

Security Implications of Linux Kernel Electric Fence

Studying and Attacking KFENCE

Ernesto Martínez García

Linux Security Summit Europe 2026



Ernesto Martínez García

PhD Student @ Graz University of Technology

✉ `ernesto.martinezgarcia@tugraz.at`

🌐 `ernestomartinezgarcia.com`

🌐 `linkedin.com/in/ecomaikgolf`

Background



Attackers exploit bugs in the kernel to escalate privileges

As with userspace exploitation, **allocator exploitation** gained traction

↳ Attackers may solely exploit allocated memory (**data-only** attacks)

Linux allocator hardening is crucial nowadays

↳ Defense-in-depth against CVEs

Threat Model

- Unprivileged userspace command execution
- An attacker found an UAF on a kernel object
- We assume a hardened configuration (`hardening.config`)

Slab Hardening

Slab (SLUB) is the core Linux Kernel heap **memory allocator**

Contains comprehensive defense-in-depth mechanisms against exploitation

D1: Heap Segregation

SLAB_BUCKETS

SLAB_VIRTUAL

TYPED_KMALLOC_CACHES

D2: Sanitization

{KASAN, KMSAN}_*

KASAN_HW_TAGS

INIT_ON_{ALLOC, FREE}

D3: Randomization(s)

RANDOM_KMALLOC_CACHES^{D1}

SLAB_FREELIST_RANDOM

Most importantly, **objects are segregated**:

```
1 free(vuln_sketchy_object) // -> UAF Weak Code Object
2 very_important_object = alloc() // -> Even With Same Size
3 write(vuln_sketchy_object) 🛡️ // -> No UAF Memory Overlap
```

D4: Other

SAMSUNG RKP

HUAWEI HYPERVISOR

OTHER DOWNSTREAM

Cross-cache Attacks

Technique to reclaim cross-domain memory via the physical allocator

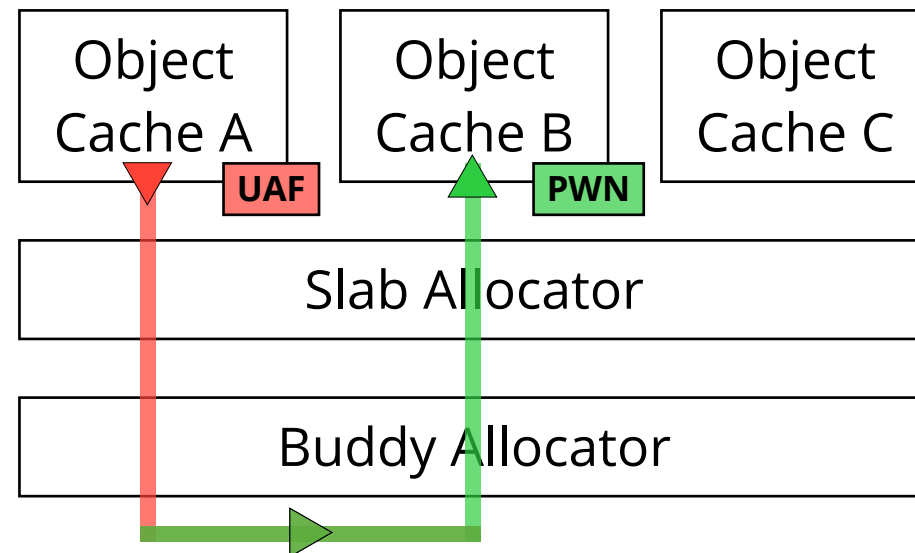
Apply **free pressure** on Cache A

↳ Page(s) containing old A are freed

Apply **alloc pressure** on Cache B

↳ Page is re-allocated for new B

↳ UAF moved from A to B



Future of Cross-cache Attacks

⋮ SLAB_VIRTUAL deterministically mitigates them (UAF) Not Deployed

↔ Memory Tagging probabillistically defends against them In Deployment

Kernel Electric Fence



Kernel Electric Fence

A sample-based memory safety error detector for **production kernels**.

Developed by Marco Elver and Alexander Potapenko (Google) in 2020-21 (5.12)

☼ Follows a similar design as userspace's **GWP-ASan**

↳ Small sample of allocations are served with sanitizer instrumentation

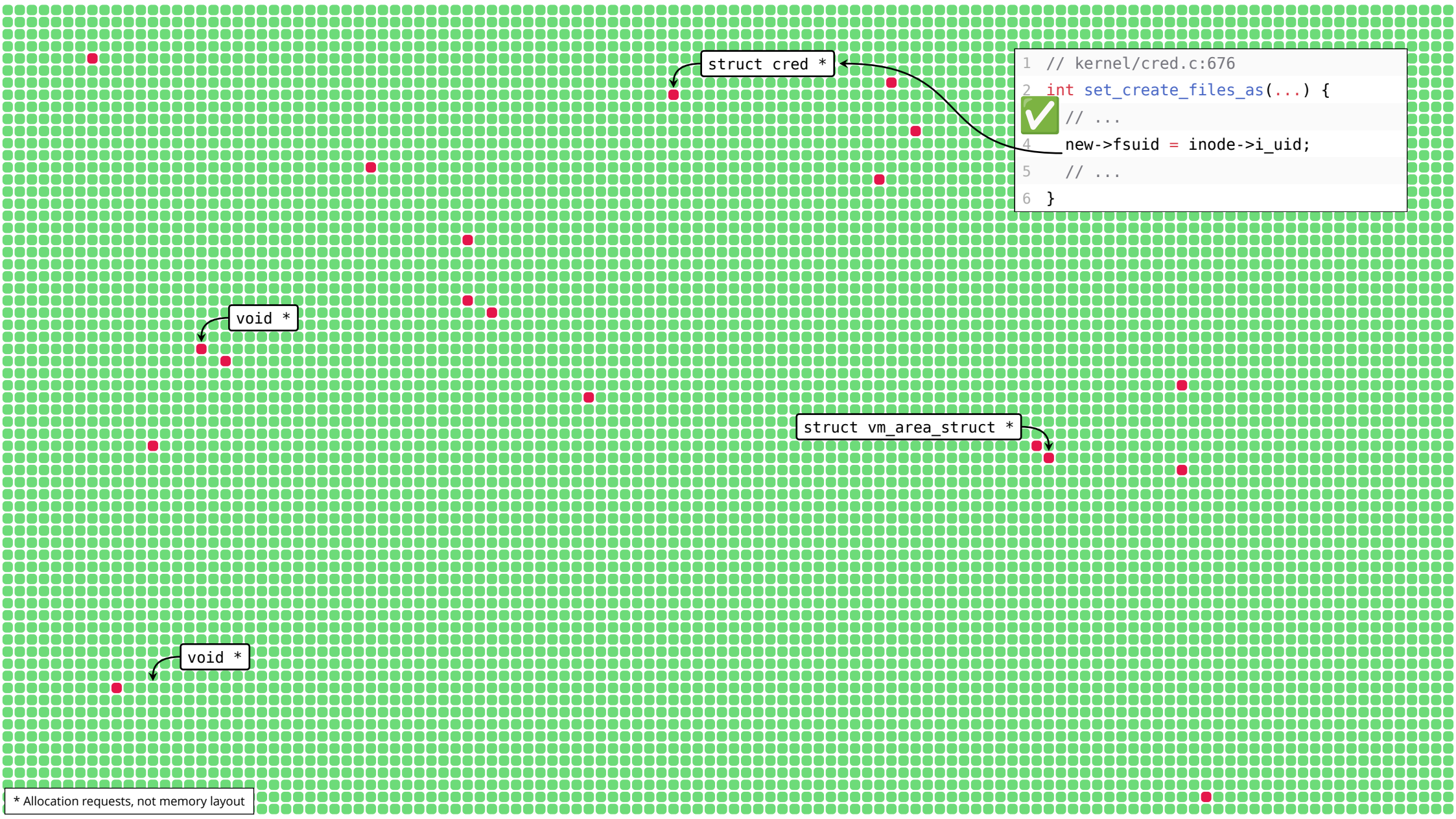
Catches complicated bugs on actual production systems

Properties

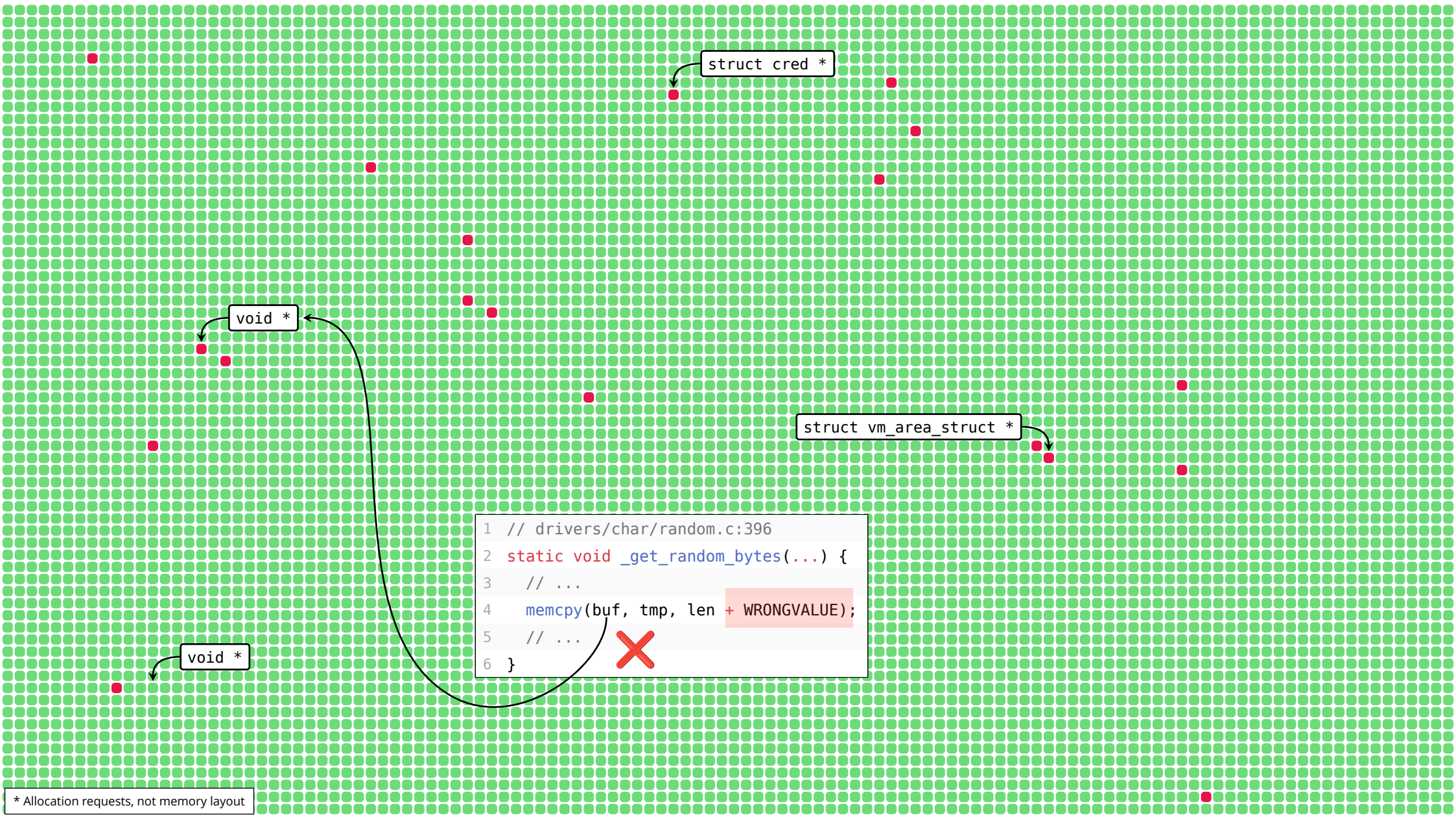
- Near zero overhead
- Probabilistic approach
- No complicated setup

Default Enabled





* Allocation requests, not memory layout



struct cred *

void *

struct vm_area_struct *

void *

```
1 // drivers/char/random.c:396
2 static void _get_random_bytes(...) {
3     // ...
4     memcpy(buf, tmp, len + WRONGVALUE);
5     // ...
6 }
```



* Allocation requests, not memory layout

struct cred *

BUG: KFENCE: use-after-free read in drm_gem_object_release_handle+0x47/0x1e0 drivers/gpu/drm/drm_gem.c:374

Use-after-free read at 0xffff88823be98e88 (in kfence-#75):
drm_gem_object_release_handle+0x47/0x1e0 drivers/gpu/drm/drm_gem.c:374
drm_gem_object_release_idr_handle+0x69/0x80 drivers/gpu/drm/drm_gem.c:403
idr_for_each+0x1c6/0x2a0 lib/idr.c:210
drm_gem_release+0x28/0x40 drivers/gpu/drm/drm_gem.c:1105
drm_file_free+0x729/0xa00 drivers/gpu/drm/drm_file.c:261
drm_close_helper drivers/gpu/drm/drm_file.c:290 [inline]
drm_release+0x2de/0x3f0 drivers/gpu/drm/drm_file.c:438
__fput+0x44f/0xa60 fs/file_table.c:510
task_work_run+0x1d9/0x270 kernel/task_work.c:233
resume_user_mode_work include/linux/resume_user_mode.h:50 [inline]
__exit_to_user_mode_loop kernel/entry/common.c:67 [inline]
exit_to_user_mode_loop+0xed/0x480 kernel/entry/common.c:98
__exit_to_user_mode_prepare include/linux/irq-entry-common.h:226 [inline]
syscall_exit_to_user_mode_prepare include/linux/irq-entry-common.h:256 [inline]
syscall_exit_to_user_mode include/linux/entry-common.h:325 [inline]
do_syscall_64+0x33e/0xf80 arch/x86/entry/syscall_64.c:100
entry_SYSCALL_64_after_hwframe+0x77/0x7f

kfence-#75: 0xffff88823be98c00-0xffff88823be98ee7, size=744, cache=kmalloc-1k

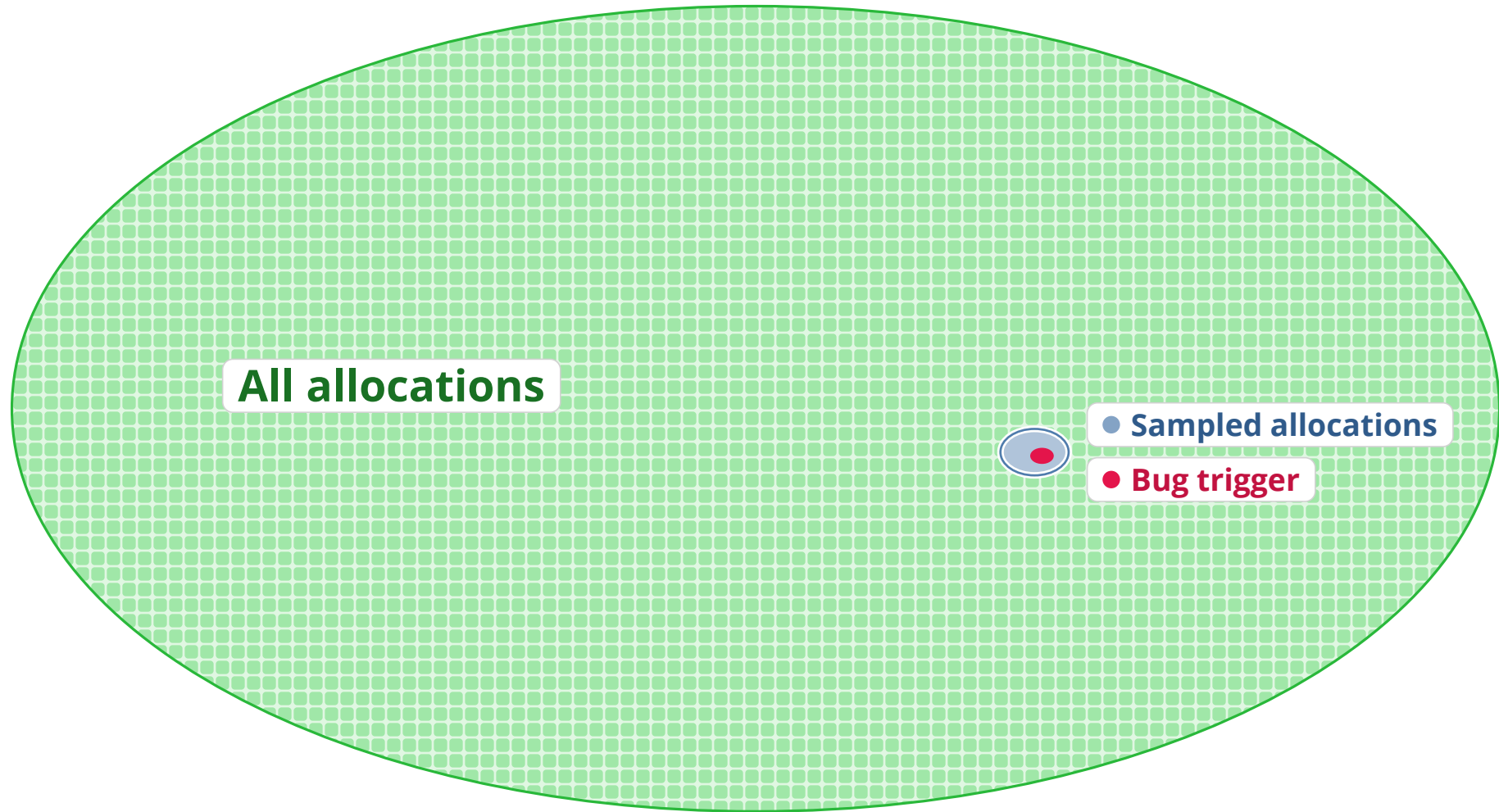
allocated by task 6583 on cpu 0 at 133.209896s (0.096709s ago):
kmalloc_noprof include/linux/slab.h:950 [inline]
kzalloc_noprof include/linux/slab.h:1188 [inline]
__drm_gem_shmem_create+0xc4/0x2e0 drivers/gpu/drm/drm_gem_shmem_helper.c:130
drm_gem_shmem_create drivers/gpu/drm/drm_gem_shmem_helper.c:157 [inline]
drm_gem_shmem_create_with_handle drivers/gpu/drm/drm_gem_shmem_helper.c:460 [inline]
drm_gem_shmem_dumb_create+0x72/0x120 drivers/gpu/drm/drm_gem_shmem_helper.c:549
drm_mode_create_dumb drivers/gpu/drm/drm_dumb_buffers.c:227 [inline]
drm_mode_create_dumb_ioctl+0x2bd/0x340 drivers/gpu/drm/drm_dumb_buffers.c:236
drm_ioctl_kernel+0x2df/0x3b0 drivers/gpu/drm/drm_ioctl.c:804
drm_ioctl+0x6ba/0xb80 drivers/gpu/drm/drm_ioctl.c:901
vfs_ioctl fs/ioctl.c:51 [inline]
__do_sys_ioctl fs/ioctl.c:597 [inline]
__se_sys_ioctl+0xfc/0x170 fs/ioctl.c:583
do_syscall_x64 arch/x86/entry/syscall_64.c:63 [inline]
do_syscall_64+0x15f/0xf80 arch/x86/entry/syscall_64.c:94
entry_SYSCALL_64_after_hwframe+0x77/0x7f

freed by task 6585 on cpu 1 at 133.211010s (0.147010s ago):
drm_gem_handle_delete+0x7b/0xb0 drivers/gpu/drm/drm_gem.c:430
drm_ioctl_kernel+0x2df/0x3b0 drivers/gpu/drm/drm_ioctl.c:804
drm_ioctl+0x6ba/0xb80 drivers/gpu/drm/drm_ioctl.c:901
vfs_ioctl fs/ioctl.c:51 [inline]
__do_sys_ioctl fs/ioctl.c:597 [inline]
__se_sys_ioctl+0xfc/0x170 fs/ioctl.c:583
do_syscall_x64 arch/x86/entry/syscall_64.c:63 [inline]
do_syscall_64+0x15f/0xf80 arch/x86/entry/

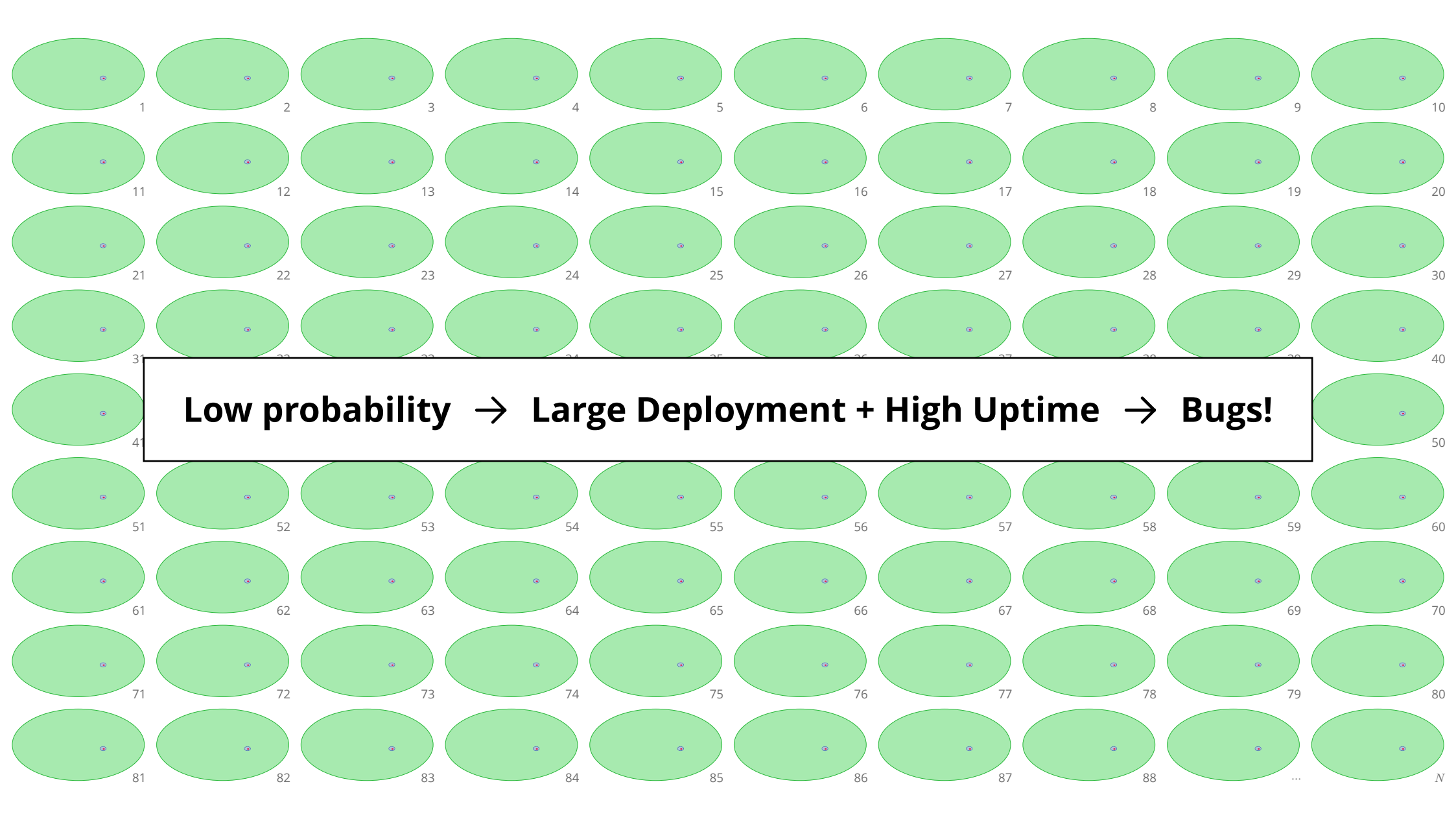
void *

void *

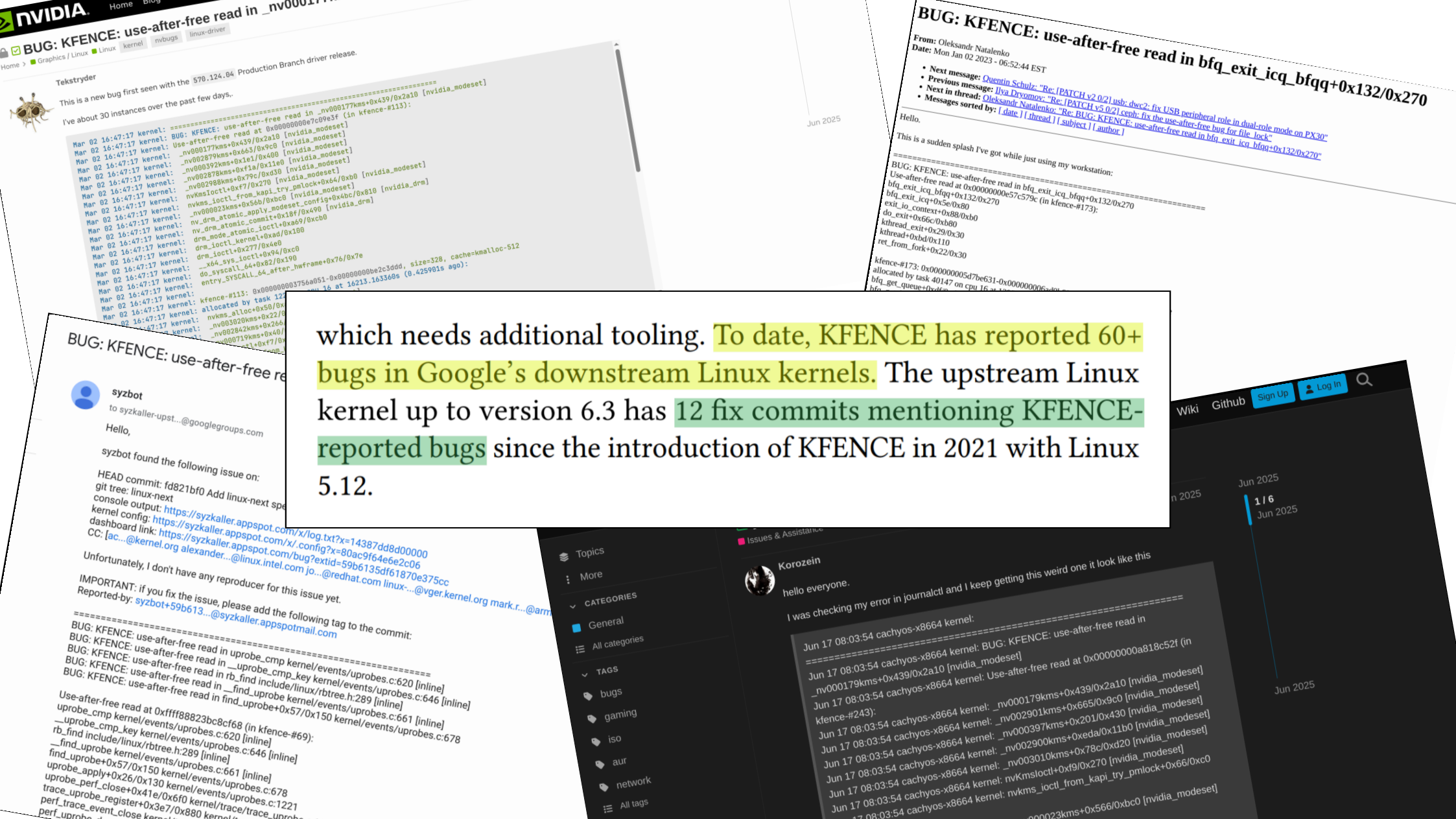
* Sample



- Most allocations are fast
- Low probability of detection → sampled AND bug trigger



Low probability → Large Deployment + High Uptime → Bugs!



which needs additional tooling. To date, KFENCE has reported 60+ bugs in Google's downstream Linux kernels. The upstream Linux kernel up to version 6.3 has 12 fix commits mentioning KFENCE-reported bugs since the introduction of KFENCE in 2021 with Linux 5.12.

Every **100ms**, KFENCE short-circuits the Slab allocator for a single allocation

1. slab_alloc_node()

```
1 object = kfence_alloc(s, orig_size, ...);
2 if (unlikely(object))
3     goto out;
4 object = __slab_alloc_node(s, ...);
```

2. kfence_alloc()

```
1 if (atomic_rd(&kfence_allocation_gate) > 0)
2     return NULL;
3
4 return __kfence_alloc(s, size, flags);
```

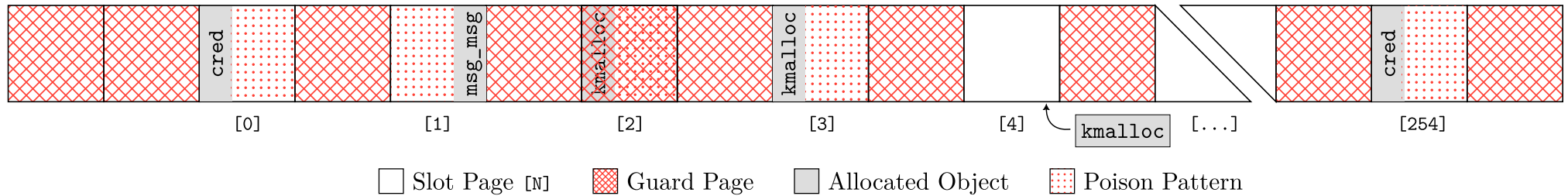
3. toggle_allocation_gate()

```
1 // INIT_DELAYED_WORK(&kfence_timer, toggle_allocation_gate);
2 atomic_set(&kfence_allocation_gate, -kfence_burst);
3 // ...
4 queue_delayed_work(..., &kfence_timer, msecs_to_jiffies(kfence_sample_interval));
```

Allocation Pool

Served objects are placed in the KFENCE pool: **__kfence_pool**

Objects are placed in **slot pages** surrounded by **guard pages**



$$(255 + 1) \times 2 \times 4096 = 2\text{MiB}$$

Slot pages hold a **single object** placed at the start or the end of the page

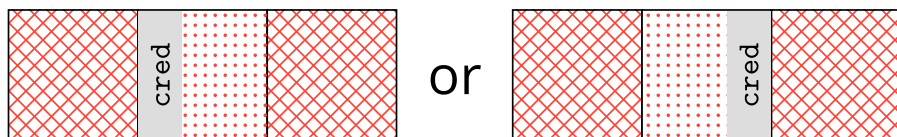
Guard pages trap on access

```
1 if (protect)
2   set_pte(pte, __pte(pte_val(*pte) & ~_PAGE_PRESENT));
```

Memory Error Detection

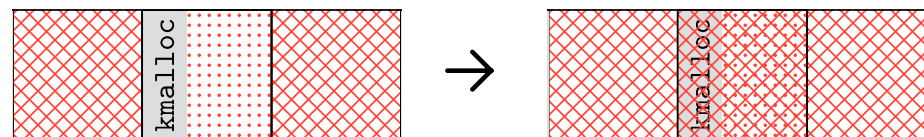
Spatial and temporal memory safety errors are detected on sampled objects

Spatial Memory Safety



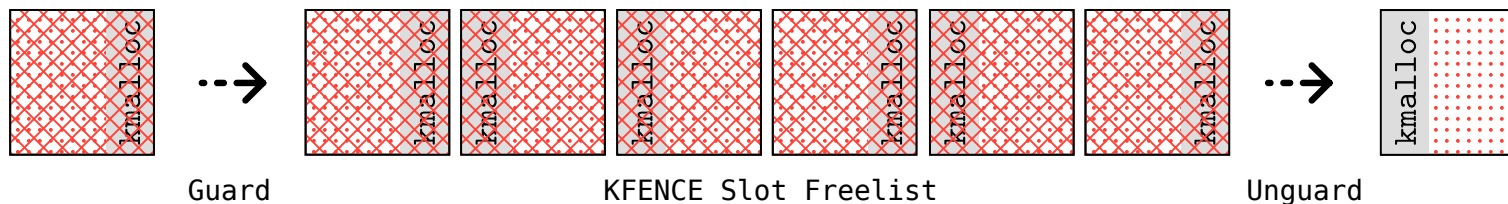
- Randomized start/end placement
- Poison checked on free

Temporal Memory Safety



- Freed objects are guarded
- Metadata checks: cache, addr, etc.

🛡️ Maximizes temporal safety with a **FIFO** freelist:



KFENCE has a special pressure mode, activated at **75% pool occupancy**

Solves the problem of frequently allocated objects hoarding KFENCE

↳ No coverage on less frequent paths 😞

After pressure mode kicks in:

```
__alloc_skb+0x72/0x450
iwl_mld_rx_mpdu+0x12a/0x700 [iwlmlld]
iwl_pcie_rx_handle_rb.isra.0+0x202/0x430
iwl_pcie_rx_handle+0x1ef/0x650 [iwlwifi]
iwl_pcie_napi_poll_msix+0x3b/0xf0 [iwlwifi]
__napi_poll+0x10f/0x120
net_rx_action+0x10/0x30
handle_softirq+0x10/0x30
do_softirq+0x10/0x30
__local_bh_enable_ip+0x10/0x70
iwl_pcie_irq_rx_msix_handler+0xdc/0x150 [iwlwifi]
irq_thread_fn+0x25/0x60
irq_thread+0xc7/0x1c0
kthread+0xe4/0x120
ret_from_fork+0x1a1/0x270
ret_from_fork_asm+0x1a/0x30
```

Allocation A

```
xhci_urb_enqueue+0x6e/0x390
usb_hcd_submit_urb+0xac/0x330
intr_callback+0x80/0x200 [r8152]
__usb_hcd_giveback_urb+0xa0/0x120
usb_giveback_urb_bh+0xb3/0x150
process_one_work+0x10/0x370
bh_worker+0x1cb/0x200
tasklet_hi_execute+0x13/0x30
handle_softirq+0x10/0x30
__irq_exit+0x10/0x30
common_interrupt+0x85/0xa0
asm_common_interrupt+0x26/0x40
cpuidle_enter_state+0xcd/0x650
cpuidle_enter+0x31/0x50
cpuidle_idle_call+0x10f/0x200
do_idle+0x10/0x120
cpu_startup_entry+0x29/0x30
```

Allocation B

```
__alloc_skb+0x72/0x450
iwl_mld_rx_mpdu+0x12a/0x700 [iwlmlld]
iwl_pcie_rx_handle_rb.isra.0+0x202/0x430
iwl_pcie_rx_handle+0x1ef/0x650 [iwlwifi]
iwl_pcie_napi_poll_msix+0x3b/0xf0 [iwlwifi]
__napi_poll+0x10f/0x120
net_rx_action+0x10/0x30
handle_softirq+0x10/0x30
do_softirq+0x10/0x30
__local_bh_enable_ip+0x10/0x70
iwl_pcie_irq_rx_msix_handler+0xdc/0x150 [iwlwifi]
irq_thread_fn+0x25/0x60
irq_thread+0xc7/0x1c0
kthread+0xe4/0x120
ret_from_fork+0x1a1/0x270
ret_from_fork_asm+0x1a/0x30
```

Allocation C

```
[ecomaikgolf@laptop ~/]$ uptime -p  
up 4 days, 13 hours, 41 minutes
```

```
[ecomaikgolf@laptop ~/]$ sudo cat /sys/kernel/debug/kfence/stats  
enabled: 1  
currently allocated: 191  
total allocations: 1466277  
total frees: 1466086  
[...]  
skipped allocations (incompatible): 729  
skipped allocations (capacity): 0  
skipped allocations (covered): 22066
```

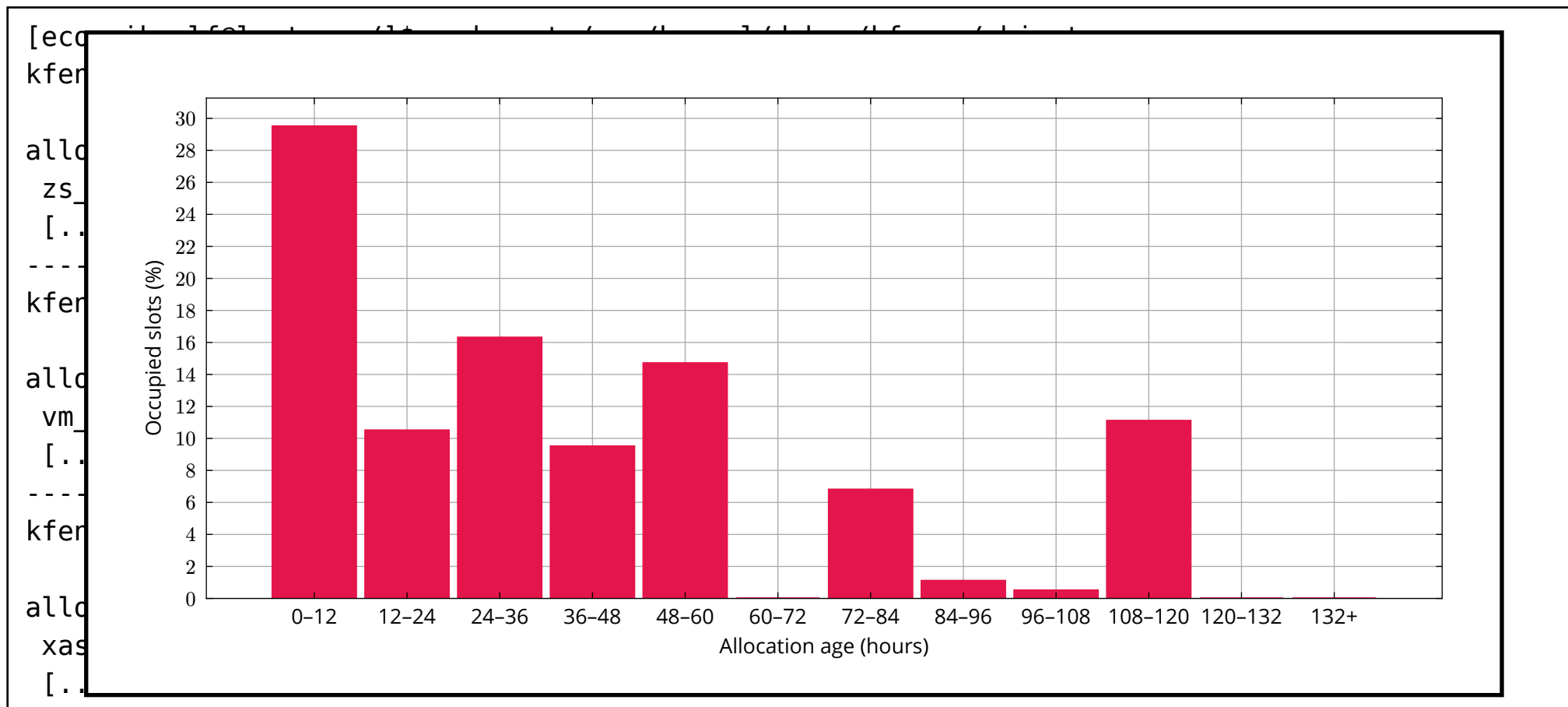
- **1.4M** allocations and frees
- **75%** currently allocated slots
- **1.5%** of total allocations skipped due to pressure mode

```
[ecomaikgolf@laptop ~/]$ sudo cat /sys/kernel/debug/kfence/objects
kfence-#0: 0x0000000019fd7bdf-0x00000000927e0e5c, size=8, cache=zs_handle

allocated by task 101 on cpu 0 at 224723.303159s (169444.589377s ago):
  zs_malloc+0x5d/0x3d0
  [...]
-----
kfence-#1: 0x00000000c2abdc15-0x00000000f4996c0d, size=192, cache=vm_area_struct

allocated by task 288050 on cpu 3 at 265381.908716s (128785.983849s ago):
  vm_area_alloc+0x1e/0x80
  [...]
-----
kfence-#2: 0x00000000ac5cb19a-0x000000002eb918d0, size=576, cache=radix_tree_node

allocated by task 265524 on cpu 7 at 368668.334847s (25499.557727s ago):
  xas_alloc+0xa3/0xd0
  [...]
```



Security Analysis



Heap Segregation

KFENCE cannot honor heap segregation with a limited global memory pool

We employ Slab caches to segregate the heap for security (amongst others):

cred

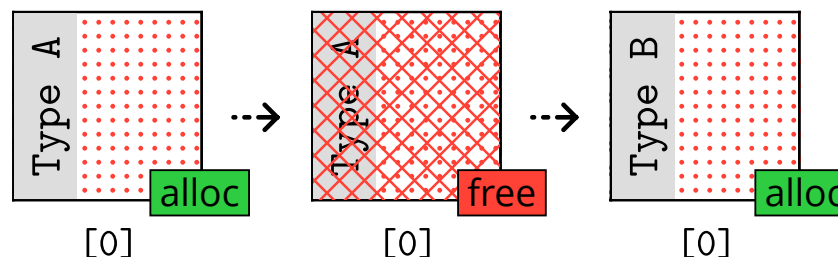
kmalloc-* - 128

msg_msg - 96

task_struct

Attackers must do in-cache (limited) or cross-cache (probabilistic) attacks

Contrary, **KFENCE does not perform heap segregation:**



Furthermore, no size segregation gives attackers have more freedom

```
# grep kfence /sys/kernel/debug/kfence/objects | head -n 30
```

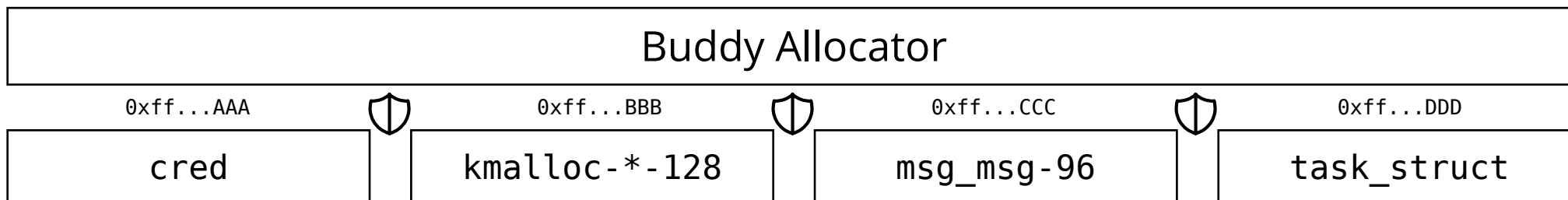
```
kfence-#0: 0xfffffa4203b402000-0xfffffa4203b4022bf, size=704, cache=skbuff_small_head
kfence-#1: 0xfffffa4203b404d40-0xfffffa4203b404fff, size=704, cache=skbuff_small_head
kfence-#2: 0xfffffa4203b406000-0xfffffa4203b4062bf, size=704, cache=skbuff_small_head
kfence-#3: 0xfffffa4203b408d40-0xfffffa4203b408fff, size=704, cache=skbuff_small_head
kfence-#4: 0xfffffa4203b40a000-0xfffffa4203b40afff, size=4096, cache=names_cache
kfence-#5: 0xfffffa4203b40c000-0xfffffa4203b40c2bf, size=704, cache=skbuff_small_head
kfence-#6: 0xfffffa4203b40efc0-0xfffffa4203b40efff, size=64, cache=kmalloc-rnd-14-64
kfence-#7: 0xfffffa4203b410f00-0xfffffa4203b410fdf, size=224, cache=skbuff_head_cache
kfence-#8: 0xfffffa4203b412d40-0xfffffa4203b412fff, size=704, cache=skbuff_small_head
kfence-#9: 0xfffffa4203b414000-0xfffffa4203b4142bf, size=704, cache=skbuff_small_head
kfence-#10: 0xfffffa4203b416000-0xfffffa4203b4162bf, size=704, cache=skbuff_small_head
kfence-#11: 0xfffffa4203b418d40-0xfffffa4203b418fff, size=704, cache=skbuff_small_head
kfence-#12: 0xfffffa4203b41a000-0xfffffa4203b41a2bf, size=704, cache=skbuff_small_head
kfence-#13: 0xfffffa4203b41cd40-0xfffffa4203b41cfff, size=704, cache=skbuff_small_head
kfence-#14: 0xfffffa4203b41ed40-0xfffffa4203b41efff, size=704, cache=skbuff_small_head
kfence-#15: 0xfffffa4203b420d40-0xfffffa4203b420fff, size=704, cache=skbuff_small_head
kfence-#16: 0xfffffa4203b422d40-0xfffffa4203b422fff, size=704, cache=skbuff_small_head
kfence-#17: 0xfffffa4203b424d40-0xfffffa4203b424fff, size=704, cache=skbuff_small_head
kfence-#18: 0xfffffa4203b426d40-0xfffffa4203b426fff, size=704, cache=skbuff_small_head
kfence-#19: 0xfffffa4203b428d40-0xfffffa4203b428fff, size=704, cache=skbuff_small_head
kfence-#20: 0xfffffa4203b42a000-0xfffffa4203b42a00f, size=16, cache=kmalloc-rnd-14-16
kfence-#21: 0xfffffa4203b42cd40-0xfffffa4203b42cfff, size=704, cache=skbuff_small_head
kfence-#22: 0xfffffa4203b42e000-0xfffffa4203b42e2bf, size=704, cache=skbuff_small_head
kfence-#23: 0xfffffa4203b430000-0xfffffa4203b4302bf, size=704, cache=skbuff_small_head
kfence-#24: 0xfffffa4203b432000-0xfffffa4203b4322bf, size=704, cache=skbuff_small_head
```

```

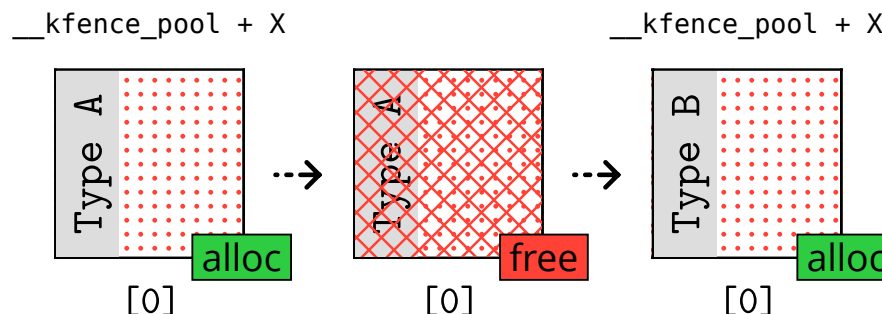
[140] maple_node (L) -> pid (L) -> cred (R) -> kmalloc-rnd-06-96 (R) -> cred (L) -> msg_msg-64 (L) -> msg_msg-64 (L) -> cred (R) -> cred (R) -> cred (R)
[141] maple_node (R) -> cred (R) -> cred (L) -> vm_area_struct (L) -> cred (L) -> msg_msg-64 (R) -> msg_msg-64 (R) -> cred (R) -> msg_msg-64 (L) -> msg_msg-64 (L) -> cred (L) -> cred (L)
[142] maple_node (L) -> cred (L) -> cred (L) -> kmalloc-rnd-06-96 (R) -> cred (L) -> msg_msg-64 (R) -> msg_msg-64 (R) -> vm_area_struct (L) -> msg_msg-64 (L) -> msg_msg-64 (L) -> vm_area_struct (R) -> msg_msg-64 (R) -> cred (R) -> cred (R) -> cred (R)
[143] maple_node (R) -> maple_node (L) -> cred (R) -> sigqueue (L) -> cred (R) -> msg_msg-64 (L) -> msg_msg-64 (R) -> cred (R) -> msg_msg-64 (R) -> msg_msg-64 (L) -> vm_area_struct (L) -> kmalloc-rnd-10-8 (R) -> cred (R) -> cred (L) -> cred (L)
[144] maple_node (L) -> cred (L) -> cred (L) -> cred (L) -> cred (R) -> msg_msg-64 (L) -> msg_msg-64 (R) -> cred (R) -> msg_msg-64 (L) -> msg_msg-64 (L) -> cred (L) -> msg_msg-64 (R) -> cred (L) -> cred (R)
[145] maple_node (R) -> cred (R) -> cred (L) -> kmalloc-rnd-06-128 (L) -> cred (L) -> msg_msg-64 (L) -> msg_msg-64 (L) -> cred (L) -> msg_msg-64 (R) -> msg_msg-64 (R) -> cred (L) -> kmalloc-rnd-10-8 (R) -> cred (R) -> maple_node (L)
[146] maple_node (R) -> cred (R) -> cred (R) -> cred (R) -> cred (R) -> msg_msg-64 (R) -> msg_msg-64 (L) -> cred (L) -> msg_msg-64 (L) -> msg_msg-64 (L) -> cred (R) -> msg_msg-64 (L) -> vm_area_struct (R) -> vm_area_struct (R)
[147] maple_node (R) -> vm_area_struct (R) -> vm_area_struct (L) -> cred (L) -> cred (L) -> msg_msg-64 (R) -> msg_msg-64 (L) -> cred (R) -> msg_msg-64 (R) -> maple_node (R) -> cred (L) -> msg_msg-64 (R) -> cred (L) -> cred (L)
[148] vm_area_struct (R) -> cred (L) -> cred (L) -> cred (R) -> cred (L) -> msg_msg-64 (R) -> msg_msg-64 (L) -> cred (L) -> msg_msg-64 (L) -> vm_area_struct (R) -> vm_area_struct (R) -> msg_msg-64 (L) -> cred (R) -> cred (L)
[149] maple_node (R) -> fs_cache (L) -> cred (L) -> pid (R) -> cred (R) -> msg_msg-64 (L) -> msg_msg-64 (L) -> cred (R) -> msg_msg-64 (L) -> vm_area_struct (R) -> pid (R) -> msg_msg-64 (L) -> cred (R) -> cred (L) -> cred (L)
[150] maple_node (L) -> cred (L) -> cred (R) -> cred (R) -> mm_struct (R) -> msg_msg-64 (L) -> msg_msg-64 (L) -> sigqueue (L) -> vm_area_struct (L) -> skbuff_fclone_cache (R) -> cred (L) -> msg_msg-64 (R) -> cred (L) -> cred (R)
[151] maple_node (L) -> skbuff_small_head (R) -> anon_vma (L) -> cred (L) -> vm_area_struct (R) -> cred (R) -> msg_msg-64 (R) -> maple_node (R) -> msg_msg-64 (R) -> cred (L) -> msg_msg-64 (R) -> cred (R) -> cred (R) -> cred (R)
[152] maple_node (R) -> cred (R) -> cred (L) -> cred (R) -> cred (R) -> cred (L) -> msg_msg-64 (L) -> sigqueue (R) -> skbuff_fclone_cache (L) -> msg_msg-64 (R) -> cred (R) -> msg_msg-64 (R) -> cred (L) -> cred (L)
[153] maple_node (R) -> vm_area_struct (L) -> cred (L) -> fs_cache (R) -> cred (L) -> cred (L) -> msg_msg-64 (R) -> cred (L) -> msg_msg-64 (R) -> msg_msg-64 (R) -> cred (L) -> msg_msg-64 (L) -> cred (L) -> cred (L)
[154] maple_node (L) -> cred (R) -> cred (L) -> cred (R) -> cred (L) -> msg_msg-64 (R) -> cred (R) -> kmalloc-rnd-10-8 (R) -> cred (L) -> msg_msg-64 (R) -> files_cache (L) -> sigqueue (R)
[155] maple_node (R) -> files_cache (R) -> mm_struct (R) -> cred (R) -> cred (L) -> cred (R) -> msg_msg-64 (L) -> vm_area_struct (R) -> msg_msg-64 (L) -> msg_msg-64 (R) -> maple_node (L) -> msg_msg-64 (R) -> cred (L) -> cred (L)
[156] maple_node (L) -> cred (L) -> cred (L) -> cred (L) -> cred (R) -> cred (R) -> maple_node (L) -> cred (R) -> msg_msg-64 (R) -> msg_msg-64 (R) -> vm_area_struct (L) -> msg_msg-64 (L) -> msg_msg-64 (R) -> cred (L)
[157] maple_node (L) -> sigqueue (L) -> cred (L) -> cred (L) -> cred (L) -> cred (R) -> cred (R) -> msg_msg-64 (L) -> vm_area_struct (L) -> msg_msg-64 (L) -> msg_msg-64 (L) -> cred (R) -> msg_msg-64 (L) -> cred (L)
[158] maple_node (R) -> sigqueue (L) -> cred (R) -> cred (L) -> vm_area_struct (R) -> pid (R) -> msg_msg-64 (R) -> sigqueue (R) -> msg_msg-64 (R) -> msg_msg-64 (L) -> cred (L) -> vm_area_struct (R) -> msg_msg-64 (L) -> cred (R)
[159] ... -> skbuff_small_head (R) -> msg_msg-64 (R) -> mm_struct (L) -> vm_area_struct (L) -> cred (L) -> kmalloc-rnd-10-8 (R) -> maple_node (R) -> kmalloc-rnd-10-8 (L) -> msg_msg-64 (R) -> sigqueue (R) -> vm_area_struct (L) -> msg_msg-64 (R) -> fs_cache (R)
[160] maple_node (L) -> sigqueue (L) -> msg_msg-64 (R) -> cred (L) -> cred (L) -> vm_area_struct (L) -> msg_msg-64 (R) -> cred (R) -> msg_msg-64 (L) -> msg_msg-64 (L) -> cred (L) -> msg_msg-64 (L) -> cred (R)
[161] maple_node (R) -> cred (L) -> msg_msg-64 (L) -> cred (L) -> maple_node (L) -> cred (L) -> msg_msg-64 (R) -> cred (L) -> sigqueue (L) -> msg_msg-64 (R) -> cred (R) -> msg_msg-64 (R) -> kmalloc-rnd-10-8 (L) -> cred (L)
[162] maple_node (L) -> cred (R) -> msg_msg-64 (R) -> cred (R) -> kmalloc-rnd-06-96 (R) -> cred (L) -> kmalloc-rnd-10-8 (R) -> cred (L) -> msg_msg-64 (L) -> msg_msg-64 (L) -> cred (R) -> msg_msg-64 (R) -> kmalloc-rnd-10-8 (R) -> cred (L)
[163] vm_area_struct (R) -> pid (R) -> msg_msg-64 (R) -> kmalloc-rnd-06-96 (R) -> cred (L) -> kmalloc-rnd-06-96 (L) -> msg_msg-64 (L) -> cred (R) -> msg_msg-64 (L) -> msg_msg-64 (L) -> vm_area_struct (R) -> msg_msg-64 (R) -> kmalloc-rnd-10-8 (R) -> cred (L)
[164] vm_area_struct (L) -> cred (L) -> msg_msg-64 (R) -> maple_node (R) -> cred (L) -> cred (L) -> msg_msg-64 (L) -> cred (L) -> msg_msg-64 (R) -> maple_node (R) -> msg_msg-64 (L) -> msg_msg-64 (L) -> pid (L)
[165] vm_area_struct (L) -> sigqueue (R) -> maple_node (R) -> kmalloc-rnd-06-96 (R) -> cred (R) -> cred (L) -> msg_msg-64 (L) -> cred (R) -> kmalloc-rnd-10-8 (R) -> kmalloc-rnd-10-8 (L) -> cred (L) -> msg_msg-64 (L) -> msg_msg-64 (R) -> cred (R)
[166] vm_area_struct (L) -> cred (L) -> vm_area_struct (R) -> cred (R) -> cred (R) -> cred (L) -> msg_msg-64 (R) -> cred (L) -> msg_msg-64 (L) -> msg_msg-64 (R) -> sigqueue (L) -> msg_msg-64 (R) -> msg_msg-64 (L) -> signal_cache (L)
[167] vm_area_struct (L) -> maple_node (R) -> msg_msg-64 (L) -> cred (R) -> cred (L) -> sigqueue (L) -> msg_msg-64 (R) -> sigqueue (L) -> msg_msg-64 (R) -> msg_msg-64 (L) -> cred (R) -> msg_msg-64 (R) -> msg_msg-64 (R) -> cred (L)
[168] vm_area_struct (R) -> vm_area_struct (L) -> vm_area_struct (R) -> cred (R) -> cred (L) -> kmalloc-rnd-10-8 (R) -> cred (L) -> msg_msg-64 (R) -> msg_msg-64 (R) -> cred (L) -> msg_msg-64 (L) -> kmalloc-rnd-10-8 (R) -> vm_area_struct (R)
[169] vm_area_struct (R) -> cred (L) -> skbuff_fclone_cache (L) -> cred (R) -> mm_struct (L) -> cred
```

KFENCE does not honor SLAB_VIRTUAL hardened segregation guarantees

SLAB_VIRTUAL defends against cross-cache attacks by partitioning the VA space:



Cross-cache **use-after-free** exploitation can be **revived** under KFENCE



```
1 while(1) {
2     alloc_msg();
3     alloc_cred();
4 }
```

Objects served by KFENCE are not instrumented with KASAN

Most importantly, not under **KASAN_HW_TAGS** (MTE) production mode

```
1 void * __kasan_slab_alloc(..., void *object)
2 {
3     // ...
4     if (is_kfence_address(object))
5         return (void *)object;
6
7     tag = assign_tag(cache, object, false);
8     tagged_object = set_tag(object, tag);
9     // ...
10    return tagged_object;
11 }
```

Kernel

```
1 switch (cmd) {  
2     case TEST_IOCTL_ALLOC:  
3         // ...  
4         new_buf = kmalloc(alloc_req.size, GFP_KERNEL);  
5         // ...  
6         if (copy_to_user((void __user *)arg, ..., ...))  
7             BUG_ON(1);  
8  
9         return 0;  
}
```

+

User

```
1 while(1) {  
2     printf(..., alloc_kmalloc(...));  
3     usleep(1000); //1ms  
4 }
```



```
# setprop arm64.memtag.bootctl memtag,memtag-kernel
```

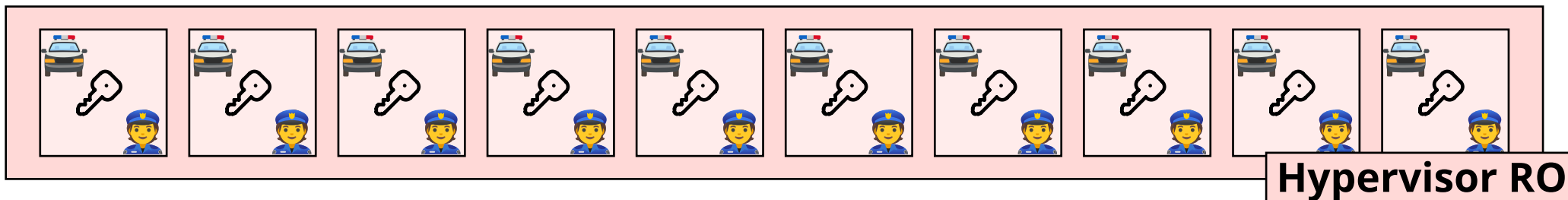
```
kmalloc(..., GFP_KERNEL) -> 0xf8ffff8928695700 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xf0ffff88819e0040 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xfdffff89179fcb40 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xf4ffff89179fcb40 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xfaffff804e5eea40 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xf9ffff89179fcb40 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xf2ffff89179fcb40 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xf7ffff89179fcb40 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xffffffff896a381fc0 (kfence) [NO MTE]
kmalloc(..., GFP_KERNEL) -> 0xfbffff804e5eec80 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xf7ffff8913860200 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xf2ffff8913860200 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xfdffff8913860200 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xfcffff89179fcb40 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xf0ffff89179fcb40 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xf8ffff89179fcb40 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xf9ffff89179fcb40 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xf0ffff89179fcb40 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xf9ffff89179fcb40 (slab ) [MTE]
[...]
kmalloc(..., GFP_KERNEL) -> 0xf1ffff8926faf800 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xfcffff8860329700 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xfcffff8926faf800 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xffffffff896a353fc0 (kfence) [NO MTE]
kmalloc(..., GFP_KERNEL) -> 0xf4ffff8926faf800 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xf2ffff8860329700 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xf0ffff8860329700 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xf0ffff8860329700 (slab ) [MTE]
kmalloc(..., GFP_KERNEL) -> 0xfaffff89277ebc00 (slab ) [MTE]
```

Downstream Hardening

KFENCE may weaken downstream hardening in unexpected ways

Objects are not always backed by the same type of memory

Hypervisors such as **Samsung RKP** mark sensitive slab pages as **read-only**:



If you harden certain caches, you may need to **avoid KFENCE sampling**:

```
/sys/kernel/slab/cred/skip_kfence
```

Other protections may have different quirks

1

P



P

[

1

7

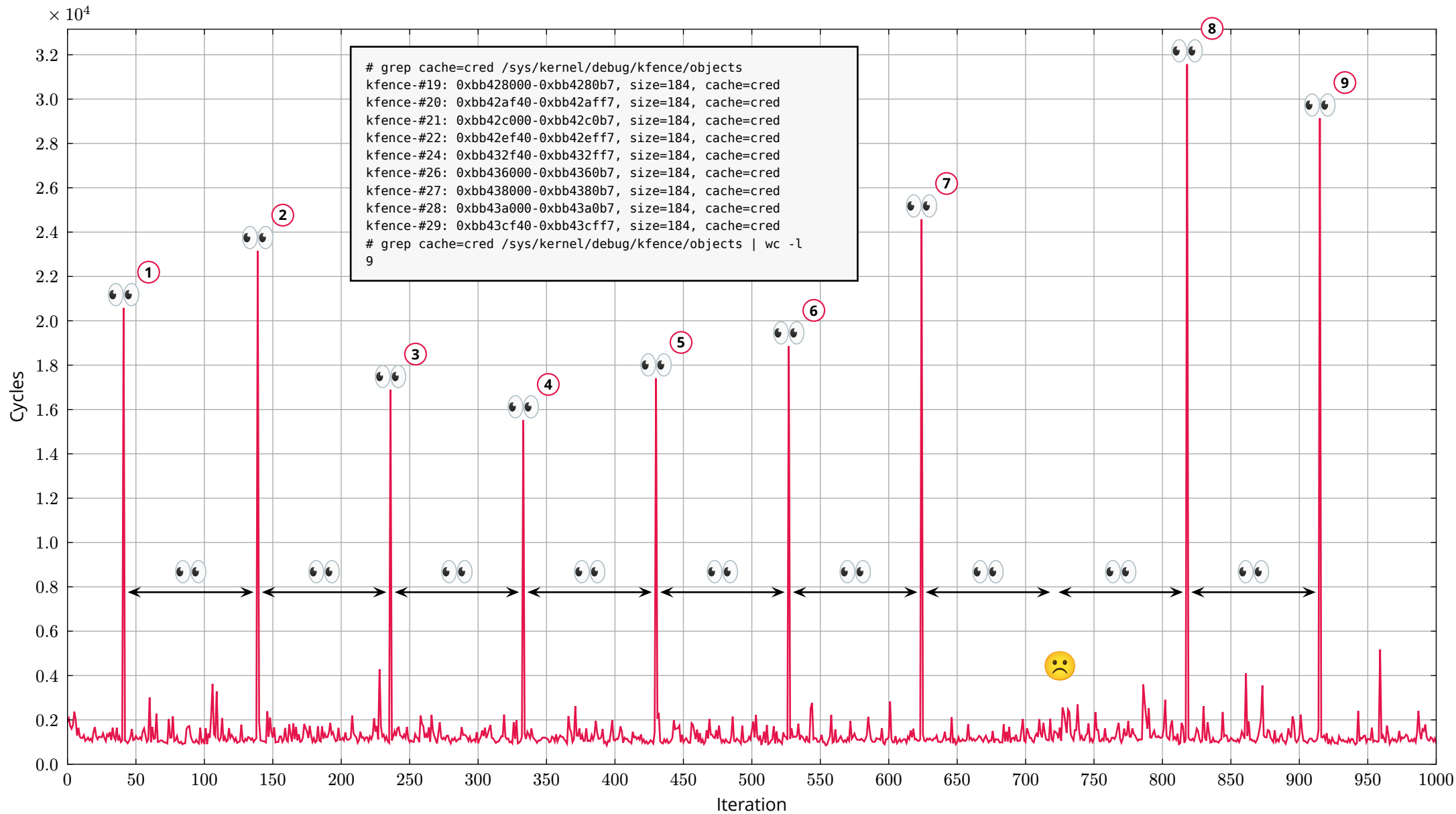
ref	
-----	--

Fence2Pwn

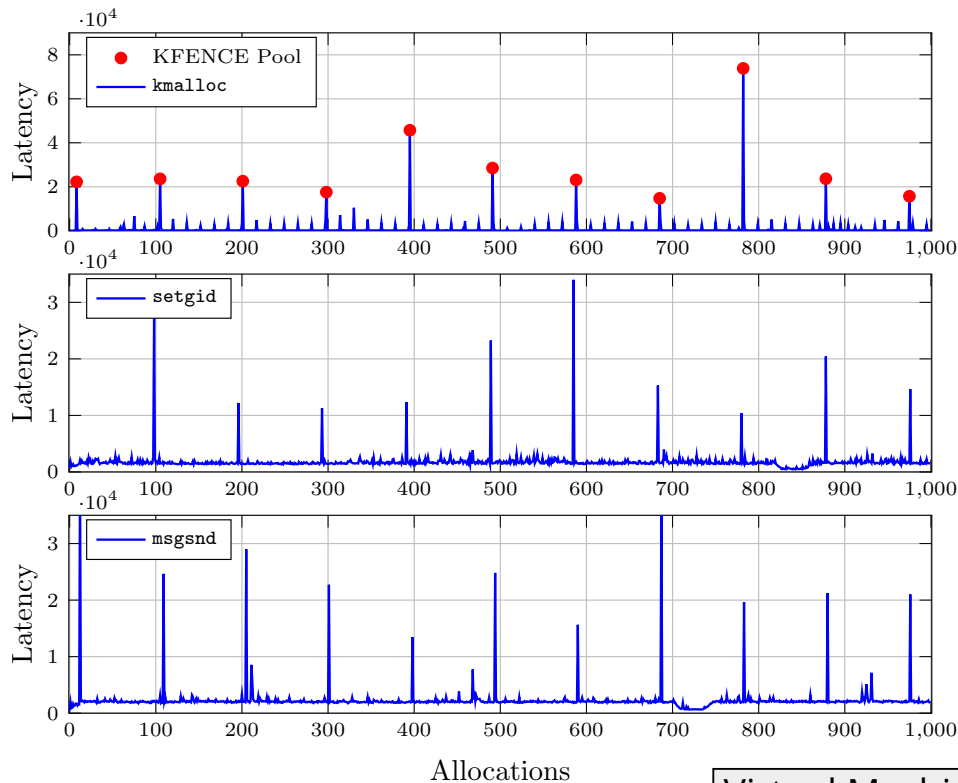


User

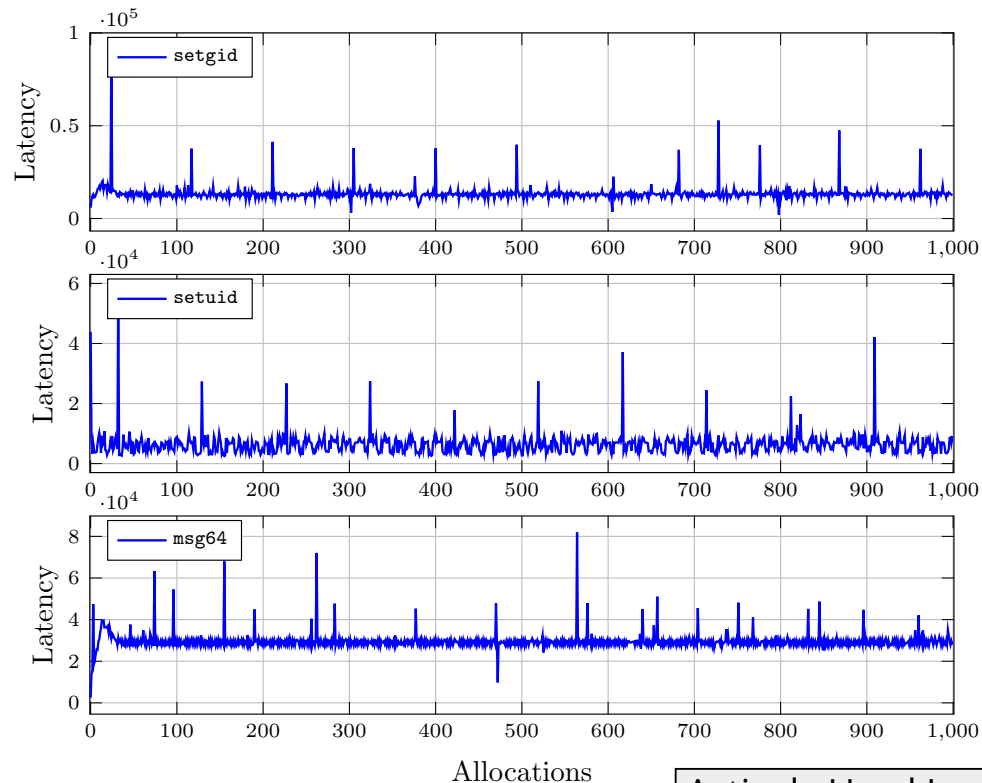
```
1 pin_cpu(0);
2 while(1) {
3     a = rdtsc();
4     setgid(gid);
5     b = rdtsc();
6     printf(..., b - a);
7     nanosleep(&(struct timespec){.tv_nsec = 1000000}, NULL); // 1ms
8 }
```



Attacker may **pick KFENCE** allocations, **predict and spray** future windows

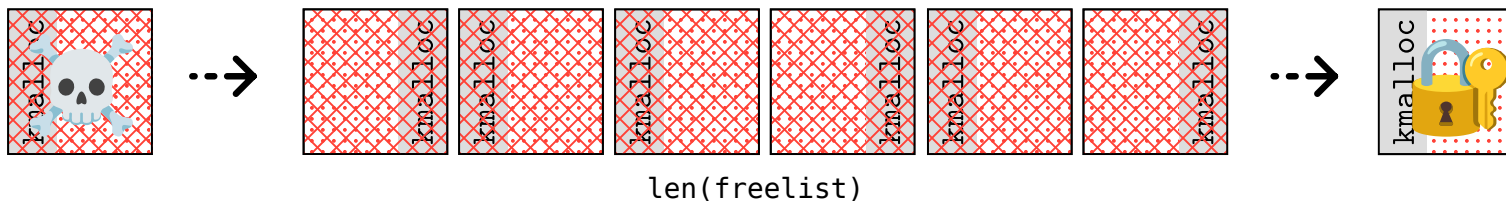


Virtual Machine



Actively Used Laptop

Use-after-free reclaiming on a FIFO freelist requires guessing its length



$$\text{free(UAF)} > (\text{alloc} + \text{free}) * N > \text{alloc}()$$

Option 1: Pressure Mode

If KFENCE stops serving same-callstack allocations, pressure mode has started

↳ This means 75% occupation → freelist 25% → 63 alloc + free

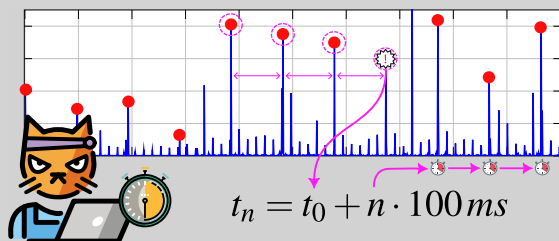
Option 2: Assume 25%

KFENCE tends to range around the pressure-mode cutoff → ~63 alloc + free

A cache-agnostic object-overlapping technique that bypasses Slab

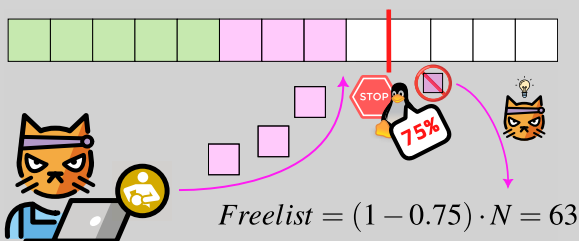
Fence2Pwn

Step 1: Calibration (Section 5.1)



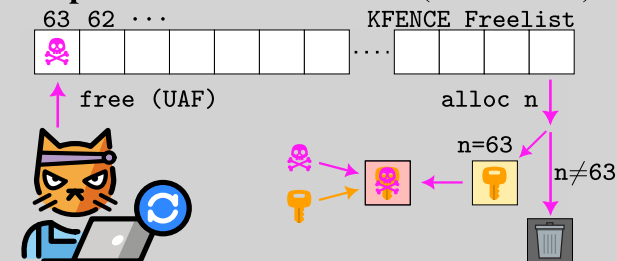
Target KFENCE

Step 2: Massaging (Section 5.2)



Estimate len.freelist)

Step 3: Rotation (Section 5.3)

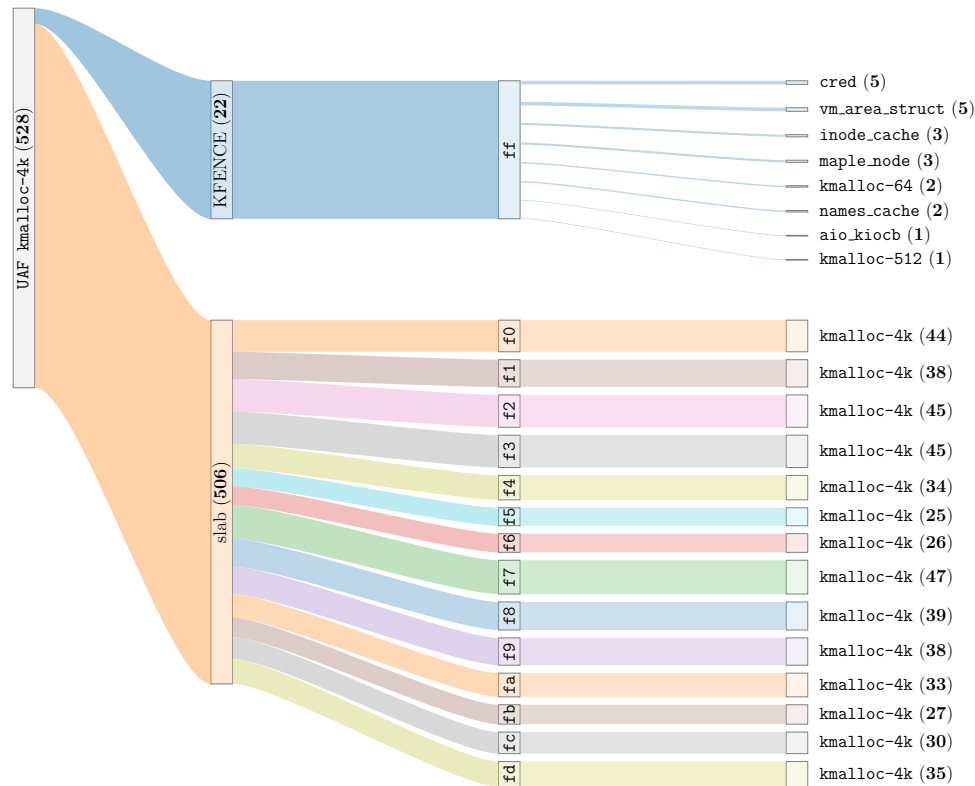
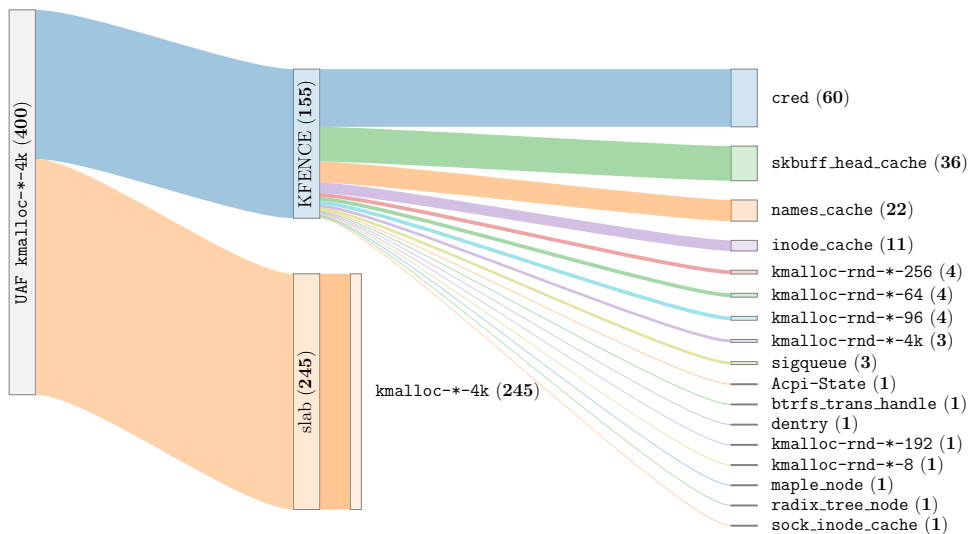


Discard N + Reclaim

Failed attempts due to non-overlapping KFENCE objects do not crash

↳ An **attacker can retry indefinitely** until good 1/2 random placement

Object type diagram of **reclaimed memory** from previous synthetic **UAF**



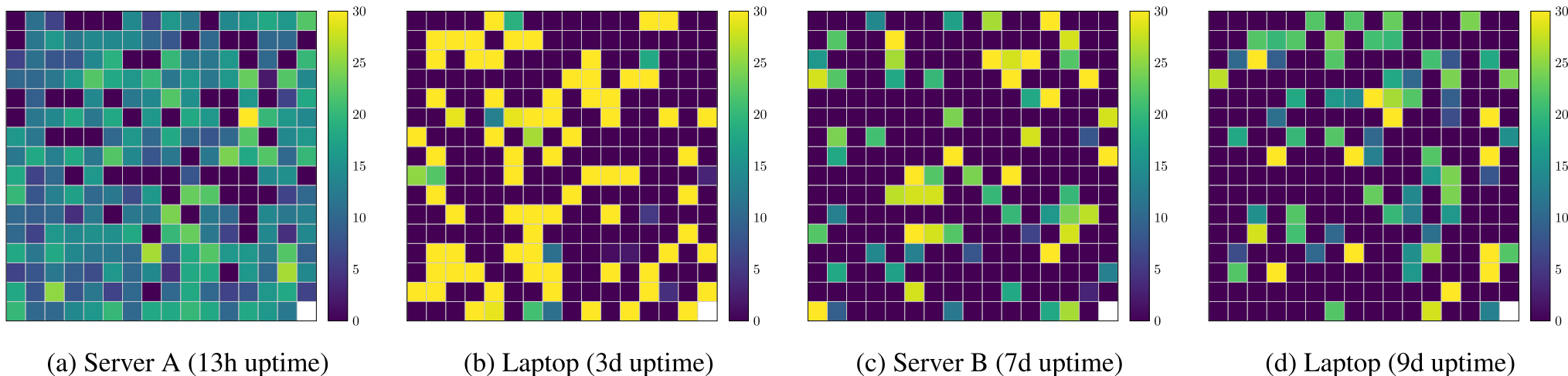
Bypass of Slab Hardening

↳ Heap Segregation (+ SLAB_VIRTUAL)

↳ Memory Tagging

Naive Spray-based Exploitation

High-uptime systems **concentrate reclaim activity** on fewer slots



Long-lived objects hoard KFENCE memory slots as they do not deallocate

↳ Allocation and deallocations concentrates on few slots

↳ More likely to **reclaim** target **UAF** object slots during **sprays**

Conclusion



Upstream Patches

Upstream implemented three patches

1. Disabling KFENCE on MTE

↳ MTE disables KFENCE at boot time

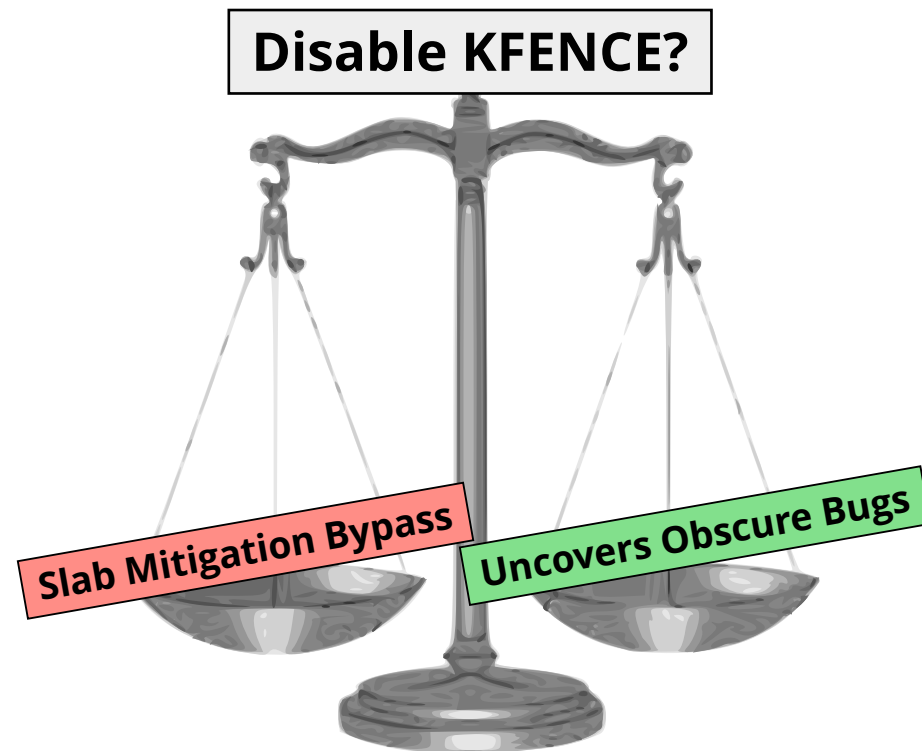
2. Dedicated crash-on-fault toggles

↳ Better KFENCE fault controls

3. Freelist Fisher-Yates Shuffle

↳ Further freelist randomization

Heap segregation remains unsolved



No clean solution to solve the heap segregation problem

$$(255 + 1) \times 2 \times 4096 = 2\text{MiB}$$

Scaling to one pool per Slab cache is unreasonable: 500x overhead

- ↳ Static smaller pools may waste memory and not scale
- ↳ Dynamic-sized pools may lead to cross-cache attacks in KFENCE

🔗 Maintainability

- With two allocators an attacker picks the **weakest link**
- Both must be **equally secure**
- What if there is new Slab hardening techniques in the future?



Download Article



- Download Article <
- Download Slides
- Download Infographic
- Website
- Email

Ernesto Martínez García

PhD Student @ Graz University of Technology

✉ ernesto.martinezgarcia@tugraz.at

🌐 ernestomartinezgarcia.com

🌐 linkedin.com/in/ecomaikgolf

Looking for cool Linux security research projects/ideas :)