black hat USA 2024

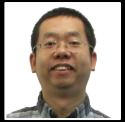
AUGUST 7-8, 2024

BRIEFINGS

The Way to Android Root:

Exploiting Your GPU On Smartphone

Xiling Gong



Xuan Xing

Eugene Rodionov

Whoami

ANDROID RED TEAM

Increase Android and Pixel security by attacking key components and features, identifying critical vulnerabilities before adversaries

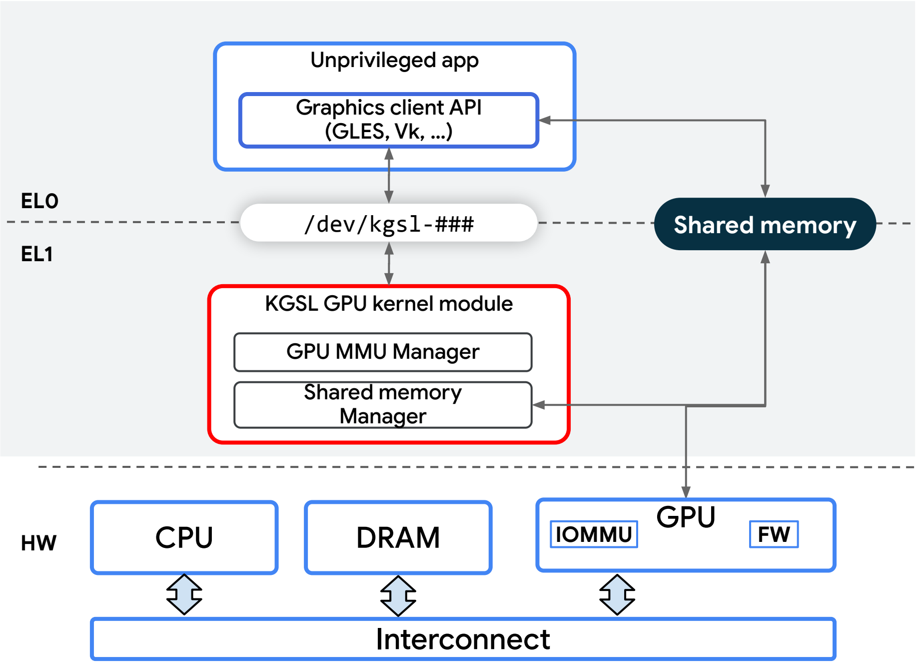
- Offensive Security Reviews to verify (break) security assumptions
- Scale through tool development (e.g. continuous fuzzing)
- Develop proof of concepts to demonstrate real-world impact
- Assess the efficacy of security mitigations

Agenda

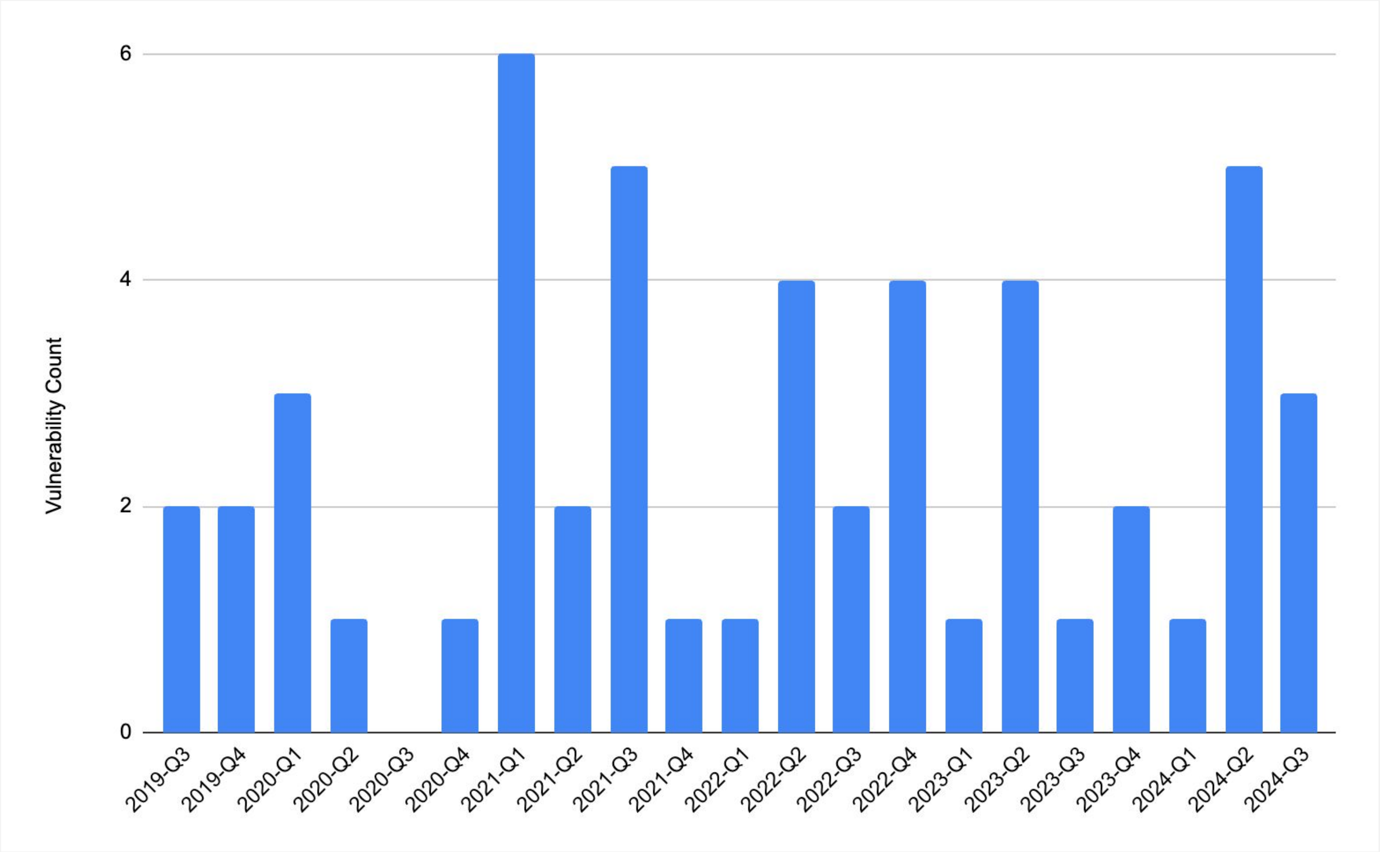
- Background Introduction
- Qualcomm Adreno GPU Introduction
- CVE-2024-23380 and Exploitation
- Vulnerability and Methodology Discussion

Background

- Why Android GPU Driver?
    - No Permission Required
    - Powerful Functions
    - High Complexity
- Why Qualcomm Adreno GPU?
    - Qualcomm is one of the most important smartphone SoC vendors
    - Adreno is the GPU used in most of the Qualcomm SoCs
    - Evolved architecture recently

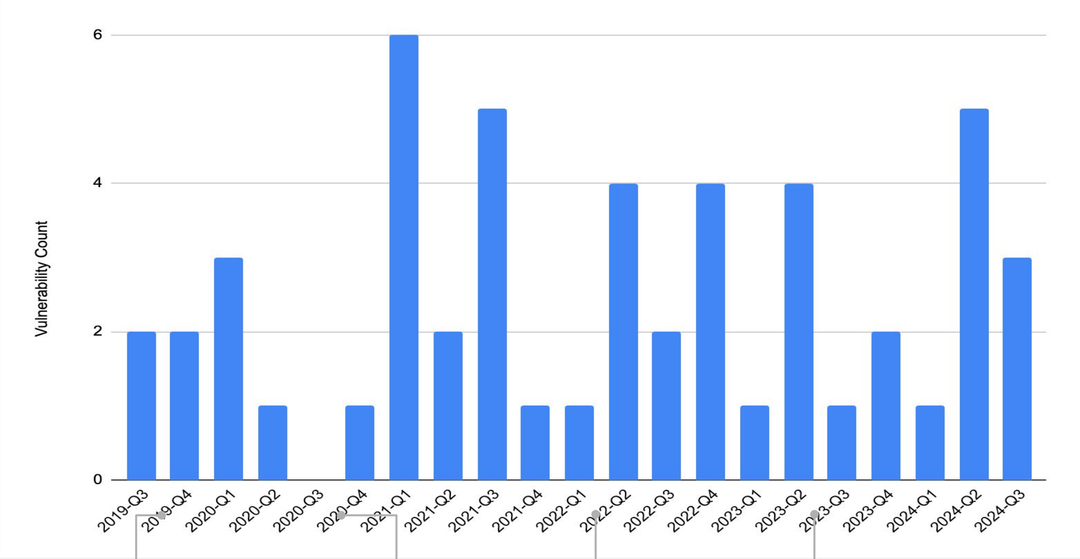


Adreno Driver Issues



Source: https://docs.qualcomm.com/product/publicresources/securitybulletin/

Adreno Driver Issues



2019 TiYunZong Exploit Chain - Gong Guang

2020 Attacking the Qualcomm Adreno GPU - Project Zero

2022 The Android kernel mitigations obstacle race - Man Yue Mo

2023 code in user-writable mapping is executed in non-protected mode - Project Zero

Recent Issues - Qualcomm Security Bulletin

Bulletin	CVE	Rating	Date	Reporter	Tech Area	Exploitability
2024 July	CVE-2024-23380	High	12/13/2023	Xiling Gong	VBO IOMMU - Use After Free	Yes - Easy - Stable
	CVE-2024-23373	High	12/18/2023	Man Yue Mo	IOMMU - Use After Free	Yes - Medium - Stable
	CVE-2024-23372	High	12/12/2023	Fish of Pangu Team	VBO IOMMU - Integer Overflow	Yes - Easy - Stable
2024 June	CVE-2024-21478	High	11/03/2023	Necip Fazil Yildiran	Fence - Type Confusion	Yes - Easy - Stable
2024 May	CVE-2024-21471	High	–	Qualcomm Internal	IOMMU - Use After Free	Yes - Medium - Stable
	CVE-2024-23351	High	–	Qualcomm Internal	Hardware - Improper Access Control	Yes - Medium - Stable
	CVE-2024-23354	High	11/24/2022	ayano23th	VBO - Use After Free	Yes - — - —

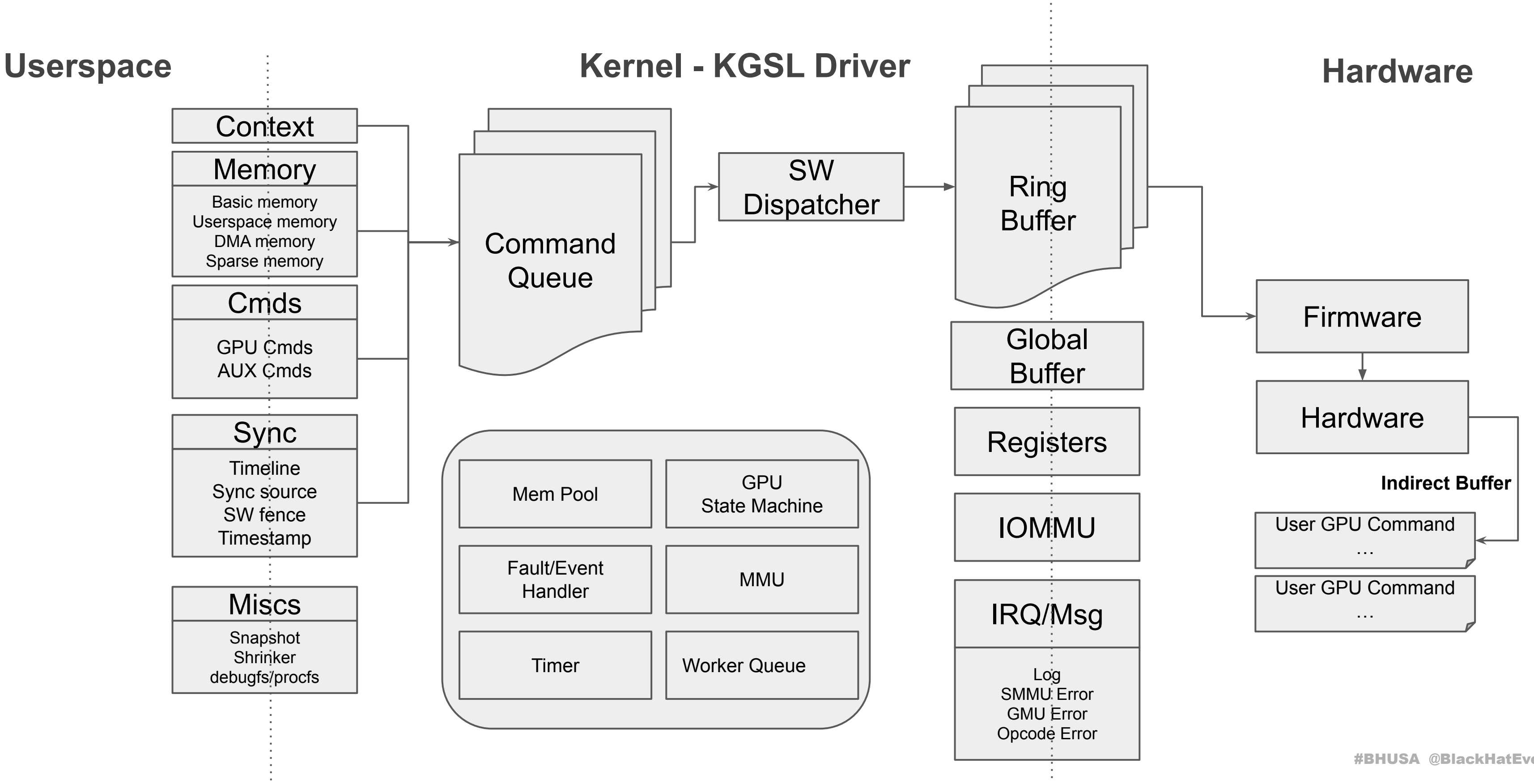
The Issues We Found

- **9+ issues* discovered**
- **We have exploited CVE-2024-23380**
  - Qualcomm notified customers with patch around April 2024
  - Issue published in the Qualcomm Security Bulletin of July 2024
  - We will discuss the issue and exploit

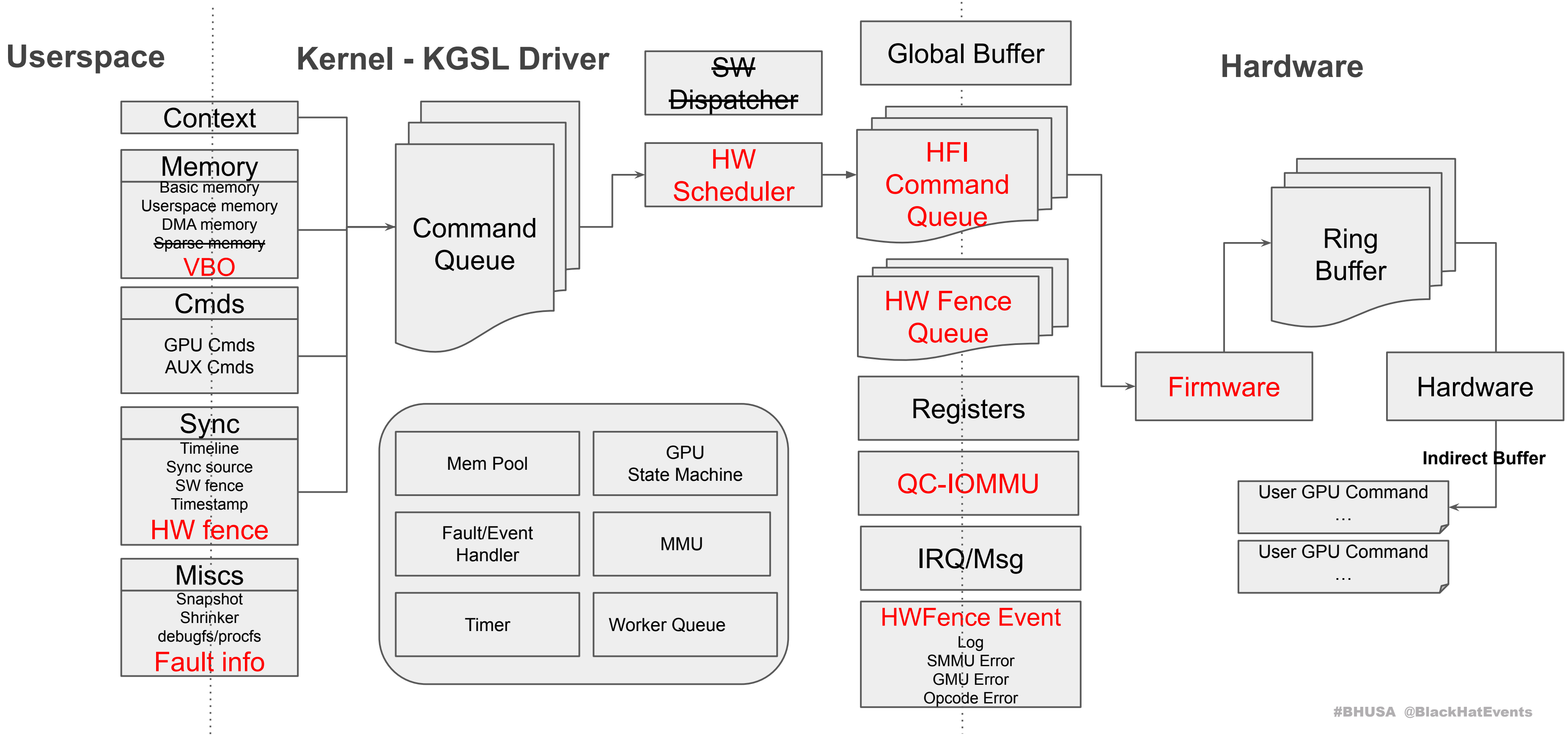
* https://docs.qualcomm.com/product/publicresources/securitybulletin/

Qualcomm Adreno GPU Introduction

KGSL Introduction: Architecture



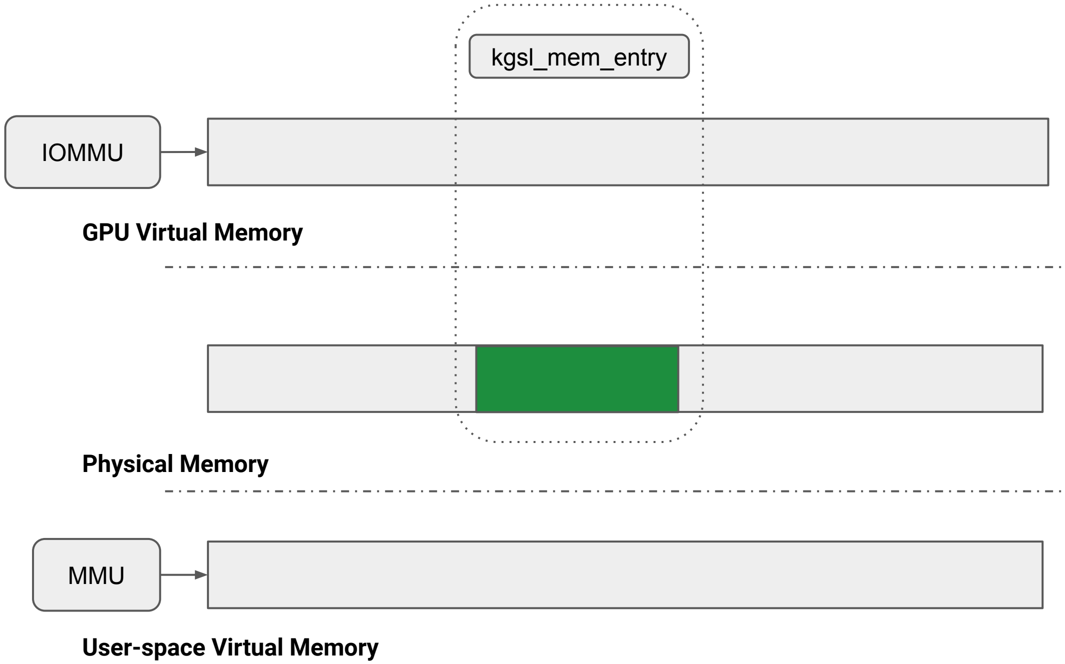
KGSL Introduction: Architecture (Gen7+)



GPU Basic Memory Object Allocation

`IOCTL_KGSL_GPUMEM_ALLOC` ioctl:

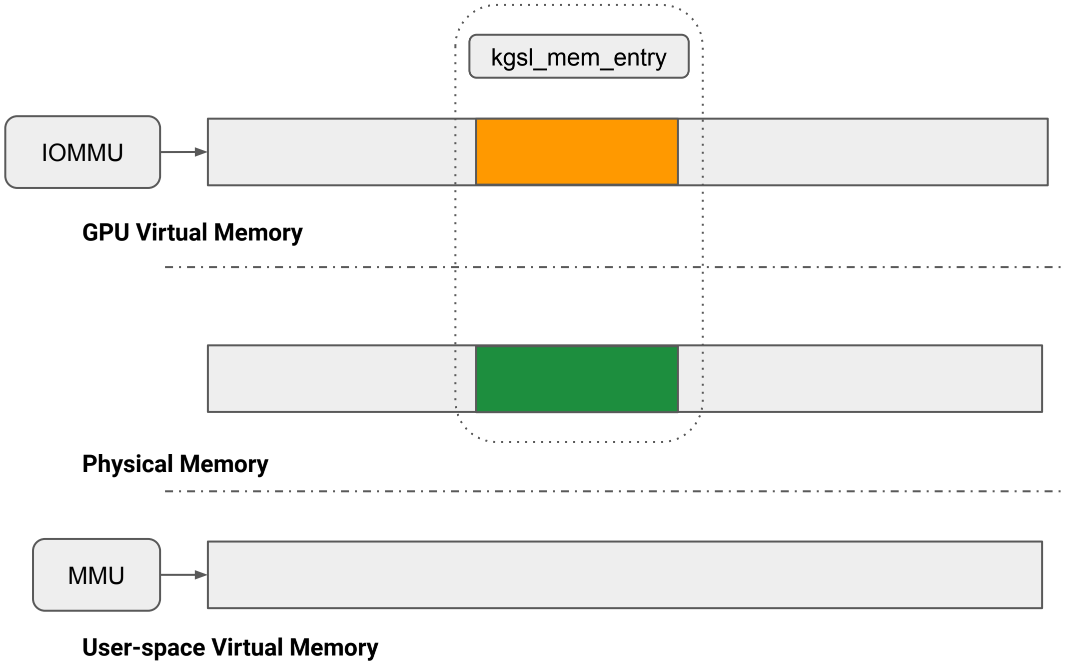
1. **Allocate number of requested physical pages**



GPU Basic Memory Object Allocation

`IOCTL_KGSL_GPUMEM_ALLOC` ioctl:

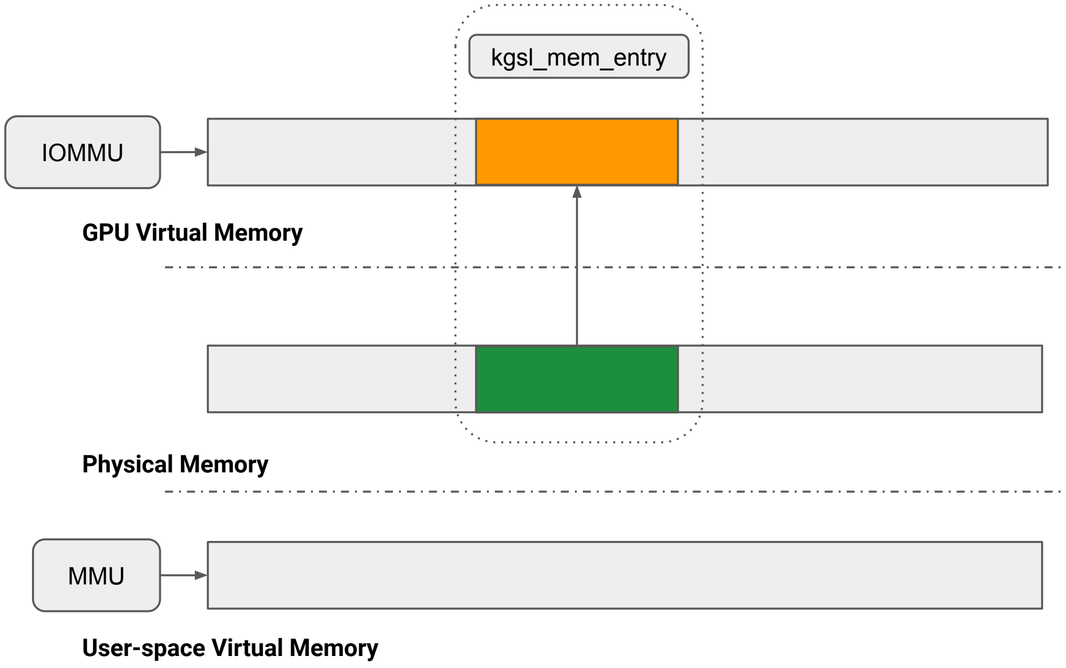
1. Allocate number of requested physical pages
2. **Allocate an address range in GPU virtual address space**



GPU Basic Memory Object Allocation

`IOCTL_KGSL_GPUMEM_ALLOC` ioctl:

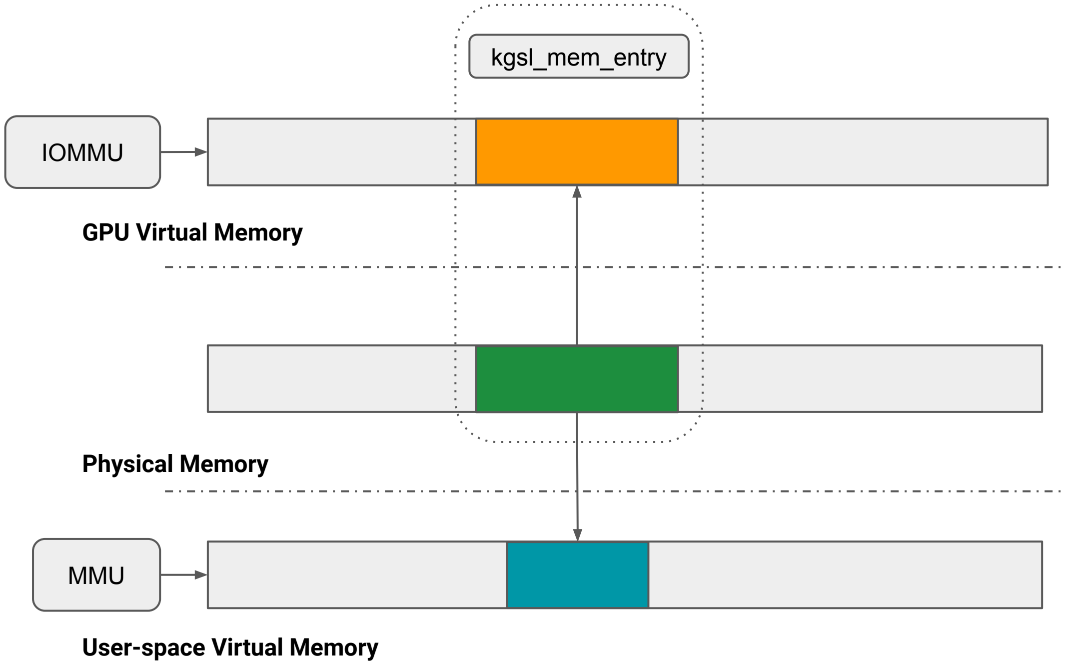
1. Allocate number of requested physical pages
2. Allocate an address range in GPU virtual address space
3. **Map allocated the physical pages to allocated virtual GPU address range**



GPU Basic Memory Object Allocation

`IOCTL_KGSL_GPUMEM_ALLOC` ioctl:

1. Allocate number of requested physical pages
2. Allocate an address range in GPU virtual address space
3. Map allocated the physical pages to allocated virtual GPU address range
4. **Optionally mmap physical pages into user-space virtual memory**

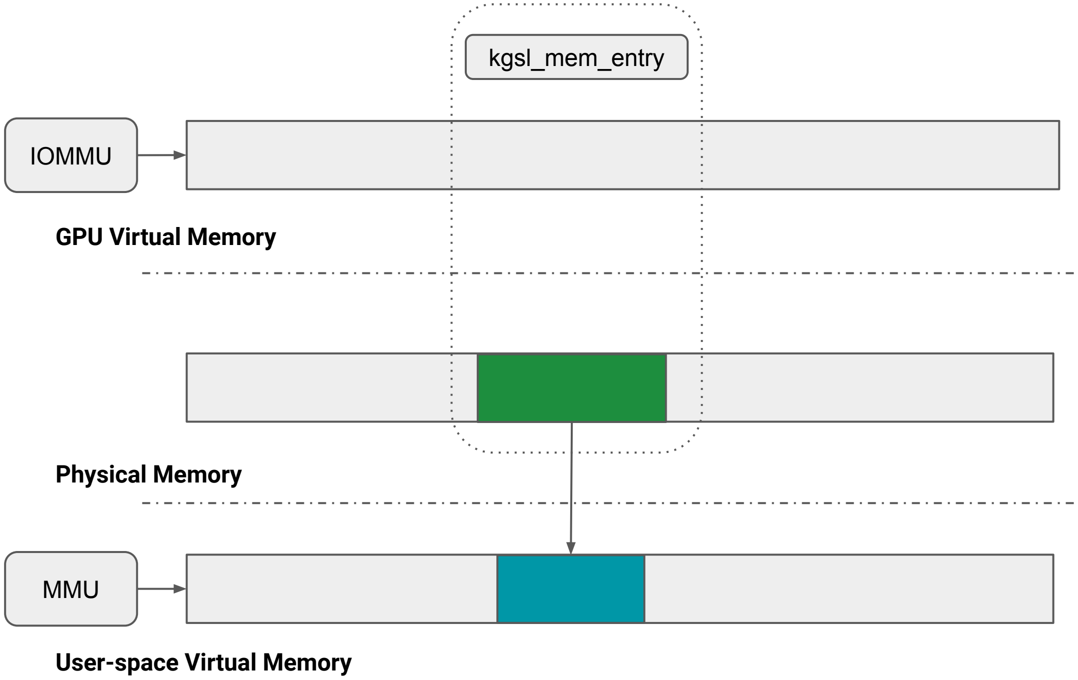


Importing User-space Memory Object

Initial state: allocated user-space memory object

`IOCTL_KGSL_GPUOBJ_IMPORT` ioctl:

1. **Get scatter list of physical pages corresponding to the user-space memory object**

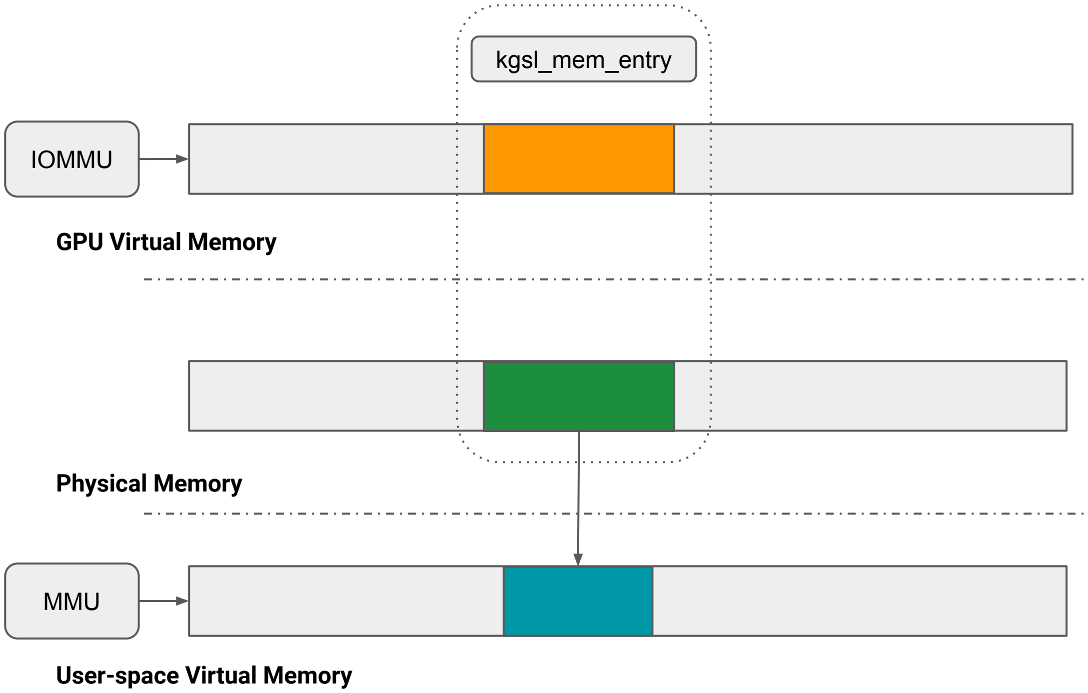


Importing User-space Memory Object

Initial state: allocated user-space memory object

`IOCTL_KGSL_GPUOBJ_IMPORT` ioctl:

1. Get scatter list of physical pages corresponding to the user-space memory object
2. **Allocate address range in GPU virtual address space**

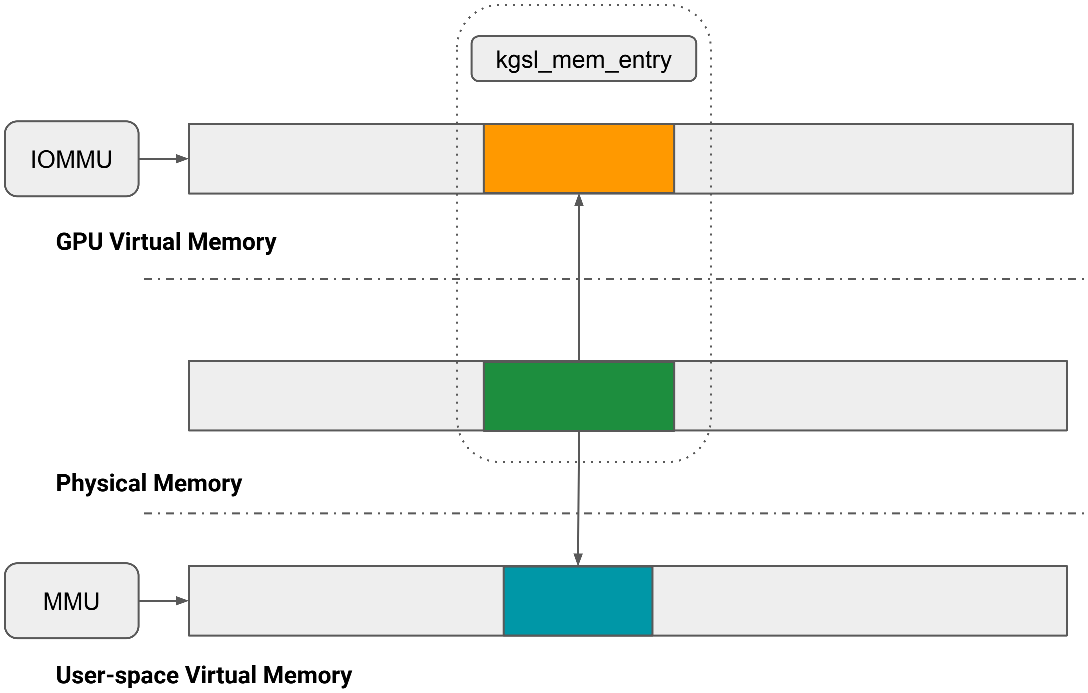


Importing User-space Memory Object

Initial state: allocated user-space memory object

`IOCTL_KGSL_GPUOBJ_IMPORT` ioctl:

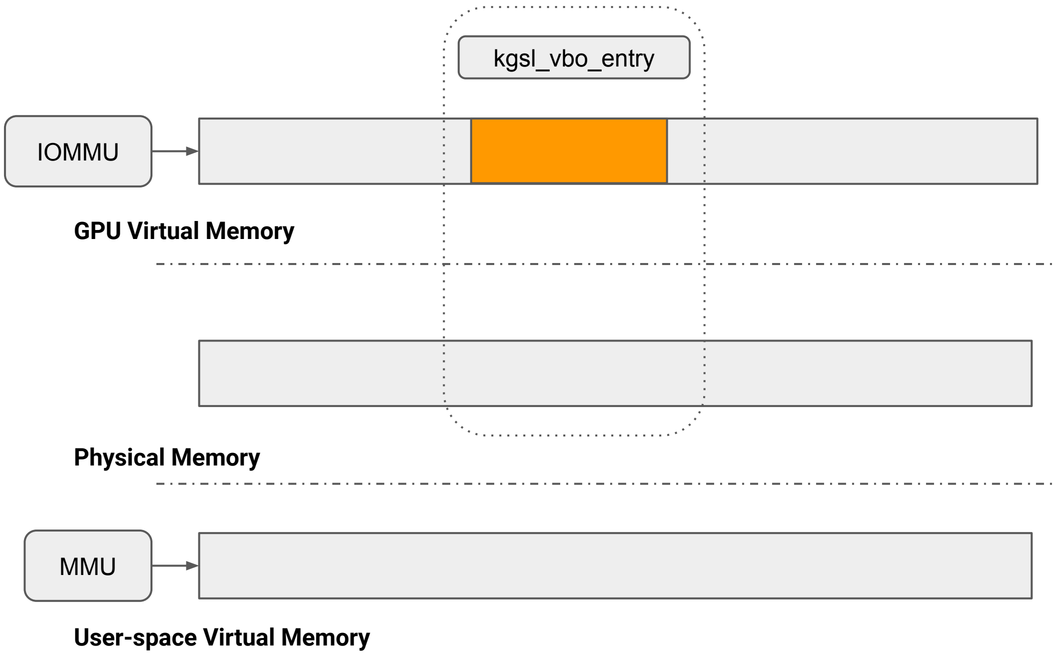
1. Get scatter list of physical pages corresponding to the user-space memory object
2. Allocate address range in GPU virtual address space
3. **Map physical pages in the scatter list to the allocated GPU virtual address range**



Virtual Buffer Object (VBO) Allocation

`IOCTL_KGSL_GPUMEM_ALLOC` ioctl (`flag = KGSL_MEMFLAGS_VBO`):

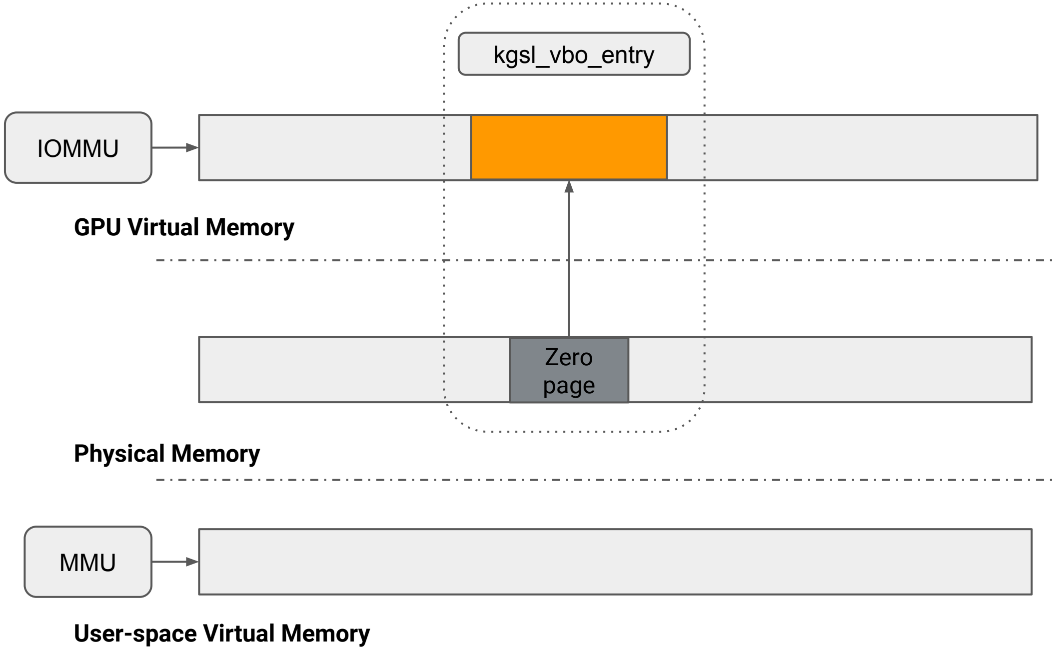
1. **Allocate address range in GPU virtual address space**



Virtual Buffer Object (VBO) Allocation

`IOCTL_KGSL_GPUMEM_ALLOC` ioctl
(`flag = KGSL_MEMFLAGS_VBO`):

1. Allocate address range in GPU virtual address space
2. **Map a placeholder zero page to the allocated GPU virtual address range**

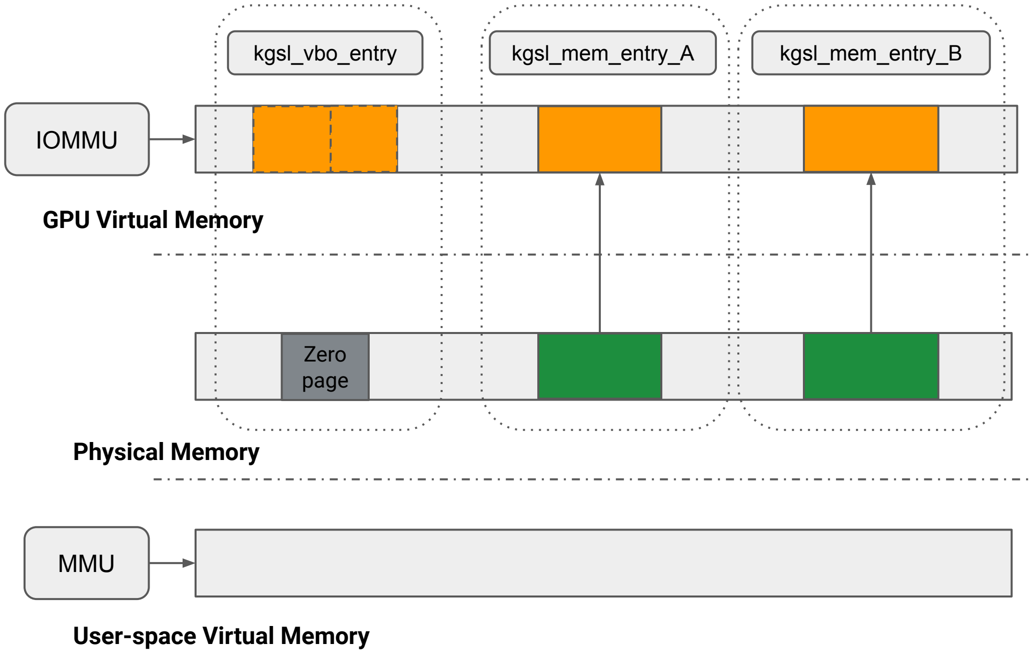


VBO: Binding Basic Memory Object

Initial state: allocated basic memory objects A, B and VBO

`IOCTL_KGSL_GPUMEM_BIND_RANGES` ioctl:

1. **Remove existing mapping from VBO to the zero page**

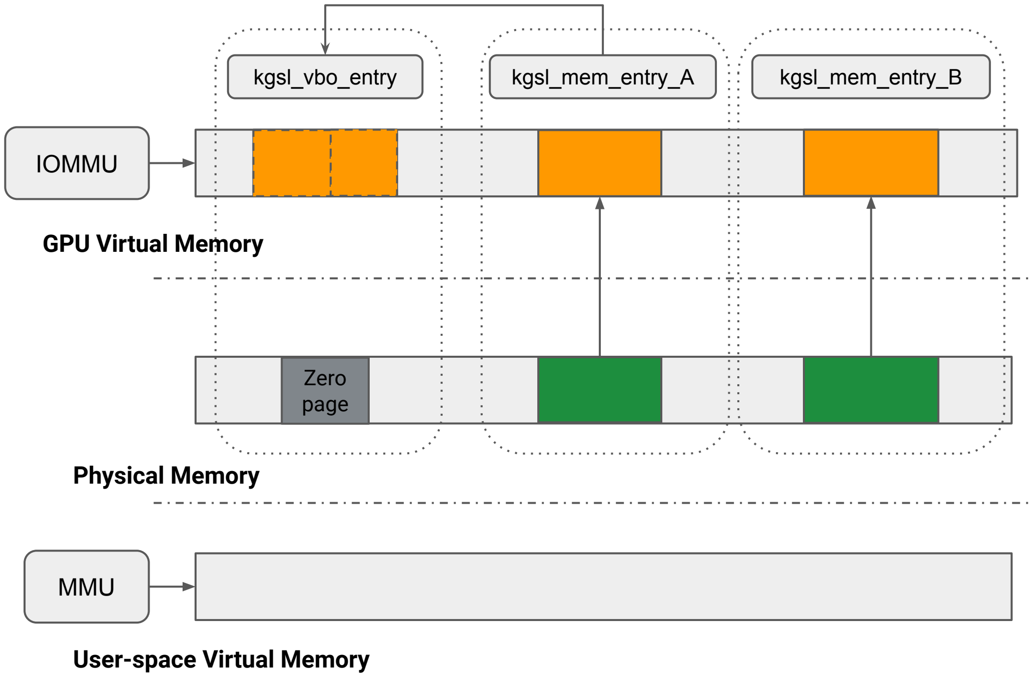


VBO: Binding Basic Memory Object

Initial state: allocated basic memory objects A, B and VBO

`IOCTL_KGSL_GPUMEM_BIND_RANGES` ioctl:

1. Remove existing mapping from VBO to the zero page
2. **Get list of physical pages corresponding to object A**

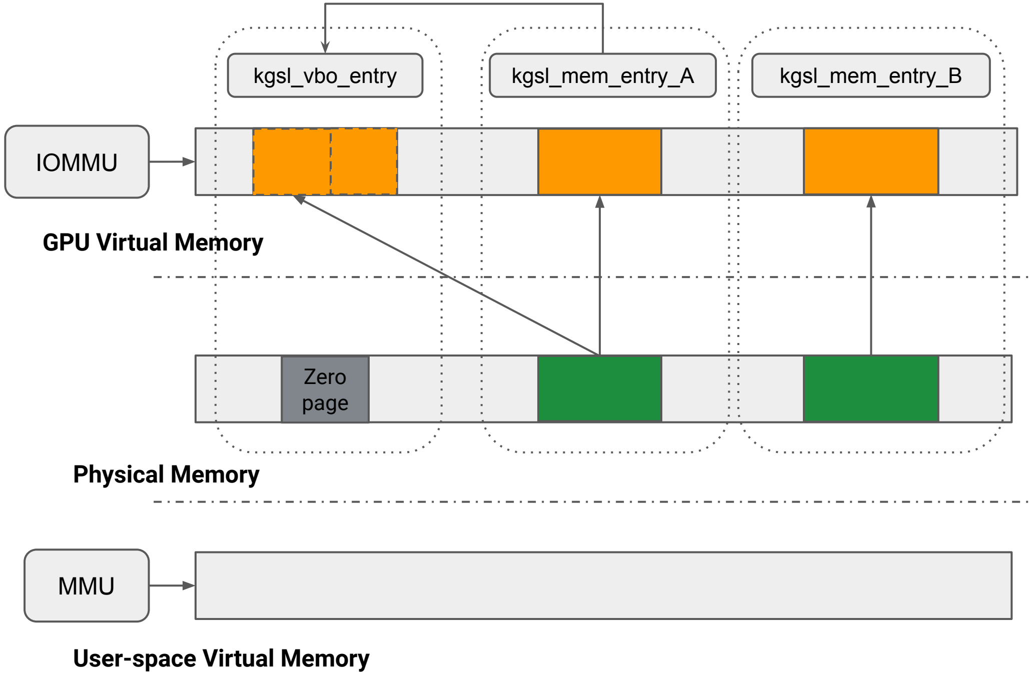


VBO: Binding Basic Memory Object

Initial state: allocated basic memory objects A, B and VBO

`IOCTL_KGSL_GPUMEM_BIND_RANGES` ioctl:

1. Remove existing mapping from VBO to the zero page
2. Get list of physical pages corresponding to object A
3. **Map object's A physical pages into VBO's region at a specified offset**

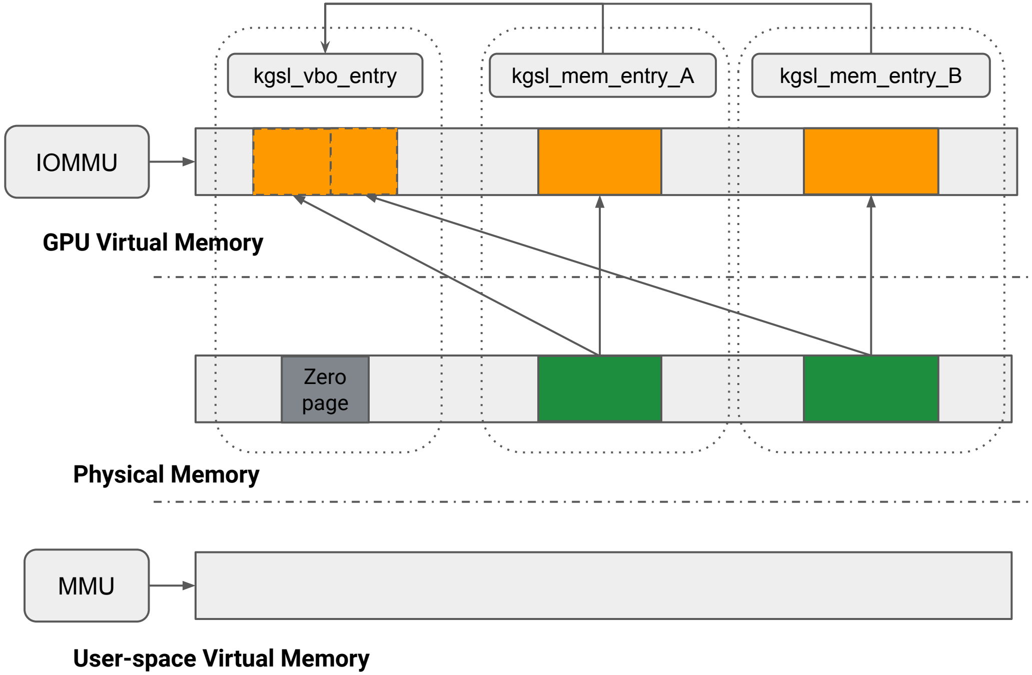


VBO: Binding Basic Memory Object

Initial state: allocated basic memory objects A, B and VBO

`IOCTL_KGSL_GPUMEM_BIND_RANGES` ioctl:

1. Remove existing mapping from VBO to the zero page
2. Get list of physical pages corresponding to object A
3. Map object's A physical pages into VBO's region at a specified offset
4. **Repeat steps 2-3 for all memory objects to bind to the VBO**

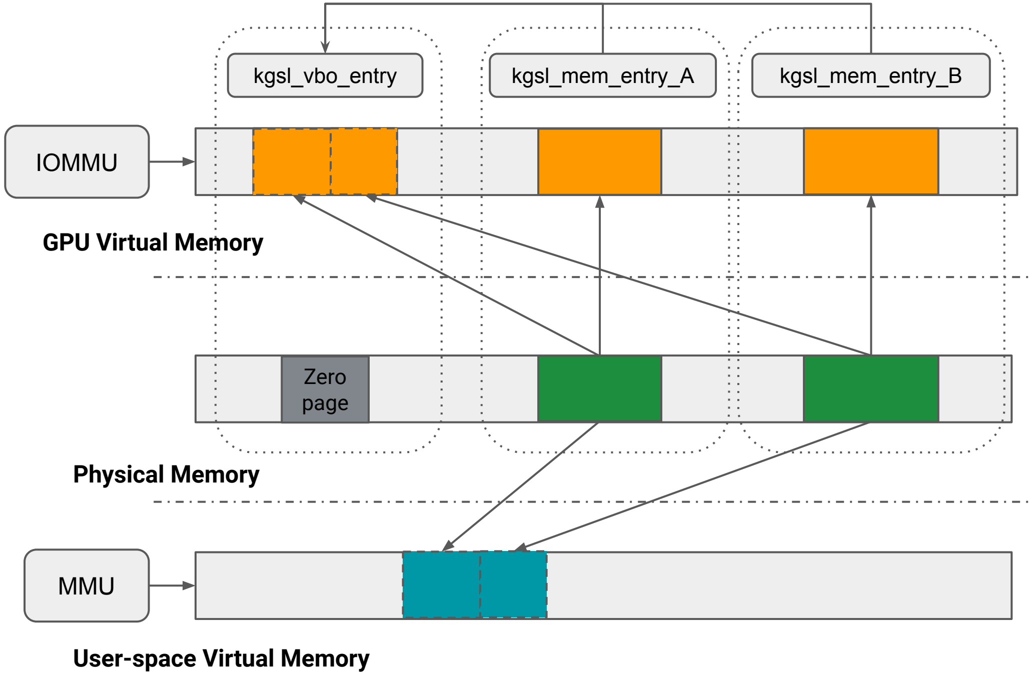


VBO: Binding Basic Memory Object

Initial state: allocated basic memory objects A, B and VBO

`IOCTL_KGSL_GPUMEM_BIND_RANGES` ioctl:

1. Remove existing mapping from VBO to the zero page
2. Get list of physical pages corresponding to object A
3. Map object's A physical pages into VBO's region at a specified offset
4. Repeat steps 2-3 for all memory objects to bind to the VBO
5. **Optionally map physical pages into user-space virtual memory**



CVE-2024-23380 and Exploitation

CVE-2024-23380: Vulnerability Description (1)

- Binding a basic memory object to a VBO:
  - driver inserts the bound basic memory object into the range interval tree for bookkeeping

```
static int kgsl_memdesc_add_range(struct kgsl_mem_entry *target,
            u64 start, u64 last, struct kgsl_mem_entry *entry, u64 offset)
{
    struct kgsl_memdesc *memdesc = &target->memdesc;

    ...
    mutex_lock(&memdesc->ranges_lock);

    /* Add the new range */
    interval_tree_insert(&range->range, &memdesc->ranges);

    trace_kgsl_mem_add_bind_range(target, range->range.start,
        range->entry, bind_range_len(range));
    mutex_unlock(&memdesc->ranges_lock);


    return kgsl_mmu_map_child(memdesc->pagetable, memdesc, start,
        &entry->memdesc, offset, last - start + 1);
}
```

CVE-2024-23380: Vulnerability Description (2)

- Binding a basic memory object to a VBO:
  - driver maps the bound basic memory object into the GPU virtual memory

```
static int kgsl_memdesc_add_range(struct kgsl_mem_entry *target,
            u64 start, u64 last, struct kgsl_mem_entry *entry, u64 offset)
{
    struct kgsl_memdesc *memdesc = &target->memdesc;

    ...
    mutex_lock(&memdesc->ranges_lock);

    /* Add the new range */
    interval_tree_insert(&range->range, &memdesc->ranges);

    trace_kgsl_mem_add_bind_range(target, range->range.start,
        range->entry, bind_range_len(range));
    mutex_unlock(&memdesc->ranges_lock);


    return kgsl_mmu_map_child(memdesc->pagetable, memdesc, start,
        &entry->memdesc, offset, last - start + 1);
}
```

CVE-2024-23380: Vulnerability Description (3)

- Binding a basic memory object to a VBO:
  - the global state change is protected via `memdesc->ranges_lock` mutex

```
static int kgsl_memdesc_add_range(struct kgsl_mem_entry *target,
           u64 start, u64 last, struct kgsl_mem_entry *entry, u64 offset)
{
    struct kgsl_memdesc *memdesc = &target->memdesc;

    ...
    mutex_lock(&memdesc->ranges_lock);

    /* Add the new range */
    interval_tree_insert(&range->range, &memdesc->ranges);

    trace_kgsl_mem_add_bind_range(target, range->range.start,
        range->entry, bind_range_len(range));
    mutex_unlock(&memdesc->ranges_lock);


    return kgsl_mmu_map_child(memdesc->pagetable, memdesc, start,
        &entry->memdesc, offset, last - start + 1);
}
```

CVE-2024-23380: Vulnerability Description (4)

- Binding a basic memory object to a VBO:
  - however, `kgsl_mmu_map_child` isn't protected by the mutex!
  - thus, basic memory object may be unbound by the time `kgsl_mmu_map_child` is invoked

```
static int kgsl_memdesc_add_range(struct kgsl_mem_entry *target,
           u64 start, u64 last, struct kgsl_mem_entry *entry, u64 offset)
{
    struct kgsl_memdesc *memdesc = &target->memdesc;

    ...
    mutex_lock(&memdesc->ranges_lock);

    /* Add the new range */
    interval_tree_insert(&range->range, &memdesc->ranges);

    trace_kgsl_mem_add_bind_range(target, range->range.start,
        range->entry, bind_range_len(range));
    mutex_unlock(&memdesc->ranges_lock);

    // the child's region might not be in the target's interval tree at this point
    return kgsl_mmu_map_child(memdesc->pagetable, memdesc, start,
        &entry->memdesc, offset, last - start + 1);
}
```

CVE-2024-23380: The fix

```diff
static int kgsl_memdesc_add_range(struct kgsl_mem_entry *target,
        u64 start, u64 last, struct kgsl_mem_entry *entry, u64 offset)
{
    struct  interval_tree_node *node, *next;
    struct kgsl_memdesc *memdesc = &target->memdesc;
    struct kgsl_memdesc_bind_range *range =
        bind_range_create(start, last, entry);
    int ret = 0;

    if (IS_ERR(range))
        return PTR_ERR(range);

    mutex_lock(&memdesc->ranges_lock);
        }
    }

    /* Add the new range */
    interval_tree_insert(&range->range, &memdesc->ranges);

    trace_kgsl_mem_add_bind_range(target, range->range.start,
        range->entry, bind_range_len(range));
    mutex_unlock(&memdesc->ranges_lock);

-   return kgsl_mmu_map_child(memdesc->pagetable, memdesc, start,
-           &entry->memdesc, offset, last - start + 1);
```

```diff
static int kgsl_memdesc_add_range(struct kgsl_mem_entry *target,
        u64 start, u64 last, struct kgsl_mem_entry *entry, u64 offset)
{
    struct  interval_tree_node *node, *next;
    struct kgsl_memdesc *memdesc = &target->memdesc;
    struct kgsl_memdesc_bind_range *range =
        bind_range_create(start, last, entry);
    int ret = 0;

    if (IS_ERR(range))
        return PTR_ERR(range);

    mutex_lock(&memdesc->ranges_lock);
        }
    }

+   ret = kgsl_mmu_map_child(memdesc->pagetable, memdesc, start,
+           &entry->memdesc, offset, last - start + 1);
+   if (ret)
+       goto error;
+
    /* Add the new range */
    interval_tree_insert(&range->range, &memdesc->ranges);

    trace_kgsl_mem_add_bind_range(target, range->range.start,
        range->entry, bind_range_len(range));
    mutex_unlock(&memdesc->ranges_lock);

+   return ret;
```

CVE-2024-23380: Triggering the vulnerability (1)

- GPU IOMMU misconfiguration can be achieved by invoking kgsl_memdesc_add_range and kgsl_memdesc_remove_range concurrently in two threads.

```c
static int kgsl_memdesc_add_range(struct kgsl_mem_entry *target,
        u64 start, u64 last, struct kgsl_mem_entry *entry, u64 offset)
{
...
    mutex_lock(&memdesc->ranges_lock);
...
    /* Add the new range */
    interval_tree_insert(&range->range, &memdesc->ranges);
...
    mutex_unlock(&memdesc->ranges_lock);

    return kgsl_mmu_map_child(memdesc->pagetable, memdesc, start,
            &entry->memdesc, offset, last - start + 1);
```

```c
static void kgsl_memdesc_remove_range(struct kgsl_mem_entry *target,
        u64 start, u64 last, struct kgsl_mem_entry *entry)
{
...
    mutex_lock(&memdesc->ranges_lock);

    next = interval_tree_iter_first(&memdesc->ranges, start, last);
    while (next) {
        node = next;
        range = bind_to_range(node);
...
        if (!entry || range->entry->id == entry->id) {
            if (kgsl_mmu_unmap_range(memdesc->pagetable,
                memdesc, range->range.start, bind_range_len(range)))
                continue;

            interval_tree_remove(node, &memdesc->ranges);
...
            kgsl_mem_entry_put(range->entry);
            kfree(range);
        }
    }

    mutex_unlock(&memdesc->ranges_lock);
}
```

CVE-2024-23380: Triggering the vulnerability (2)

Thread A

```
static int kgsl_memdesc_add_range(struct kgsl_mem_entry *target,
        u64 start, u64 last, struct kgsl_mem_entry *entry, u64 offset)
{
...
    mutex_lock(&memdesc->ranges_lock);
...

    /* Add the new range */
    interval_tree_insert(&range->range, &memdesc->ranges);
...
    mutex_unlock(&memdesc->ranges_lock);
```

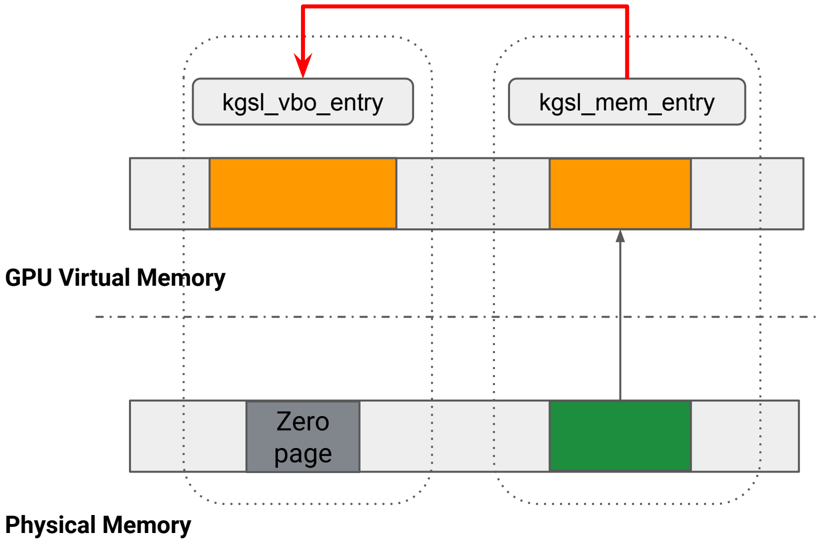
```
    return kgsl_mmu_map_child(memdesc->pagetable, memdesc, start,
            &entry->memdesc, offset, last - start + 1);
```

Thread B

```
static void kgsl_memdesc_remove_range(struct kgsl_mem_entry *target,
        u64 start, u64 last, struct kgsl_mem_entry *entry)
{
...
    mutex_lock(&memdesc->ranges_lock);

    next = interval_tree_iter_first(&memdesc->ranges, start, last);
    while (next) {
        node = next;
        range = bind_to_range(node);
...
        if (!entry || range->entry->id == entry->id) {
            if (kgsl_mmu_unmap_range(memdesc->pagetable,
                memdesc, range->range.start, bind_range_len(range)))
                continue;

            interval_tree_remove(node, &memdesc->ranges);
...
            kgsl_mem_entry_put(range->entry);
            kfree(range);
        }
    }

    mutex_unlock(&memdesc->ranges_lock);
}
```

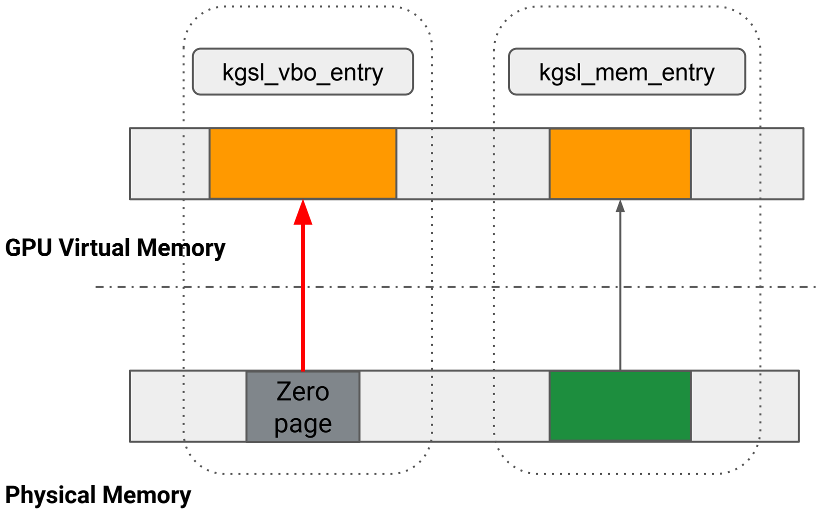
CVE-2024-23380: Triggering the vulnerability (3)

Thread A	Thread B
1. Acquire mutex **2. Add victim basic memory object to the target VBO interval tree** **3. Release the mutex**	
	1. Acquire the mutex 2. Remove the victim basic memory object from the target VBO interval tree 3. Release the mutex
4. Map the victim's physical pages to the target VBO address range	
5. Delete the victim basic memory object and release its physical pages back to kernel	



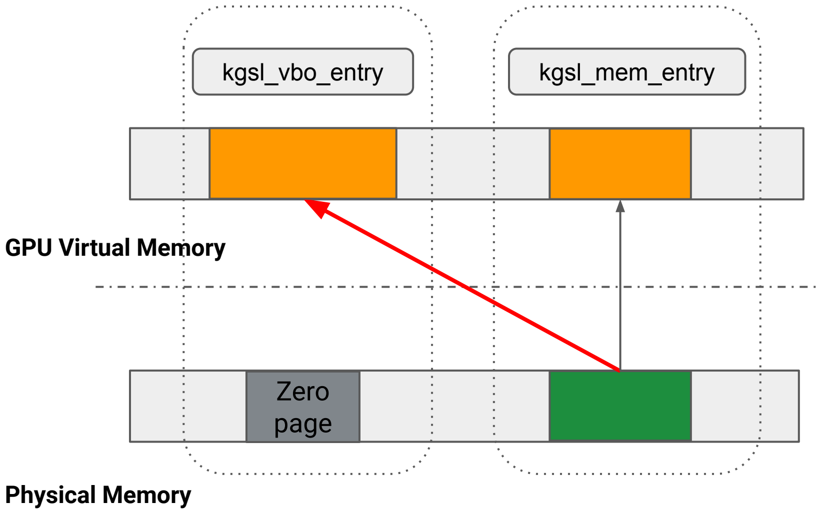
CVE-2024-23380: Triggering the vulnerability (4)

Thread A	Thread B
1. Acquire mutex 2. Add victim basic memory object to the target VBO interval tree 3. Release the mutex	
	1. Acquire the mutex **2. Remove the victim basic memory object from the target VBO interval tree** **3. Release the mutex**
4. Map the victim's physical pages to the target VBO address range	
5. Delete the victim basic memory object and release its physical pages back to kernel	



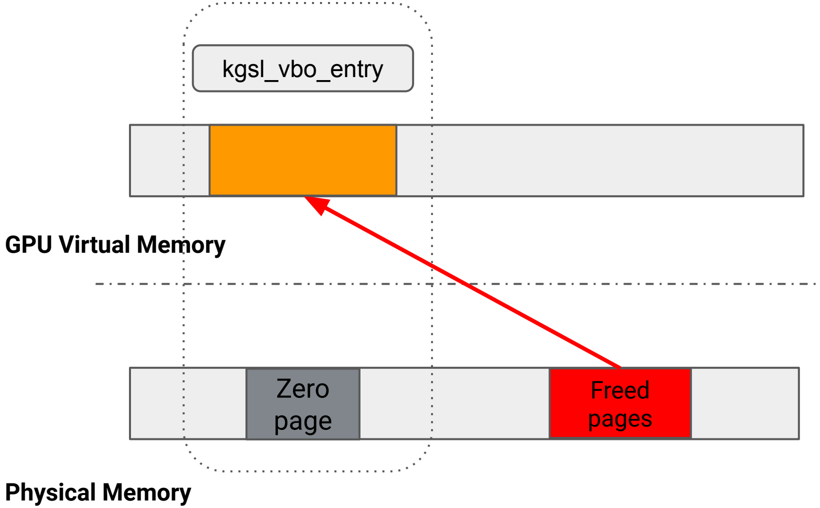
CVE-2024-23380: Triggering the vulnerability (5)

Thread A	Thread B
1. Acquire mutex 2. Add victim basic memory object to the target VBO interval tree 3. Release the mutex	
	1. Acquire the mutex 2. Remove the victim basic memory object from the target VBO interval tree 3. Release the mutex
4. Map the victim's physical pages to the target VBO address range	
5. Delete the victim basic memory object and release its physical pages back to kernel	



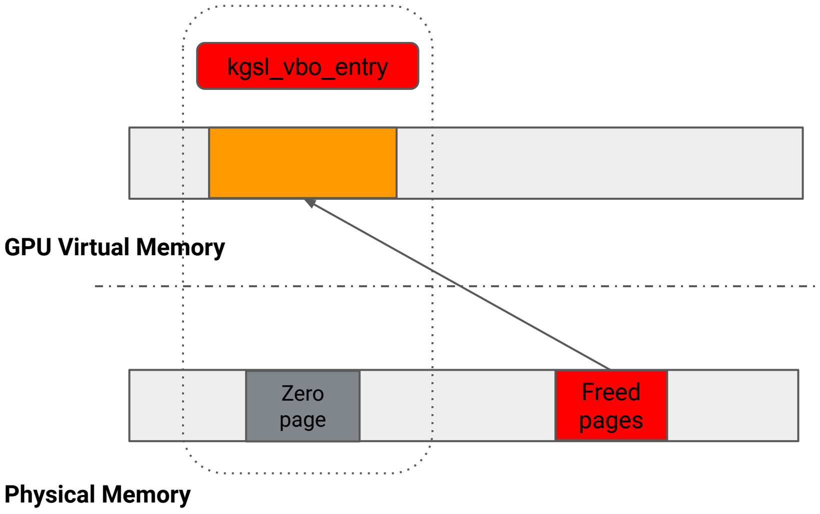
CVE-2024-23380: Triggering the vulnerability (6)

Thread A	Thread B
1. Acquire mutex 2. Add victim basic memory object to the target VBO interval tree 3. Release the mutex	
	1. Acquire the mutex 2. Remove the victim basic memory object from the target VBO interval tree 3. Release the mutex
4. Map the victim's physical pages to the target VBO address range	
5. Delete the victim basic memory object and release its physical pages back to kernel	



CVE-2024-23380: Exploitation primitive (1)

- **Leads to physical memory pages use-after-free**
  - read/write access to the freed kernel physical pages from the GPU context
- **Race-condition bug**
  - need multiple threads doing binding and unbinding to the target VBO concurrently



CVE-2024-23380: Exploitation primitive (2)

- **Physical page use-after-free**
  - We retain this physical-virtual mapping. So we don't have to worry about the kASLR while manipulating the physical page with a known virtual GPU address
- **Stable**
  - When we run the race-condition to trigger the issue, it's no harm if the issue fails to trigger - The kernel status is good and will not crash
- **Feedback on issues triggered or not**
  - We have a trick a get the status of whether the issues is triggered or not - by putting a special Sentinel into the memory - If the issue is triggered, the Sentinel will remain in the memory.
- **Easy to trigger**
  - The two threads hold the same mutex. When one of the threads releases the mutex, it's very likely that the other thread will be scheduled. The success rate of triggering the issue is very high without any special techniques (more list entries, priority adjustments, etc.)

CVE-2024-23380 Exploit

- To exploit the vulnerability we need to answer the following question:
  - Suppose we control a large amount of physical memory pages, what are we going to do?
- There are lots of answers and solutions for an experienced kernel hacker :)
- For example:
  - Spray many struct cred into the kernel heap by creating lots of userspace processes
  - GPU page table modification via “Kernel Space Memory Mirroring” attack[1]

[1] https://i.blackhat.com/briefings/asia/2018/asia-18-WANG-KSMA-Breaking-Android-kernel-isolation-and-Rooting-with-ARM-MMU-features.pdf

CVE-2024-23380 Exploit: Our method

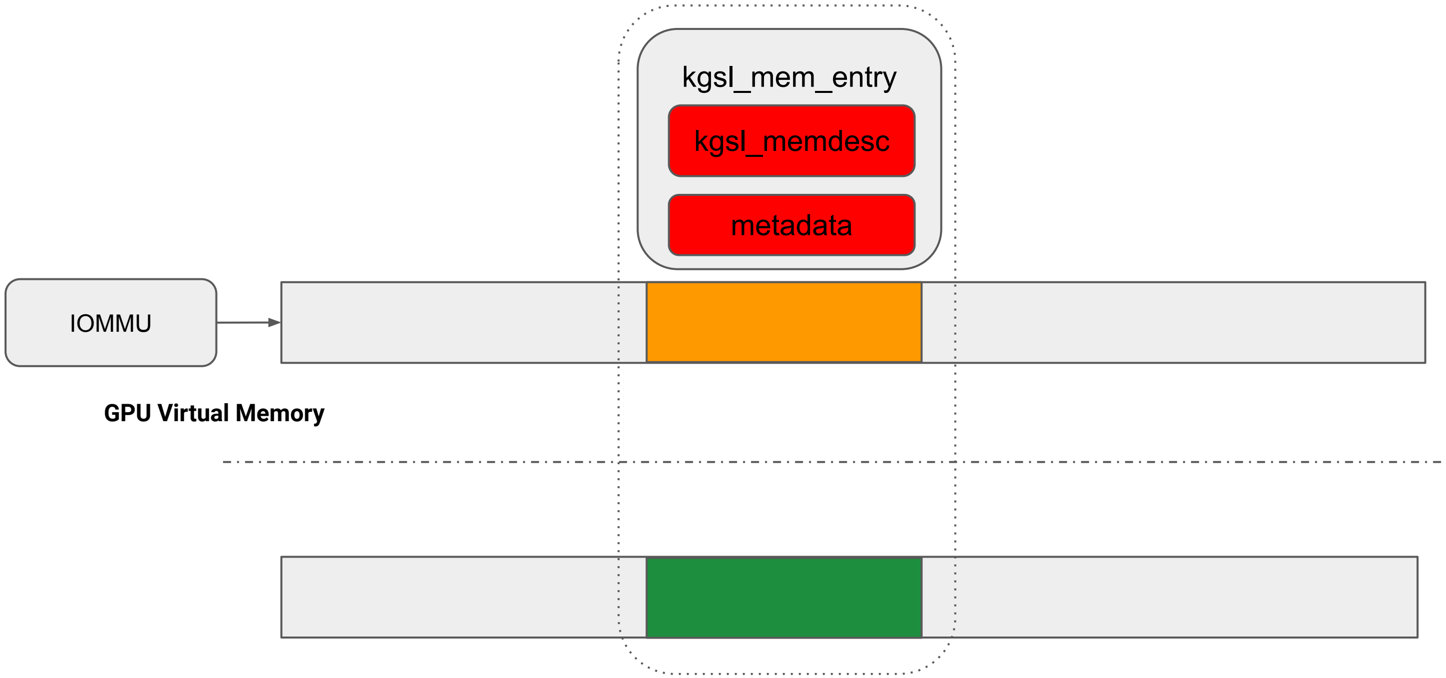
Overwrite contents of `struct kgsl_memdesc`

- **Easy to spray**
  - simply call IOCTL_KGSL_GPUOBJ_ALLOC
- **Easy to be found in the memory**
  - by set a unique metadata from userspace
- **Easy to use**
  - could be used to map the whole kernel physical memory to userspace
- **Easy to check success**
  - do mmap on the modified object
- **Bonus**
  - Possible kASLR bypass

```c
struct kgsl_mem_entry {
    struct kref refcount;
    struct kgsl_memdesc memdesc;
    void *priv_data;
    struct rb_node node;
    unsigned int id;
    struct kgsl_process_private *priv;
    int pending_free;
    char metadata[KGSL_GPUOBJ_ALLOC_METADATA_MAX + 1];
    struct work_struct work;
```

```c
struct kgsl_memdesc {
    struct kgsl_pagetable *pagetable;
    void *hostptr;
    unsigned int hostptr_count;
    uint64_t gpuaddr;
    phys_addr_t physaddr;
    uint64_t size;
    unsigned int priv;
    struct sg_table *sgt;
    const struct kgsl_memdesc_ops *ops;
```

CVE-2024-23380 Exploit: struct kgsl_memdesc (1)



CVE-2024-23380 Exploit: struct kgsl_memdesc (2)

```
struct kgsl_memdesc {
    struct kgsl_pagetable *pagetable;
    void *hostptr;
    unsigned int hostptr_count;
    uint64_t gpuaddr;
    phys_addr_t physaddr;
    uint64_t size;
    unsigned int priv;
    struct sg_table *sgt;
    const struct kgsl_memdesc_ops *ops;
```

```
static const struct kgsl_memdesc_ops kgsl_page_ops = {
    .free = kgsl_free_pages,
    .vmflags = VM_DONTDUMP | VM_DONTEXPAND | VM_DONTCOPY
    .vmfault = kgsl_paged_vmfault,
    .map_kernel = kgsl_paged_map_kernel,
    .unmap_kernel = kgsl_paged_unmap_kernel,
    .put_gpuaddr = kgsl_unmap_and_put_gpuaddr,
};
```

```
static const struct kgsl_memdesc_ops kgsl_contiguous_ops = {
    .free = kgsl_contiguous_free,
    .vmflags = VM_DONTDUMP | VM_PFNMAP | VM_DONTEXPAND | VM_D
    .vmfault = kgsl_contiguous_vmfault,
    .put_gpuaddr = kgsl_unmap_and_put_gpuaddr,
};
```

```
static vm_fault_t kgsl_contiguous_vmfault(struct kgsl_memdesc *memdesc,
                struct vm_area_struct *vma,
                struct vm_fault *vmf)
{
    unsigned long offset, pfn;

    offset = ((unsigned long) vmf->address - vma->vm_start) >>
        PAGE_SHIFT;

    pfn = (memdesc->physaddr >> PAGE_SHIFT) + offset;
    return vmf_insert_pfn(vma, vmf->address, pfn);
}
```

1. Do mmap on this kgsl_memdesc
2. When userspace probes the mmap'ed memory, it will trigger vmfault and setup the mapping for memdesc->physaddr

CVE-2024-23380: Exploitation

Trigger the vulnerability

x200 times

↓

Get a controlled physical page

↓ Now GPU controls 200 physical pages

Force low memory to return all cached memory to system

↓ But these 200 pages still will be mapped to GPU

Kernel heap spray with kgsl_memdesc objects

↓ Some entries will be located in the controlled page

Scan controlled memory pages for sprayed kgsl_memdesc objects

↓ kgsl_memdesc->ops contains the kASLR information

Modify kgsl_memdesc's mmap ops, phys addr, size via GPU

↓ Phys addr = Kernel phys addr & size = kernel size

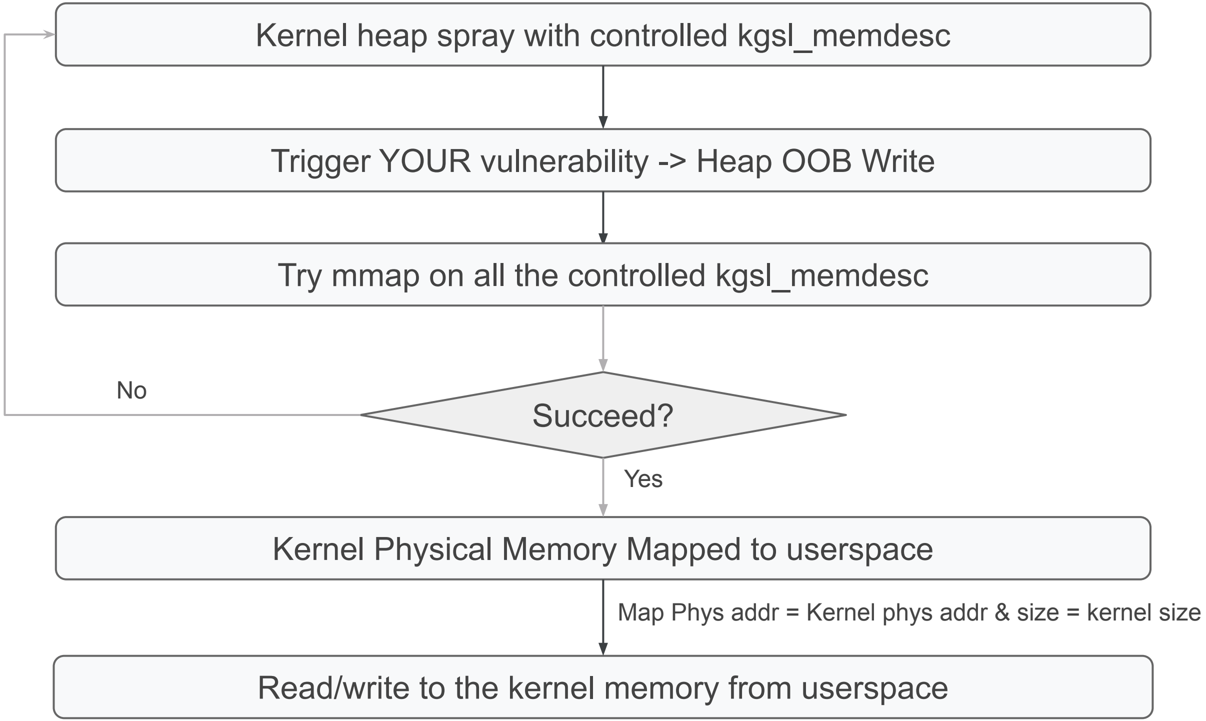
Userspace mmap controlled kgsl_memdesc

↓

Read/write to the kernel memory from userspace

```
struct kgsl_memdesc {
    struct kgsl_pagetable *pagetable;
    void *hostptr;
    unsigned int hostptr_count;
    uint64_t gpuaddr;
    phys_addr_t physaddr;
    uint64_t size;
    unsigned int priv;
    struct sg_table *sgt;
    const struct kgsl_memdesc_ops *ops;
```

Exploitation: Expanding to Generic Cases



Heap

Overflow

```
struct kgsl_memdesc {
    struct kgsl_pagetable *pagetable;
    void *hostptr;
    unsigned int hostptr_count;
    uint64_t gpuaddr;
    phys_addr_t physaddr;
    uint64_t size;
    unsigned int priv;
    struct sg_table *sgt;
    const struct kgsl_memdesc_ops *ops;
```

Exploitation: bypassing kASLR

Method A：Direct kALSR bypass

```
:/ # cat /proc/kallsyms | grep "ffffffdd39066f"
ffffffdd39066f60 r kgsl_contiguous_ops  [msm_kgsl]
ffffffdd39066fc0 r kgsl_secure_page_ops [msm_kgsl]
ffffffdd39066f90 r kgsl_secure_system_ops       [msm_kgsl]
ffffffdd39066ff0 r kgsl_system_ops      [msm_kgsl]
```

Ops=kgsl_secure_page_ops　0C F6 60 93 DD FF FF FF

→kgsl_contiguous_ops　06 F6 60 93 DD FF FF FF

Known　The Same

Method B：Brute Force 16 trials

```
ffffffdd39066f60 r kgsl_contiguous_ops   [msm_kgsl]
ffffffdd39067020 r kgsl_page_ops         [msm_kgsl]
```

Ops=kgsl_page_ops　02 07 60 93 DD FF FF FF

→kgsl_contiguous_ops　06 F6 60 93 DD FF FF FF

Guess these 4 bits = 16 trials

Heap

Overflow

```
struct kgsl_memdesc {
    struct kgsl_pagetable *pagetable;
    void *hostptr;
    unsigned int hostptr_count;
    uint64_t gpuaddr;
    phys_addr_t physaddr;
    uint64_t size;
    unsigned int priv;
    struct sg_table *sgt;
    const struct kgsl_memdesc_ops *ops;
```

Not used

Demo

https://youtu.be/uUJyK8DcPcs

Demo

```
msm_kgsl>
```

Methodology & Bug Discovery

Discovery Methodology: Patch Analysis

Patch Analysis is important for complicated targets like GPU:

- To further understand the issue
- To confirm whether the patch really solves the problem
- To discover new similar issues

Discovery Methodology: Fuzzing

- **The Difficulties of GPU Fuzzing**
  - Hardware dependencies
  - Detecting misconfiguration with MMU/IOMMU
  - Complicated statemachine
  - Concurrency issues
- **Possible solutions**
  - Emulation based fuzzing
  - Deterministic race condition detection

Ultimate Mitigations?

- **Sandboxing GPU interface?**
  - Out of process HAL to GPU driver
  - Performance impact?
  - Backward compatibility issues?

- **Memory safe (Rust) implementation?**
  - Race conditions
  - Integer overflows

ANDROID RED TEAM

Thank You! Questions?

https://androidoffsec.withgoogle.com

androidoffsec-external@google.com