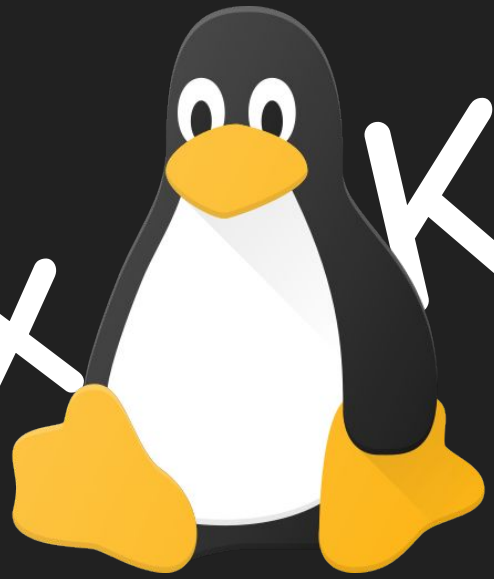


Harvest Season for SLUB !

From io_uring Vulnerability
to Novel Sheaf-Based Exploitation Techniques

OffensiveCon 26 Tokyo @ naup96321 & ddd-dong

Linux Kernel




mTHP
per-VMA
io_uring
DAMON
EEVDF
Linux Kernel
sheaf / barn
maple tree
RU
futex2
sched_ext

Attack Surface & Linux Kernel Exploitation Technique !



@ naup96321 (堇姬 Naup)

> Hao-Yu Yang (NYCU CS)

> CTF @ CakeisTheFake / Squid Proxy Lovers

> DEF CON CTF 34 @ Orange Fan Club

> Research Intern @ TRAPA Security

> Focus on Linux kernel Exploitation / hypervisor

Gmail: naup96721@gmail.com

twitter: @naup96721

Blog: https://naup.mygo.tw/





@ ddd-dong

> Yu-Ting Lin (NYCU CS)

> Two published NLP papers

> GCC 2026 Taiwan Delegate

> ASI3 2025 Best Project

> Focus on AI Safety / NLP / Binary Exploitation

Gmail: dong1214.mailbox@gmail.com

twitter: @Ddd_dong_

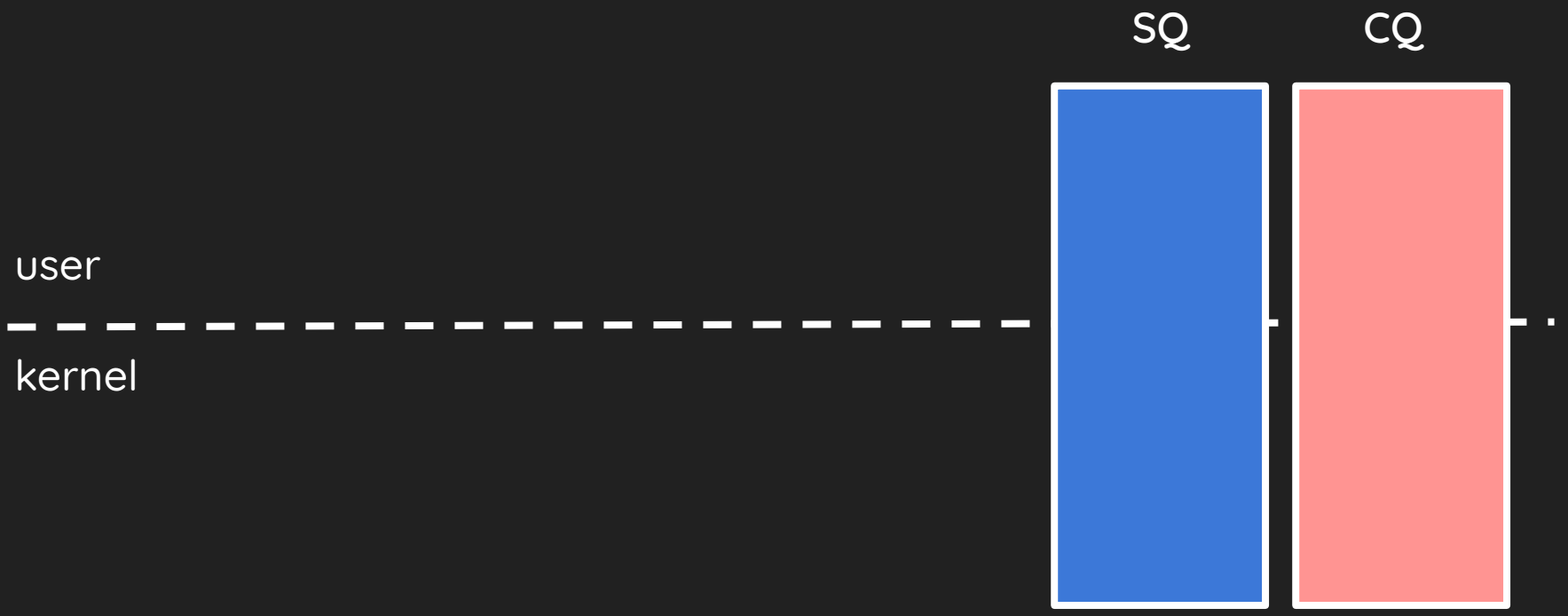
Blog: https://ddd-dong.github.io/blog/


io uring

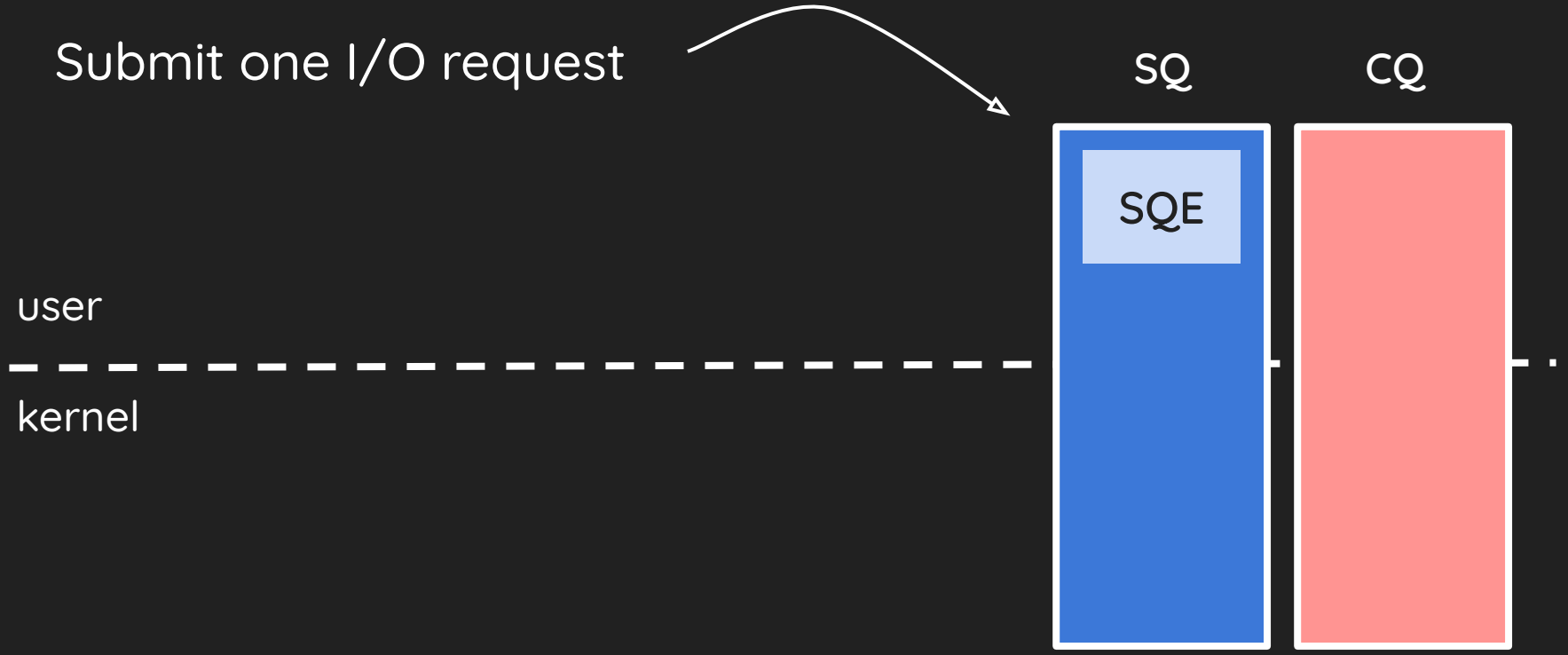
Overview io_uring

- Introduced in Linux 5.1 (2019)
- High Performance asynchronous I/O Interface
- Shared ring buffers (SQ / CQ) between user and kernel
- Supports a wide range of I/O operations
- Reduces system-call overhead

Submission Queue（SQ）and Completion Queue（CQ）

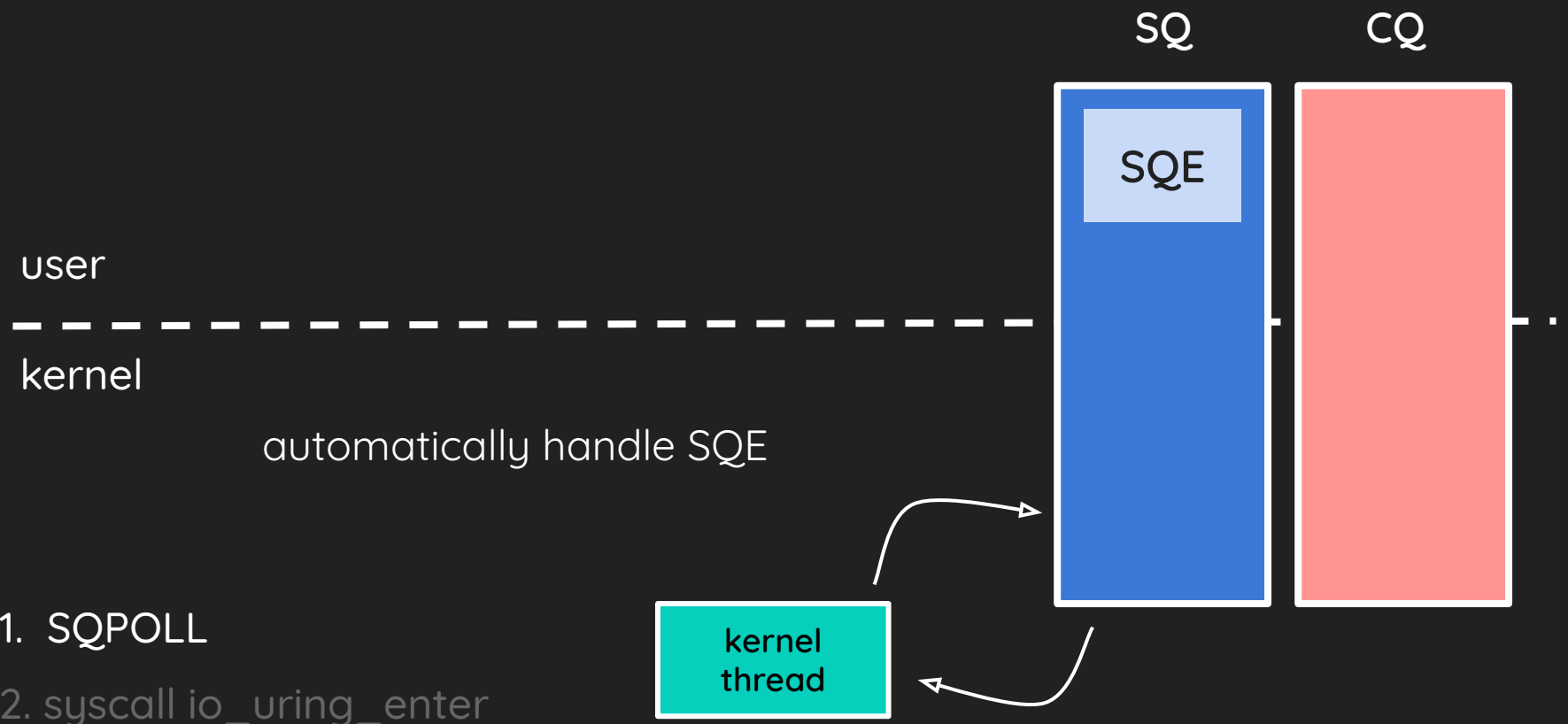


(two ring buffers shared between user and kernel)

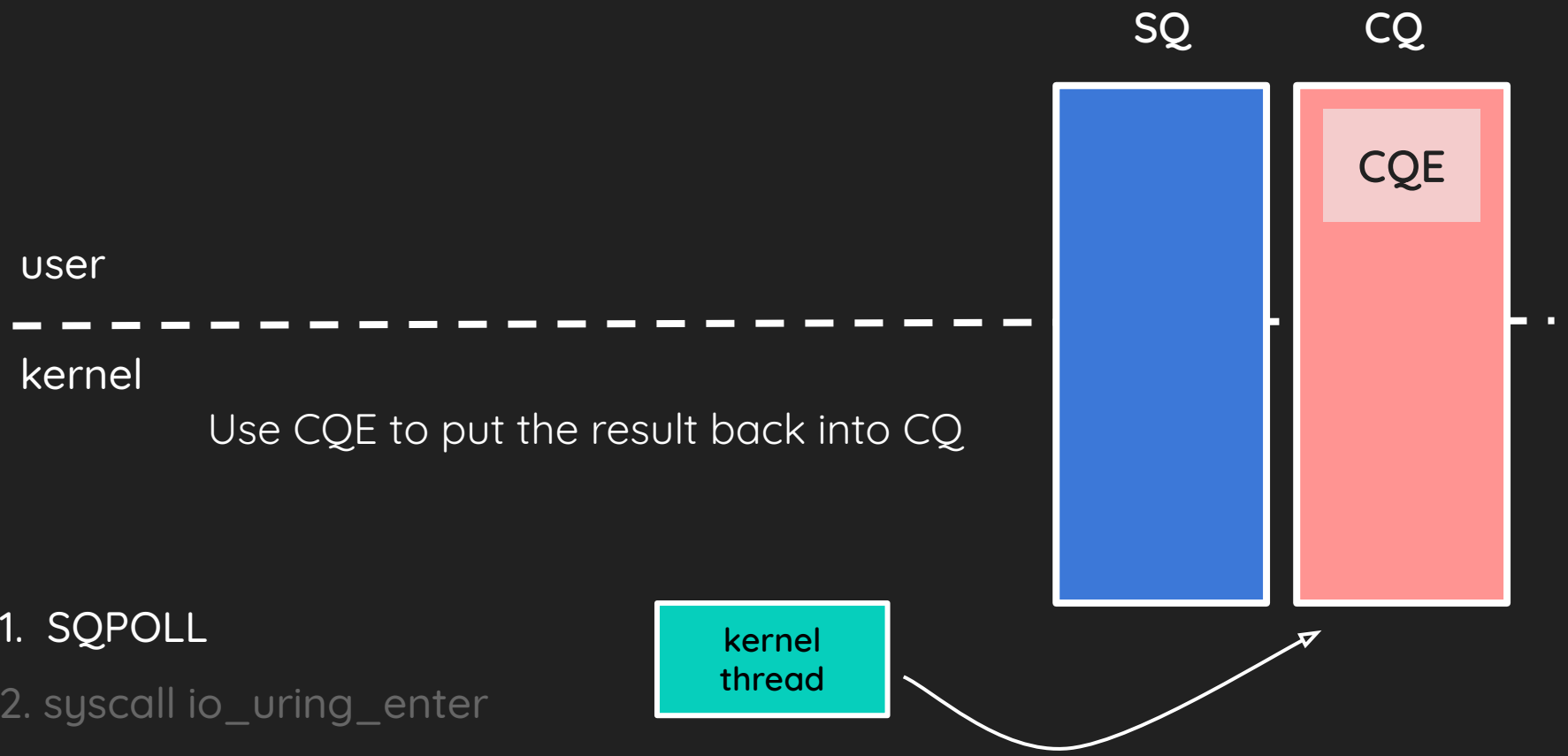

Submit one I/O request
SQ
CQ
SQE
user
kernel



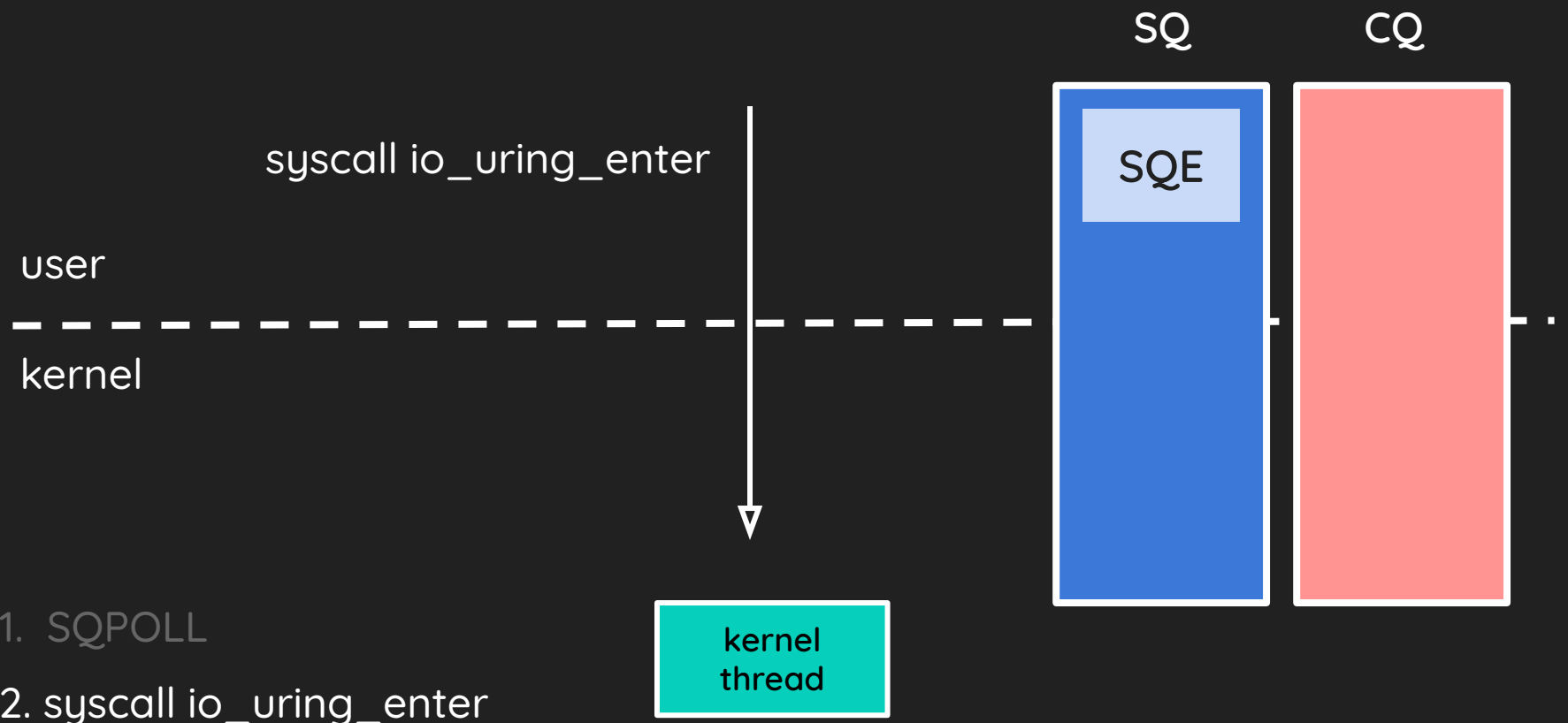
1. SQPOLL
2. syscall io_uring_enter



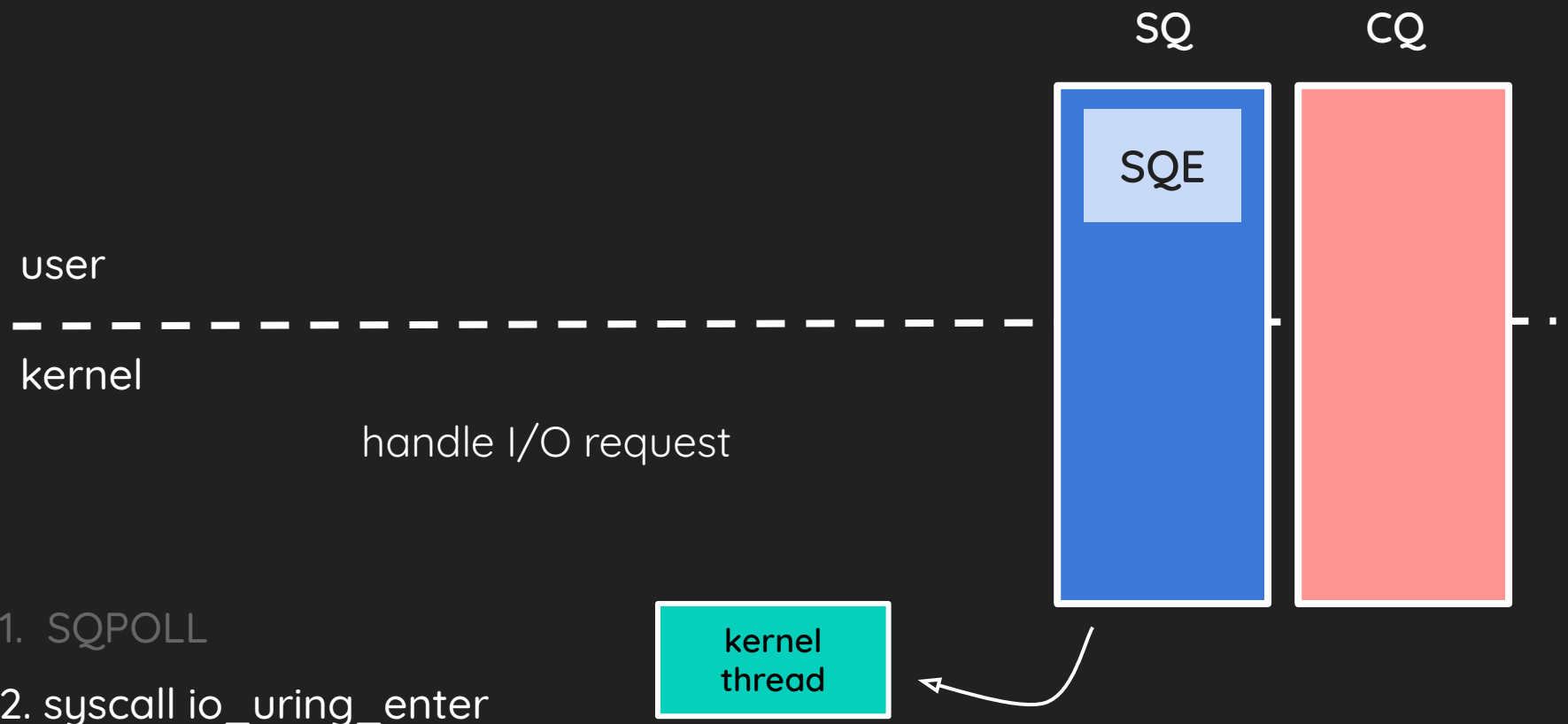
1. SQPOLL
2. syscall io_uring_enter



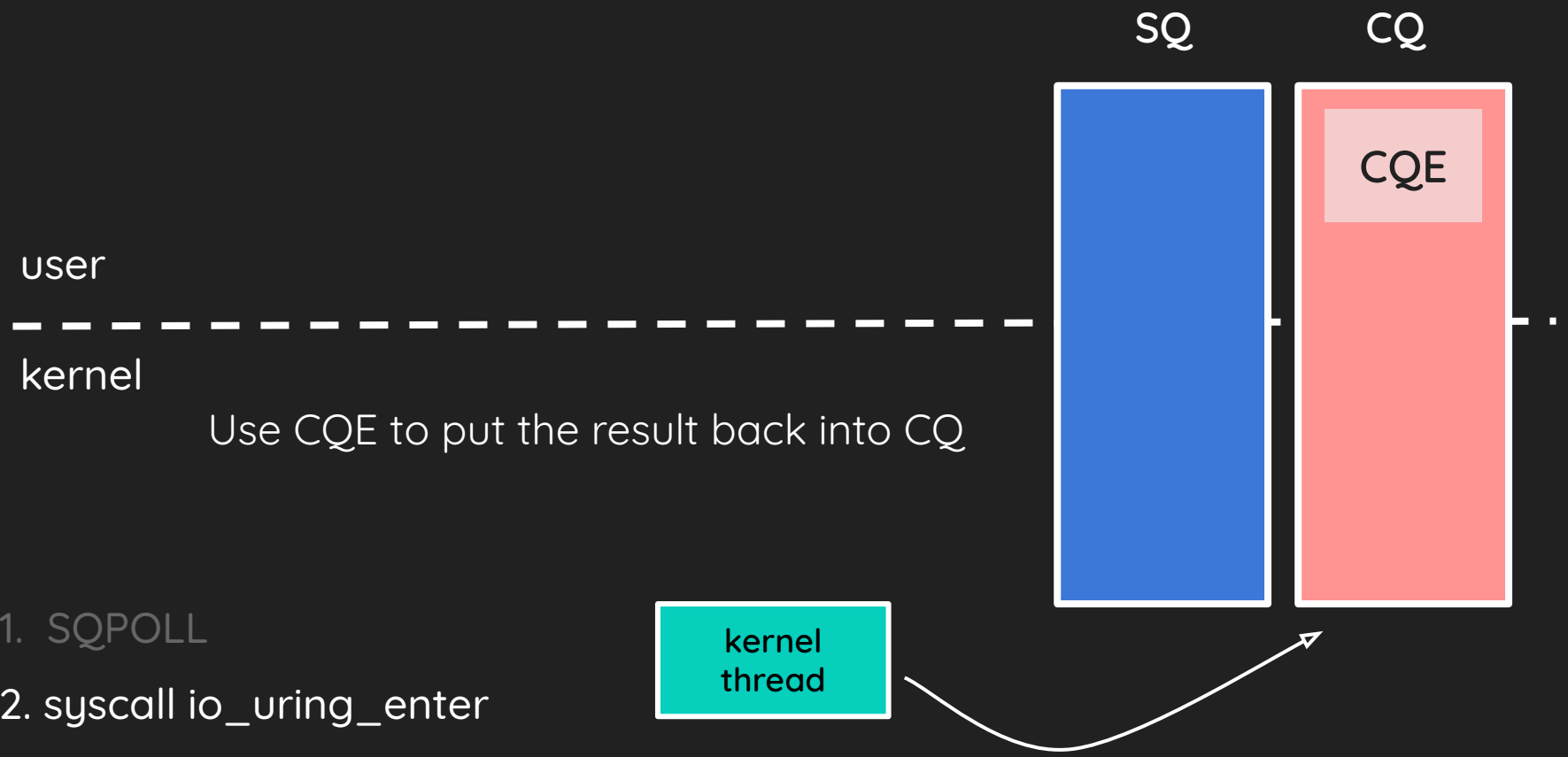
1. SQPOLL
2. syscall io_uring_enter



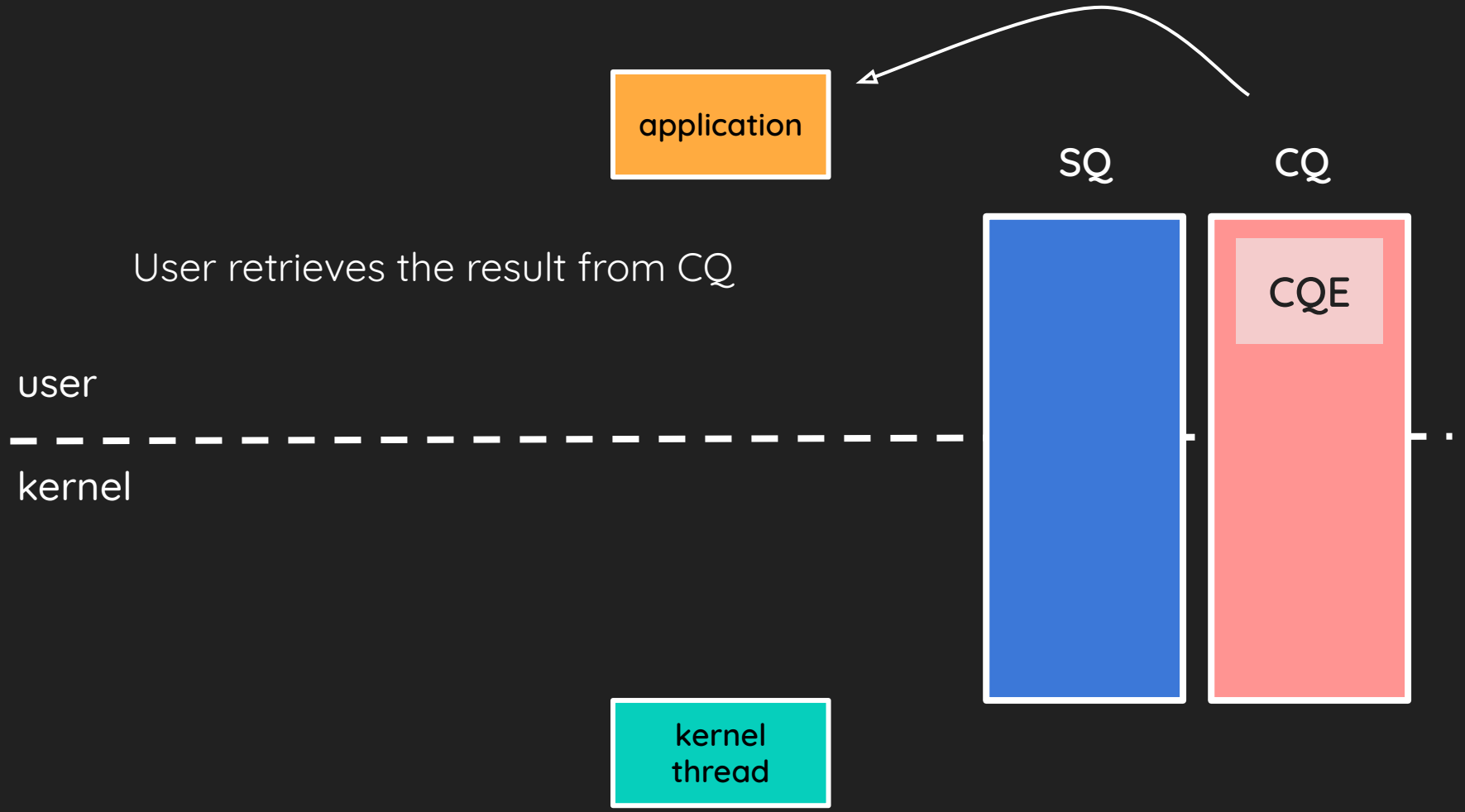
1. SQPOLL
2. syscall io_uring_enter



1. SQPOLL
2. syscall io_uring_enter



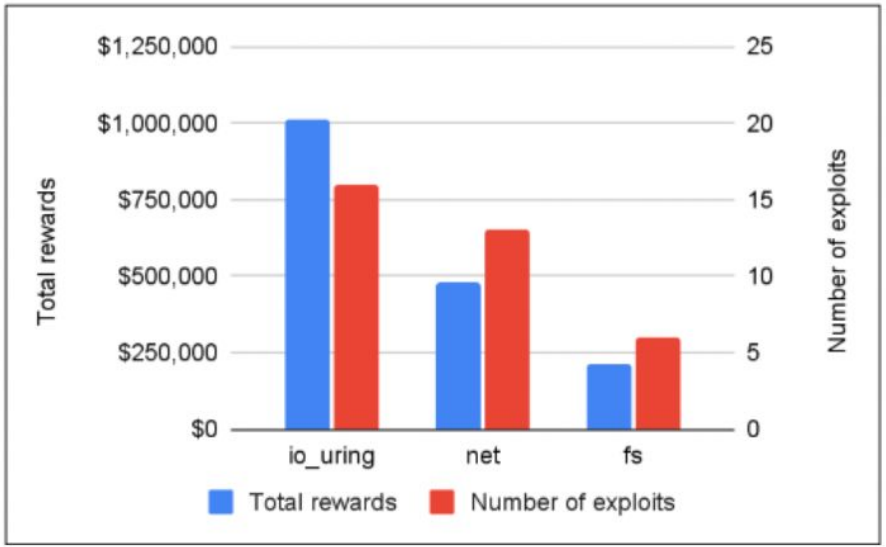
1. SQPOLL
2. syscall io_uring_enter


application
SQ
CQ
User retrieves the result from CQ
CQE
user
kernel
kernel thread

Why target io_uring ?

1. Broad attack surface (socket, file, poll, timeout, splice ...)
2. Many asynchronous operations, states and resource life cycle
3. High performance, high complexity (lockless, share memory ...)

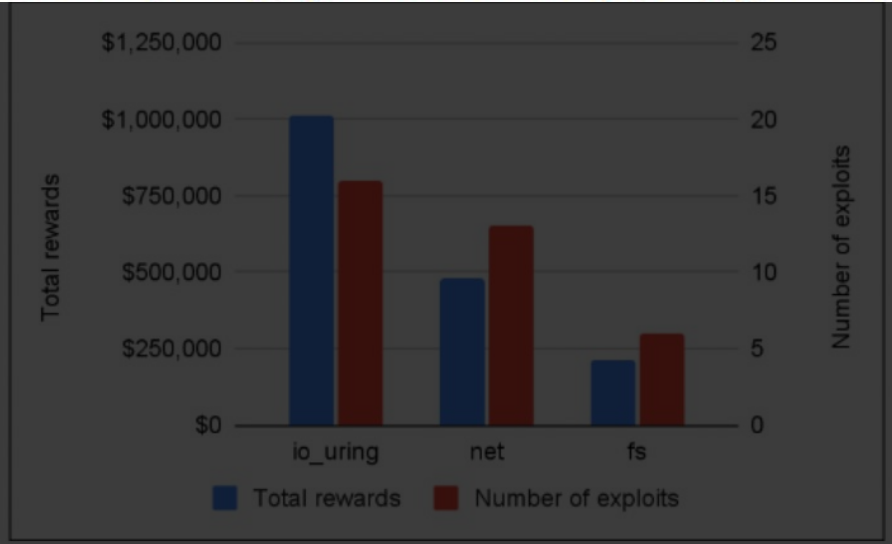
Since its inception, the program has rewarded researchers with a total of 1.8 million USD, and in the past year, there has been a clear trend: 60% of the submissions exploited the io_uring component of the Linux kernel (we paid out around 1 million USD for io_uring alone). Furthermore, io_uring vulnerabilities were used in all the submissions which bypassed our mitigations.



https://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html

Since its inception, the program has rewarded researchers with a total of 1.8 million USD, and in the past year, there has been a clear trend: 60% of the submissions exploited the io_uring component of the Linux kernel (we paid out around 1 million USD for io_uring

and in the past year, there has been a clear trend: 60% of the submissions exploited the io_uring component of the Linux kernel (we paid out around 1 million USD for io_uring





https://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html


I/O
URING

Vulnerability

IORING_OP_SEND iovec array grow failure logic bug to LPE

What is IORING_OP_SEND ?

IORING_OP_SEND is an io_uring operation used to send data through a socket

IORING_OP_SEND

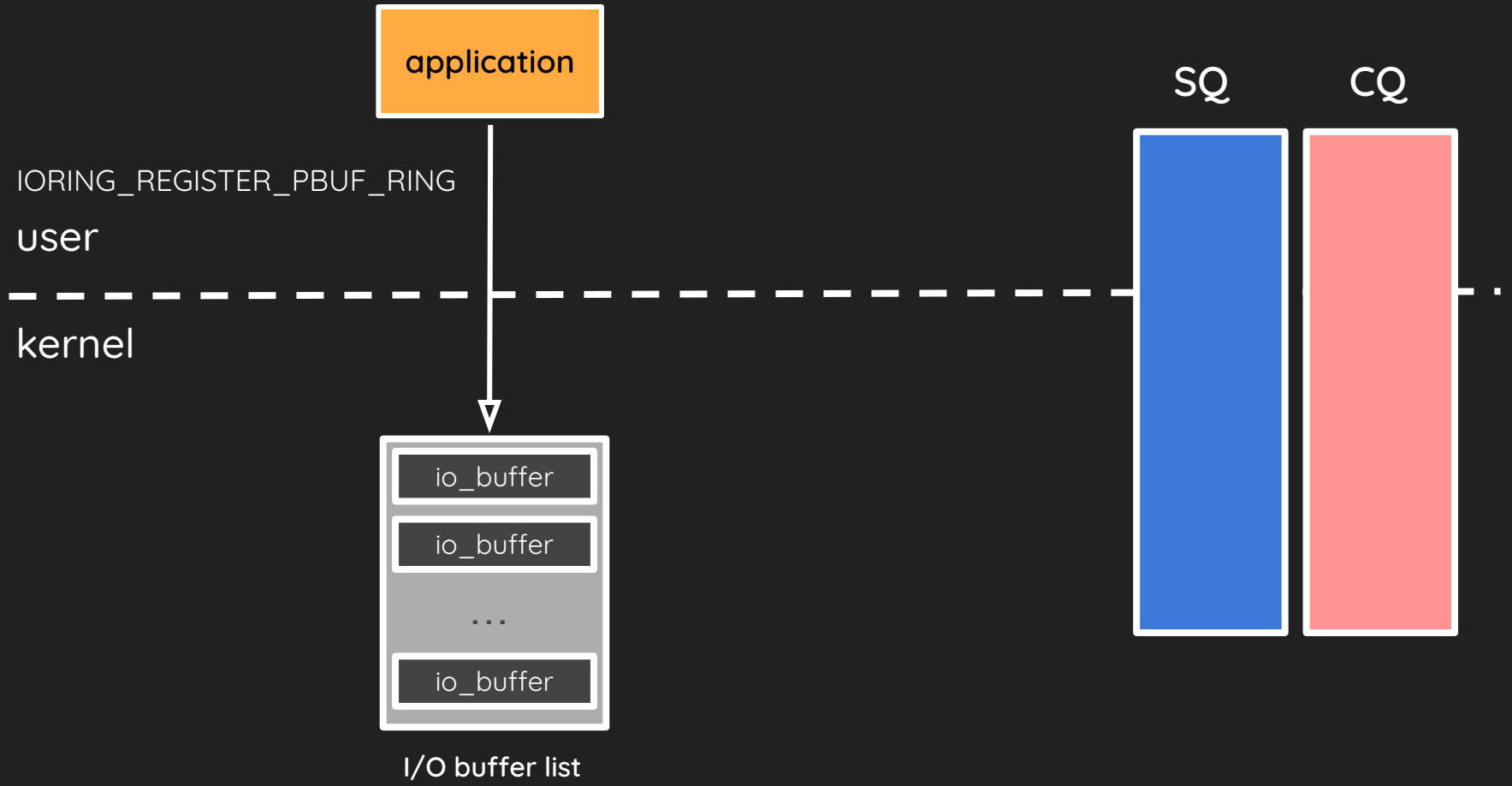
Issue the equivalent of a send(2) system call. *fd* must be set to the socket file descriptor, *addr* must contain a pointer to the buffer, *len* denotes the length of the buffer to send, and *msg_flags* holds the flags associated with the system call. See also send(2) for the general description of the related system call. Available since 5.6.

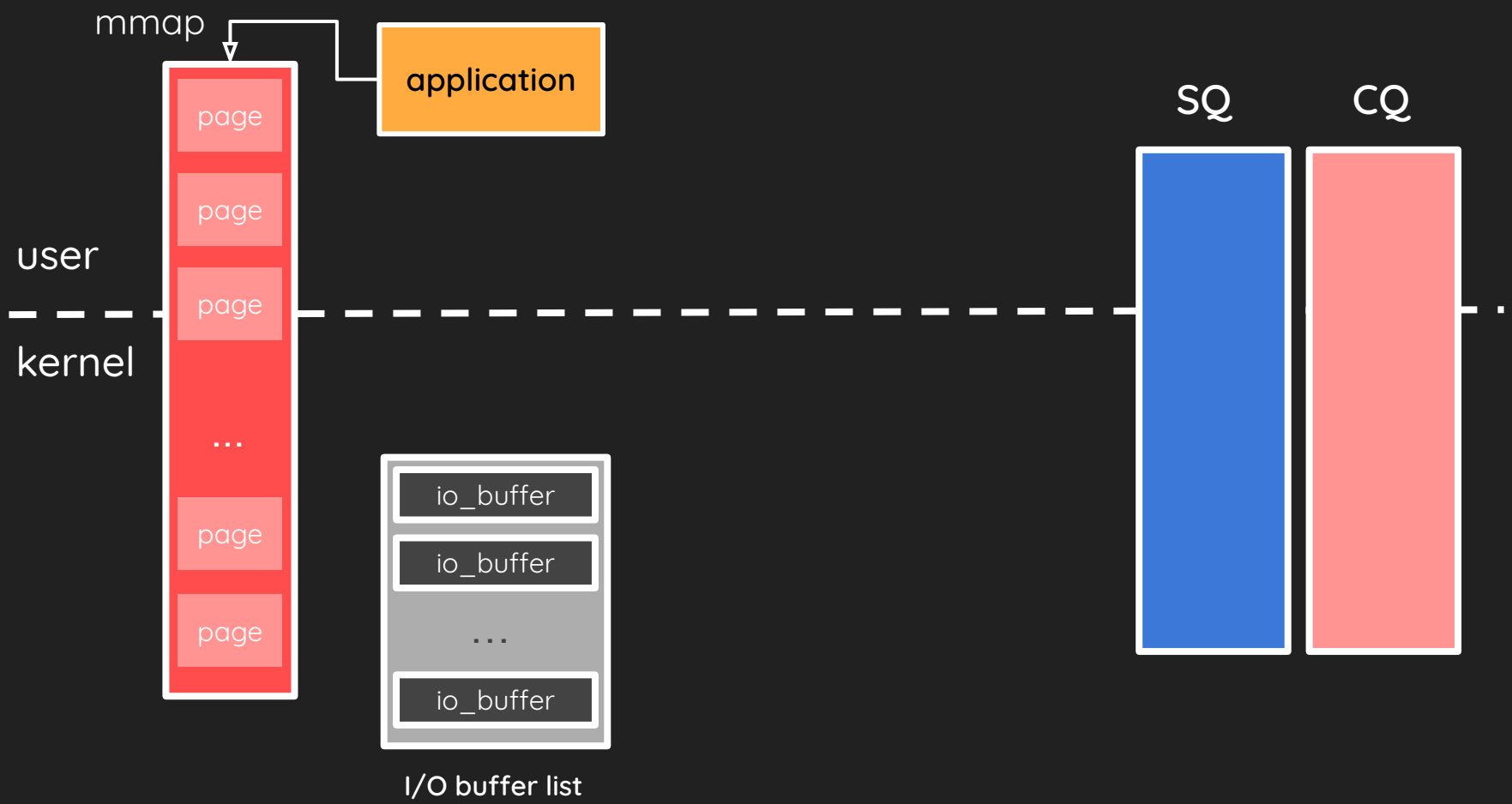
This command also supports the following modifiers in *ioprio:*

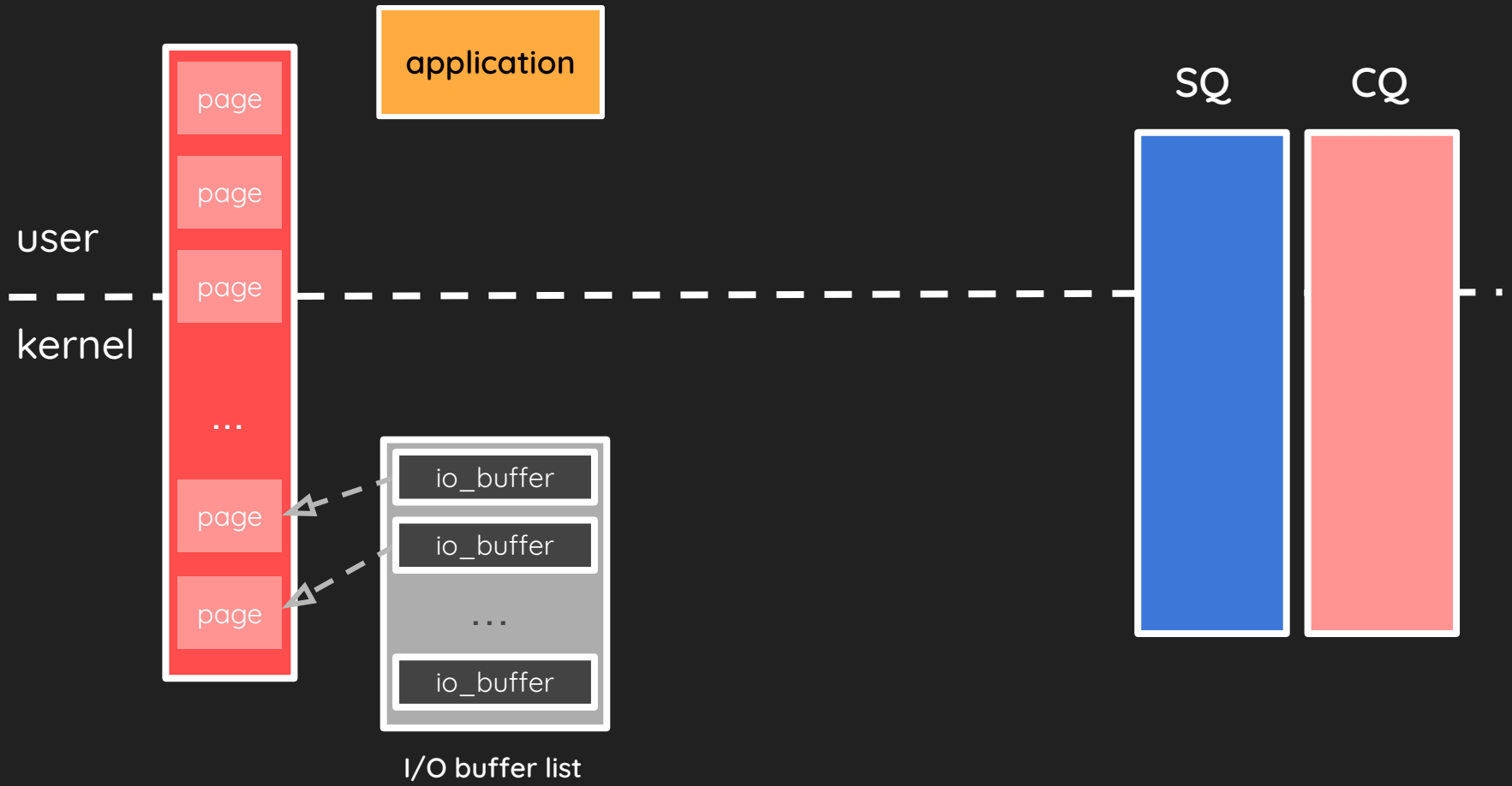
IORING_RECVSEND_POLL_FIRST If set, io_uring will assume the socket is currently full and attempting to send data will be unsuccessful. For this case, io_uring will arm internal poll and trigger a send of the data when there is enough space available. This initial send attempt can be wasteful for the case where the socket is expected to be full, setting this flag will bypass the initial send attempt and go straight to arming poll. If poll does indicate that data can be sent, the operation will proceed.

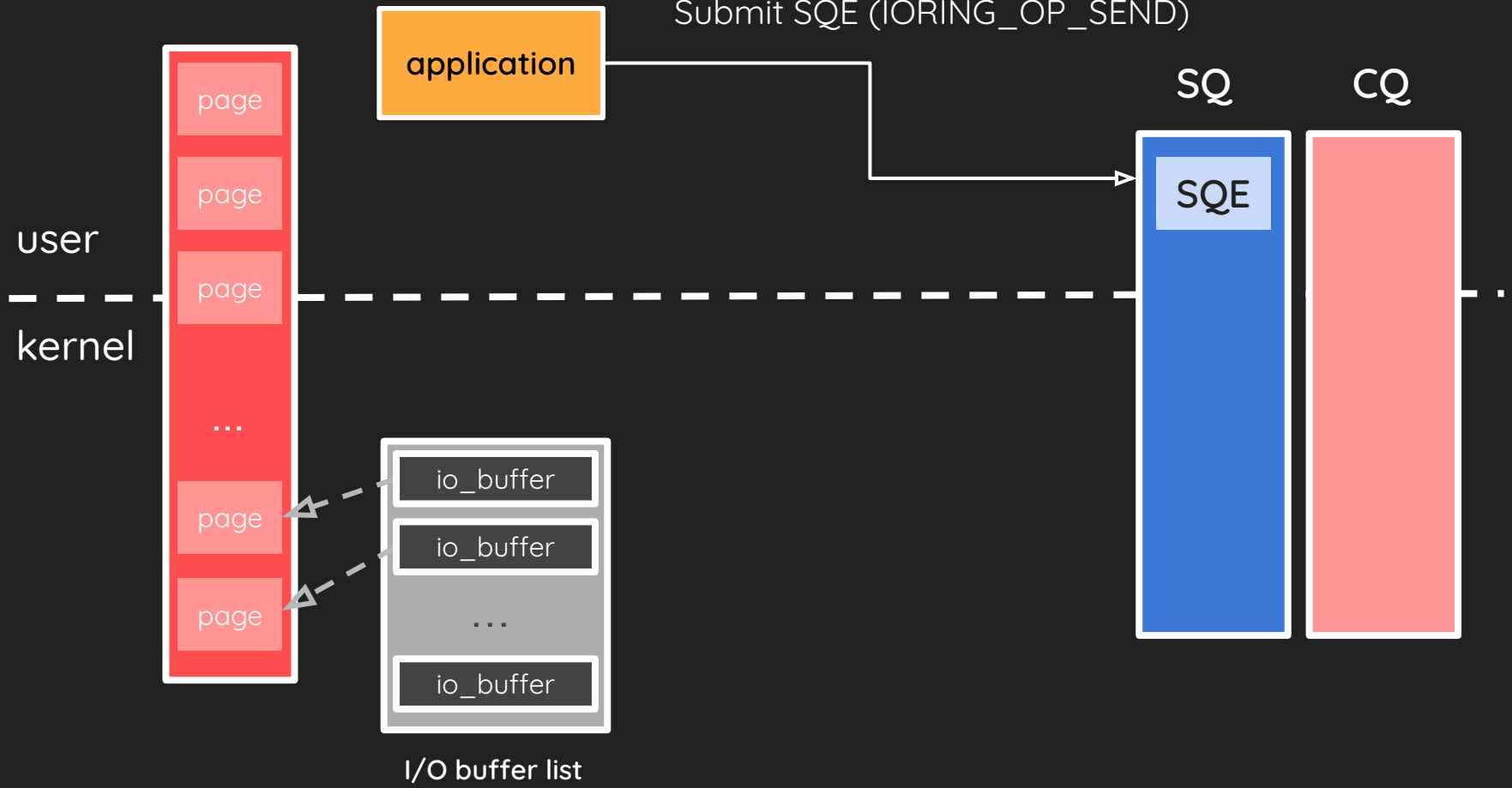


https://man7.org/linux/man-pages/man2/io_uring_enter.2.html


application
SQ
CQ
IORING_REGISTER_PBUF_RING
user
kernel
io_buffer
io_buffer
...
io_buffer
I/O buffer list


mmap
application
page
page
page
...
page
page
user
kernel
io_buffer
io_buffer
...
io_buffer
I/O buffer list
SQ
CQ


application
SQ
CQ
user
kernel
page
page
page
...
page
page
io_buffer
io_buffer
...
io_buffer
I/O buffer list


Submit SQE (IORING_OP_SEND)
application
SQ
CQ
SQE
user
kernel
page
page
page
...
page
page
io_buffer
io_buffer
...
io_buffer
I/O buffer list



/io_uring/net.c#L620 (v7.2-rc1)

```
struct buf_sel_arg arg =
...

if (kmsg->vec.iovec) {
    arg.nr_iovs = kmsg->vec.nr;
    arg.iovs = kmsg->vec.iovec;
    arg.mode = KBUF_MODE_FREE;
}

ret = io_buffers_select(req, &arg, sel, issue_flags);
if (unlikely(ret < 0))
    return ret;

if (arg.iovs != &kmsg->fast_iov && arg.iovs != kmsg->vec.iovec) {
    kmsg->vec.nr = ret;
    kmsg->vec.iovec = arg.iovs;
    req->flags |= REQ_F_NEED_CLEANUP;
}
```



/io_uring/net.c#L620 (v7.2-rc1)

```
struct buf_sel_arg arg =

if (kmsg->vec.iovec) {
    arg.nr_iovs = kmsg->vec.nr;
    arg.iovs = kmsg->vec.iovec;
    arg.mode = KBUF_MODE_FREE;
}

ret = io_buffers_select(req, &arg, sel, issue_flags);

if (arg.iovs != &kmsg->fast_iov && arg.iovs != kmsg->vec.iovec) {
    kmsg->vec.nr = ret;
    kmsg->vec.iovec = arg.iovs;
    req->flags |= REQ_F_NEED_CLEANUP;
}
```



/io_uring/kbuf.c#L352 (v7.2-rc1)

```
int io_buffers_select(struct io_kiocb *req, struct buf_sel_arg *arg,
                      struct io_br_sel *sel, unsigned int issue_flags)
{
    ...

    sel->buf_list = io_buffer_get_list(ctx, arg->buf_group);
    ...

    if (sel->buf_list->flags & IOBL_BUF_RING) {
        ret = io_ring_buffers_peek(req, arg, sel->buf_list);

    ...
    return ret;
```



/io_uring/kbuf.c#L352 (v7.2-rc1)

```
int io_buffers_select(struct io_kiocb *req, struct buf_sel_arg *arg,
                      struct io_br_sel *sel, unsigned int issue_flags)
{
    sel->buf_list = io_buffer_get_list(ctx, arg->buf_group);

    if (sel->buf_list->flags & IOBL_BUF_RING) {
        ret = io_ring_buffers_peek(req, arg, sel->buf_list);
    ...
    return ret;
```



/io_uring/kbuf.c#L269 (v7.2-rc1)

```
static int io_ring_buffers_peek(struct io_kiocb *req, struct buf_sel_arg *arg,
                struct io_buffer_list *bl)
{
    ...
    buf = io_ring_head_to_buf(br, head, bl->mask);
    if (arg->max_len) {
        u32 len = READ_ONCE(buf->len);
        size_t needed;

        if (unlikely(!len))
            return -ENOBUFS;
        needed = (arg->max_len + len - 1) / len;
        needed = min_not_zero(needed, (size_t) PEEK_MAX_IMPORT);
        if (nr_avail > needed)
            nr_avail = needed;
    }
    ...
        iov->iov_base = u64_to_user_ptr(READ_ONCE(buf->addr));
        iov->iov_len = len;
```



/io_uring/kbuf.c#L269 (v7.2-rc1)

```
static int io_ring_buffers_peek(struct io_kiocb *req, struct buf_sel_arg *arg,
                struct io_buffer_list *bl)
{
    ...
    buf = io_ring_head_to_buf(br, head, bl->mask);
    if (arg->max_len) {
        u32 len = READ_ONCE(buf->len);
        size_t needed;

        if (unlikely(!len))
            return -ENOBUFS;
        needed = (arg->max_len + len - 1) / len;
        needed = min_not_zero(needed, (size_t) PEEK_MAX_IMPORT);
        if (nr_avail > needed)
            nr_avail = needed;
```

```
iov->iov_base = u64_to_user_ptr(READ_ONCE(buf->addr));
iov->iov_len = len;
```



/io_uring/kbuf.c#L287 (v7.2-rc1)

```
static int io_ring_buffers_peek(struct io_kiocb *req, struct buf_sel_arg *arg,
                struct io_buffer_list *bl)
{
    ...
    if (arg->mode & KBUF_MODE_EXPAND && nr_avail > nr_iovs && arg->max_len) {
        iov = kmalloc_objs(struct iovec, nr_avail);
        if (arg->mode & KBUF_MODE_FREE)
            kfree(arg->iovs);
        arg->iovs = iov;
        nr_iovs = nr_avail;
    }
    ...
    do {
        if (unlikely(!access_ok(iov->iov_base, len))) {
            if (arg->iovs != org_iovs)
                kfree(arg->iovs);
            return -EFAULT;
        }
```



/io_uring/kbuf.c#L287 (v7.2-rc1)

```
static int io_ring_buffers_peek(struct io_kiocb *req, struct buf_sel_arg *arg,
                struct io_buffer_list *bl)
{
    ...
    if (arg->mode & KBUF_MODE_EXPAND && nr_avail > nr_iovs && arg->max_len) {
        iov = kmalloc_objs(struct iovec, nr_avail);
        arg->iovs = iov;
        nr_iovs = nr_avail;
    }
    ...
    do {
        if (unlikely(!access_ok(iov->iov_base, len))) {
            if (arg->iovs != org_iovs)
                kfree(arg->iovs);
            return -EFAULT;
        }
```



/io_uring/kbuf.c#L287 (v7.2-rc1)

```
static int io_ring_buffers_peek(struct io_kiocb *req, struct buf_sel_arg *arg,
                struct io_buffer_list *bl)
{
    ...
    if (arg->mode & KBUF_MODE_EXPAND && nr_avail > nr_iovs && arg->max_len) {
        iov = kmalloc_objs(struct iovec, nr_avail);
        if (arg->mode & KBUF_MODE_FREE)
            kfree(arg->iovs);
        arg->iovs = iov;
```

```
if (arg->mode & KBUF_MODE_FREE)
    kfree(arg->iovs);
```

```
        if (unlikely(!access_ok(iov->iov_base, len))) {
            if (arg->iovs != org_iovs)
                kfree(arg->iovs);
            return -EFAULT;
        }
```



/io_uring/kbuf.c#L287 (v7.2-rc1)

```
static int io_ring_buffers_peek(struct io_kiocb *req, struct buf_sel_arg *arg,
                struct io_buffer_list *bl)
{
    ...
    if (arg->mode & KBUF_MODE_EXPAND && nr_avail > nr_iovs && arg->max_len) {
        iov = kmalloc_objs(struct iovec, nr_avail);
        if (arg->mode & KBUF_MODE_FREE)
```

```
arg->iovs = iov;
```

```
    }
    ...
    do {
        if (unlikely(!access_ok(iov->iov_base, len))) {
            if (arg->iovs != org_iovs)
                kfree(arg->iovs);
            return -EFAULT;
        }
```



/io_uring/net.c#L620 (v7.2-rc1)

```
struct buf_sel_arg arg =
...

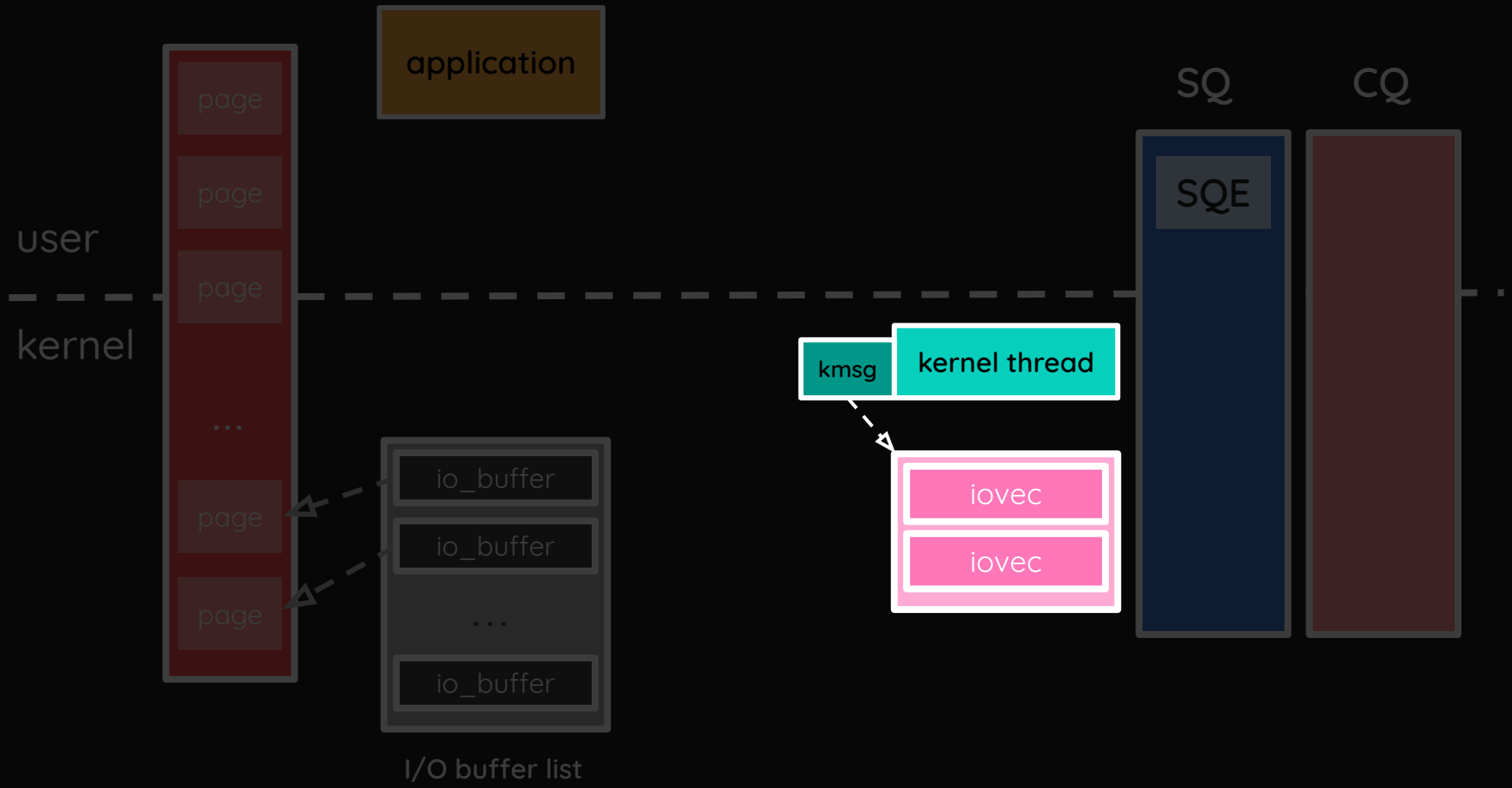
if (kmsg->vec.iovec) {
    arg.nr_iovs = kmsg->vec.nr;
    arg.iovs = kmsg->vec.iovec;
    arg.mode = KBUF_MODE_FREE;
}

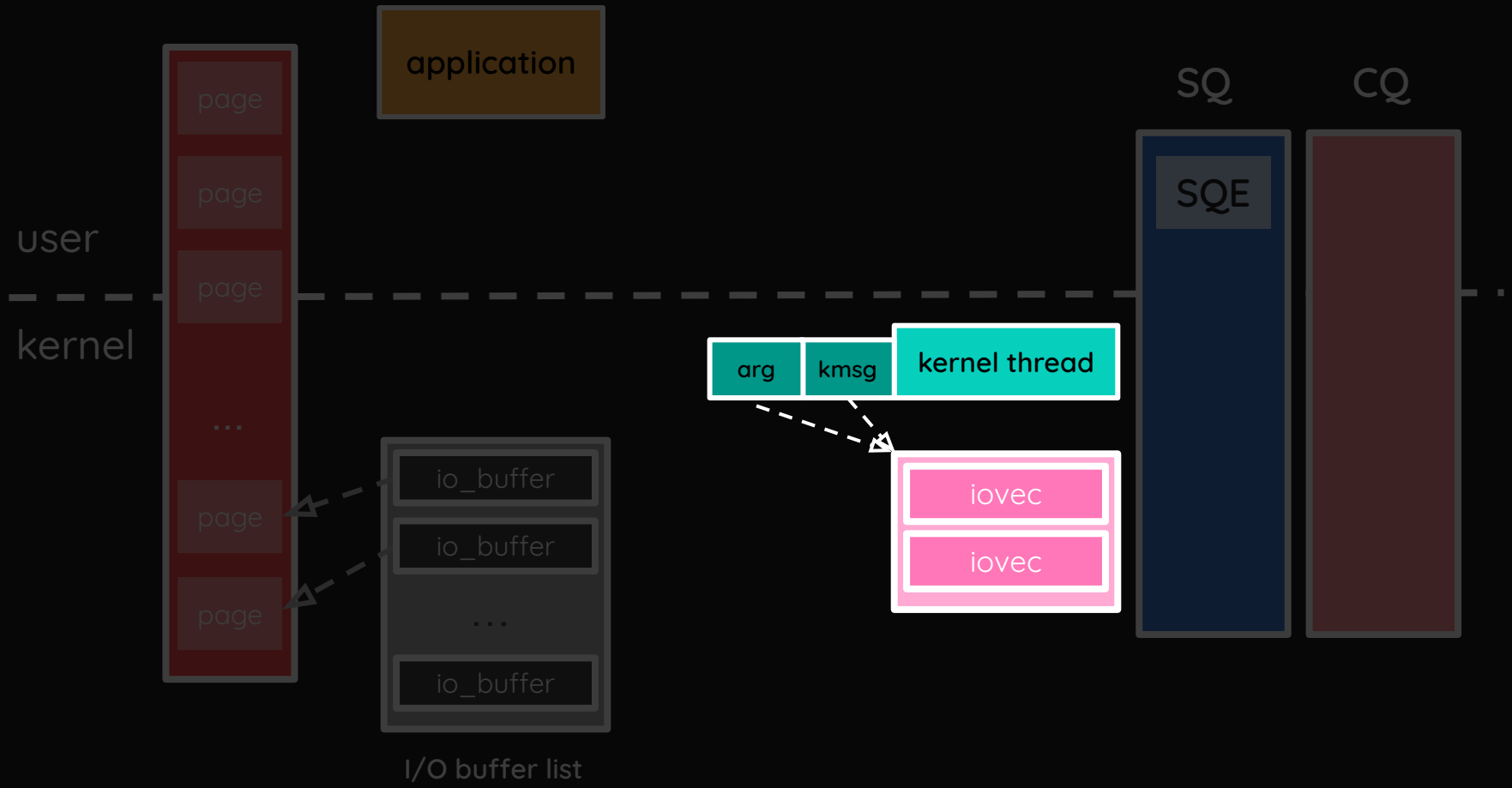
ret = io_buffers_select(req, &arg, sel, issue_flags);
if (unlikely(ret < 0))
    return ret;

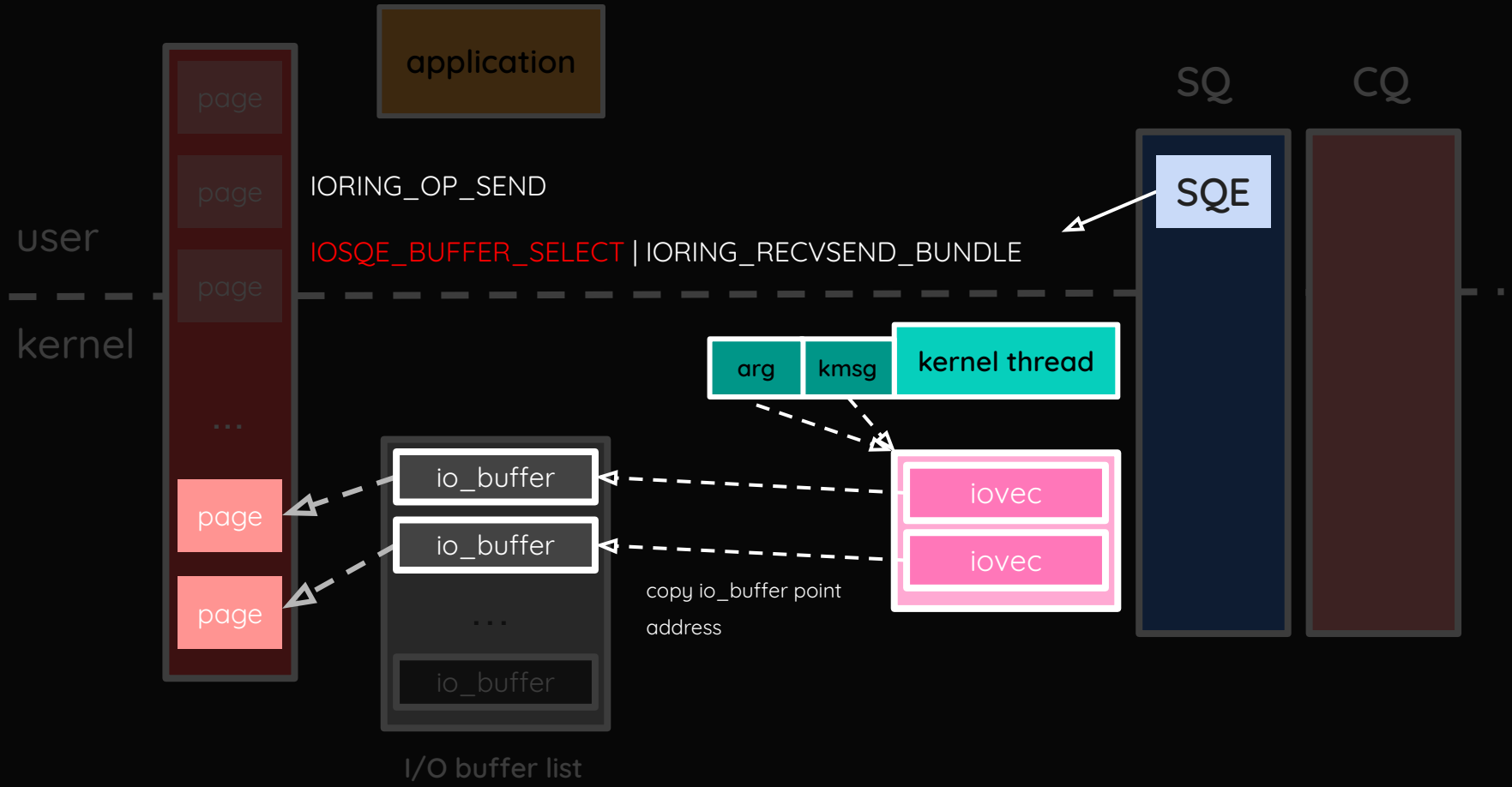
if (arg.iovs != &kmsg->fast_iov && arg.iovs != kmsg->vec.iovec) {
```

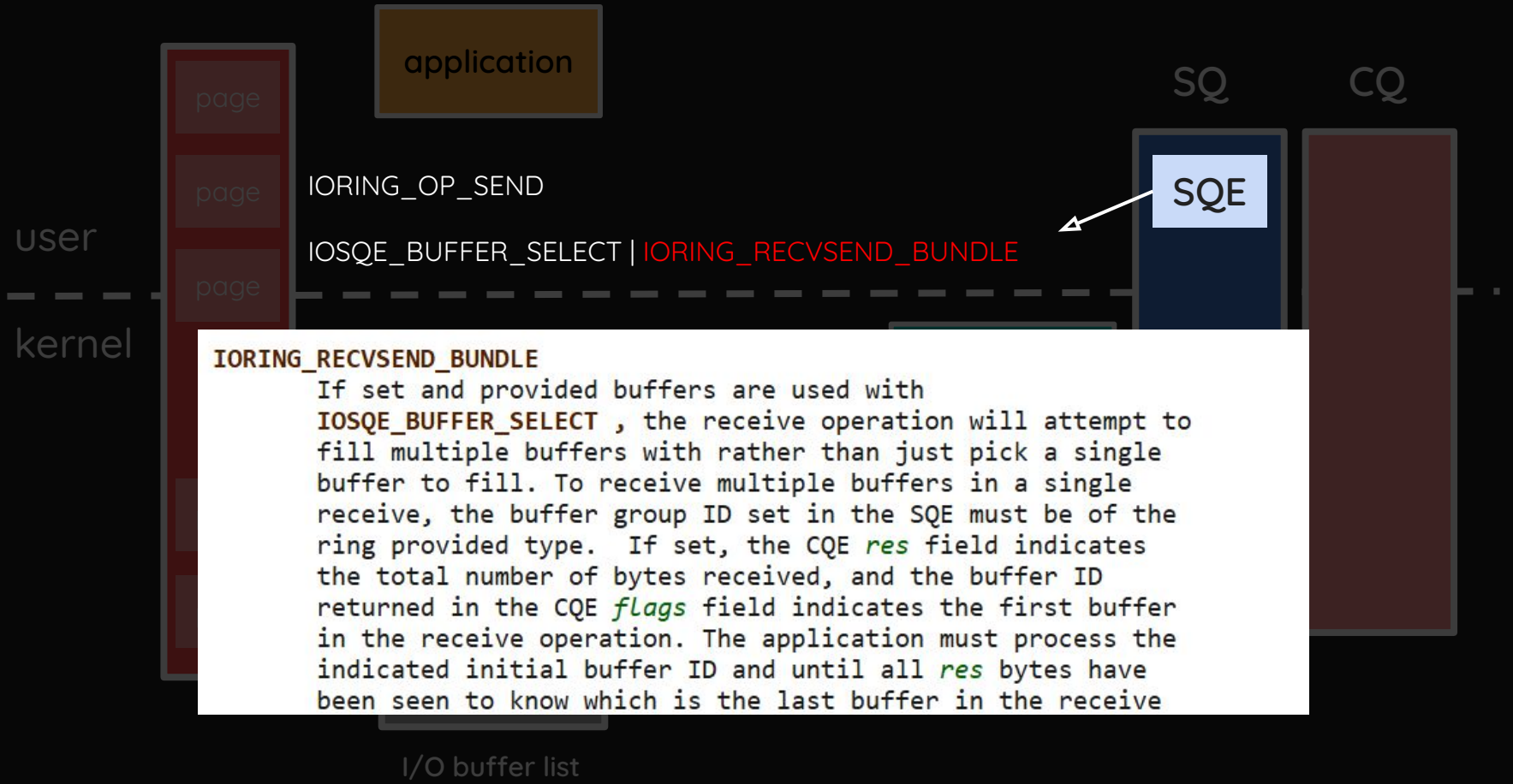


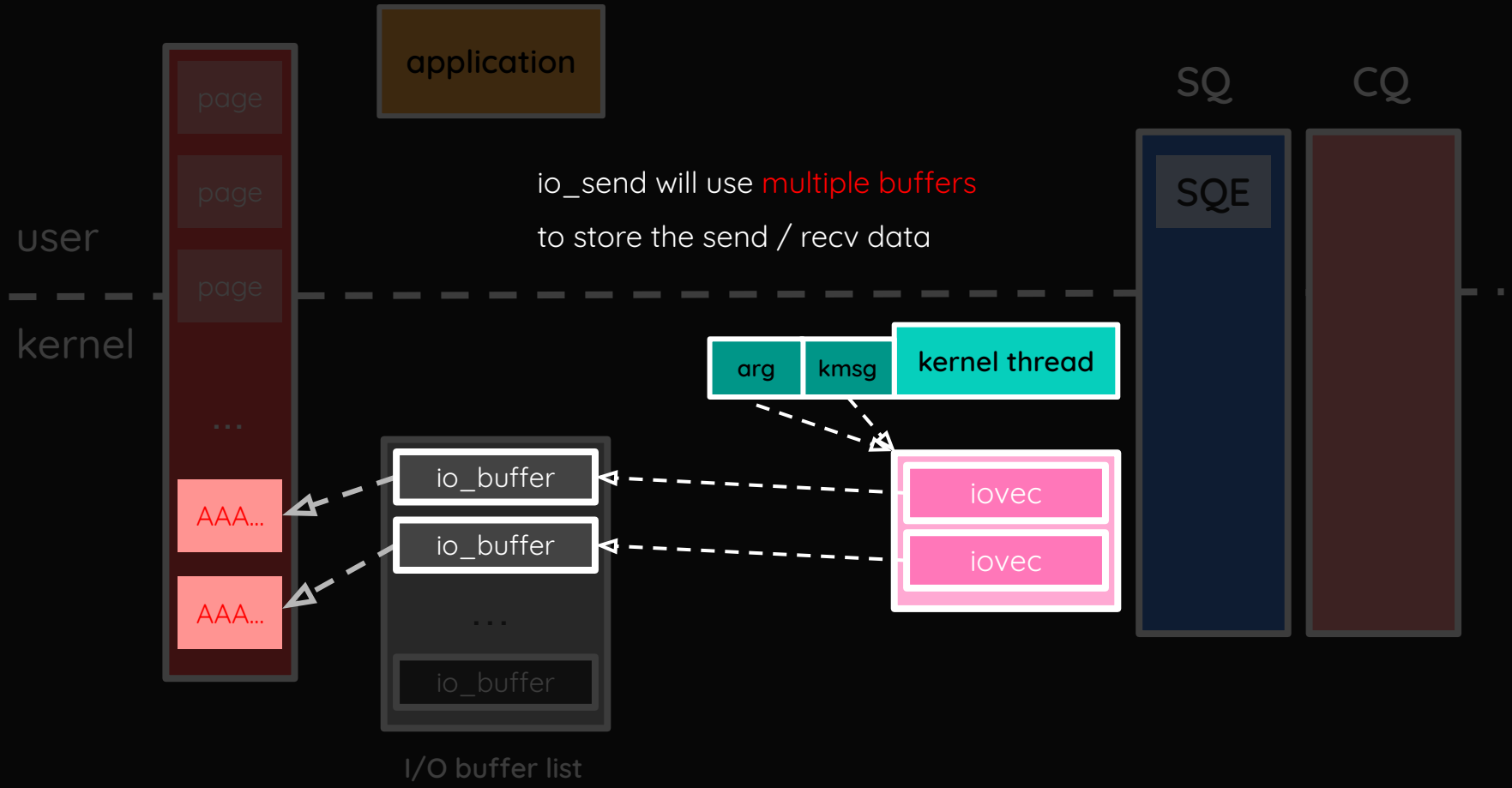
`kmsg->vec.iovec = arg.iovs;`

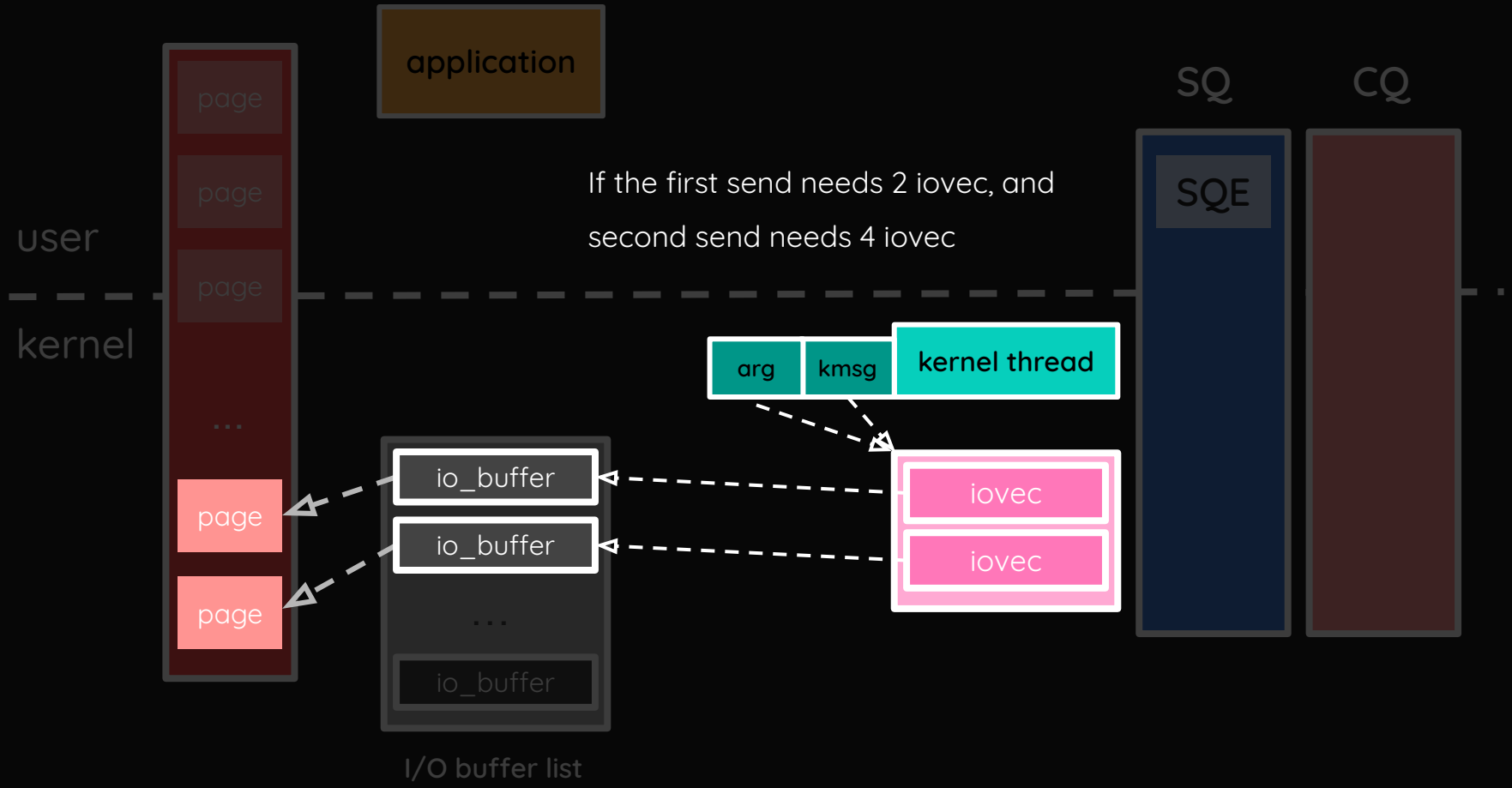

application
SQ
CQ
SQE
user
kernel
page
page
page
...
page
page
io_buffer
io_buffer
...
io_buffer
I/O buffer list
kmsg
kernel thread
iovec
iovec

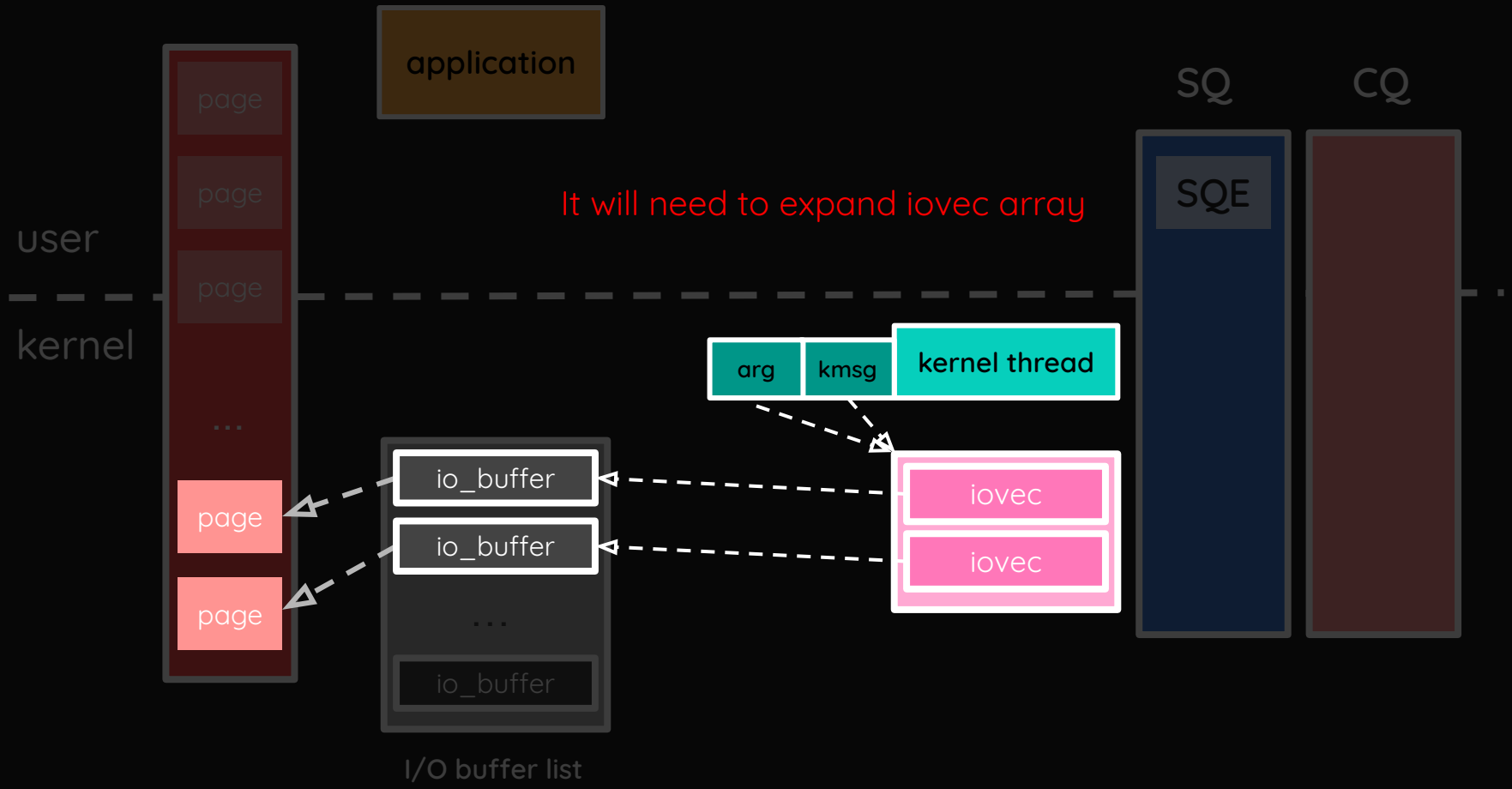

application
SQ
CQ
SQE
user
kernel
page
page
page
...
page
page
io_buffer
io_buffer
...
io_buffer
I/O buffer list
arg
kmsg
kernel thread
iovec
iovec

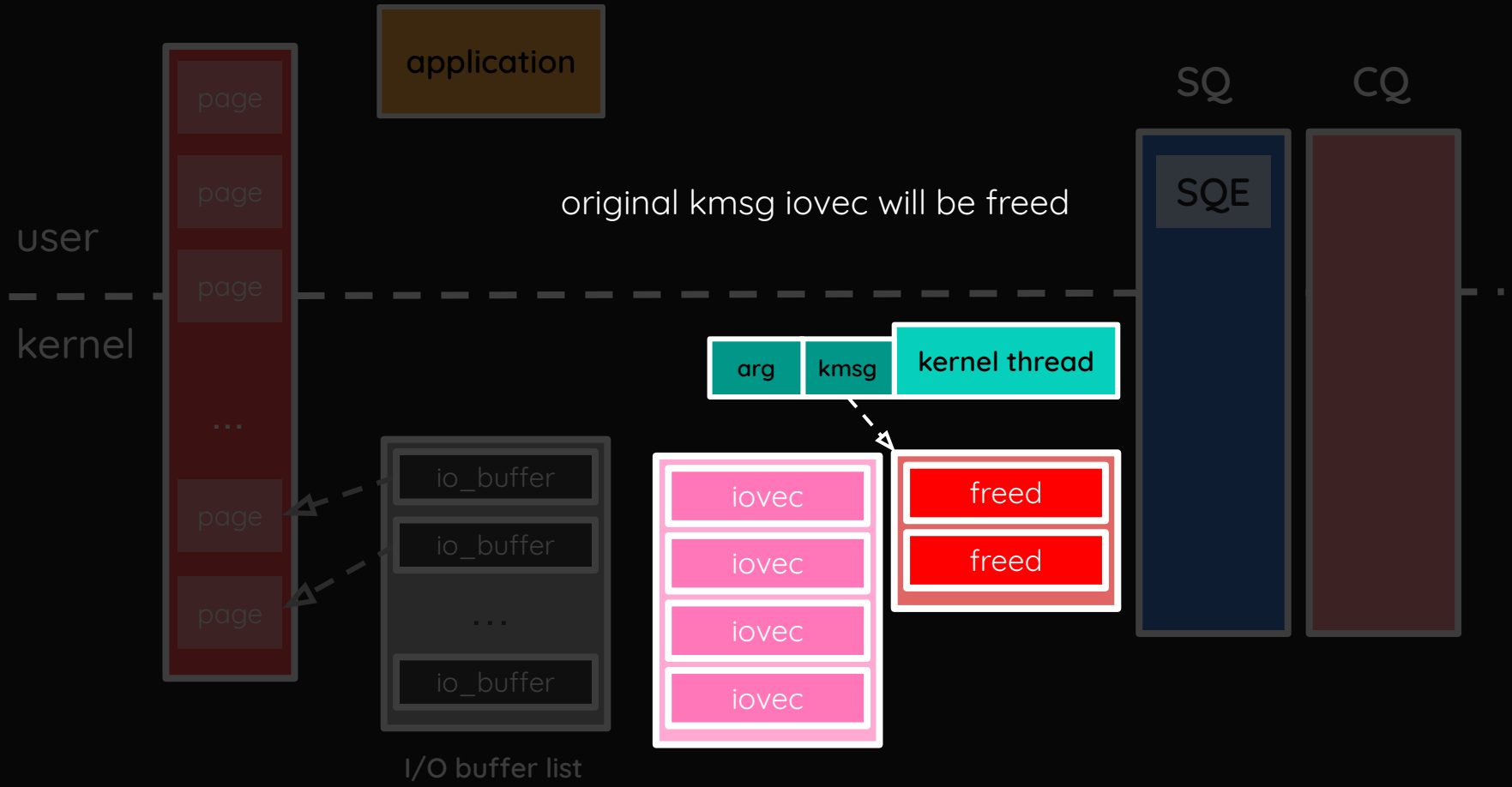

application
SQ
CQ
SQE
page
page
page
...
page
page
user
kernel
IORING_OP_SEND
IOSQE_BUFFER_SELECT | IORING_RECVSEND_BUNDLE
arg
kmsg
kernel thread
io_buffer
io_buffer
...
io_buffer
iovec
iovec
copy io_buffer point
address
I/O buffer list

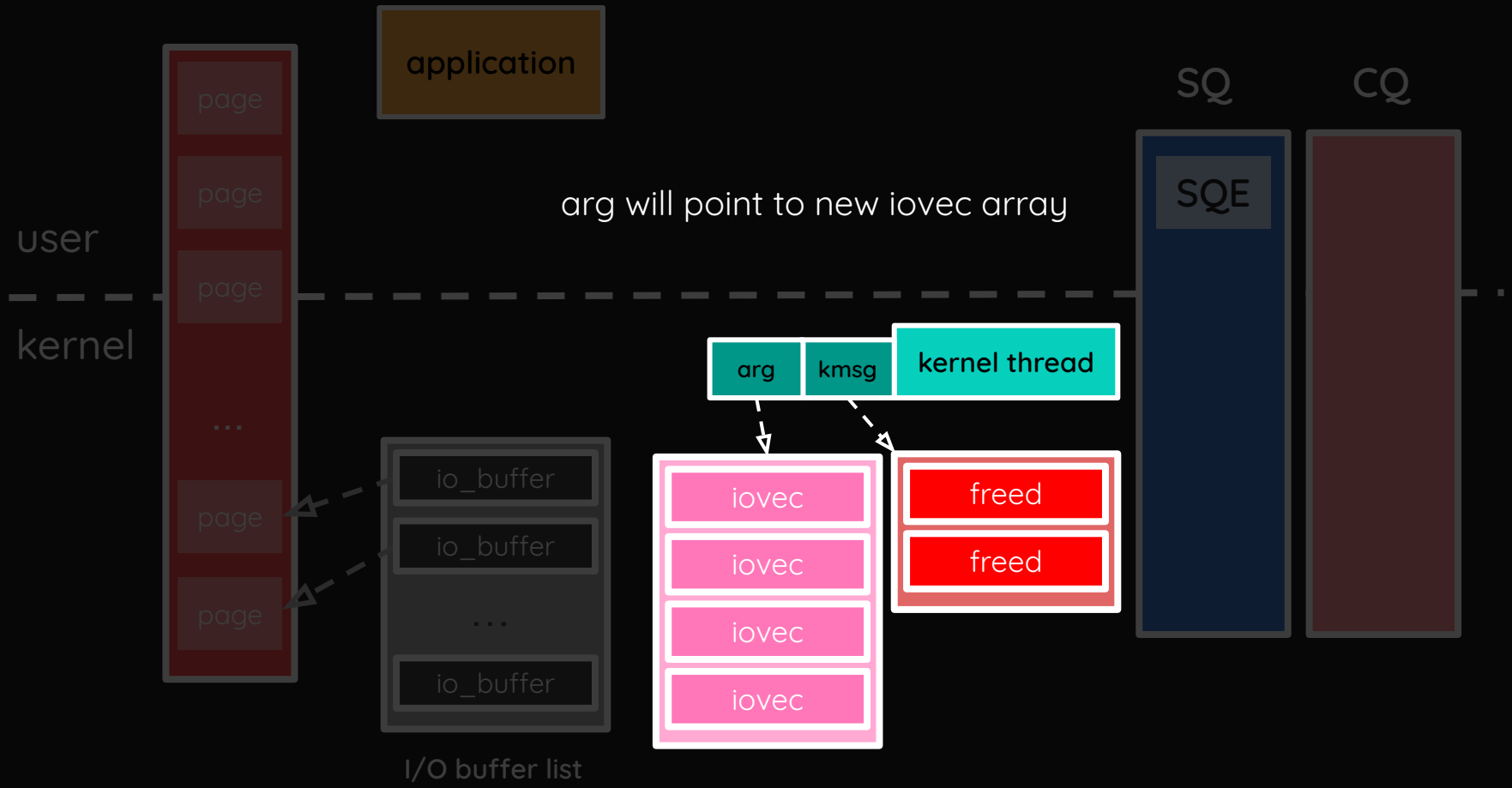

application
page
page
page
SQ
CQ
SQE
IORING_OP_SEND
IOSQE_BUFFER_SELECT | IORING_RECVSEND_BUFFER
user
kernel
IORING_RECVSEND_BUFFER
If set and provided buffers are used with
IOSQE_BUFFER_SELECT , the receive operation will attempt to
fill multiple buffers with rather than just pick a single
buffer to fill. To receive multiple buffers in a single
receive, the buffer group ID set in the SQE must be of the
ring provided type. If set, the CQE res field indicates
the total number of bytes received, and the buffer ID
returned in the CQE flags field indicates the first buffer
in the receive operation. The application must process the
indicated initial buffer ID and until all res bytes have
been seen to know which is the last buffer in the receive
I/O buffer list

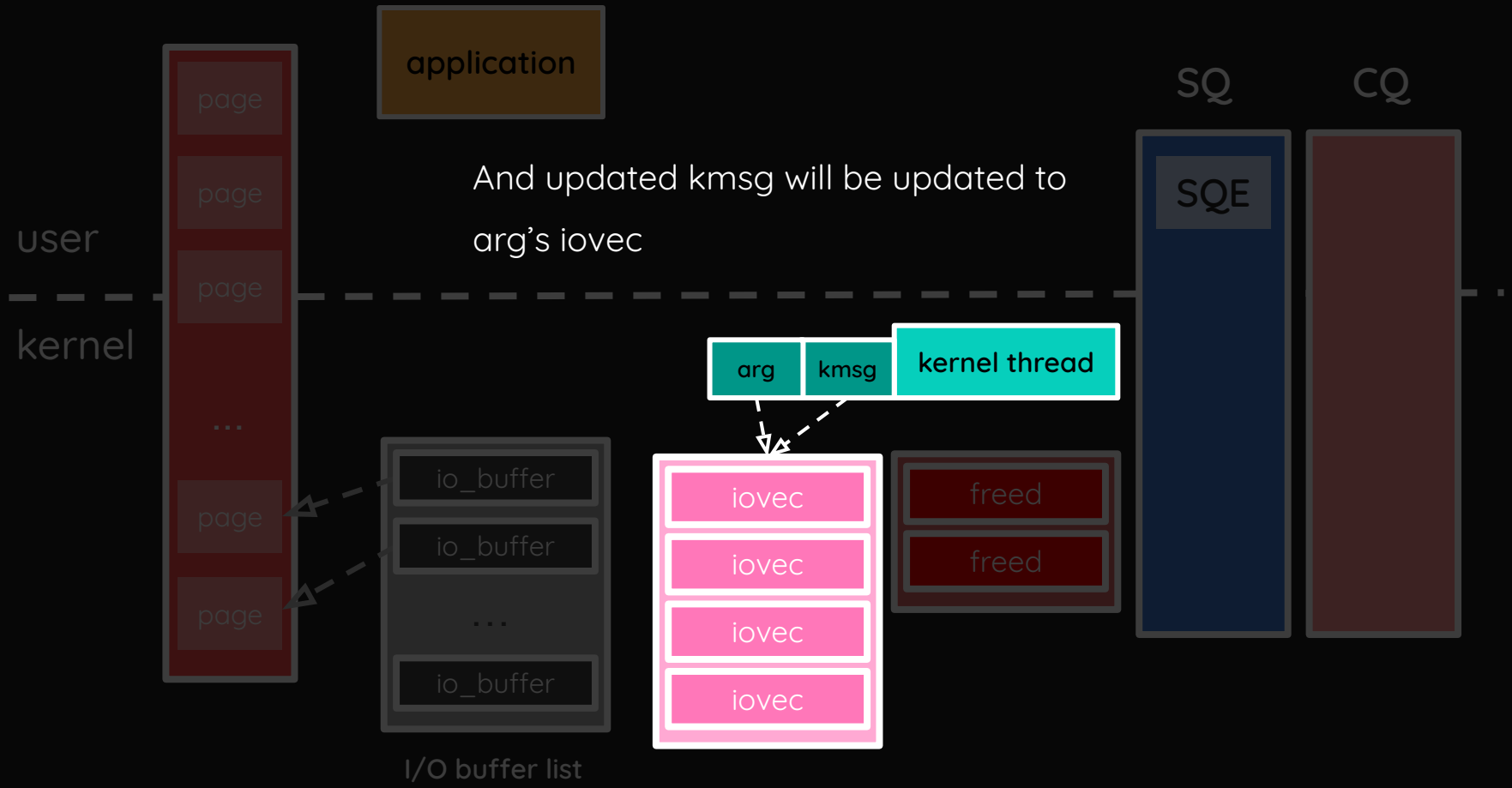

application
SQ
CQ
SQE
page
page
page
...
user
kernel
io_send will use multiple buffers
to store the send / recv data
arg
kmsg
kernel thread
io_buffer
io_buffer
...
io_buffer
iovec
iovec
AAA...
AAA...
I/O buffer list


application
SQ
CQ
SQE
If the first send needs 2 iovec, and
second send needs 4 iovec
user
kernel
page
page
page
...
page
page
arg
kmsg
kernel thread
io_buffer
io_buffer
...
io_buffer
iovec
iovec
I/O buffer list

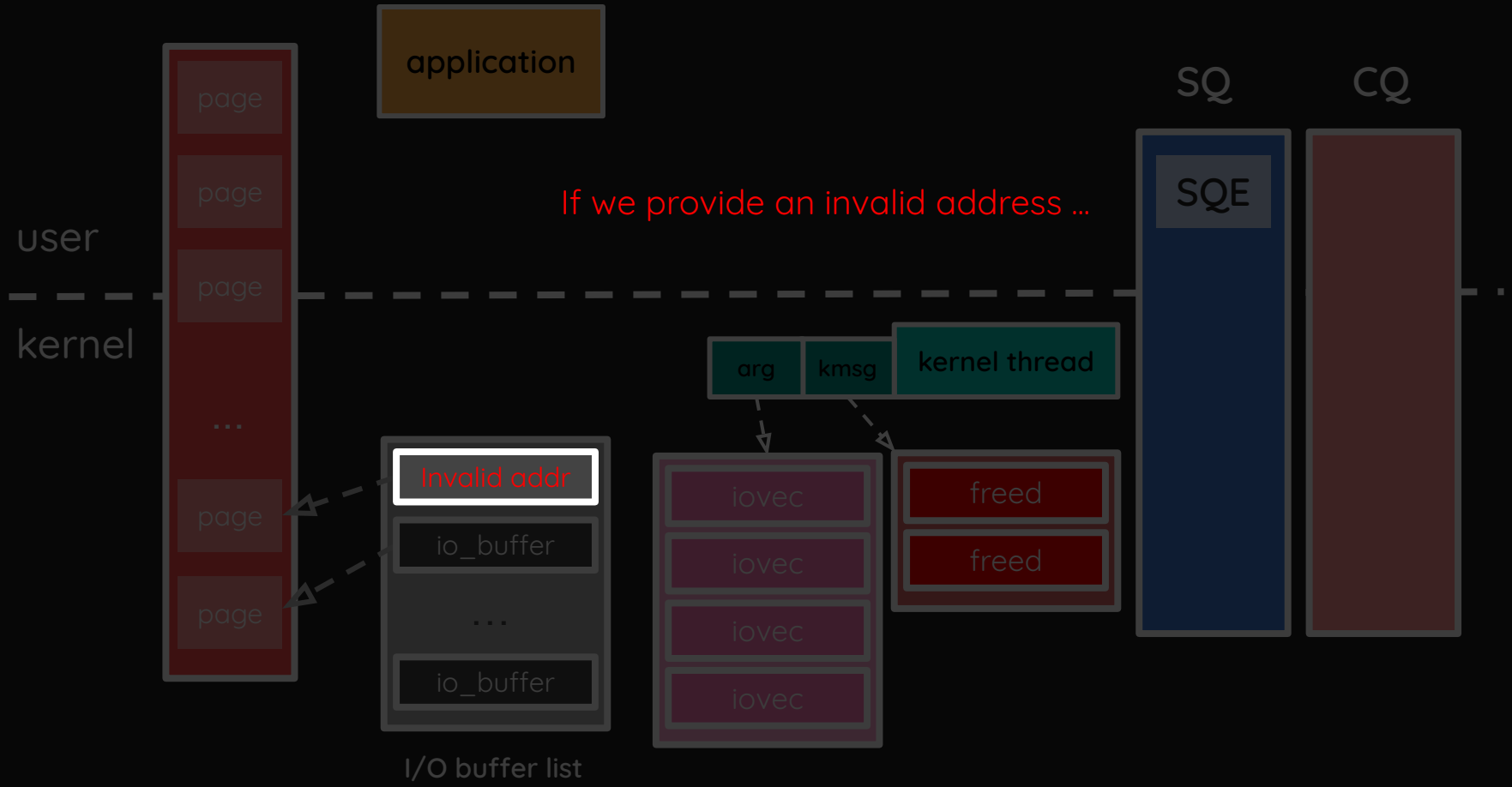

application
SQ
CQ
SQE
It will need to expand iovec array
user
kernel
page
page
page
...
page
page
arg
kmsg
kernel thread
io_buffer
io_buffer
...
io_buffer
iovec
iovec
I/O buffer list


application
SQ
CQ
SQE
original kmsg iovec will be freed
user
kernel
page
page
page
...
page
page
arg
kmsg
kernel thread
io_buffer
io_buffer
...
io_buffer
iovec
iovec
iovec
iovec
freed
freed
I/O buffer list


application
SQ
CQ
SQE
arg will point to new iovec array
user
kernel
page
page
page
...
page
page
arg
kmsg
kernel thread
io_buffer
io_buffer
...
io_buffer
iovec
iovec
iovec
iovec
freed
freed
I/O buffer list


application
SQ
CQ
SQE
And updated kmsg will be updated to
arg's iovec
user
kernel
page
page
page
...
page
page
arg
kmsg
kernel thread
io_buffer
io_buffer
...
io_buffer
iovec
iovec
iovec
iovec
freed
freed
I/O buffer list

But …


application
SQ
CQ
SQE
If we provide an invalid address ...
user
kernel
page
page
page
...
page
page
arg
kmsg
kernel thread
Invalid addr
io_buffer
...
io_buffer
iovec
iovec
iovec
iovec
freed
freed
I/O buffer list



/io_uring/kbuf.c#L287 (v7.2-rc1)

```
static int io_ring_buffers_peek(struct io_kiocb *req, struct buf_sel_arg *arg,
                struct io_buffer_list *bl)
{
    ...
    if (arg->mode & KBUF_MODE_EXPAND && nr_avail > nr_iovs && arg->max_len) {
        iov = kmalloc_objs(struct iovec, nr_avail);
        if (arg->mode & KBUF_MODE_FREE)
            kfree(arg->iovs);
        arg->iovs = iov;
        nr_iovs = nr_avail;
```

```
if (unlikely(!access_ok(iov->iov_base, len))) {
    if (arg->iovs != org_iovs)
        kfree(arg->iovs);
    return -EFAULT;
}
```





/io_uring/net.c#L620 (v7.2-rc1)

```
struct buf_sel_arg arg =
...

if (kmsg->vec.iovec) {
    arg.nr_iovs = kmsg->vec.nr;
    arg.iovs = kmsg->vec.iovec;
    arg.mode = KBUF_MODE_FREE;
```



```
ret = io_buffers_select(req, &arg, sel, issue_flags);
if (unlikely(ret < 0))
    return ret;
```

```
if (arg.iovs != &kmsg->fast_iov && arg.iovs != kmsg->vec.iovec) {
    kmsg->vec.nr = ret;
    kmsg->vec.iovec = arg.iovs;
    req->flags |= REQ_F_NEED_CLEANUP;
}
```



/io_uring/net.c#L620 (v7.2-rc1)

```
struct buf_sel_arg arg =
...

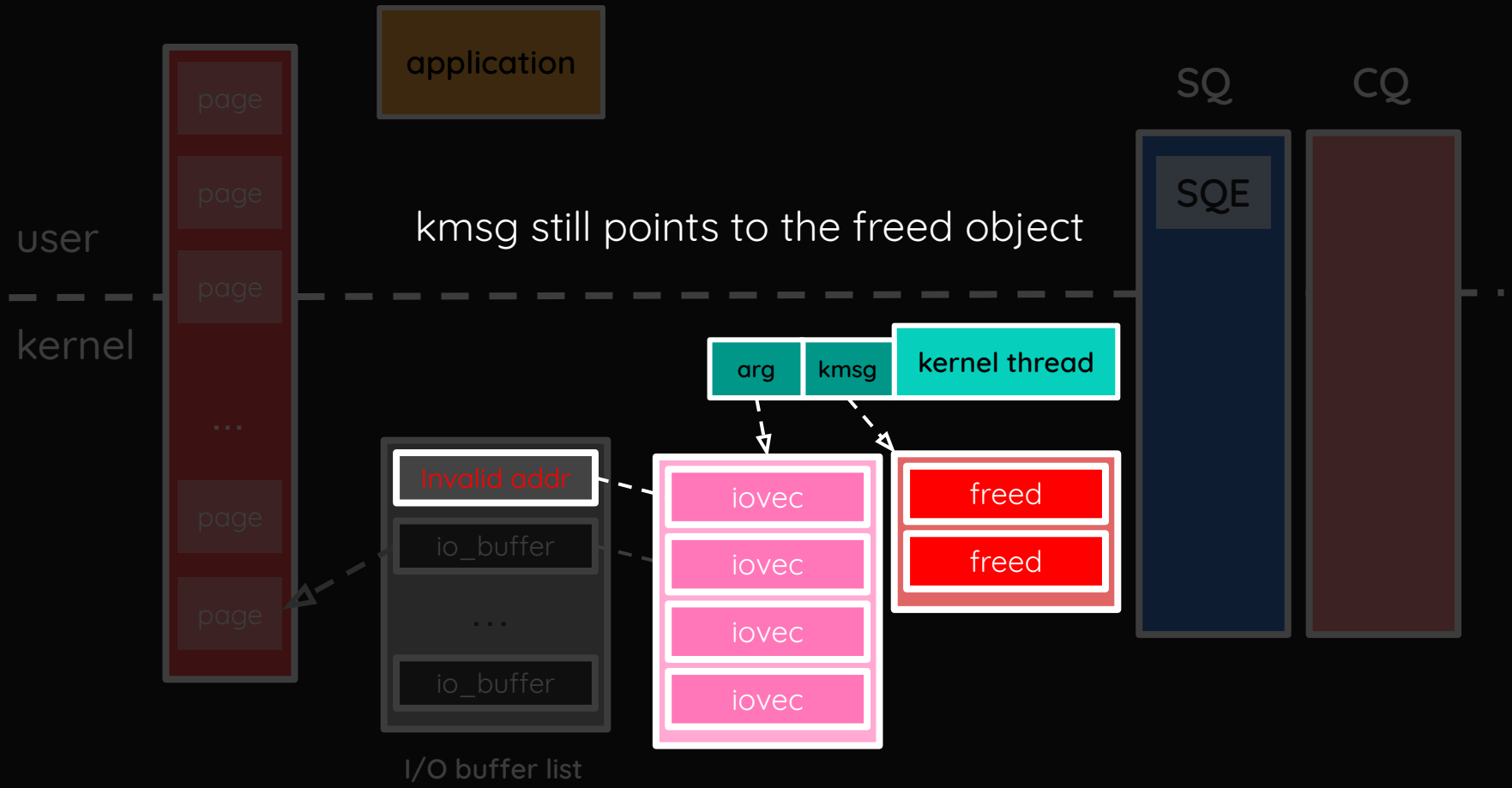
if (kmsg->vec.iovec) {
    arg.nr_iovs = kmsg->vec.nr;
    arg.iovs = kmsg->vec.iovec;
    arg.mode = KBUF_MODE_FREE;
}

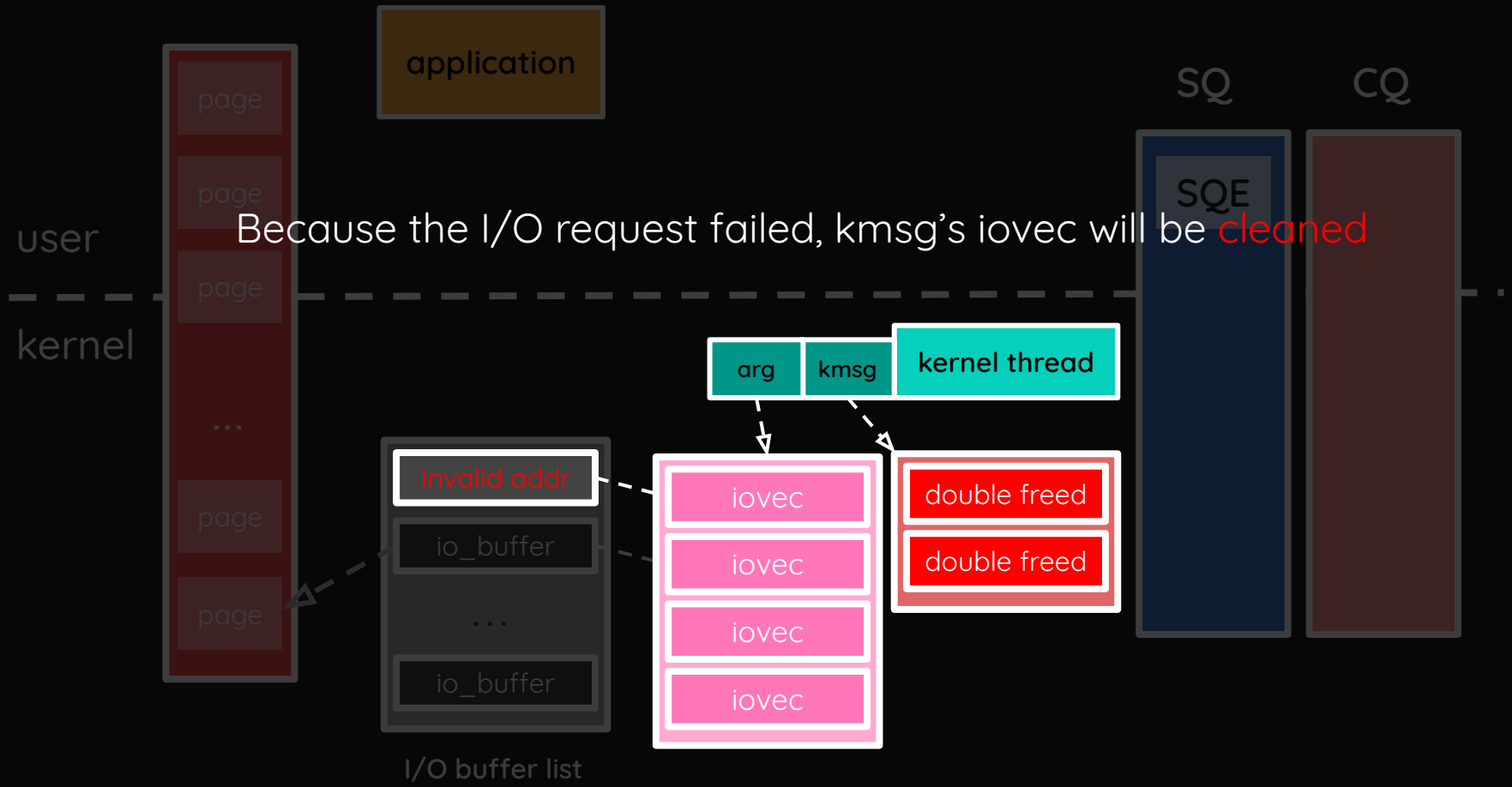
ret = io_buffers_select(req, &arg, sel, issue_flags);
if (unlikely(ret < 0))
    return ret;

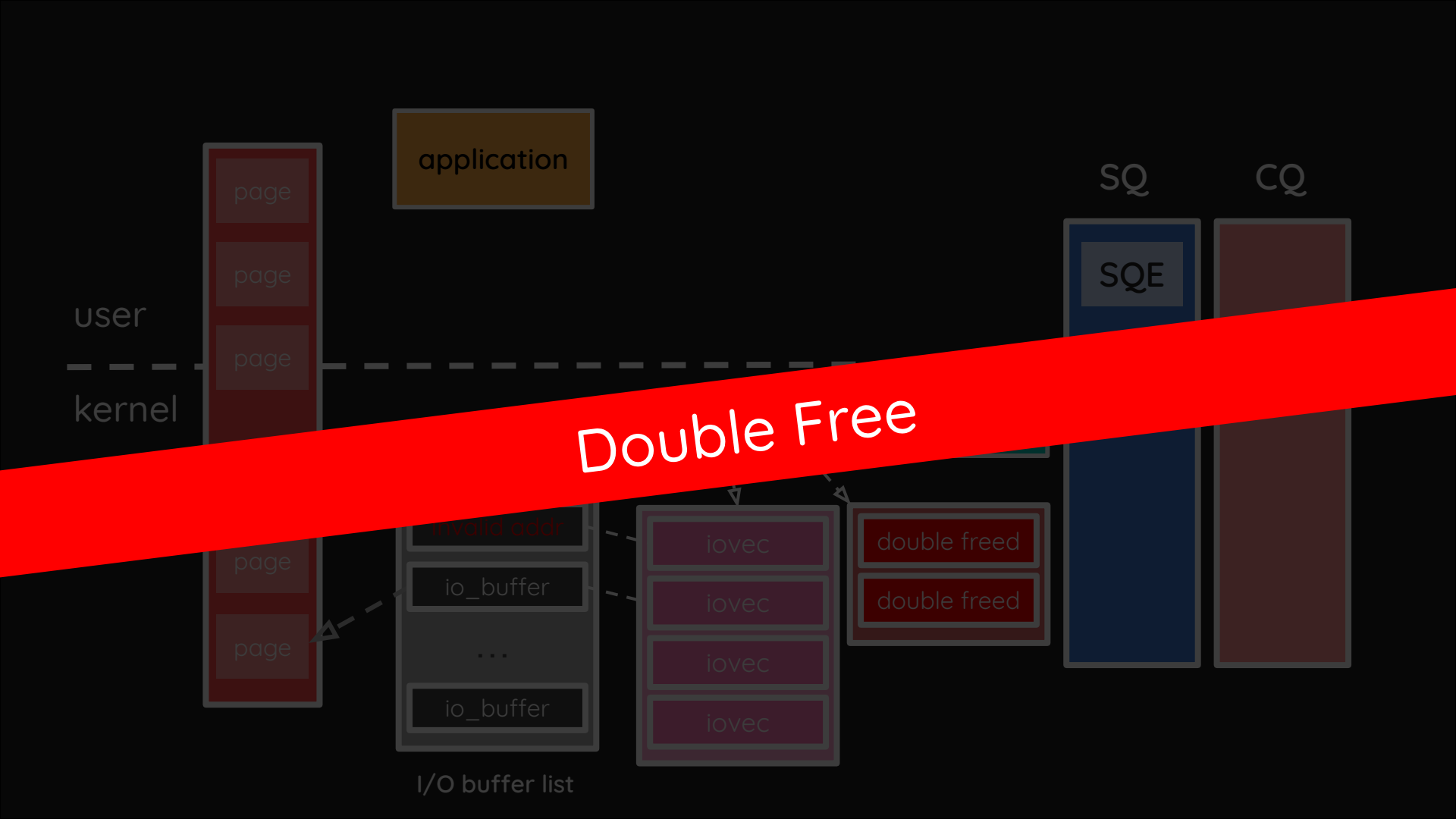
if (arg.iovs != &kmsg->fast_iov && arg.iovs != kmsg->vec.iovec) {
```



kmsg->vec.iovec = arg.iovs;


application
SQ
CQ
SQE
page
page
page
...
page
page
user
kernel
kmsg still points to the freed object
arg
kmsg
kernel thread
Invalid addr
io_buffer
...
io_buffer
iovec
iovec
iovec
iovec
freed
freed
I/O buffer list


application
SQ
CQ
SQE
user
kernel
page
page
page
...
page
page
Because the I/O request failed, kmsg's iovec will be cleaned
arg
kmsg
kernel thread
Invalid addr
io_buffer
...
io_buffer
iovec
iovec
iovec
iovec
double freed
double freed
I/O buffer list


application
SQ
CQ
SQE
user
kernel
page
page
page
page
page
invalid addr
io_buffer
...
io_buffer
iovec
iovec
iovec
iovec
double freed
double freed
I/O buffer list
Double Free

How to get root ?

How to get root ?

> Double Free primitive

How to get root ?

> Double Free primitive

> kmalloc-N object double free

/io_uring/kbuf.c#L287 (v7.2-rc1)

```
static int io_ring_buffers_peek(struct io_kiocb *req, struct buf_sel_arg *arg,
                struct io_buffer_list *bl)
{
    ...
    if (arg->mode & KBUF_MODE_EXPAND && nr_avail > nr_iovs && arg->max_len) {
        iov = kmalloc_objs(struct iovec, nr_avail);
        ...
        arg->iovs = iov;
        nr_iovs = nr_avail;
    }
    ...
    do {
        if (unlikely(!access_ok(iov->iov_base, len))) {
            if (arg->iovs != org_iovs)
                kfree(arg->iovs);
            return -EFAULT;
        }
```

How to get root ?

> Double Free primitive 

> kmalloc-N object double free 

> Overwrite task_struct cred pointer to root cred

How to get root ?

- Double Free primitive 
- kmalloc-N object double free 
- Overwrite task_struct cred pointer to root cred
  - Leak KASLR
  - Leak task_struct address
  - Arbitrary write primitive

How to get root ?

- Double Free primitive ✓
- kmalloc-N object double free ✓
- Overwrite task_struct cred pointer to root cred
  - Leak KASLR
  - Leak task_struct address
  - Arbitrary write primitive

Double-Free Primitive Workflow

> Send 1: Provide n pbufs, there will be n corresponding iovecs

ex: 4 pbuf → 4 iovec (from 4 x 16 = kmalloc-64)

Double-Free Primitive Workflow

> Send 1: Provide n pbufs, there will be n corresponding iovecs

ex: 4 pbuf → 4 iovec (from 4 x 16 = kmalloc-64)

> Send 2: Provide more than n pbufs, with one pbuf assigned an invalid address

Double-Free Primitive Workflow

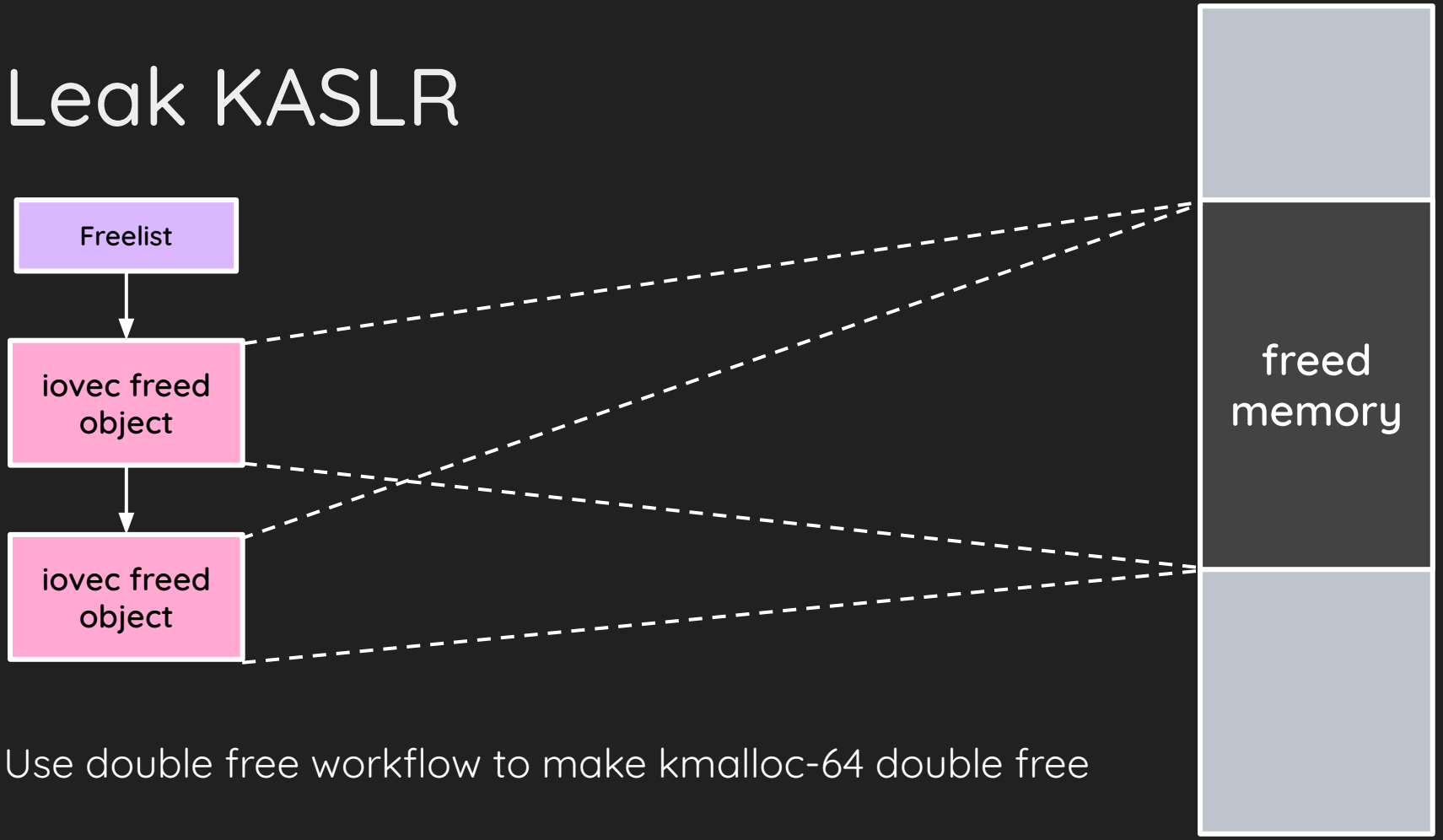
> Send 1: Provide n pbufs, there will be n corresponding iovecs

ex: 4 pbuf → 4 iovec (from 4 x 16 = kmalloc-64)

> Send 2: Provide more than n pbufs, with one pbuf assigned an invalid address

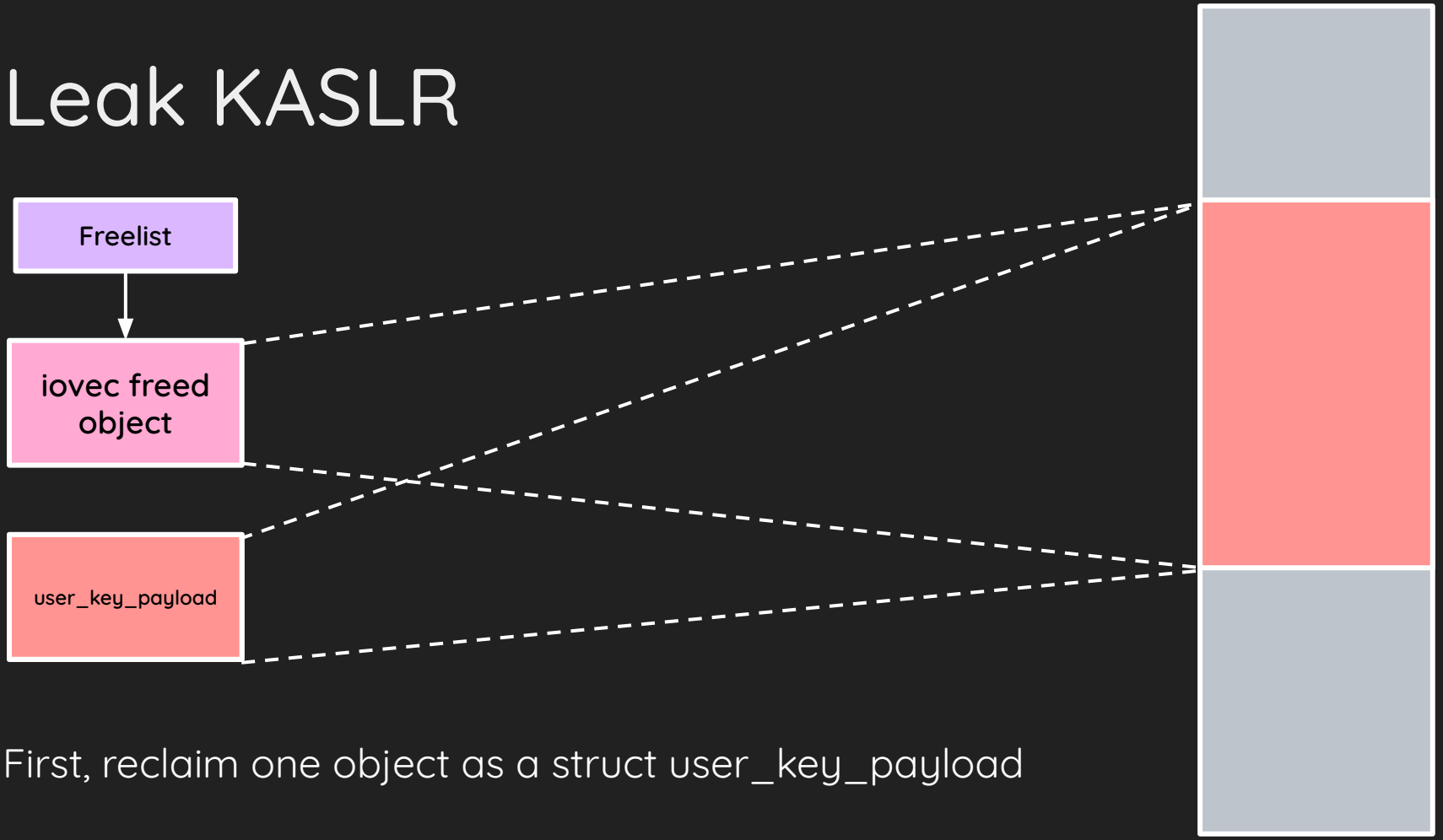
> The iovec array object allocated in Send 1 will be double-freed, leading to an arbitrary kmalloc-N double-free primitive.

Leak KASLR



Use double free workflow to make kmalloc-64 double free

Leak KASLR

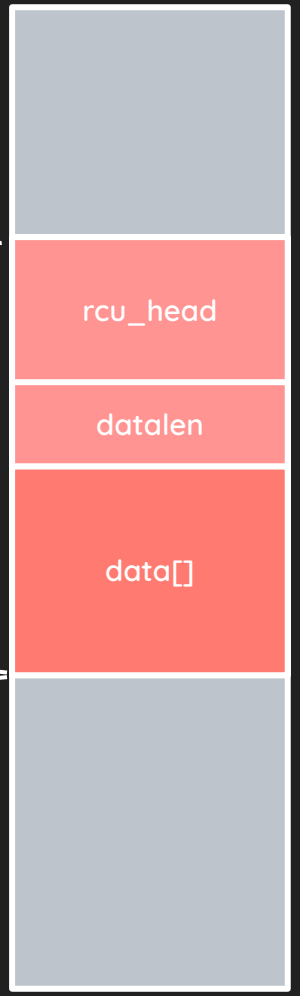


First, reclaim one object as a struct user_key_payload

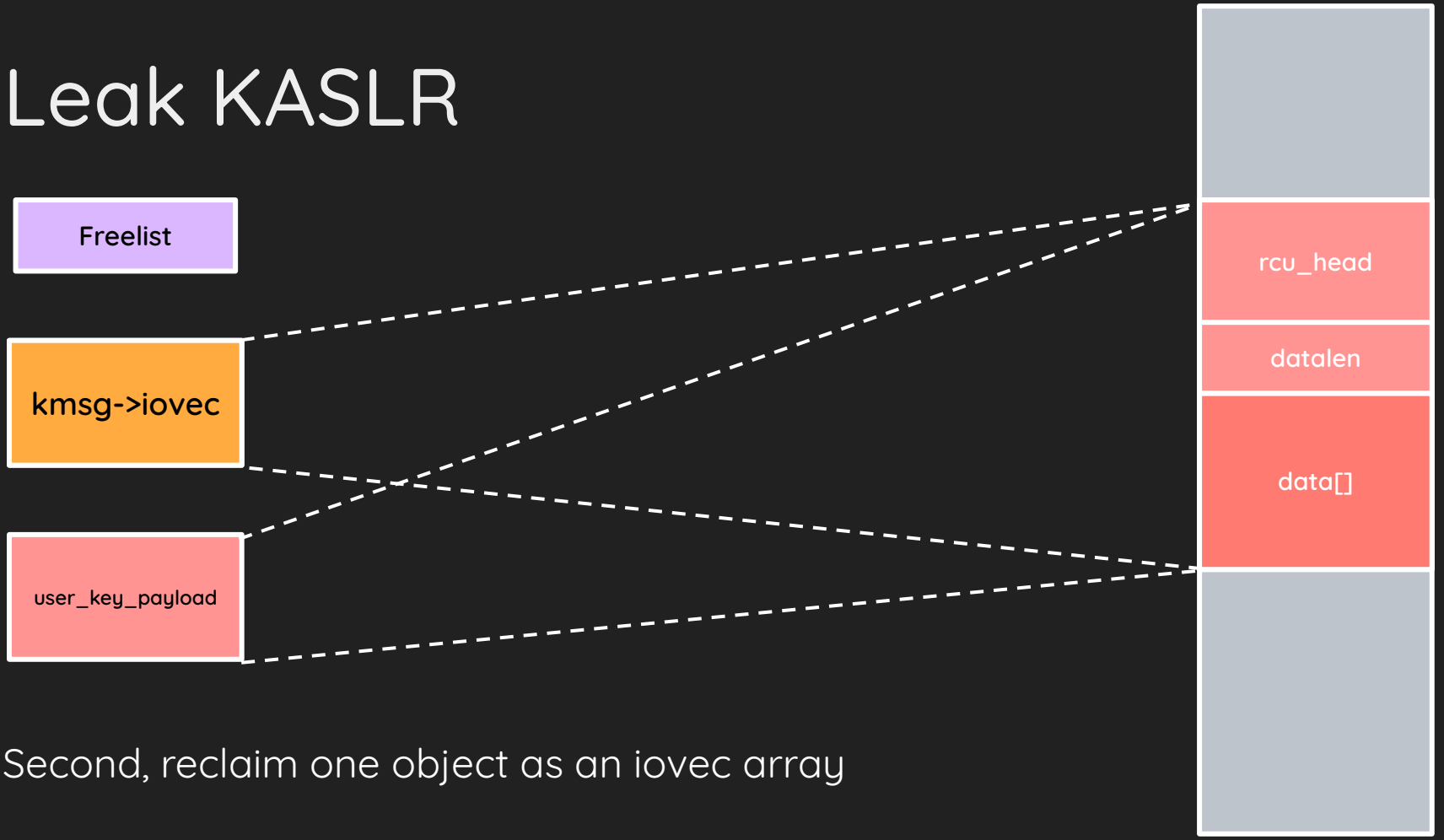
Leak KASLR

/include/keys/user-type.h#L27 (v7.2-rc1)

```
struct user_key_payload {
    struct rcu_head    rcu;         /* RCU destructor */
    unsigned short     datalen;     /* length of this data */
    char               data[] __aligned(__alignof__(u64)); /* actual data */
};
```

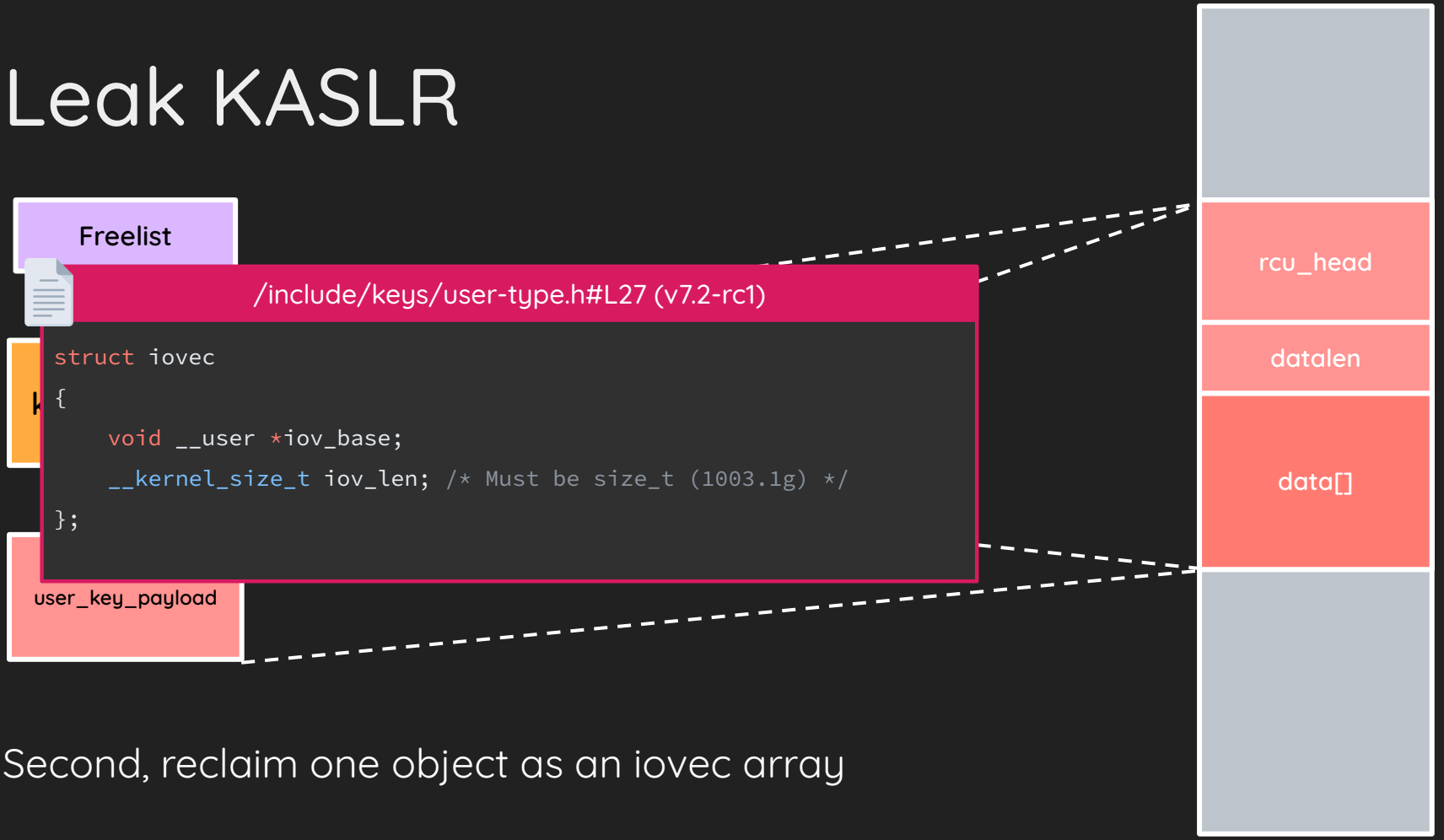


Leak KASLR



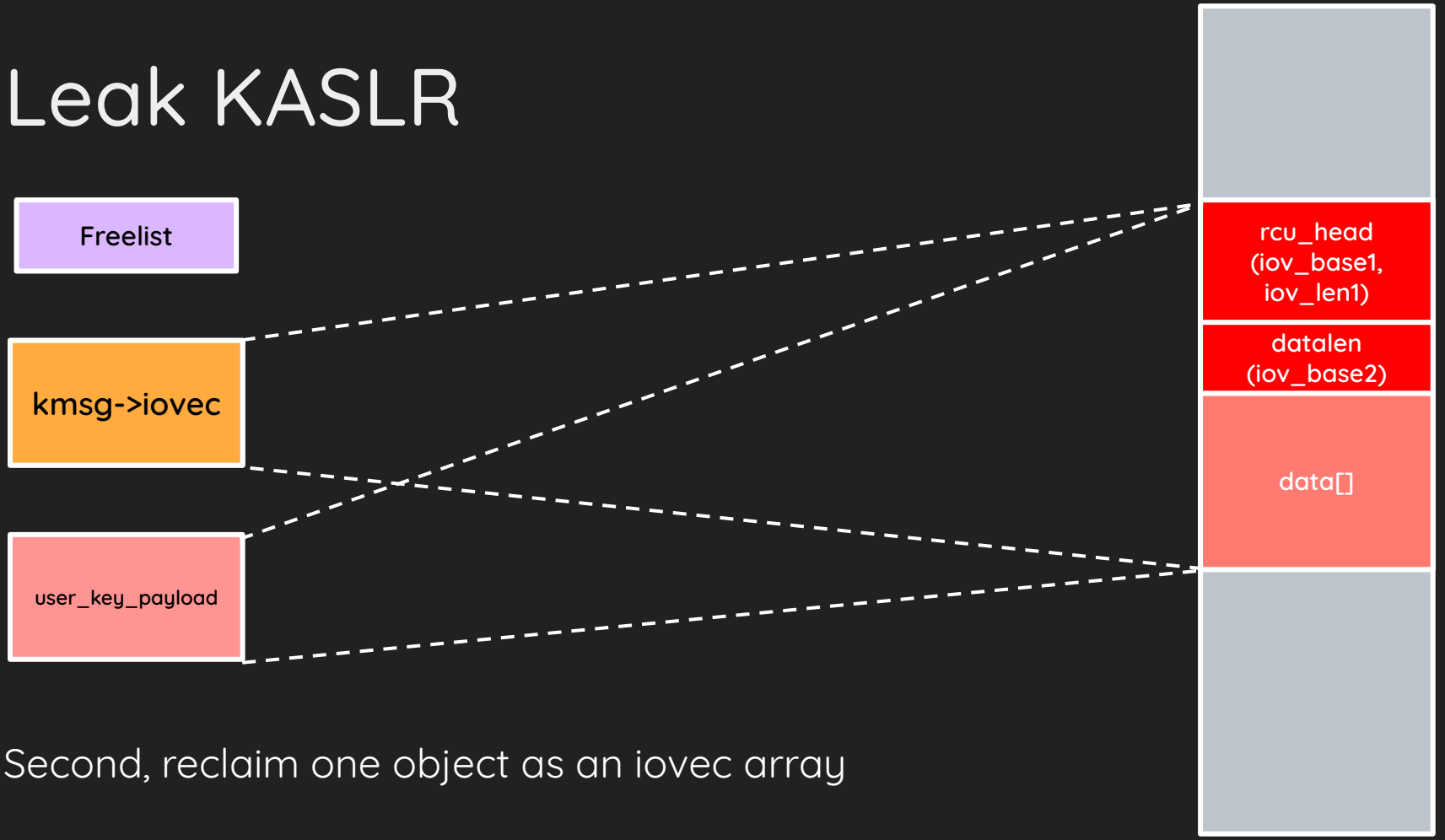
Second, reclaim one object as an iovec array

Leak KASLR



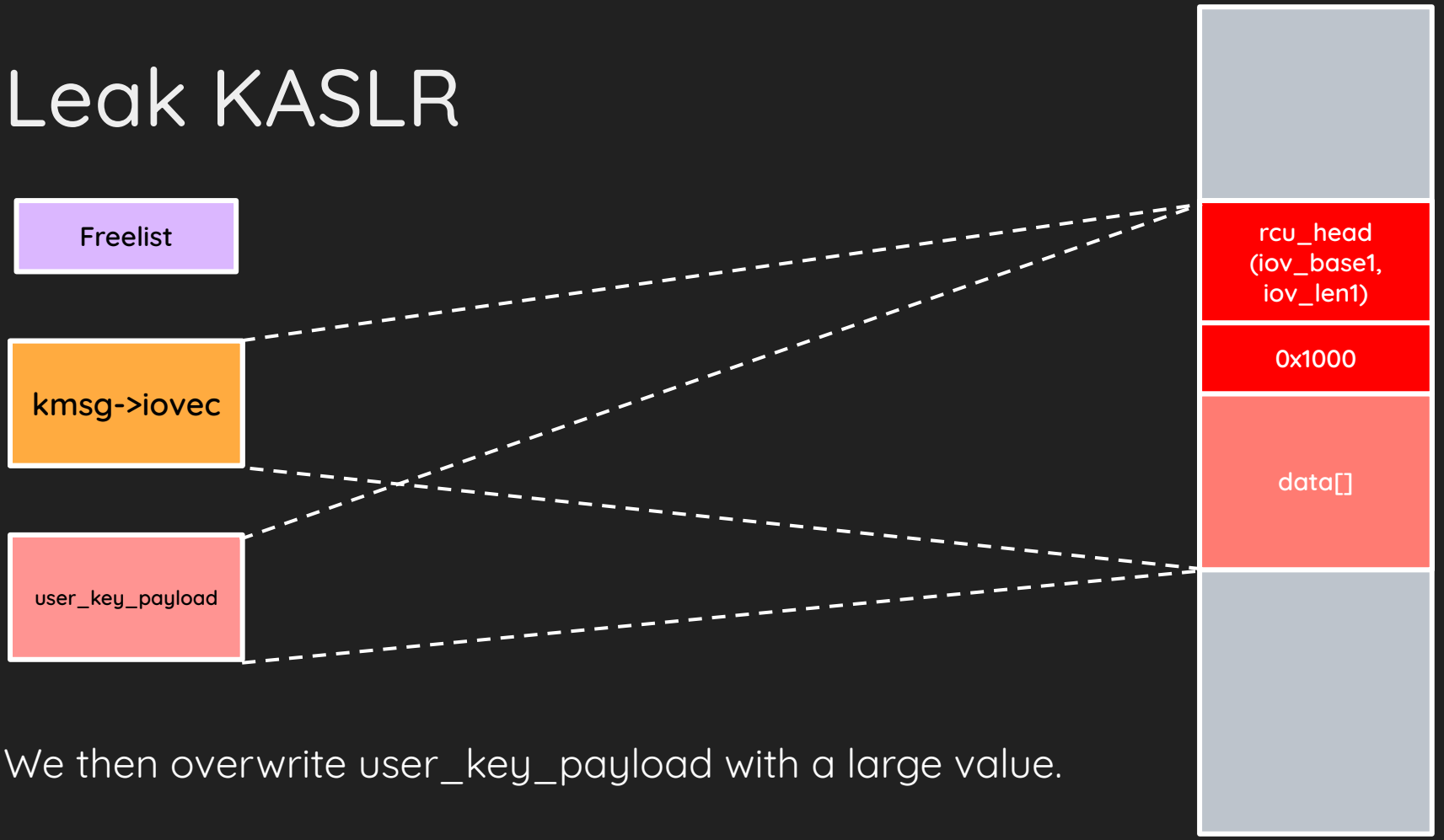
Second, reclaim one object as an iovec array

Leak KASLR



Second, reclaim one object as an iovec array

Leak KASLR



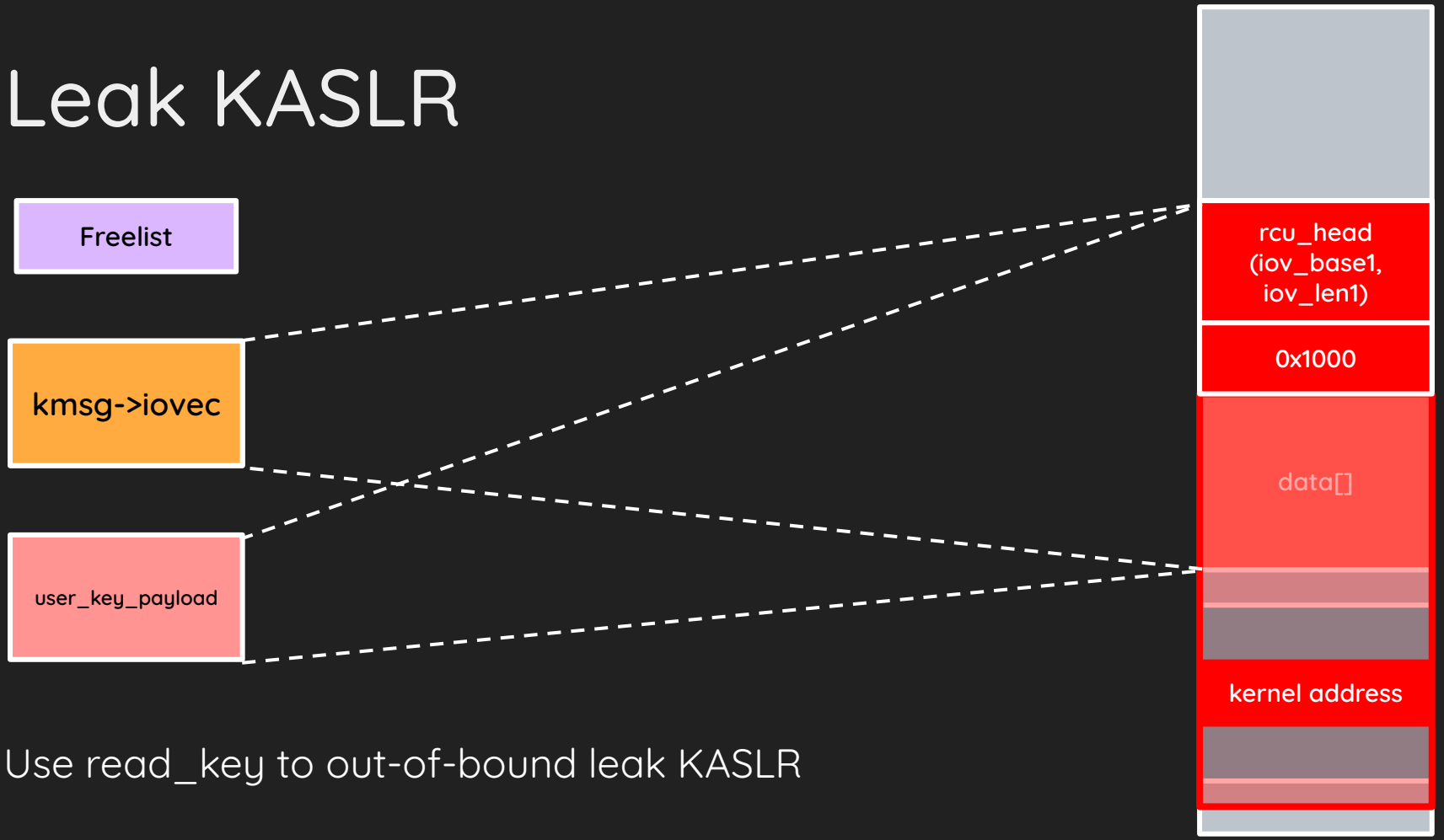
We then overwrite user_key_payload with a large value.

Leak KASLR



By using heap spraying, we can place a kernel address near user_key_payload

Leak KASLR



Use read_key to out-of-bound leak KASLR

Leak KASLR



Use read_key to out-of-bound leak KASLR

How to get root ?

- > Double Free primitive
- > kmalloc-N object double free
- > Overwrite task_struct cred pointer to root cred
  - > Leak KASLR
  - > Leak task_struct address
  - > Arbitrary write primitive

Leak task_struct

> Now we have an out-of-bounds read by overwriting user_key_payload

Leak task_struct

> Now we have an out-of-bounds read by overwriting user_key_payload

> If we can make the current process's task_struct address appear near user_key_payload, we can leak this address

Leak task_struct

> Now we have an out-of-bounds read by overwriting user_key_payload

> If we can make the current process's task_struct address appear near user_key_payload, we can leak this address

IORING_SETUP_SINGLE_ISSUER

IORING_SETUP_SINGLE_ISSUER

When created with IORING_SETUP_SINGLE_ISSUER set, the io_uring instance is bound to the current task and can only be submitted by that single task



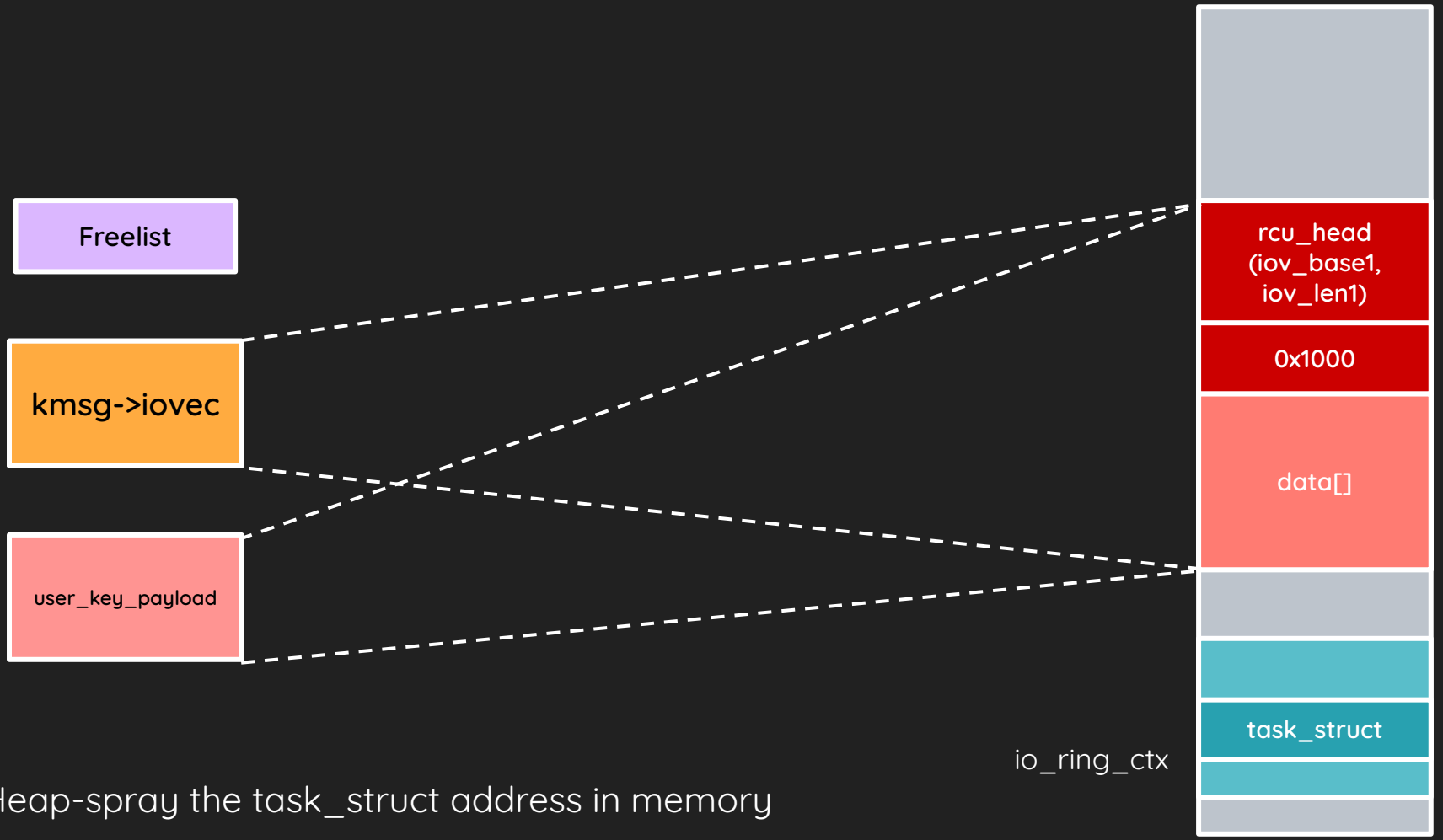
/io_uring/io_uring.c#L3065 (v7.2-rc1)

```
if (ctx->flags & IORING_SETUP_SINGLE_ISSUER
    && !(ctx->flags & IORING_SETUP_R_DISABLED))
    ctx->submitter_task = get_task_struct(current);
```

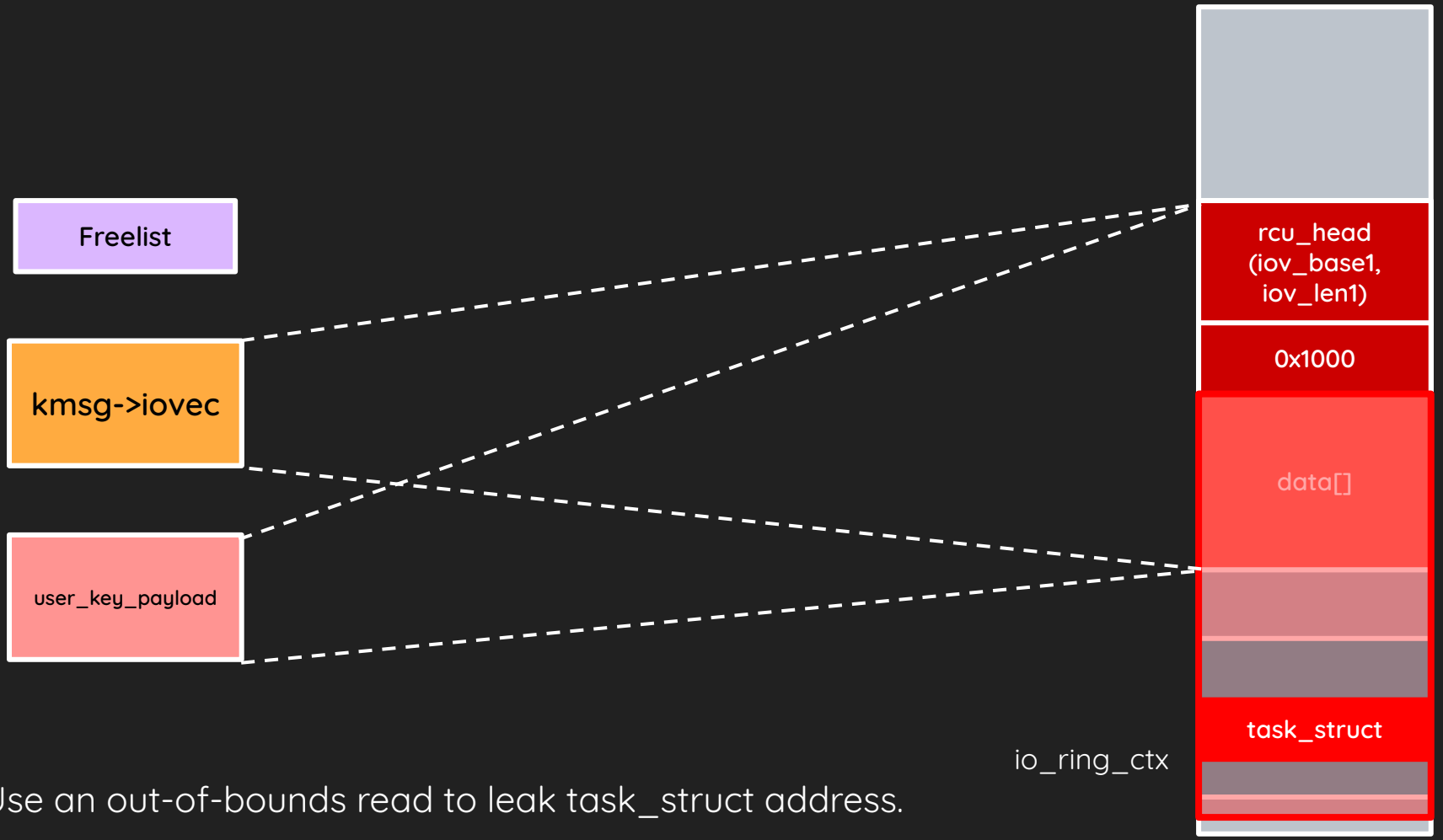
PS: ctx (io_ring_ctx from kmalloc-2048)

IORING_SETUP_SINGLE_ISSUER
A hint to the kernel that only a single task (or thread) will submit requests, which is used for internal optimisations. The submission task is either the task that created the ring, or if **IORING_SETUP_R_DISABLED** is specified then it is the task that enables the ring through io_uring_register(2). The kernel enforces this rule, failing requests with **-EEXIST** if the restriction is violated. Note that when **IORING_SETUP_SQPOLL** is set it is considered that the polling task is doing all submissions on behalf of the userspace and so it always complies with the rule disregarding how many userspace tasks do io_uring_enter(2). Available since 6.0.

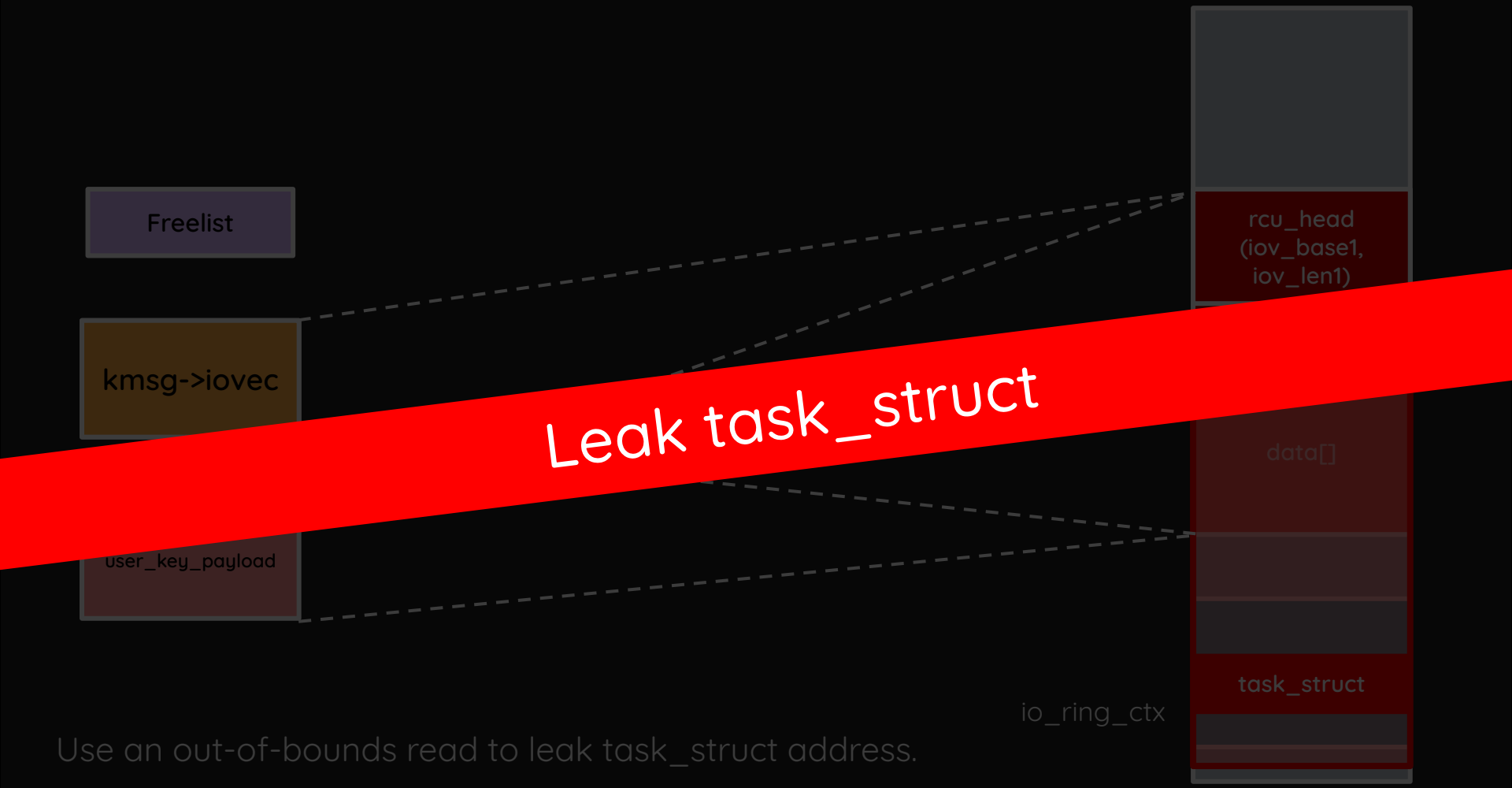
https://man7.org/linux/man-pages/man2/io_uring_setup.2.html



Heap-spray the task_struct address in memory



Use an out-of-bounds read to leak task_struct address.

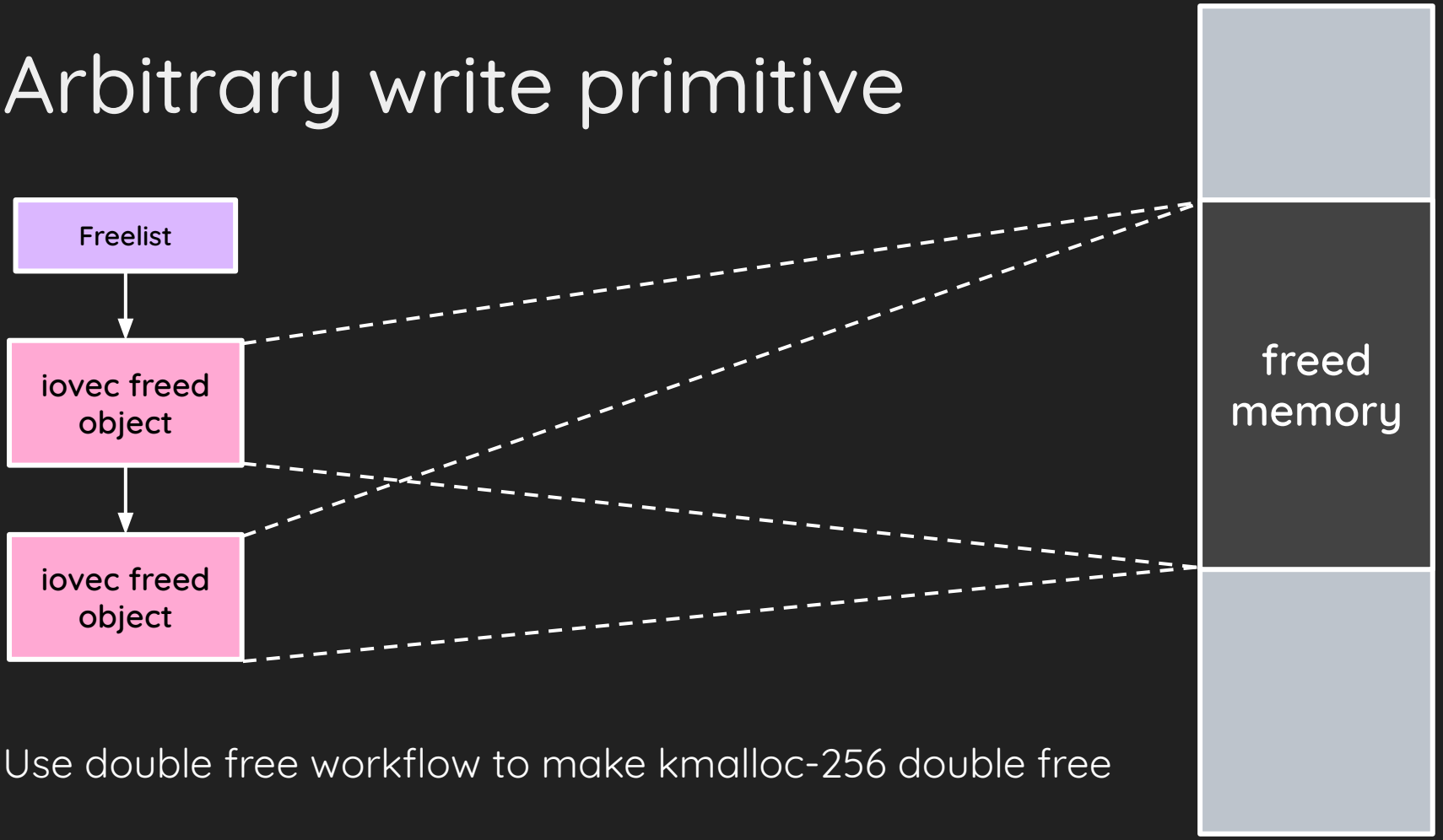


Use an out-of-bounds read to leak task_struct address.

How to get root ?

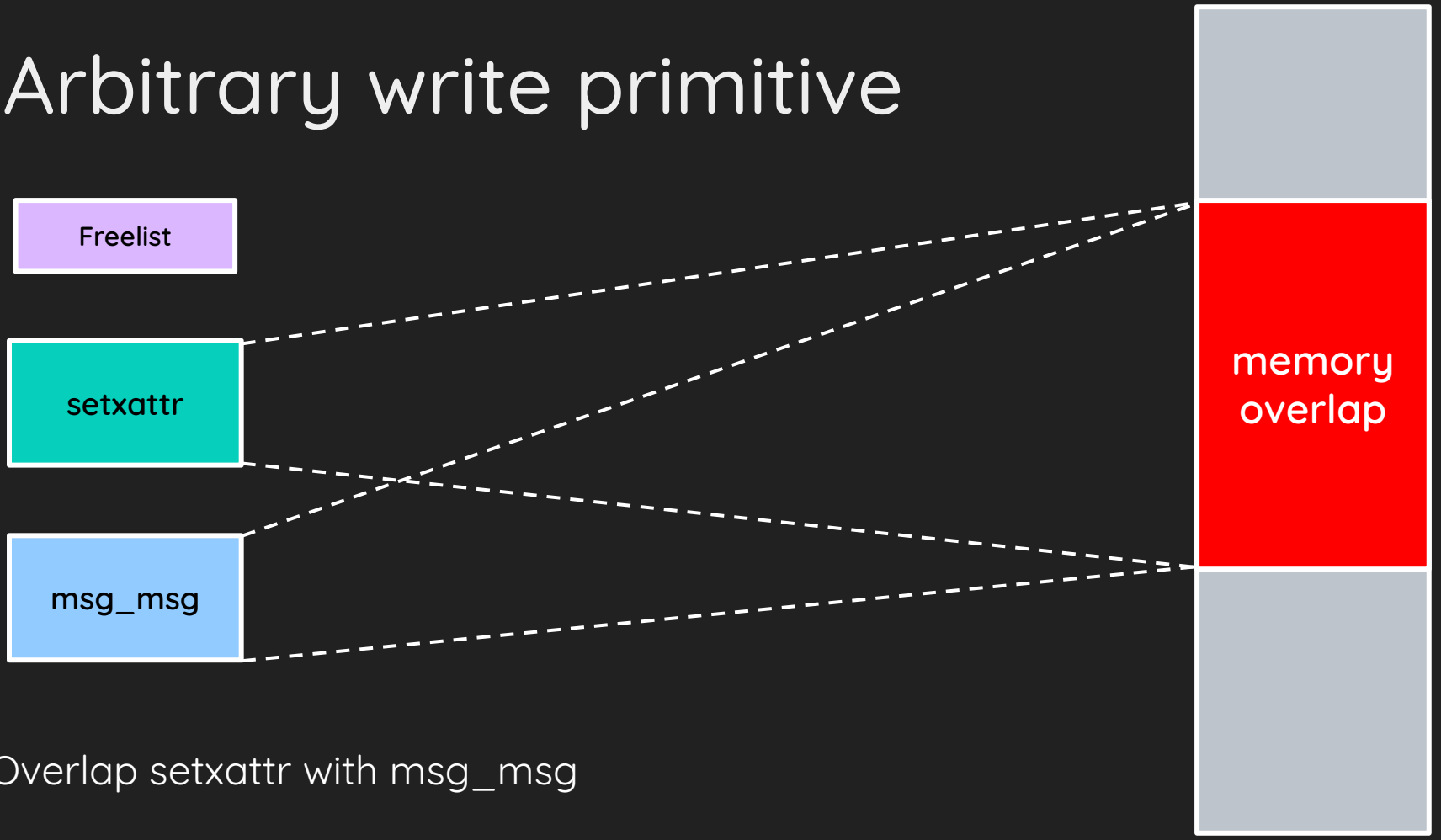
- Double Free primitive 
- kmalloc-N object double free 
- Overwrite task_struct cred pointer to root cred
  - Leak KASLR 
  - Leak task_struct address 
  - Arbitrary write primitive

Arbitrary write primitive

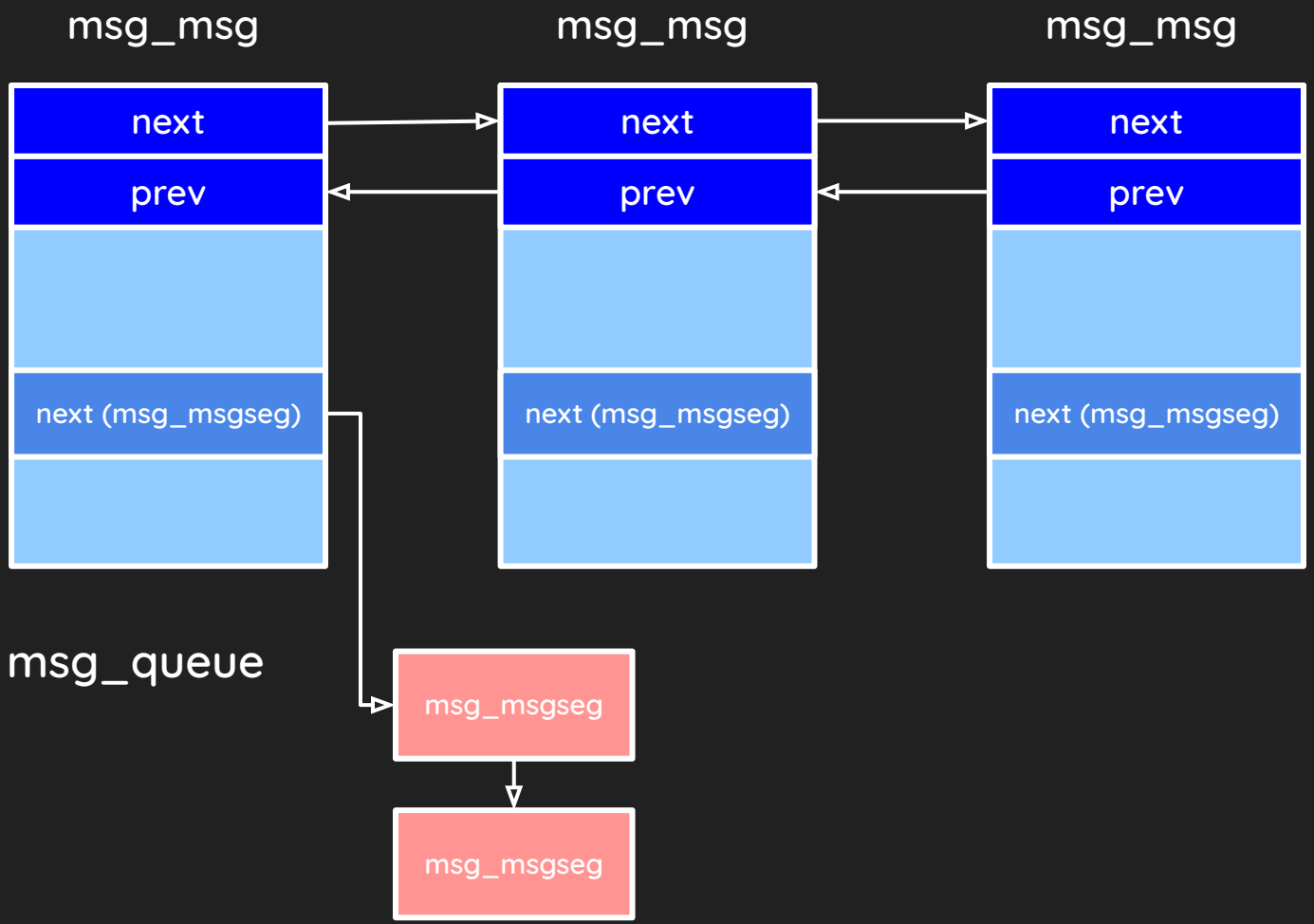


Use double free workflow to make kmalloc-256 double free

Arbitrary write primitive



Overlap setxattr with msg_msg

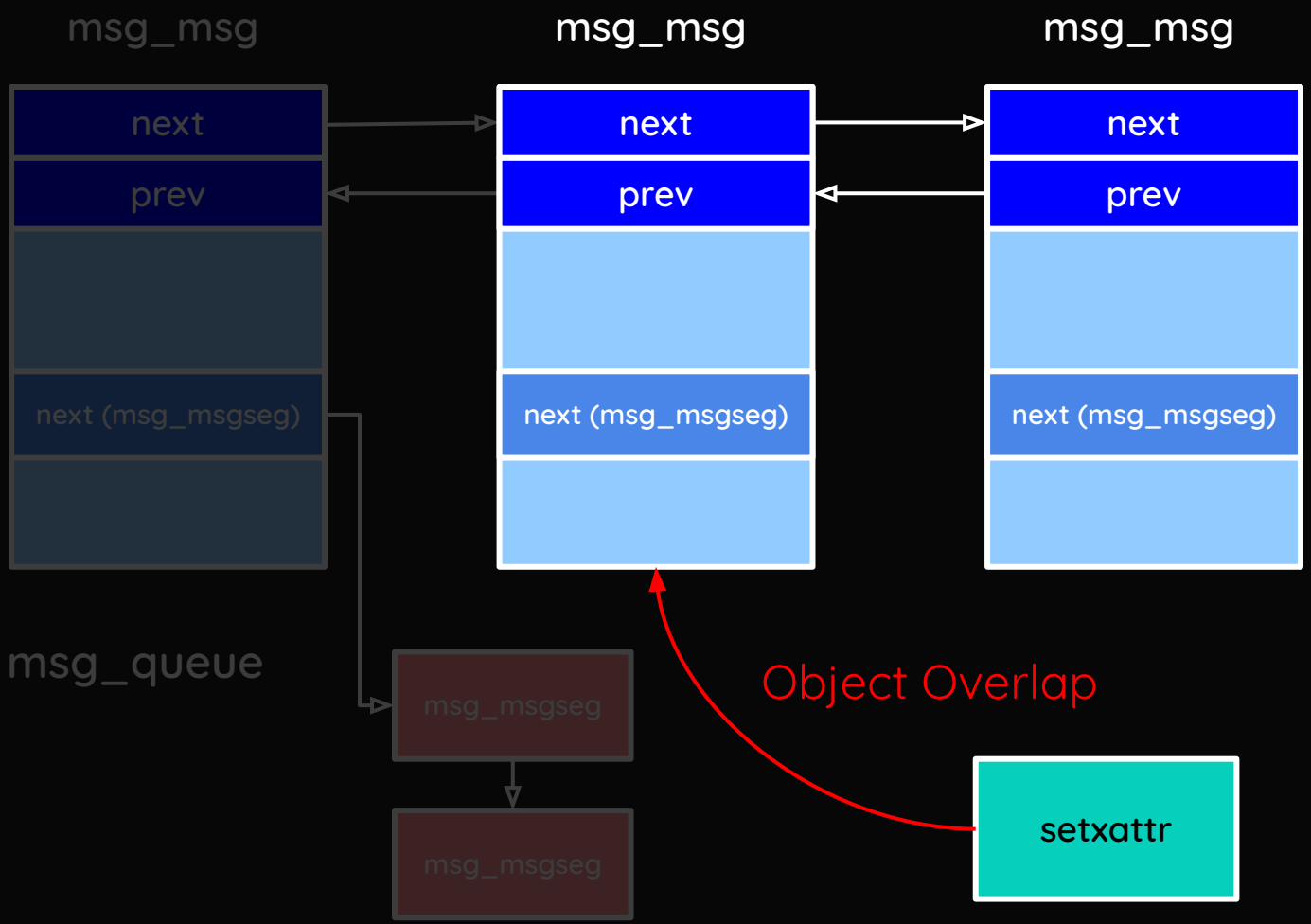

msg_msg
msg_msg
msg_msg
next
prev
next (msg_msgseg)
next
prev
next (msg_msgseg)
next
prev
next (msg_msgseg)
msg_queue
msg_msgseg
msg_msgseg

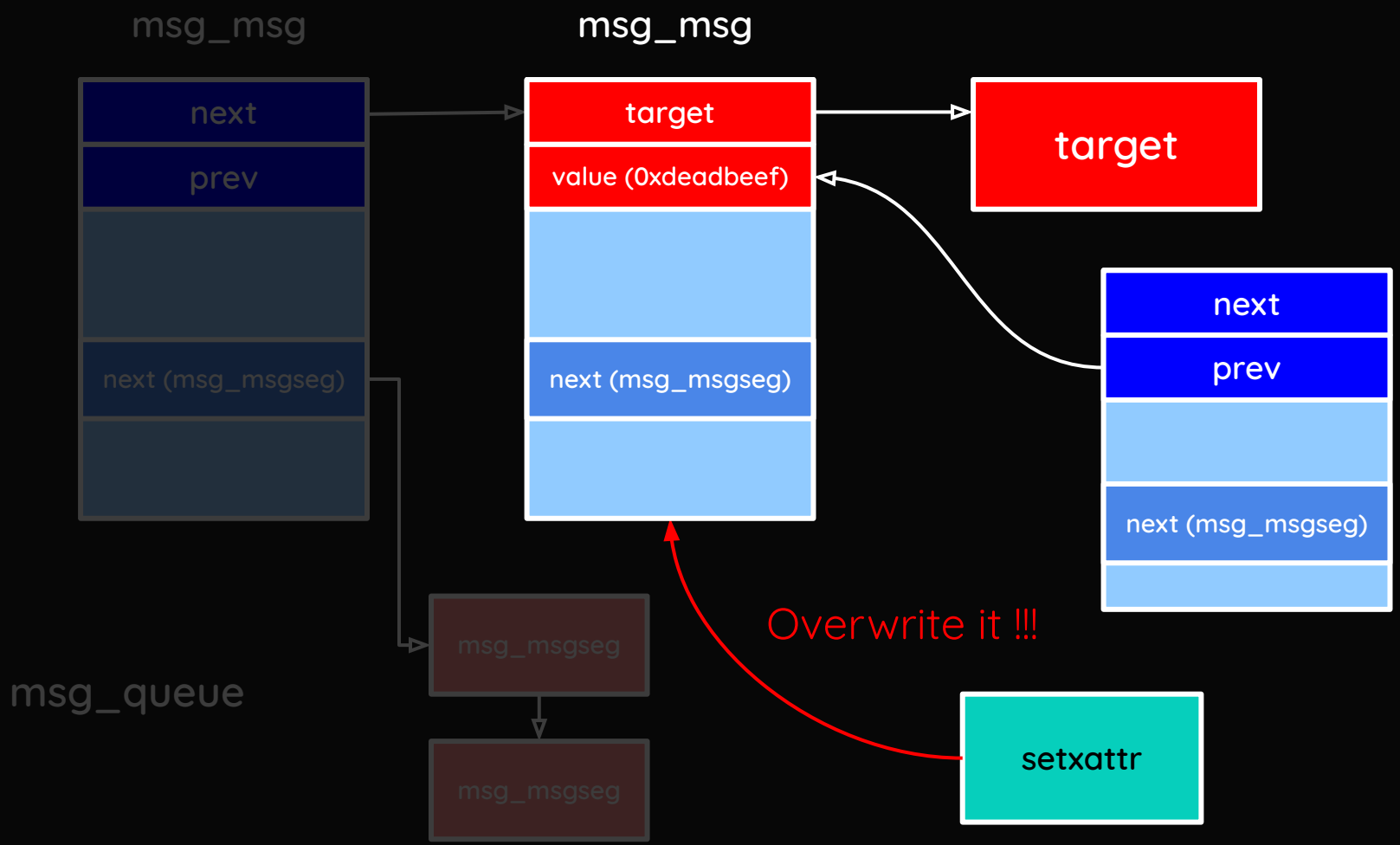
/ipc/msg.c#L1098 (v7.2-rc1)

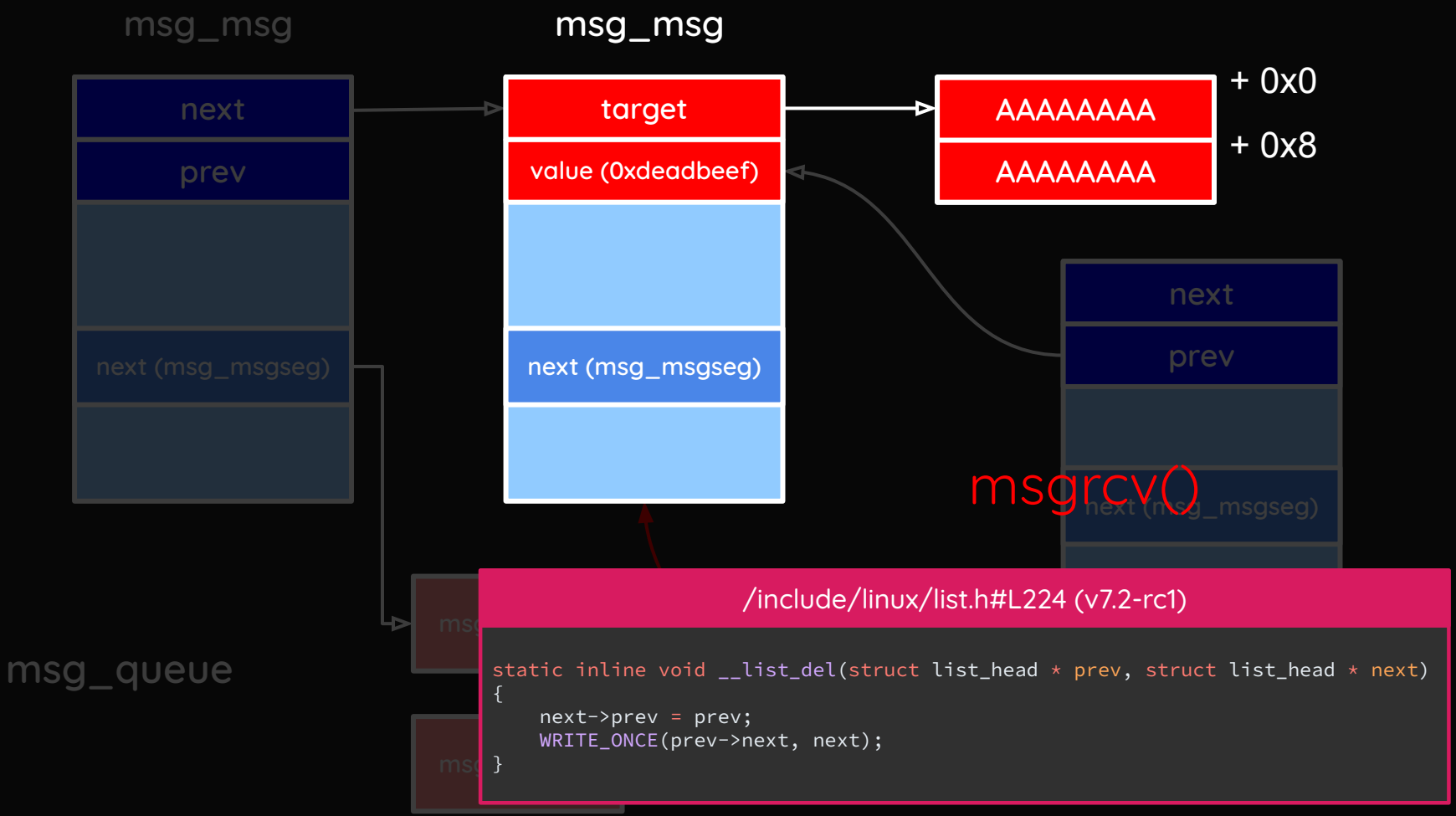
```
static long do_msgrcv(int msqid, void __user *buf, size_t bufsz, long msgtyp, int
msgflg,
           long (*msg_handler)(void __user *, struct msg_msg *, size_t))
{
    struct msg_queue *msq;
    ...
    for (;;) {
        ...
        msg = find_msg(msq, &msgtyp, mode);
        if (!IS_ERR(msg)) {
            ...
            list_del(&msg->m_list);
```

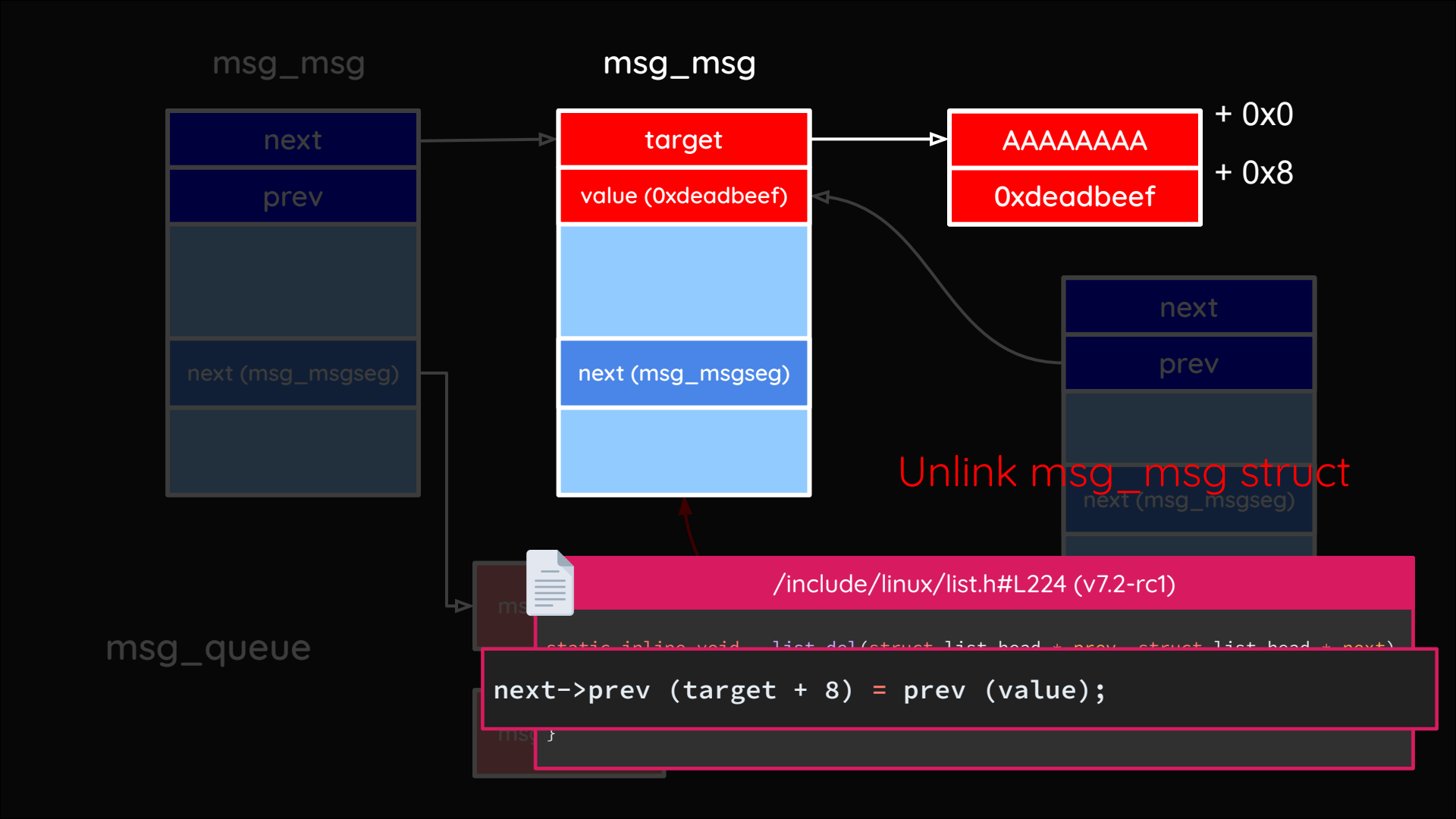
/include/linux/list.h#L224 (v7.2-rc1)

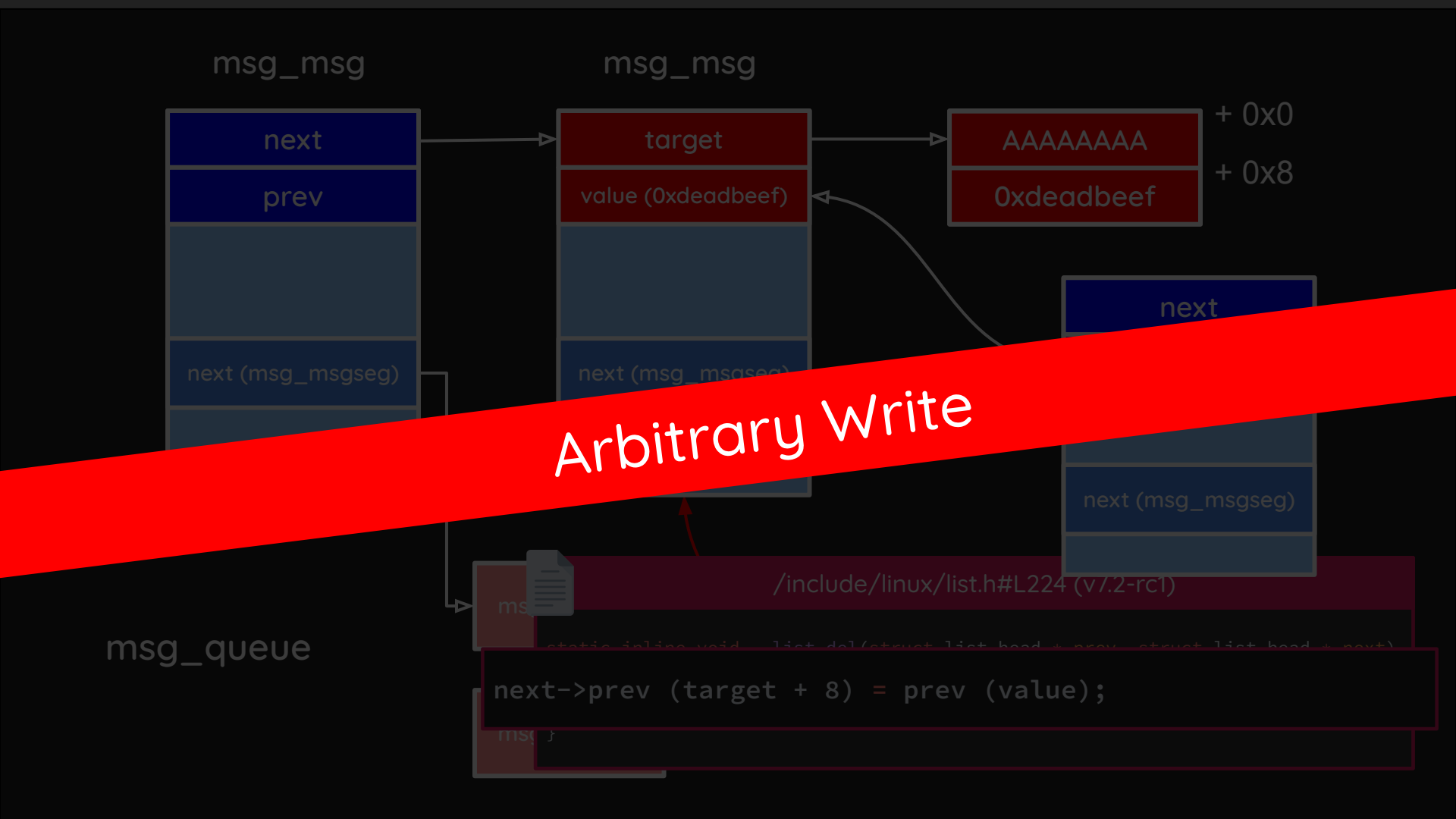
```
static inline void __list_del(struct list_head * prev, struct list_head * next)
{
    next->prev = prev;
    WRITE_ONCE(prev->next, next);
}
```


msg_msg
msg_msg
msg_msg
next
prev
next (msg_msgseg)
next
prev
next (msg_msgseg)
next
prev
next (msg_msgseg)
msg_queue
msg_msgseg
msg_msgseg
Object Overlap
setxattr


msg_msg
msg_msg
next
prev
next (msg_msgseg)
target
value (0xdeadbeef)
next (msg_msgseg)
target
next
prev
next (msg_msgseg)
msg_msgseg
msg_msgseg
msg_queue
Overwrite it !!!
setxattr

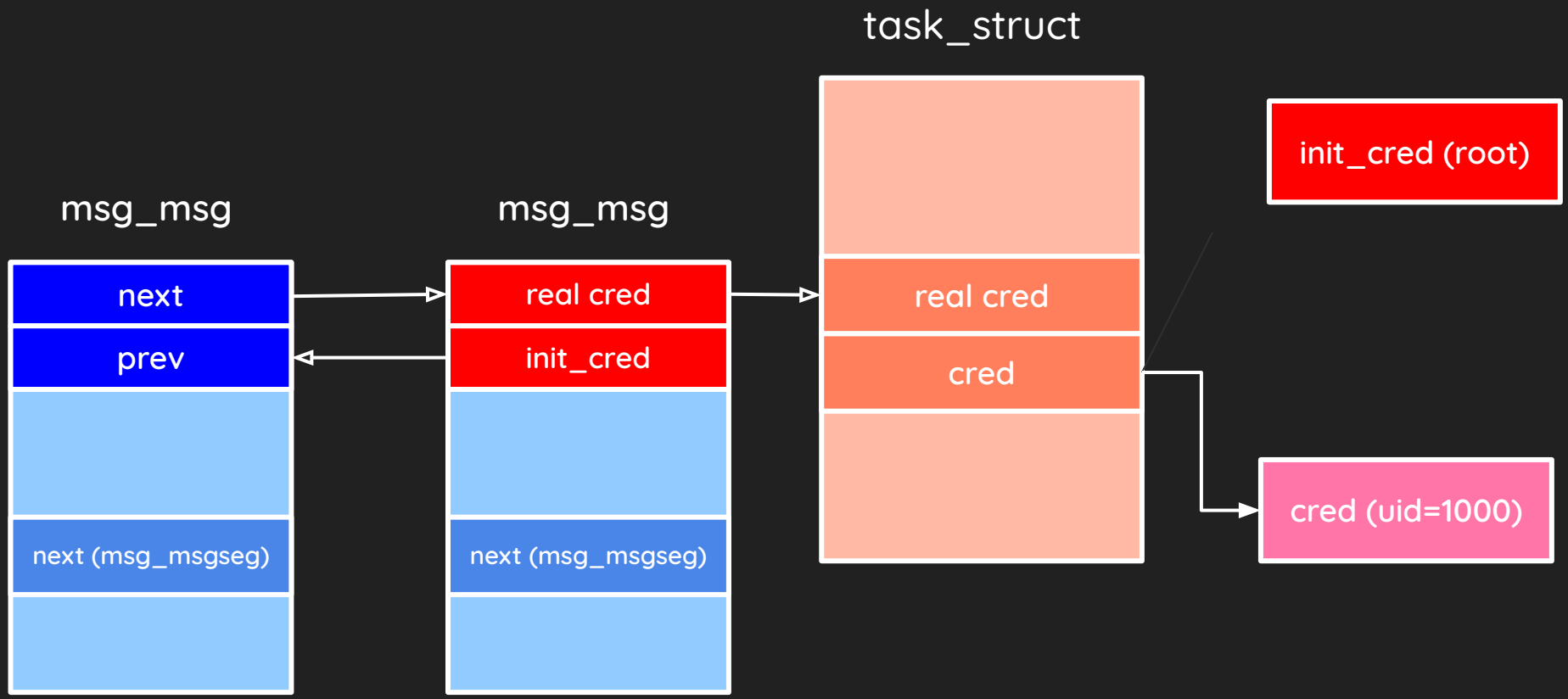

msg_msg
msg_msg
next
prev
next (msg_msgseg)
target
value (0xdeadbeef)
next (msg_msgseg)
AAAAAAAA
AAAAAAAA
+ 0x0
+ 0x8
next
prev
msgrcv()
msg_queue
/include/linux/list.h#L224 (v7.2-rc1)
static inline void __list_del(struct list_head * prev, struct list_head * next)
{
next->prev = prev;
WRITE_ONCE(prev->next, next);
}

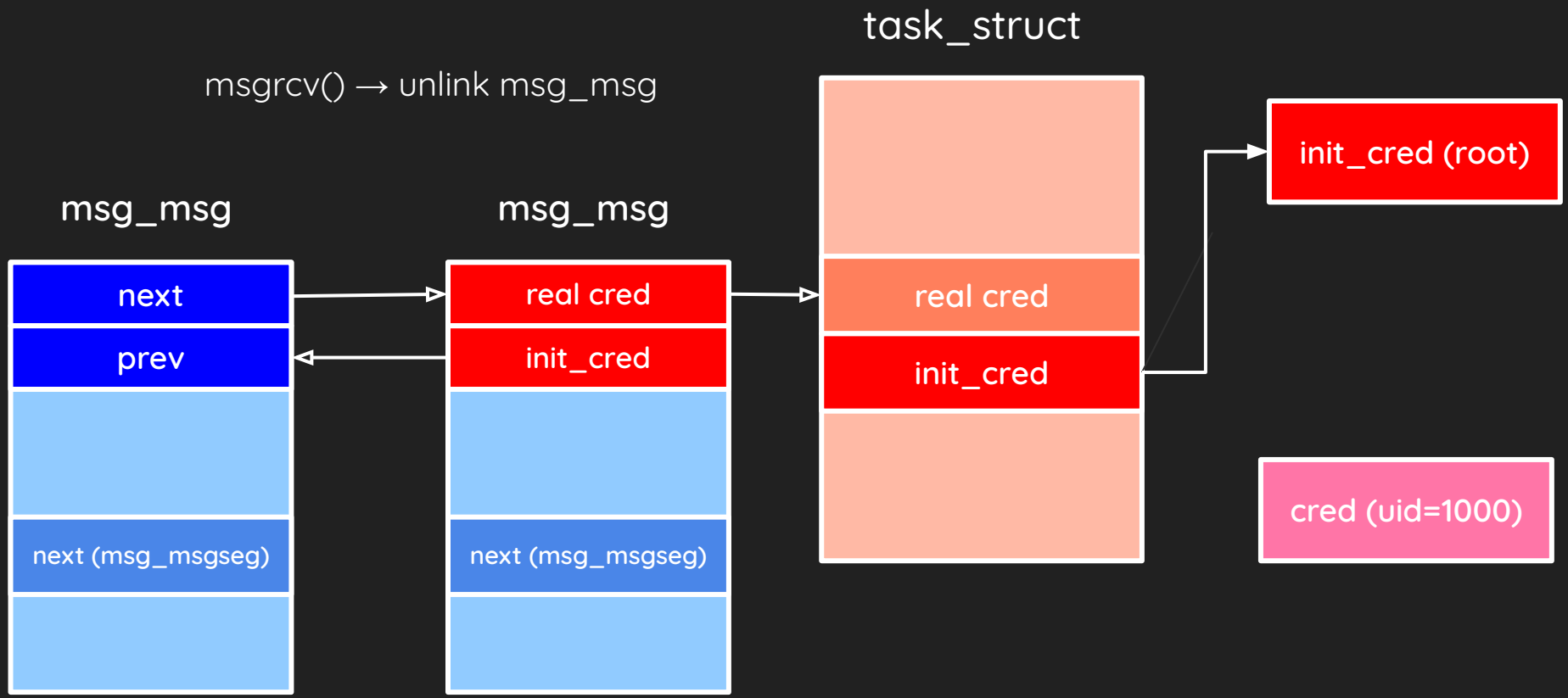

msg_msg
msg_msg
next
prev
next (msg_msgseg)
target
value (0xdeadbeef)
next (msg_msgseg)
AAAAAAAA
0xdeadbeef
+ 0x0
+ 0x8
next
prev
next (msg_msgseg)
Unlink msg_msg struct
msg_queue
/include/linux/list.h#L224 (v7.2-rc1)
next->prev (target + 8) = prev (value);

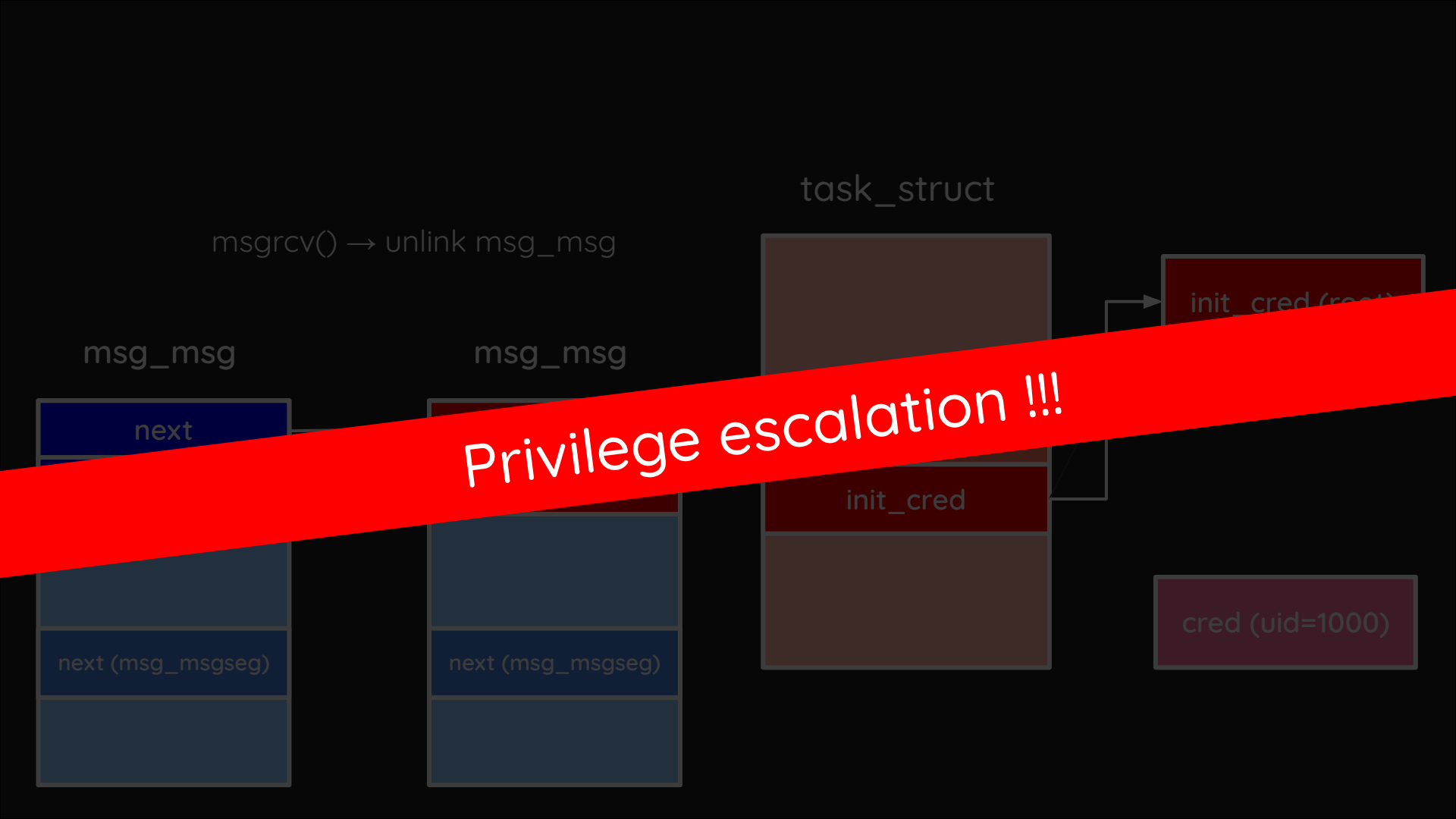

msg_msg
msg_msg
next
prev
next (msg_msgseg)
target
value (0xdeadbeef)
next (msg_msgseg)
AAAAAAAA
0xdeadbeef
+ 0x0
+ 0x8
next
next (msg_msgseg)
Arbitrary Write
msg_queue
/include/linux/list.h#L224 (v7.2-rc1)
next->prev (target + 8) = prev (value);

How to get root ?

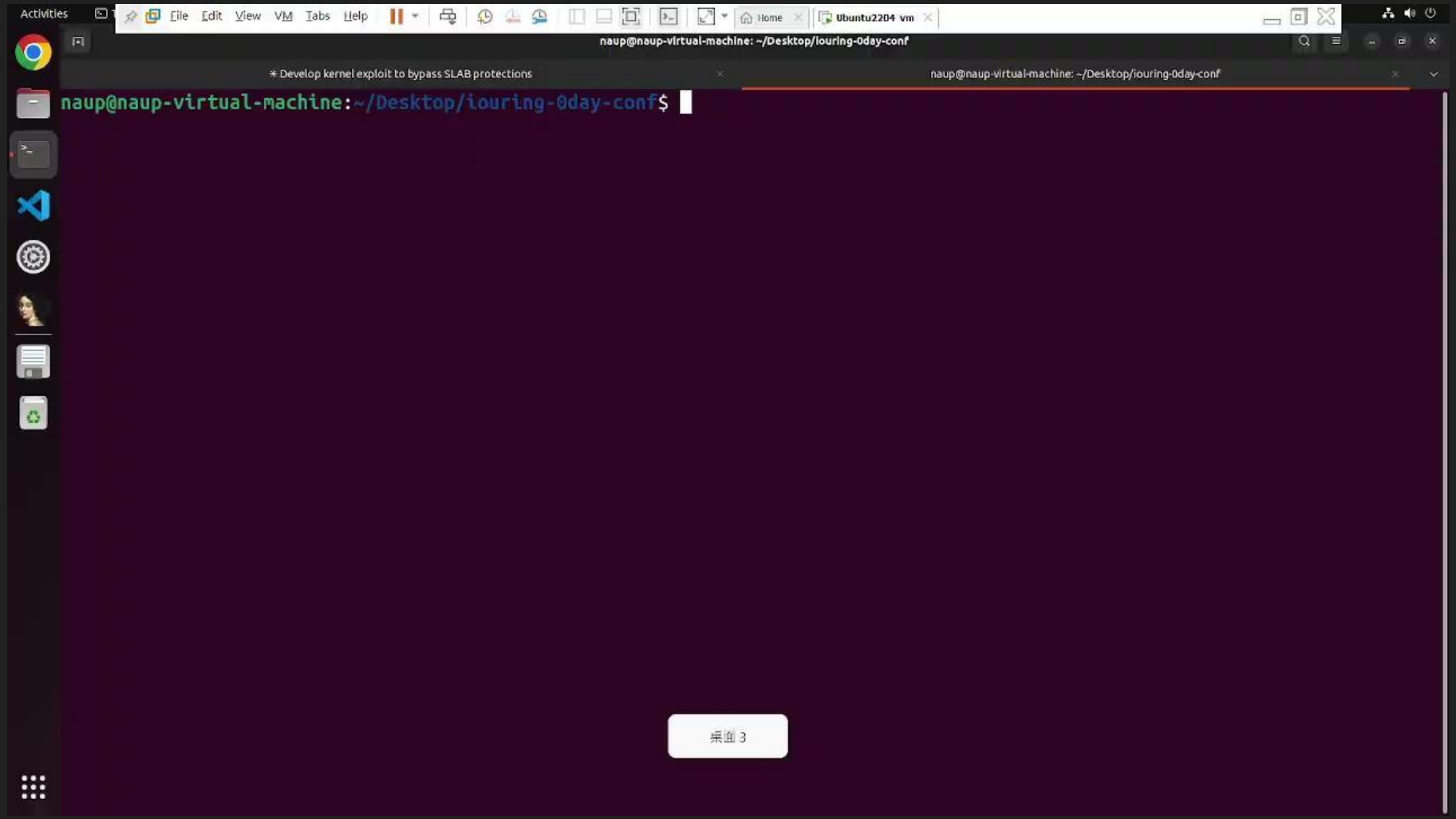
- > Double Free primitive 
- > kmalloc-N object double free 
- > Overwrite task_struct cred pointer to root cred 
  - > Leak KASLR 
  - > Leak task_struct address 
  - > Arbitrary write primitive 


task_struct
init_cred (root)
msg_msg
msg_msg
next
prev
real cred
init_cred
real cred
cred
next (msg_msgseg)
next (msg_msgseg)
cred (uid=1000)


task_struct
msgrcv() → unlink msg_msg
msg_msg
msg_msg
next
prev
next (msg_msgseg)
real cred
init_cred
next (msg_msgseg)
real cred
init_cred
init_cred (root)
cred (uid=1000)


task_struct
msgrcv() → unlink msg_msg
msg_msg
msg_msg
next
next (msg_msgseg)
next (msg_msgseg)
init_cred
cred (uid=1000)
Privilege escalation !!!

Demo Time


Activities
File Edit View VM Tabs Help
Home
Ubuntu2204 vm
naup@naup-virtual-machine: ~/Desktop/iouring-0day-conf
✳ Develop kernel exploit to bypass SLAB protections
naup@naup-virtual-machine: ~/Desktop/iouring-0day-conf
naup@naup-virtual-machine:~/Desktop/iouring-0day-conf$
桌面 3

Patch

```
Fixes: 46800585ae04 ("io_uring/kbuf: validate ring provided buffer addresses with access_ok()")
Signed-off-by: Hao-Yu Yang <naup96721@gmail.com>
---
 io_uring/kbuf.c | 5 +++--
 1 file changed, 3 insertions(+), 2 deletions(-)

diff --git a/io_uring/kbuf.c b/io_uring/kbuf.c
index 3cd29477fff2..b6b969b55e12 100644
--- a/io_uring/kbuf.c
+++ b/io_uring/kbuf.c
@@ -287,8 +287,6 @@ static int io_ring_buffers_peek(struct io_kiocb *req, struct buf_sel_arg *arg,
                iov = kmalloc_objs(struct iovec, nr_avail);
                if (unlikely(!iov))
                        return -ENOMEM;
-               if (arg->mode & KBUF_MODE_FREE)
-                       kfree(arg->iovs);
                arg->iovs = iov;
                nr_iovs = nr_avail;
        } else if (nr_avail < nr_iovs) {
@@ -330,6 +328,9 @@ static int io_ring_buffers_peek(struct io_kiocb *req, struct buf_sel_arg *arg,
                buf = io_ring_head_to_buf(br, ++head, bl->mask);
        } while (--nr_iovs);

+       if (arg->mode & KBUF_MODE_FREE)
+               kfree(arg->iovs);
+
        if (head == tail)
                req->flags |= REQ_F_BL_EMPTY;

--
2.34.1
```


Eat plenty, and grow up big and healthy for me, okay?
たくさん食べて 早く大きく 健康に育ってくださいね


1

io_uring/kbuf: free the replaced iovec after a successful grow ???



```
Fixes: cd053d788c3f ("io_uring: fix dangling iovec after provided-buffer bundle grow failure")
Assisted-by: Codex:gpt-5.3-codex-spark
Signed-off-by: Jaeyeong Lee <iostreampy@proton.me>
---
 io_uring/kbuf.c | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)

diff --git a/io_uring/kbuf.c b/io_uring/kbuf.c
index b6b969b55e12..de0129bceaba 100644
--- a/io_uring/kbuf.c
+++ b/io_uring/kbuf.c
@@ -328,8 +328,8 @@ static int io_ring_buffers_peek(struct io_kiocb *req, struct buf_sel_arg *arg,
                buf = io_ring_head_to_buf(br, ++head, bl->mask);
        } while (--nr_iovs);

-       if (arg->mode & KBUF_MODE_FREE)
-               kfree(arg->iovs);
+       if (arg->iovs != org_iovs && (arg->mode & KBUF_MODE_FREE))
+               kfree(org_iovs);

        if (head == tail)
                req->flags |= REQ_F_BL_EMPTY;
--
2.43.0
```

io_uring/kbuf: free the replaced iovec after a successful grow ???



```
Fixes: cd053d788c3f ("io_uring: fix dangling iovec after provided-buffer bundle grow failure")
Assisted-by: Codex:gpt-5.3-codex-spark
Signed-off-by: Jaeyeong Lee <iostreampy@proton.me>
---
 io_uring/kbuf.c | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)

diff --git a/io_uring/kbuf.c b/io_uring/
index b6b969b55e12,
--- a/i
                                        io_kiocb *req, struct buf_sel_arg *arg,
                                        (br, ++head, bl->mask);
                        _iovs);

-       if (arg->mode & KBUF_MODE_FREE)
-               kfree(arg->iovs);
+       if (arg->iovs != org_iovs && (arg->mode & KBUF_MODE_FREE))
+               kfree(org_iovs);

        if (head == tail)
                req->flags |= REQ_F_BL_EMPTY;
--
2.43.0
```

~~Supply Chain Attack~~


蓳姬 Naup @naup96721 · Jul 5
I found a linux kernel 0 day and get root shell :). This is the first time I successfully achieved LPE. I reported it to the Linux Kernel Security Team today. When it has been patched, I'll publish the full research record on my blog.
~ $ id
uid=1000 gid=1000 groups=1000
~ $ cat flag.txt
cat: can't open 'flag.txt': Permission denied
~ $ /tmp/e
[+]
[+] get root shell, pwned by naup96321 uid=0
~ # id
uid=0(root) gid=0(root) groups=1000
~ # cat flag.txt
flag{pwned_by_naup :)}
49
144
2.6K
97K





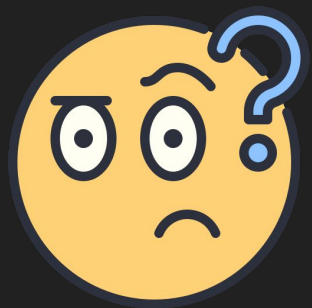
49 144 2.6K 97K


io_uring
vulnerability
exploit chain 1
get root !
report & patch

The End ?

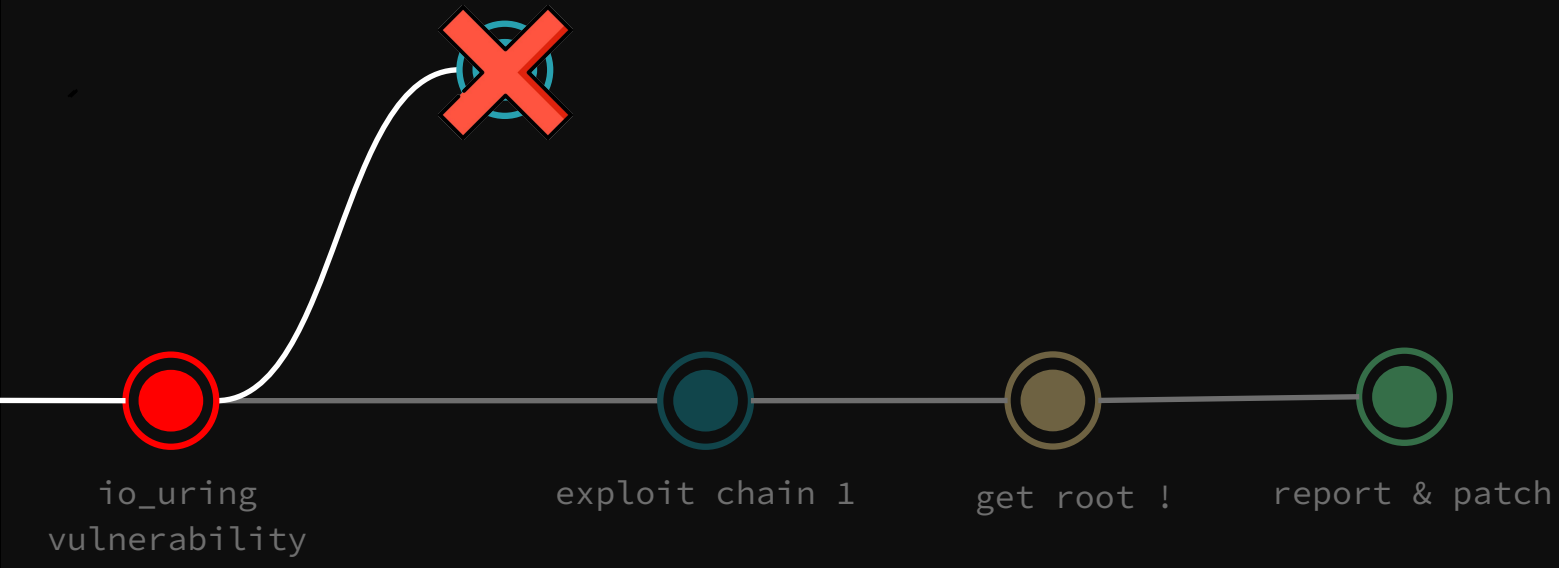
Although this exploit chain may seem simple in the end

But when I was exploiting this vulnerability, many strange things happened, which made me very confused.

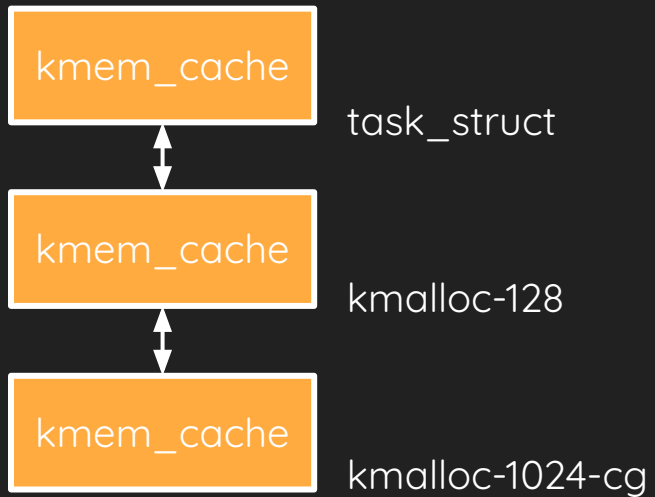


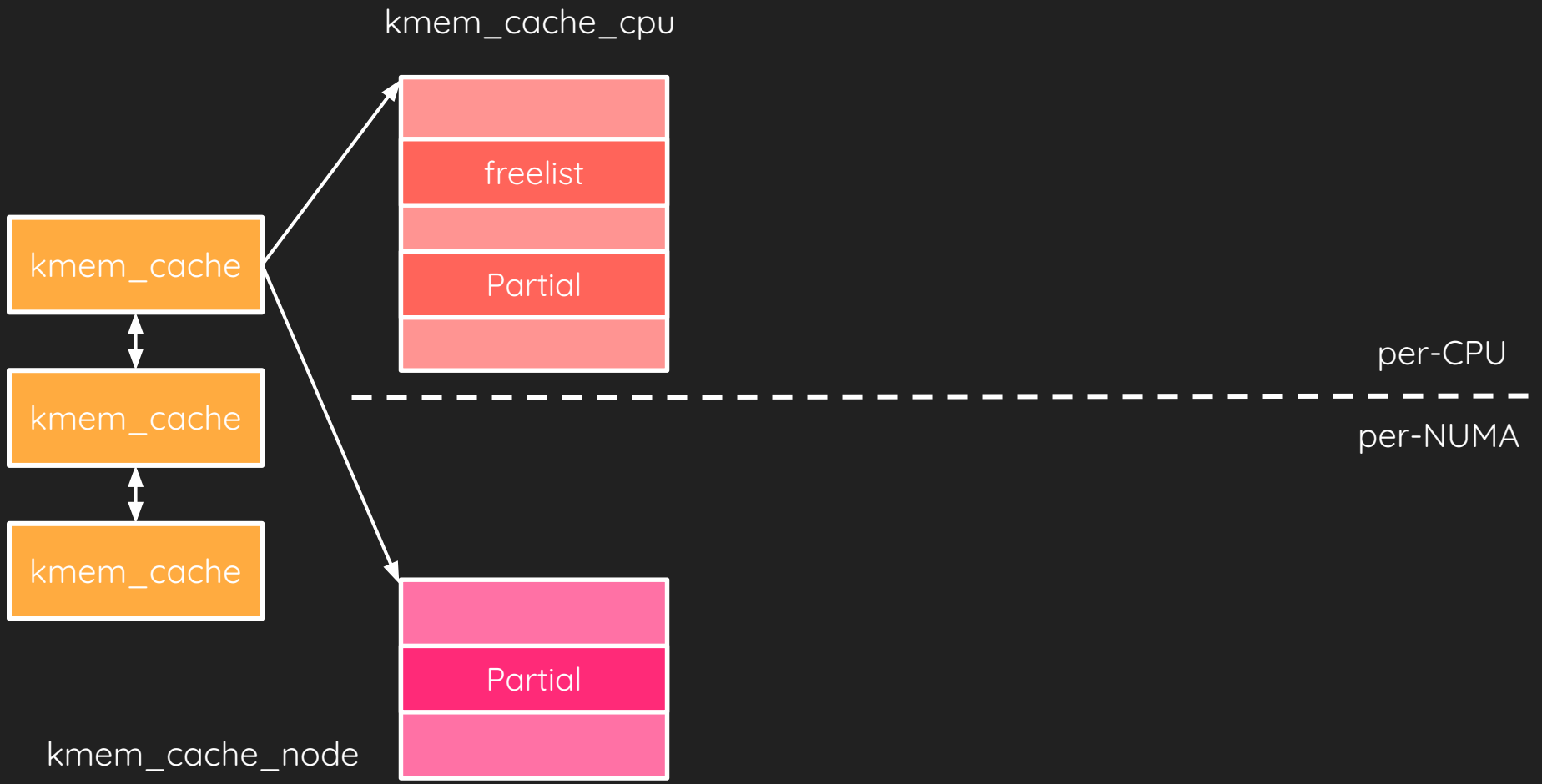
My Failure Experience

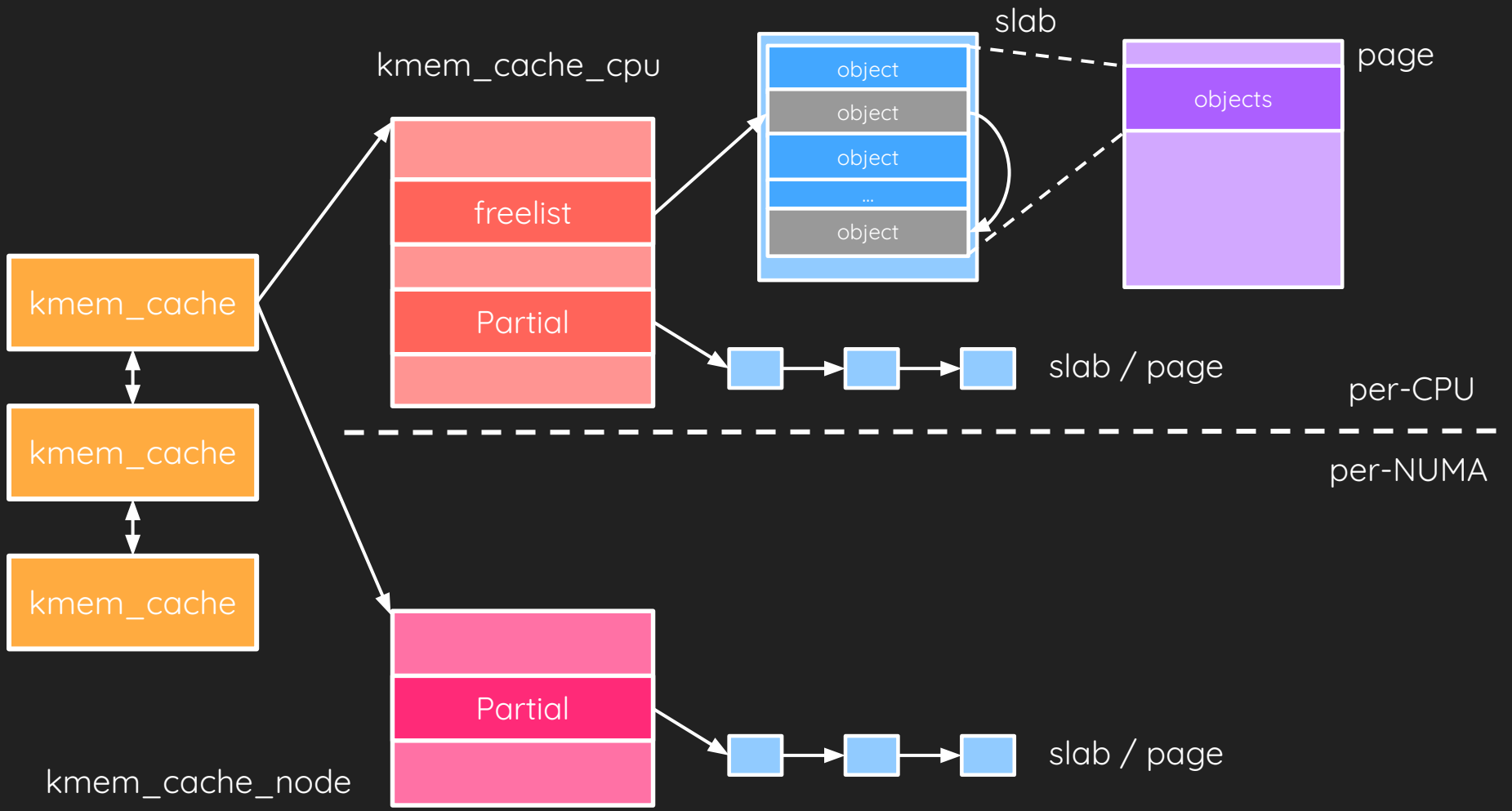
Can Cross-Cache provide a path to LPE or not ?



Many memory pools allocate the specified object



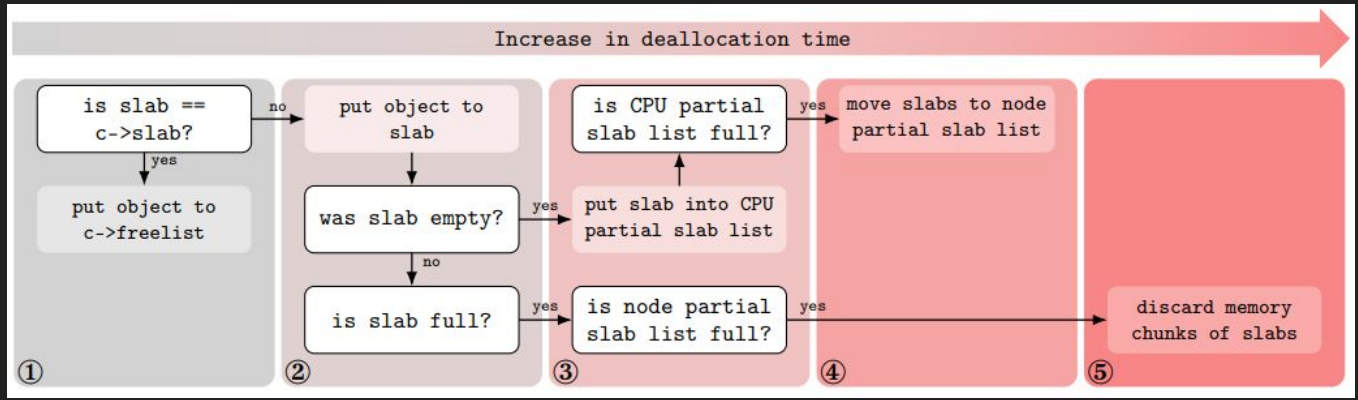

kmem_cache_cpu
freelist
Partial
kmem_cache
kmem_cache
kmem_cache
per-CPU
per-NUMA
Partial
kmem_cache_node


slab
page
kmem_cache_cpu
object
object
object
...
object
objects
freelist
Partial
kmem_cache
kmem_cache
kmem_cache
slab / page
per-CPU
per-NUMA
Partial
slab / page
kmem_cache_node

Cross-Cache attack

> If we only have a UAF in one kmem_cache object but don't have any useful objects to exploit

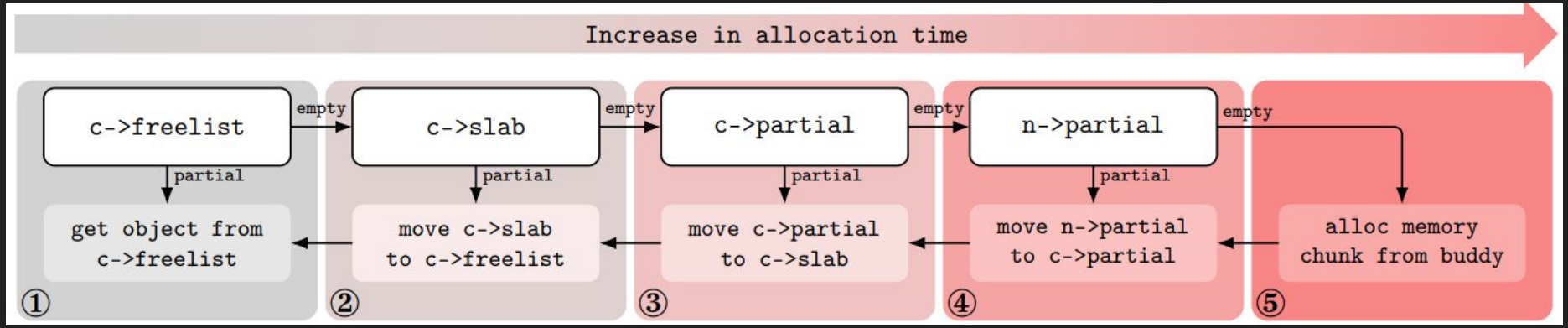
> Try to make the UAF object cross into other memory pools



https://www.usenix.org/system/files/usenixsecurity24-maar-slubstick.pdf

Cross-Cache attack

> Fill the node's partial freelist so that the Buddy System reclaims the page containing the UAF object, allowing another kmem_cache to reuse that page



https://www.usenix.org/system/files/usenixsecurity24-maar-slubstick.pdf

My Original Idea

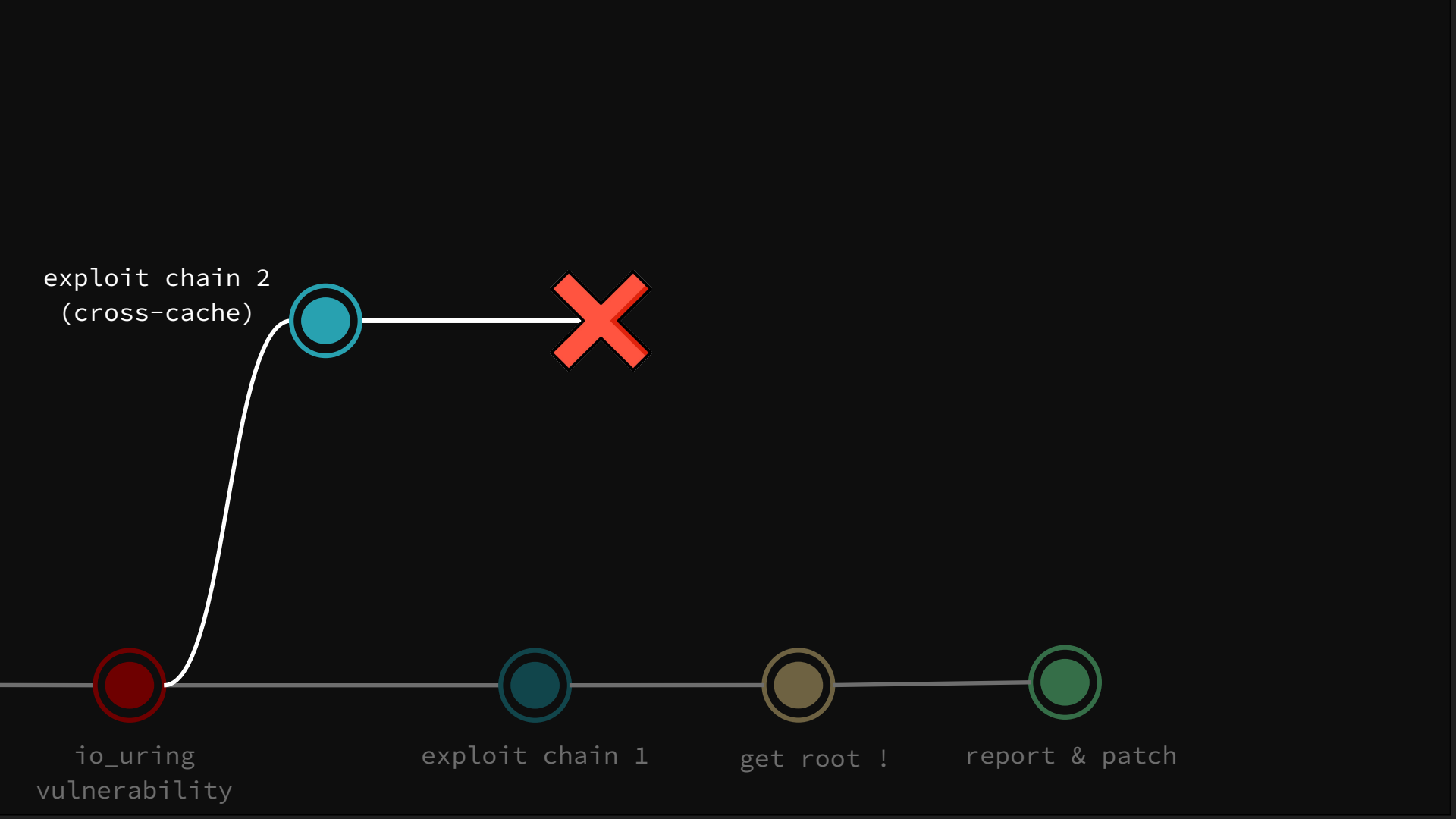
> Cross-cache the kmalloc-N UAF object into cred_jar, then overwrite the cred's UID/GID to obtain a root shell

My Original Idea

> Cross-cache the kmalloc-N UAF object into cred_jar, then overwrite the cred's UID/GID to obtain a root shell

> But ... I couldn't get the page containing the UAF object to be reclaimed by the Buddy System.




exploit chain 2
(cross-cache)
io_uring
vulnerability
exploit chain 1
get root !
report & patch

The Latest Sheaves Work To Hopefully Improve Linux Performance

Written by Michael Larabel in Linux Kernel on 24 October 2025 at 01:49 PM EDT. 7 Comments

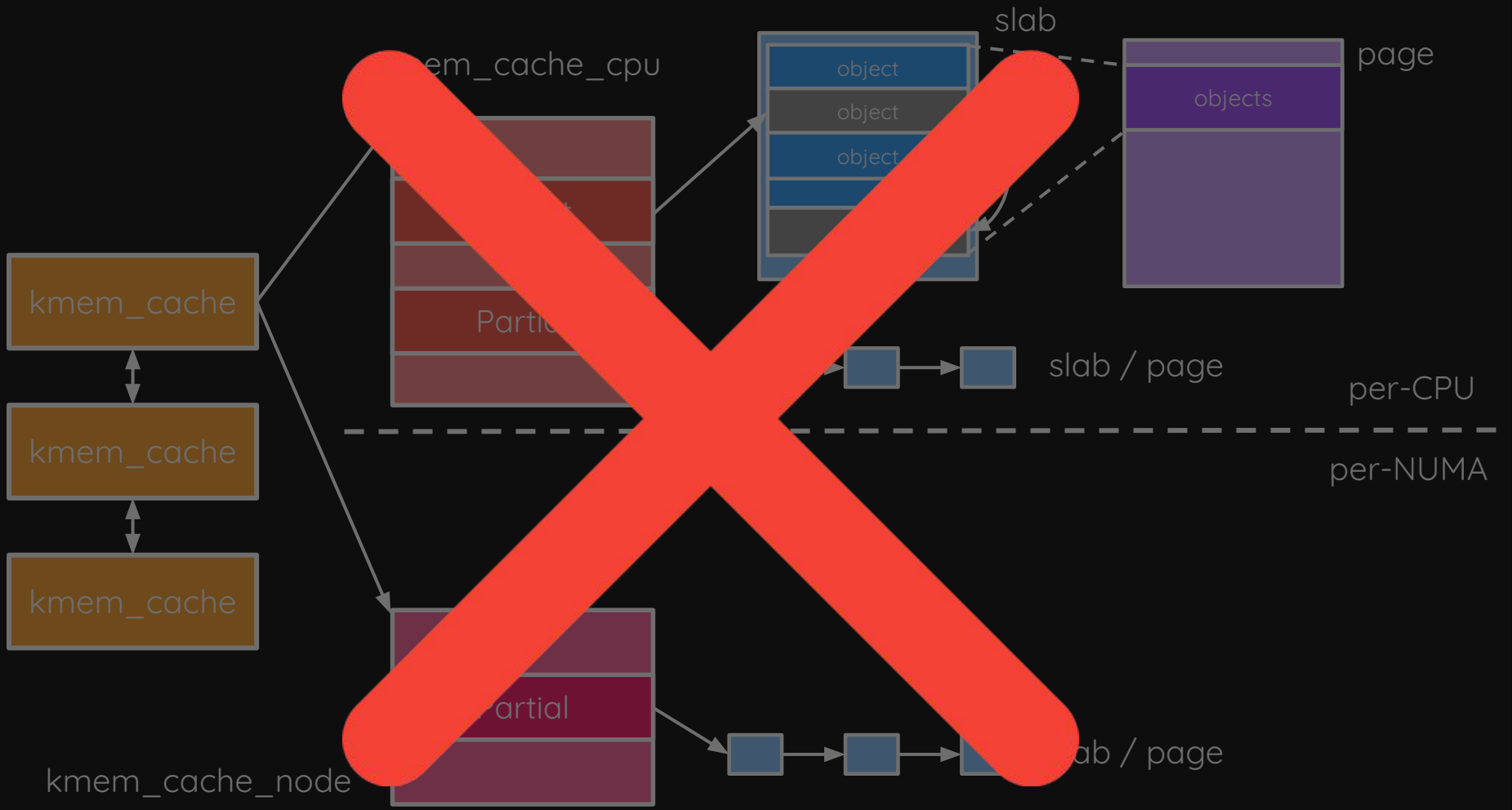


Merged for Linux 6.18 was a new feature called Sheaves as an opt-in, per-CPU array-based caching layer. Plus there is a per-NUMA-node cache of Sheaves called a "Barn". In continuing to build out the Linux kernel usage of Sheaves, a set of initial patches were posted this week to replace the CPU slabs with Sheaves within the slub allocator code.

Vlastimil Babka of SUSE explained within the new patch series entitled slab: replace cpu (partial) slabs with sheaves:

> ***"Percpu sheaves caching was introduced as opt-in but the goal was to eventually move all caches to them. This is the next step, enabling sheaves for all caches (except the two bootstrap ones) and then removing the per cpu (partial) slabs and lots of associated code.***
>
> ***Besides (hopefully) improved performance, this removes the rather complicated code related to the lockless fastpaths (using this_cpu_try_cmpxchg128/64) and its complications with PREEMPT_RT or kmalloc_nolock().***

https://www.phoronix.com/news/Sheaves-Replace-CPU-Slabs

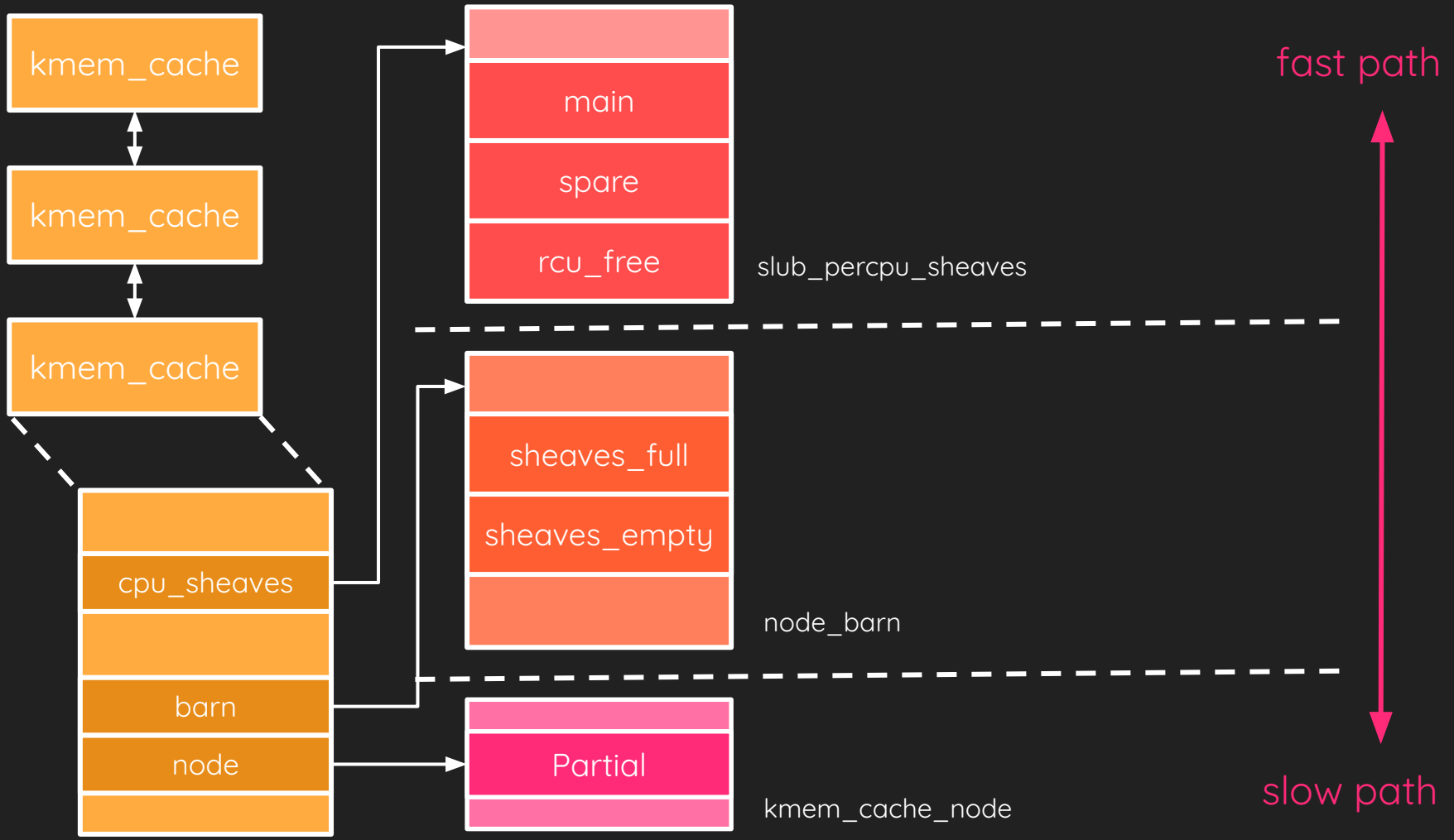

slab
em_cache_cpu
object
object
object
page
objects
kmem_cache
Partia
kmem_cache
slab / page
per-CPU
kmem_cache
per-NUMA
artial
ab / page
kmem_cache_node

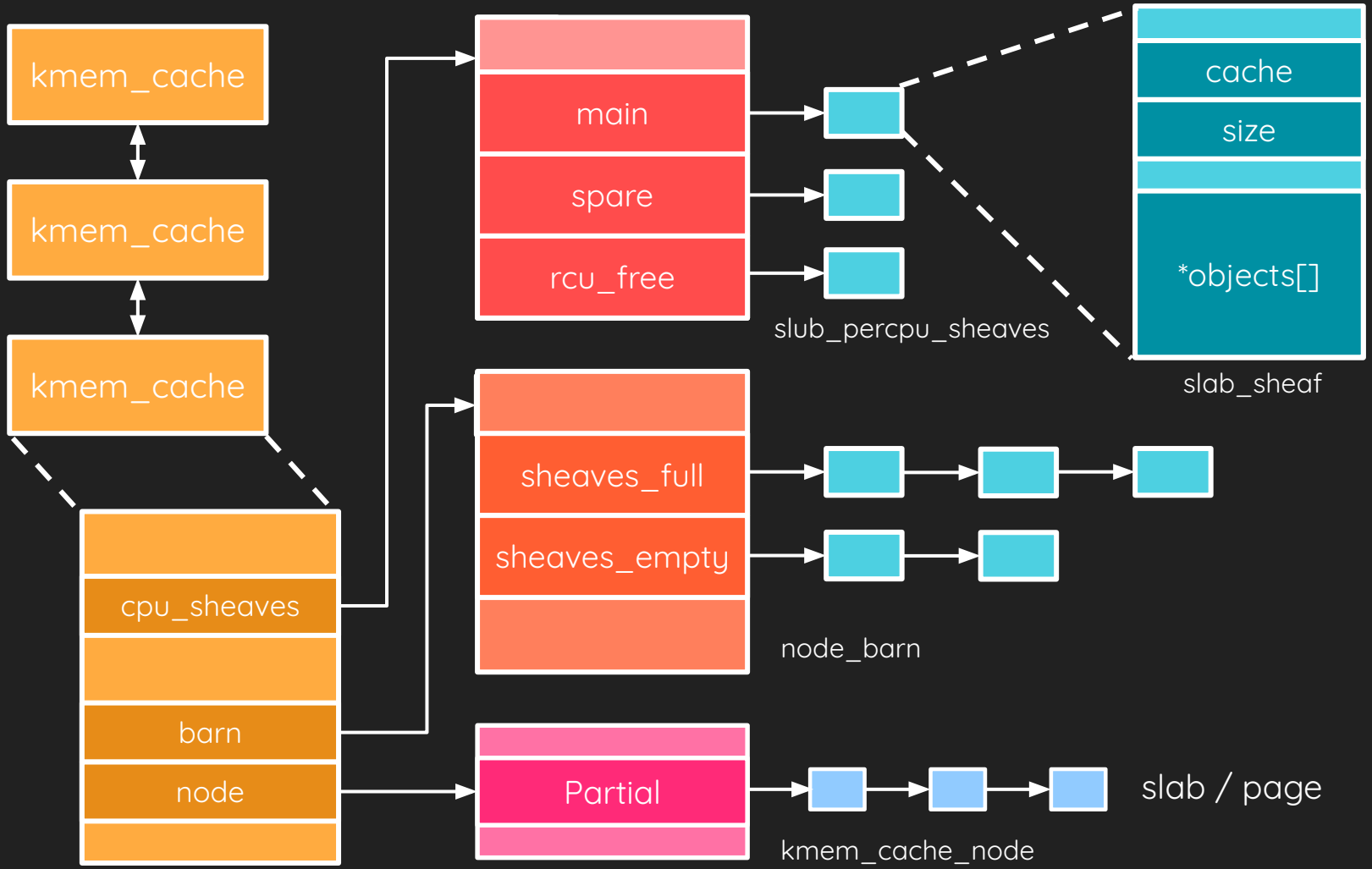
Sheaf / barn

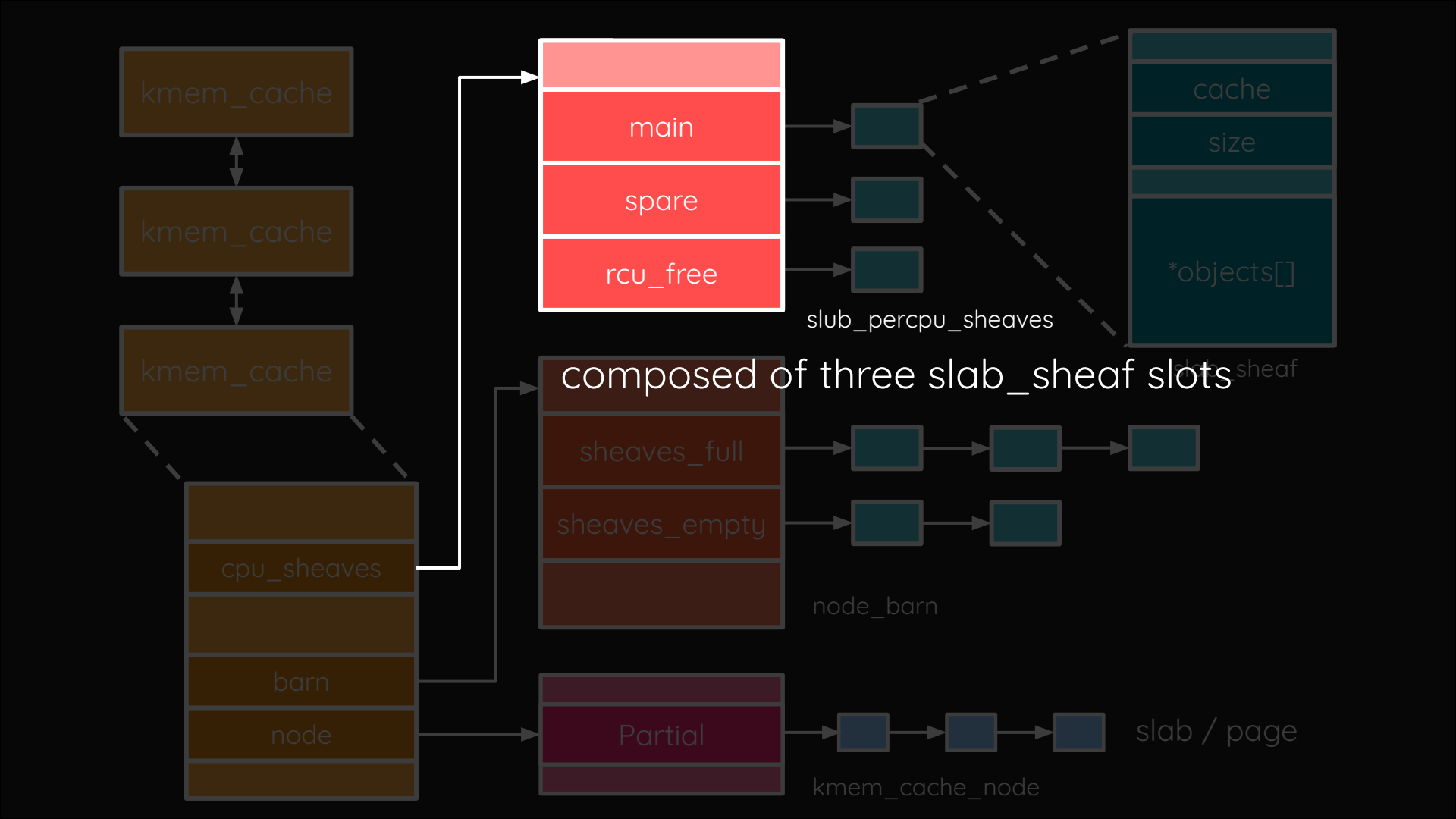


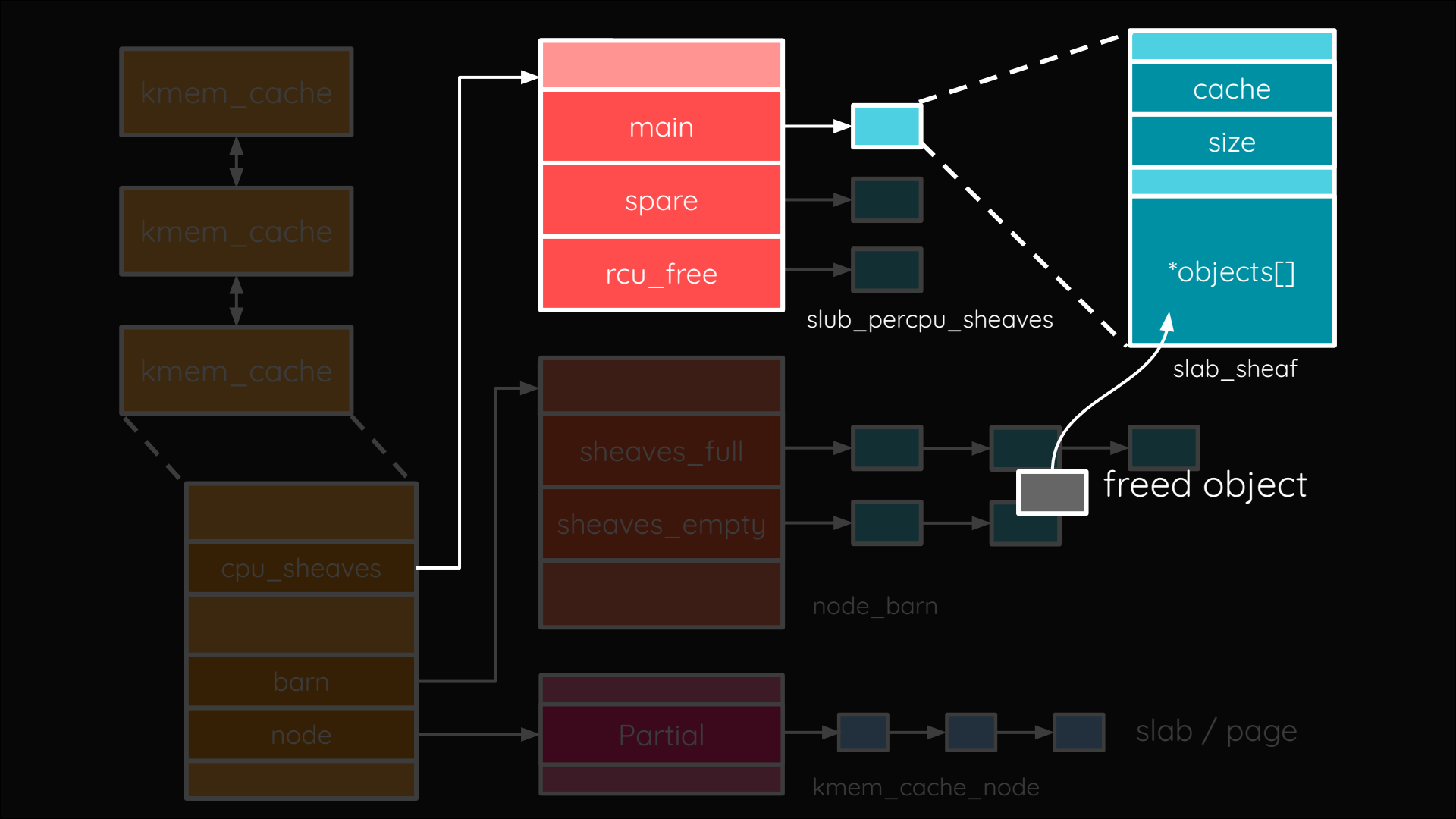
Overview sheaf / barn

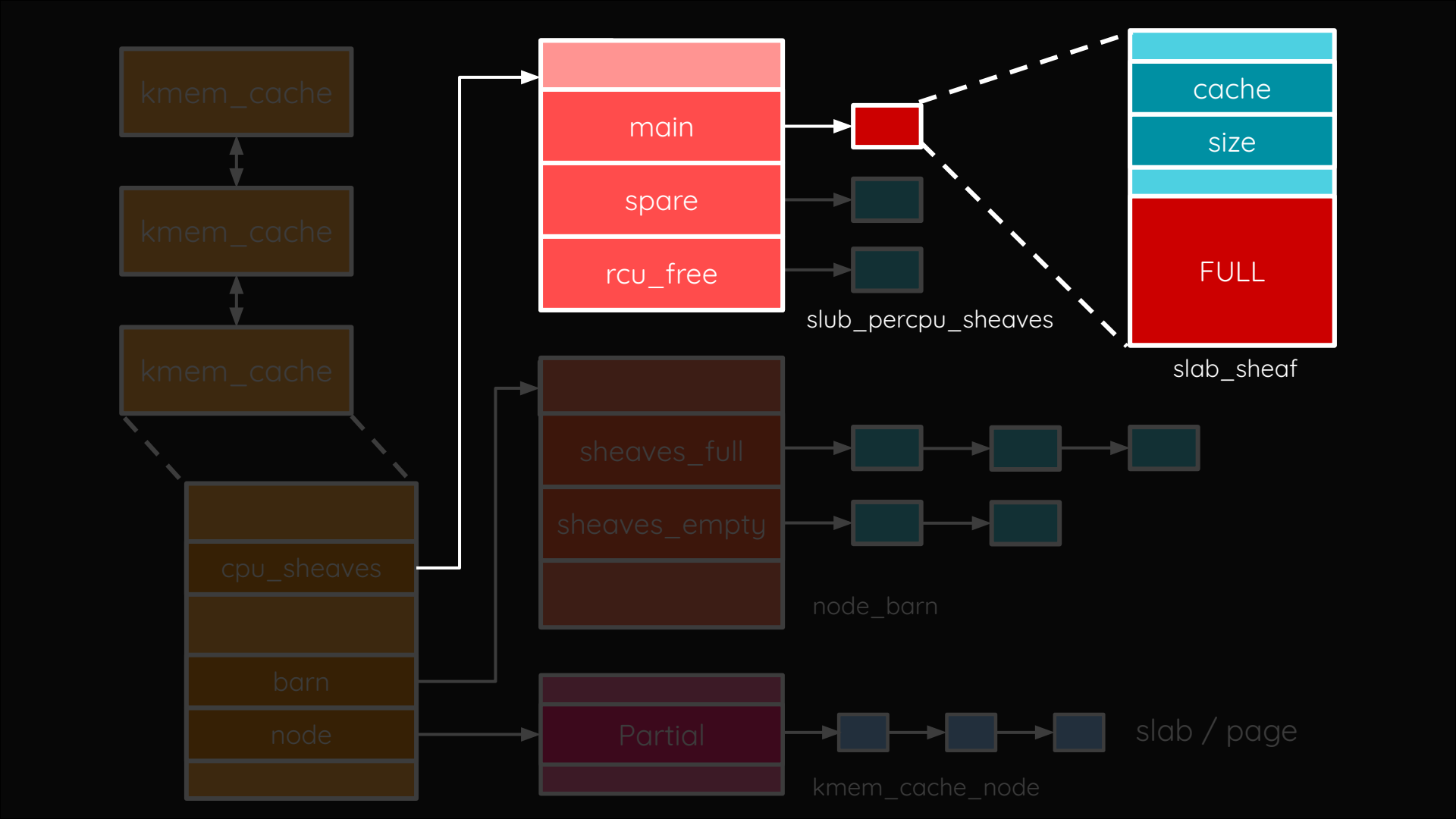
- Introduced in Linux 6.18 (2025)
- Default kernel in Ubuntu 26.04, Fedora 44 or other distros
- Two caching layers built on top of SLUB
  - Sheaf : Per-CPU array-based cache
  - Barn : Per-NUMA-node shared cache
- Optimize VMA locking and Maple Tree operations

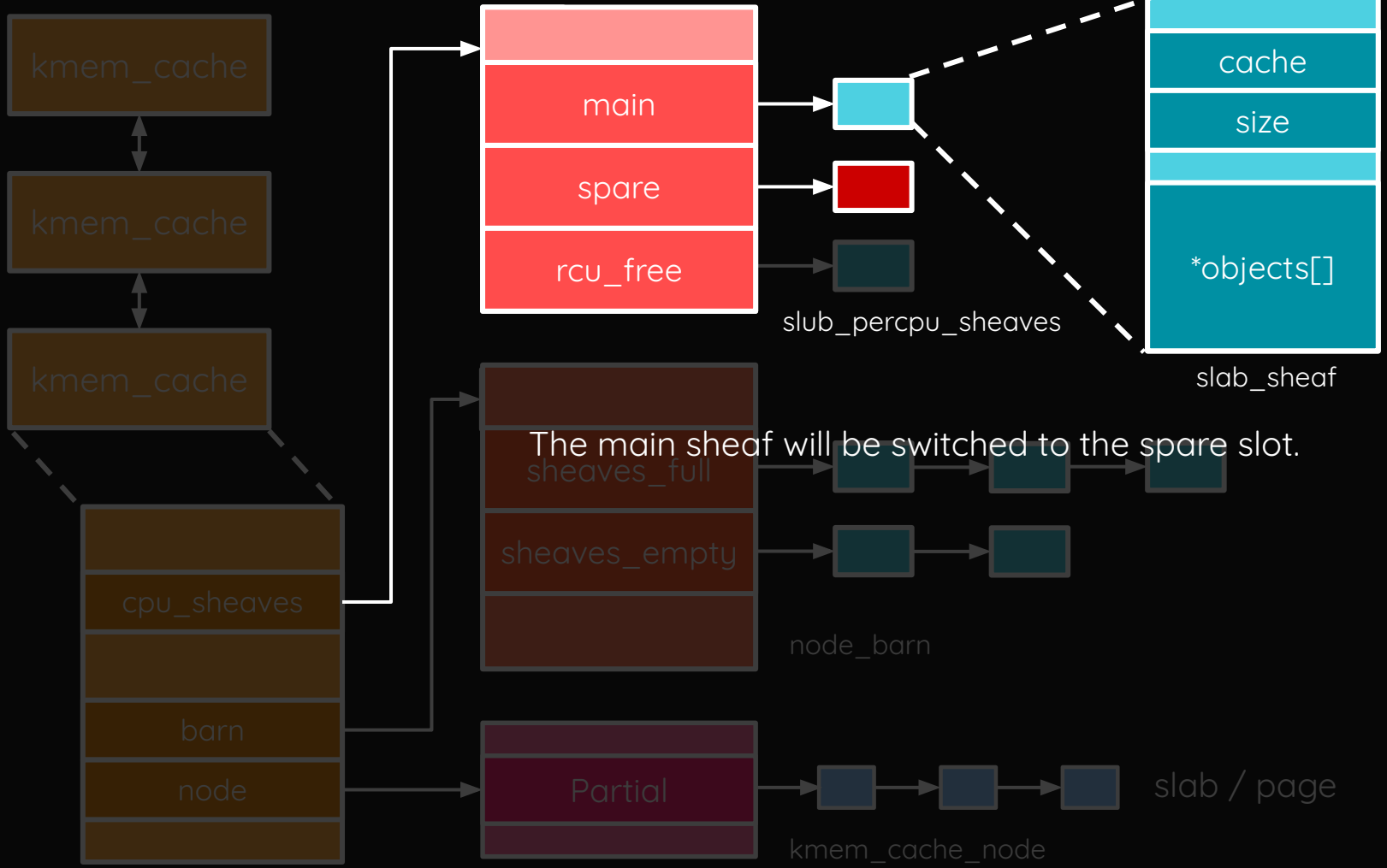

kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_sheaves
sheaves_full
sheaves_empty
node_barn
Partial
kmem_cache_node
fast path
slow path

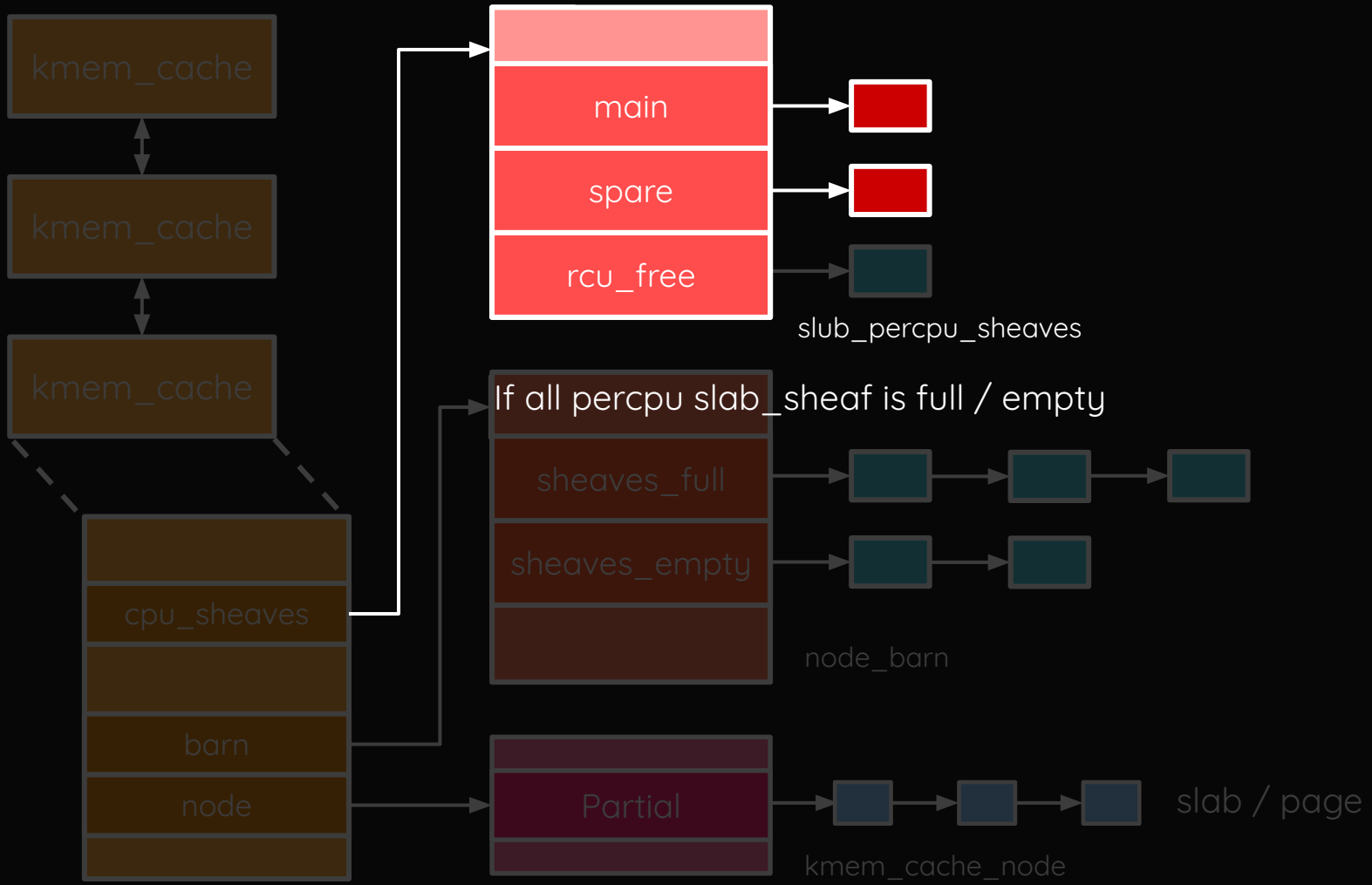

kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_sheaves
cache
size
*objects[]
slab_sheaf
sheaves_full
sheaves_empty
node_barn
Partial
kmem_cache_node
slab / page

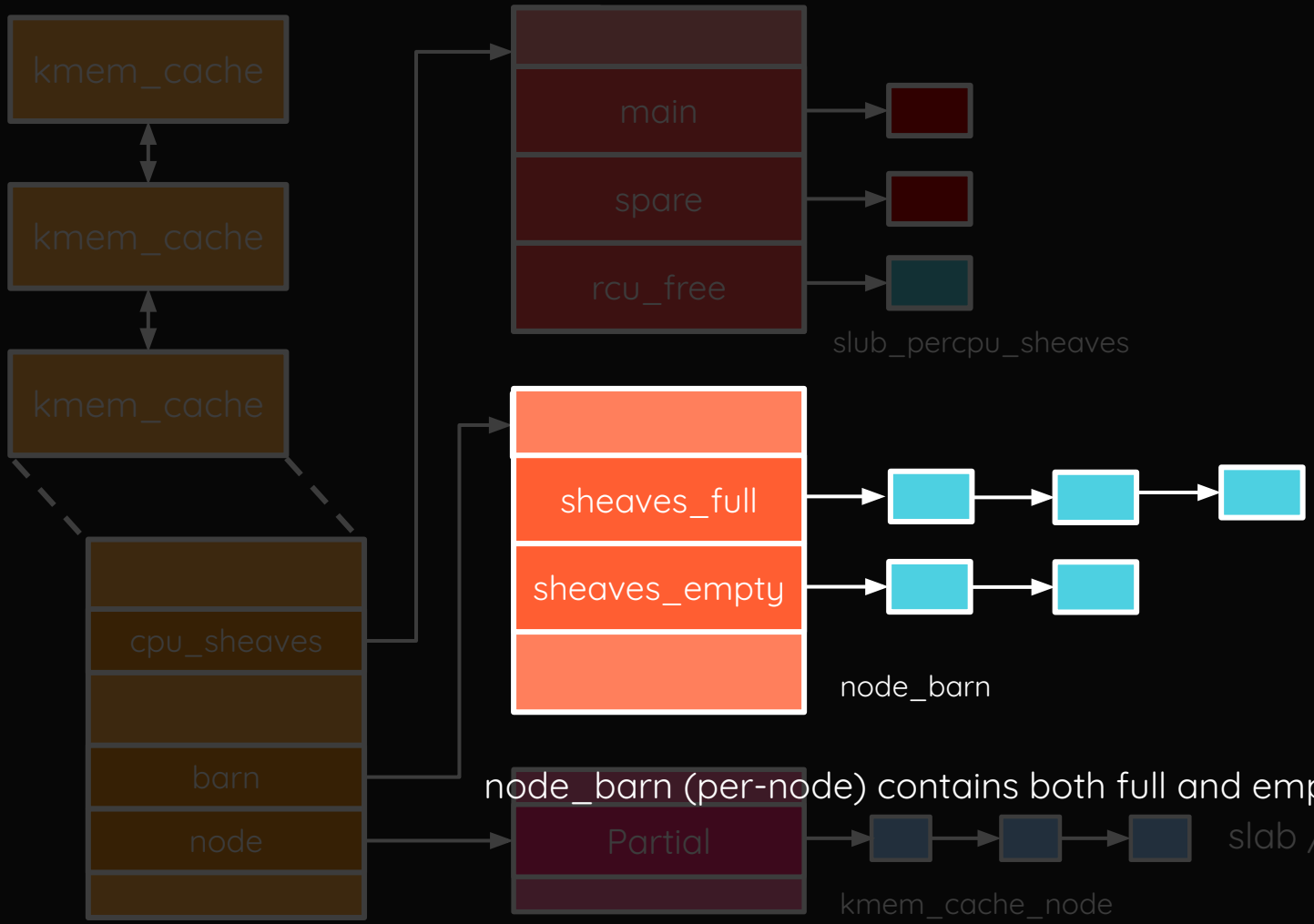

kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_sheaves
cache
size
*objects[]
composed of three slab_sheaf slots
sheaves_full
sheaves_empty
node_barn
Partial
kmem_cache_node
slab / page

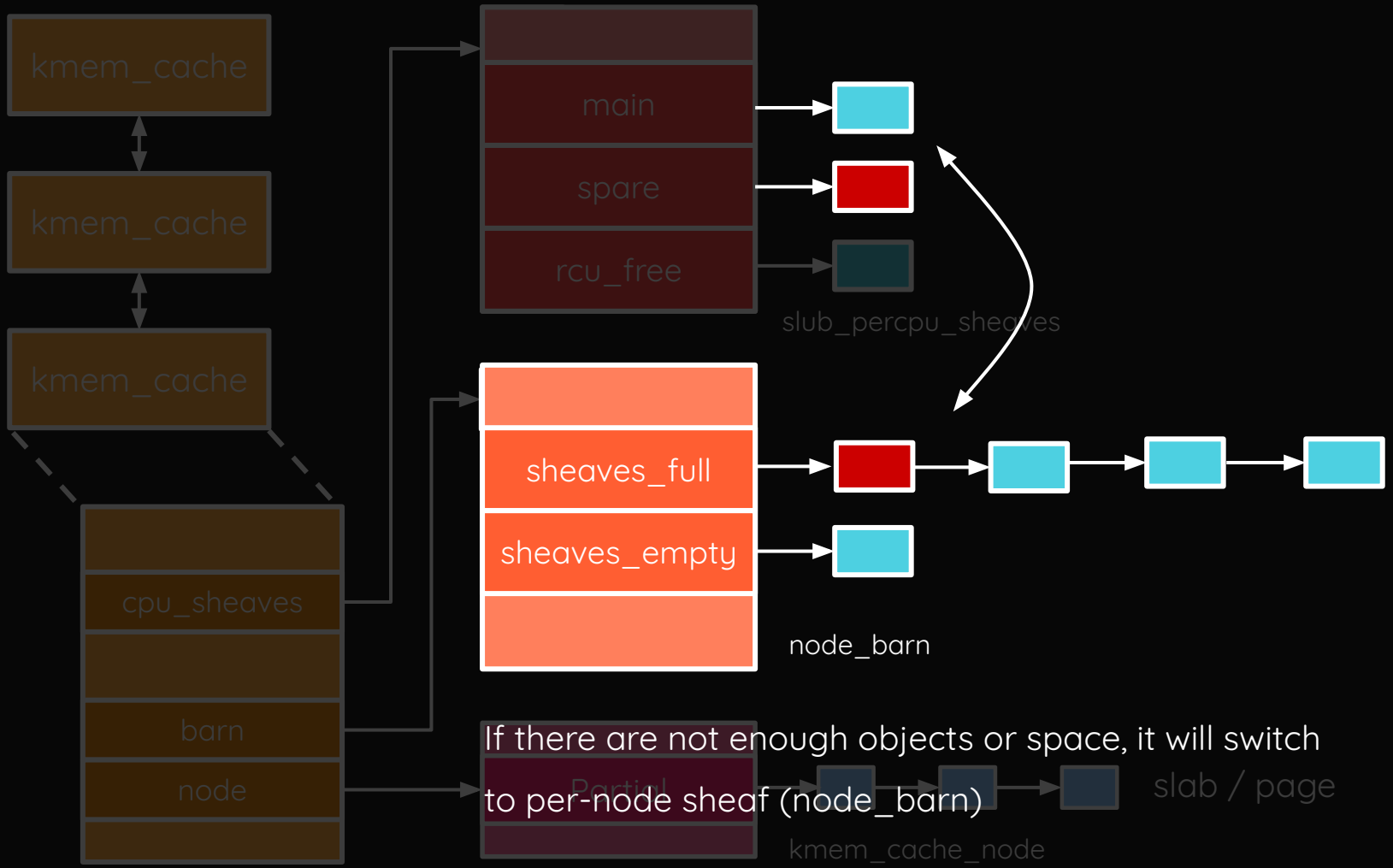

kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_sheaves
cache
size
*objects[]
slab_sheaf
sheaves_full
sheaves_empty
node_barn
freed object
Partial
kmem_cache_node
slab / page


kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_sheaves
cache
size
FULL
slab_sheaf
sheaves_full
sheaves_empty
node_barn
Partial
kmem_cache_node
slab / page


kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_sheaves
cache
size
*objects[]
slab_sheaf
The main sheaf will be switched to the spare slot.
sheaves_full
sheaves_empty
node_barn
Partial
kmem_cache_node
slab / page


kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_sheaves
If all percpu slab_sheaf is full / empty
sheaves_full
sheaves_empty
node_barn
Partial
kmem_cache_node
slab / page


kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_sheaves
sheaves_full
sheaves_empty
node_barn
node_barn (per-node) contains both full and empty slab_sheaf
Partial
slab / page
kmem_cache_node


kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_sheaves
sheaves_full
sheaves_empty
node_barn
If there are not enough objects or space, it will switch to per-node sheaf (node_barn)
Partial
kmem_cache_node
slab / page

Why use sheaf / barn

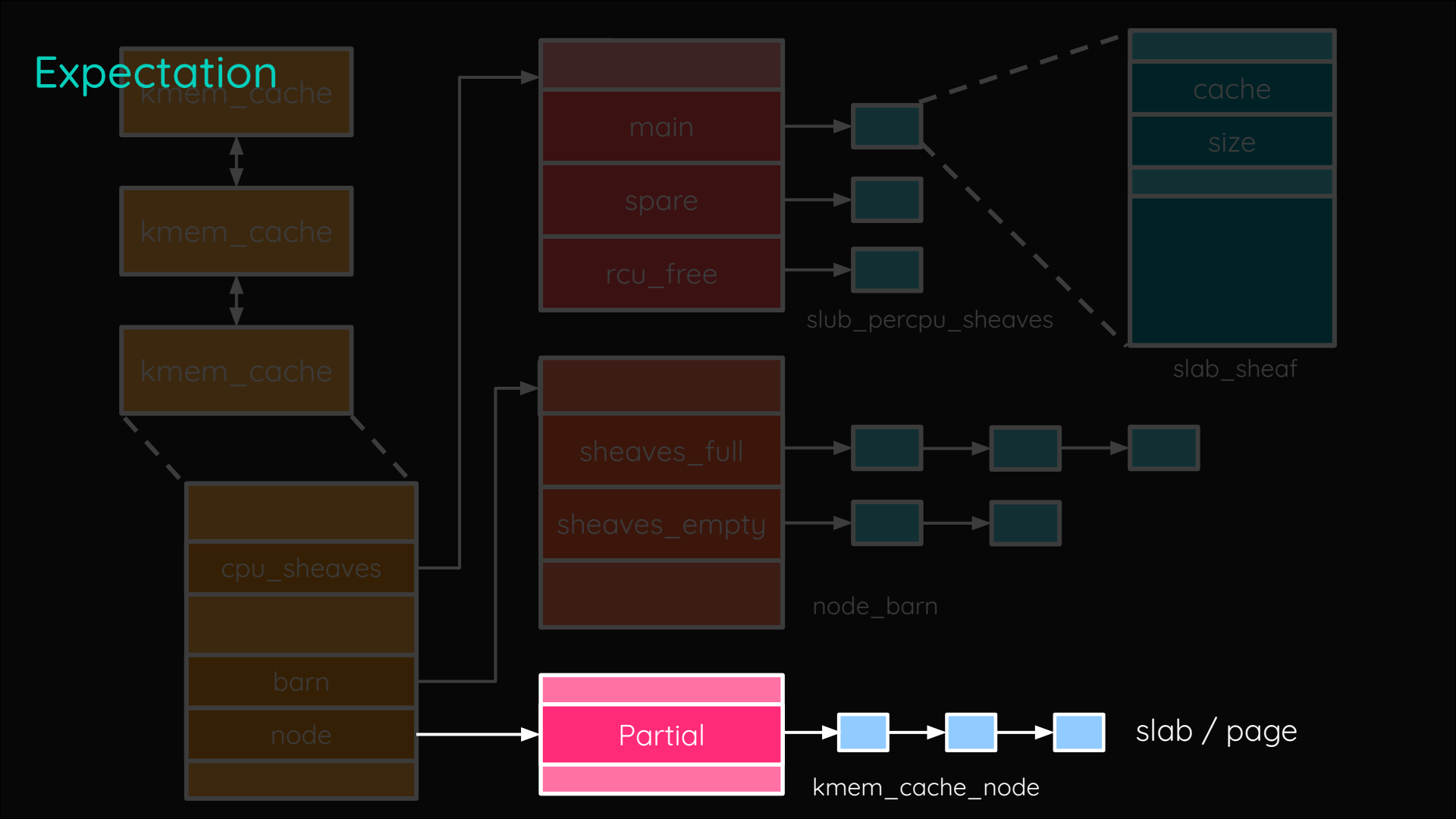
> Batched exchange design

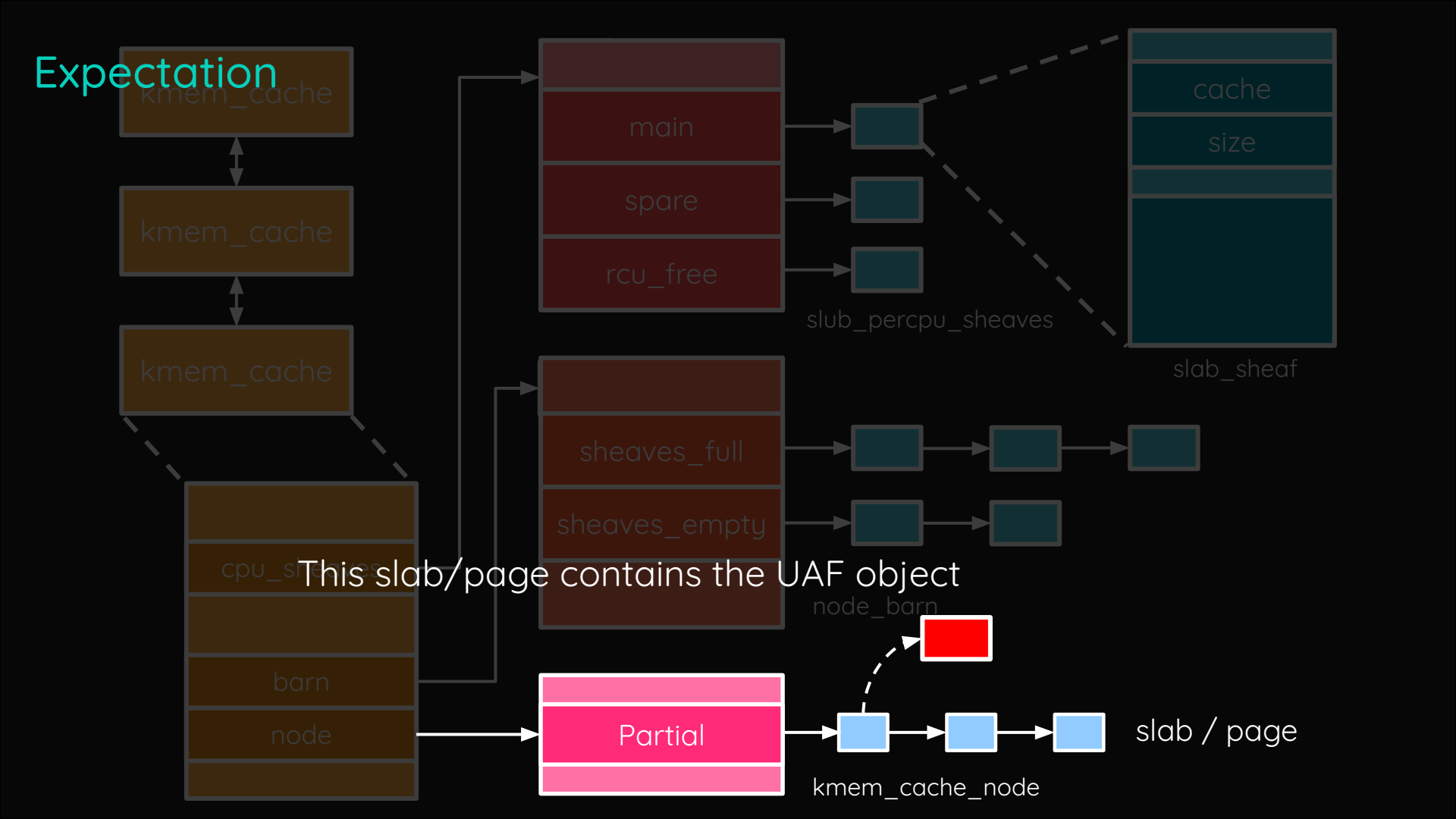
> Improve CPU locality

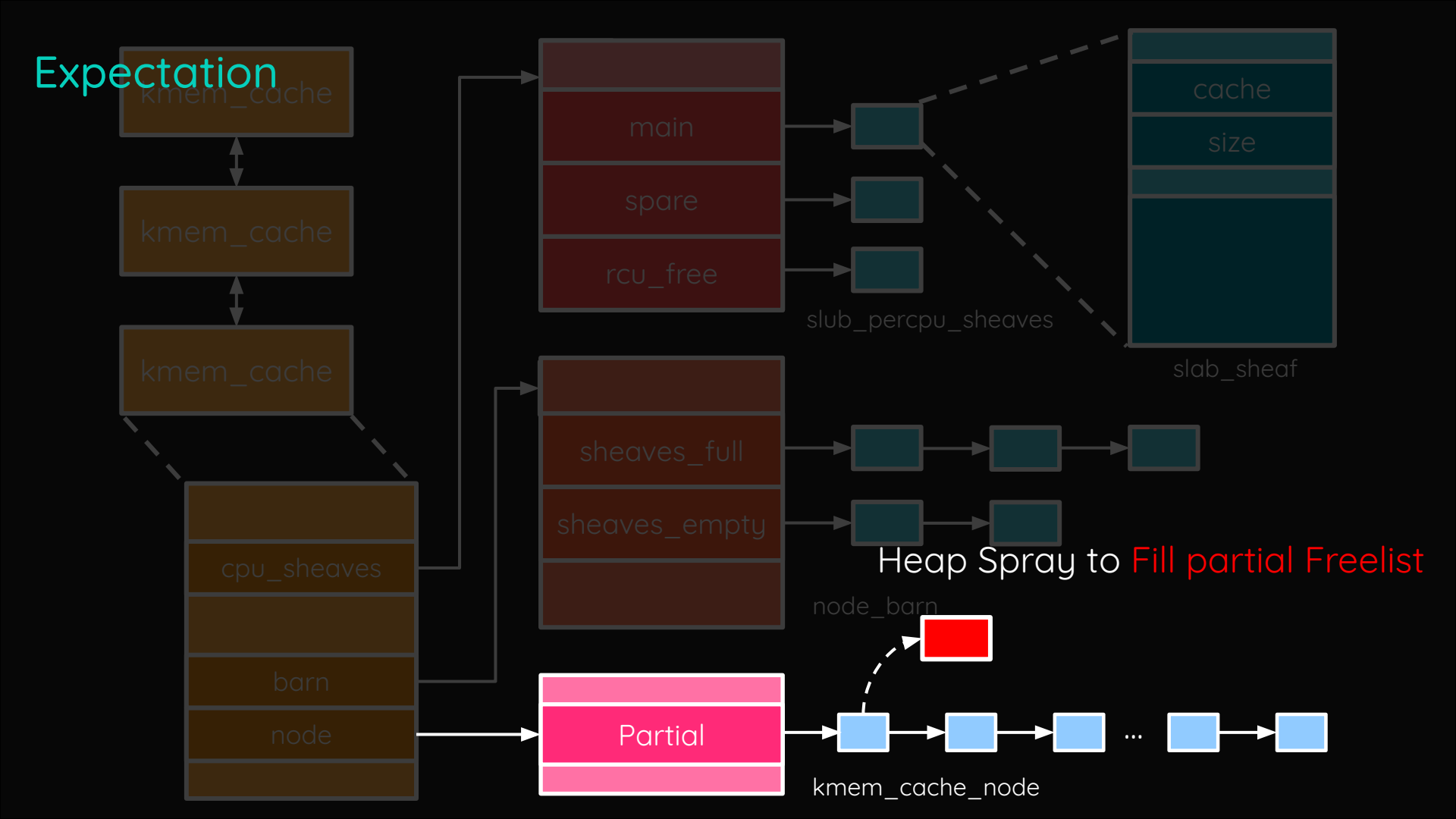
> Avoiding frequent lock contention (VMA locking and Maple Tree)

Fast !!!

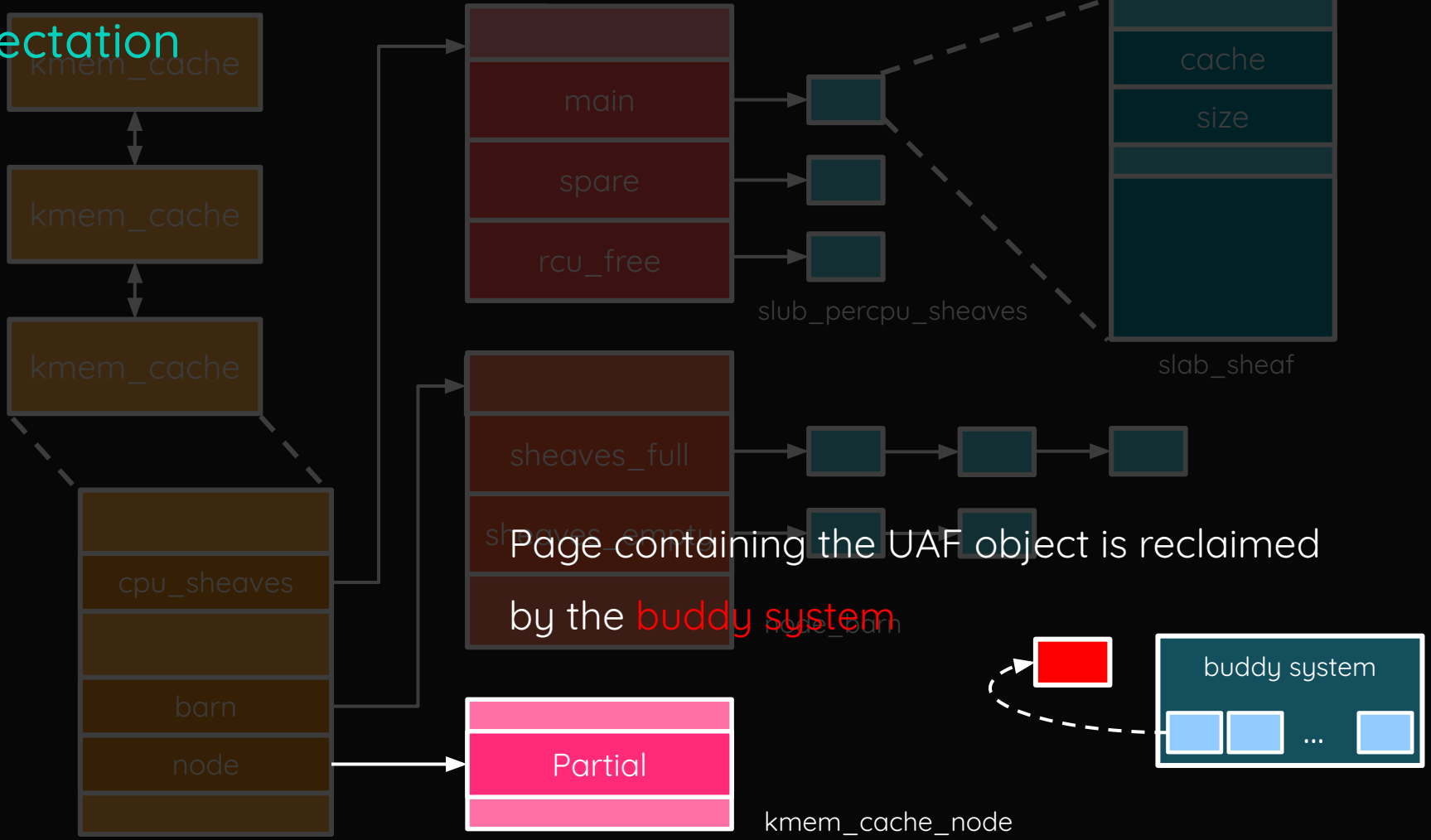
Why cross-cache failed ?


Expectation
kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_sheaves
cache
size
slab_sheaf
sheaves_full
sheaves_empty
node_barn
Partial
kmem_cache_node
slab / page

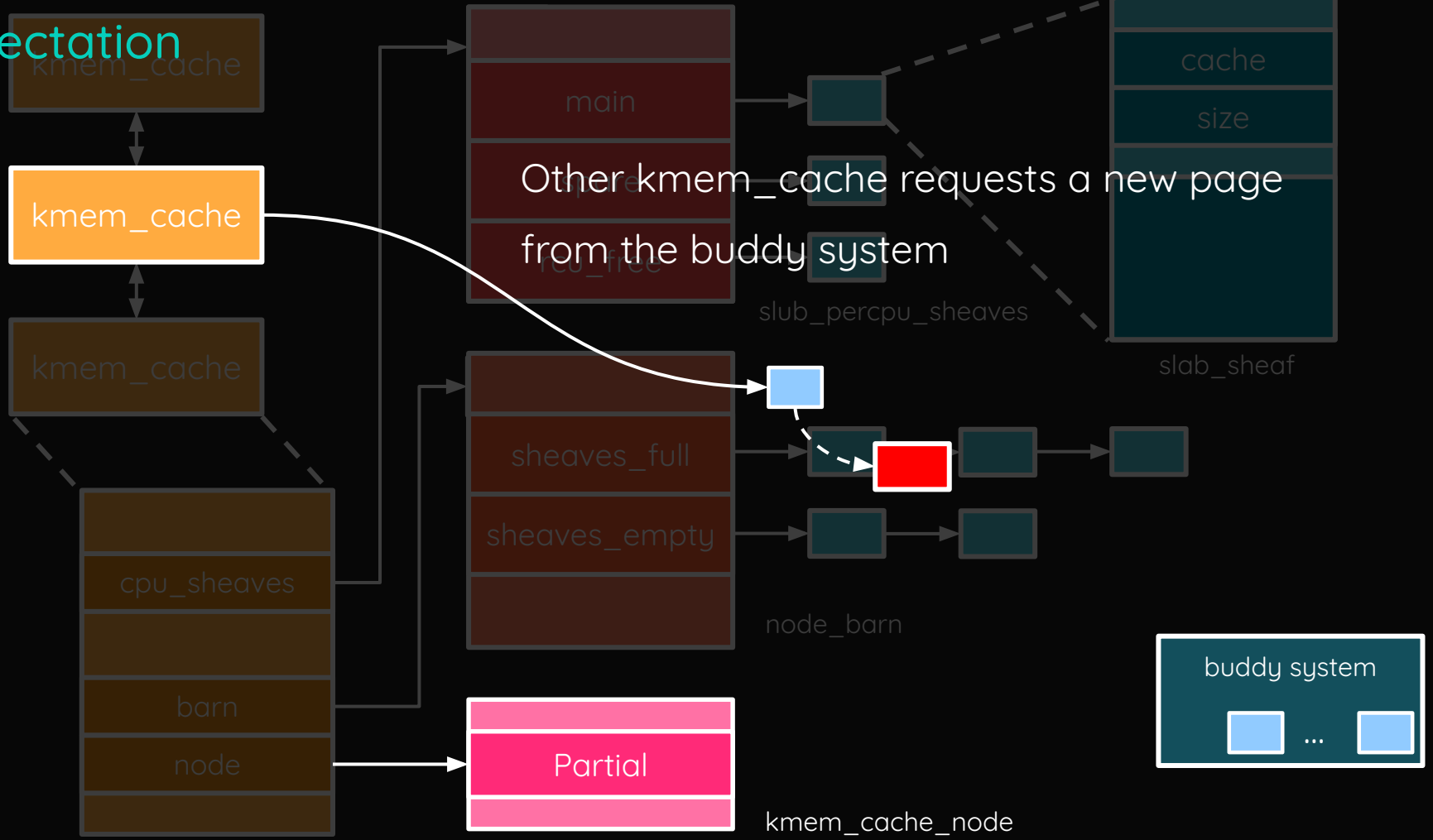

Expectation
kmem_cache
kmem_cache
kmem_cache
main
spare
rcu_free
slub_percpu_sheaves
cache
size
slab_sheaf
sheaves_full
sheaves_empty
node_barn
This slab/page contains the UAF object
barn
node
Partial
kmem_cache_node
slab / page

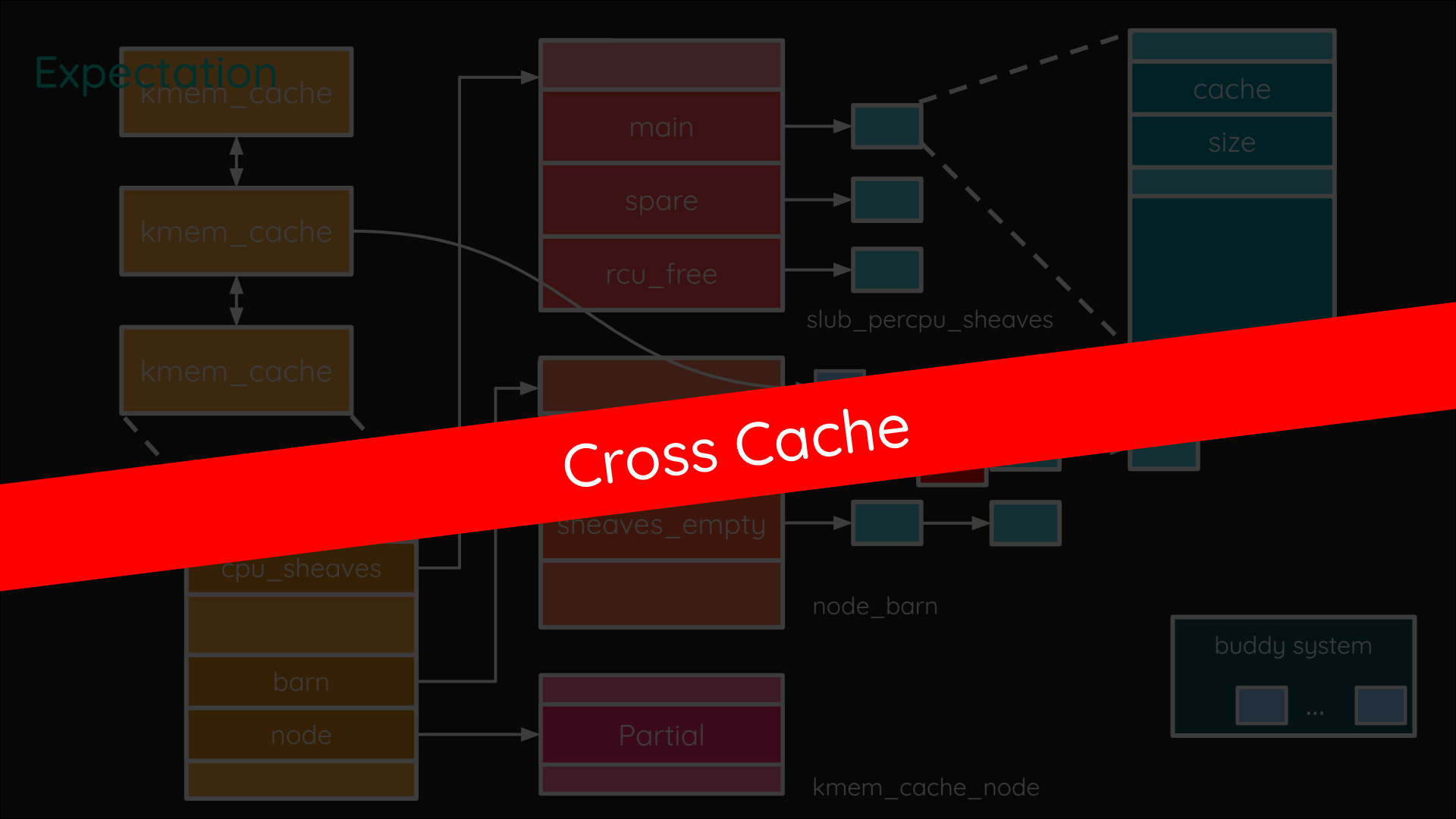

Expectation
kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_sheaves
cache
size
slab_sheaf
sheaves_full
sheaves_empty
node_barn
Heap Spray to Fill partial Freelist
Partial
...
kmem_cache_node

Expectation

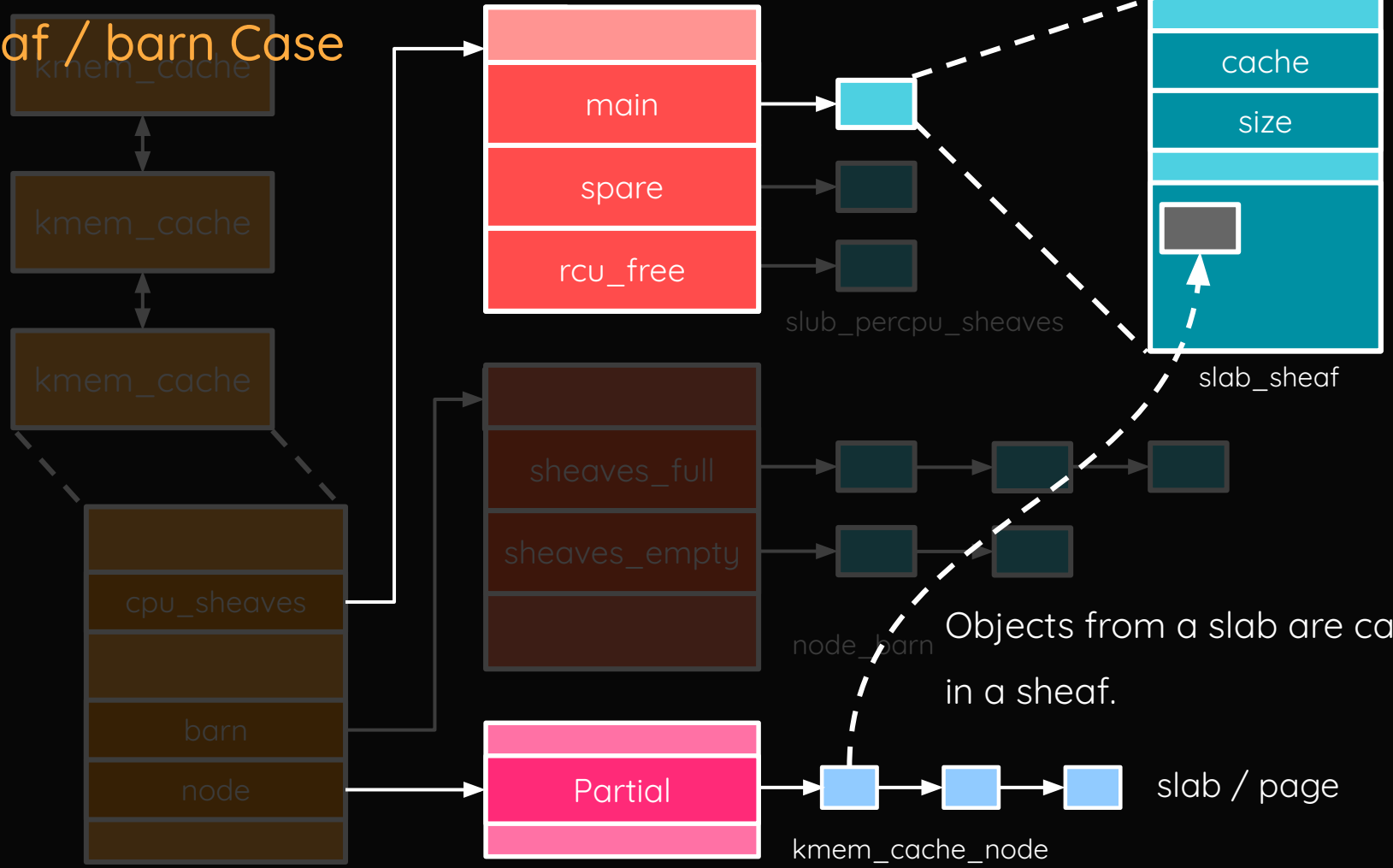


Expectation

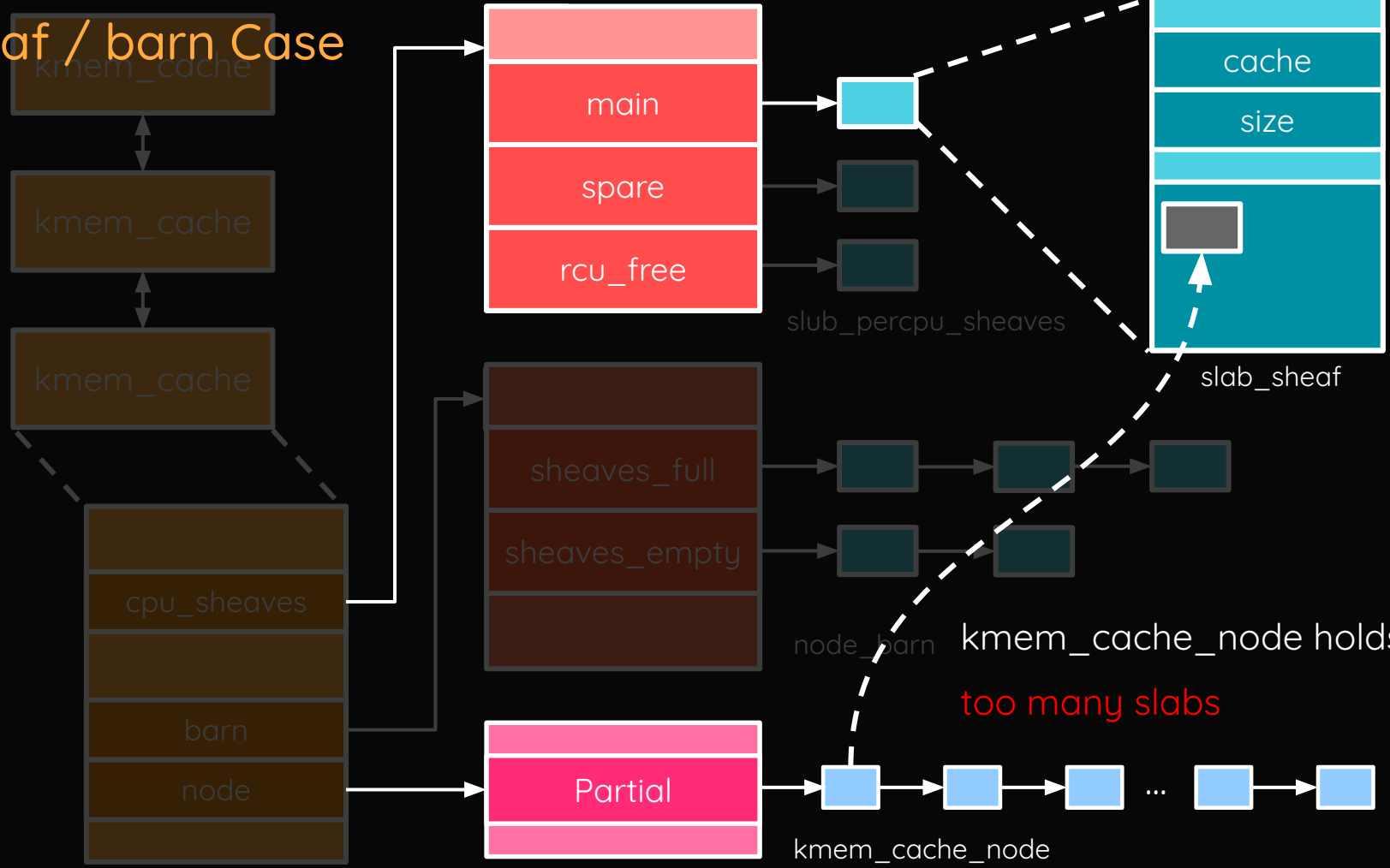



Expectation
kmem_cache
kmem_cache
kmem_cache
main
spare
rcu_free
slub_percpu_sheaves
cache
size
Cross Cache
sheaves_empty
node_barn
cpu_sheaves
barn
node
Partial
kmem_cache_node
buddy system
...

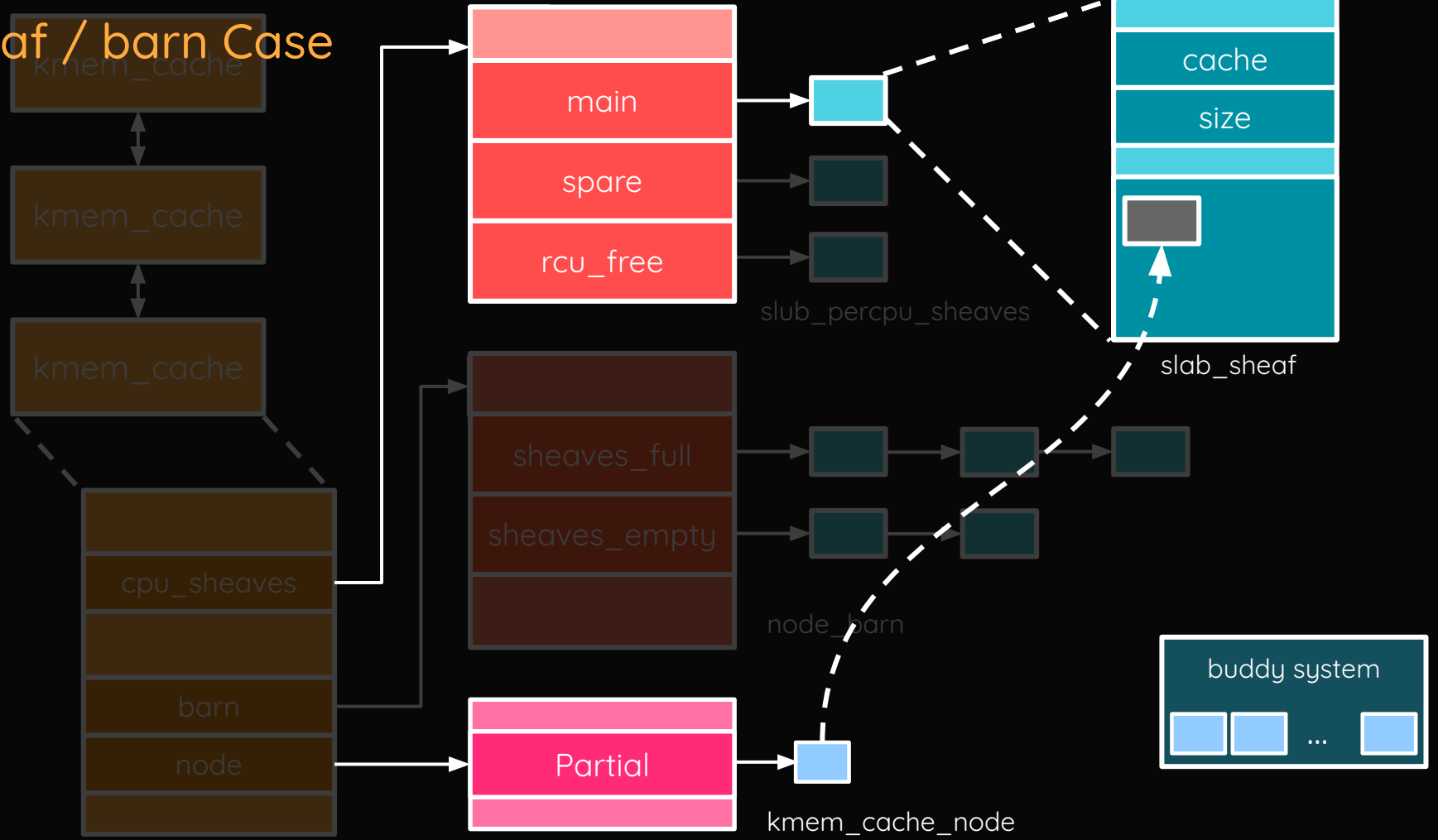
Sheaf / barn Case



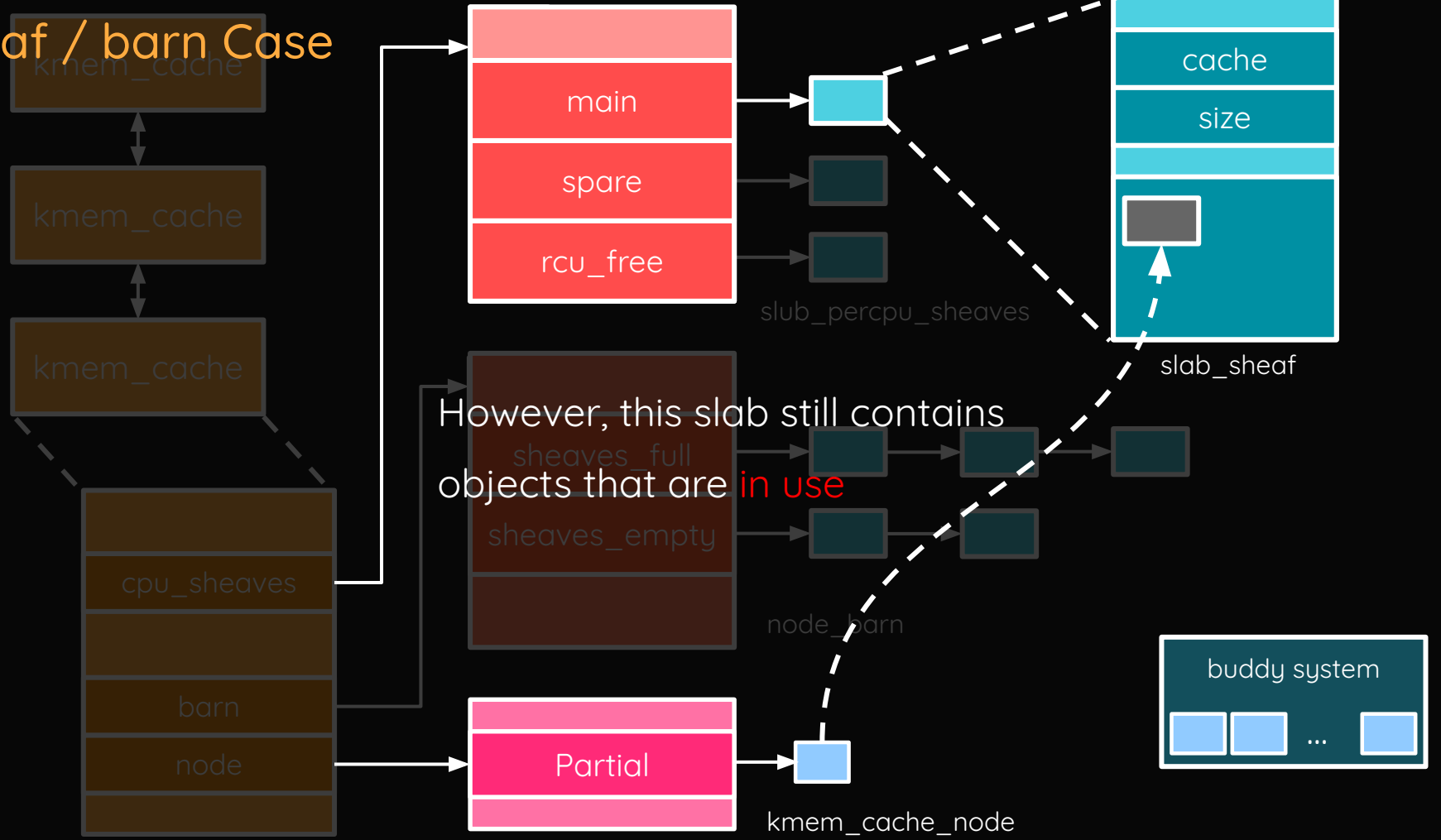
Sheaf / barn Case

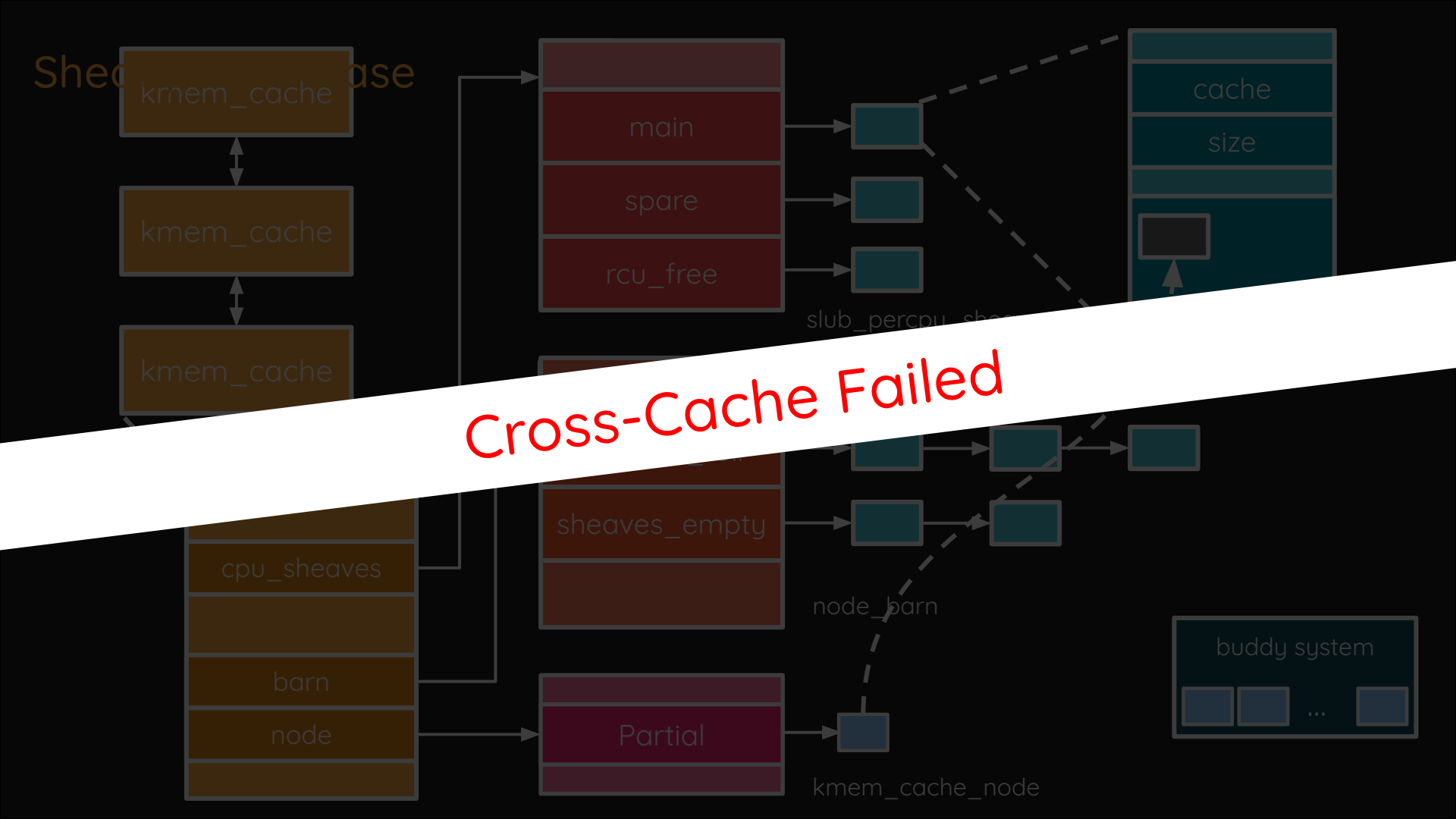


Sheaf / barn Case



Sheaf / barn Case



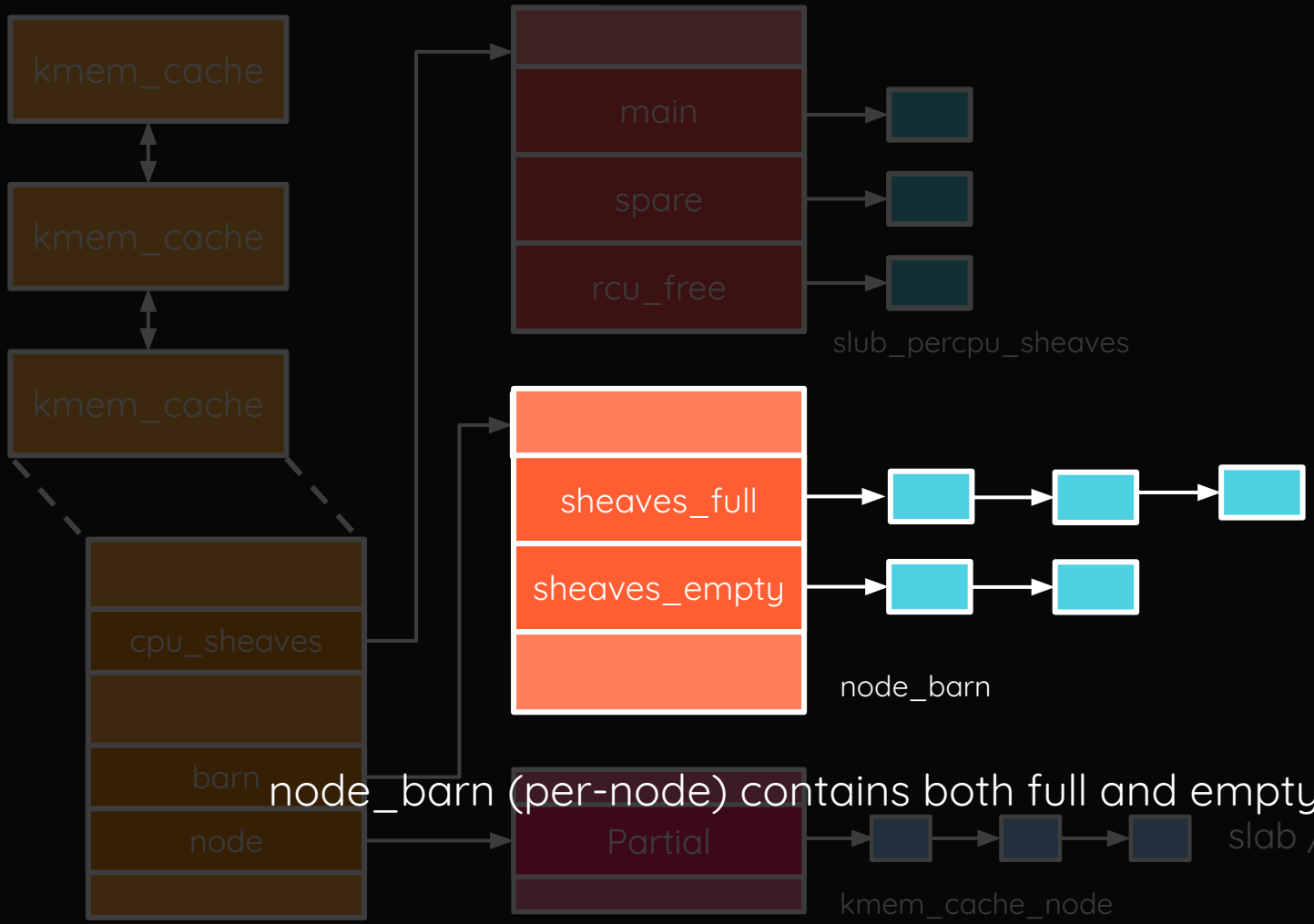

kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
sheaves_empty
Partial
cache
size
node_barn
buddy system
...
kmem_cache_node
Cross-Cache Failed

Cross-Cache’s Return to Grace

Achieving cross-cache in the Sheaf / Barn Environment

How to Cross-Cache ?

> Using the existing sheaf / barn mechanism



node_barn (per-node) contains both full and empty slab_sheaf



The upper limits on the sheaves stored in the barn are

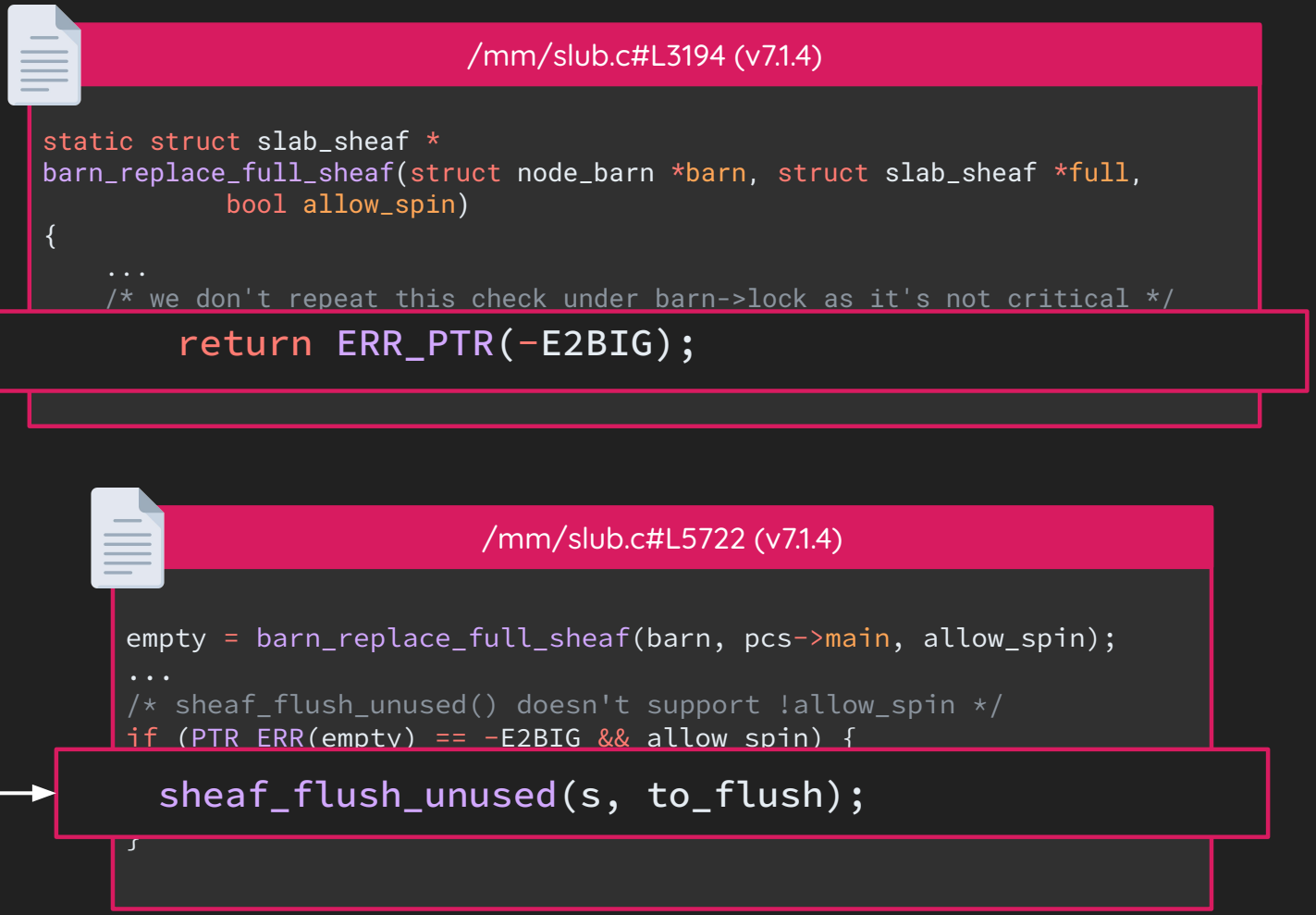
MAX_FULL_SHEAVES and MAX_EMPTY_SHEAVES

/mm/slub.c#L3194 (v7.1.4)

```
static struct slab_sheaf *
barn_replace_full_sheaf(struct node_barn *barn, struct slab_sheaf *full,
            bool allow_spin)
{
    ...
    /* we don't repeat this check under barn->lock as it's not critical */
    if (data_race(barn->nr_full) >= MAX_FULL_SHEAVES)
        return ERR_PTR(-E2BIG);
```

/mm/slub.c#L5722 (v7.1.4)

```
empty = barn_replace_full_sheaf(barn, pcs->main, allow_spin);
...
/* sheaf_flush_unused() doesn't support !allow_spin */
if (PTR_ERR(empty) == -E2BIG && allow_spin) {
    ...
    sheaf_flush_unused(s, to_flush);
}
```


/mm/slub.c#L3194 (v7.1.4)
static struct slab_sheaf *
barn_replace_full_sheaf(struct node_barn *barn, struct slab_sheaf *full,
bool allow_spin)
{
...
/* we don't repeat this check under barn->lock as it's not critical */
return ERR_PTR(-E2BIG);
/mm/slub.c#L5722 (v7.1.4)
empty = barn_replace_full_sheaf(barn, pcs->main, allow_spin);
...
/* sheaf_flush_unused() doesn't support !allow_spin */
if (PTR_ERR(empty) == -E2BIG && allow_spin) {
sheaf_flush_unused(s, to_flush);
}

sheaf_flush_unused() --> __kmem_cache_free_bulk()

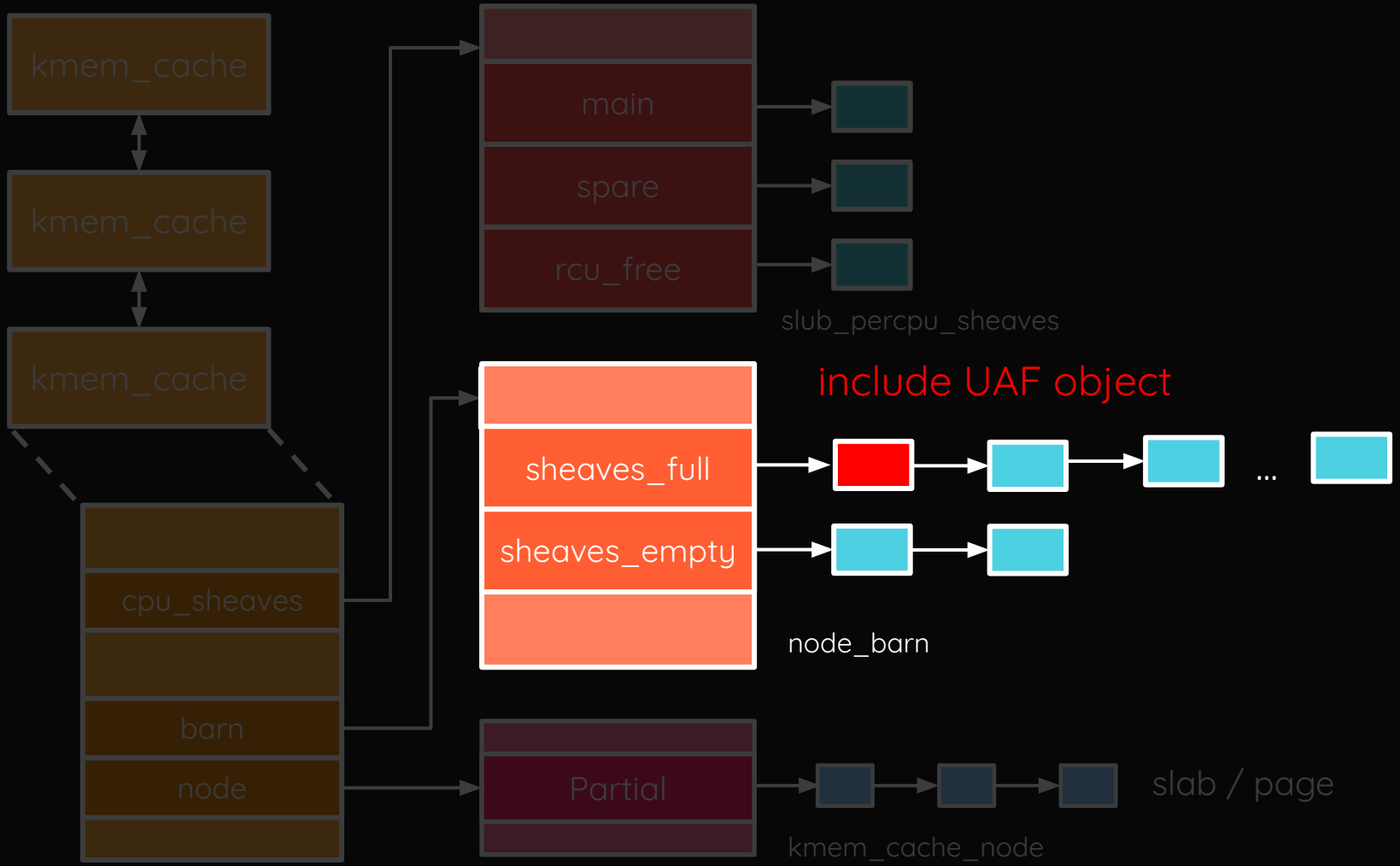


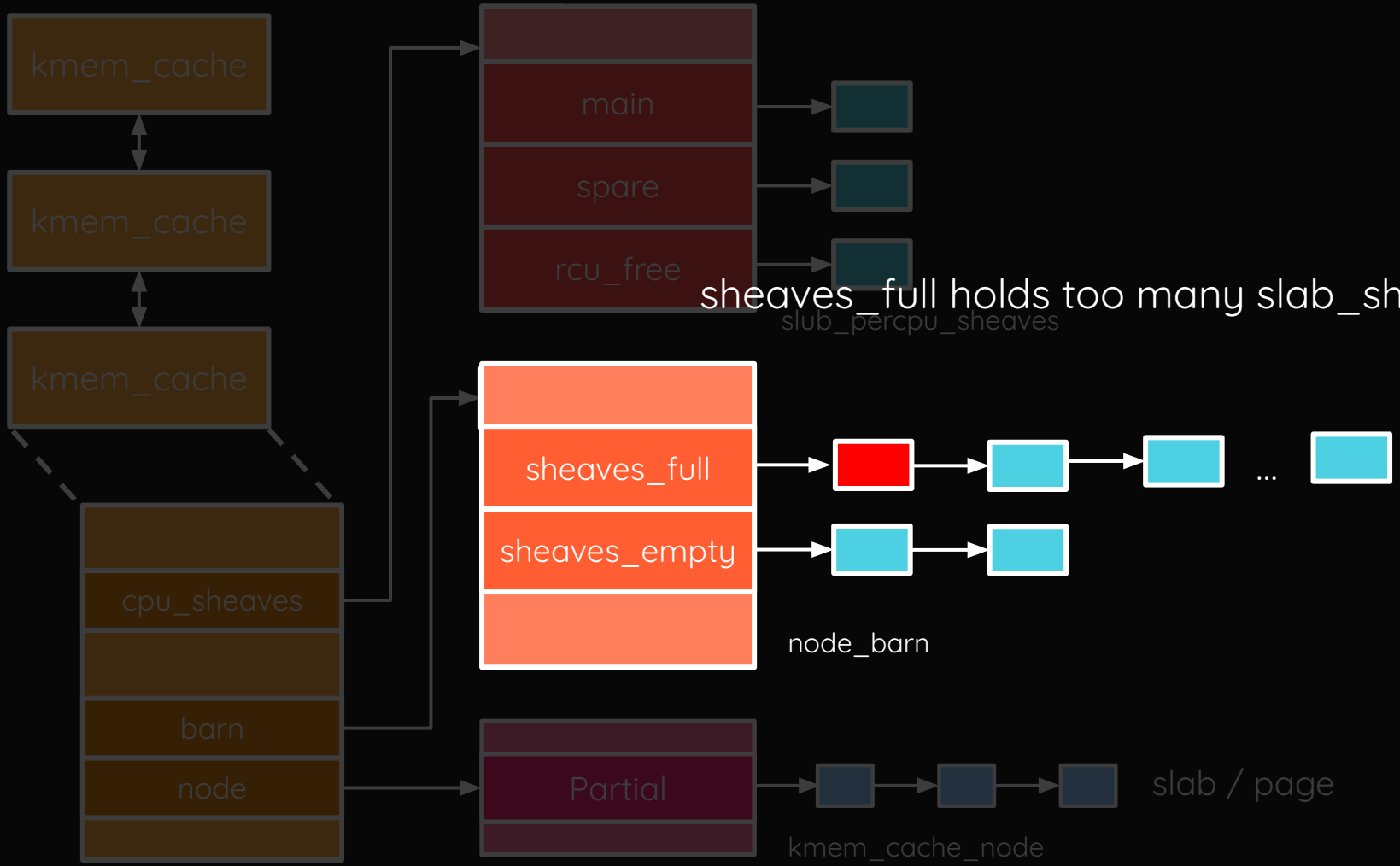
/mm/slub.c#L7063 (v7.1.4)

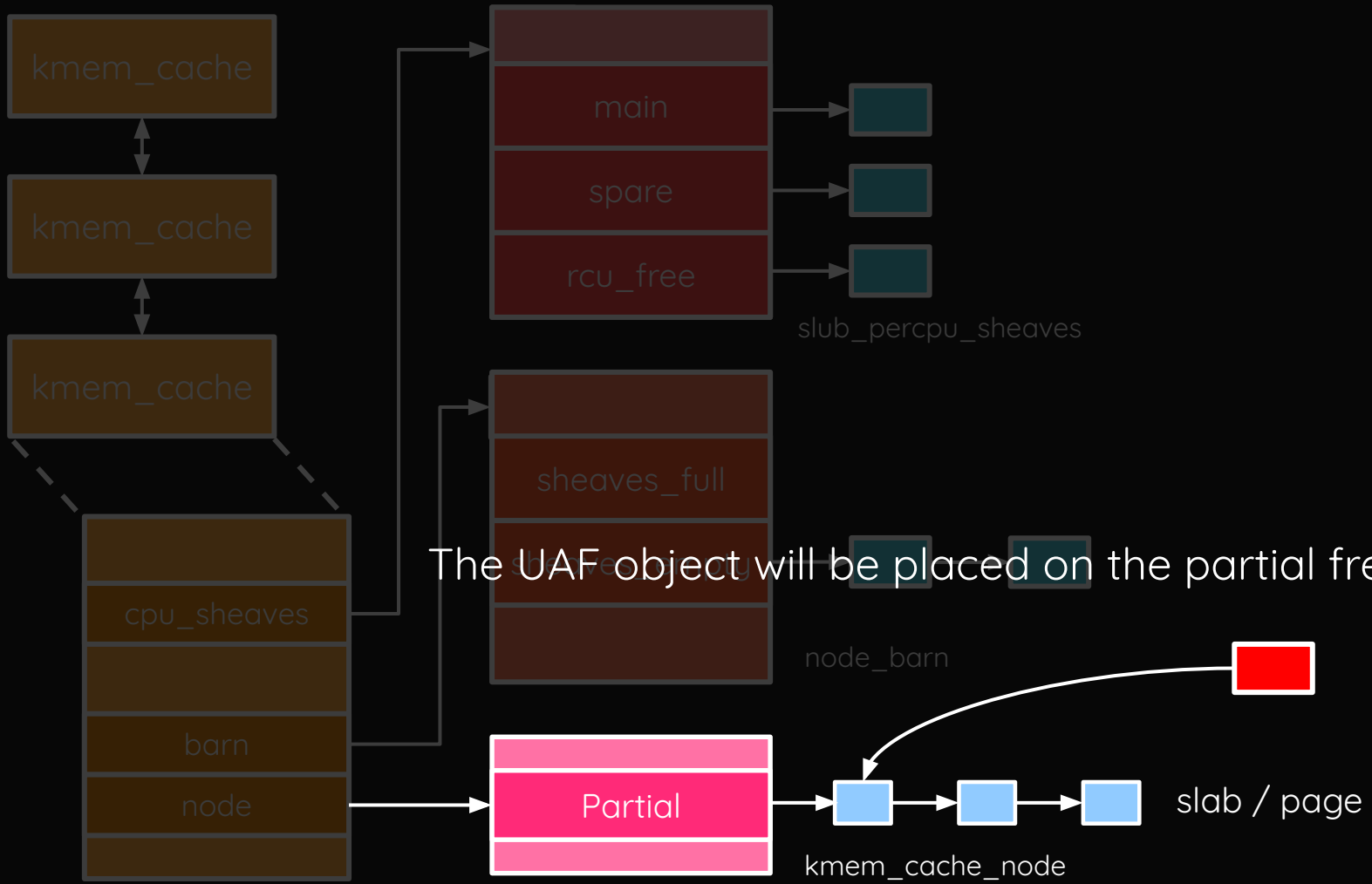
```
static void __kmem_cache_free_bulk(struct kmem_cache *s, size_t size, void **p)
{
    ...
    do {
        struct detached_freelist df;

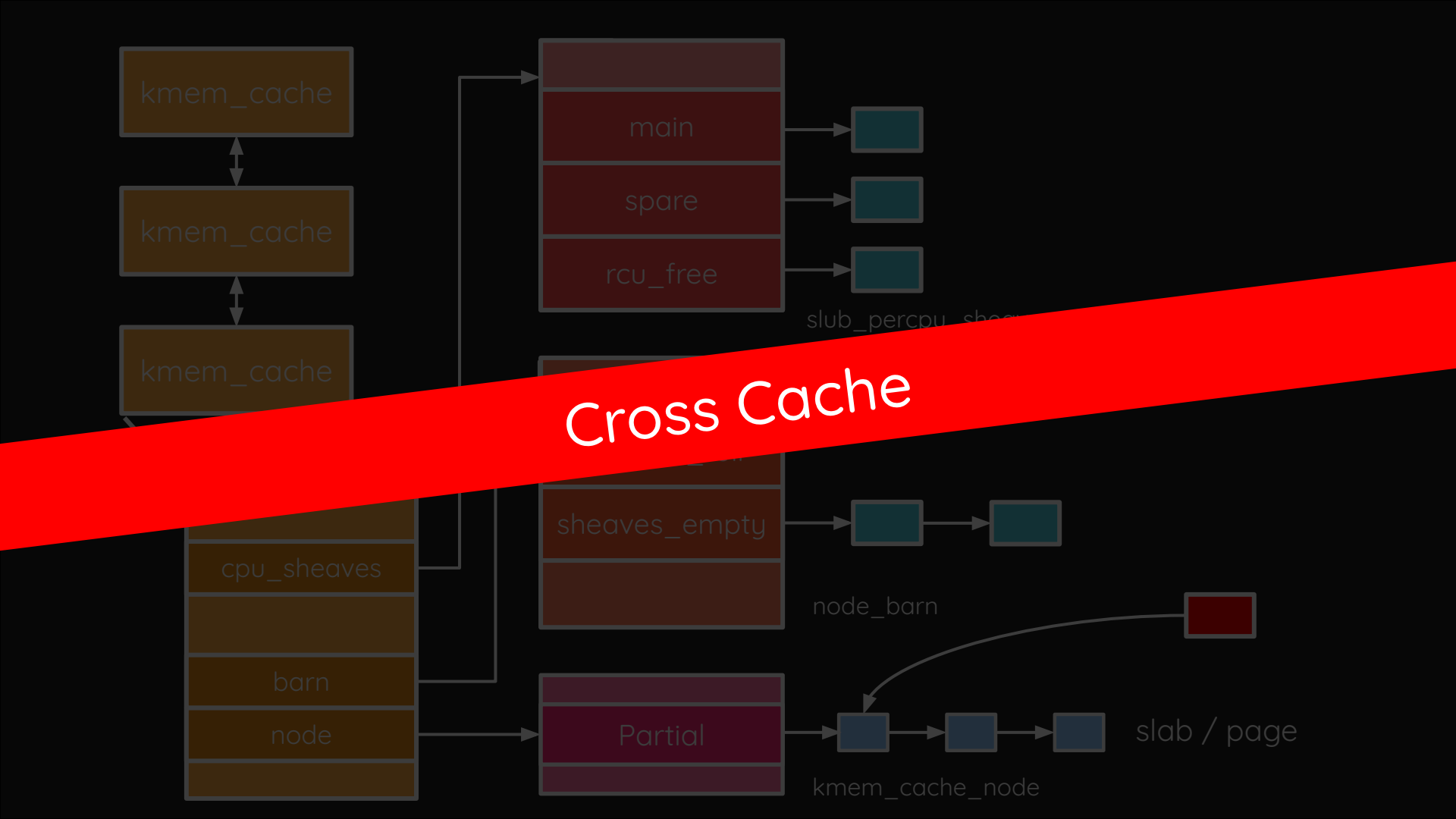
        size = build_detached_freelist(s, size, p, &df);
        ...
        __slab_free(df.s, df.slab, df.freelist, df.tail, df.cnt,
                    _RET_IP_);
    } while (likely(size));
}
```

Flush all objects back to the slab


kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_sheaves
include UAF object
sheaves_full
sheaves_empty
node_barn
...
Partial
slab / page
kmem_cache_node


kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_sheaves
sheaves_full holds too many slab_sheaf
sheaves_full
sheaves_empty
node_barn
...
Partial
slab / page
kmem_cache_node


kmem_cache
kmem_cache
kmem_cache
main
spare
rcu_free
slub_percpu_sheaves
sheaves_full
node_barn
cpu_sheaves
barn
node
Partial
kmem_cache_node
slab / page
The UAF object will be placed on the partial freelist


kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_shea
sheaves_empty
node_barn
Partial
slab / page
kmem_cache_node
Cross Cache

How to Cross-Cache ?

- Using the existing sheaf / barn mechanism
  - MAX_FULL_SHEAVES + 2 (main, spare) + 1 (trigger flush)

How to Cross-Cache ?

- Using the existing sheaf / barn mechanism
  - MAX_FULL_SHEAVES + 2 (main, spare) + 1 (trigger flush)
  - But this seems complicated. Is there a simpler way?

[PATCH v6 06/10] slab: skip percpu sheaves for remote object freeing

Vlastimil Babka vbabka at suse.cz
Wed Aug 27 01:26:38 PDT 2025

- Previous message (by thread): [PATCH v6 05/10] slab: determine barn status racily outside of lock
- Next message (by thread): [PATCH v6 07/10] slab: allow NUMA restricted allocations to use percpu sheaves
- **Messages sorted by:** [date] [thread] [subject] [author]

```
Since we don't control the NUMA locality of objects in percpu sheaves,
allocations with node restrictions bypass them. Allocations without
restrictions may however still expect to get local objects with high
probability, and the introduction of sheaves can decrease it due to
freed object from a remote node ending up in percpu sheaves.

The fraction of such remote frees seems low (5% on an 8-node machine)
but it can be expected that some cache or workload specific corner cases
exist. We can either conclude that this is not a problem due to the low
fraction, or we can make remote frees bypass percpu sheaves and go
directly to their slabs. This will make the remote frees more expensive,
but if if's only a small fraction, most frees will still benefit from
the lower overhead of percpu sheaves.

This patch thus makes remote object freeing bypass percpu sheaves,
including bulk freeing, and kfree_rcu() via the rcu_free sheaf. However
it's not intended to be 100% guarantee that percpu sheaves will only
contain local objects. The refill from slabs does not provide that
guarantee in the first place, and there might be cpu migrations
```

https://lists.infradead.org/pipermail/maple-tree/2025-August/004477.html

[PATCH skip percpu sheaves for remote object freeing

Vlastimil Babka vbabka at suse.cz
Wed Aug 27 01:26:38 PDT 2025



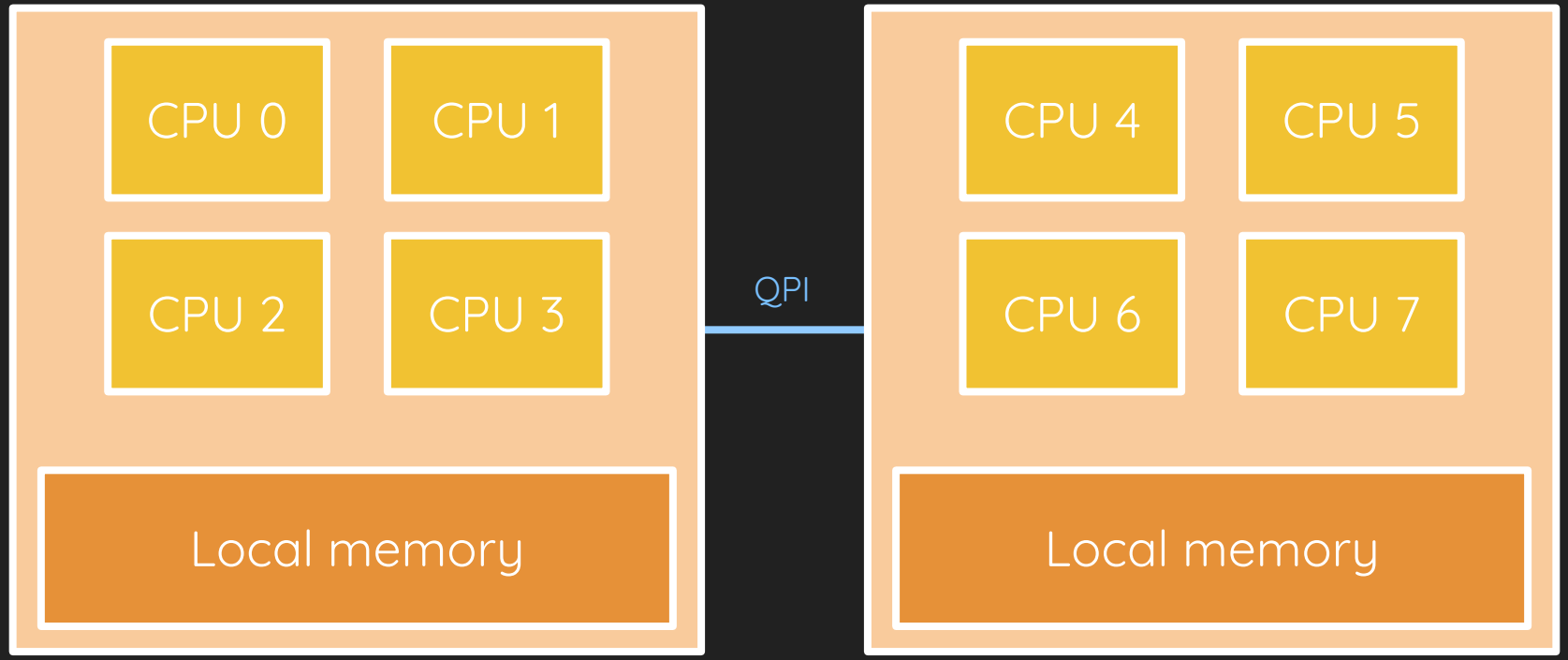
- Previous message (by thread): [PATCH v6 05/10] slab: determine barn status racily outside of lock
- Next message (by thread): [PATCH v6 07/10] slab: allow NUMA restricted allocations to use percpu sheaves
- **Messages sorted by:** [date] [thread] [subject] [author]

```
Since we don't control the NUMA locality of objects in percpu sheaves,
allocations with node restrictions bypass them. Allocations without
restrictions may however still expect to get local objects with high
probability, and the introduction of sheaves can decrease it due to
freed object from a remote node ending up in percpu sheaves.

The fraction of such remote frees seems low (5% on an 8-node machine)
but it can be expected that some cache or workload specific corner cases
exist. We can either conclude that this is not a problem due to the low
fraction, or we can make remote frees bypass percpu sheaves and go
directly to their slabs. This will make the remote frees more expensive,
but if if's only a small fraction, most frees will still benefit from
the lower overhead of percpu sheaves.

This patch thus makes remote object freeing bypass percpu sheaves,
including bulk freeing, and kfree_rcu() via the rcu_free sheaf. However
it's not intended to be 100% guarantee that percpu sheaves will only
contain local objects. The refill from slabs does not provide that
guarantee in the first place, and there might be cpu migrations
```

https://lists.infradead.org/pipermail/maple-tree/2025-August/004477.html


CPU 0
CPU 1
CPU 2
CPU 3
QPI
CPU 4
CPU 5
CPU 6
CPU 7
Local memory
Local memory

NUMA node



/mm/slub.c#L6246 (v7.1.4)

```
static __always_inline bool can_free_to_pcs(struct slab *slab)
{
    int slab_node;
    int numa_node;

    if (!IS_ENABLED(CONFIG_NUMA))
        goto check_pfmemalloc;

    slab_node = slab_nid(slab);
    ...
    numa_node = numa_mem_id();
    if (likely(slab_node == numa_node))
        goto check_pfmemalloc;
    ...
    numa_node = numa_node_id();
    /* freed object is from this cpu's node, proceed */
    if (likely(slab_node == numa_node))
        goto check_pfmemalloc;
    ...
    return false;

check_pfmemalloc:
    return likely(!slab_test_pfmemalloc(slab));
}
```

/mm/slub.c#L6246 (v7.1.4)

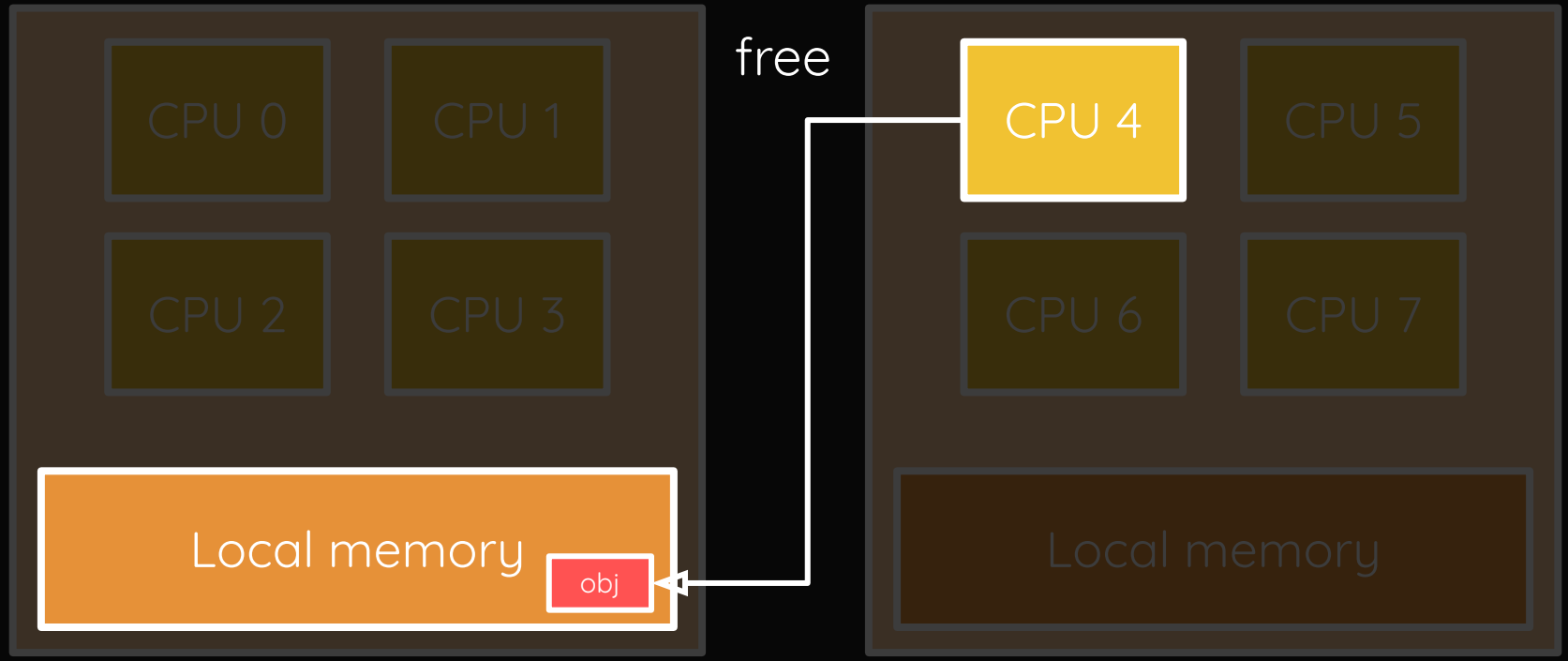
```
static __always_inline bool can_free_to_pcs(struct slab *slab)
{
    int slab_node;
    int numa_node;

    if (!IS_ENABLED(CONFIG_NUMA))
        goto check_pfmemalloc;

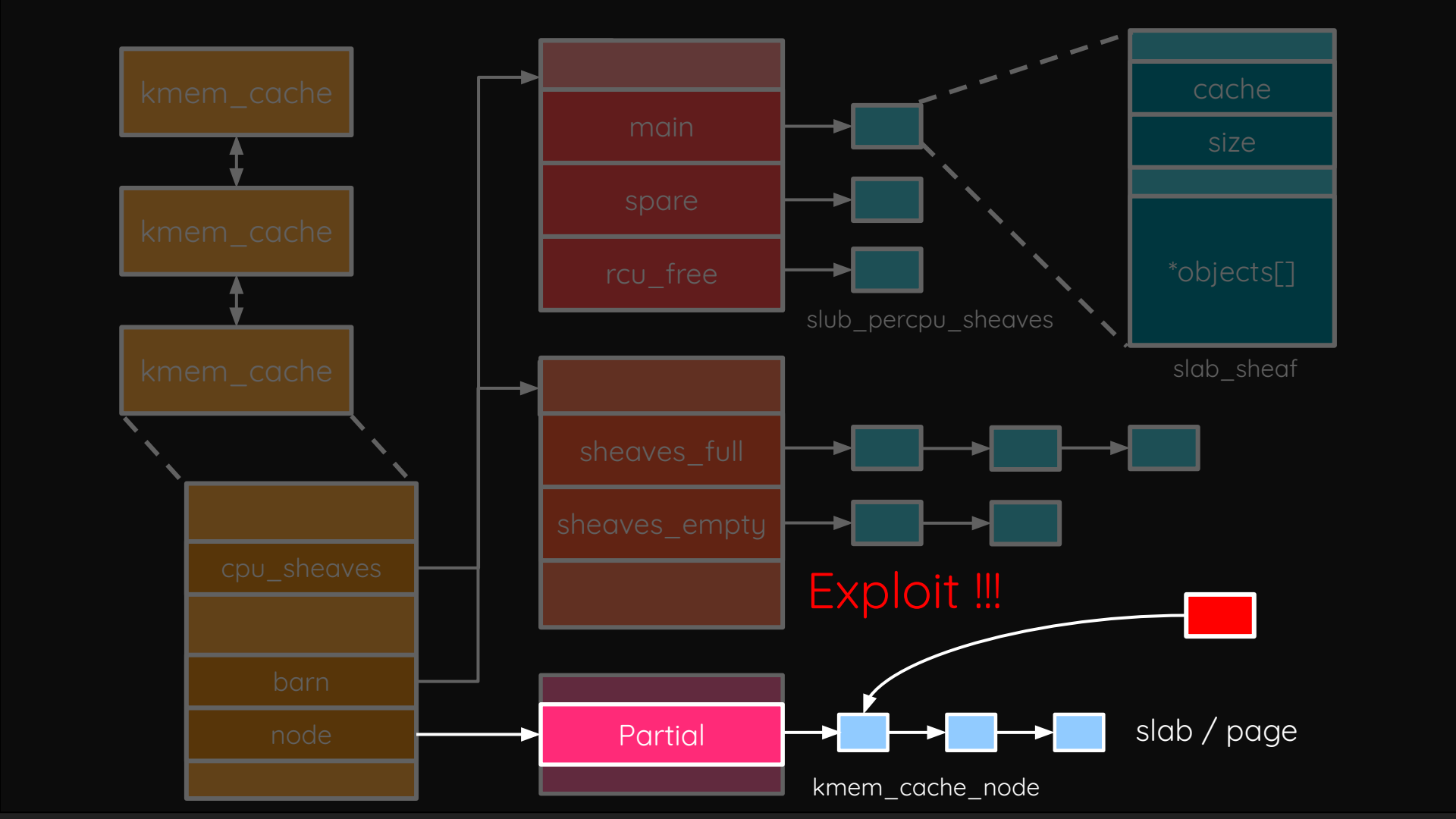
    numa_node = numa_mem_id();
    if (likely(slab_node == numa_node))
        goto check_pfmemalloc;

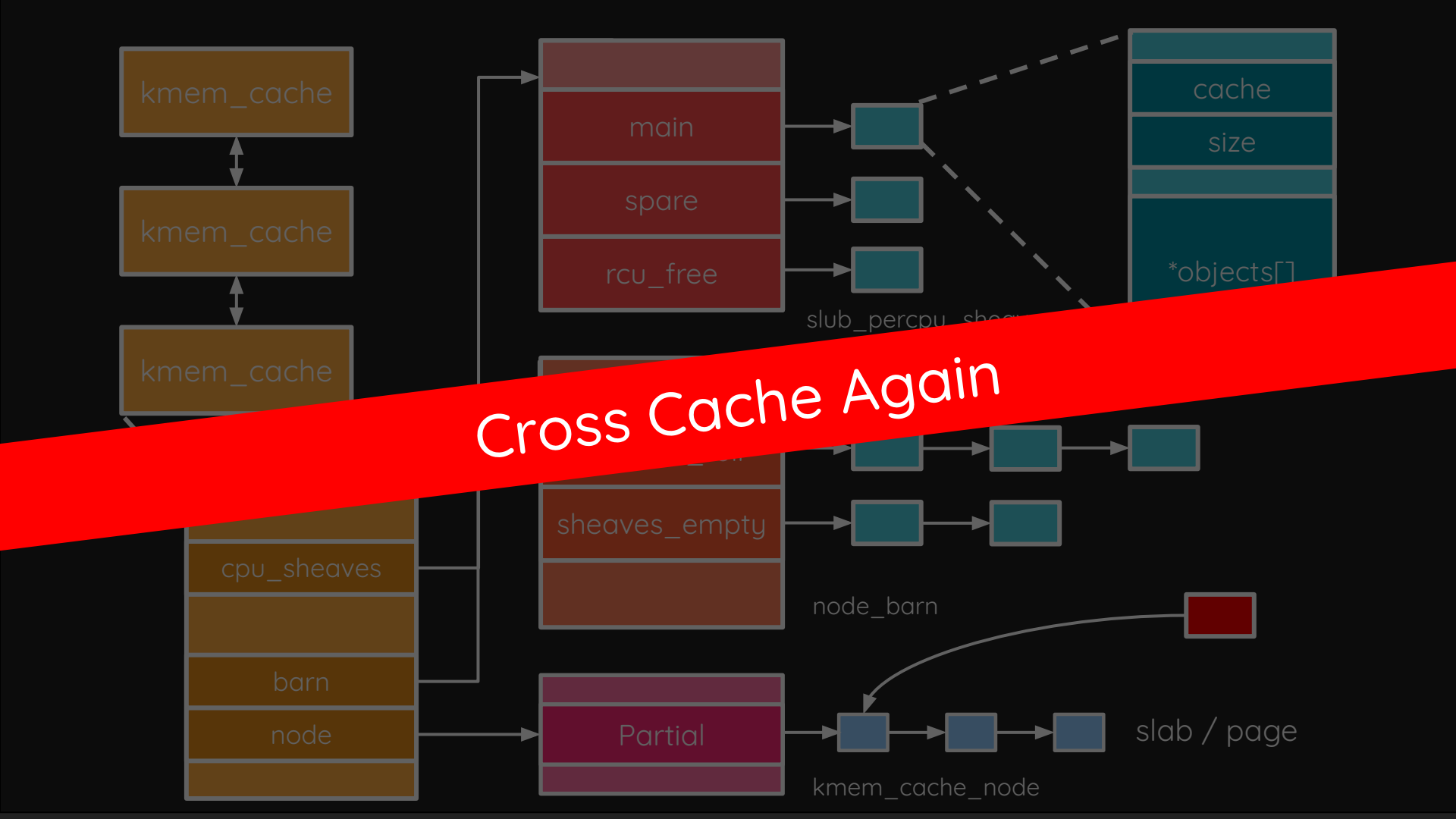
    /* freed object is from this cpu's node, proceed */
    if (likely(slab_node == numa_node))
        goto check_pfmemalloc;
    ...
    return false;

check_pfmemalloc:
    return likely(!slab_test_pfmemalloc(slab));
}
```



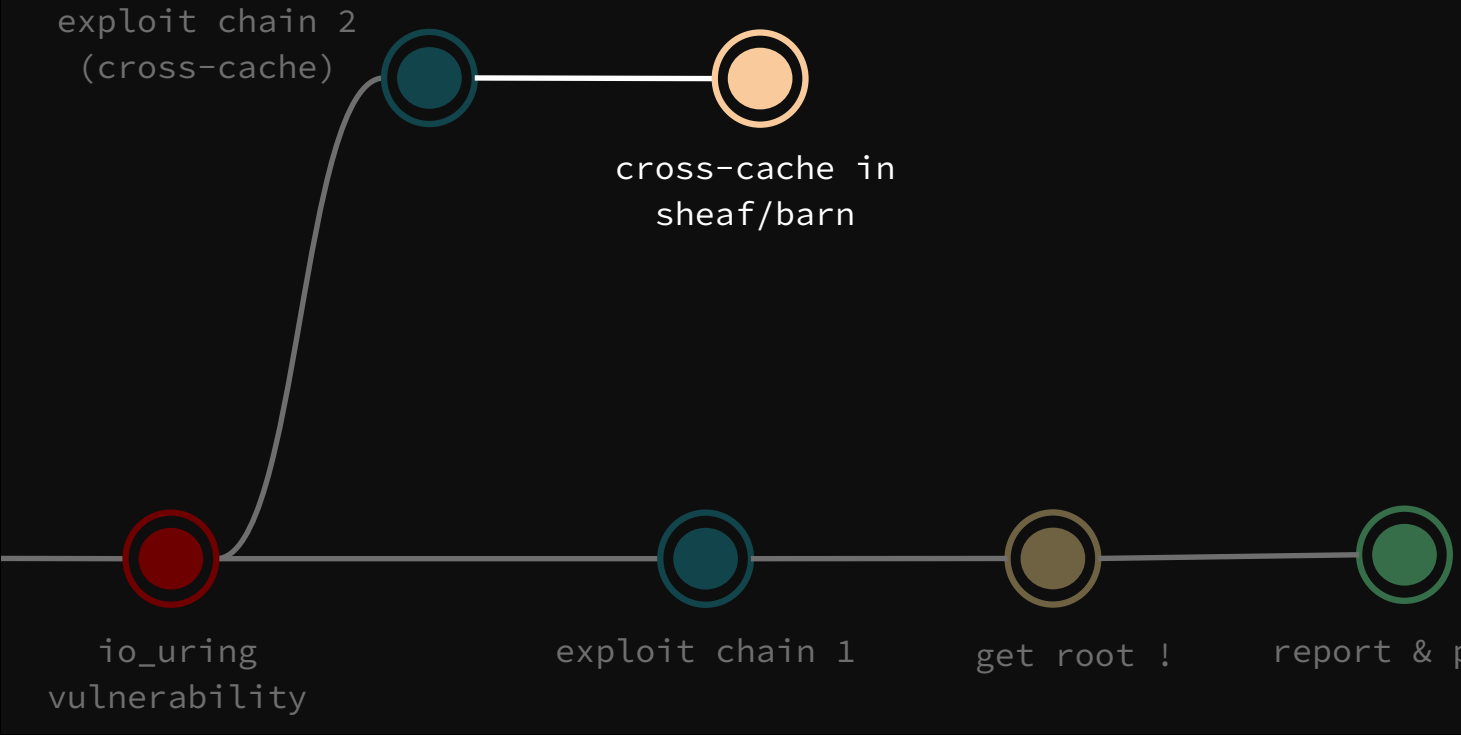
Let the object be allocated on NUMA node 1 but freed on NUMA node 2


kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_sheaves
cache
size
*objects[]
slab_sheaf
sheaves_full
sheaves_empty
Exploit !!!
Partial
kmem_cache_node
slab / page


kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_shea
cache
size
*objects[]
sheaves_empty
node_barn
Partial
kmem_cache_node
slab / page
Cross Cache Again

How to Cross-Cache ?

- Using the existing sheaf / barn mechanism
  - MAX_FULL_SHEAVES + 2 (main, spare) + 1 (trigger flush)
  - But this seems complicated. Is there a simpler way? (Yes)
- Use NUMA nodes to prevent your objects from being cached by the sheaf


exploit chain 2
(cross-cache)
cross-cache in
sheaf/barn
io_uring
vulnerability
exploit chain 1
get root !
report & patch

We already know that the sheaf/barn mechanism affects many traditional SLUB exploitation techniques

Does this mechanism only make exploitation

more difficult and inconvenient?

NO

Sheaf Design Flaws

Inconsistency Between the Sheaf and Traditional Slab Implementations

If you've ever exploited the Linux kernel

this exploit path just feels

If you've ever exploited the Linux kernel

Weird !!!

this exploit path just feels

CONFIG_SLAB_FREELIST_HARDENED

CONFIG_SLAB_FREELIST_HARDENED

> Double Free Check

> Pointer Protection

```
config SLAB_FREELIST_HARDENED
        bool "Harden slab freelist metadata"
        depends on !SLUB_TINY
        help
          Many kernel heap attacks try to target slab cache metadata and
          other infrastructure. This options makes minor performance
          sacrifices to harden the kernel slab allocator against common
          freelist exploit methods.
```

CONFIG_SLAB_FREELIST_HARDENED will check double free

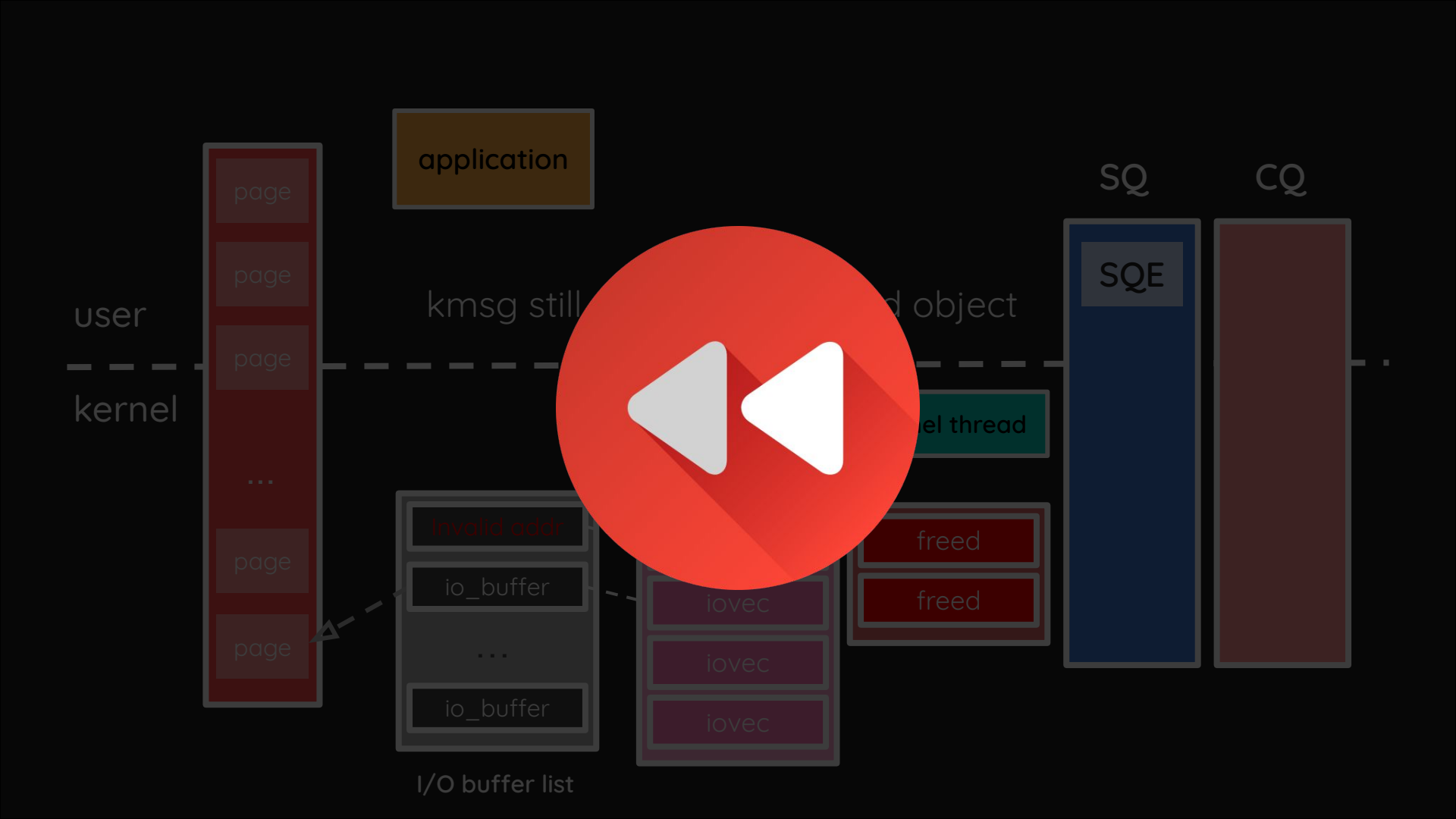


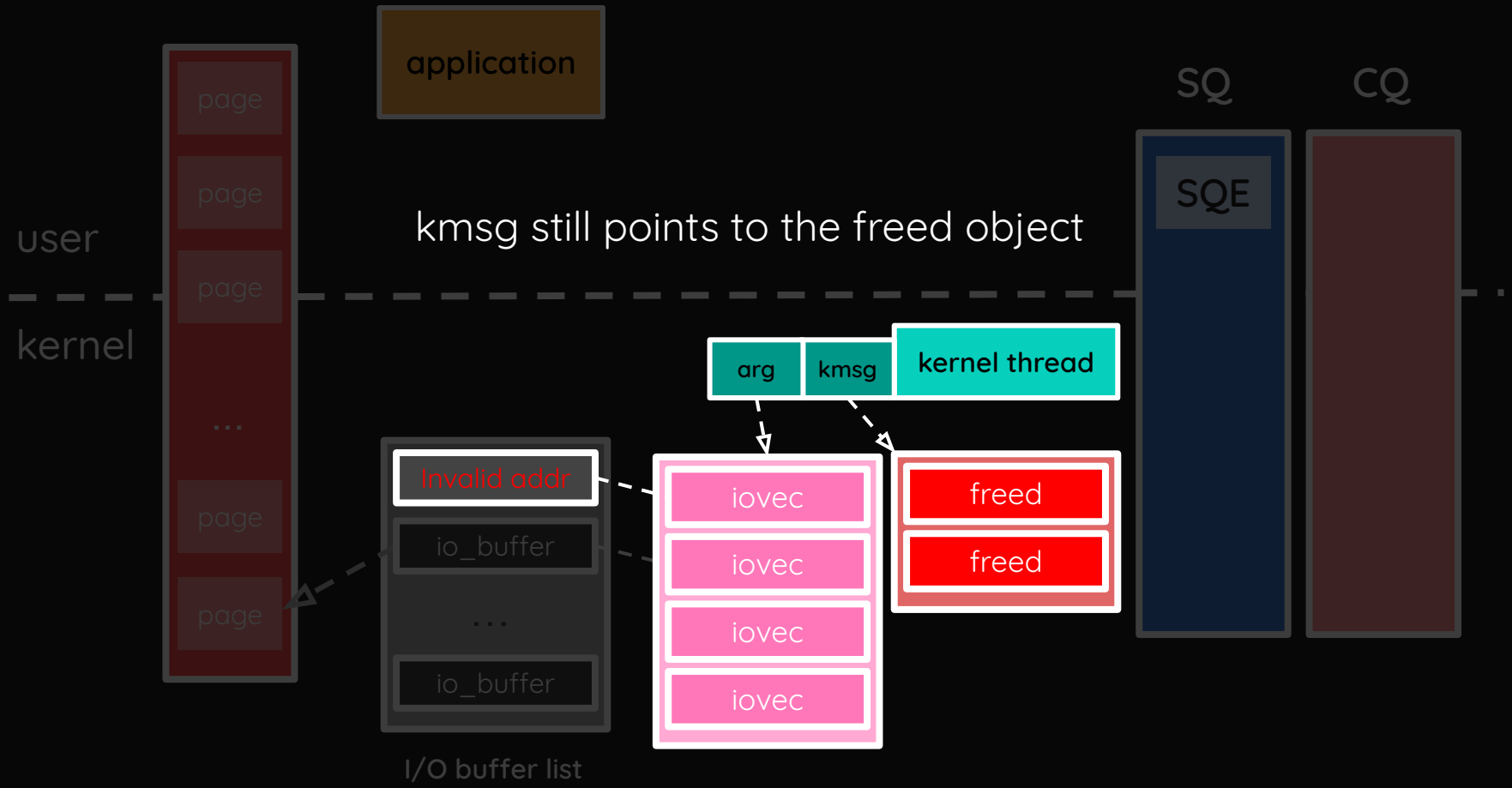
/mm/slub.c#L541

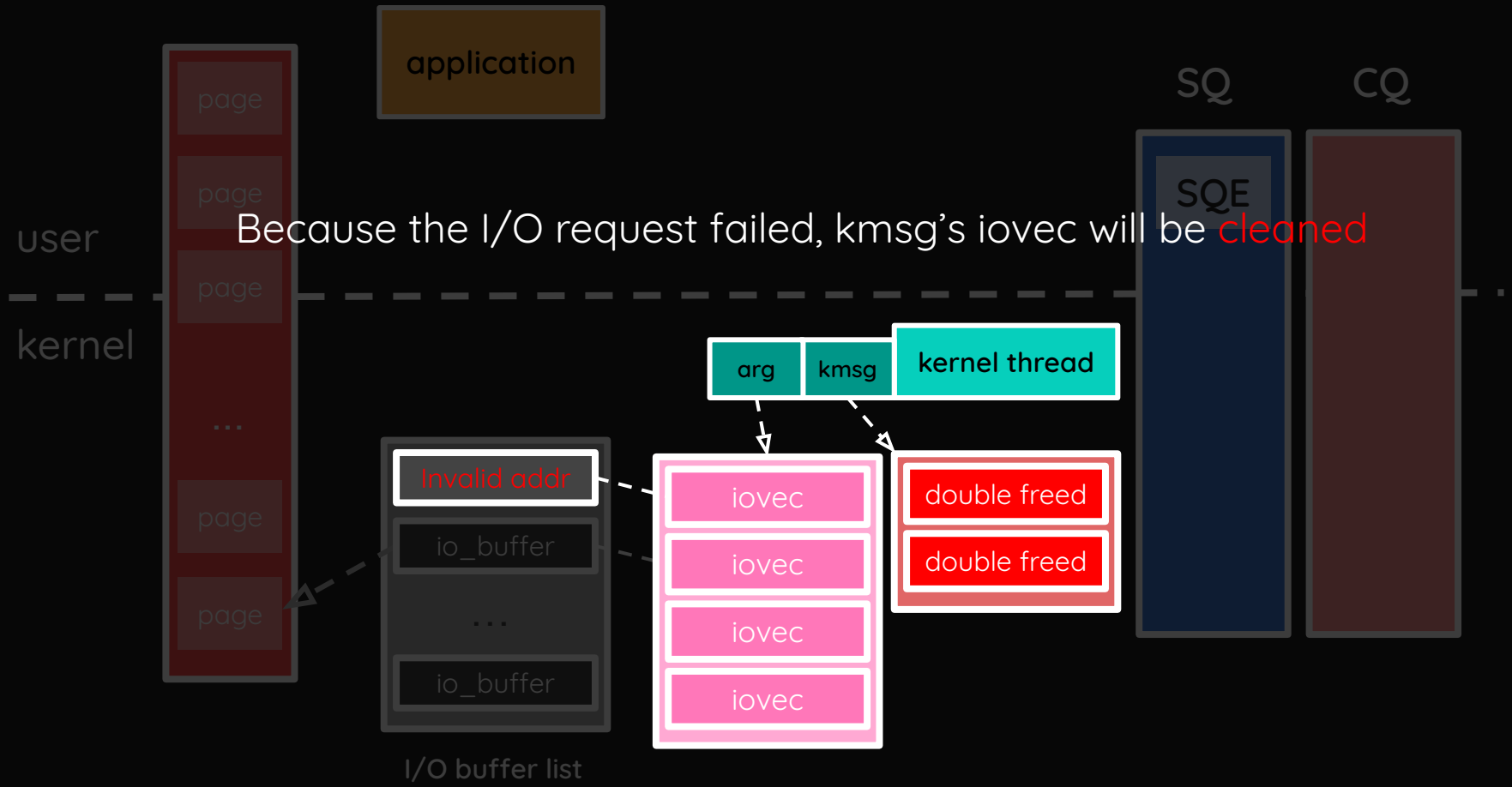
```
static inline void set_freepointer(struct kmem_cache *s, void *object, void *fp)
{

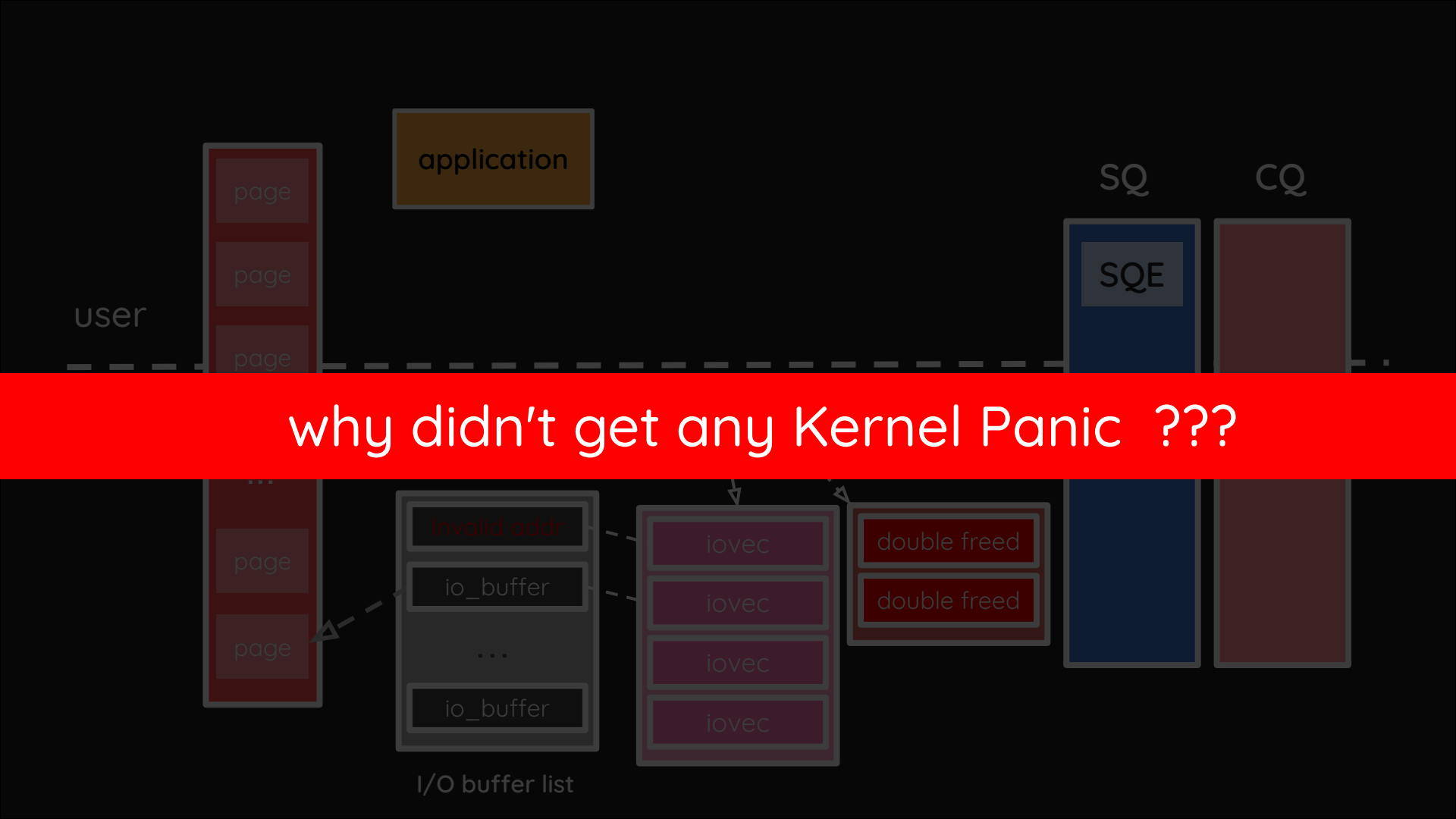
#ifdef CONFIG_SLAB_FREELIST_HARDENED
    BUG_ON(object == fp); /* naive detection of double free or corruption */
#endif

}
```


application
SQ
CQ
SQE
user
kernel
kmsg still
d object
page
page
page
...
page
page
el thread
freed
freed
Invalid addr
io_buffer
...
io_buffer
iovec
iovec
iovec
I/O buffer list


application
SQ
CQ
SQE
page
page
page
...
page
page
user
kernel
kmsg still points to the freed object
arg
kmsg
kernel thread
Invalid addr
io_buffer
...
io_buffer
iovec
iovec
iovec
iovec
freed
freed
I/O buffer list


application
SQ
CQ
SQE
user
kernel
page
page
page
...
page
page
Because the I/O request failed, kmsg's iovec will be cleaned
arg
kmsg
kernel thread
Invalid addr
io_buffer
...
io_buffer
iovec
iovec
iovec
iovec
double freed
double freed
I/O buffer list



why didn't get any Kernel Panic ???

kfree()



/mm/slub.c#L5806 (v7.1.4)

```
static __fastpath_inline
void slab_free(struct kmem_cache *s, struct slab *slab, void *object,
               unsigned long addr) {
    ...
    if (likely(can_free_to_pcs(slab)) && likely(free_to_pcs(s, object, true)))
        return;

    __slab_free(s, slab, object, object, 1, addr);
    stat(s, FREE_SLOWPATH);
}
```

fast path

slow path

Slow Path

/mm/slub.c#L5806 (v7.1.4)

```
static __fastpath_inline
void slab_free(struct kmem_cache *s, struct slab *slab, void *object,
           unsigned long addr) {
    ...
    if (likely(can_free_to_pcs(slab)) && likely(free_to_pcs(s, object, true)))
        return;

    __slab_free(s, slab, object, object, 1, addr);

}
```

/mm/slub.c#L5512 (v7.1.4)

```
static void __slab_free(struct kmem_cache *s, struct slab *slab,
            void *head, void *tail, int cnt,
            unsigned long addr) {
    ...
        set_freepointer(s, tail, old.freelist);
}
```

Fast Path

/mm/slub.c#L5806 (v7.1.4)

```
static __fastpath_inline
void slab_free(struct kmem_cache *s, struct slab *slab, void *object,
               unsigned long addr) {

    if (likely(can_free_to_pcs(slab)) && likely(free_to_pcs(s, object, true)))

    __slab_free(s, slab, object, object, 1, addr);
    stat(s, FREE_SLOWPATH);
}
```

/mm/slub.c#L6246 (v7.1.4)

```
static __fastpath_inline
bool free_to_pcs(struct kmem_cache *s, void *object, bool allow_spin)
{
    ...
    pcs->main->objects[pcs->main->size++] = object;
    ...
}
```

❌ set_freepointer()

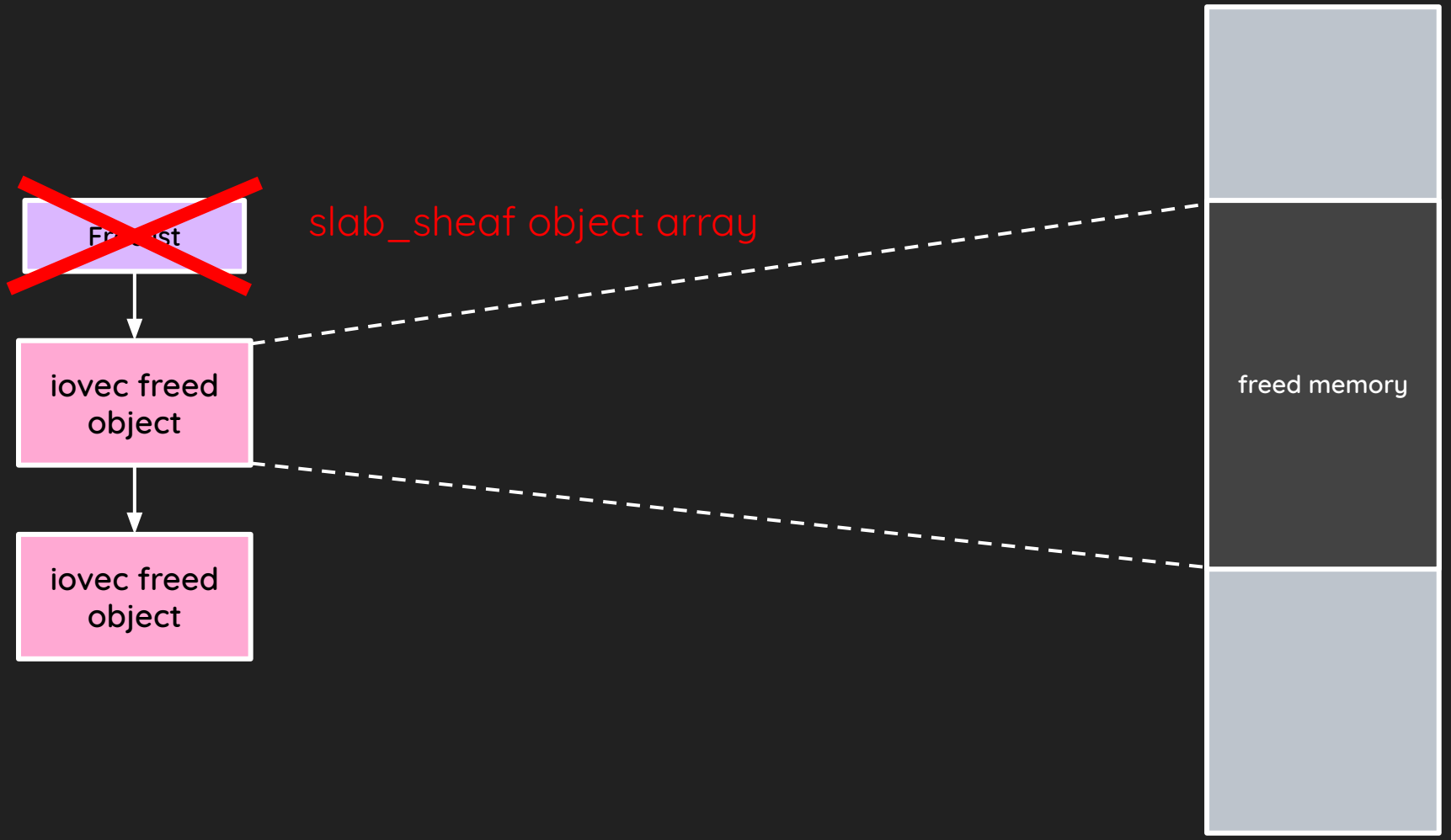
/mm/slub.c#L504 (v7.1.4)

```
static inline freeptr_t freelist_ptr_encode(const struct kmem_cache *s,
                                            void *ptr, unsigned long ptr_addr)
{
    ...
#ifdef CONFIG_SLAB_FREELIST_HARDENED
    encoded = (unsigned long)ptr ^ s->random ^ swab(ptr_addr);
    ...
}
```

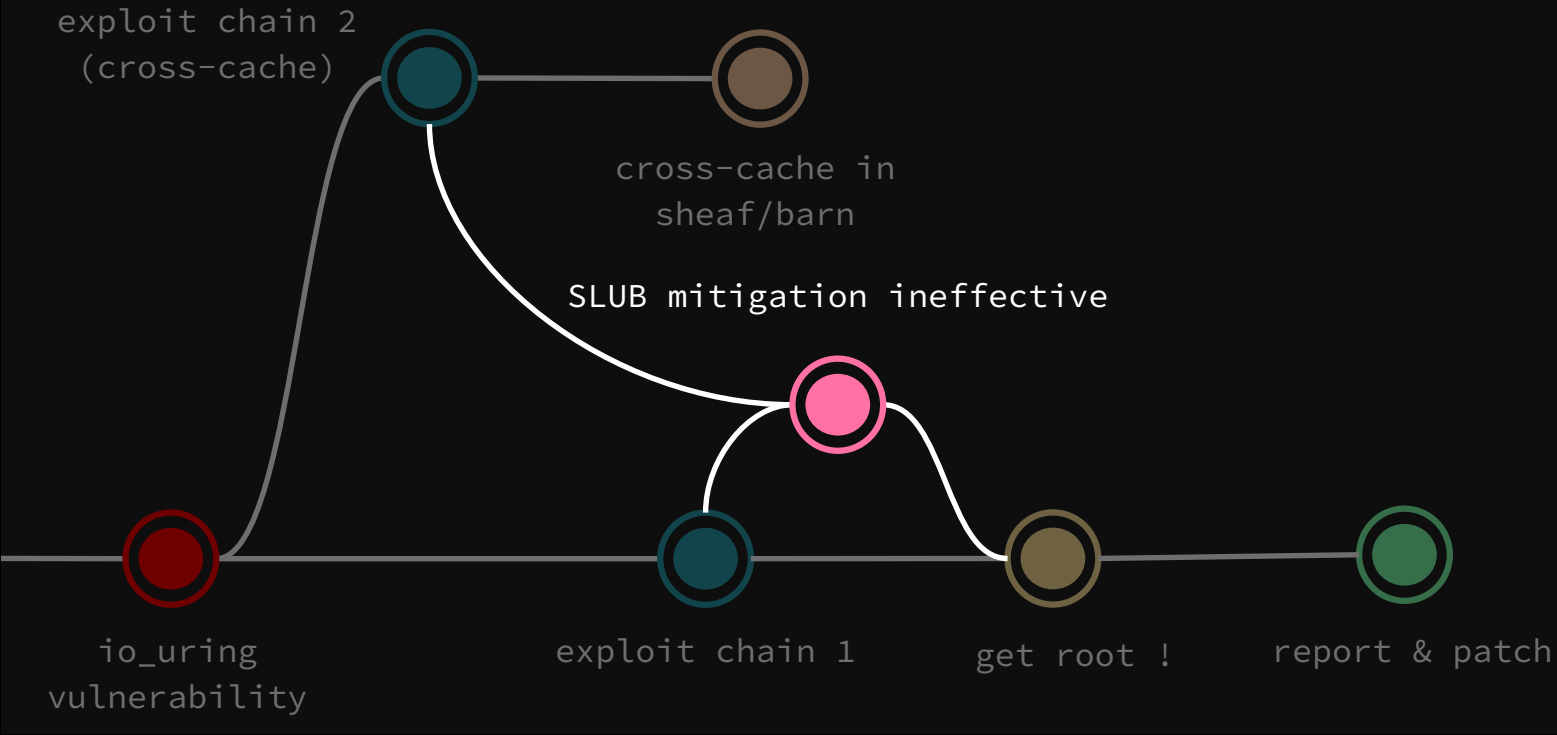
/mm/slub.c#L6246 (v7.1.4)

```
static __fastpath_inline
bool free_to_pcs(struct kmem_cache *s, void *object, bool allow_spin)
{
    ...
    pcs->main->objects[pcs->main->size++] = object;
    ...
}
```

The pointer protection mitigation is also ineffective


slab_sheaf object array
iovec freed object
iovec freed object
freed memory

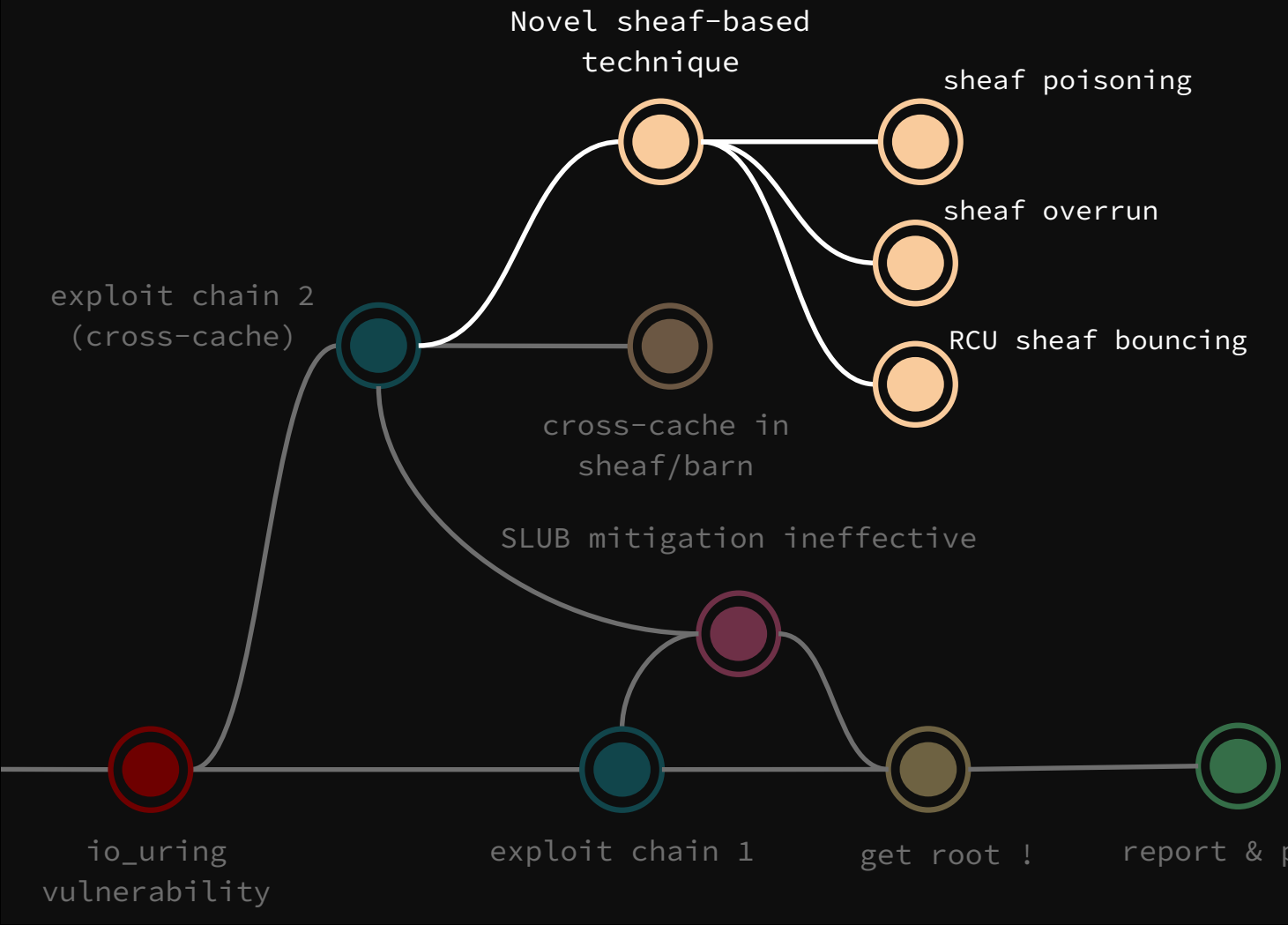

slab_sheaf object array
iovec freed object
Broken CONFIG_SLAB_FREELIST_HARDENED


exploit chain 2
(cross-cache)
cross-cache in
sheaf/barn
SLUB mitigation ineffective
io_uring
vulnerability
exploit chain 1
get root !
report & patch

Not only are SLUB’s existing mitigation weakened, but the mechanism itself also introduces some new techniques

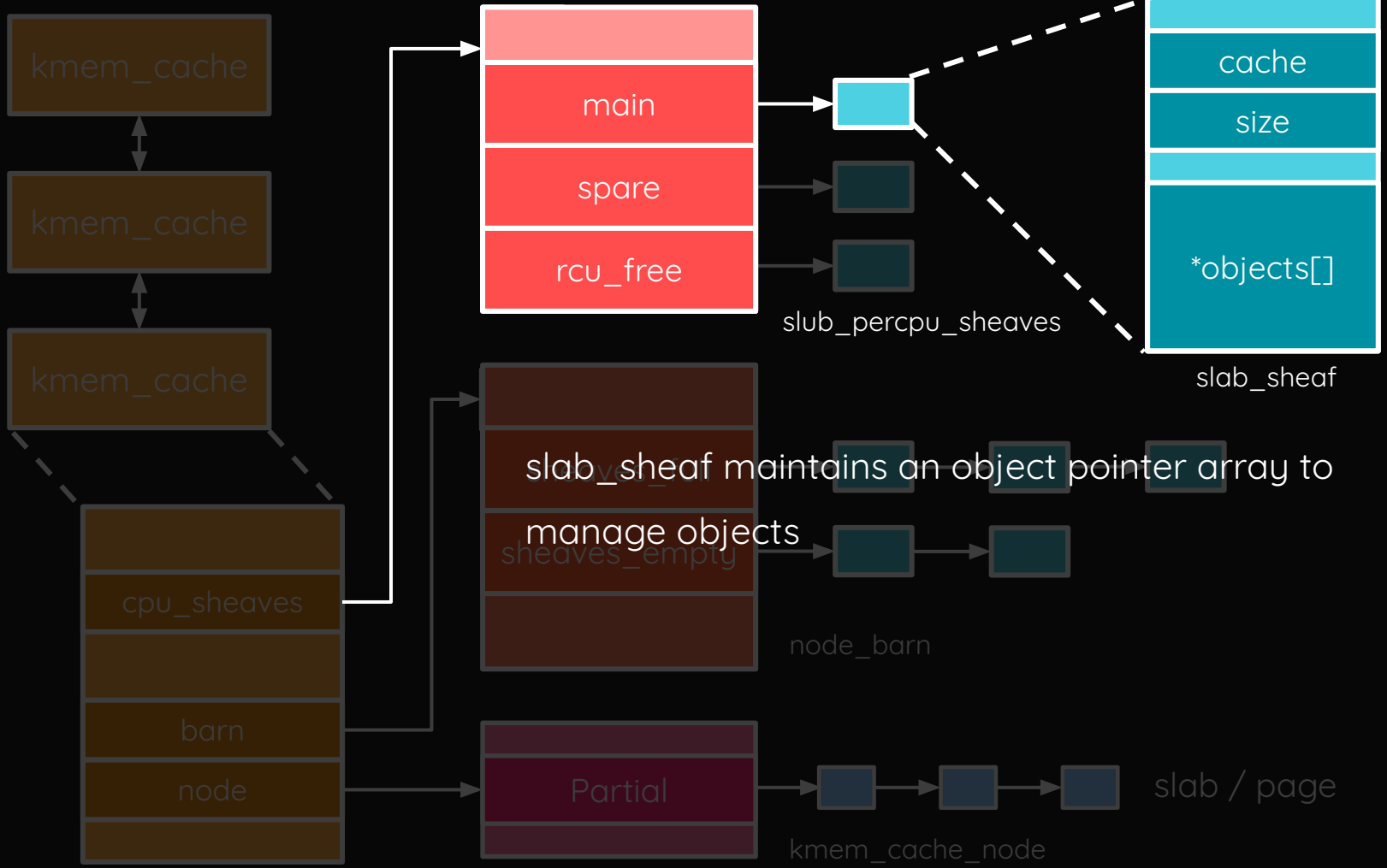
Novel Sheaf-Based Exploitation

New Exploitation Techniques based on Sheaf/Barn


Novel sheaf-based technique
sheaf poisoning
sheaf overrun
RCU sheaf bouncing
exploit chain 2 (cross-cache)
cross-cache in sheaf/barn
SLUB mitigation ineffective
io_uring vulnerability
exploit chain 1
get root !
report & patch

#1 sheaf poisoning

object array pointer overwrite


kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_sheaves
cache
size
*objects[]
slab_sheaf
slab_sheaf maintains an object pointer array to manage objects
sheaves_empty
node_barn
Partial
slab / page
kmem_cache_node



/mm/slub.c#L404 (v7.1.4)

```
struct slab_sheaf {
    union {
        struct rcu_head rcu_head;
        struct list_head barn_list;
        /* only used for prefilled sheafs */
        struct {
            unsigned int capacity;
            bool pfmemalloc;
        };
    };
    struct kmem_cache *cache;
    unsigned int size;
    int node; /* only used for rcu_sheaf */
    void *objects[];
};
```



/mm/slub.c#L404 (v7.1.4)

```
struct slab_sheaf {
    union {
        struct rcu_head rcu_head;
        struct list_head barn_list;
        /* only used for prefilled sheafs */
        unsigned int capacity;
        bool pfmemalloc;
        };
    };
    struct kmem_cache *cache;
    unsigned int size;
    int node; /* only used for rcu_sheaf */
    void *objects[];
};
```



/mm/slub.c#L404 (v7.1.4)

```
struct slab_sheaf {
    union {
        struct rcu_head rcu_head;
        struct list_head barn_list;
        /* only used for prefilled sheafs */
        struct {
            unsigned int capacity;
            bool pfmemalloc;
        };
    };
    struct kmem_cache *cache;
    int node; /* only used for rcu_sheaf */
    void *objects[];
};
```



/mm/slub.c#L404 (v7.1.4)

```
struct slab_sheaf {
    union {
        struct rcu_head rcu_head;
        struct list_head barn_list;
        /* only used for prefilled sheafs */
        struct {
            unsigned int capacity;
            bool pfmemalloc;
        };
    };
    unsigned int size;
    int node; /* only used for rcu_sheaf */
    void *objects[];
};
```



/mm/slub.c#L404 (v7.1.4)

```
struct slab_sheaf {
    union {
        struct rcu_head rcu_head;
        struct list_head barn_list;
        /* only used for prefilled sheafs */
        struct {
            unsigned int capacity;
            bool pfmemalloc;
        };
    };
    struct kmem_cache *cache;
    unsigned int size;
    void *objects[];
};
```



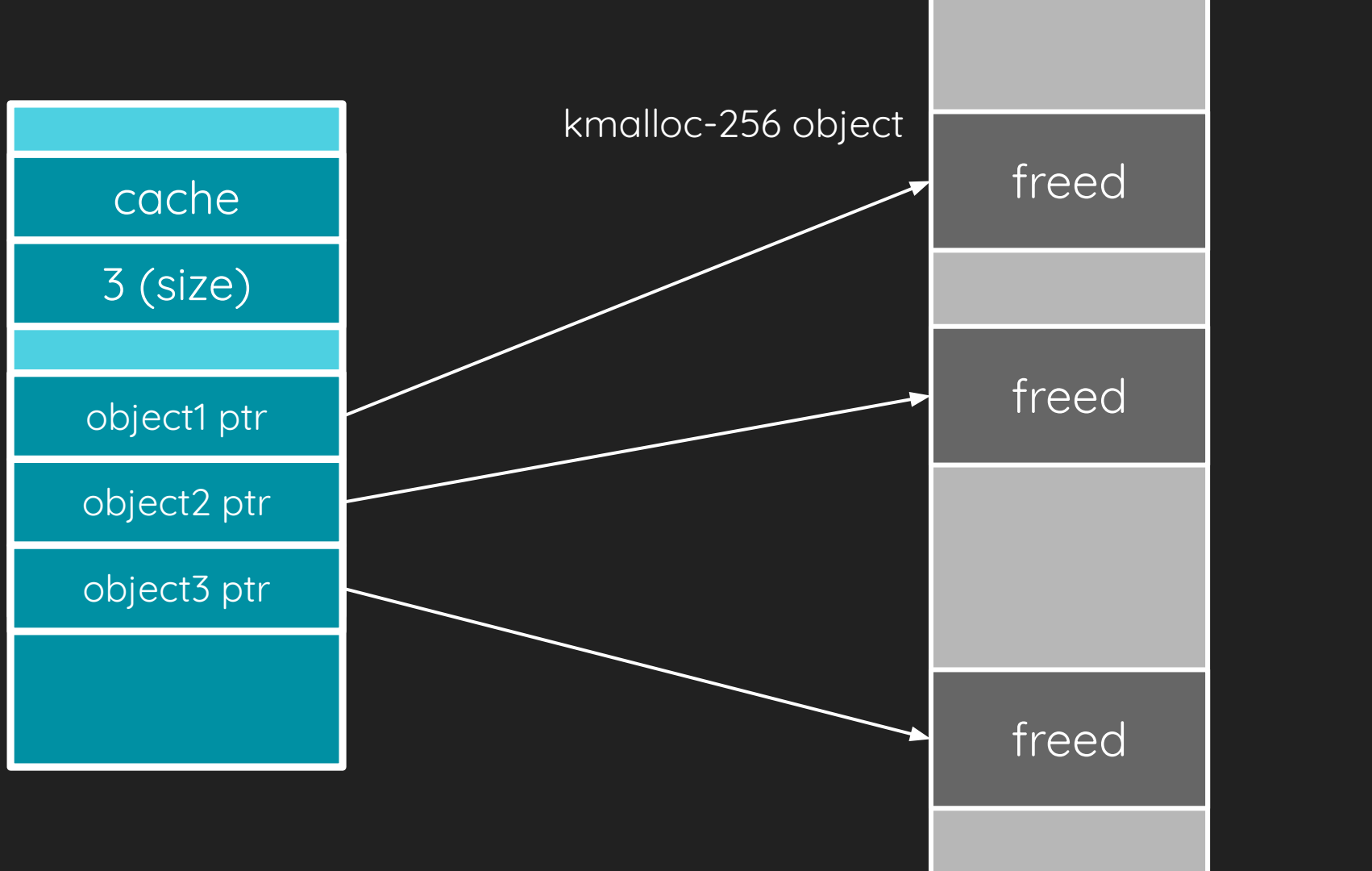
/mm/slub.c#L7741 (v7.1.4)

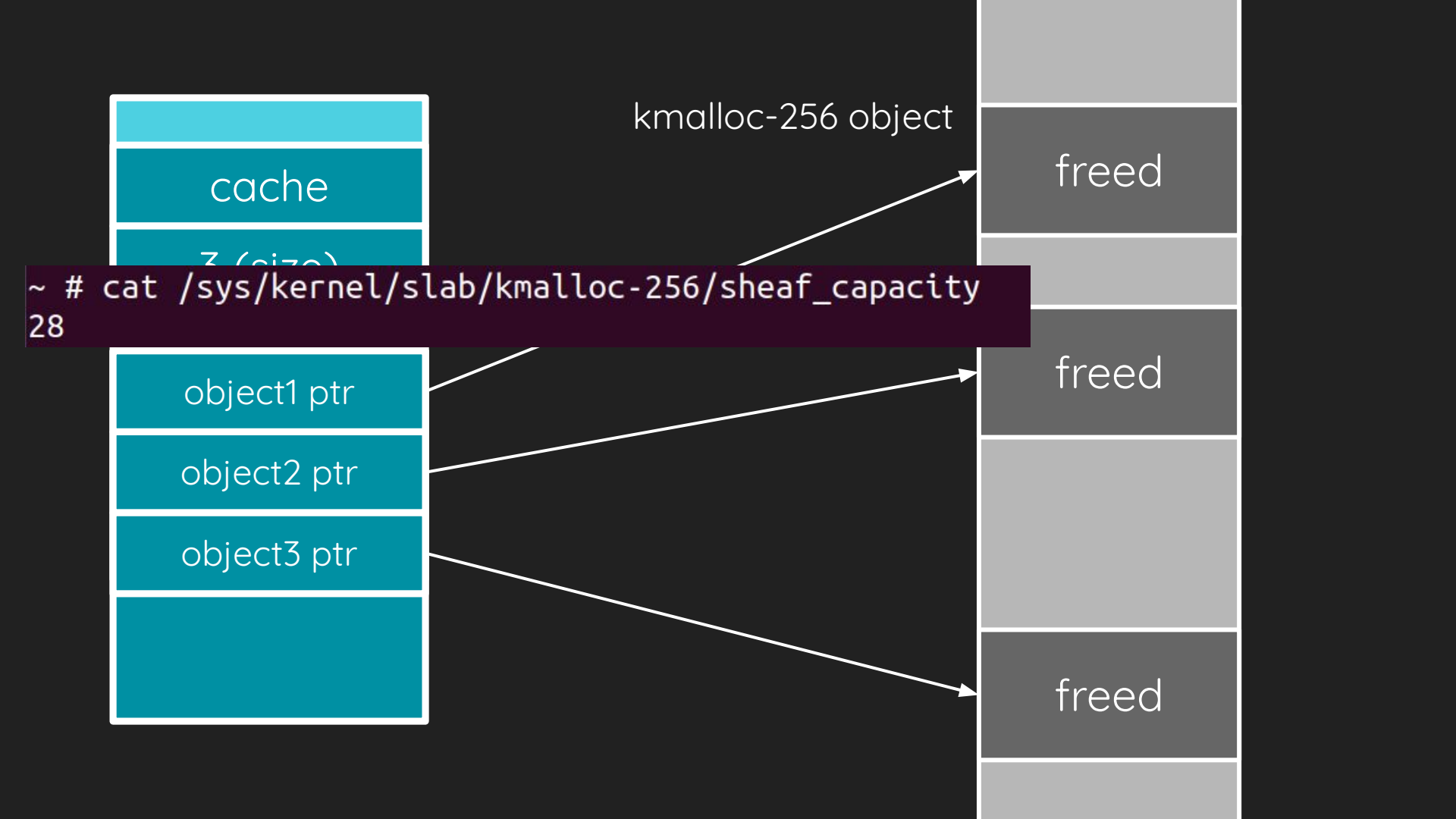
```
if (s->size >= PAGE_SIZE)
    capacity = 4;
else if (s->size >= 1024)
    capacity = 12;
else if (s->size >= 256)
    capacity = 26;
else
    capacity = 60;

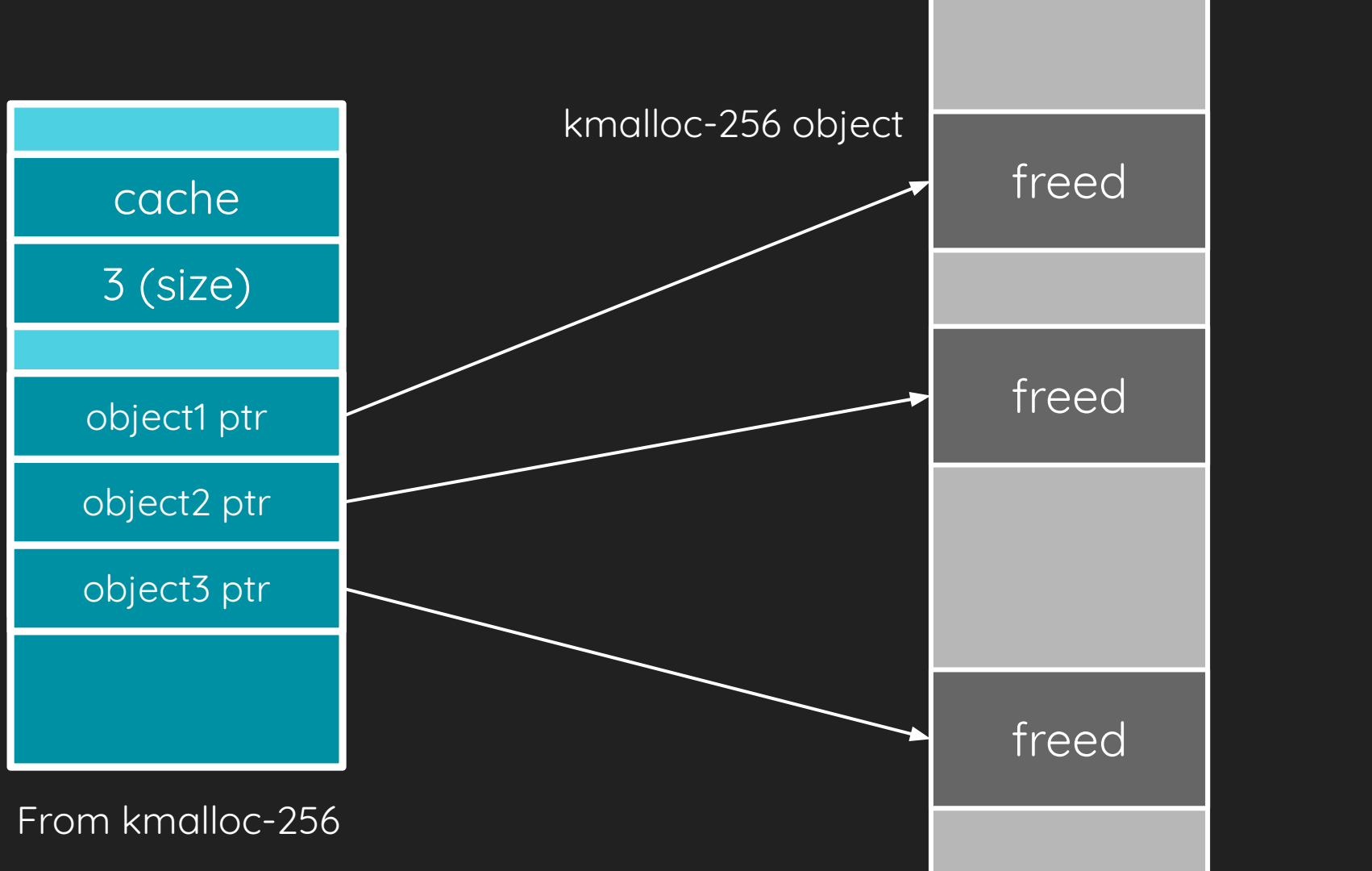
/* Increment capacity to make sheaf exactly a kmalloc size bucket */
size = struct_size_t(struct slab_sheaf, objects, capacity);
size = kmalloc_size_roundup(size);
capacity = (size - struct_size_t(struct slab_sheaf, objects, 0)) / sizeof(void *);
```

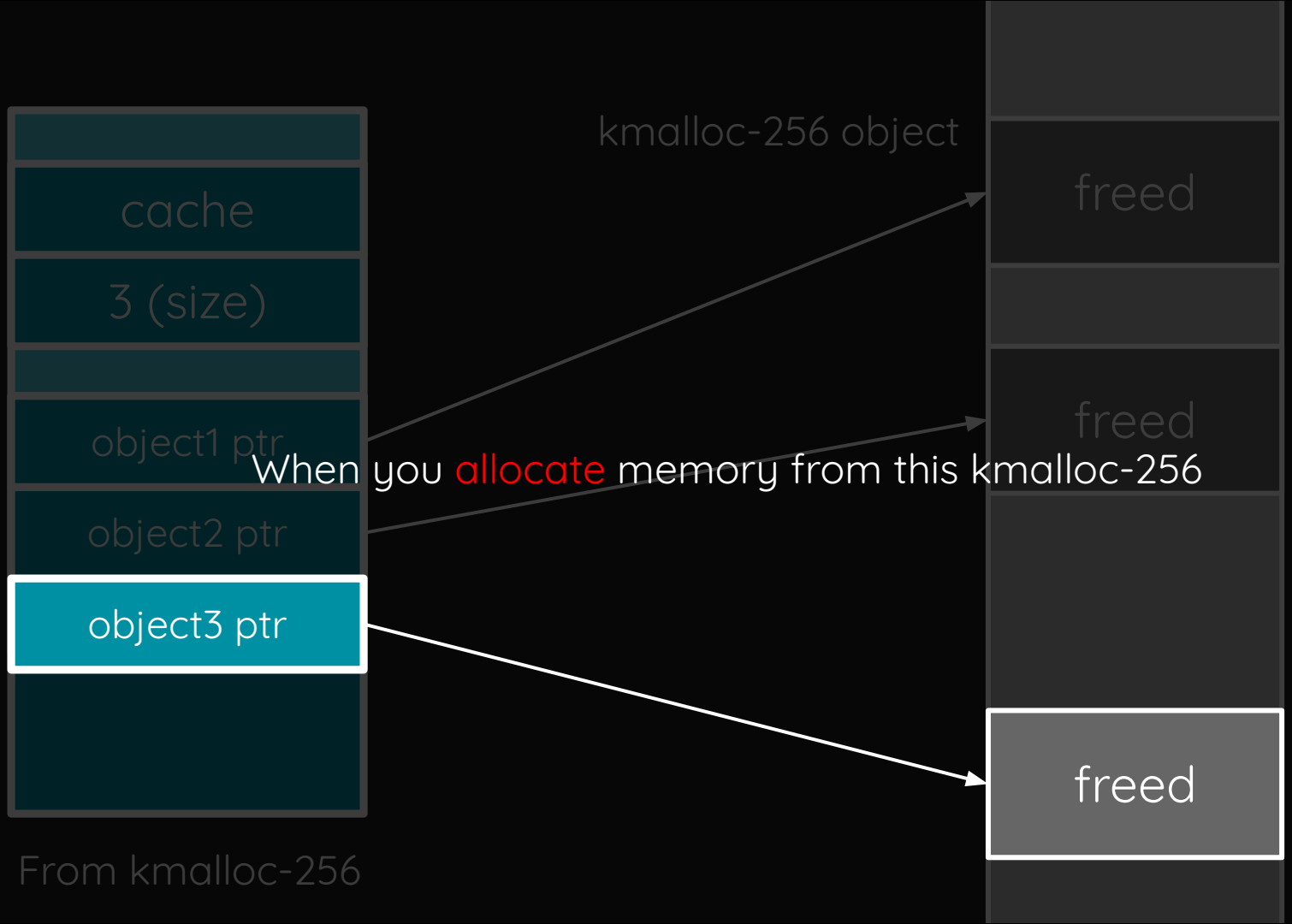
The slab_sheaf determines which memory pool an object comes from based on the object's size

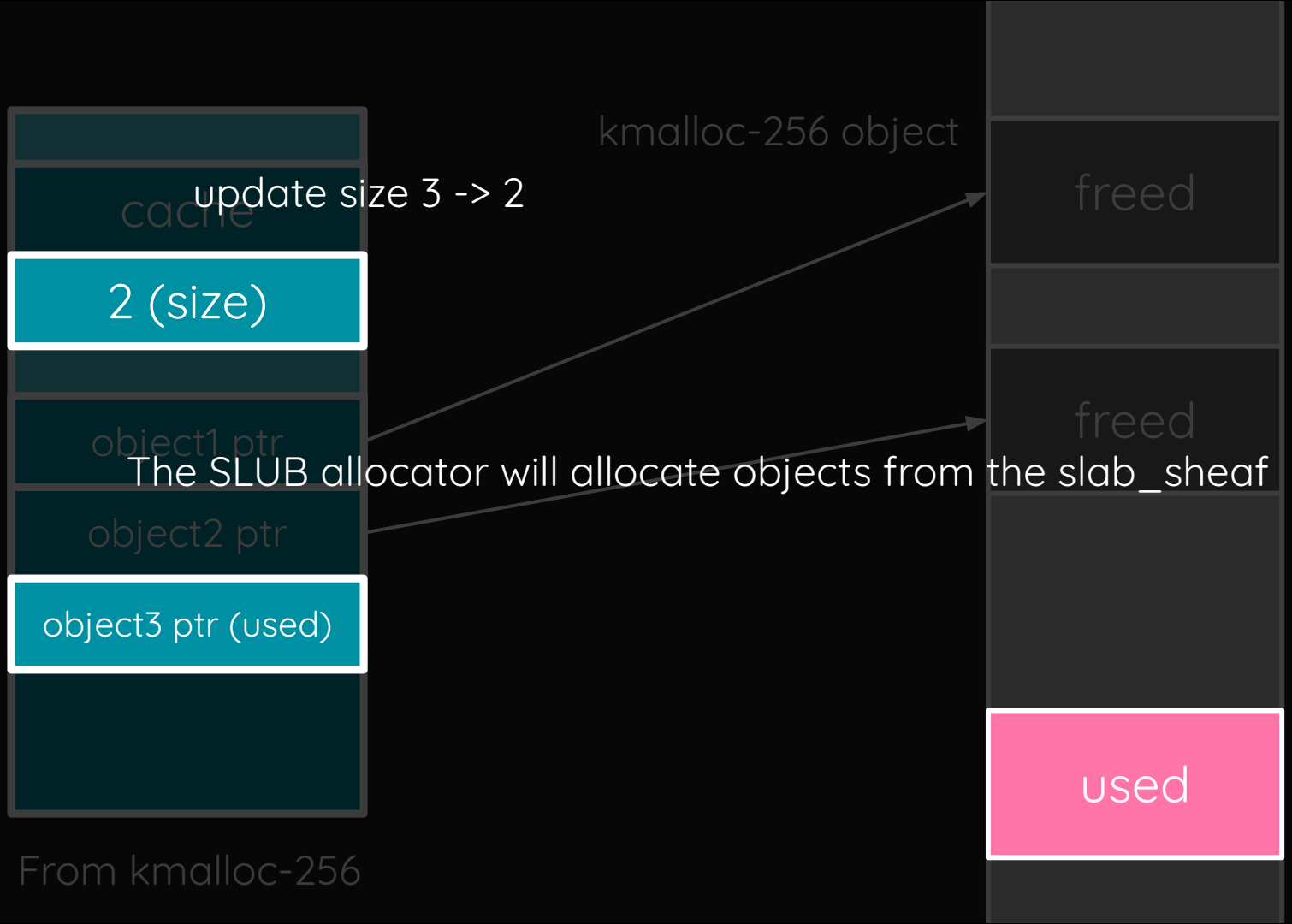
object size	Capacity	slab_sheaf kmem_cache
< 256	60	kmalloc-512
>= 256 & < 1024	26	kmalloc-256
>= 1024 & < PAGE_SIZE	12	kmalloc-128
>= PAGE_SIZE	4	kmalloc-64

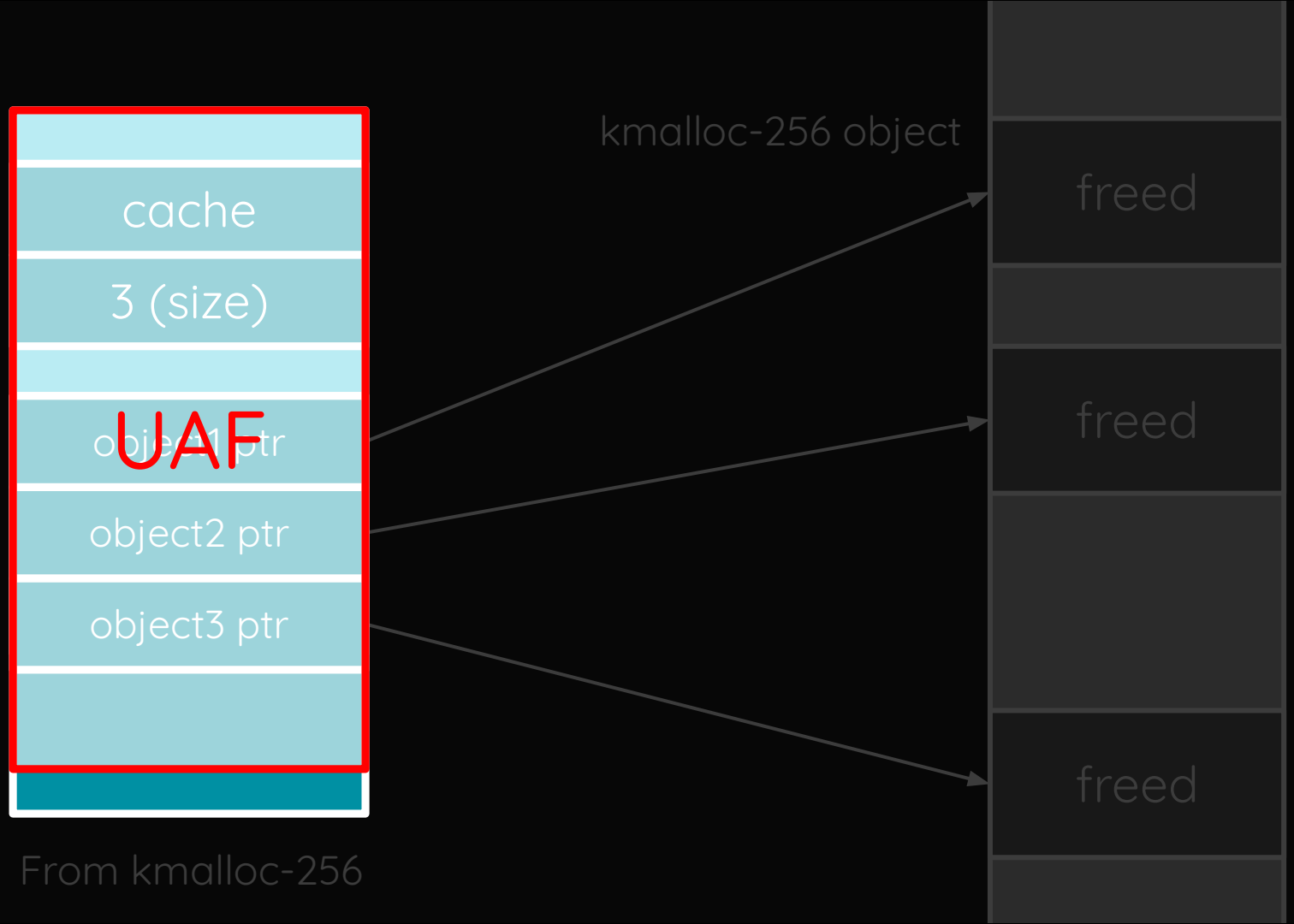

kmalloc-256 object
cache
3 (size)
object1 ptr
object2 ptr
object3 ptr
freed
freed
freed

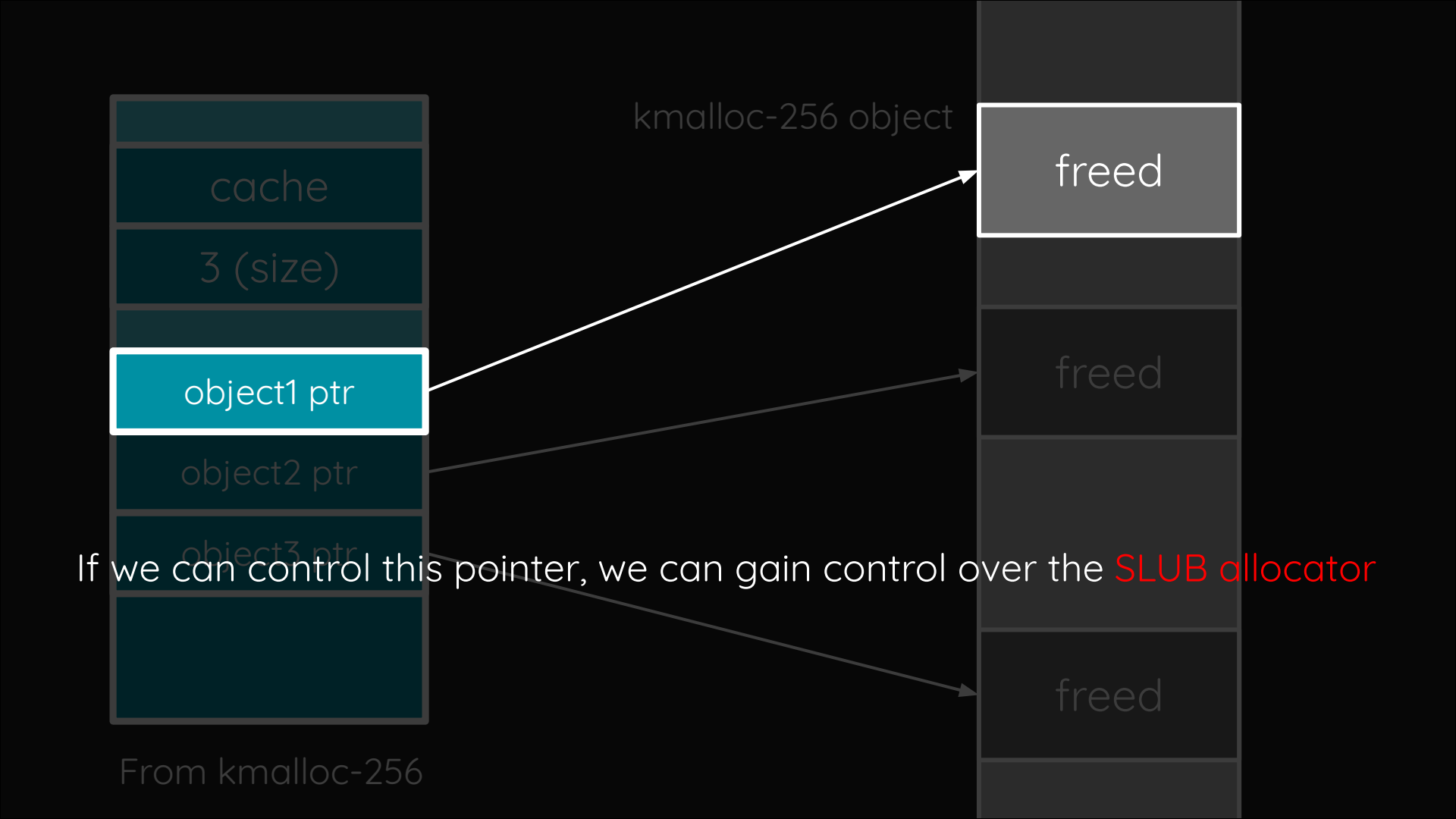

kmalloc-256 object
cache
~ # cat /sys/kernel/slab/kmalloc-256/sheaf_capacity
28
object1 ptr
object2 ptr
object3 ptr
freed
freed
freed

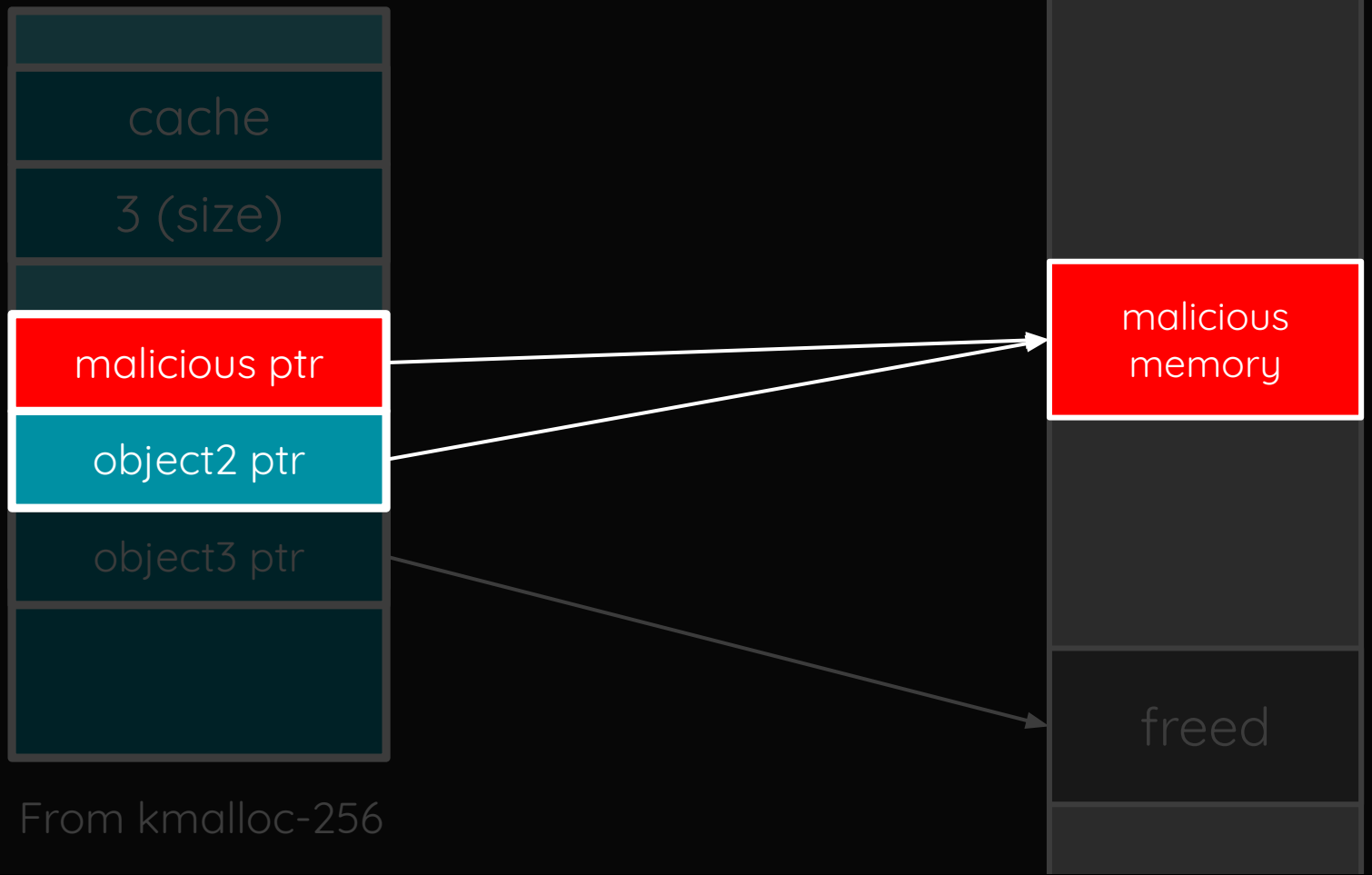

cache
3 (size)
object1 ptr
object2 ptr
object3 ptr
From kmalloc-256
kmalloc-256 object
freed
freed
freed


kmalloc-256 object
cache
3 (size)
object1 ptr
object2 ptr
object3 ptr
freed
freed
freed
When you allocate memory from this kmalloc-256
From kmalloc-256

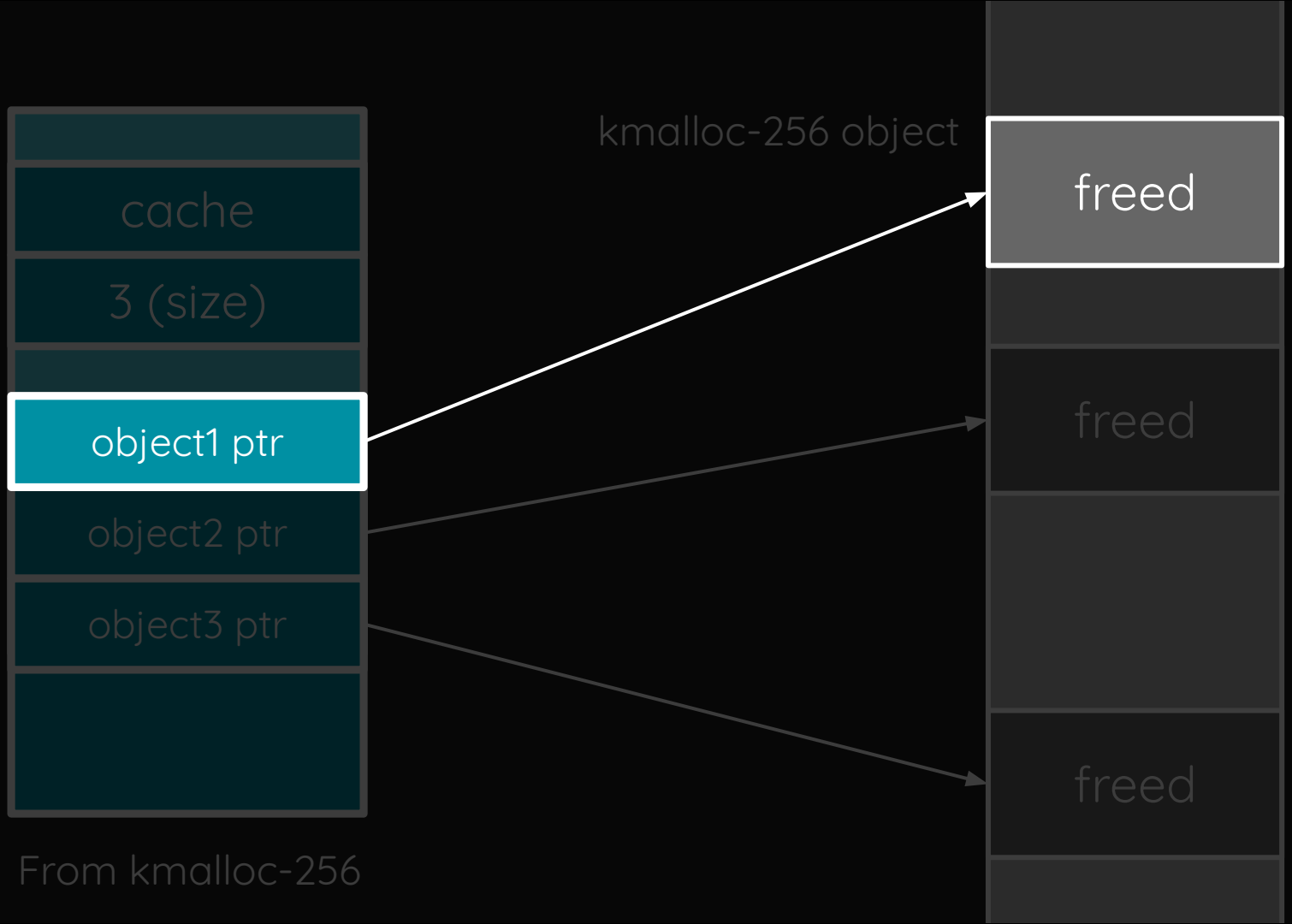

kmalloc-256 object
update size 3 -> 2
cache
2 (size)
object1 ptr
object2 ptr
object3 ptr (used)
freed
freed
The SLUB allocator will allocate objects from the slab_sheaf
used
From kmalloc-256

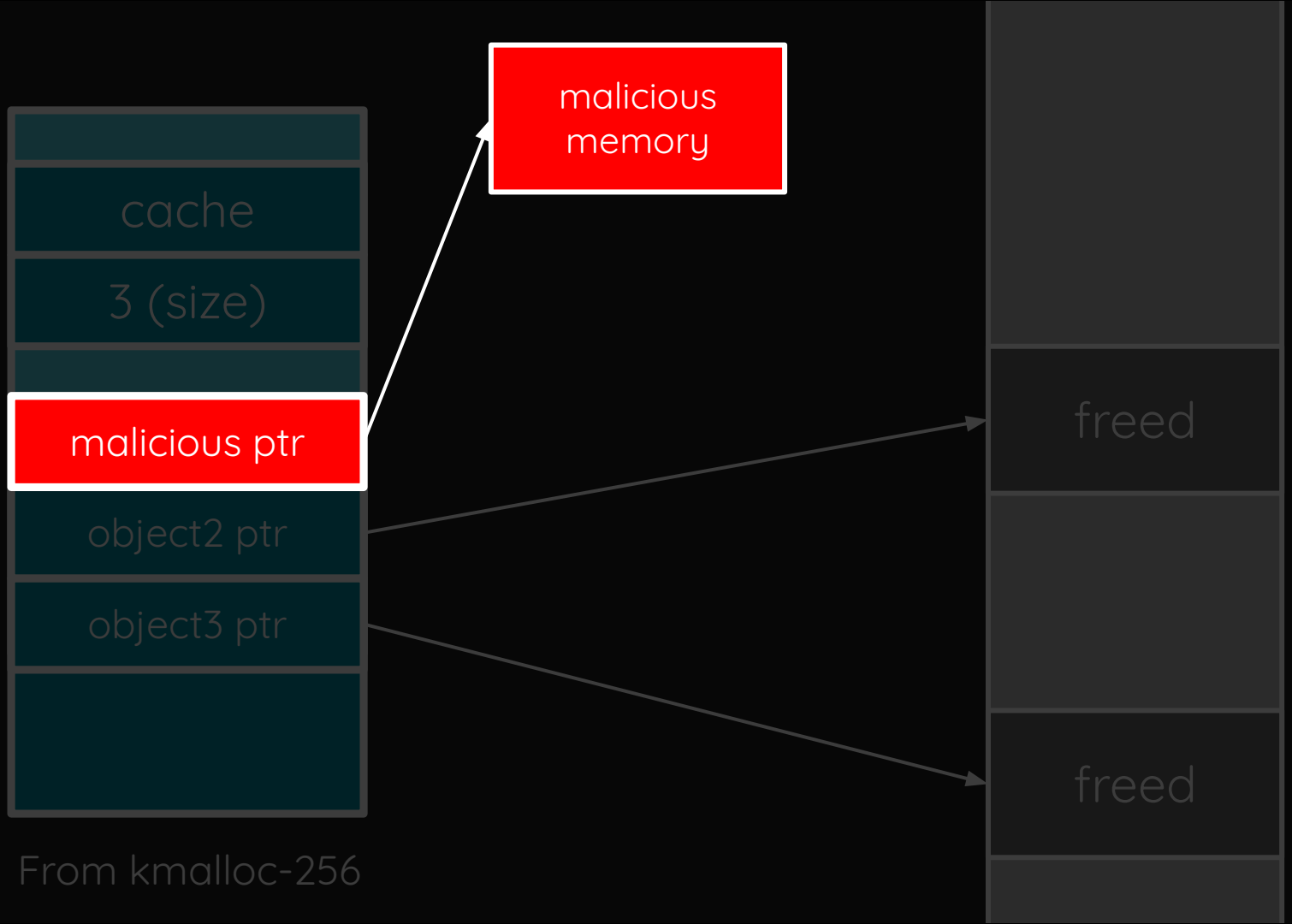

kmalloc-256 object
cache
3 (size)
UAF
object2 ptr
object3 ptr
freed
freed
freed
From kmalloc-256

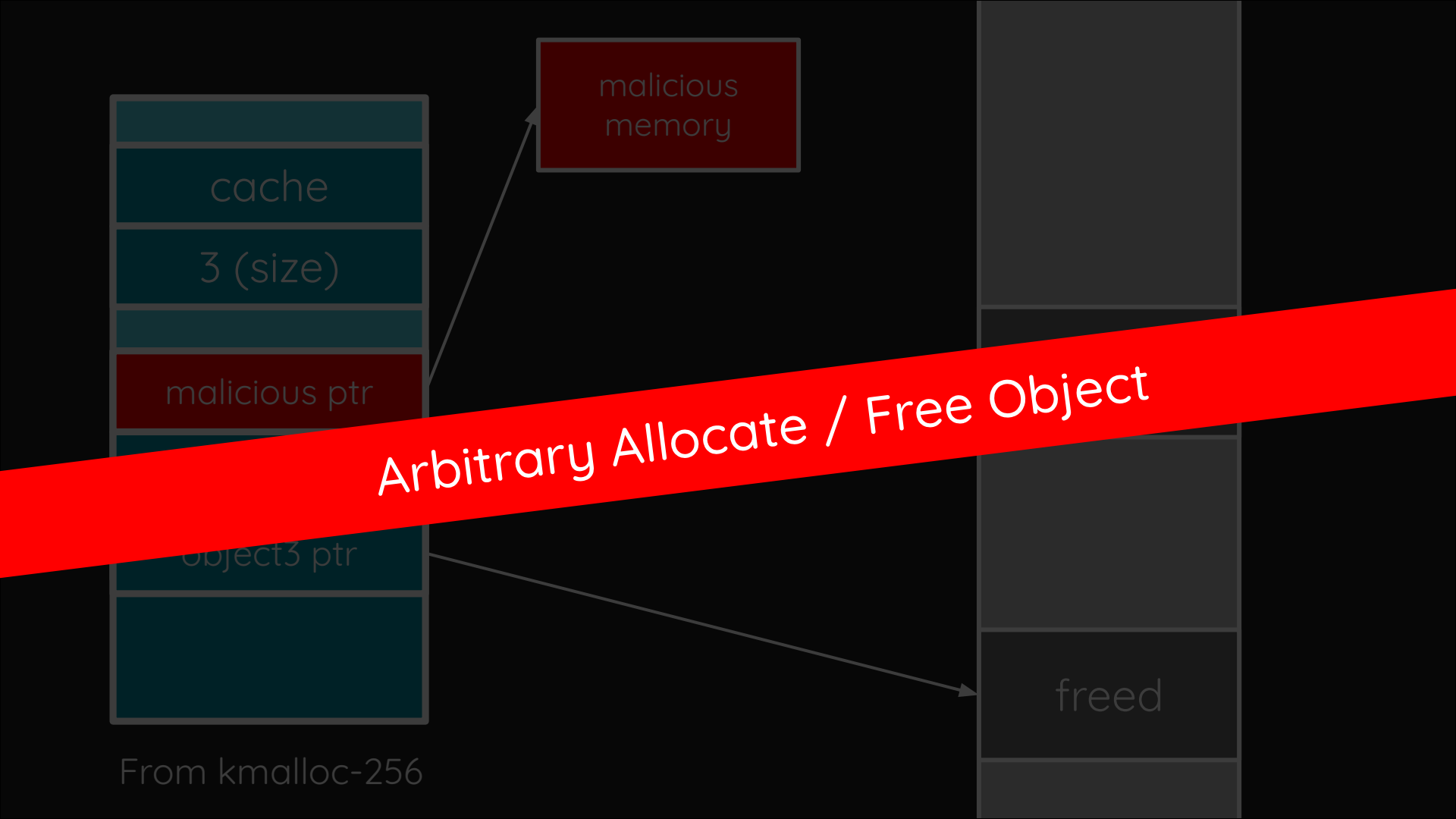

kmalloc-256 object
cache
3 (size)
object1 ptr
object2 ptr
object3 ptr
freed
freed
freed
From kmalloc-256
If we can control this pointer, we can gain control over the SLUB allocator


cache
3 (size)
malicious ptr
object2 ptr
object3 ptr
From kmalloc-256
malicious memory
freed


cache
3 (size)
malicious ptr
object3 ptr
freed
From kmalloc-256
Double Free


kmalloc-256 object
cache
3 (size)
object1 ptr
object2 ptr
object3 ptr
From kmalloc-256
freed
freed
freed


malicious memory
cache
3 (size)
malicious ptr
object2 ptr
object3 ptr
freed
freed
From kmalloc-256


malicious memory
cache
3 (size)
malicious ptr
Arbitrary Allocate / Free Object
object3 ptr
freed
From kmalloc-256

object size	Capacity	slab_sheaf kmem_cache
< 256	60	kmalloc-512
>= 256 & < 1024	28 / 32	kmalloc-256
>= 1024 & < PAGE_SIZE	12	kmalloc-128
>= PAGE_SIZE	4	kmalloc-64





object size	Capacity	slab_sheaf kmem_cache
< 256	60	kmalloc-512
>= 256 & < 1024	28 / 32	kmalloc-256
>= 1024 & < PAGE_SIZE	12	kmalloc-128
>= PAGE_SIZE	4	kmalloc-64

kmalloc-256 UAF = Control all object >=256 and < 1024

object size	Capacity	slab_sheaf kmem_cache
< 256	60	kmalloc-512
>= 256 & < 1024	28 / 32	kmalloc-256
>= 1024 & < PAGE_SIZE	12	kmalloc-128
>= PAGE_SIZE	4	kmalloc-64

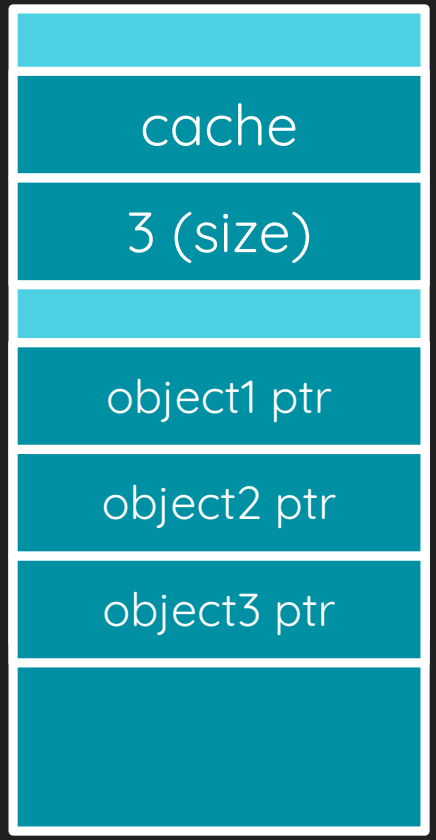
object size	Capacity	slab_sheaf kmem_cache
< 256	60	kmalloc-512
>= 256 & < 1024	28 / 32	kmalloc-256
>= 1024 & < PAGE_SIZE	12	kmalloc-128
>= PAGE_SIZE	4	kmalloc-64

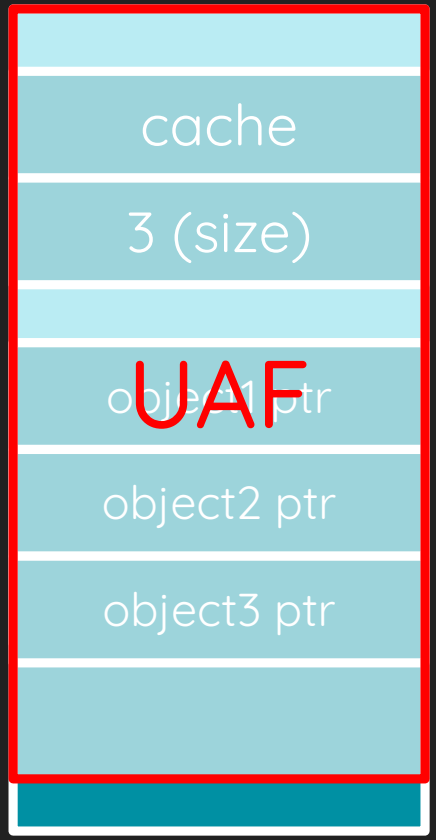
object size	Capacity	slab_sheaf kmem_cache
< 256	60	kmalloc-512
>= 256 & < 1024	28 / 32	
& < PAGE_SIZE	12	kmalloc-128
>= PAGE_SIZE	4	kmalloc-64

Compromised more kmem_cache

#2 sheaf Overrun

object size hijack to out-of-bound write pointer


cache
3 (size)
object1 ptr
object2 ptr
object3 ptr


cache
3 (size)
UAF
object1 ptr
object2 ptr
object3 ptr



/mm/slub.c#L5806 (v7.1.4)

```
static __fastpath_inline
bool free_to_pcs(struct kmem_cache *s, void *object, bool allow_spin)
{
    struct slub_percpu_sheaves *pcs;
    ...
    if (unlikely(pcs->main->size == s->sheaf_capacity)) {
    ...
    }

    pcs->main->objects[pcs->main->size++] = object;
```



/mm/slub.c#L5806 (v7.1.4)

```
static __fastpath_inline
bool free_to_pcs(struct kmem_cache *s, void *object, bool allow_spin)
{
    struct slub_percpu_sheaves *pcs;
    ...
    if (unlikely(pcs->main->size == s->sheaf_capacity)) {
    ...
    }
```

```
pcs->main->objects[pcs->main->size++] = object;
```



/mm/slub.c#L5806 (v7.1.4)

```
static __fastpath_inline
bool free_to_pcs(struct kmem_cache *s, void *object, bool allow_spin)
{
    struct slub_percpu_sheaves *pcs;
    ...
    if (unlikely(pcs->main->size == s->sheaf_capacity)) {
    ...
    }

    pcs->main->objects[pcs->main->size++] = object;
```



/mm/slub.c#L5806 (v7.1.4)

```
static __fastpath_inline
bool free_to_pcs(struct kmem_cache *s, void *object, bool allow_spin)
{
    if (unlikely(pcs->main->size == s->sheaf_capacity)) {
    ...
    }

    pcs->main->objects[pcs->main->size++] = object;
```

PASS

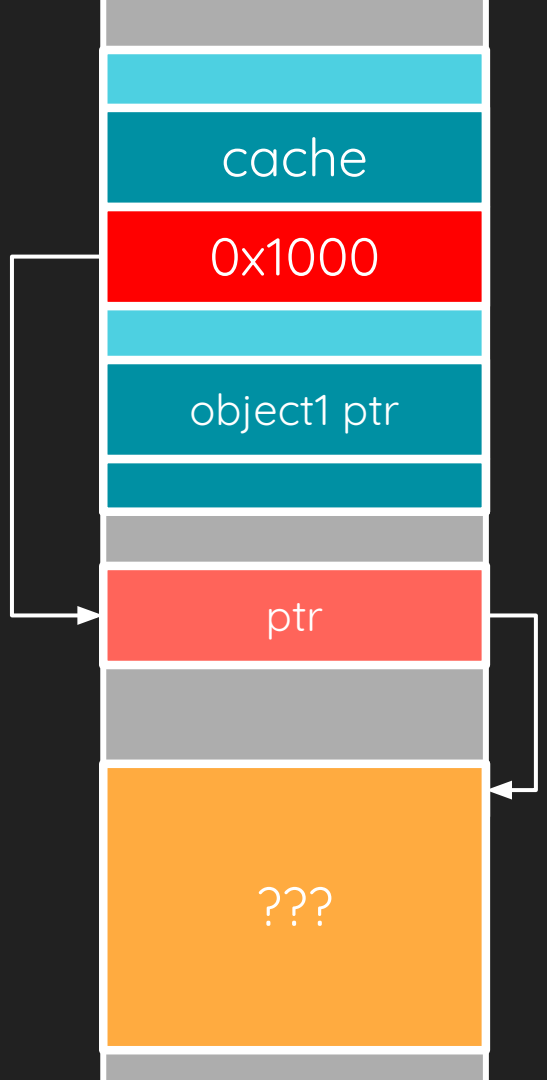


/mm/slub.c#L5806 (v7.1.4)

```
static __fastpath_inline
bool free_to_pcs(struct kmem_cache *s, void *object, bool allow_spin)
{
    struct slub_percpu_sheaves *pcs;
    ...
    if (unlikely(pcs->main->size == s->sheaf_capacity)) {
    ...
```

```
pcs->main->objects[pcs->main->size++] = object;
```

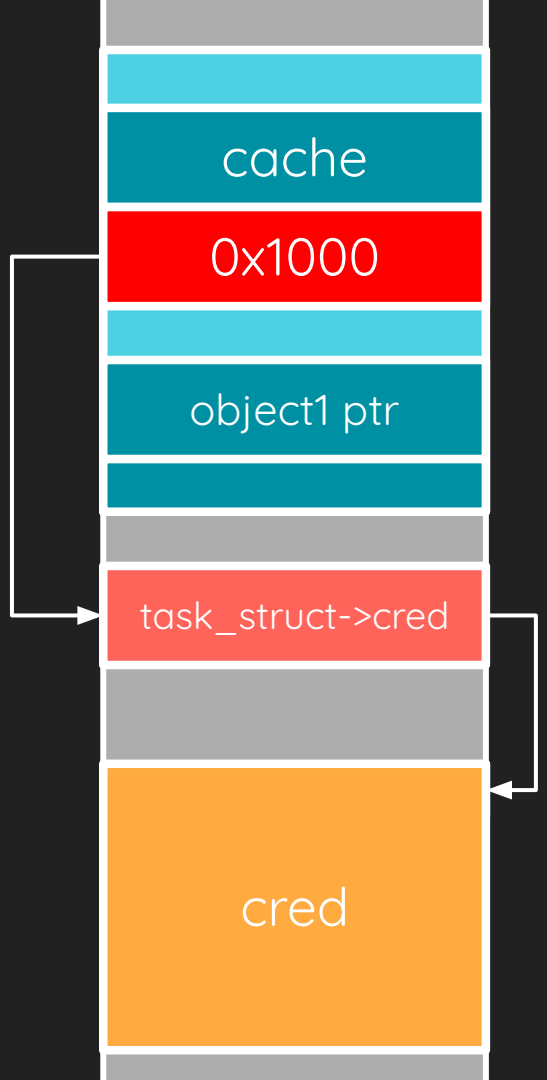
Allocate



Exploitable kernel structs (ex: cred, file, etc.)

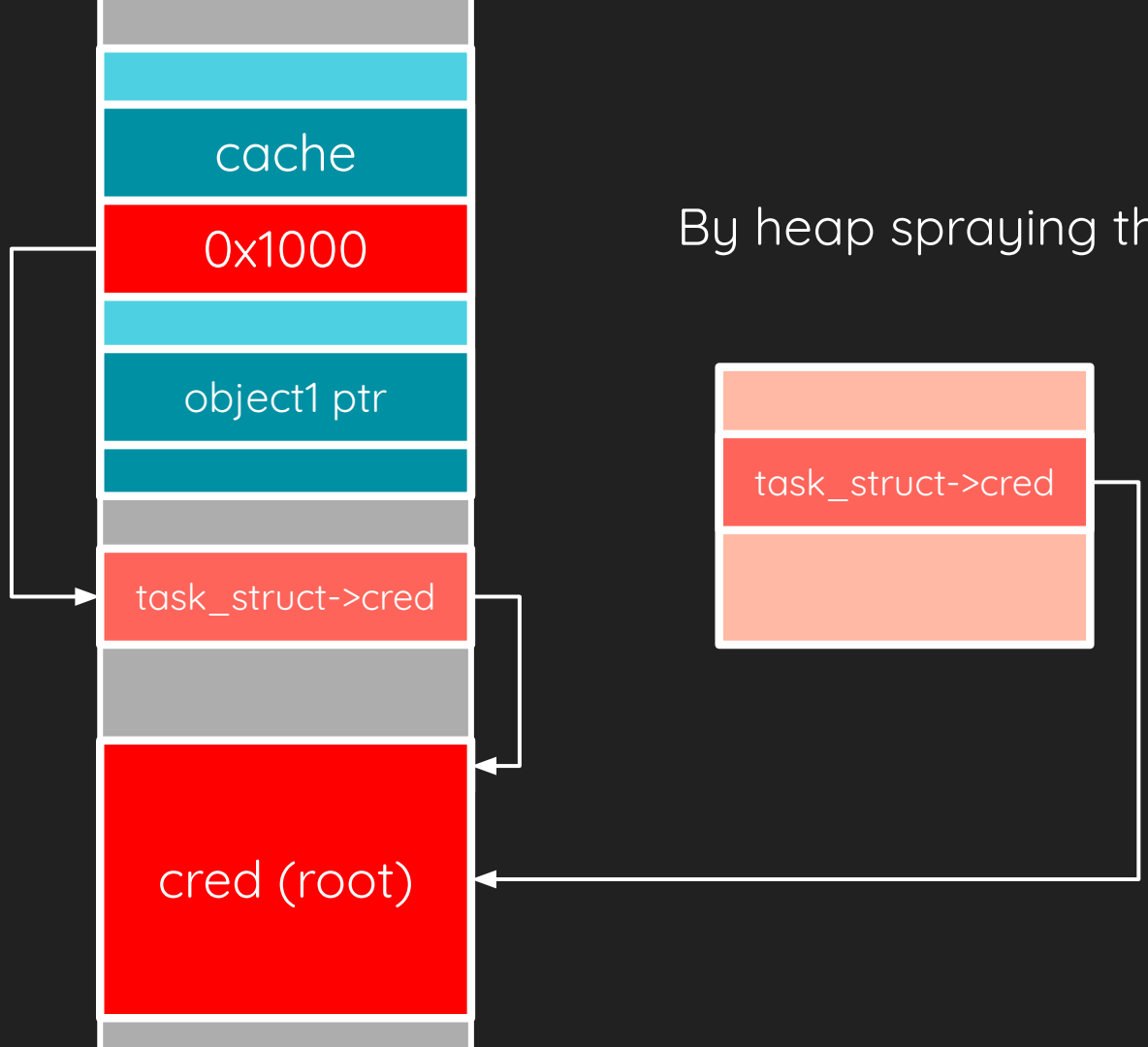
Allocate

By heap spraying the root cred struct



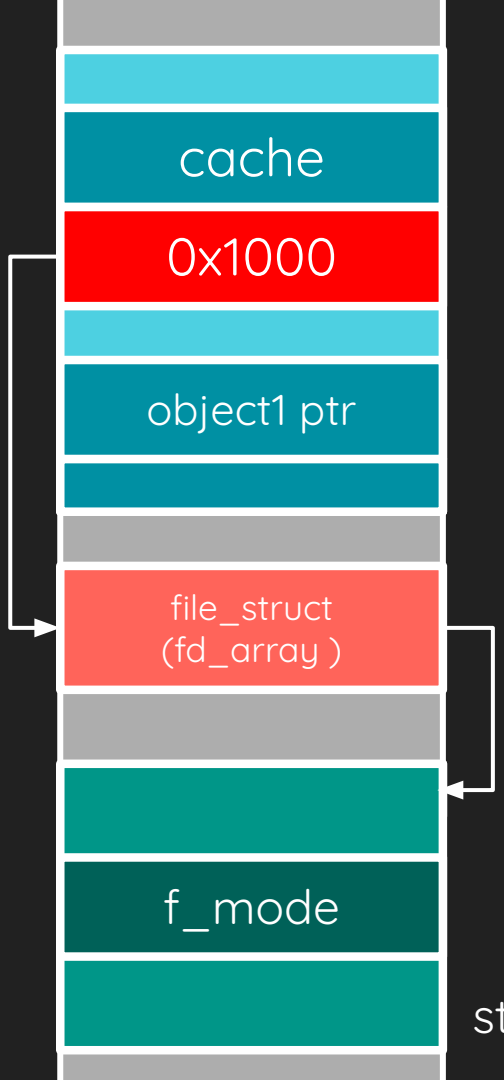
Allocate

By heap spraying the root cred struct



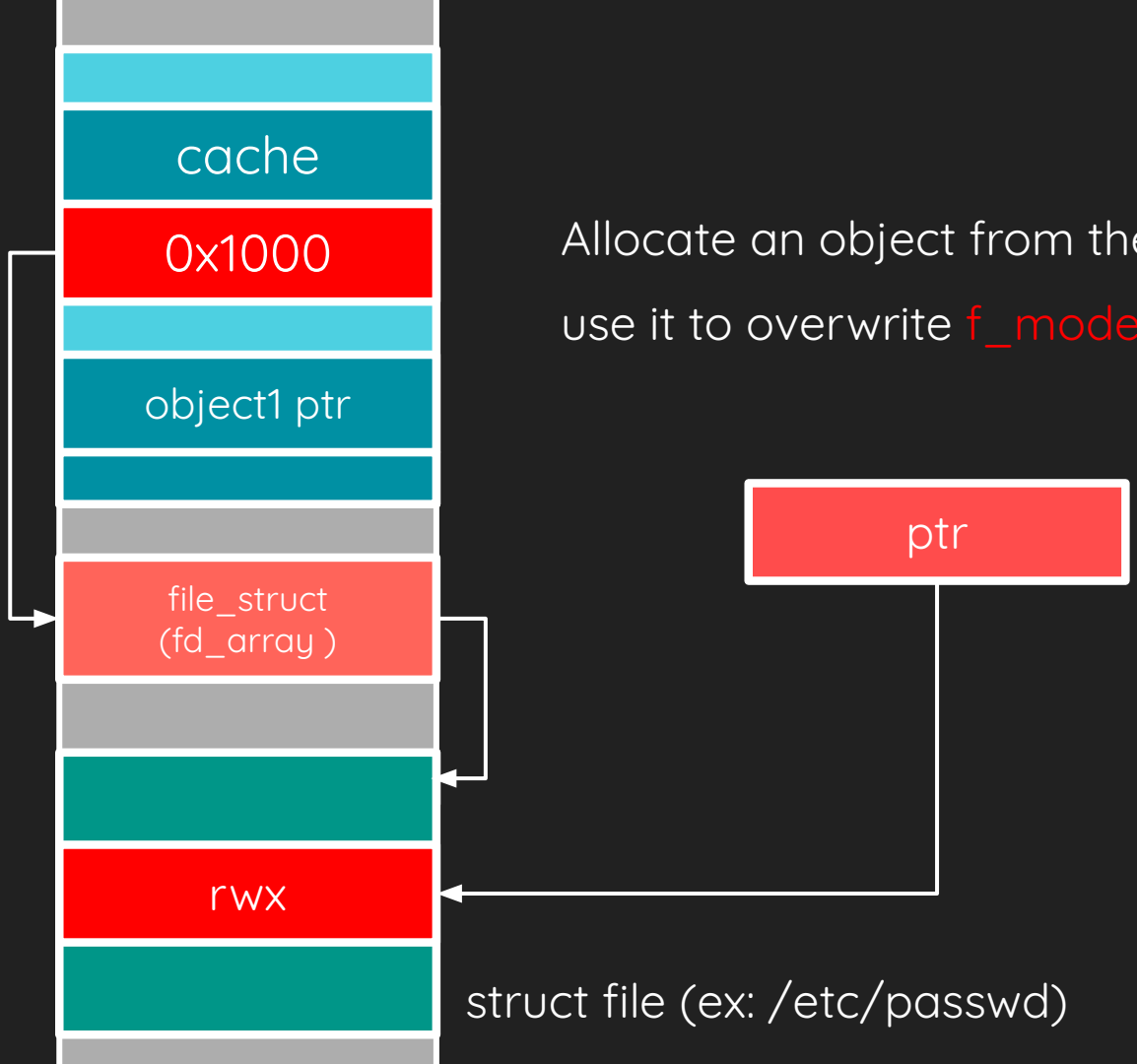

Allocate
cache
0x1000
object1 ptr
By heap spraying the root cred struct
task_struct
Privilege escalation
cred (root)

Allocate

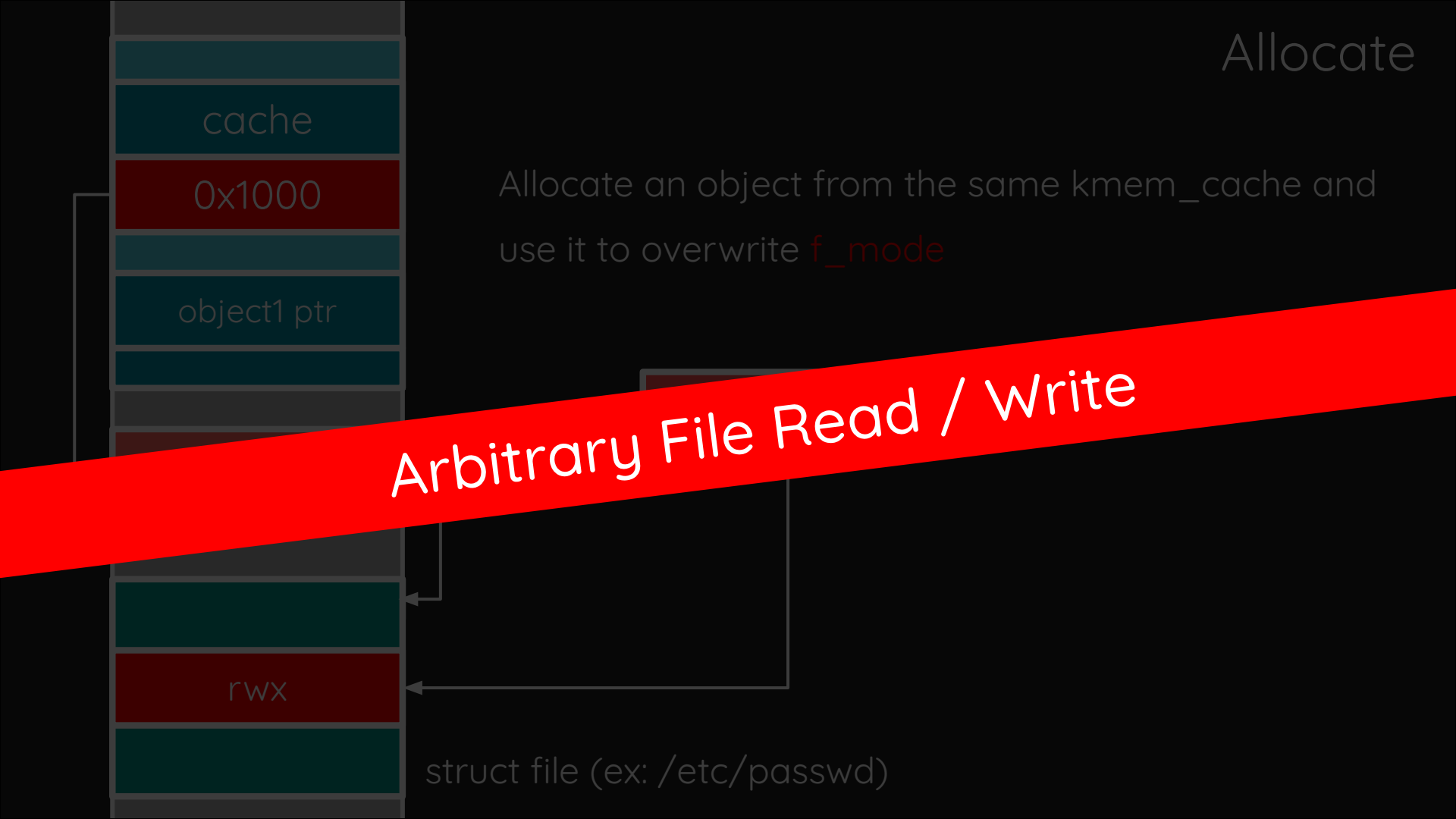


Allocate an object from the same kmem_cache and use it to overwrite f_mode

Allocate

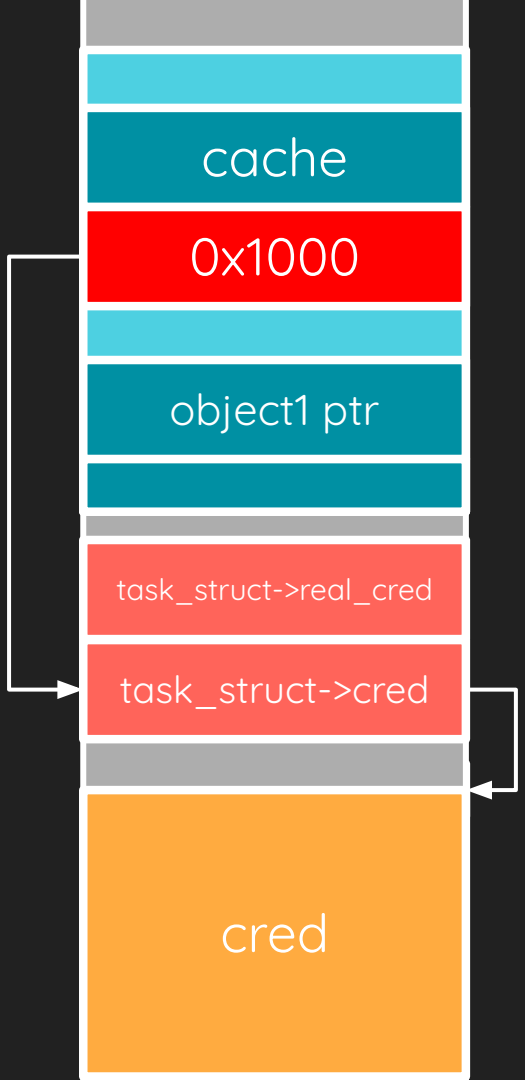


Allocate an object from the same kmem_cache and use it to overwrite f_mode

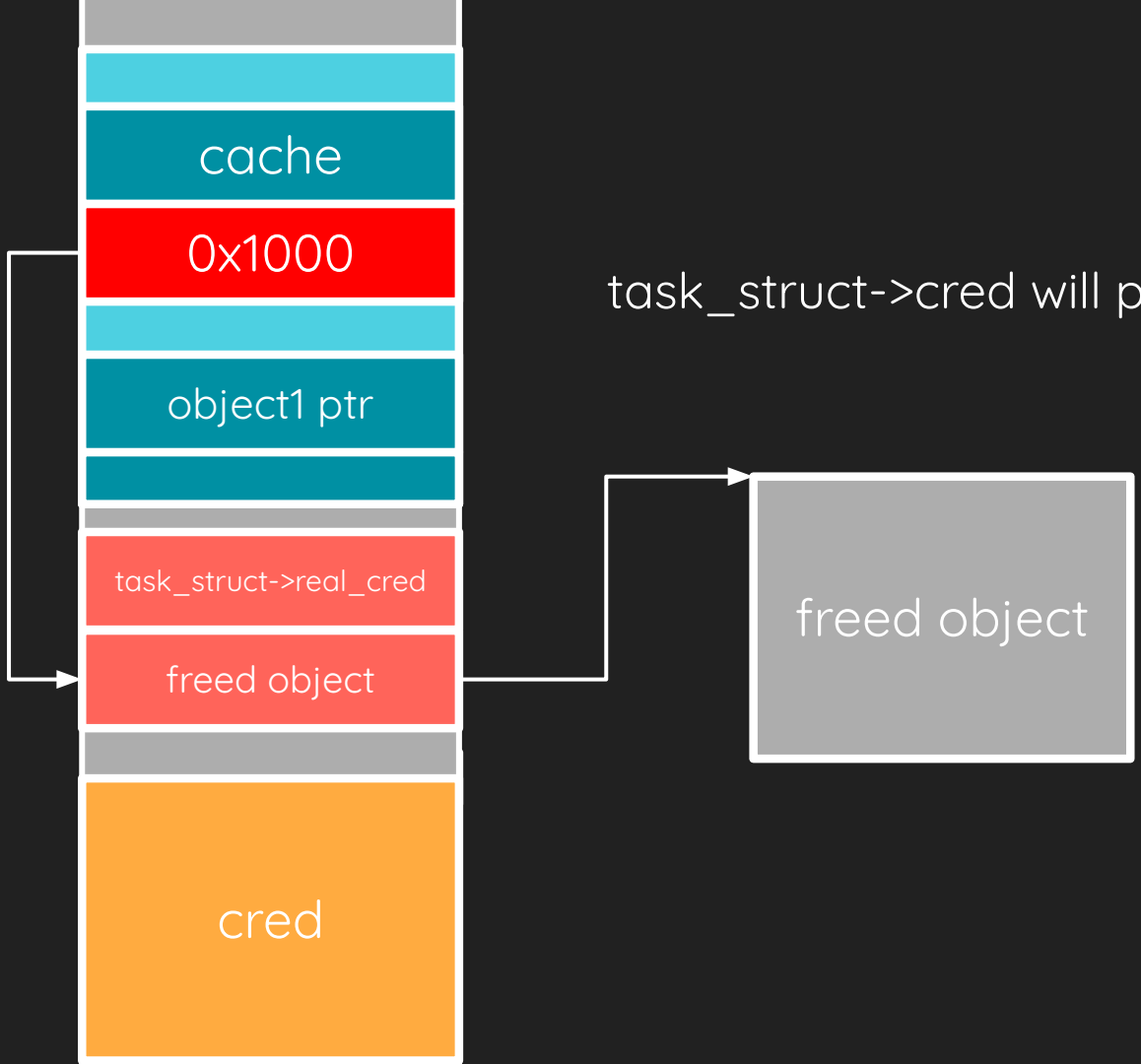

Allocate
cache
0x1000
object1 ptr
Allocate an object from the same kmem_cache and
use it to overwrite f_mode
Arbitrary File Read / Write
rwx
struct file (ex: /etc/passwd)

Free

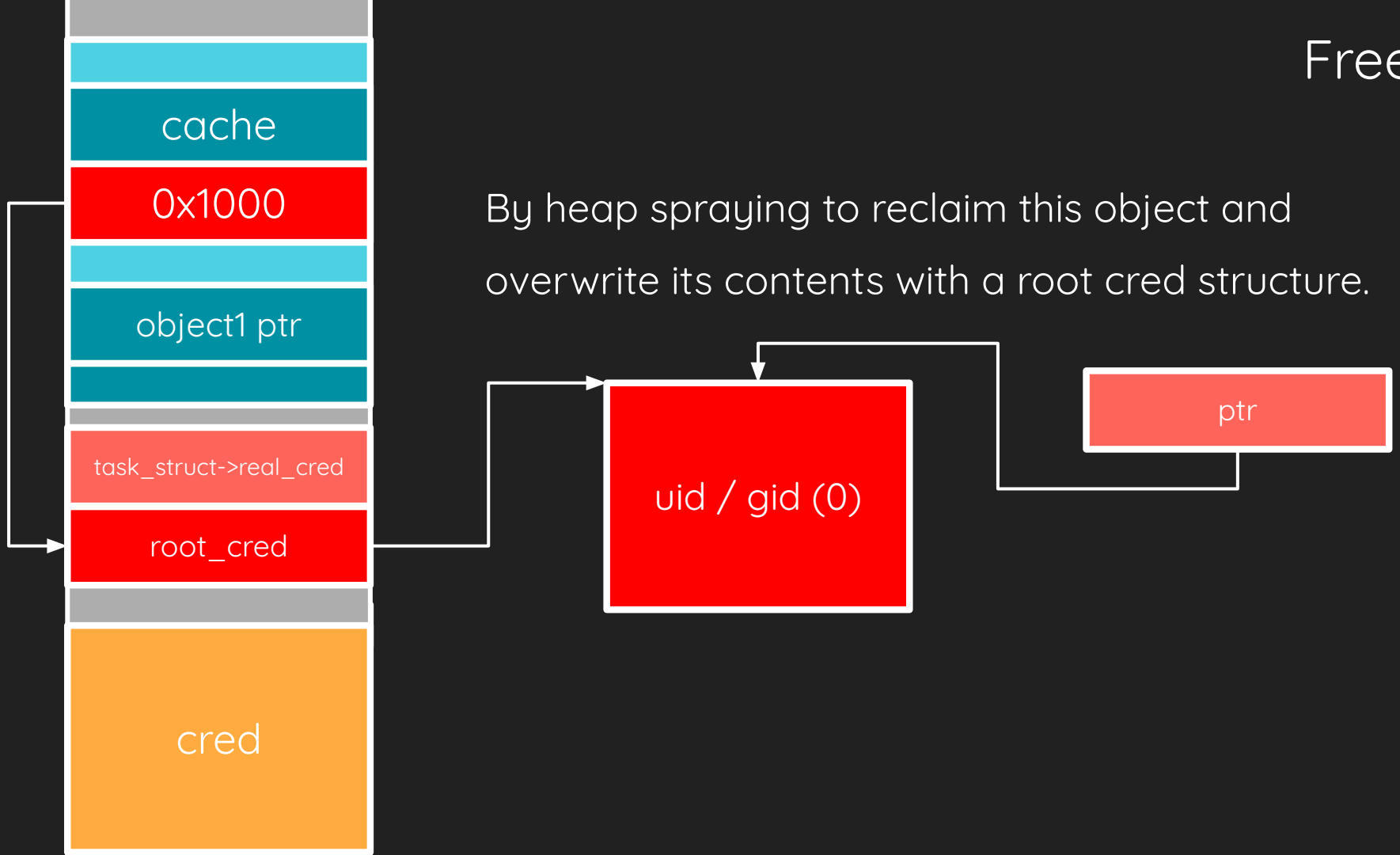
When we free an object



Free



task_struct->cred will point to a freed object


cache
0x1000
object1 ptr
task_struct->real_cred
root_cred
cred
By heap spraying to reclaim this object and
overwrite its contents with a root cred structure.
uid / gid (0)
ptr

Free

cache

0x1000

object1 ptr

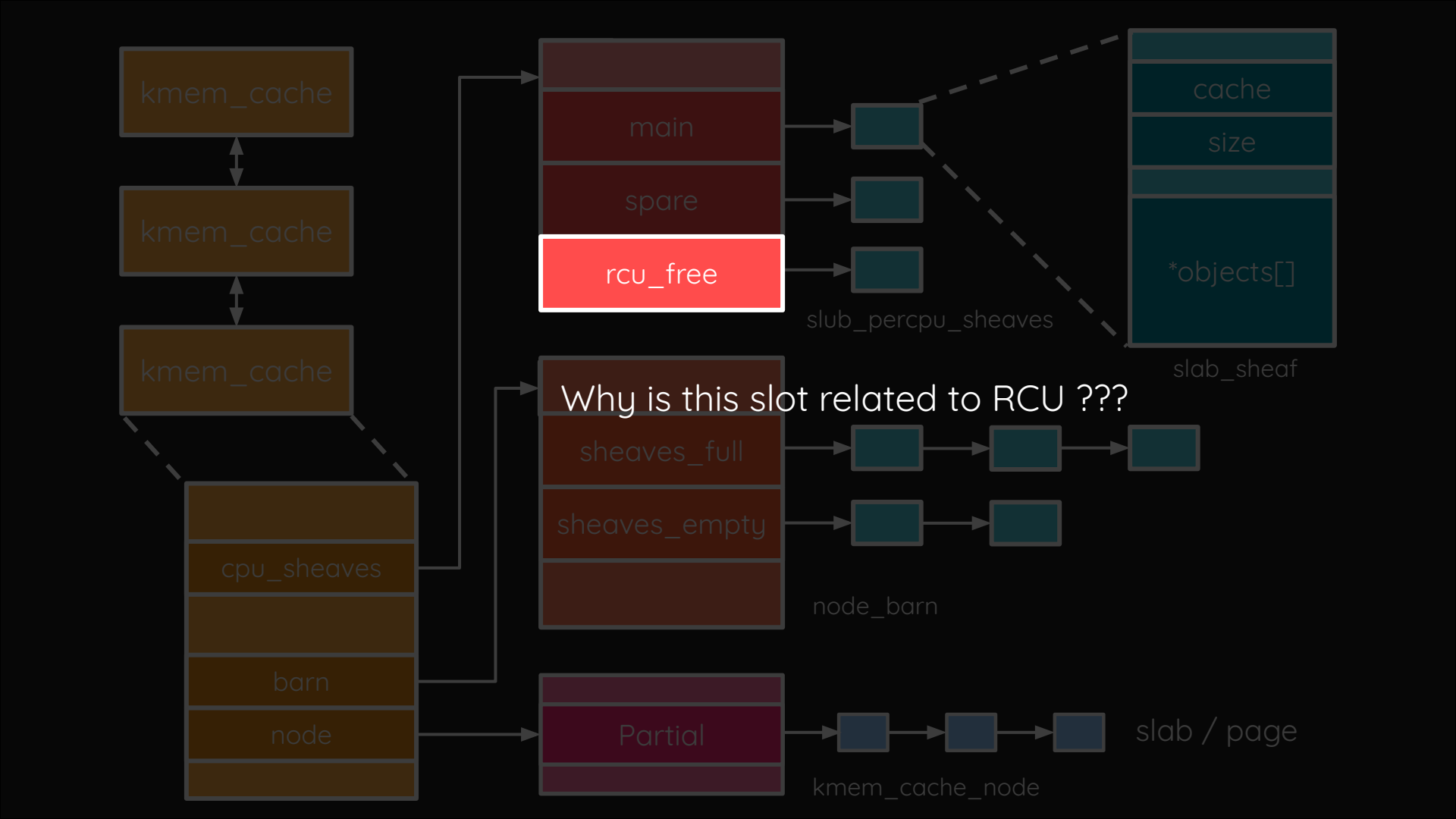
cred

By heap spraying to reclaim this object and overwrite its contents with a root cred structure.

Privilege escalation

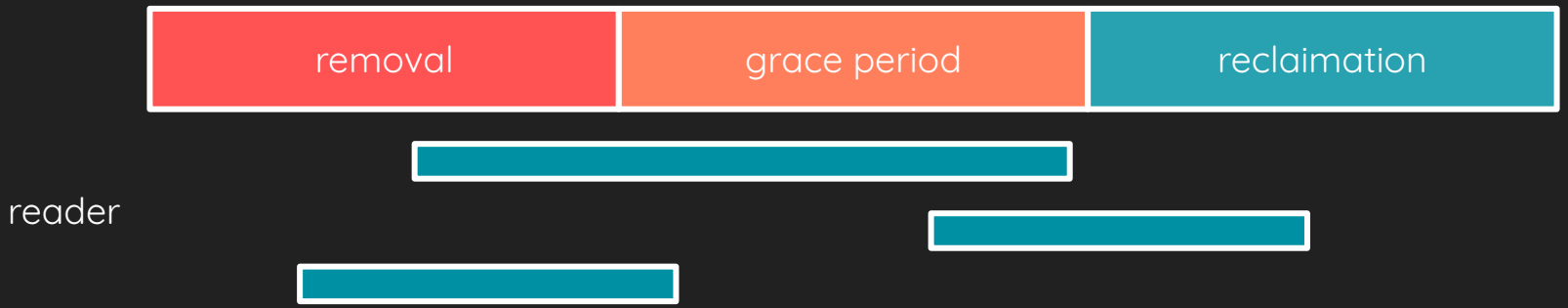
#3 RCU sheaf Bouncing

From poisoning rcu_sheaf to arbitrary cross-cache


kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
slub_percpu_sheaves
cache
size
*objects[]
slab_sheaf
Why is this slot related to RCU ???
sheaves_full
sheaves_empty
node_barn
Partial
kmem_cache_node
slab / page

RCU (Read-Copy-Update)

- Multiple readers and one writer can access simultaneously
- RCU splits updates into two stages:
  - Copy & Update : Copy and modify the data, then atomically switch the pointer to it
  - Grace Period : Wait for all old readers to finish before freeing the old data



Why was rcu sheaf introduced ?

> Maple Tree and similar data structures make extensive use of kfree_rcu to free objects

> High-frequency freeing or RCU_NOCB offloading (ex: Android) can cause per-cpu object array overflow and repeated flush/refill

> Batch RCU-deferred freeing of multiple objects into a single call_rcu()

What's next for the SLUB allocator

LWN.net needs you!

Without subscribers, LWN would simply not exist. Please consider signing up for a subscription and helping to keep LWN publishing.

By **Jonathan Corbet**
May 20, 2024

LSFMM+BPF

There are two fundamental levels of memory allocator in the Linux kernel: the page allocator, which allocates memory in units of pages, and the slab allocator, which allocates arbitrarily-sized chunks that are usually (but not necessarily) smaller than a page. The slab allocator is the one that stands behind commonly used kernel functions like `kmalloc()`. At the 2024 Linux Storage, Filesystem, Memory Management, and BPF Summit, slab maintainer Vlastimil Babka provided an update on recent changes at the slab level and discussed the changes that are yet to come.

Once upon a time, the kernel contained three slab-allocator implementations. That number had dropped to two in the 6.4 release, when the SLOB allocator (aimed at low-memory systems) was removed. At the 2023 summit, Babka began, the decision had been made to remove SLAB (one of the two general-purpose allocators), leaving only SLUB in the kernel. That removal happened in 6.8. Kernel developers now have greater freedom to improve SLUB without worrying about breaking the others. He thought that nobody was unhappy about this removal, he said, until he saw the recent report from the Embedded Open Source Summit, which contained some complaints. Even there, though, the primary complaint seemed to be that the removal had happened too quickly — even though

https://lwn.net/Articles/974138/

kfree_rcu() --> kvfree_call_rcu() --> kfree_rcu_sheaf() --> __kfree_rcu_sheaf()



/mm/slub.c#L5974 (v7.1.4)

```
bool __kfree_rcu_sheaf(struct kmem_cache *s, void *obj)
{
    ...

    rcu_sheaf->objects[rcu_sheaf->size++] = obj;
    ...
    if (rcu_sheaf)
        call_rcu(&rcu_sheaf->rcu_head, rcu_free_sheaf);
}
```

rcu_free_sheaf is an RCU callback function



/mm/slub.c#L5831 (v7.1.4)

```
static void rcu_free_sheaf(struct rcu_head *head)
{
    struct slab_sheaf *sheaf;
    struct node_barn *barn = NULL;
    struct kmem_cache *s;

    s = sheaf->cache;
    ...

    barn = get_barn_node(s, sheaf->node);
    ...

    if (data_race(barn->nr_full) < MAX_FULL_SHEAVES) {
        barn_put_full_sheaf(barn, sheaf);
        return;
    }
    ...
flush:
    stat(s, BARN_PUT_FAIL);
    sheaf_flush_unused(s, sheaf);
}
```



/mm/slub.c#L5831 (v7.1.4)

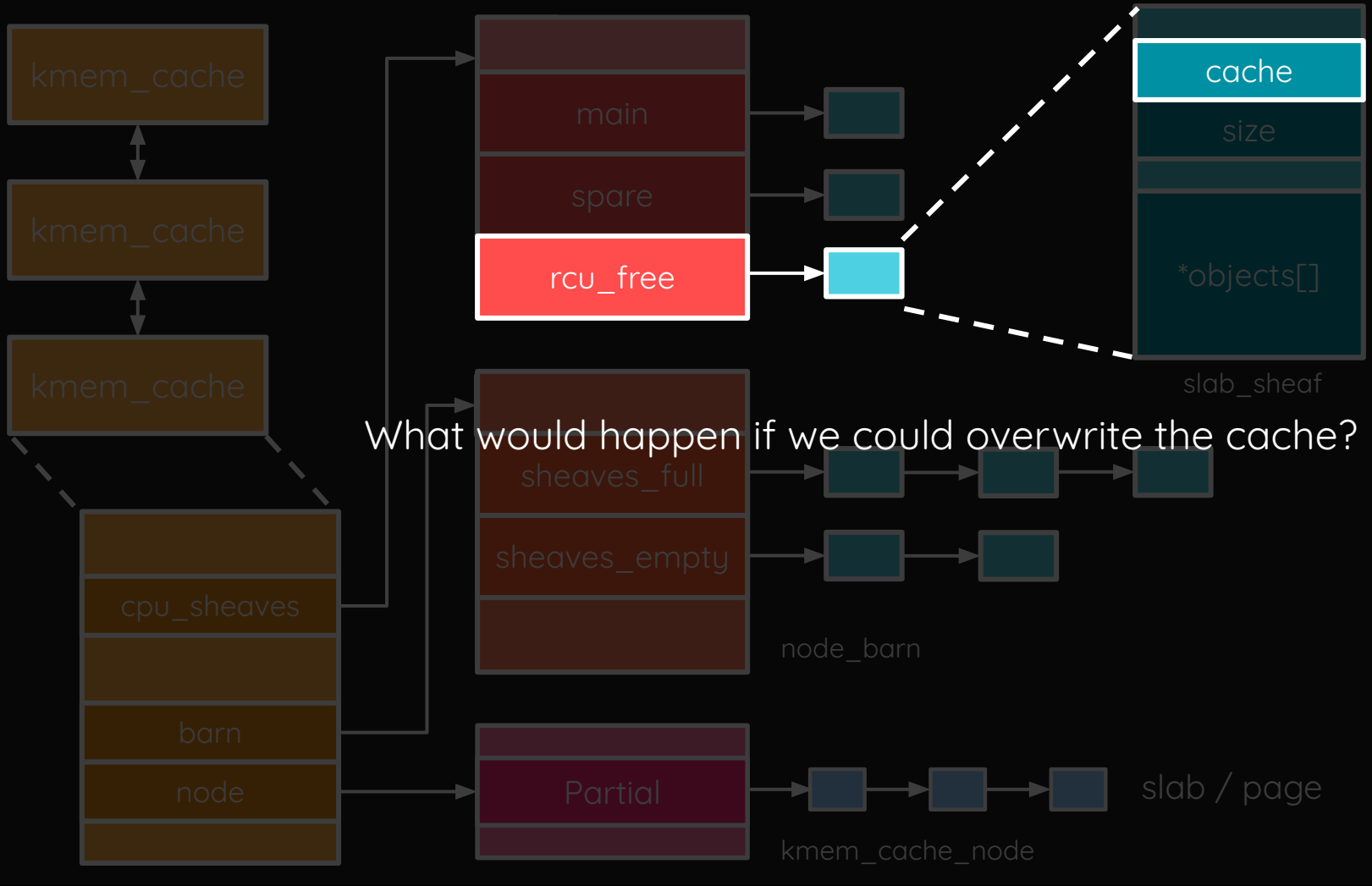
```
static void rcu_free_sheaf(struct rcu_head *head)
{
    struct slab_sheaf *sheaf;
    struct node_barn *barn = NULL;
    struct kmem_cache *s;

    s = sheaf->cache;

    barn = get_barn_node(s, sheaf->node);
    ...

    if (data_race(barn->nr_full) < MAX_FULL_SHEAVES) {
        barn_put_full_sheaf(barn, sheaf);
        return;
    }
    ...
flush:
    stat(s, BARN_PUT_FAIL);
    sheaf_flush_unused(s, sheaf);
}
```



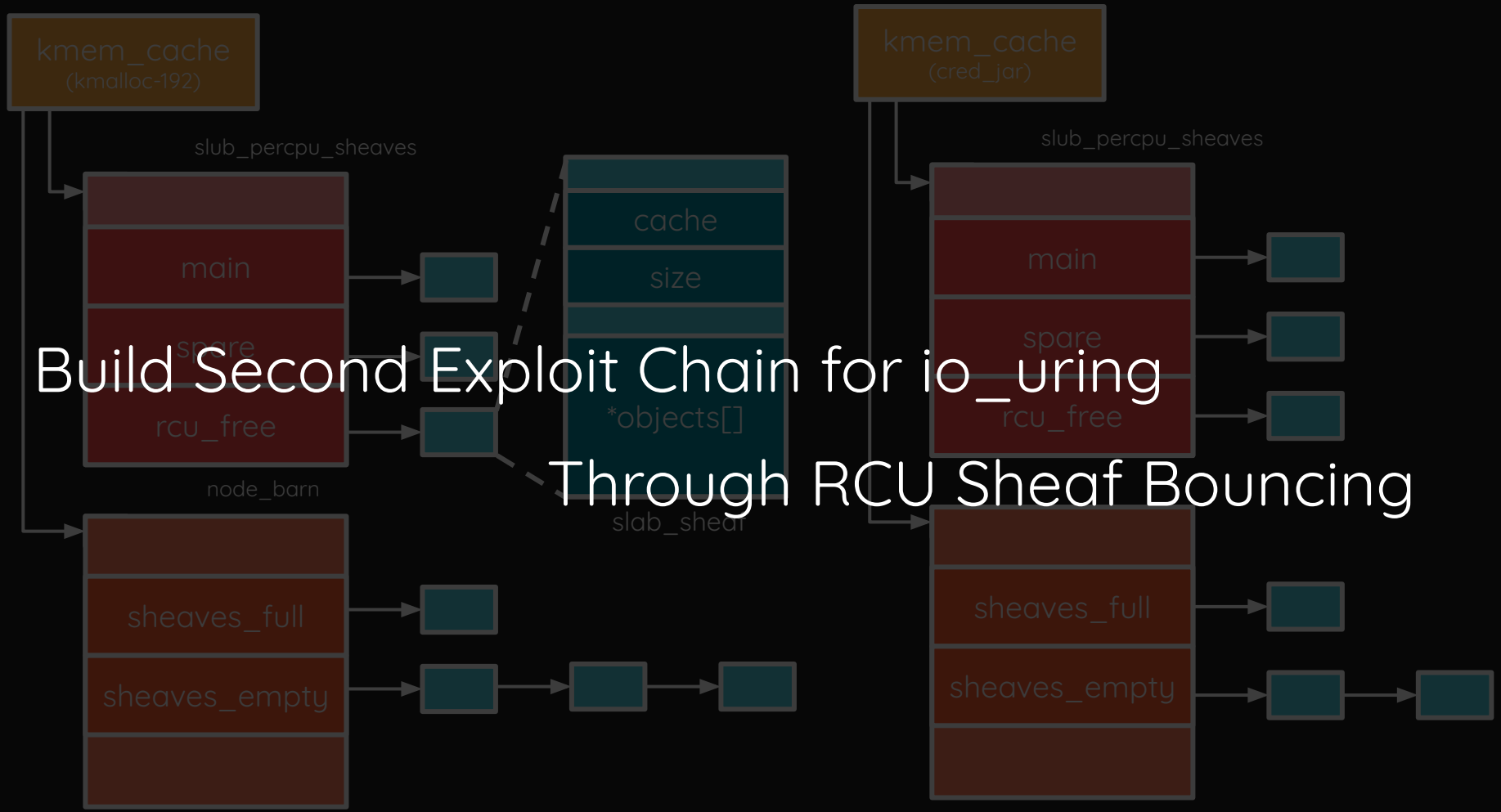

kmem_cache
kmem_cache
kmem_cache
cpu_sheaves
barn
node
main
spare
rcu_free
cache
size
*objects[]
slab_sheaf
What would happen if we could overwrite the cache?
sheaves_full
sheaves_empty
node_barn
Partial
slab / page
kmem_cache_node

My Original Idea

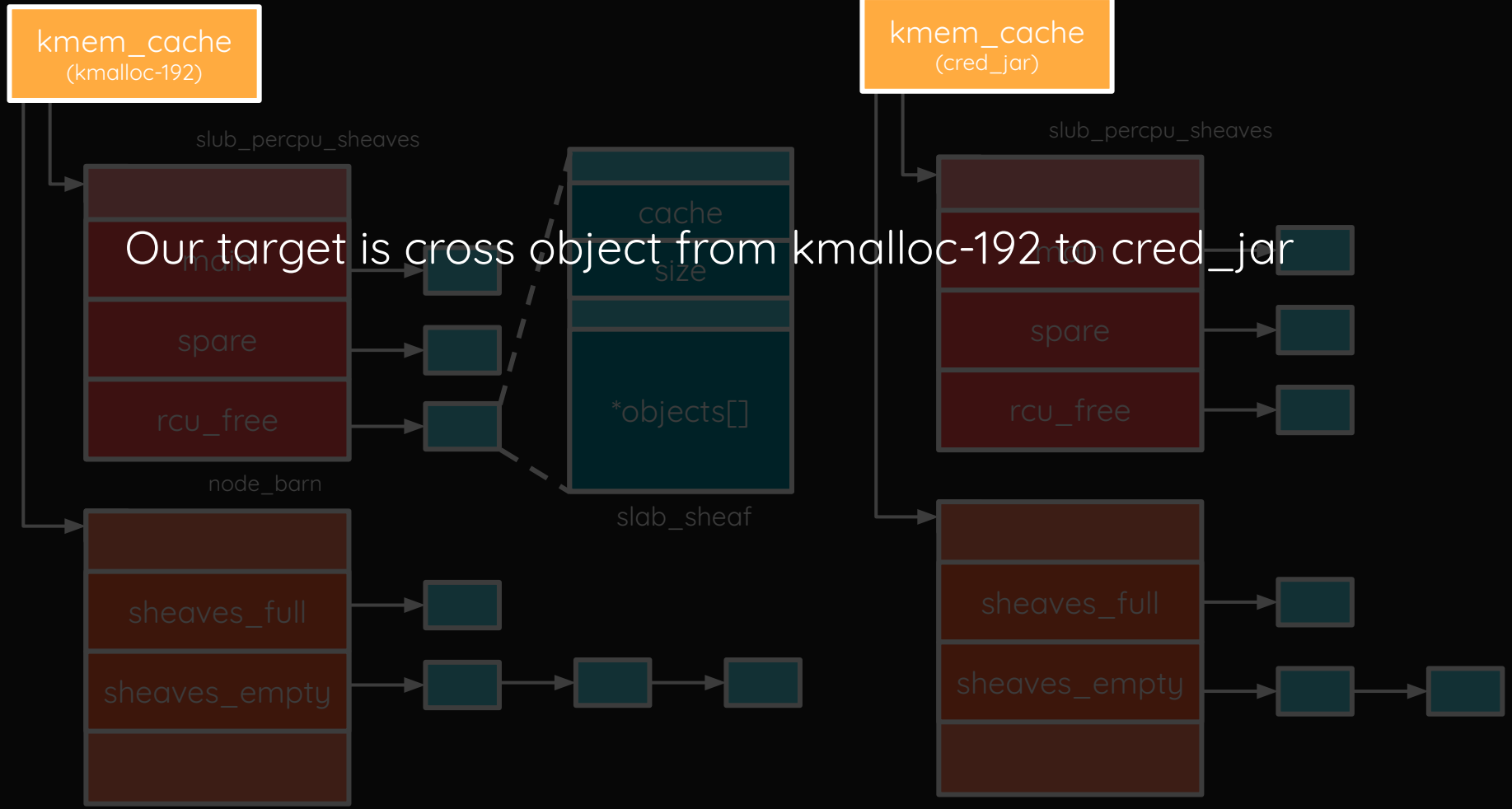
> Cross-cache the kmalloc-N UAF object into cred_jar, then overwrite the cred's UID/GID to obtain a root shell

> But ... I couldn't get the page containing the UAF object to be reclaimed by the Buddy System. ❌

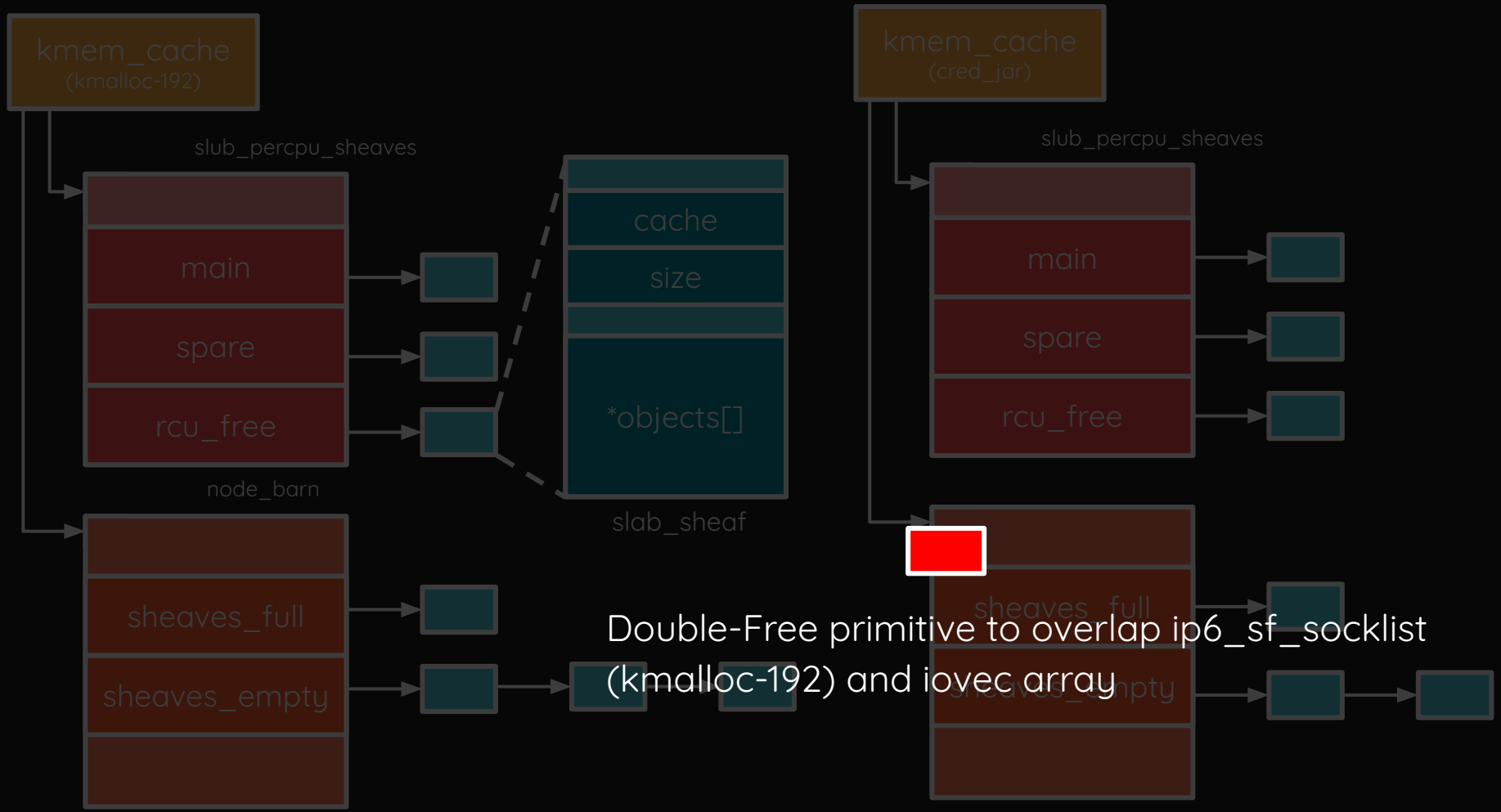
Cross Cache by RCU can get root shell !!!

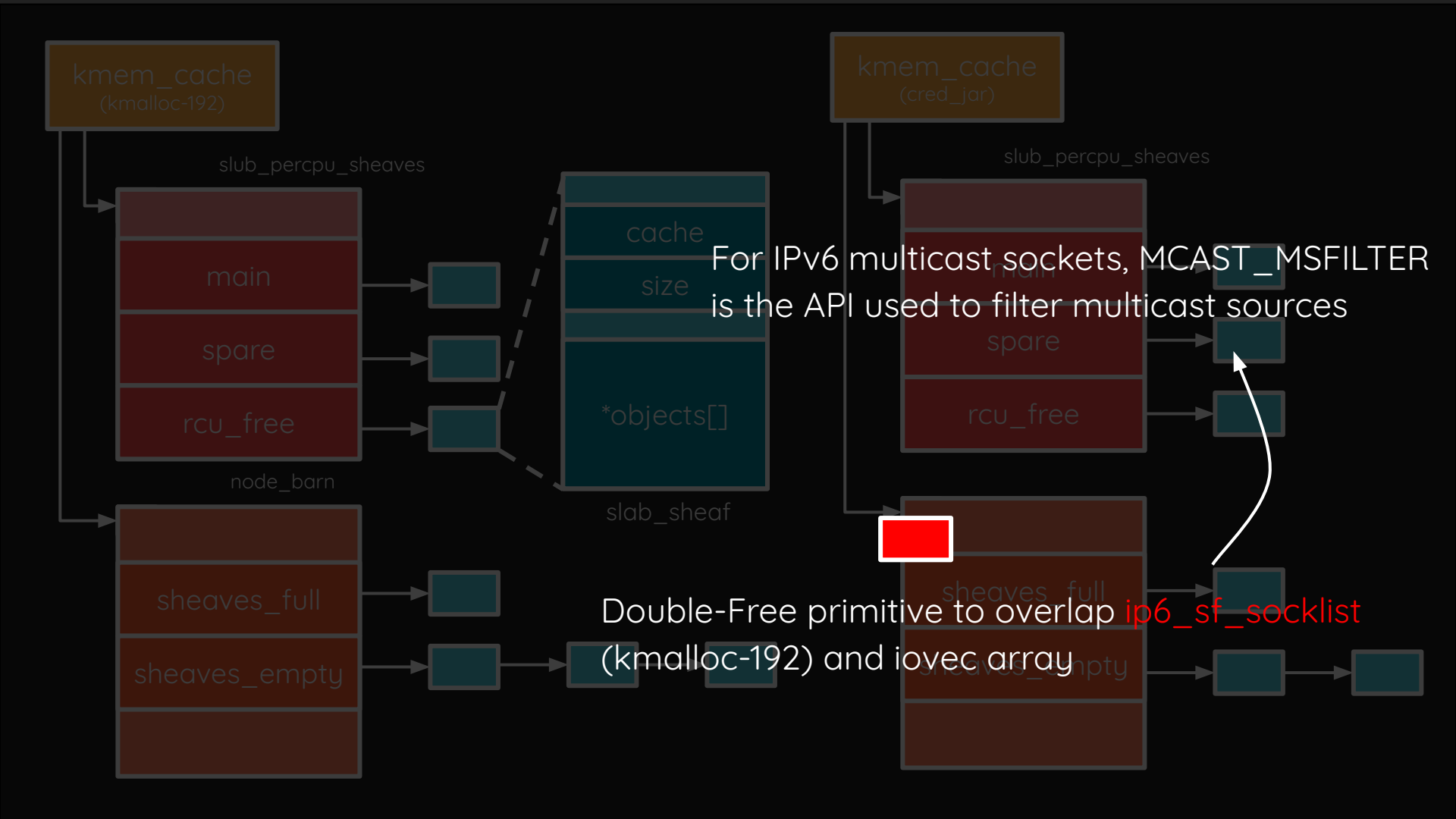


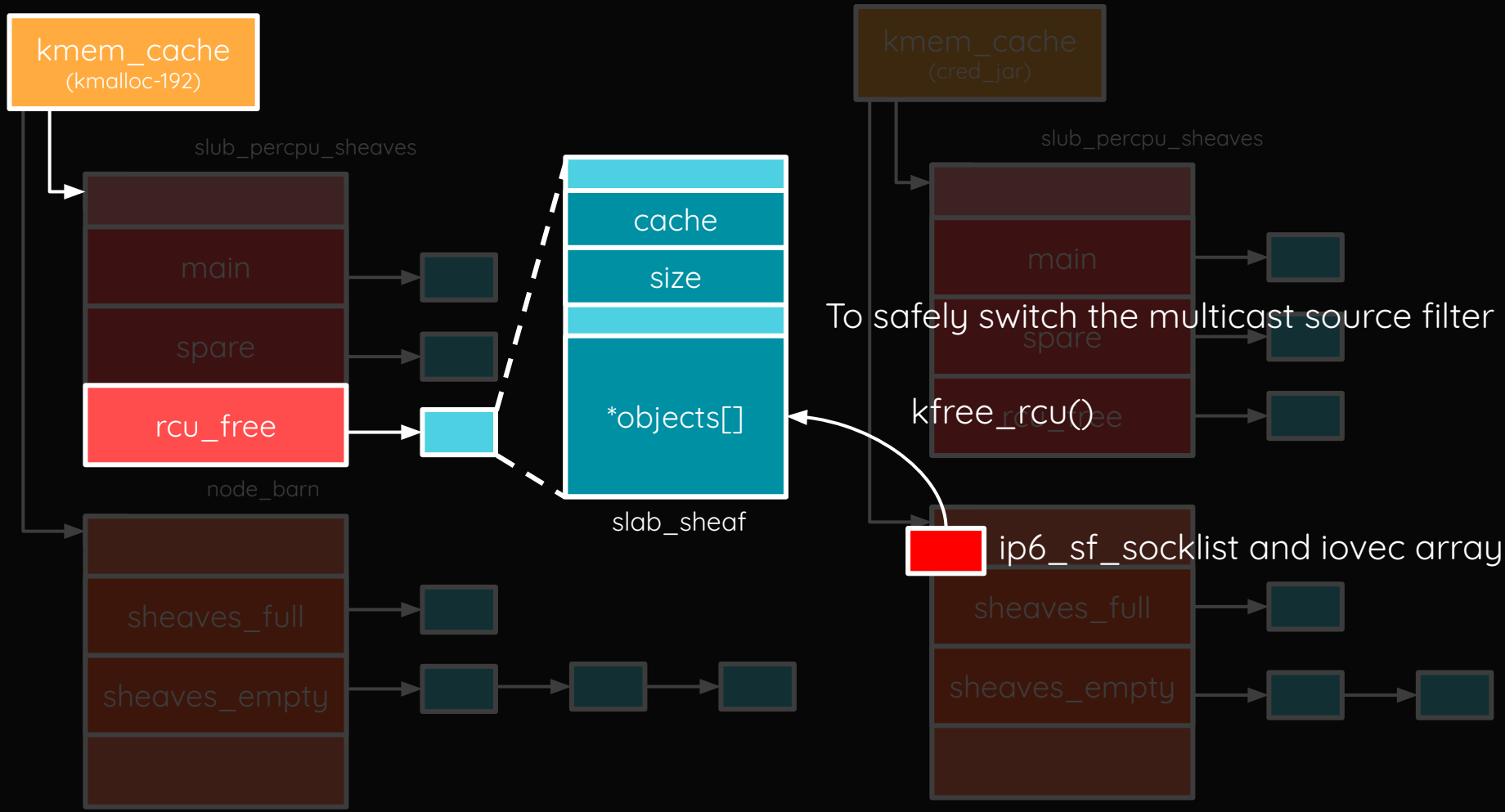
Build Second Exploit Chain for io_uring Through RCU Sheaf Bouncing

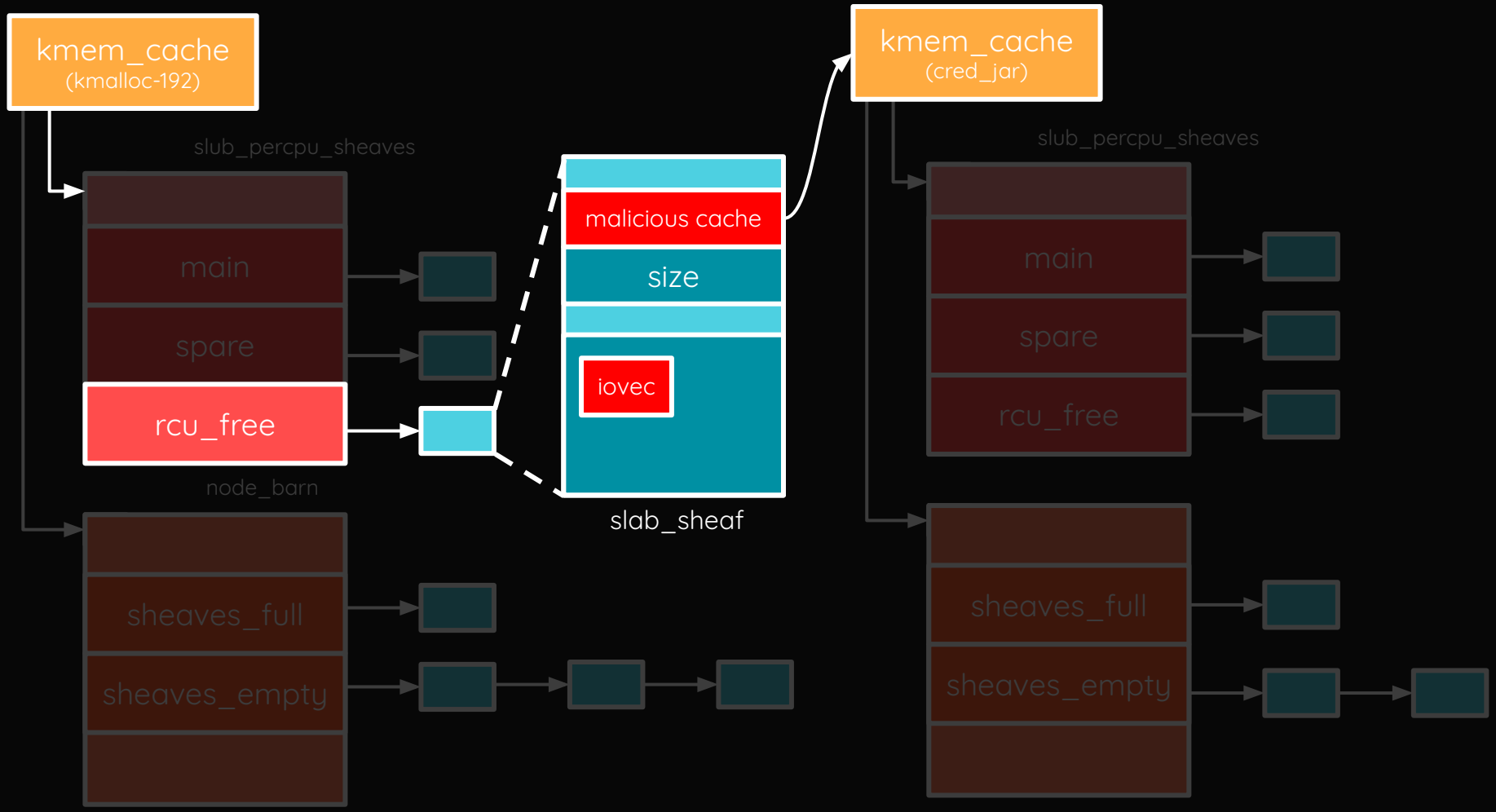


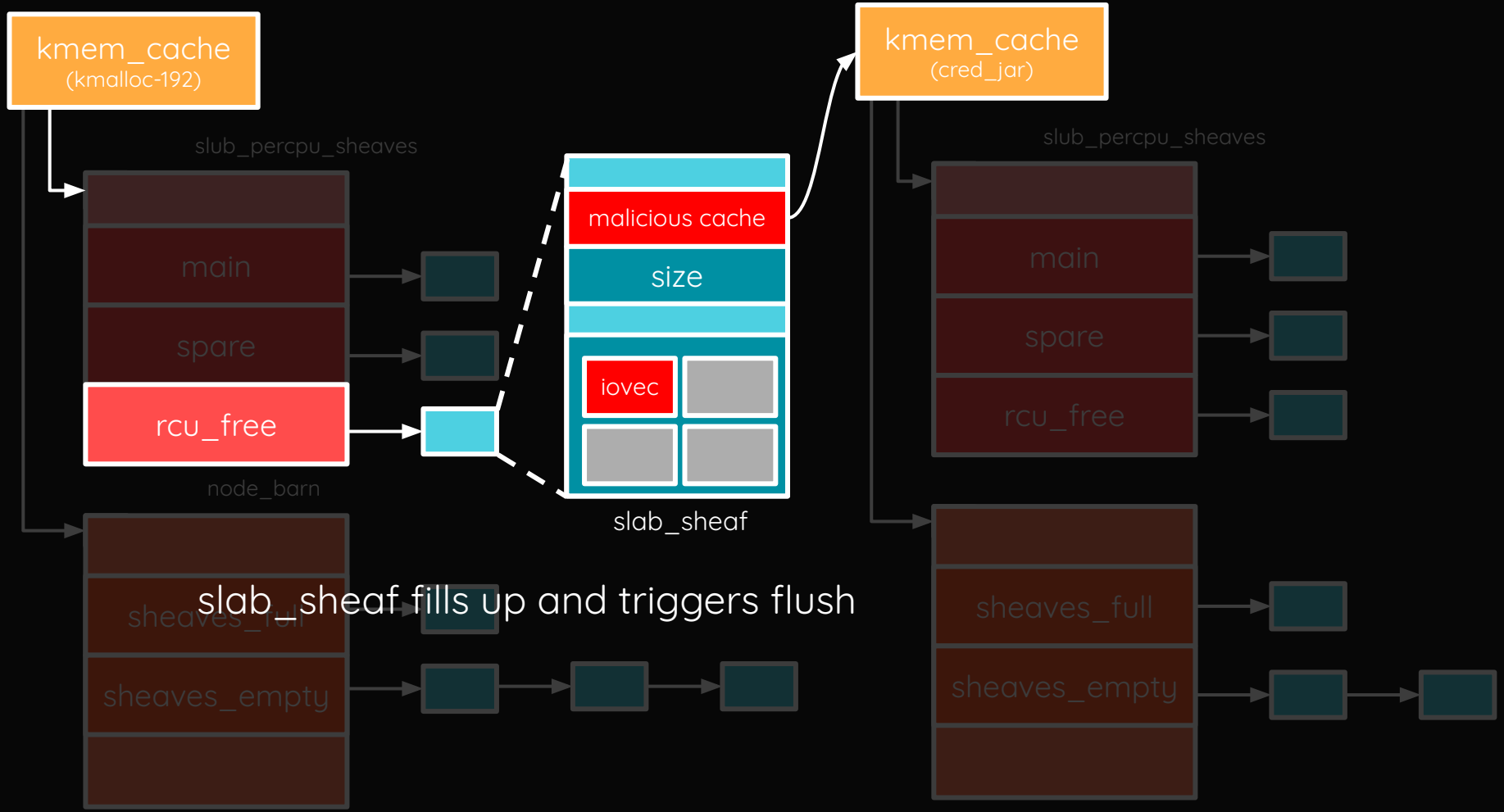
Our target is cross object from kmalloc-192 to cred_jar

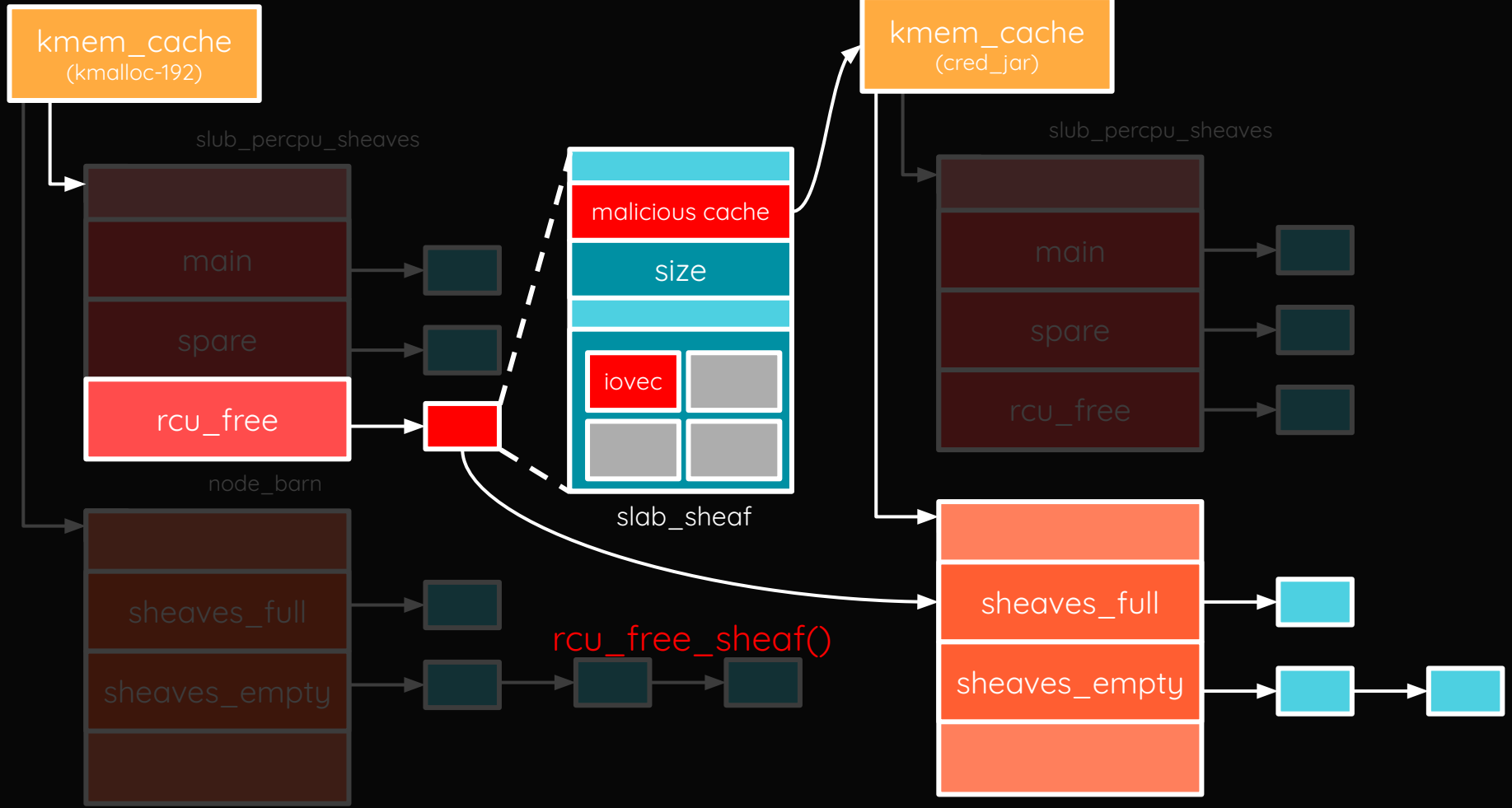

kmem_cache
(kmalloc-192)
slub_percpu_sheaves
main
spare
rcu_free
node_barn
sheaves_full
sheaves_empty
cache
size
*objects[]
slab_sheaf
kmem_cache
(cred_jar)
slub_percpu_sheaves
main
spare
rcu_free
sheaves_full
Double-Free primitive to overlap ip6_sf_socklist
(kmalloc-192) and iovec array

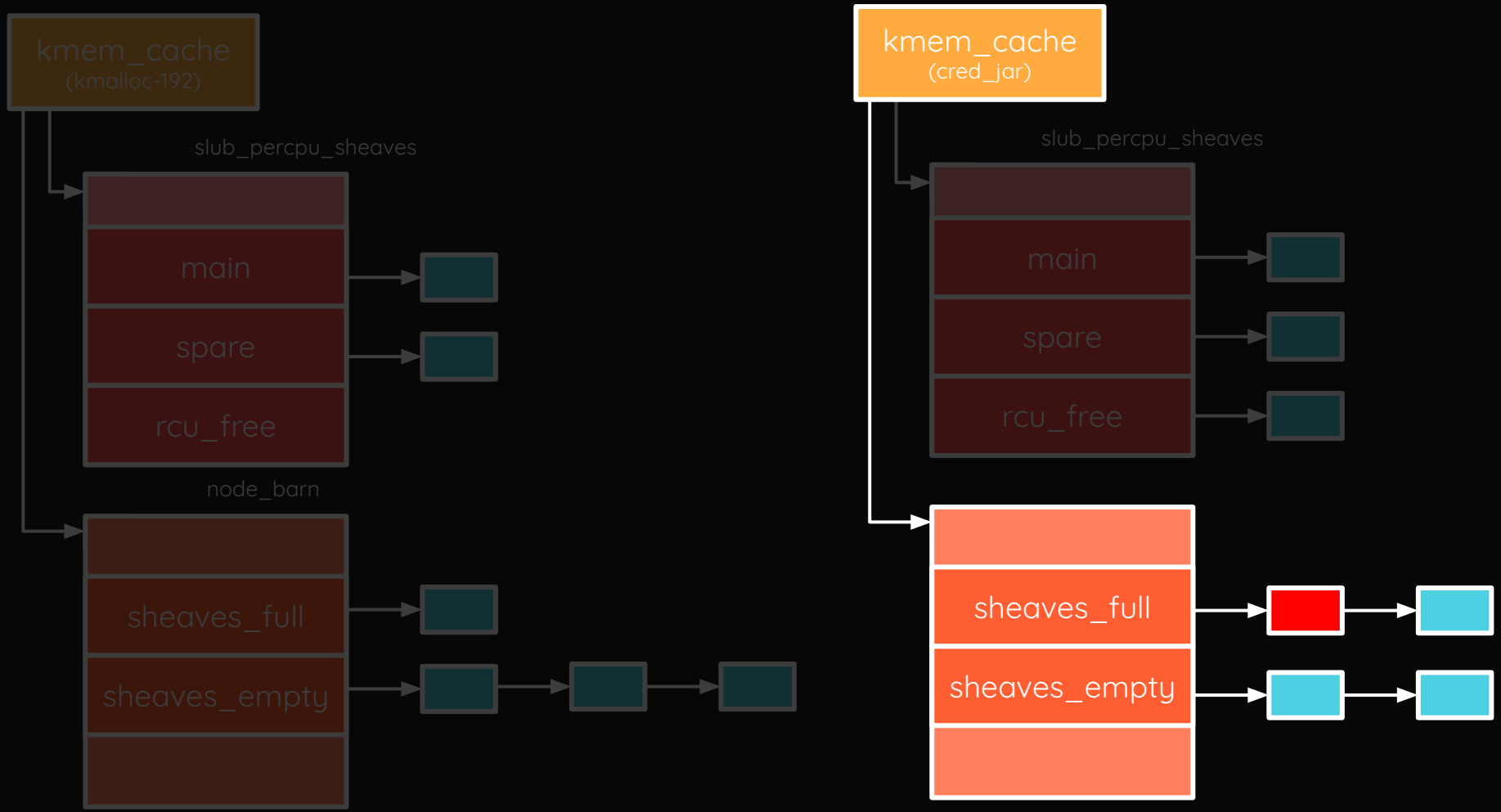

kmem_cache
(kmalloc-192)
slub_percpu_sheaves
main
spare
rcu_free
node_barn
sheaves_full
sheaves_empty
cache
size
*objects[]
slab_sheaf
kmem_cache
(cred_jar)
slub_percpu_sheaves
spare
rcu_free
sheaves_full
For IPv6 multicast sockets, MCAST_MSFILTER
is the API used to filter multicast sources
Double-Free primitive to overlap ip6_sf_socklist
(kmalloc-192) and iovec array

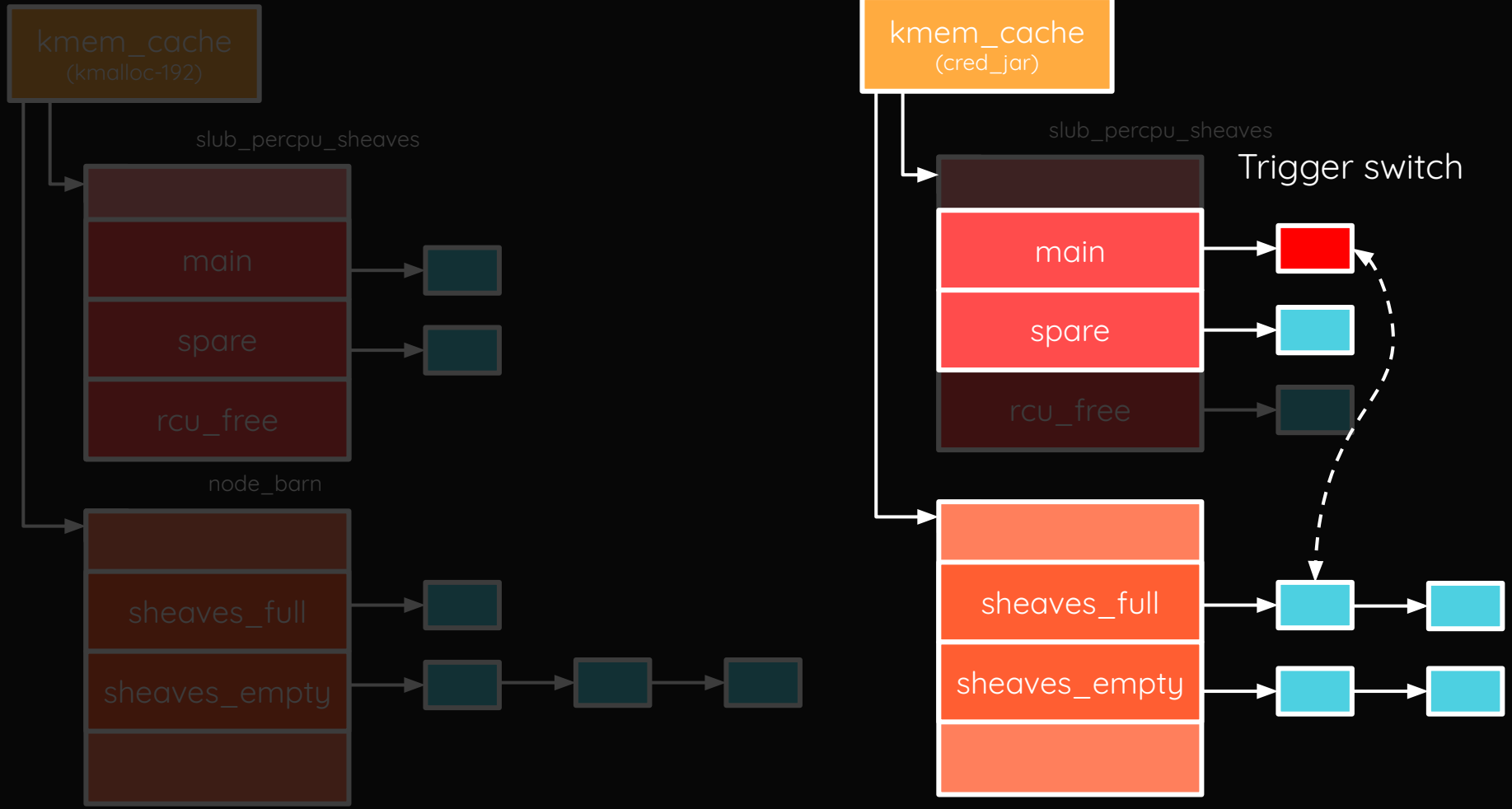

kmem_cache
(kmalloc-192)
slub_percpu_sheaves
main
spare
rcu_free
node_barn
sheaves_full
sheaves_empty
cache
size
*objects[]
slab_sheaf
kmem_cache
(cred_jar)
slub_percpu_sheaves
main
spare
To safely switch the multicast source filter
kfree_rcu()
ip6_sf_socklist and iovec array
sheaves_full
sheaves_empty

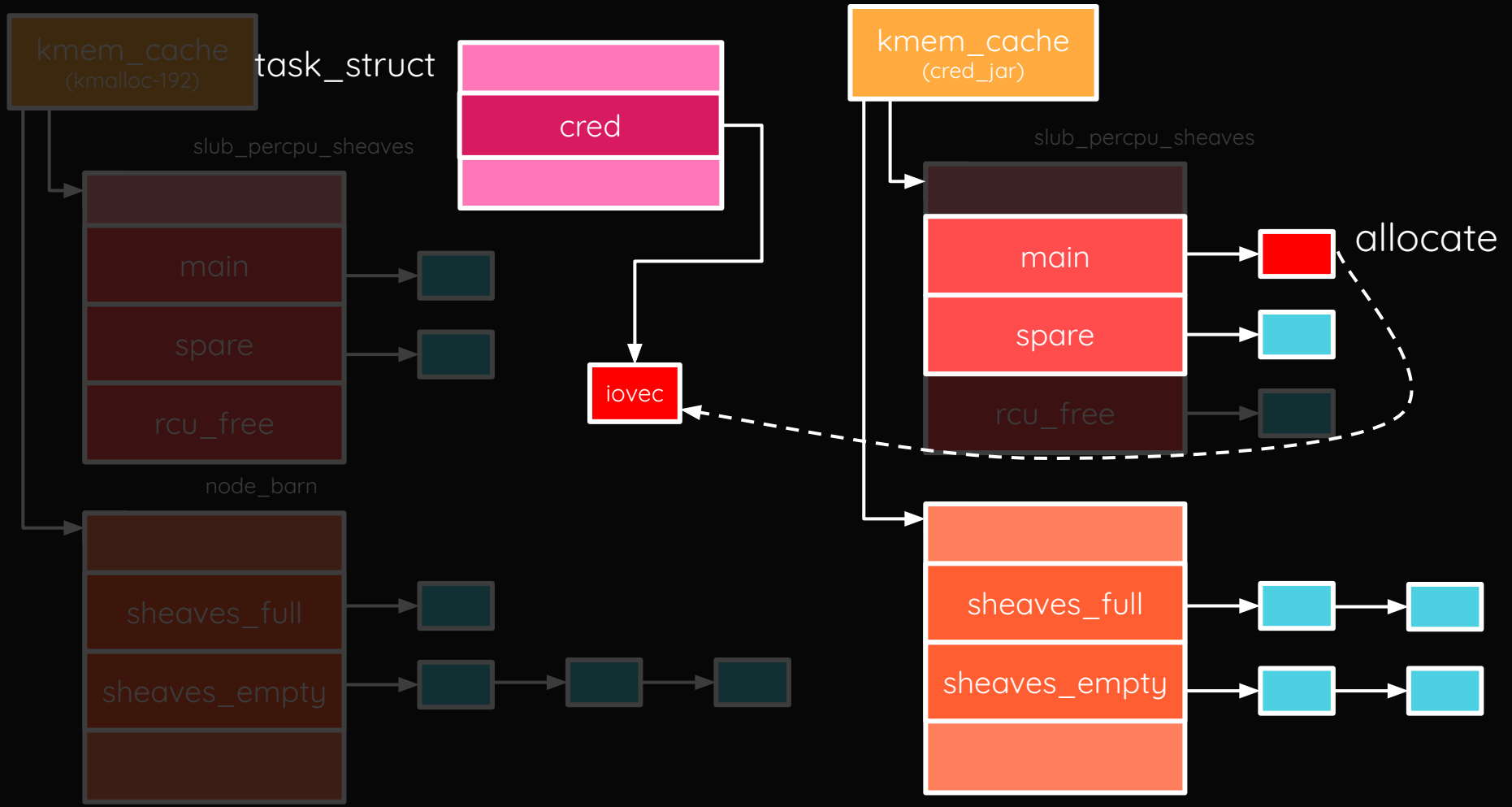

kmem_cache
(kmalloc-192)
slub_percpu_sheaves
main
spare
rcu_free
node_barn
sheaves_full
sheaves_empty
malicious cache
size
iovec
slab_sheaf
kmem_cache
(cred_jar)
slub_percpu_sheaves
main
spare
rcu_free
sheaves_full
sheaves_empty

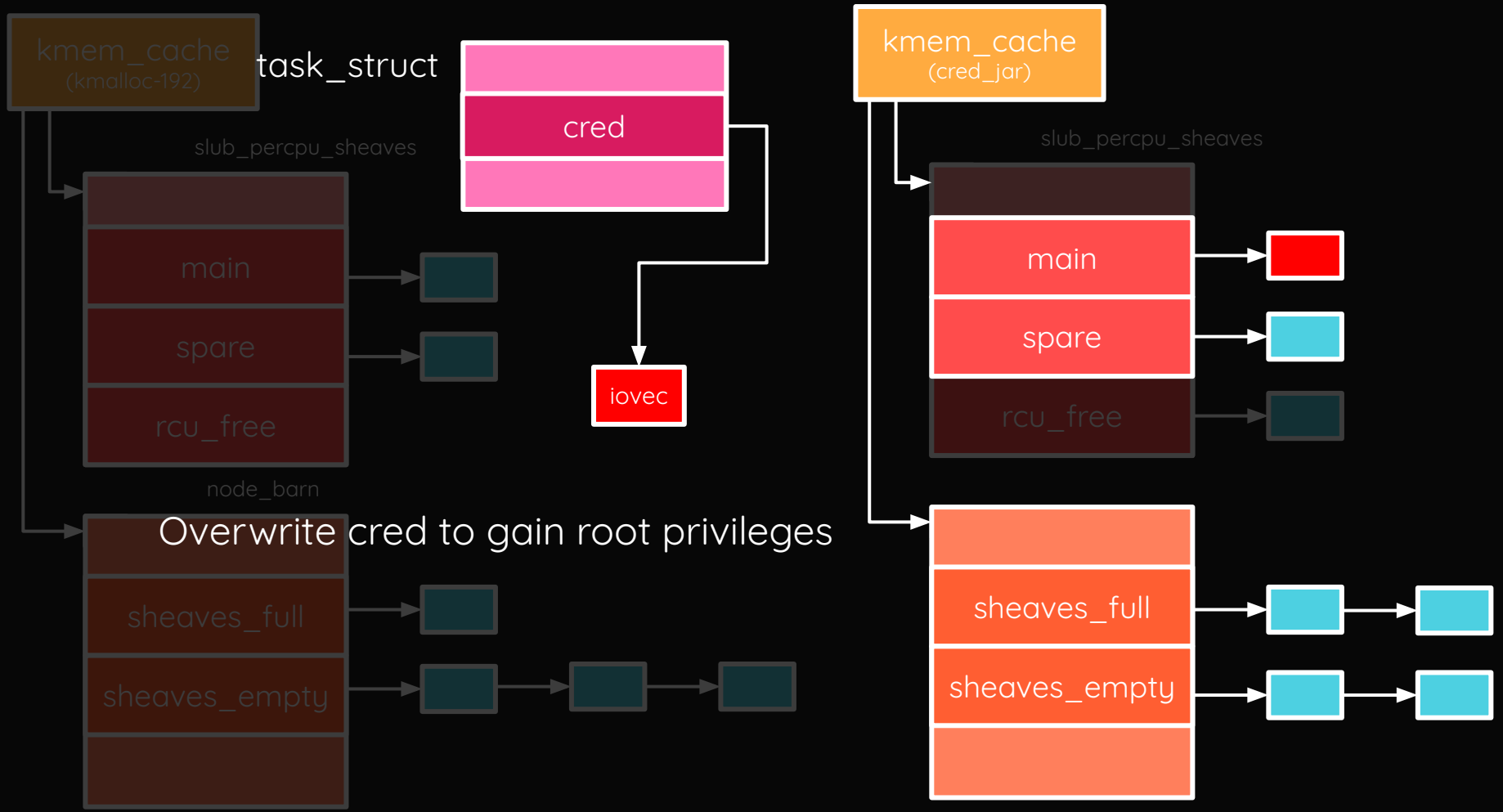

kmem_cache
(kmalloc-192)
slub_percpu_sheaves
main
spare
rcu_free
node_barn
sheaves_full
sheaves_empty
malicious cache
size
iovec
slab_sheaf
kmem_cache
(cred_jar)
slub_percpu_sheaves
main
spare
rcu_free
sheaves_full
sheaves_empty
slab_sheaf fills up and triggers flush

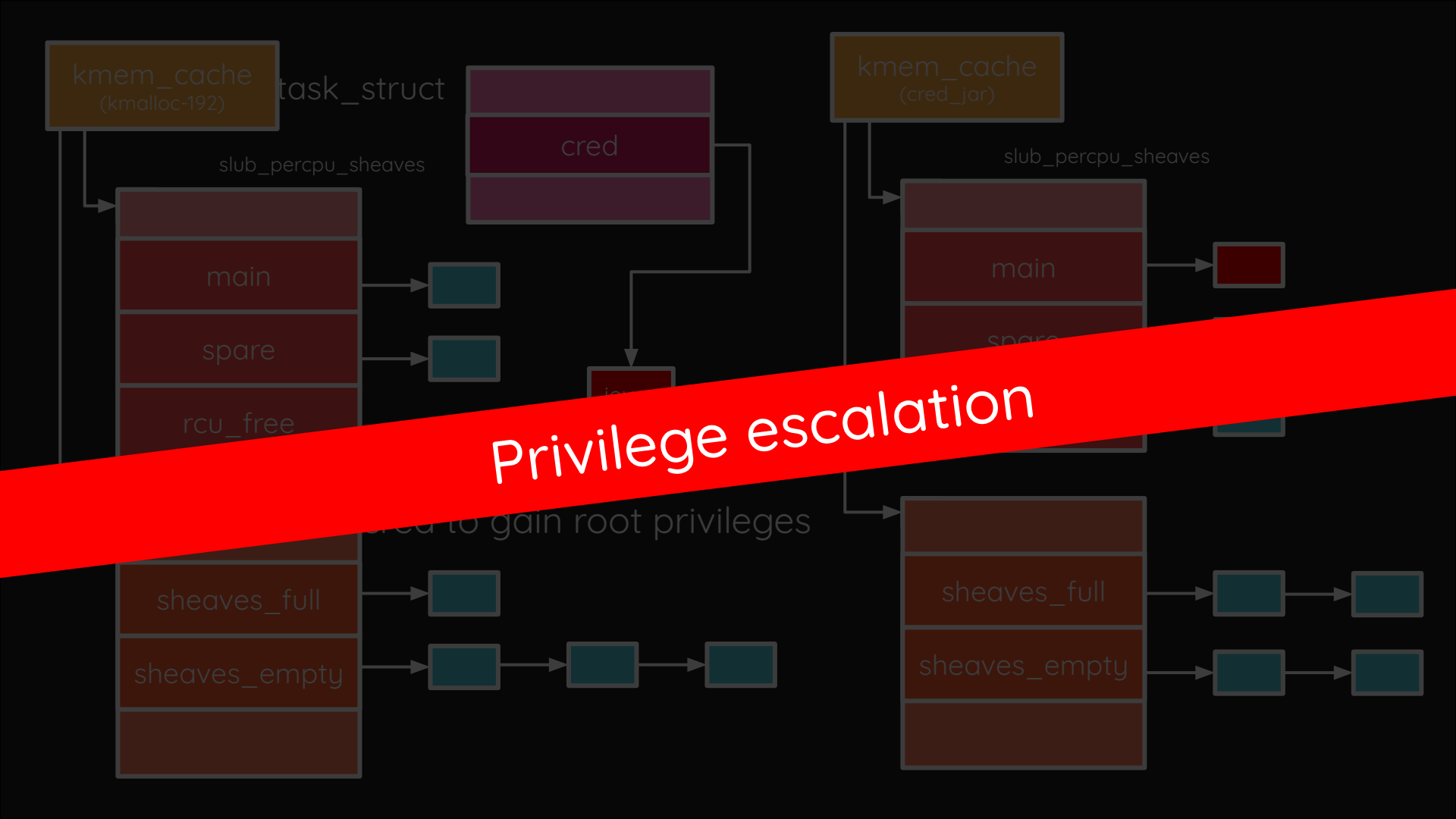

kmem_cache
(kmalloc-192)
slub_percpu_sheaves
main
spare
rcu_free
node_barn
sheaves_full
sheaves_empty
malicious cache
size
iovec
slab_sheaf
rcu_free_sheaf()
kmem_cache
(cred_jar)
slub_percpu_sheaves
main
spare
rcu_free
sheaves_full
sheaves_empty


kmem_cache
(kmalloc-192)
slub_percpu_sheaves
main
spare
rcu_free
node_barn
sheaves_full
sheaves_empty
kmem_cache
(cred_jar)
slub_percpu_sheaves
main
spare
rcu_free
sheaves_full
sheaves_empty

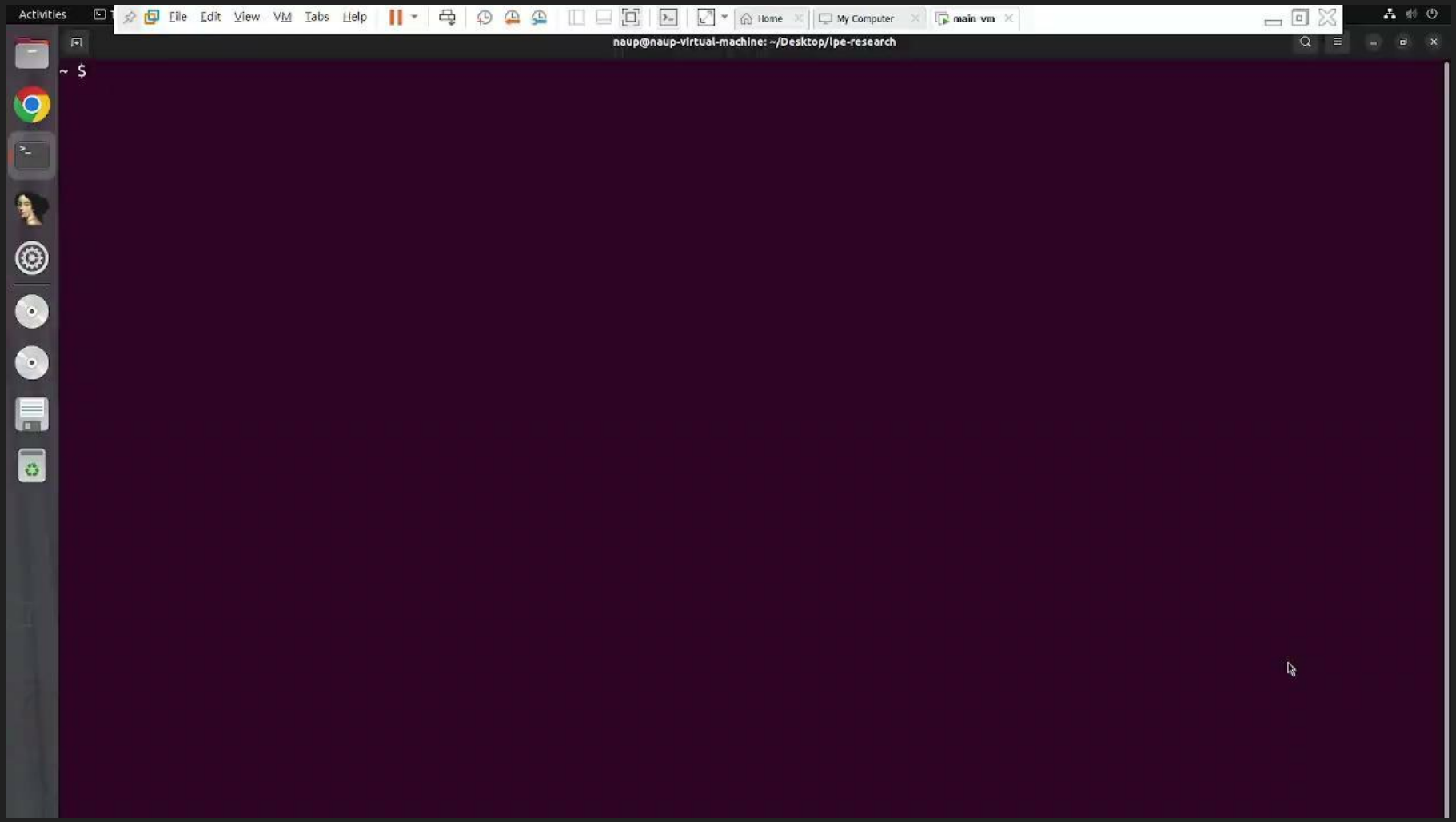

kmem_cache
(kmalloc-192)
slub_percpu_sheaves
main
spare
rcu_free
node_barn
sheaves_full
sheaves_empty
kmem_cache
(cred_jar)
slub_percpu_sheaves
Trigger switch
main
spare
rcu_free
sheaves_full
sheaves_empty


kmem_cache
(kmalloc-192)
task_struct
cred
kmem_cache
(cred_jar)
slub_percpu_sheaves
main
spare
rcu_free
slub_percpu_sheaves
main
spare
rcu_free
allocate
iovec
node_barn
sheaves_full
sheaves_empty
sheaves_full
sheaves_empty

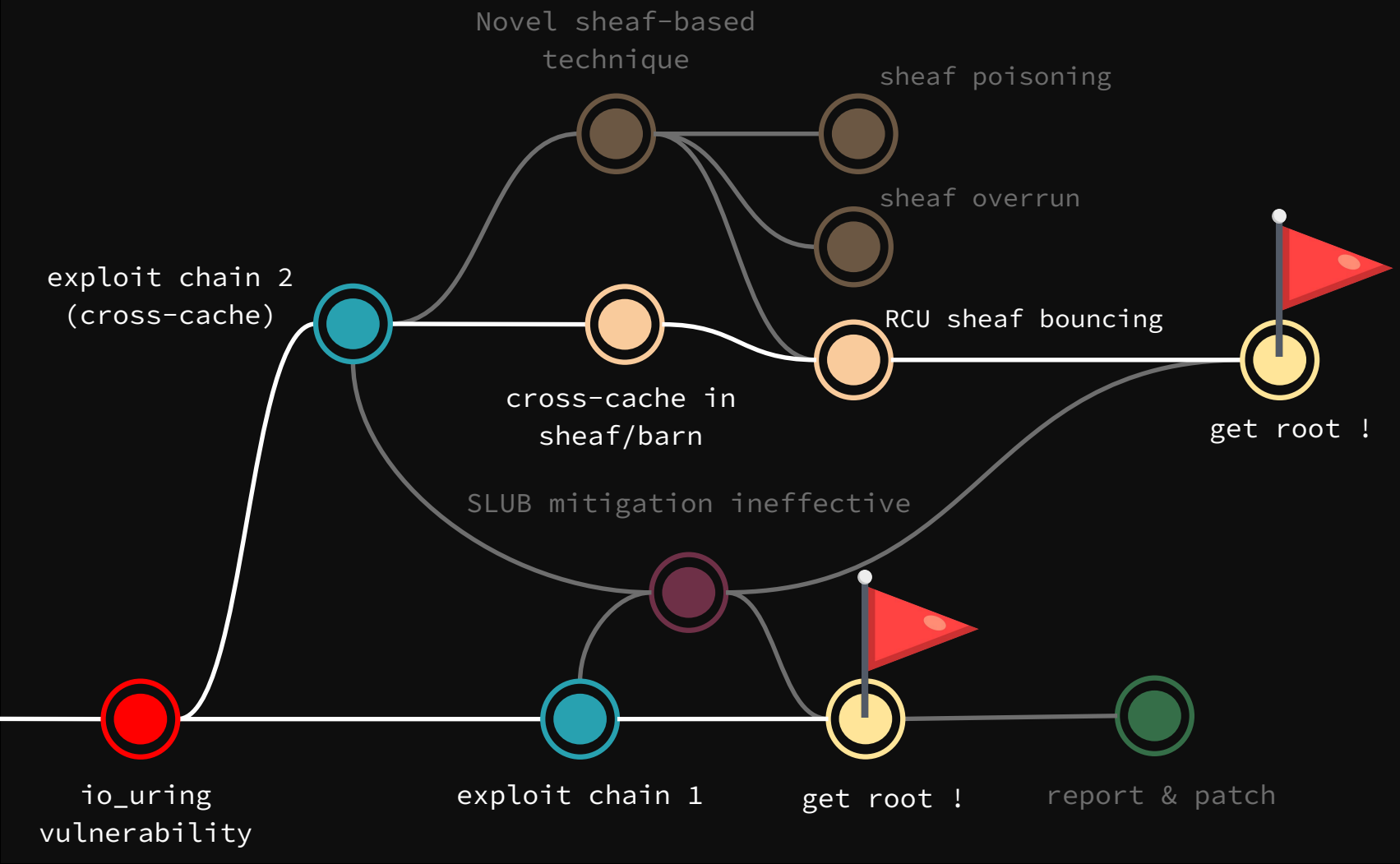

kmem_cache
(kmalloc-192)
task_struct
cred
slub_percpu_sheaves
main
spare
rcu_free
iovec
node_barn
Overwrite cred to gain root privileges
sheaves_full
sheaves_empty
kmem_cache
(cred_jar)
slub_percpu_sheaves
main
spare
rcu_free
sheaves_full
sheaves_empty


kmem_cache
(kmalloc-192)
task_struct
cred
kmem_cache
(cred_jar)
slub_percpu_sheaves
slub_percpu_sheaves
main
spare
rcu_free
main
sheaves_full
sheaves_empty
sheaves_full
sheaves_empty
Privilege escalation

Demo Time


Activities
File Edit View VM Tabs Help
Home
My Computer
main vm
naup@naup-virtual-machine: ~/Desktop/lpe-research
~ $


LEFT
EXIT 12
buddy system
sheaf RCU
cross cache


Novel sheaf-based technique
sheaf poisoning
sheaf overrun
exploit chain 2 (cross-cache)
RCU sheaf bouncing
cross-cache in sheaf/barn
get root !
SLUB mitigation ineffective
io_uring vulnerability
exploit chain 1
get root !
report & patch

Conclusion

Takeaways and future work !

Takeaways !

> SLUB allocator has added the sheaf/barn mechanism, so many traditional SLUB exploitation techniques no longer work as expected

    > Heap spray more objects or use NUMA node

> sheaf/barn mechanism introduces a new attack surface and techniques

    > bypass existing SLUB mitigations (ex: SLAB_FREELIST_HARDENED)

    > providing additional privilege escalation paths (Three novel techniques)

Future Work !

I believe this mechanism still has unexplored aspects worth researching :

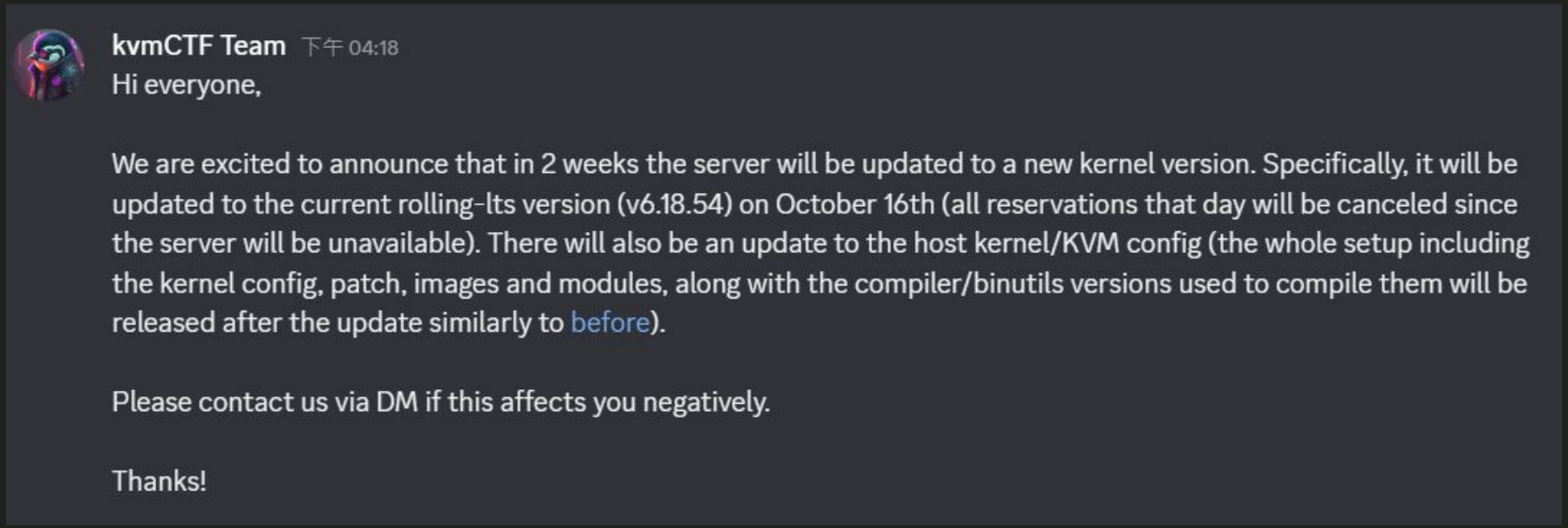
> Discover new techniques and combine existing techniques for privilege escalation under more restricted conditions (ex: #1 + #2 = leakless to LPE)

> Bypass more mitigations, including slab_virtual and others

> Timing Side Channel to leak SLUB allocator status for sheaf / barn mechanism (ex: SLUBstick, pspray ...)

Future Work !

> Apply it in real world scenarios (ex: kernelCTF, kvmCTF and other distros)



kvmCTF Team 下午 04:18

Hi everyone,

We are excited to announce that in 2 weeks the server will be updated to a new kernel version. Specifically, it will be updated to the current rolling-lts version (v6.18.54) on October 16th (all reservations that day will be canceled since the server will be unavailable). There will also be an update to the host kernel/KVM config (the whole setup including the kernel config, patch, images and modules, along with the compiler/binutils versions used to compile them will be released after the update similarly to before).

Please contact us via DM if this affects you negatively.

Thanks!

This talk is just the first part of this series. I'll be sharing more research

on this attack surface in the future, so stay tuned ~



THANKS FOR YOUR LISTENING


naup96721@gmail.com


dong1214.mailbox@gmail.com