=P, the freepointer will be verified during alloc/free, if corruption finds, the rest of the freepointer will be zapped) the error before finding the culprit. For the kernel older than v4.2-rc1, please cherry-pick my patch to support slub_debug on specific object size: 4066c33d0308 mm/slab_common: support the slub_debug boot option on specific object size
4. Isolate one platform to deploy the KASan enabled kernel with "slub_debug=PZU,kmalloc-32 slub_nomerge" inside the kernel command line to capture the bug.

How to deal with memory corruption bugs?

KASan's limitation and how to handle?

- KASan cannot handle the case that the object is being overwritten when the object is validly allocated.

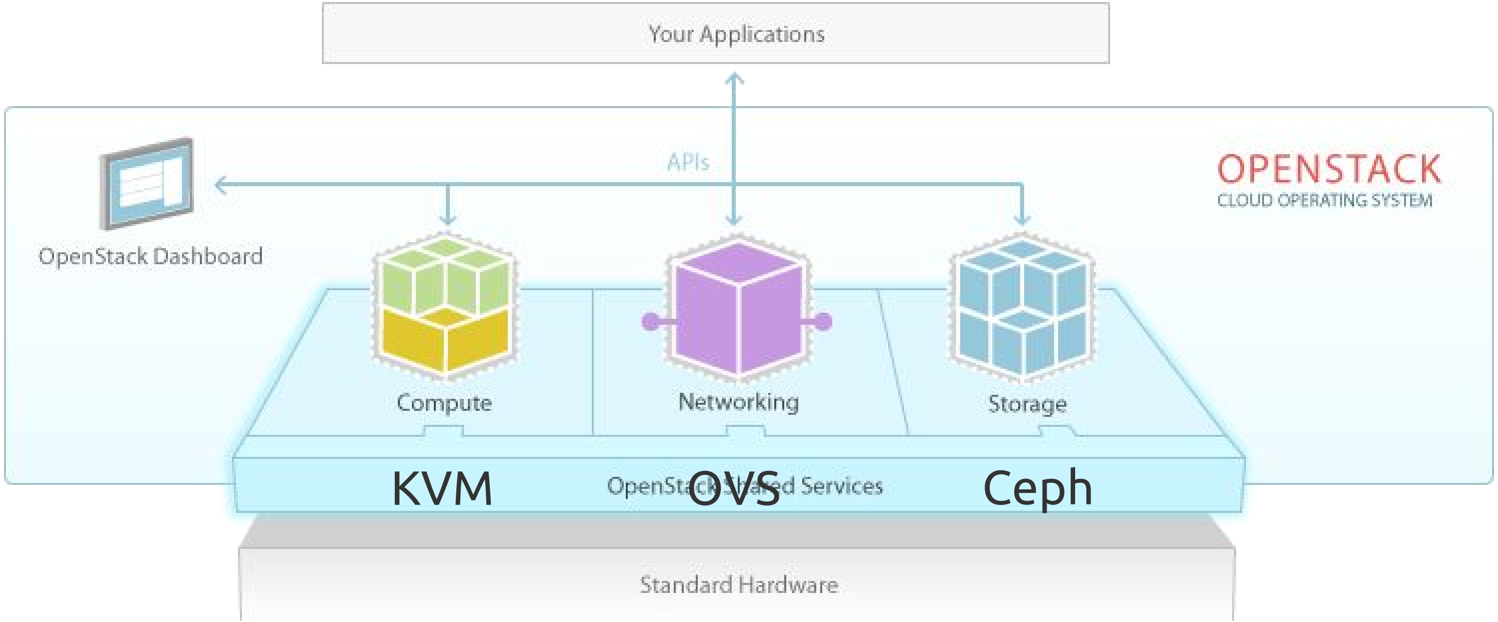
- We need to hack the check_mem_region() which is the general function called by load/store hook point of KASan.

KASan's to be done in Ubuntu kernel LTS v3.13

- Fuzzing (Trinity, iknowthis, perf_fuzzer, syzkaller)
- KTSan (Kernel Thread Sanitizer)
- Stack depot[1] (With SLUB in 4.8-rc1[4])
  - Store stacks in separately allocated pages. Duplicate stacks are not stored, instead a hashtable is used to index the stack contents.
  - SLUB_DEBUG stacks have overhead of 256 bytes per object. Stack depot takes ~100 times less.[2]
- Quarantine (Introduced in 4.7-rc1[3], with SLUB in 4.8-rc1[4])
  - As the object can be quickly reallocated, freed objects are first added to per-cpu quarantine queues which helps to detect use-after-free errors.

Reference:
[1]. cd11016e5f52 mm, kasan: stackdepot implementation. Enable stackdepot for SLAB
[2]. Re: [RFC] slub memory quarantine
http://www.spinics.net/lists/linux-mm/msg85219.html
[3]. 55834c59098d mm: kasan: initial memory quarantine implementation
[4]. 80a9201a5965 mm, kasan: switch SLUB to stackdepot, enable memory quarantine for SLUB

OpenStack Quick Overview



Compute (Nova)

- Manages the lifecycle of compute instances in an OpenStack environment. Responsibilities include spawning, scheduling and decommissioning of virtual machines on demand.

- nova reboot:

  Nova API -> Nova Compute -> Libvirt (destroy & create) -> Qemu

OpenStack Performance Issue

performance issue with numa balancing

A customer complained this issue and raised a ticket to us. He claimed to see a large number of numa migration taken place after some stress tests (concurrent & repetitive nova reboot), which caused system performance degradation. But they were not necessary and caused a lot of extra numa faults.

But the system doesn't produce any information in regards of the performance degradation. We got stuck and were not able to move alone the investigation for several weeks. It was until KASan enabled on one of the hosts we found an important clue which finally led to a fix.

Numa Balancing Issue

KASan Output and Root Cause

-

From the kansan output, we were finally able to locate the root cause. It was a race in two code paths - page_fault and finish_task_switch. task_numa_compare didn't increase task.usage of the current task on the destination queue by any means. Thus it could be freed in finish_task_switch and the memory could be reused again while task_numa_compare still holding a reference to the memory, which was the case for the issue we were facing. So having the root cause identified, it didn't take long for us to produce a fix and push it upstream.

Details:

http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=1dff76b92f69051e579bdc131e01500da9fa2a91

Thank you!