Déjà Vu in Linux io_uring: Breaking Memory Sharing Again After Generations of Fixes

Pumpkin Chang (@u1f383)

October 10, 2025

DEVCORE

HEXACON 2025

$ whoami

- Pumpkin 🎃 (@u1f383)
- Security researcher at DEVCORE
- Focus on Linux Kernel, Android and VM

$ Outline

- Linux io_uring Introduction
    - Architecture & Memory Sharing
- Review of Three Previous Bugs
    - Memory Sharing Design Issues in io_uring
- CVE-2025-21836: Root Cause & Exploitation
- Takeaways

Introduction

$ Overview

- io_uring
  - Introduced in Linux 5.1 in 2019
  - Provides high-performance async I/O
  - Reduces syscall overhead by asynchronous handling
  - Shares objects between user and kernel space for fast data exchange

$ Overview

- Three core components:
  - I/O Command
    → Encapsulates each I/O request as a command
    → Prepares and validates it based on command type
  - Command Worker
    → Processes I/O requests in the kernel I/O thread
  - Memory Sharing
    → Shares memory with user space to avoid costly data copying

$ Overview

- Three core components:
  - I/O Command
    → Encapsulates each I/O request as a command
    → Prepares and validates it based on command type
  - Command Worker
    → Processes I/O requests in the kernel I/O thread
  - Memory Sharing
    → Shares memory with user space to avoid costly data copying

$ Overview

- Three core components:
  - I/O Command
    → Encapsulates each I/O request as a command
    → Prepares and validates it based on command type
  - Command Worker
    → Processes I/O requests in the kernel I/O thread
  - Memory Sharing
    → Shares memory with user space to avoid costly data copying

$ Memory Sharing

- io_uring memory sharing:
  1. Submission Queue (SQ) and Completion Queue (CQ)
     - Used for submitting commands and fetching results
  2. Kernel buffers
     - Reallocated memory for zero-copy data transfer

$ Memory Sharing

- io_uring memory sharing:
  1. Submission Queue (SQ) and Completion Queue (CQ)
     - Used for submitting commands and fetching results
  2. Kernel buffers
     - Store user data
     - Reallocated memory for zero-copy data transfer

$ Memory Sharing

- io_uring memory sharing:
  1. Submission Queue (SQ) and Completion Queue (CQ)
     - Used for submitting commands and fetching results
  2. Kernel buffers
     - Store user data
     - Reallocated memory for zero-copy data transfer

What we focus on !

$ Memory Sharing

- I/O Buffer (Kernel buffer)
  - Abstracts shared memory management

$ Memory Sharing

- I/O Buffer (Kernel buffer)
  - Abstracts shared memory management
  - Process can register shared memory to io_uring context

$ Memory Sharing

- I/O Buffer (Kernel buffer)
  - Abstracts shared memory management
  - Process can register shared memory to io_uring context
  - Managed by I/O buffer object
    - Stored in an array within the context

$ Memory Sharing

- I/O Buffer (Kernel buffer)
  - Abstracts shared memory management
  - Process can register shared memory to io_uring context
  - Managed by I/O buffer object
    - Stored in an array within the context
  - Command can specify the buffer index to use it

User space
Kernel space
1. Create an array to manage I/O buffer object
Process
I/O buffer array
Index
I/O Buffer
0
1
...
RAM


User space
Kernel space
2. Assume there is an I/O buffer object with some backing pages
Process
I/O buffer array
Index
I/O Buffer
0
Object
1
...
Page
Page
...


User space
Kernel space
io_uring worker
Process
I/O buffer array
READ
Index
I/O Buffer
0
Object
1
…
Flags: REQ_F_BUFFER_SELECT
Buffer index: 0
3. Process specifies the buffer index in I/O command
Page
Page
...


User space
Kernel space
Process
io_uring worker
READ
Index: 0
I/O buffer array
Index
I/O Buffer
0
Object
1
...
4. The read data will be stored
in I/O buffer object
0x4141
0x4141
0x4141

$ Memory Sharing

- I/O Buffer (Kernel buffer)
  - Ring buffer: a large, continuous memory region
    - [1] Memory can be pre-allocated by process

      [2] or reserved by kernel
  - Provided buffer: multiple small, non-contiguous buffers
    - Use fragmented user-space memory to store data


User space
Kernel space
Process
1. Preallocate some anonymous pages
I/O buffer array
Index
I/O Buffer
0
1
...
Page
Page
...

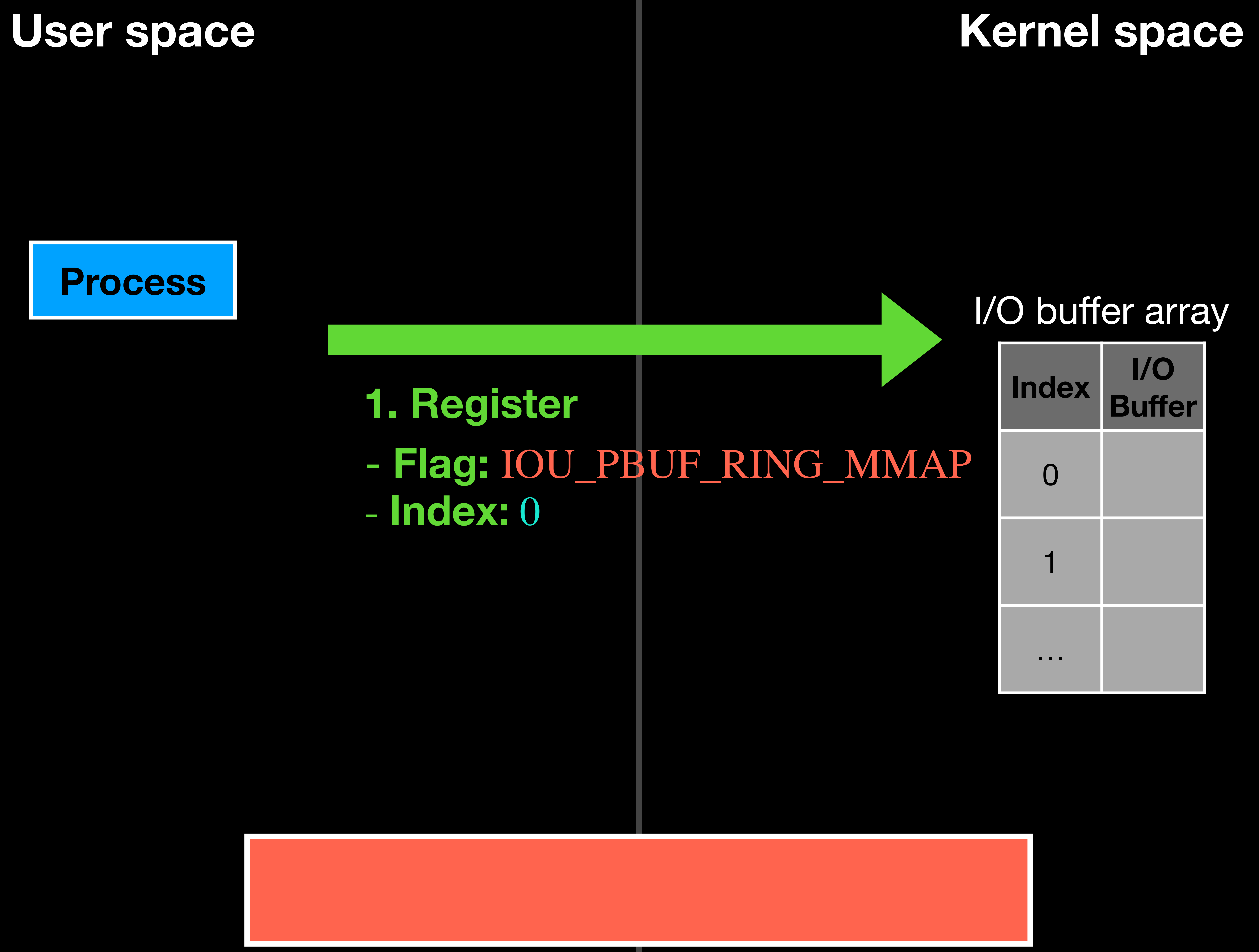

User space
Kernel space
Process
2. Register
- Flag: 0
- Index: 0
I/O buffer array
Index
I/O Buffer
0
1
...
Page
Page
...


User space
Kernel space
3. Create an I/O buffer object
to manage these pages
Process
I/O buffer array
Index
I/O Buffer
0
Object
1
...
Page
Page
...

$ Memory Sharing

- I/O Buffer (Kernel buffer)
  - Ring buffer: a large, continuous memory region
    - [1] Memory can be pre-allocated by process

      [2] or reserved by kernel
  - Provided buffer: multiple small, non-contiguous buffers
    - Use fragmented user-space memory to store data


User space
Kernel space
Process
1. Register
- Flag: IOU_PBUF_RING_MMAP
- Index: 0
I/O buffer array
Index
I/O Buffer
0
1
...


User space
Kernel space
2. Allocate pages based on the request size
Process
I/O buffer array
Index
I/O Buffer
0
Object
1
...
Page
Page
...

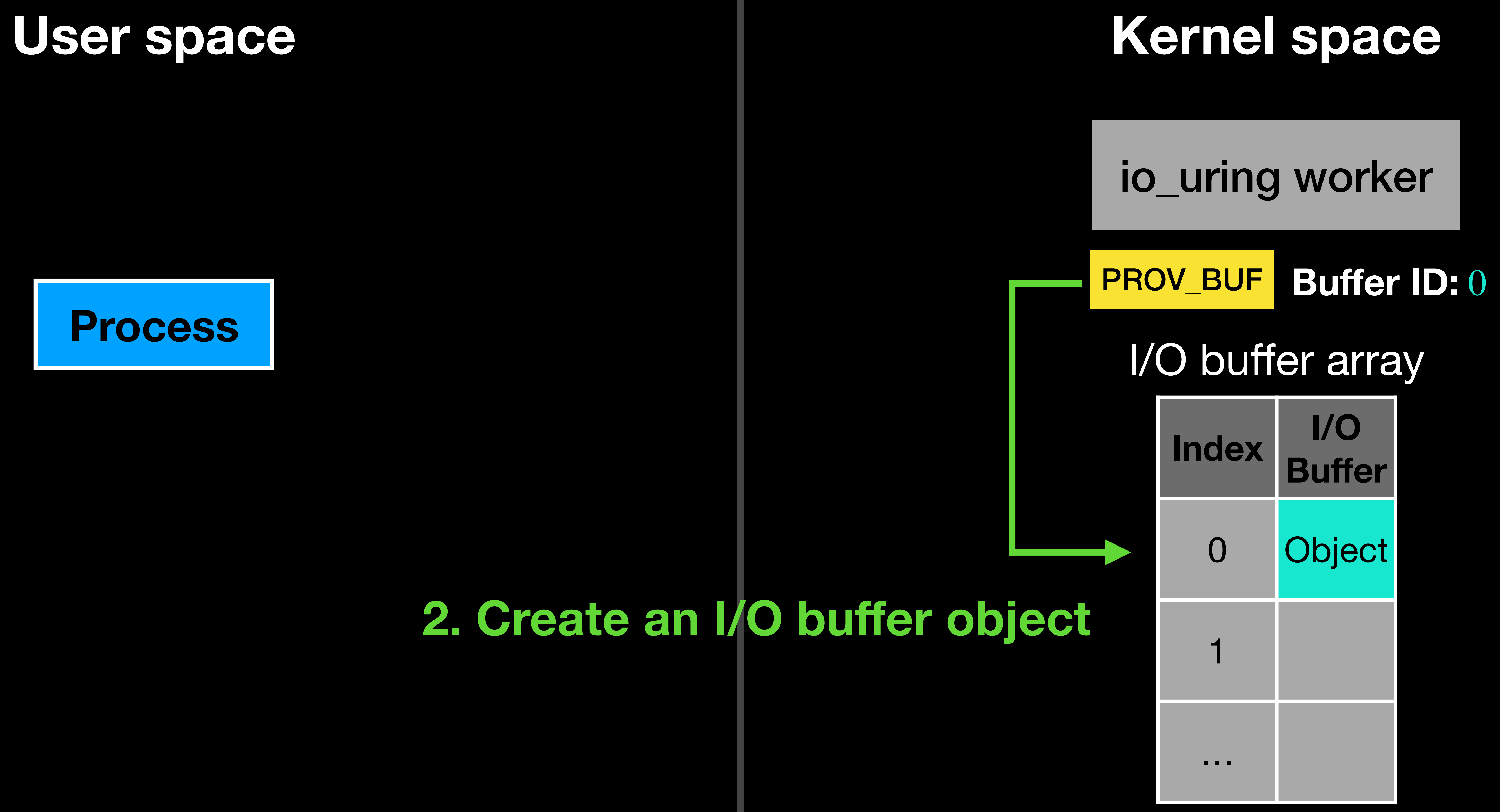
User space
Kernel space
Process
3. Map the backing pages
to user space
I/O buffer array
Index
I/O Buffer
0
Object
1
...
Page
Page
...


User space
Kernel space
Process
I/O buffer array
Index
I/O Buffer
0
Object
1
…
4. The memory can now be accessed
0x4141
0x4141
0x4141

$ Memory Sharing

- I/O Buffer (Kernel buffer)
  - Ring buffer: a large, continuous memory region
    - [1] Memory can be pre-allocated by process
      [2] or reserved by kernel
  - Provided buffer: multiple small, non-contiguous buffers
    - Use fragmented user-space memory to store data


User space
Kernel space
io_uring worker
Process
PROVIDE_ BUFFERS
Address: 0x7ffff7c00000
nbuf: 3
len: 0x100
Buffer ID: 0
I/O buffer array
Index
I/O Buffer
0
1
…
1. Send PROVIDE_BUFFERS request


User space
Kernel space
Process
io_uring worker
PROV_BUF
Buffer ID: 0
I/O buffer array
Index
I/O Buffer
0
Object
1
...
2. Create an I/O buffer object

User space

Process

Kernel space

io_uring worker

I/O buffer array

Index	I/O Buffer
0	Object
1	
…	

Address: 0x7ffff7c00000
nbuf: 3
len: 0x100

3. The address, size and count are stored in the object


User space
Kernel space
io_uring worker
Process
REMOVE_
BUFFERS
nbuf: 1
Buffer ID: 0
4. Send REMOVE_BUFFERS request
I/O buffer array
Index
I/O
Buffer
0
Object
1
…
Address: 0x7ffff7c00000
nbuf: 3
len: 0x100


User space
Kernel space
io_uring worker
Process
REMV_BUF
nbuf: 1
Buffer ID: 0
I/O buffer array
Index
I/O Buffer
0
Object
1
...
5. Remove several sub-buffers
Address: 0x7ffff7c00000
nbuf: 2
len: 0x100

$ Memory Sharing

- Both types of buffer use the same io_buffer_list structure
- How to identify the buffer type?
  - Determined by flag fields: is_mapped and is_mmap
  - These fields are initialized during object creation

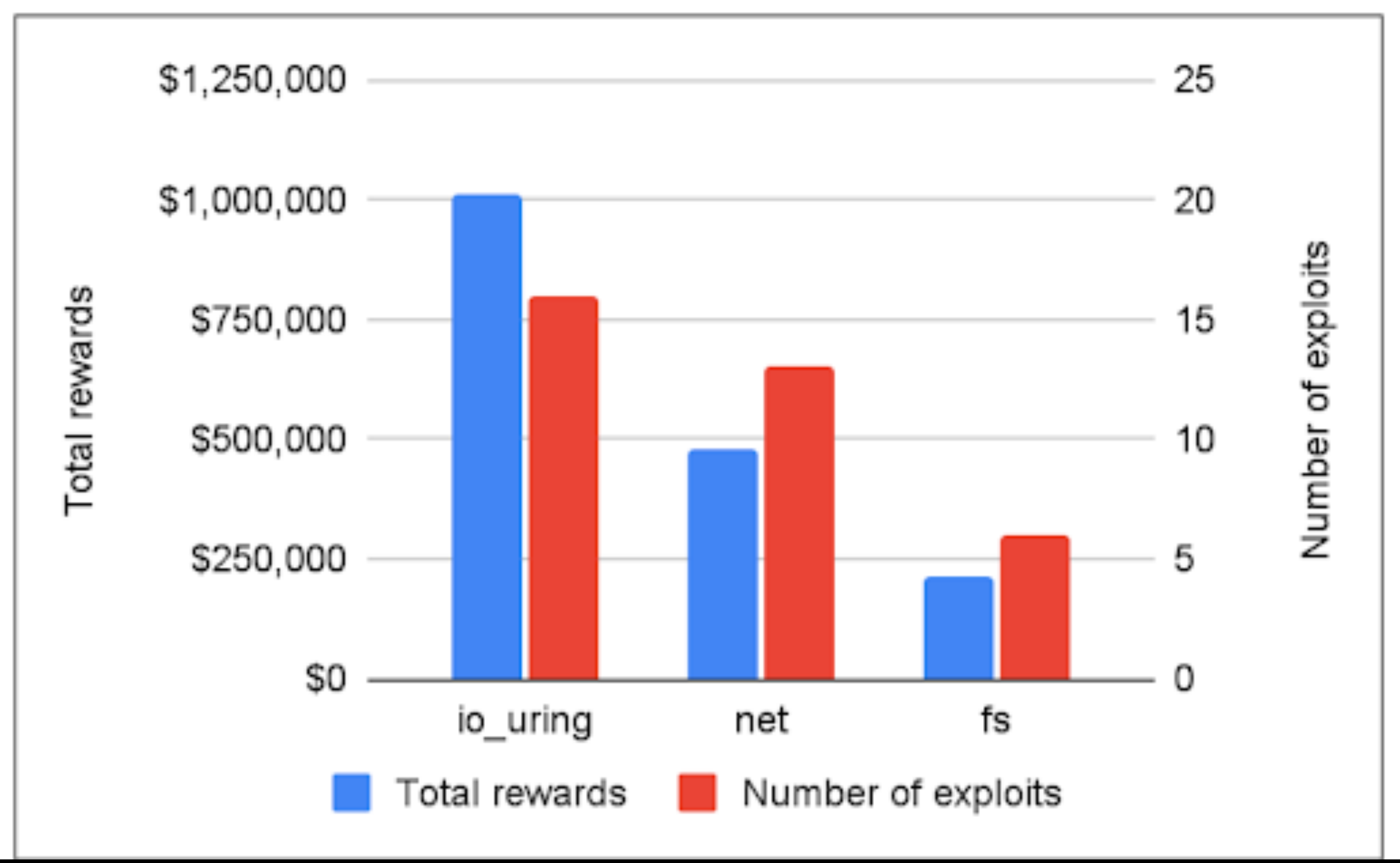
```c
struct io_buffer_list {
	 * If ->buf_nr_pages is set, then buf_pages/buf_ring are use
	 * then these are classic provided buffers and ->buf_list is
	 */
	union {
		struct list_head buf_list;
		struct {
			struct page **buf_pages;
			struct io_uring_buf_ring *buf_ring;
		};
		struct rcu_head rcu;
	};
	__u16 bgid;

	/* below is for ring provided buffers */
	__u16 buf_nr_pages;
	__u16 nr_entries;
	__u16 head;
	__u16 mask;

	atomic_t refs;
```

is_mapped	is_mmap	Type
0	0	Provided buffer
0	1	SHOULD NOT HAPPEN
1	0	Ring buffer (with pre-allocated memory)
1	1	Ring buffer (with reserved memory)

```c
/* ring mapped provided buffers */
__u8 is_mapped;
/* ring mapped provided buffers, but mmap'ed by application */
__u8 is_mmap;
```

The Evolution of Shared Memory

$ RECON



Limiting io_uring

To protect our users, we decided to limit the usage of io_uring in Google products:

- **ChromeOS:** We disabled io_uring (while we explore new ways to sandbox it).
- **Android:** Our seccomp-bpf filter ensures that io_uring is unreachable to apps.
  Future Android releases will use SELinux to limit io_uring access to a select few system processes.
- **GKE AutoPilot:** We are investigating disabling io_uring by default.
- It is disabled on production Google servers.

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

$ RECON

- Memory sharing
  - Easy to exploit
    - Often leads to strong primitives
  - Hard to maintain
    - State transitions across alloc/update/release make bugs more likely

$ Bug 1 - CVE-2024-0582

- io_uring/kbuf: Defer release of mapped buffer rings
  - Kernel reserves pages for the ring buffer
  - Mapped via remap_pfn_range() without incrementing refcount

$ Bug 1 - CVE-2024-0582

- io_uring/kbuf: Defer release of mapped buffer rings
  - Kernel reserves pages for the ring buffer
  - Mapped via remap_pfn_range() without incrementing refcount
  - Unregister an I/O buffer will also release its pages

$ Bug 1 - CVE-2024-0582

- io_uring/kbuf: Defer release of mapped buffer rings
  - Kernel reserves pages for the ring buffer
  - Mapped via remap_pfn_range() without incrementing refcount
  - Unregister an I/O buffer will also release its pages
  - However, the user mapping remains accessible, resulting in a page UAF


User space
Kernel space
Process
1. Register
- Flag: IOU_PBUF_RING_MMAP
- Index: 0
I/O buffer array
Index
I/O Buffer
0
Object
1
...
Page
Page
...


User space
Kernel space
Process
I/O buffer array
Index
I/O Buffer
0
Object
1
...
2. Map the these pages
Page
Page
...


User space
Kernel space
Process
3. Unregister
- Index: 0
I/O buffer array
Index
I/O Buffer
0
Object
1
...
Page
Page
...


User space
Kernel space
4. The I/O buffer object is freed along with its backing pages
I/O buffer array
Process
Index
I/O Buffer
0
Object
1
...
Page
Page
...


User space
Kernel space
Process
I/O buffer array
Index
I/O Buffer
0
1
...
5. These page are still accessible from user space
0x414141414141...

$ Bug 1 - CVE-2024-0582

- How to fix?
  - Prevent unregistration of mapped ring buffer

$ Bug 1 - CVE-2024-0582

- How to fix?
  - Prevent unregistration of mapped ring buffer
  - Update refcount of page or I/O buffer object via VMA op

$ Bug 1 - CVE-2024-0582

- How to fix?
  - Prevent unregistration of mapped ring buffer
  - Update refcount of page or I/O buffer object via VMA op
  - Add counter field in object to track mappings

$ Bug 1 - CVE-2024-0582

- How to fix?

  ❌ Prevent unregistration of mapped ring buffer

  ❌ Update refcount of page or I/O buffer object via VMA op

  ❌ Add counter field in object to track mappings

  ✅ Defer releasing these pages until io_uring context is closed


User space
Kernel space
1. Add a new field
io_buf_list
Process
I/O buffer array
Index
I/O Buffer
0
Object
1
...
Page
Page
...


User space
Kernel space
io_buf_list
Process
I/O buffer array
Index
I/O Buffer
0
Object
1
...
2. Unregistration frees I/O buffer,
but pages are still alive
Page
Page
...


User space
Kernel space
io_buf_list
Process
I/O buffer array
Index
I/O Buffer
0
1
...
3. Safely access!
Page
Page
...

$ Bug 1 - CVE-2024-0582

- Behind the patch
  - Introduce a new feature as a workaround
    - More code means more potential issues

$ Bug 1 - CVE-2024-0582

- Behind the patch
  - Introduce a new feature as a workaround
    - More code means more potential issues
  - Not using reference counting that precisely
    - Should hold a refcount either page or object

$ Bug 1 - CVE-2024-0582

- Behind the patch
  - Introduce a new feature as a workaround
    - More code means more potential issues
  - Not using reference counting that precisely
    - Should hold a refcount either page or object
  - Mixes functionality with lifetime management

$ Bug 2 - Lockdep

- io_uring: free io_buffer_list entries via RCU
  - Uncovered when addressing CVE-2024-0582
  - Deadlock between the io_uring context lock and the memory management (mm) lock

Process 1

Register an I/O buffer

```
def register(uaddr):
    mutex_lock(context_lock)

    if is_not_present(uaddr):
        lock(mm)
        handle_page_fault(uaddr)
        unlock(mm)

    copy_from_user(uaddr, &req)
    obj = new_io_buffer_obj(&req)
    io_buffer_obj_arr[req.idx] = obj

    mutex_unlock(context_lock)
```

Process 2

Map an I/O buffer to user space

```
def mmap_v1(idx):
    lock(mm)
    mutex_lock(context_lock)

    obj = io_buffer_obj_arr[idx]
    if can_mmap(obj):
        handle_memory_mapping(obj)

    mutex_unlock(context_lock)
    unlock(mm)
```

Process 1

Register an I/O buffer

```
def register(uaddr):
    mutex_lock(context_lock)

    if is_not_present(uaddr):
        lock(mm)
        handle_page_fault(uaddr)
        unlock(mm)

    copy_from_user(uaddr, &req)
    obj = new_io_buffer_obj(&req)
    io_buffer_obj_arr[req.idx] = obj

    mutex_unlock(context_lock)
```

Process 2

Map an I/O buffer to user space

```
def mmap_v1(idx):
    lock(mm)
    mutex_lock(context_lock)

    obj = io_buffer_obj_arr[idx]
    if can_mmap(obj):
        handle_memory_mapping(obj)

    mutex_unlock(context_lock)
    unlock(mm)
```

Process 1

```
def register(uaddr):
    mutex_lock(context_lock)
```

Hold context lock

```
        ):
        lock(mm)
        handle_page_fault(uaddr)
        unlock(mm)

    copy_from_user(uaddr, &req)
    obj = new_io_buffer_obj(&req)
    io_buffer_obj_arr[req.idx] = obj

    mutex_unlock(context_lock)
```

Process 2

```
def mmap_v1(idx):
    lock(mm)
    mutex_lock(context_lock)

    obj = io_buffer_obj_arr[idx]
    if can_mmap(obj):
        handle_memory_mapping(obj)

    mutex_unlock(context_lock)
    unlock(mm)
```

Process 1

```
def register(uaddr):
    mutex_lock(context_lock)

    if is_not_present(uaddr):
        lock(mm)
        handle_page_fault(uaddr)
        unlock(mm)

    copy_from_user(uaddr, &req)
    obj = new_io_buffer_obj(&req)
    io_buffer_obj_arr[req.idx] = obj

    mutex_unlock(context_lock)
```

Process 2

```
def mmap_v1(idx):
    lock(mm)
    t_lock)

    obj = io_buffer_obj_arr[idx]
    if can_mmap(obj):
        handle_memory_mapping(obj)

    mutex_unlock(context_lock)
    unlock(mm)
```

Hold mm lock

Process 1

```
def register(uaddr):
    mutex_lock(context_lock)

    if is_not_present(uaddr):
        lock(mm)
        handle_page_fault(uaddr)

    copy_from_user(uaddr, &req)
    obj = new_io_buffer_obj(&req)
    io_buffer_obj_arr[req.idx] = obj

    mutex_unlock(context_lock)
```

Wait for mm lock

Process 2

```
def mmap_v1(idx):
    lock(mm)
    mutex_lock(context_lock)

    obj = io_buffer_obj_arr[idx]
    if can_mmap(obj):
        handle_memory_mapping(obj)

    mutex_unlock(context_lock)
    unlock(mm)
```

Process 1

```
def register(uaddr):
    mutex_lock(context_lock)

    if is_not_present(uaddr):
        lock(mm)
        handle_page_fault(uaddr)

    copy_from_user(uaddr, &req)
    obj = new_io_buffer_obj(&req)
    io_buffer_obj_arr[req.idx] = obj

    mutex_unlock(context_lock)
```

Wait for mm lock

Process 2

```
def mmap_v1(idx):
    lock(mm)
    mutex_lock(context_lock)

    if can_mmap(obj):
        handle_memory_mapping(obj)

    mutex_unlock(context_lock)
    unlock(mm)
```

Wait for context lock

Process 1

```
def register(uaddr):
    mutex_lock(context_lock)

    if is_not_present(uaddr):
        lock(mm)
        handle_page_fault(uaddr)

    copy_from_user(uaddr, &req)
    obj = new_io_buffer_obj(&req)
    io_buffer_obj_arr[req.idx] = obj
```

Wait for mm lock

Process 2

```
def mmap_v1(idx):
    lock(mm)
    mutex_lock(context_lock)

    if can_mmap(obj):
        handle_memory_mapping(obj)

    mutex_unlock(context_lock)
```

Wait for context lock

DEADLOCK

$ Bug 2 - Lockdep

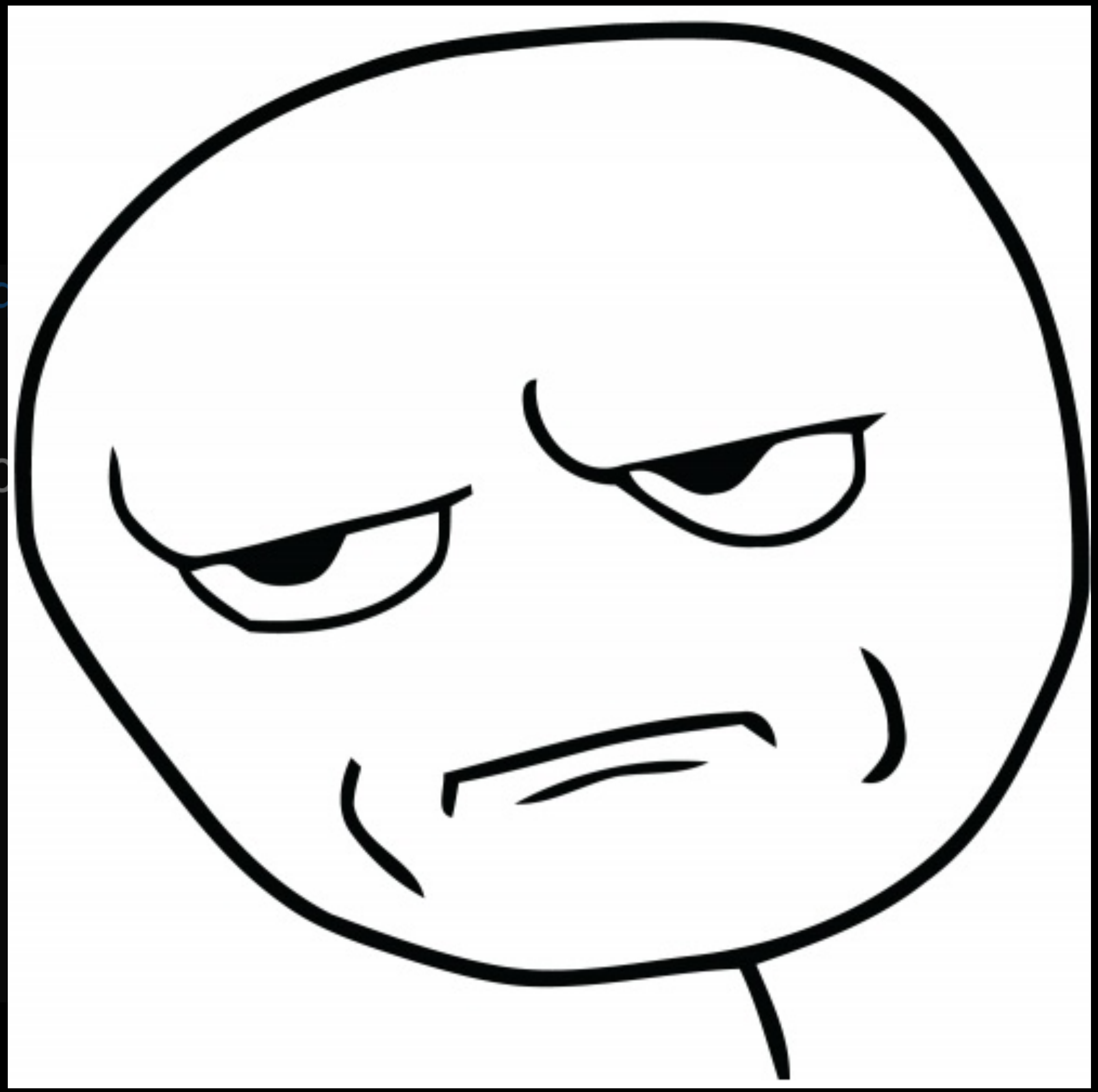
- The key patch:
  - Replace the mutex lock with an RCU lock

```
@@ -3498,9 +3498,9 @@ static void *io_uring_validate_mmap_request(struct file *file,
                unsigned int bgid;

                bgid = (offset & ~IORING_OFF_MMAP_MASK) >> IORING_OFF_PBUF_SHIFT;
-               mutex_lock(&ctx->uring_lock);
+               rcu_read_lock();
                ptr = io_pbuf_get_address(ctx, bgid);
-               mutex_unlock(&ctx->uring_lock);
+               rcu_read_unlock();
                if (!ptr)
                        return ERR_PTR(-EINVAL);
                break;
```

$ Bug 2 - Lockdep

- The key patch:
  - Replace the mutex lock with an RCU lock

```
@@ -3498,9 +3498,9 @@ static void *io_uring_validate_mmap_re
-               mutex_lock(&ctx->uring_lock);
+               rcu_read_lock();
                ptr = io_pbuf_get_address(ctx, bgid);
-               mutex_unlock(&ctx->uring_lock);
+               rcu_read_unlock();
+               rcu_read_unlock();
                if (!ptr)
                        return ERR_PTR(-EINVAL);
                break;
```



$ Bug 2 - Lockdep

- RCU (Read-Copy-Update)
  - Lock-free synchronization for read-mostly data
  - Old data is freed after readers finish
  - Writer API: synchronize_rcu(), call_rcu(), kfree_rcu()
  - Reader API: rcu_read_lock() & rcu_read_unlock()

$ Bug 2 - Lockdep

- Mutex Lock
  - Ensures only one thread accesses a shared resource at a time
    - Lead to a deadlock :(
- RCU Lock
  - Lock-free, while also protecting reads to the I/O buffer
  - However, it allows concurrent access!

$ Bug 2 - Lockdep

- Mutex Lock
  - Ensures only one thread accesses a shared resource at a time
    - Lead to a deadlock :(
- RCU Lock
  - Lock-free, while also protecting reads to the I/O buffer object
  - However, it allows concurrent access!

Process 1

```
def register(uaddr):
    mutex_lock(context_lock)

    if is_not_present(uaddr):
        lock(mm)
        handle_page_fault(uaddr)
        unlock(mm)

    copy_from_user(uaddr, &req)
    obj = new_io_buffer_obj(&req)
    io_buffer_obj_arr[req.idx] = obj

    mutex_unlock(context_lock)
```

Process 2

```
def mmap_v2(idx):
    lock(mm)
-   mutex_lock(context_lock)
+   rcu_read_lock()

    obj = io_buffer_obj_arr[idx]
    if obj.is_mmap:
        addr = obj.buf_ring
        map_address_to_user_space(addr)

-   mutex_unlock(context_lock)
+   rcu_read_unlock()
```

The mutex lock is replaced with an RCU lock

Process 1

```
def register(uaddr):
    mutex_lock(context_lock) ✅

    if is_not_present(uaddr):
        lock(mm)
        handle_page_fault(uaddr)
```

So the register handler is not blocked anymore

```
    obj = new_io_buffer_obj(&req)
    io_buffer_obj_arr[req.idx] = obj

    mutex_unlock(context_lock)
```

Process 2

```
def mmap_v2(idx):
    lock(mm)
-   mutex_lock(context_lock)
+   rcu_read_lock()

    obj = io_buffer_obj_arr[idx]
    if obj.is_mmap:
        addr = obj.buf_ring
        map_address_to_user_space(addr)

-   mutex_unlock(context_lock)
+   rcu_read_unlock()
    unlock(mm)
```

Process 1

```
def register(uaddr):
    mutex_lock(context_lock)

    if is_not_present(uaddr):
        lock(mm)
        handle_page_fault(uaddr)
        unlock(mm)

    copy_from_user(uaddr, &req)
    obj = new_io_buffer_obj(&req)
    io_buffer_obj_arr[req.idx] = obj
```

But It also means the registration handler may executes concurrently

Process 2

```
def mmap_v2(idx):
    lock(mm)
-   mutex_lock(context_lock)
+   rcu_read_lock()

    obj = io_buffer_obj_arr[idx]
    if obj.is_mmap:
        addr = obj.buf_ring
        map_address_to_user_space(addr)

-   mutex_unlock(context_lock)
+   rcu_read_unlock()
    unlock(mm)
```

$ Bug 3 - CVE-2024-35880

- io_uring/kbuf: hold io_buffer_list reference over mmap
  - After the previous patch, concurrent access to the I/O buffer is allowed
  - RCU prevents releasing I/O buffer objects, but not updating them

$ Bug 3 - CVE-2024-35880

- Unregistration does not free I/O buffer object immediately:
  1. Clear bl->is_mmap flag
  2. Reset bl->buf_list with INIT_LIST_HEAD()
  3. Finally, call kfree_rcu()

$ Bug 3 - CVE-2024-35880

- Unregistration does not free I/O buffer object immediately:
  1. Clear bl->is_mmap flag
  2. Reset bl->buf_list with INIT_LIST_HEAD()
  3. Finally, call kfree_rcu()
- Potential race between mmap handler and resource cleanup 😎

Process 1

Unregister an I/O buffer

```
def unregister(uaddr):
    mutex_lock(context_lock)

    copy_from_user(uaddr, &req)
    obj = io_buffer_obj_arr[req.idx]

    obj.refcount -= 1
    if obj.refcount == 0:
        obj.is_mmap = 0
        INIT_LIST_HEAD(&obj.buf_list)
        kfree_rcu(obj)

    mutex_unlock(context_lock)
```

Process 2

Map an I/O buffer to user space

```
def mmap_v2(idx):
    rcu_read_lock()

    obj = io_buffer_obj_arr[idx]
    if obj.is_mmap:
        addr = obj.buf_ring
        map_address_to_user_space(addr)

    rcu_read_unlock()
```

Process 1

Unregister an I/O buffer

```
def unregister(uaddr):
    mutex_lock(context_lock)

    copy_from_user(uaddr, &req)
    obj = io_buffer_obj_arr[req.idx]

    obj.refcount -= 1
    if obj.refcount == 0:
        obj.is_mmap = 0
        INIT_LIST_HEAD(&obj.buf_list)
        kfree_rcu(obj)

    mutex_unlock(context_lock)
```

Process 2

Map an I/O buffer to user space

```
def mmap_v2(idx):
    rcu_read_lock()

    obj = io_buffer_obj_arr[idx]
    if obj.is_mmap:
        addr = obj.buf_ring
        map_address_to_user_space(addr)

    rcu_read_unlock()
```

Process 1

```
def unregister(uaddr):
    mutex_lock(context_lock)

    copy_from_user(uaddr, &req)
    obj = io_buffer_obj_arr[req.idx]

    obj.refcount -= 1
    if obj.refcount == 0:
        obj.is_mmap = 0
        INIT_LIST_HEAD(&obj.buf_list)
        kfree_rcu(obj)

    mutex_unlock(context_lock)
```

Process 2

```
def mmap_v2(idx):
    rcu_read_lock()

    obj = io_buffer_obj_arr[idx]
    if obj.is_mmap:
        addr = obj.buf_ring
                  space(addr)

    rcu_read_unlock()
```

Pass the check

Process 1

```
def unregister(uaddr):
    mutex_lock(context_lock)

    copy_from_user(uaddr, &req)
    obj = io_buffer_obj_arr[req.idx]

    obj.refcount -= 1
    if obj.refcount == 0:
        obj.is_mmap = 0
        INIT_LIST_HEAD(&obj.buf_list)

    mutex_unlock(context_lock)
```

Reset the linked list buf_list

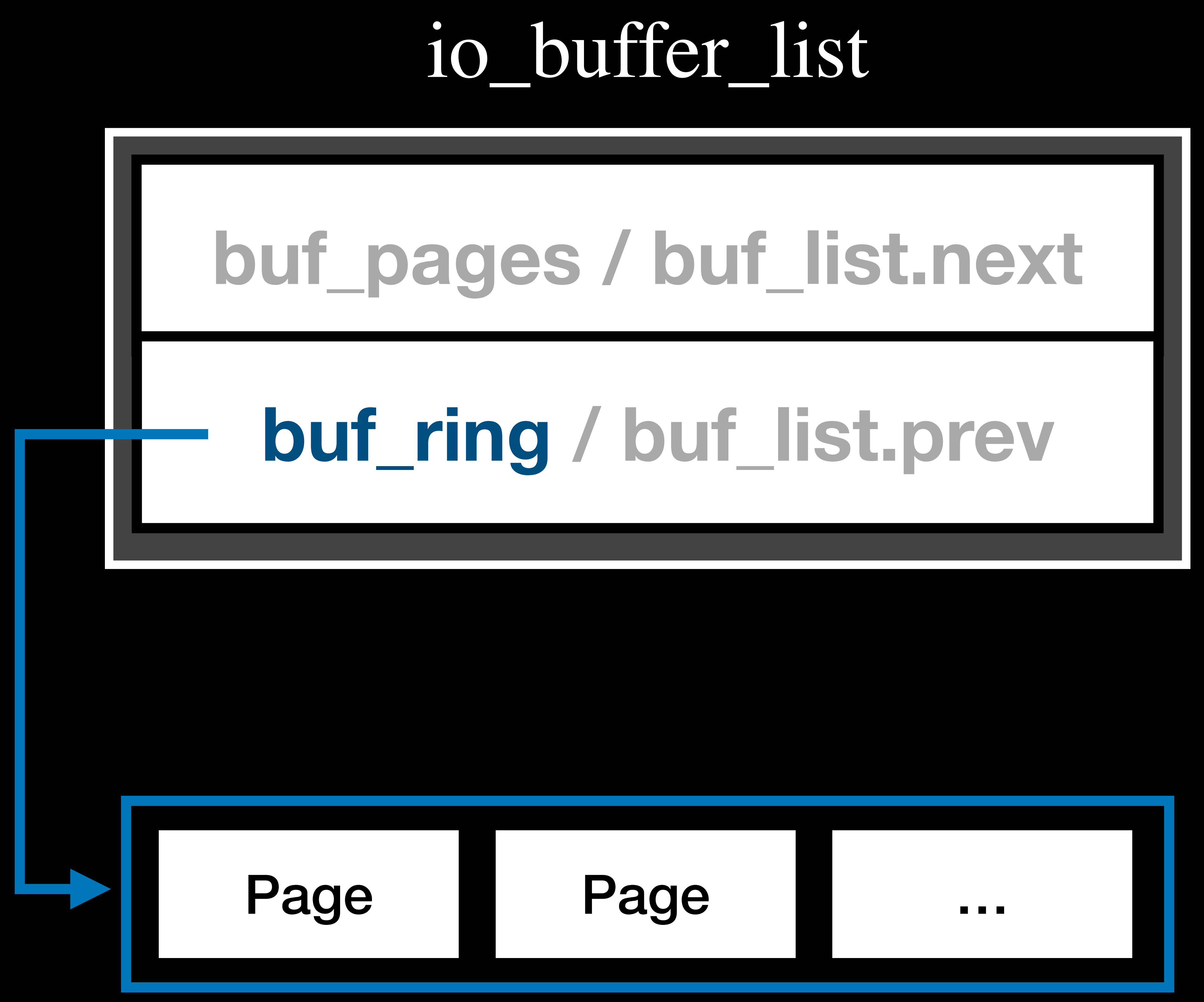
Process 2

```
def mmap_v2(idx):
    rcu_read_lock()

    obj = io_buffer_obj_arr[idx]
    if obj.is_mmap:
        addr = obj.buf_ring
        map_address_to_user_space(addr)

    rcu_read_unlock()
```

```c
struct io_buffer_list {
	/*
	 * If ->buf_nr_pages is set, then buf_pages/buf_ring are used. If not,
	 * then these are classic provided buffers and ->buf_list is used.
	 */
	union {
		struct list_head buf_list;
		struct {
			struct page **buf_pages;
			struct io_uring_buf_ring *buf_ring;
		};
	};
	__u16 mask;

	atomic_t refs;

	/* ring mapped provided buffers */
	__u8 is_mapped;
	/* ring mapped provided buffers, but mmap'ed by application */
	__u8 is_mmap;
};
```

Process 1

```
def unregister(uaddr):
    mutex_lock(context_lock)

    copy_from_user(uaddr, &req)
    obj = io_buffer_obj_arr[req.idx]

    obj.refcount -= 1
    if obj.refcount == 0:
        obj.is_mmap = 0
        INIT_LIST_HEAD(&obj.buf_list)
        kfree_rcu(obj)

    mutex_unlock(context_lock)
```



Originally, buf_ring points to shared memory

Process 1

```
def unregister(uaddr):
    mutex_lock(context_lock)

    copy_from_user(uaddr, &req)
    obj = io_buffer_obj_arr[req.idx]

    obj.refcount -= 1
    if obj.refcount == 0:
        obj.is_mmap = 0
        INIT_LIST_HEAD(&obj.buf_list)
        kfree_rcu(obj)

    mutex_unlock(context_lock)
```

Call INIT_LIST_HEAD() to update it to point to object itself

Process 1

```
def unregister(uaddr):
    mutex_lock(context_lock)

    copy_from_user(uaddr, &req)
    obj = io_buffer_obj_arr[req.idx]

    obj.refcount -= 1
    if obj.refcount == 0:
        obj.is_mmap = 0
        INIT_LIST_HEAD(&obj.buf_list)
        kfree_rcu(obj)

    mutex_unlock(context_lock)
```

Process 2

```
def mmap_v2(idx):
    rcu_read_lock()

    obj = io_buffer_obj_arr[idx]
    if obj.is_mmap:
        addr = obj.buf_ring
        map_address_to_user_space(addr)

    rcu_read_unlock()
```

Get the address from buf_ring

Process 2


```python
def mmap_v2(idx):
    rcu_read_lock()

    obj = io_buffer_obj_arr[idx]
    if obj.is_mmap:
        addr = obj.buf_ring
        map_address_to_user_space(addr)

    rcu_read_unlock()
```

But the buf_ring now points to the starting address of object

Process 2

```
def mmap_v2(idx):
    rcu_read_lock()

    obj = io_buffer_obj_arr[idx]
    if obj.is_mmap:
        addr = obj.buf_ring
        map_address_to_user_space(addr)
```

Map I/O buffer object to user space

$ Bug 3 - CVE-2024-35880

- RCU are introduced to prevent deadlock
  - Re-enable concurrent access to I/O buffer objects

$ Bug 3 - CVE-2024-35880

- RCU are introduced to prevent deadlock
  - Re-enable concurrent access to I/O buffer objects
- Lead to a race condition in unregistration process
  - Concurrently retrieve the ring buffer from the reinitialized union field causes incorrect memory mapping

$ Bug 3 - CVE-2024-35880

- Fixes:
    - Update reference count to prevent early unregistration
    - When the refcount reaches zero, the buffer can no longer be mapped

```
+           rcu_read_lock();
+           bl = xa_load(&ctx->io_bl_xa, bgid);
+           /* must be a mmap'able buffer ring and have pages */
+           ret = false;
+           if (bl && bl->is_mmap)
+                   ret = atomic_inc_not_zero(&bl->refs);
+           rcu_read_unlock();
```

Process 1

```
def unregister(uaddr):
    mutex_lock(context_lock)

    copy_from_user(uaddr, &req)
    obj = io_buffer_obj_arr[req.idx]

    obj.refcount -= 1
    if obj.refcount == 0:
        obj.is_mmap = 0
        INIT_LIST_HEAD(&obj.buf_list)
        kfree_rcu(obj)

    mutex_unlock(context_lock)
```

Process 2

```
def mmap_v3(idx):
    rcu_read_lock()

    obj = io_buffer_obj_arr[idx]

    if obj.is_mmap:
+       if atomic_inc_not_zero(&obj.refcount) == 0:
+           early return

        addr = obj.buf_ring
        map_address_to_user_space(addr)

+       if atomic_dec_and_test(&obj.refcount) == 0:
+           kfree_rcu(obj)
```

Update refcount during memory mapping

Process 1

```
def unregister(uaddr):
    mutex_lock(context_lock)

    copy_from_user(uaddr, &req)
    obj = io_buffer_obj_arr[req.idx]

    obj.refcount -= 1
    if obj.refcount == 0:
        obj.is_mmap = 0

    mutex_unlock(context_lock)
```

The ring_buf can no longer be destroyed concurrently

Process 2

```
def mmap_v3(idx):
    rcu_read_lock()

    obj = io_buffer_obj_arr[idx]

    if obj.is_mmap:
+       if atomic_inc_not_zero(&obj.refcount) == 0:
+           early return

        addr = obj.buf_ring
        map_address_to_user_space(addr)

+       if atomic_dec_and_test(&obj.refcount) == 0:
+           kfree_rcu(obj)

    rcu_read_unlock()
```

$ Mitigations

- By now:
  1. Deferred page release prevents the mapped memory from being freed
  2. RCU protection ensures the I/O buffer object is not freed too early
  3. Correct refcount updates prevent concurrent resets

$ Mitigations

- By now:
    1. Deferred page release prevents the mapped memory from being freed
    2. RCU protection ensures the I/O buffer object is not freed too early
    3. Correct refcount updates prevent concurrent resets
- Unbreakable?

Breaking I/O Buffer Again:

CVE-2025-21836

$ Insight

- According to the patch for CVE-2024-35880:
  - Concurrent access to the I/O buffer is still allowed

$ Insight

- According to the patch for CVE-2024-35880:
  - Concurrent access to the I/O buffer is still allowed
- Re-registering an existing I/O buffer will modify object fields as well
  - Also called as “upgrading”
  - Reuse an empty provided buffer as ring buffer

$ Insight

- According to the patch for CVE-2024-35880:
  - Concurrent access to the I/O buffer is still allowed
- Re-registering an existing I/O buffer will modify object fields as well
  - Also called as “upgrading”
  - Reuse an empty provided buffer as ring buffer
- Let’s revisit the registration process and see how it works!

```
def register(uaddr):
    mutex_lock(context_lock)

    copy_from_user(uaddr, &req)
    existing_obj = io_buffer_obj_arr[req.idx]

    if existing_obj and not is_empty_provided_buffer(obj):
        early return

    if existing_obj:
        obj = existing_obj
    else:
        obj = new_io_buffer_obj(&req)

    if reserved_memory_request(req):
        obj.is_mapped = 1
        obj.is_mmap = 1
        obj.buf_ring = alloc_pages()

    obj.refcount = 1
    io_buffer_obj_arr[req.idx] = obj

    mutex_unlock(context_lock)
```

```
def register(uaddr):
    mutex_lock(context_lock)

    copy_from_user(uaddr, &req)
    existing_obj = io_buffer_obj_arr[req.idx]
```

```
if existing_obj and not is_empty_provided_buffer(obj):
    early return
```

Only an empty provided buffer can be re-registered

```
    if reserved_memory_request(req):
        obj.is_mapped = 1
        obj.is_mmap = 1
        obj.buf_ring = alloc_pages()

    obj.refcount = 1
    io_buffer_obj_arr[req.idx] = obj

    mutex_unlock(context_lock)
```

```
def register(uaddr):
    mutex_lock(context_lock)

    copy_from_user(uaddr, &req)
    existing_obj = io_buffer_obj_arr[req.idx]

    if existing_obj and not is_empty_provided_buffer(obj):
        early return

    if reserved_memory_request(req):
        obj.is_mapped = 1
        obj.is_mmap = 1
        obj.buf_ring = alloc_pages()

    obj.refcount = 1
    io_buffer_obj_arr[req.idx] = obj
```

Do the same things as in the first registration

$ Root Cause

- At a glance, these seem safe operations:
  - Code reuse is common, so some dummy behavior is expected

$ Root Cause

- At a glance, these seem safe operations:
  - Code reuse is common, so some dummy behavior is expected
  - A provided buffer is not allowed to be mapped (bl->is_mmap is false)
    - So memory mapping will early return :(

$ Root Cause

- At a glance, these seem safe operations:
  - Code reuse is common, so some dummy behavior is expected
  - A provided buffer is not allowed to be mapped (bl->is_mmap is false)
    - So memory mapping will early return :(
    - … won’t it? 🤔

Process 1

Upgrade a provided buffer

```
def register(uaddr):
    mutex_lock(context_lock)

    # [...]

    obj = existing_obj

    if reserved_memory_request(req):
        obj.is_mapped = 1
        obj.is_mmap = 1
        obj.buf_ring = alloc_pages()

    obj.refcount = 1
    io_buffer_obj_arr[req.idx] = obj

    mutex_unlock(context_lock)
```

Process 2

Map an I/O buffer to user space

```
def mmap_v3(idx):
    rcu_read_lock()

    obj = io_buffer_obj_arr[idx]

    if obj.is_mmap:
        if atomic_inc_not_zero(&obj.refcount) == 0:
            early return

        addr = obj.buf_ring
        map_address_to_user_space(addr)

        if atomic_dec_and_test(&obj.refcount) == 0:
            kfree_rcu(obj)

    rcu_read_unlock()
```

Process 1

```
def register(uaddr):
    mutex_lock(context_lock)

    # [...]

    obj = existing_obj

    if reserved_memory_request(req):
        obj.is_mapped = 1
        obj.is_mmap = 1
        obj.buf_ring = alloc_pages()

    obj.refcount = 1
    io_buffer_obj_arr[req.idx] = obj

    mutex_unlock(context_lock)
```

Process 2

```
def mmap_v3(idx):
    rcu_read_lock()

    obj = io_buffer_obj_arr[idx]

    if obj.is_mmap:
        if atomic_inc_not_zero(&obj.refcount) == 0:

            map_address_to_user_space(addr)

            if atomic_dec_and_test(&obj.refcount) == 0:
                kfree_rcu(obj)

    rcu_read_unlock()
```

A provided buffer cannot be mapped since is_mmap = 0

Process 1

```
def register(uaddr):
    mutex_lock(context_lock)

    # [...]

    obj = existing_obj

    if reserved_memory_request(req):
        obj.is_mapped = 1
        obj.is_mmap = 1
        obj.buf_ring = alloc_pages()

    io_buffer_obj_arr[req.idx] = obj

    mutex_unlock(context_lock)
```

Process 2

```
def mmap_v3(idx):
    rcu_read_lock()

    obj = io_buffer_obj_arr[idx]

    if obj.is_mmap: ✅
        if atomic_inc_not_zero(&obj.refcount) == 0:
            early return

        addr = obj.buf_ring
        map_address_to_user_space(addr)

        if atomic_dec_and_test(&obj.refcount) == 0:
            kfree_rcu(obj)

    rcu_read_unlock()
```

But upgrading can make it mappable

Process 1

```
def register(uaddr):
    mutex_lock(context_lock)

    # [...]

    obj = existing_obj

    if reserved_memory_request(req):
        obj.is_mapped = 1
        obj.is_mmap = 1
        obj.buf_ring = alloc_pages()

    obj.refcount = 1
    io_buffer_obj_arr[req.idx] = obj

    mutex_unlock(context_lock)
```

Process 2

```
def mmap_v3(idx):
    rcu_read_lock()

    obj = io_buffer_obj_arr[idx]

    if obj.is_mmap:
        if atomic_inc_not_zero(&obj.refcount) == 0:
            early return

        if atomic_dec_and_test(&obj.refcount) == 0:
            kfree_rcu(obj)

    rcu_read_unlock()
```

Increase the refcount from 1 to 2

Process 1

```
def register(uaddr):
    mutex_lock(context_lock)

    # [...]

    obj = existing_obj

    if reserved_memory_request(req):
        obj.is_mapped = 1
        obj.is_mmap = 1
        obj.buf_ring = alloc_pages()

    obj.refcount = 1
```

The hardcoded assignment sets refcount to 1

Process 2

```
def mmap_v3(idx):
    rcu_read_lock()

    obj = io_buffer_obj_arr[idx]

    if obj.is_mmap:
        if atomic_inc_not_zero(&obj.refcount) == 0:
            early return

        addr = obj.buf_ring
        map_address_to_user_space(addr)

        if atomic_dec_and_test(&obj.refcount) == 0:
            kfree_rcu(obj)

    rcu_read_unlock()
```

Process 1

```
def register(uaddr):
    mutex_lock(context_lock)

    # [...]

    obj = existing_obj

    if reserved_memory_request(req):
        obj.is_mapped = 1
        obj.is_mmap = 1
        obj.buf_ring = alloc_pages()

    obj.refcount = 1
    io_buffer_obj_arr[req.idx] = obj

    mutex_unlock(context_lock)
```

Process 2

```
def mmap_v3(idx):
    rcu_read_lock()

    obj = io_buffer_obj_arr[idx]

    if obj.is_mmap:
        if atomic_inc_not_zero(&obj.refcount) == 0:
            early return

        addr = obj.buf_ring
        map_address_to_user_space(addr)

        if atomic_dec_and_test(&obj.refcount) == 0:
            kfree_rcu(obj)
```

Decrease refcount from 1 to 0

Process 1

```
def register(uaddr):
    mutex_lock(context_lock)

    # [...]

    obj = existing_obj

    if reserved_memory_request(req):
        obj.is_mapped = 1
        obj.is_mmap = 1
        obj.buf_ring = alloc_pages()

    obj.refcount = 1
    io_buffer_obj_arr[req.idx] = obj
```

Process 2

```
def mmap_v3(idx):
    rcu_read_lock()

    obj = io_buffer_obj_arr[idx]

    if obj.is_mmap:
        if atomic_inc_not_zero(&obj.refcount) == 0:
            early return

        addr = obj.buf_ring
        map_address_to_user_space(addr)

        if atomic_dec_and_test(&obj.refcount) == 0:
            kfree_rcu(obj)
```

I/O buffer object UAF

$ Root Cause

- This race is very hard to hit
  - Require memory mapping and upgrading in a specific order
- Once triggered, it allows access to the freed I/O buffer object
  - UAF on io_buffer_list
  - Full control over the mapped address

$ Exploitation

1. Environment setup
2. Try to hit the race
3. Side channel via mmap return value to detect success
4. Wait 5 seconds for RCU drain
5. Reclaim the freed buffer object via spraying
6. Overwrite the kernel data core_pattern[]

$ Exploitation

1. Environment setup
2. Try to hit the race
3. Side channel via mmap return value to detect success
4. Wait 5 seconds for RCU drain
5. Reclaim the freed buffer object via spraying
6. Overwrite the kernel data core_pattern[]

$ Exploitation

- Target object: struct io_buffer_list
  - kmalloc-64-cg (provided buffer)
  - kmalloc-64 (ring buffer)
- Reclaim using message queue mechanism
  - Spray many struct msg_msgseg

$ Exploitation

- kfree_rcu()
  - Tiny RCU: directly calls call_rcu()
  - Tree RCU:
    - Freed objects are batched
    - Batches are drained every 5 seconds (KFREE_DRAIN_JIFFIES)
- kernelCTF uses Tree RCU

$ Exploitation

1. Environment setup
2. Try to hit the race
3. Side channel via mmap return value to detect success
4. Wait 5 seconds for RCU drain
5. Reclaim the freed buffer object via spraying
6. Overwrite the kernel data core_pattern[]

$ Exploitation

- If race fails:
  - Does not lead to a kernel panic
  - The refcount of targeted io_buffer_list is zero

$ Exploitation

- Side-channel race result by mmap
    - Race fails: the mapping handler behaves normally
    - Race succeeds: the mmap handler sees a zero refcount and returns an error

$ Exploitation

- Side-channel race result by mmap
  - Race fails: the mapping handler behaves normally
  - Race succeeds: the mmap handler sees a zero refcount and returns an error
    - Thanks to atomic_inc_not_zero() 😊

$ Exploitation

- KASLR bypass
    - Not provide an information leak primitive :(

$ Exploitation

- KASLR bypass
  - Not provide an information leak primitive :(
  - EntryBleed (CVE-2022-4543): time-based side-channel on x86_64
    - Measure prefetch timing of kernel addresses
    - Leak entry_SYSCALL_64 address
    - Technique by William

$ Exploitation

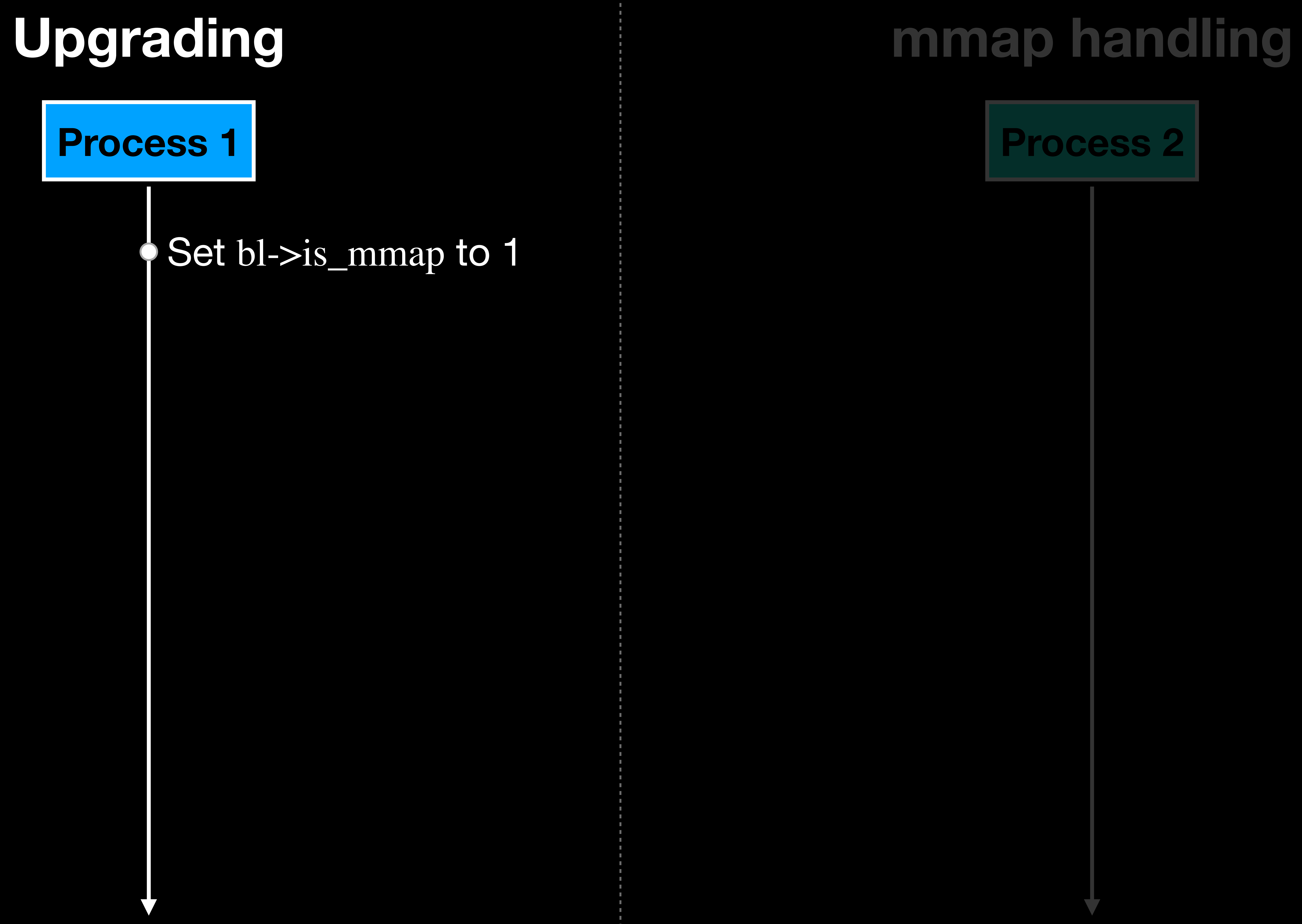
- Read the flag
    1. Set corrupted bl->buf_ring to the kernel variable core_pattern[]
    2. Map the memory into user space
    3. Overwrite it with our executable path
    4. Trigger a SEGFAULT and get flag!

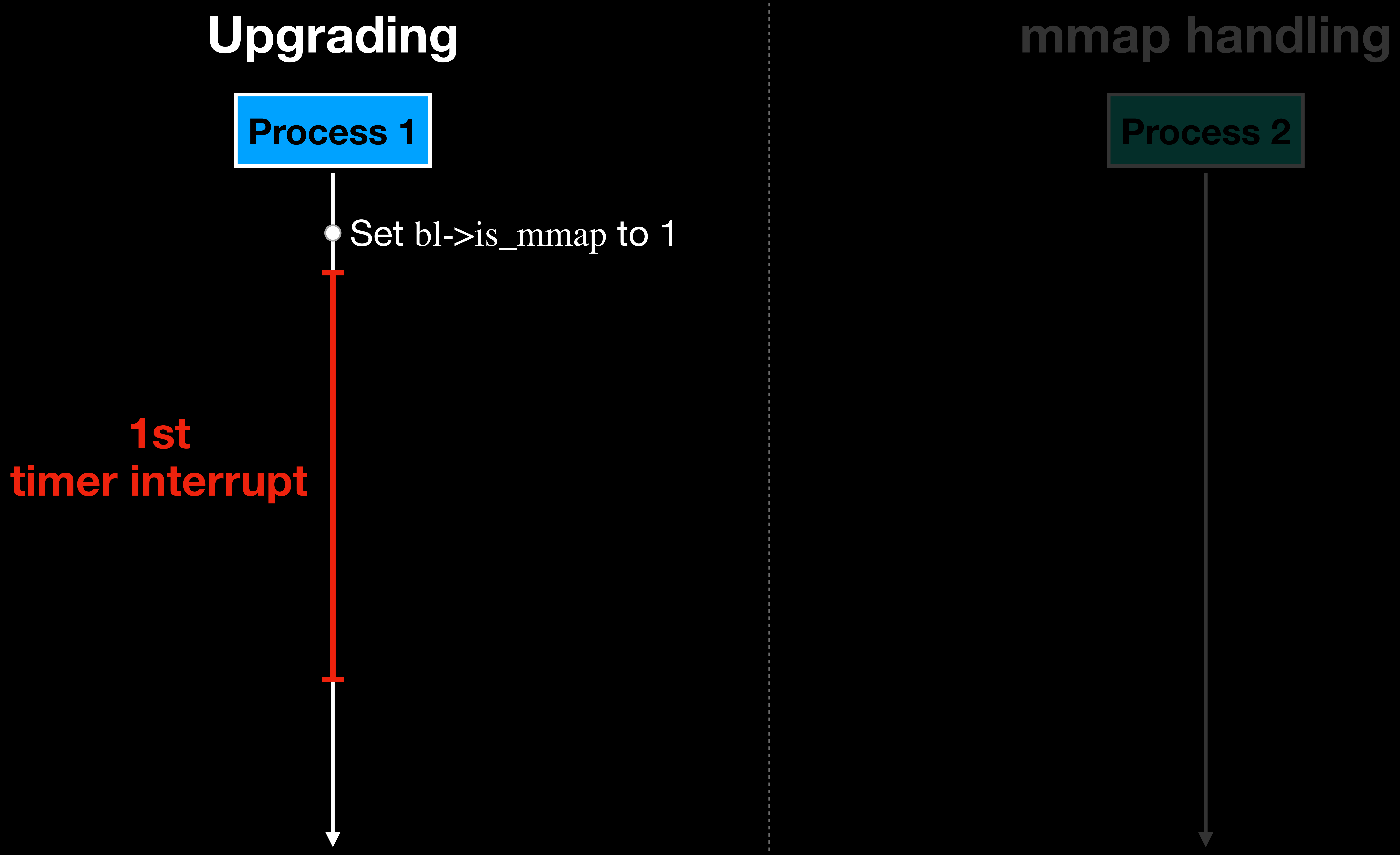
$ Exploitation

1. Environment setup
2. Try to hit the race
3. Side channel via mmap return value to detect success
4. Wait 5 seconds for RCU drain
5. Reclaim the freed buffer object via spraying
6. Overwrite the kernel data core_pattern[]

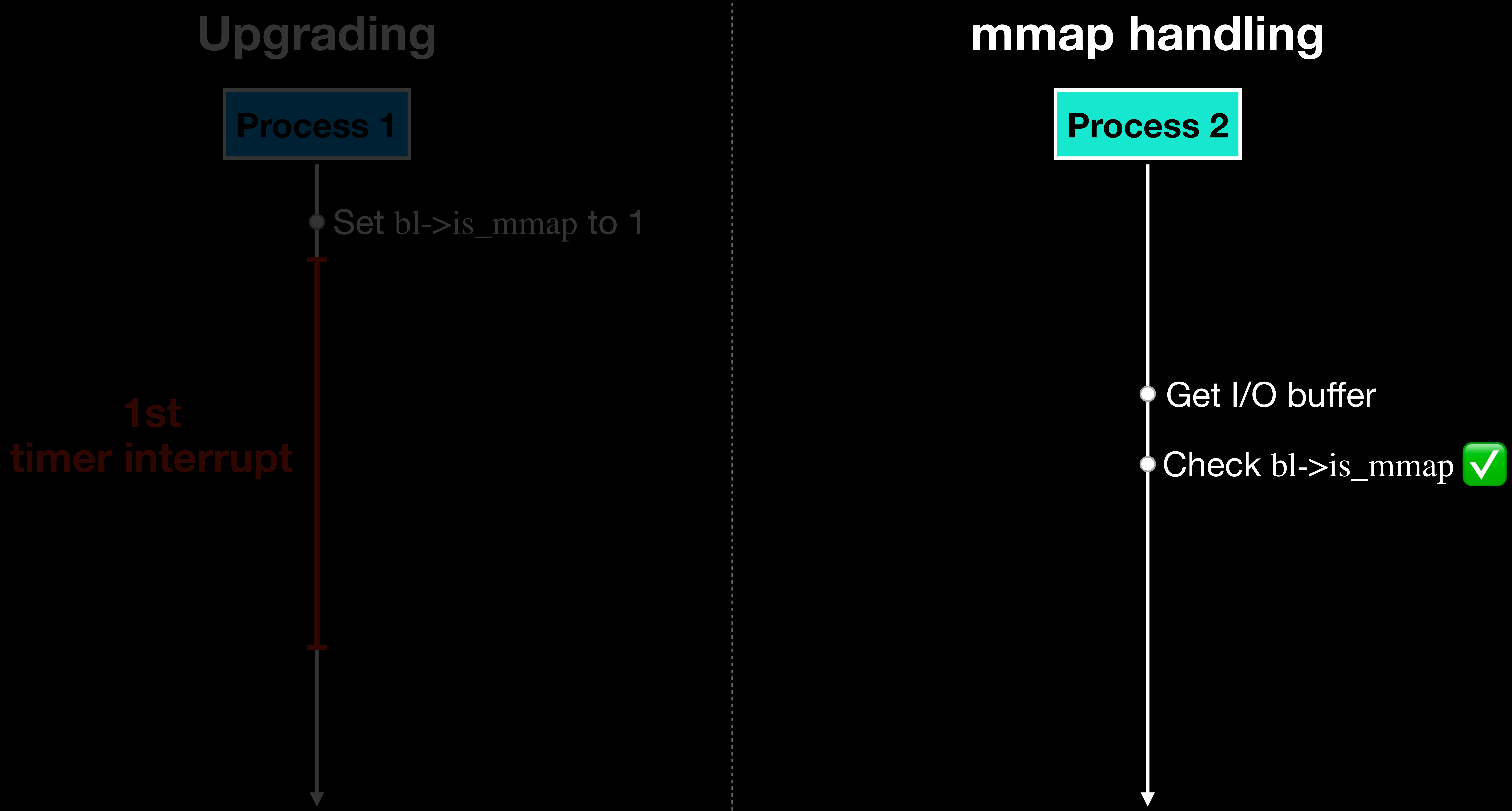
$ Exploitation

- Extending the race window with timer interrupts
  - Enqueue many timerfd waiters
    - The timer interrupt handler spends more time iterating the list
  - Proposed by Jann Horn
- In our case, we need two timerfds due to the narrow race window


Upgrading
mmap handling
Process 1
Process 2
Set bl->is_mmap to 1


Upgrading
mmap handling
Process 1
Process 2
Set bl->is_mmap to 1
1st
timer interrupt


Upgrading
Process 1
Set bl->is_mmap to 1
1st
timer interrupt
mmap handling
Process 2
Get I/O buffer
Check bl->is_mmap


Upgrading
Process 1
Set bl->is_mmap to 1
1st
timer interrupt
mmap handling
Process 2
Get I/O buffer
Check bl->is_mmap ✅

Upgrading

Process 1

- Set bl->is_mmap to 1

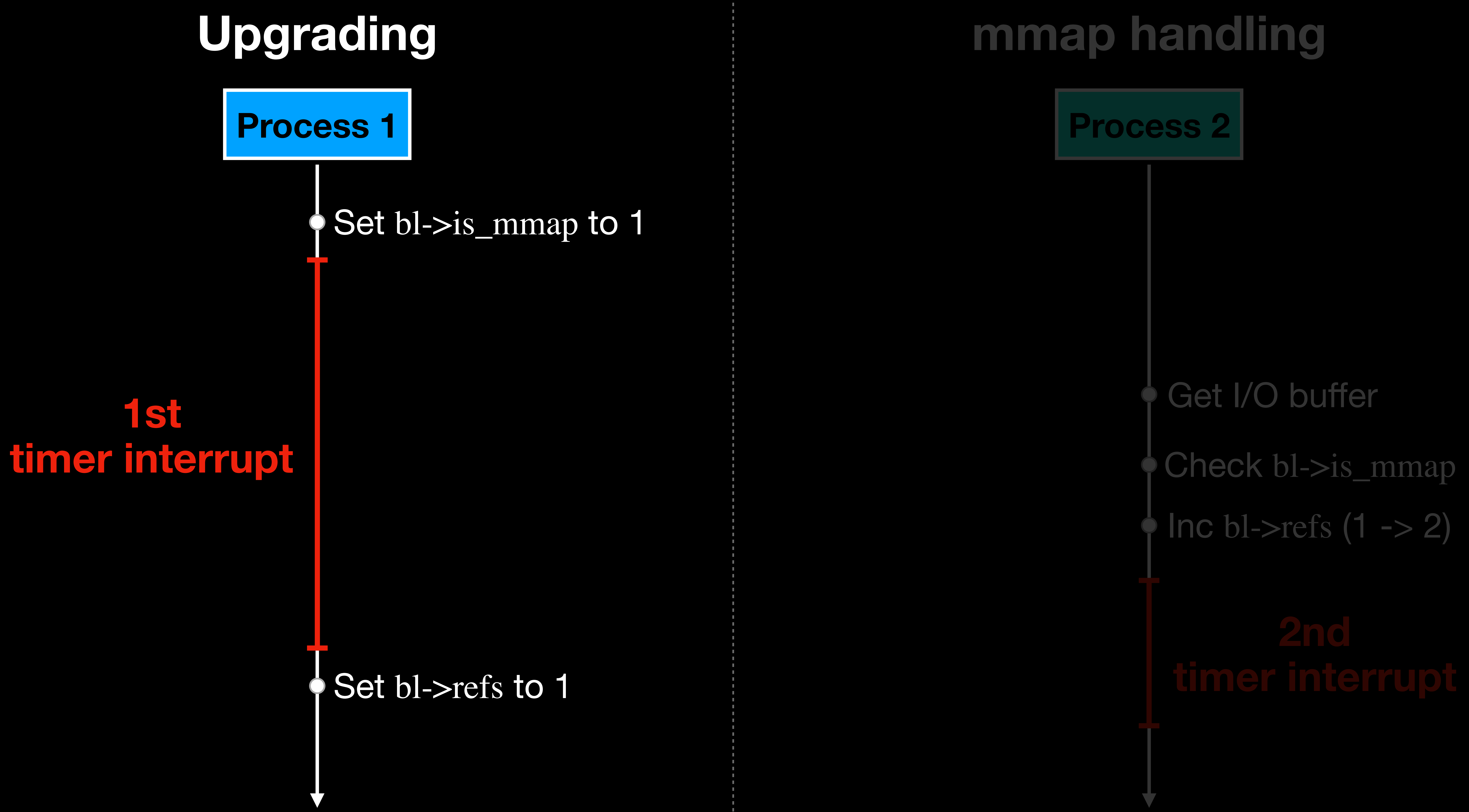
1st
timer interrupt

mmap handling

Process 2

- Get I/O buffer
- Check bl->is_mmap
- Inc bl->refs (1 -> 2)

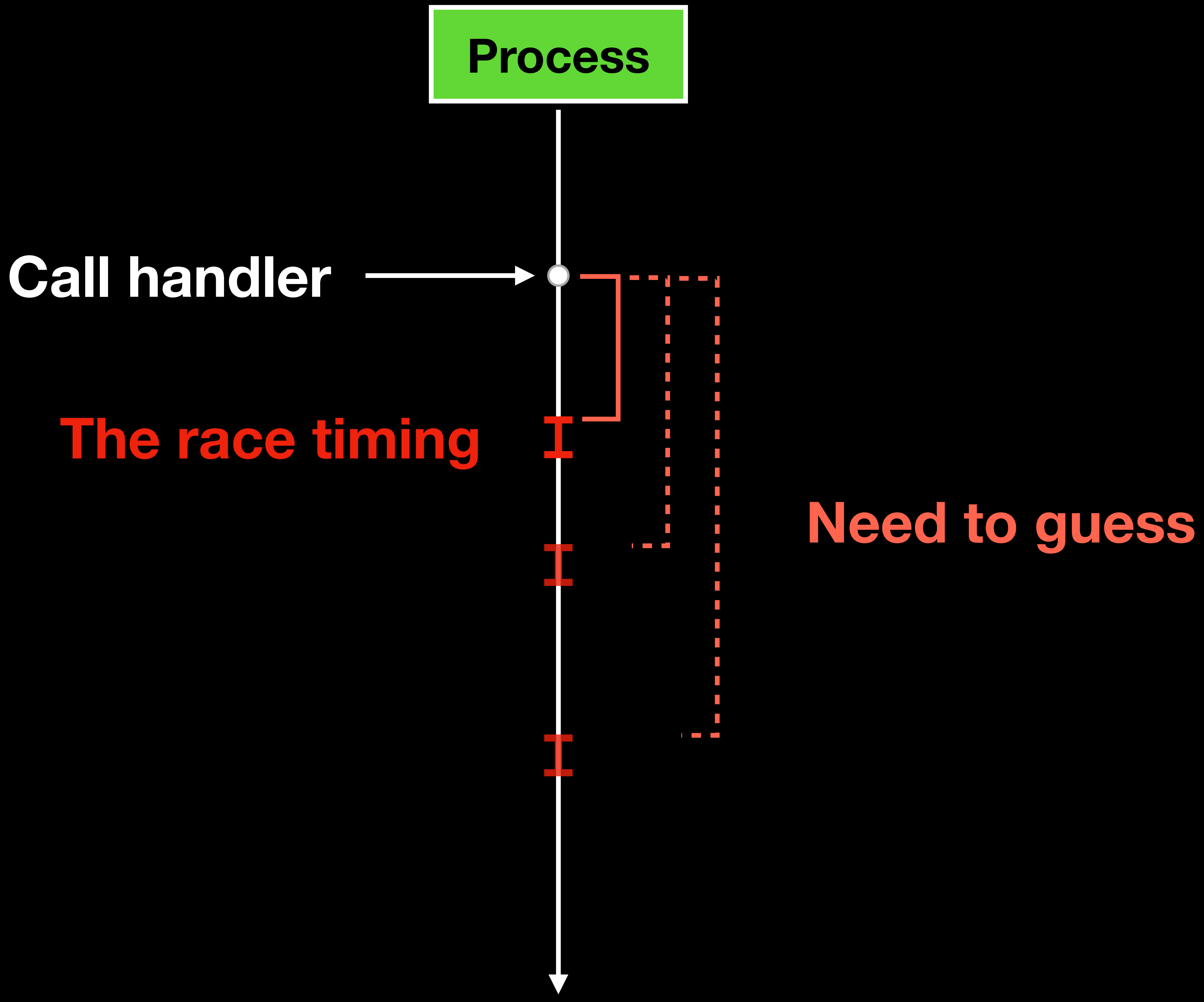

Upgrading
Process 1
Set bl->is_mmap to 1
1st
timer interrupt
mmap handling
Process 2
Get I/O buffer
Check bl->is_mmap
Inc bl->refs (1 -> 2)
2nd
timer interrupt

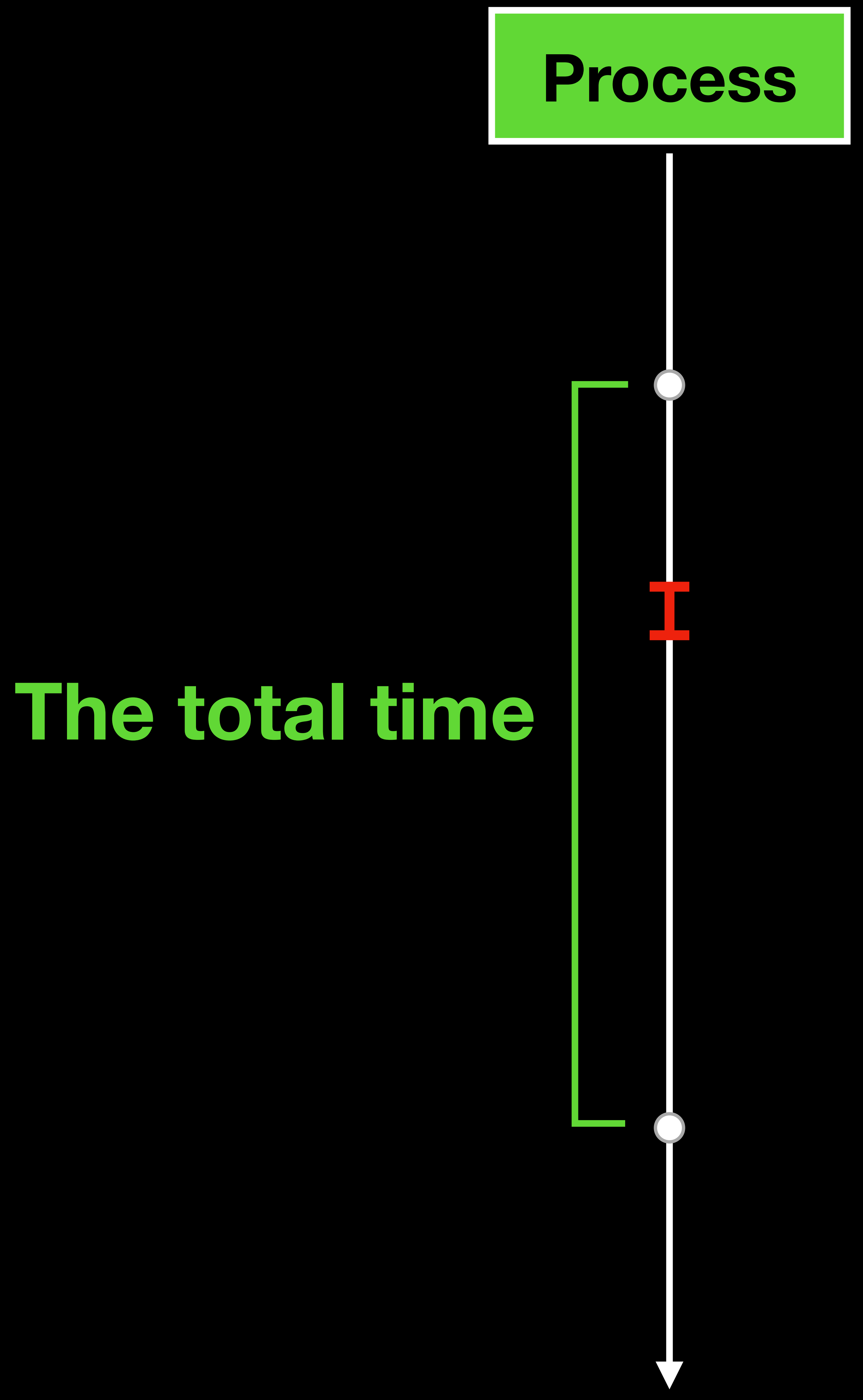

Upgrading
Process 1
Set bl->is_mmap to 1
1st
timer interrupt
Set bl->refs to 1
mmap handling
Process 2
Get I/O buffer
Check bl->is_mmap
Inc bl->refs (1 -> 2)
2nd
timer interrupt

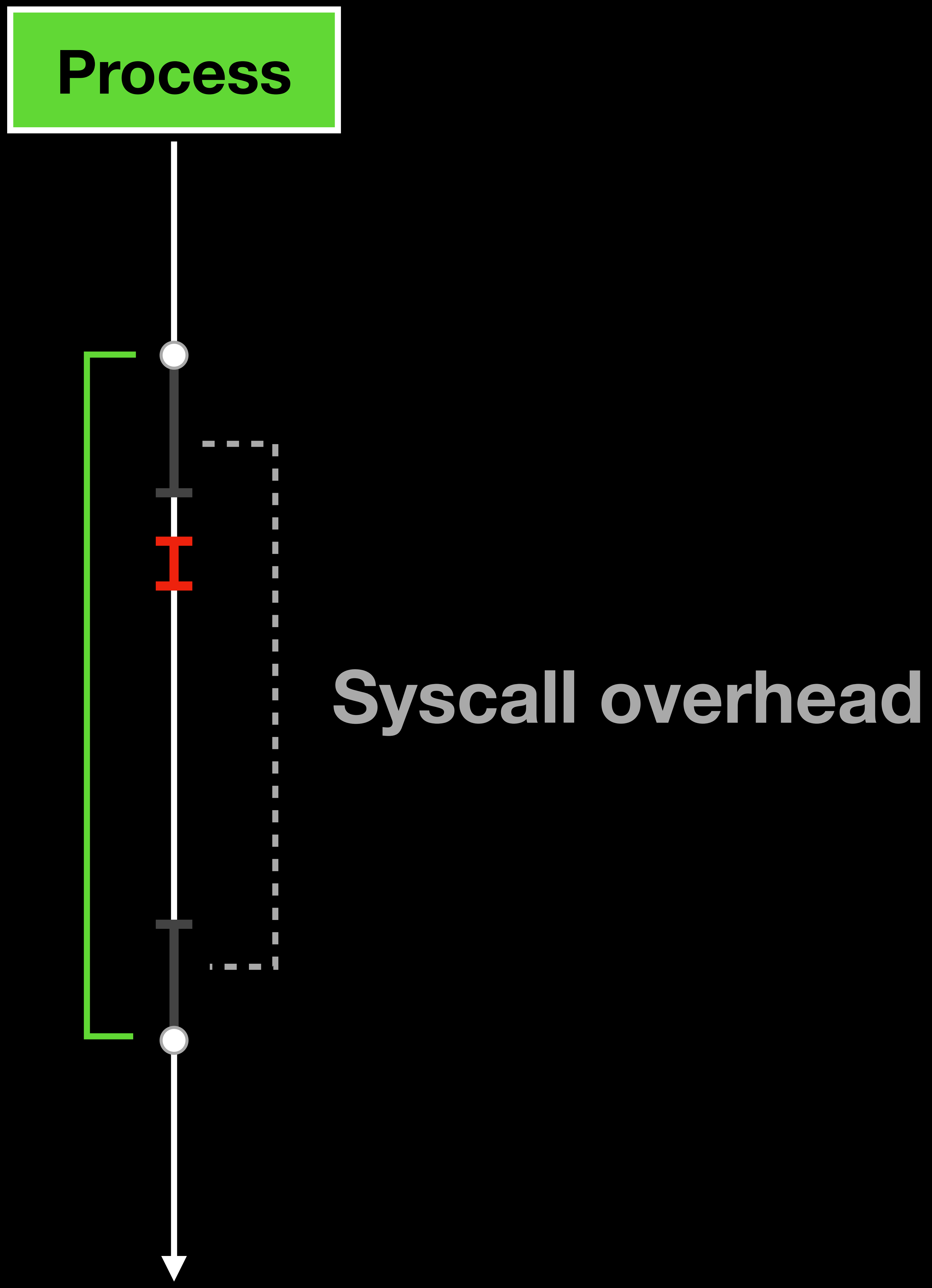

Upgrading
Process 1
Set bl->is_mmap to 1
1st
timer interrupt
Set bl->refs to 1
mmap handling
Process 2
Get I/O buffer
Check bl->is_mmap
Inc bl->refs (1 -> 2)
2nd
timer interrupt
Dec bl->refs (1 -> 0)

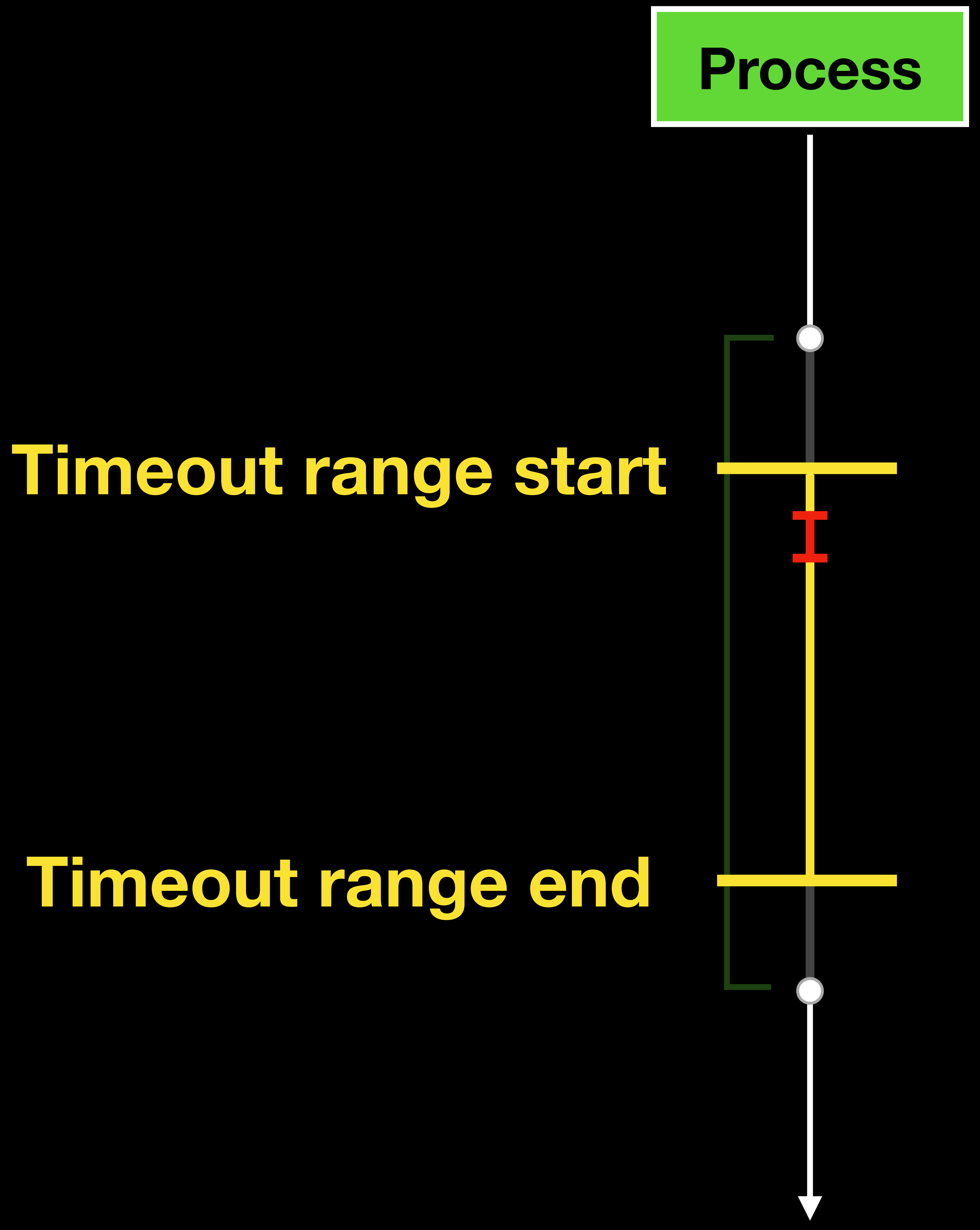
$ Exploitation

- Unpredictable timing gap between timer setup and target execution
  - Require guessing the correct race timing
  - Use a wide timeout range to increase chances
- Can we strategically bound the timeout range?


Process
Call handler
The race timing
Need to guess


Process
The total time


Process
Syscall overhead


Process
Timeout range start
Timeout range end

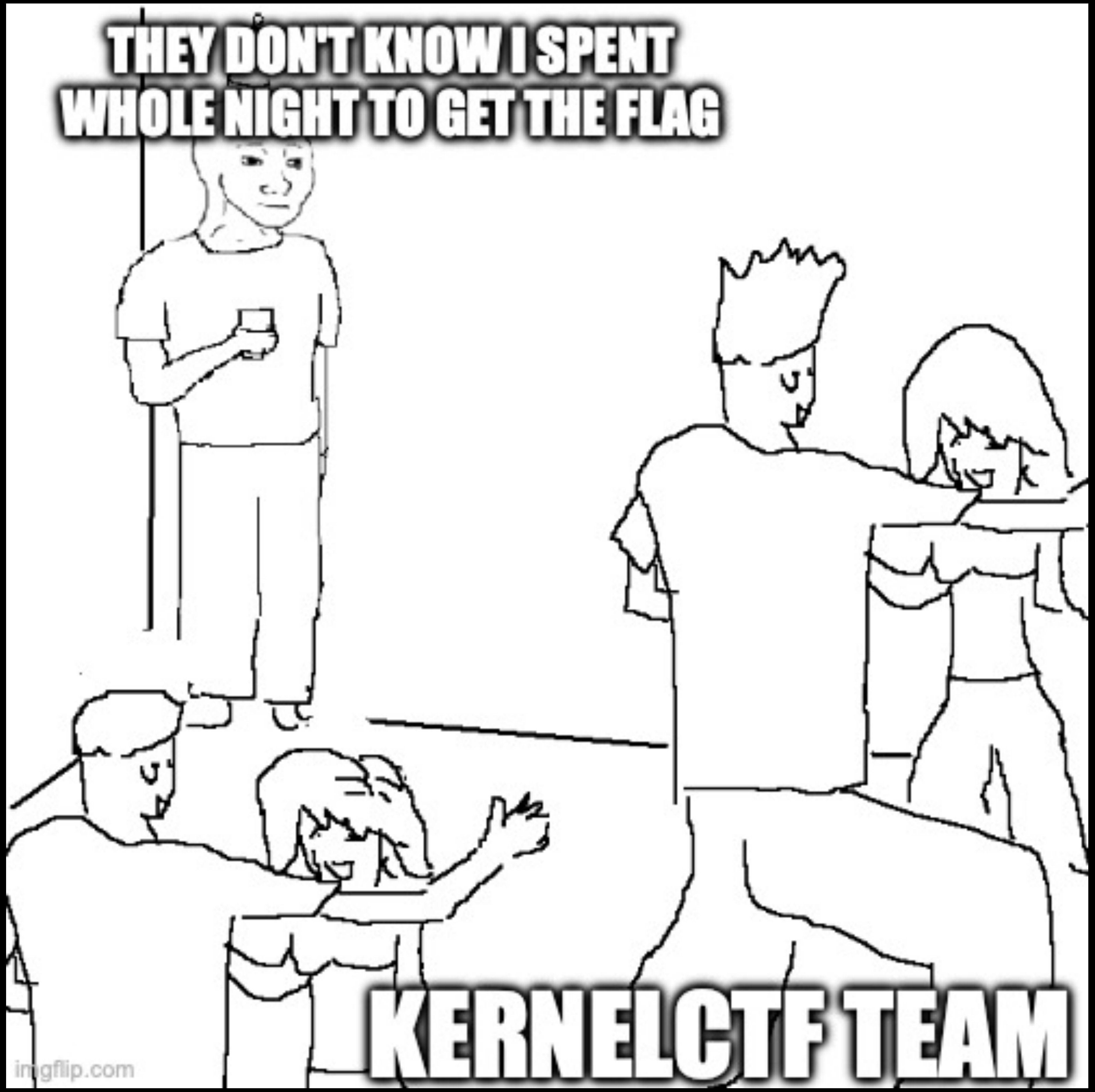
Demo time!

```
aaa@aaa:~/kernelctf/releases/lts-6.6.75/src$
```

```
[0] 0:bash*Z                                                        "aaa" 16:38 22-Sep-25
```

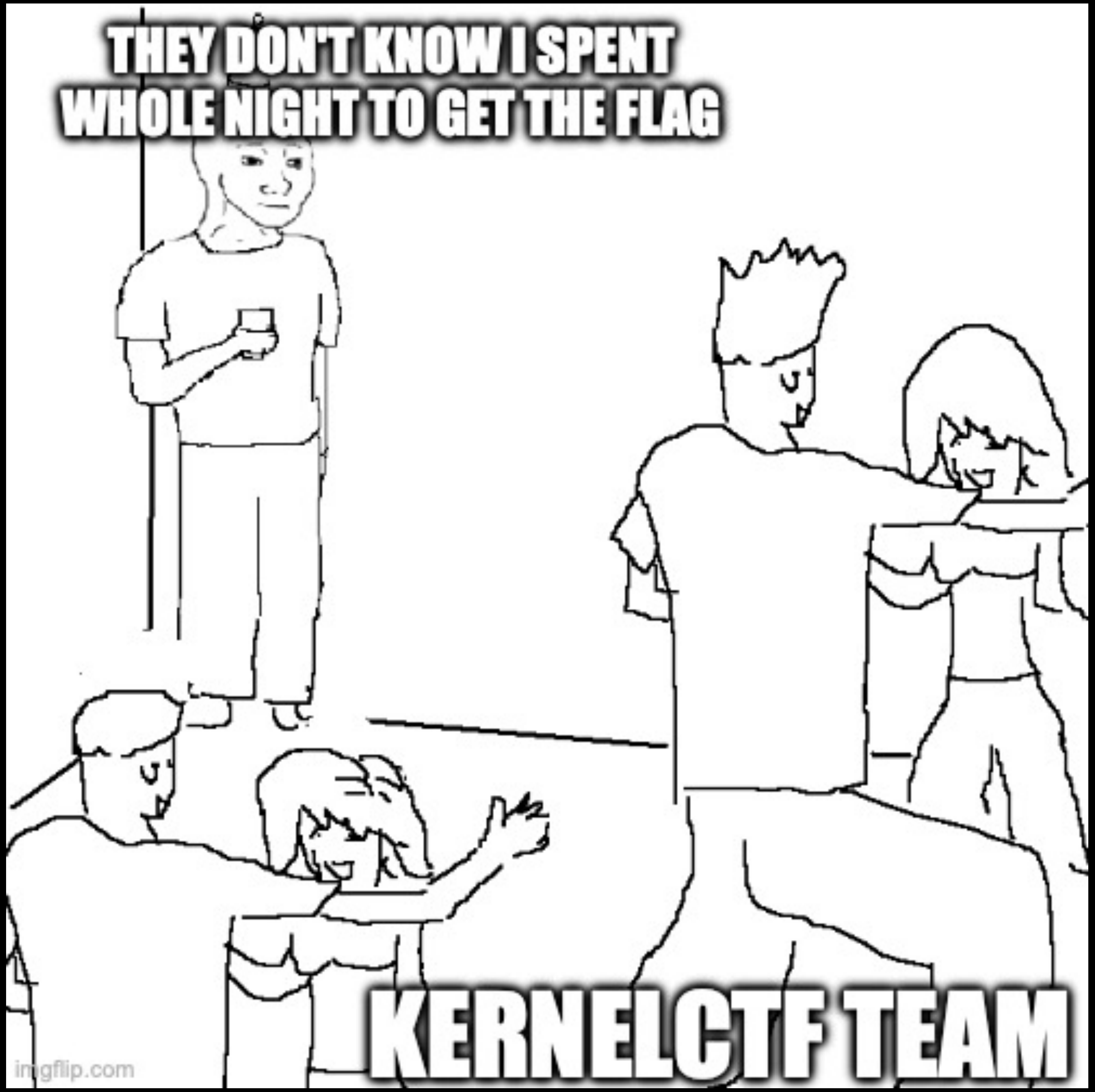
$ Exploitation

- Success rate (per 30 min)
  - kernelCTF: less than 1% 😢



$ Exploitation

- Success rate (per 30 min)
  - kernelCTF: less than 1%
  - GitHub Actions: approximately 30% 🥳



$ Patch

- io_uring/kbuf: reallocate buf lists on upgrade
  - Upgrading now allocates a new I/O buffer instead of reusing the old one

```
@@ -642,12 +648,13 @@ int io_register_pbuf_ring(struct io_ring_ctx *ctx, void __user *arg)
                /* if mapped buffer ring OR classic exists, don't allow */
                if (bl->is_mapped || !list_empty(&bl->buf_list))
                        return -EEXIST;
-       } else {
-               free_bl = bl = kzalloc(sizeof(*bl), GFP_KERNEL);
-               if (!bl)
-                       return -ENOMEM;
+               io_destroy_bl(ctx, bl);
        }

+       free_bl = bl = kzalloc(sizeof(*bl), GFP_KERNEL);
+       if (!bl)
+               return -ENOMEM;
+
```

$ Takeaways

- Memory sharing is common and requires careful handling
    - Complex ownership and lifetime management
    - Concurrency needed for performance increases risk
- Keep reference counts accurate to avoid UAF
- RCU prevents UAF, but not concurrent modifications

DEVCORE

Thanks!

Pumpkin 🎃 (@u1f383)
https://u1f383.github.io/