



AUGUST 7-8, 2024
BRIEFINGS

The Way to Android Root:

Exploiting Your GPU On Smartphone



Xiling Gong



Xuan Xing



Eugene Rodionov

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

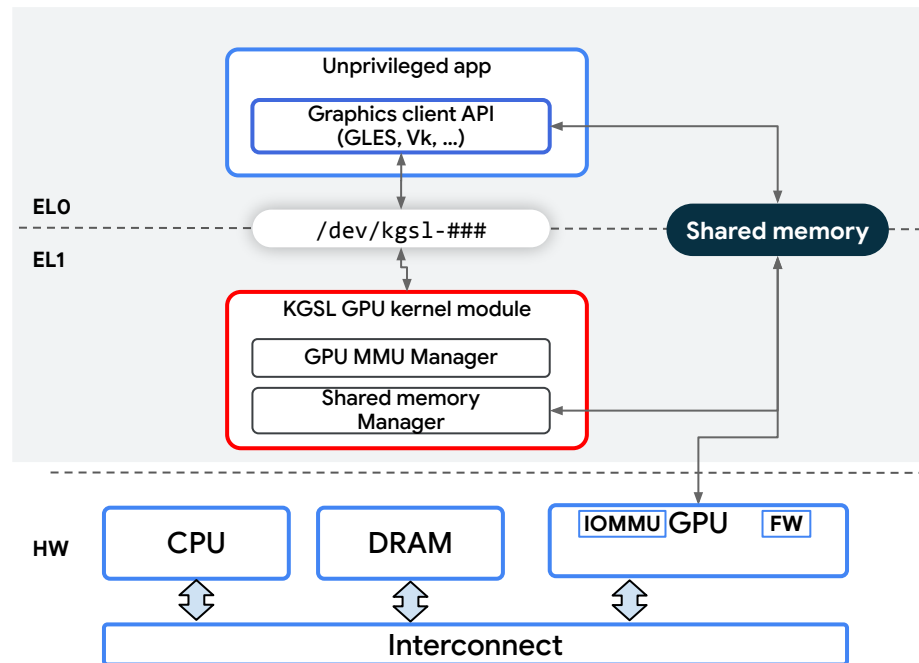
Assess the efficacy of security mitigations

Agenda

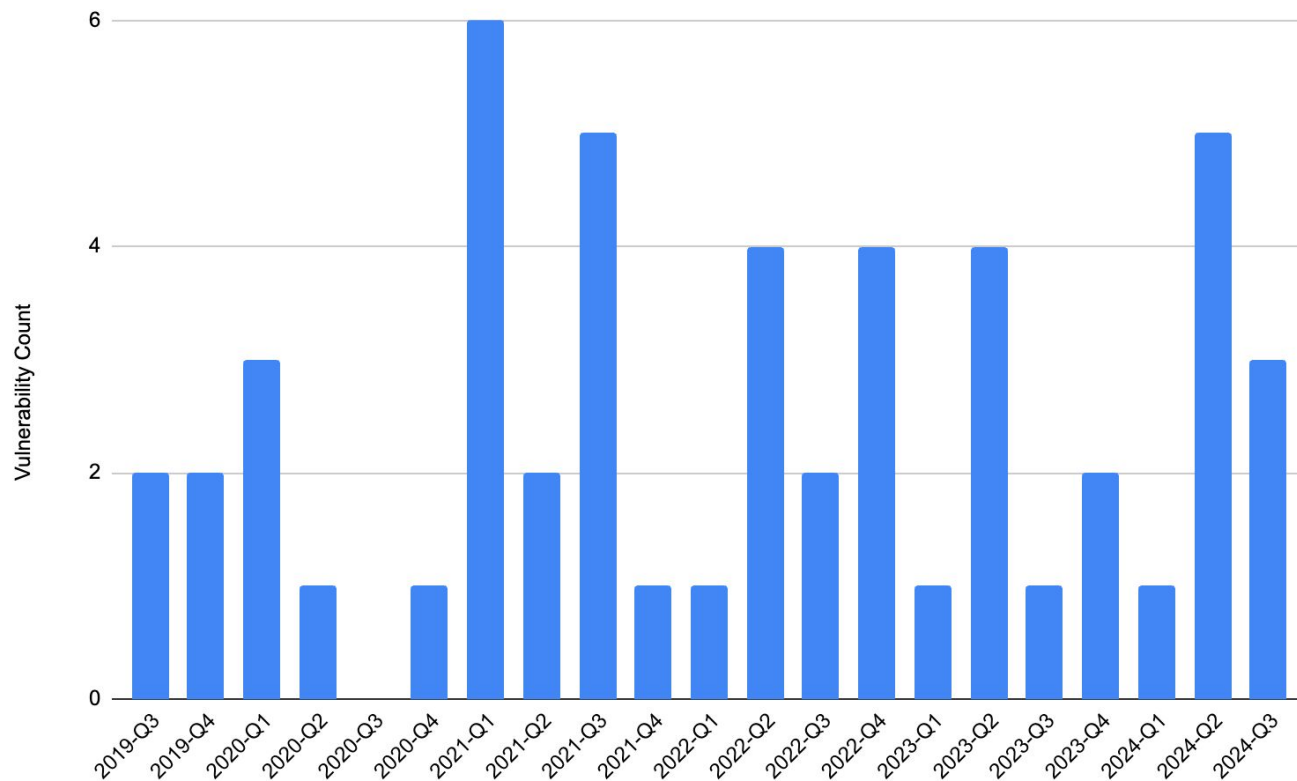
- Background Introduction
- Qualcomm Adreno GPU Introduction
- CVE-2024-23380 and Exploitation
- Vulnerability and Methodology Discussion

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

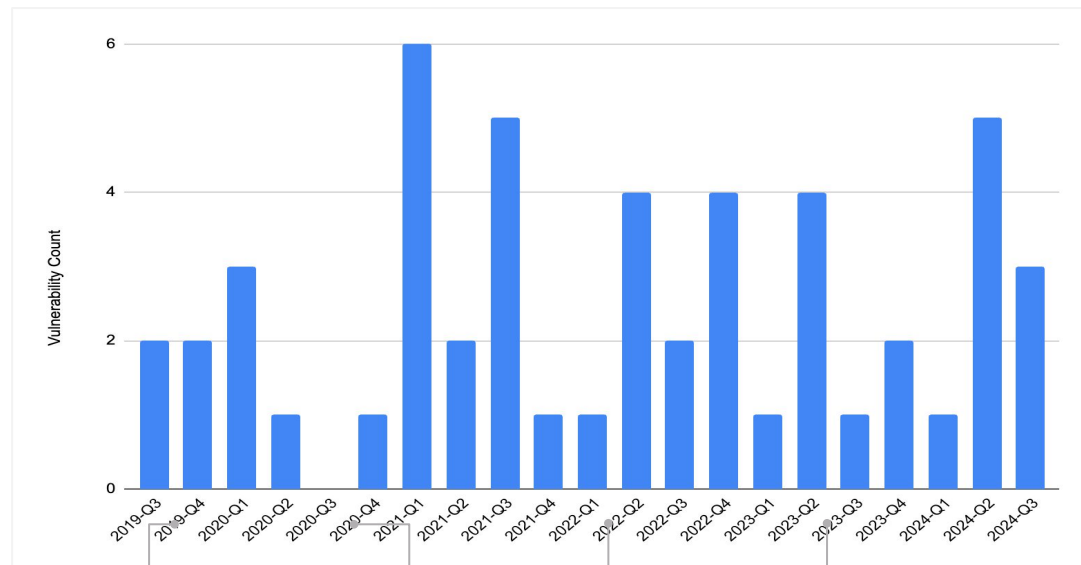


Adreno Driver Issues



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

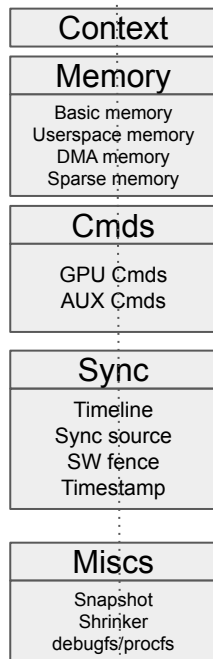
- **9+ issues* discovered**
- **We have exploited CVE-2024-23380**
 - Patched and published in the Security Bulletin of July 2024
 - We will discuss the issue and exploit
- **Timeline of CVE-2024-23380**
 - 12/13/2023 - Discovered and reported the issue to Qualcomm
 - 12/15/2023 - Qualcomm confirmed the issue
 - 12/21/2023 - Qualcomm submitted the patch
 - 04/01/2024 - Qualcomm sent out the patch and notified customers
 - 07/01/2024 - Qualcomm published the issue in Security Bulletin

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

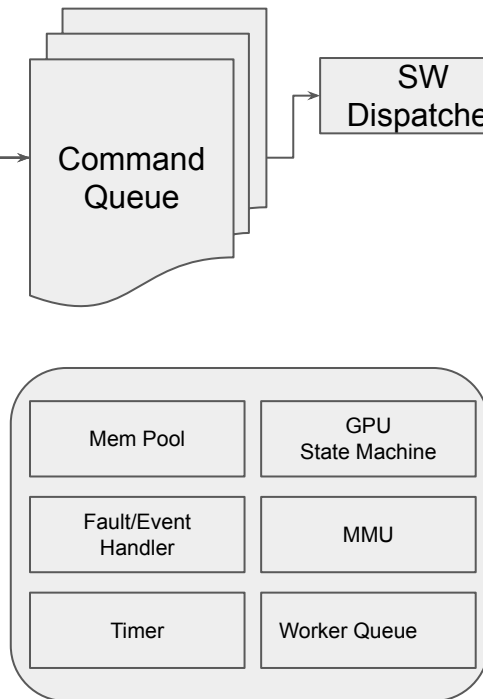
Qualcomm Adreno GPU Introduction

KGSL Introduction: Architecture

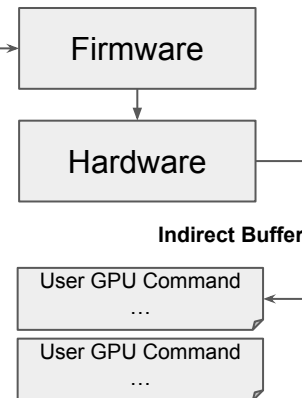
Userspace



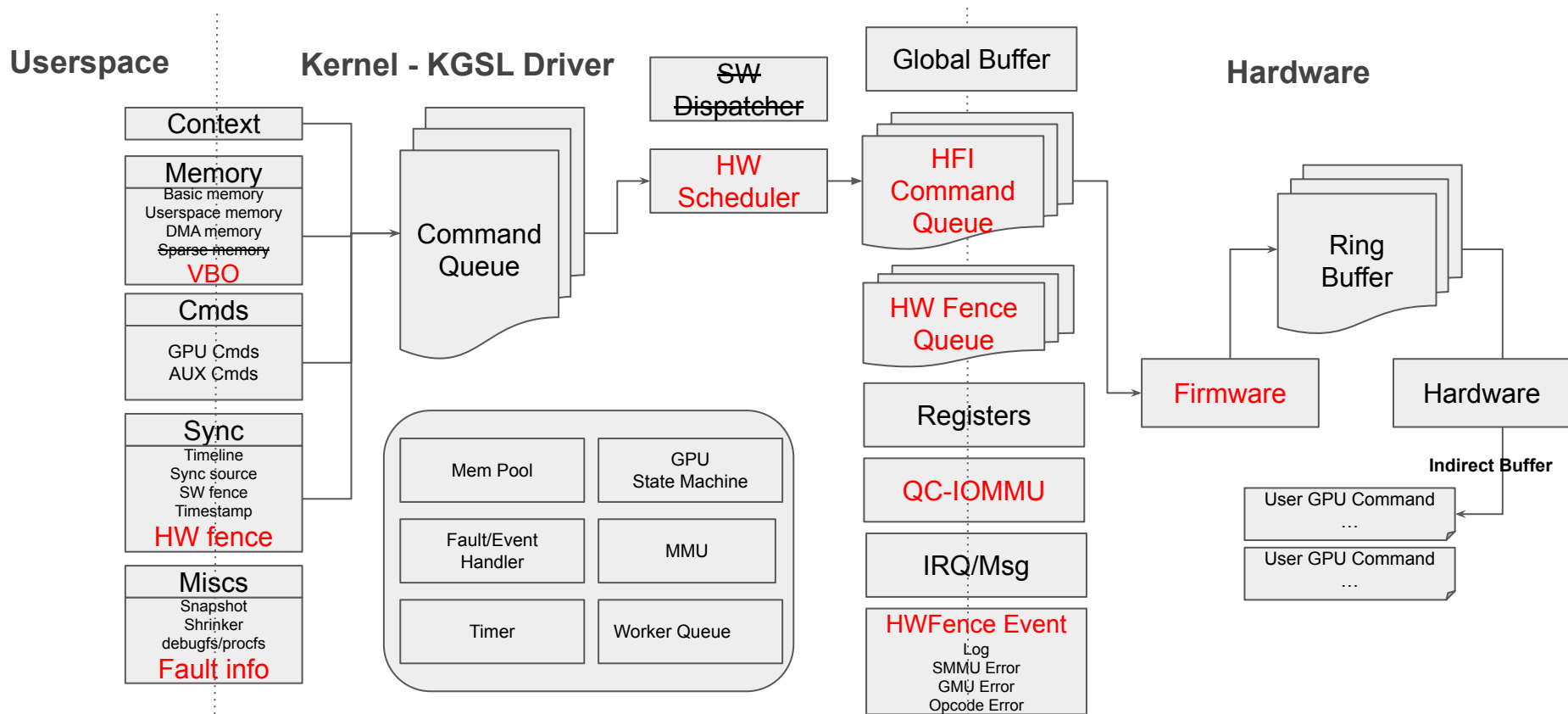
Kernel - KGSL Driver



Hardware



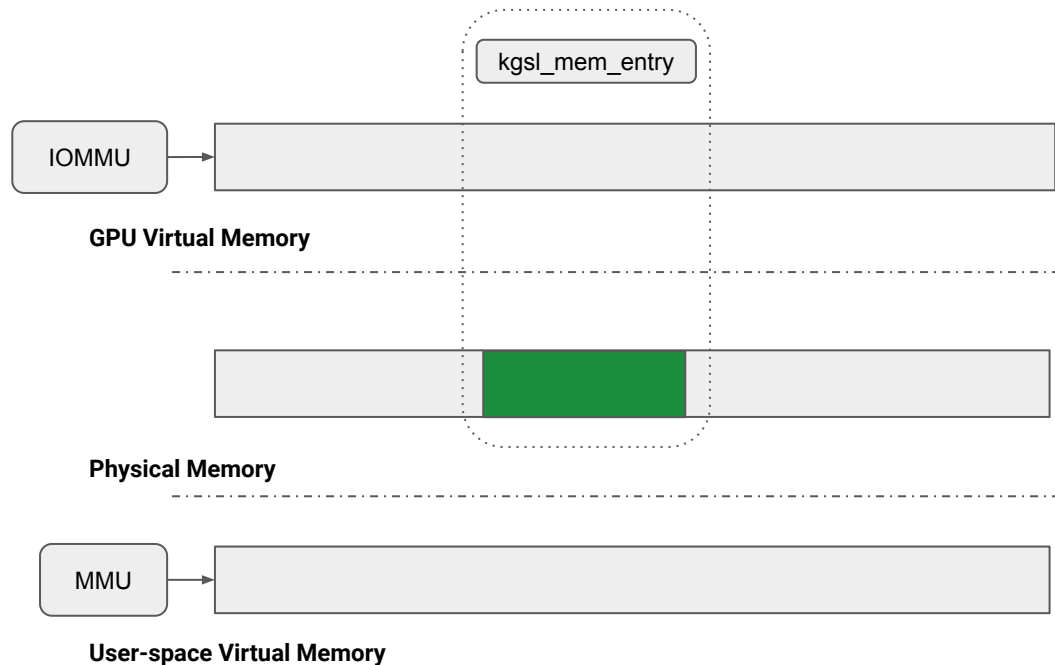
KGSL Introduction: Architecture (Gen7+)



GPU Basic Memory Object Allocation

`IOCTL_KGSL_GPUMEM_ALLOC` ioctl:

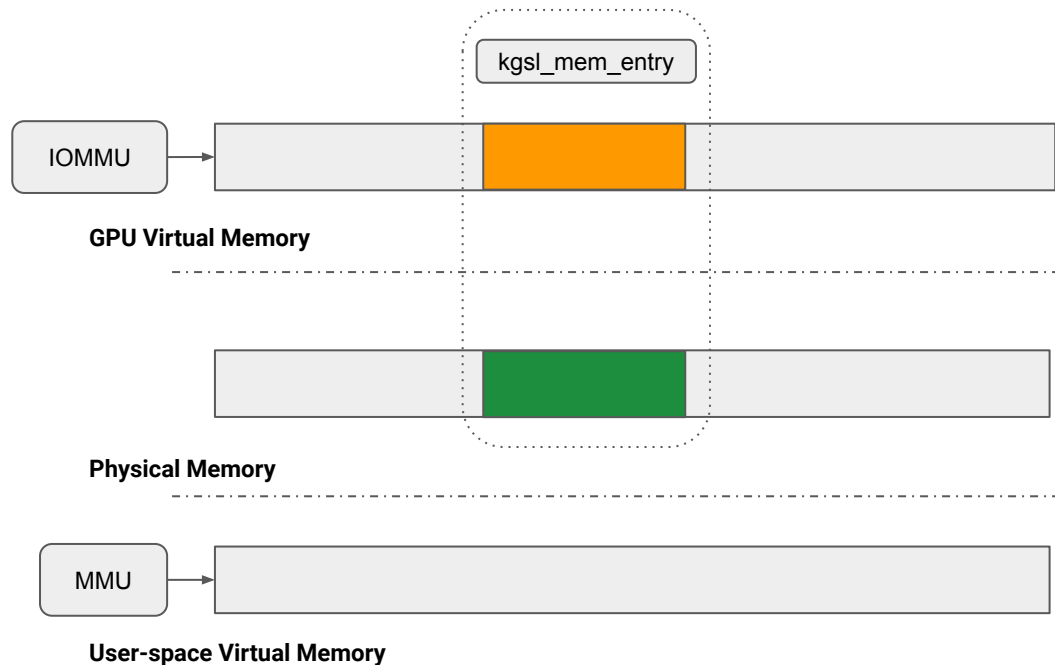
1. **Allocate number of requested physical pages**



GPU Basic Memory Object Allocation

`IOCTL_KGSL_GPUMEM_ALLOC` ioctl:

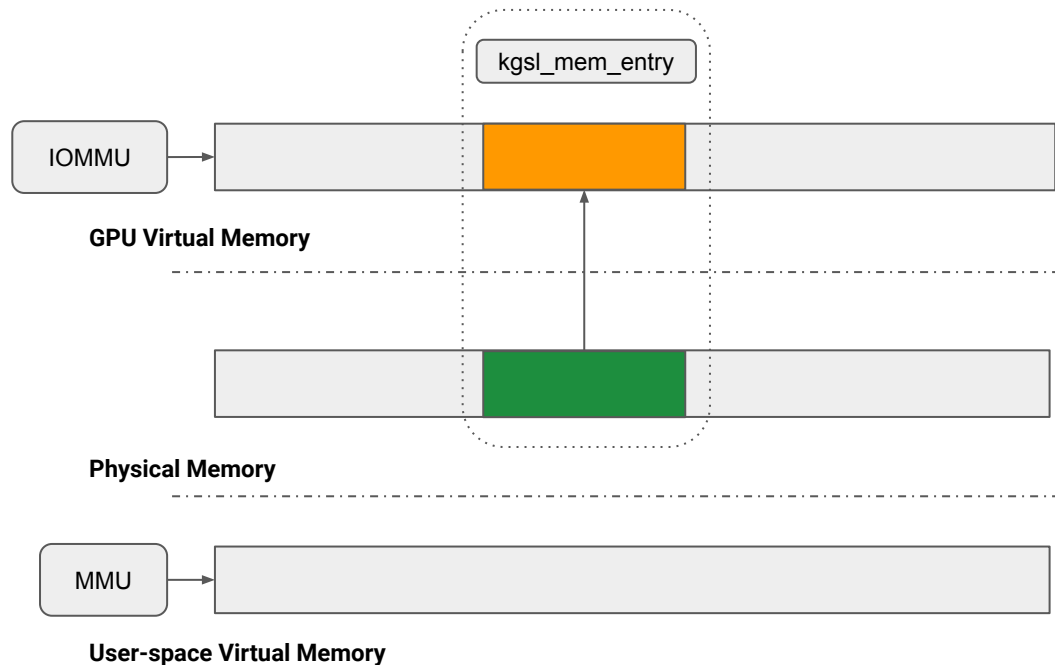
1. Allocate number of requested physical pages
2. **Allocate an address range in GPU virtual address space**



GPU Basic Memory Object Allocation

`IOCTL_KGSL_GPUMEM_ALLOC` ioctl:

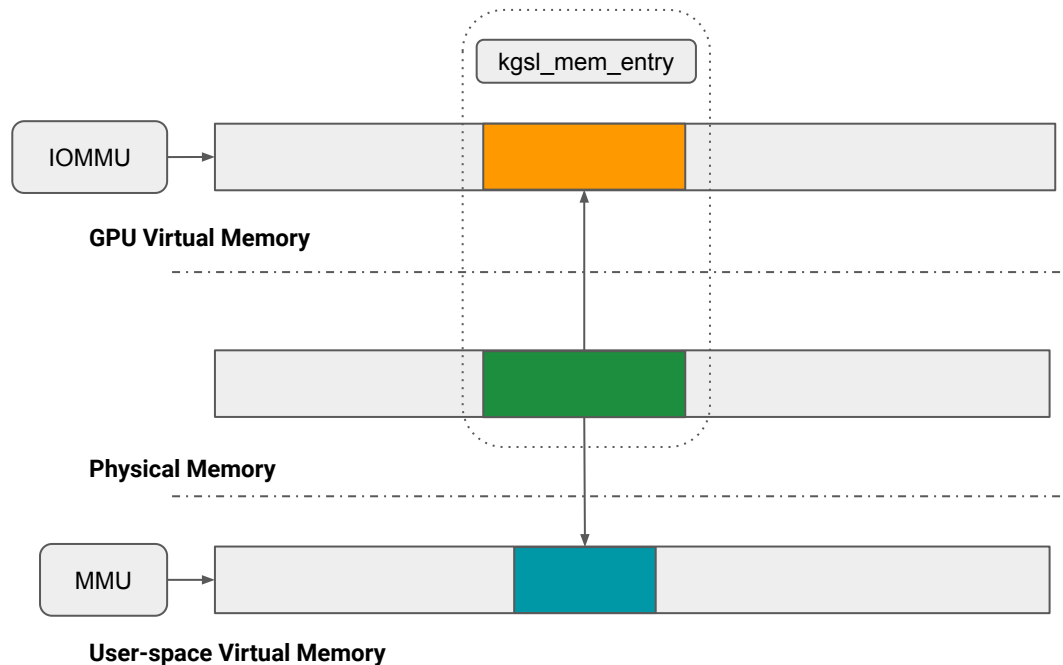
1. Allocate number of requested physical pages
2. Allocate an address range in GPU virtual address space
3. **Map allocated the physical pages to allocated virtual GPU address range**



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**

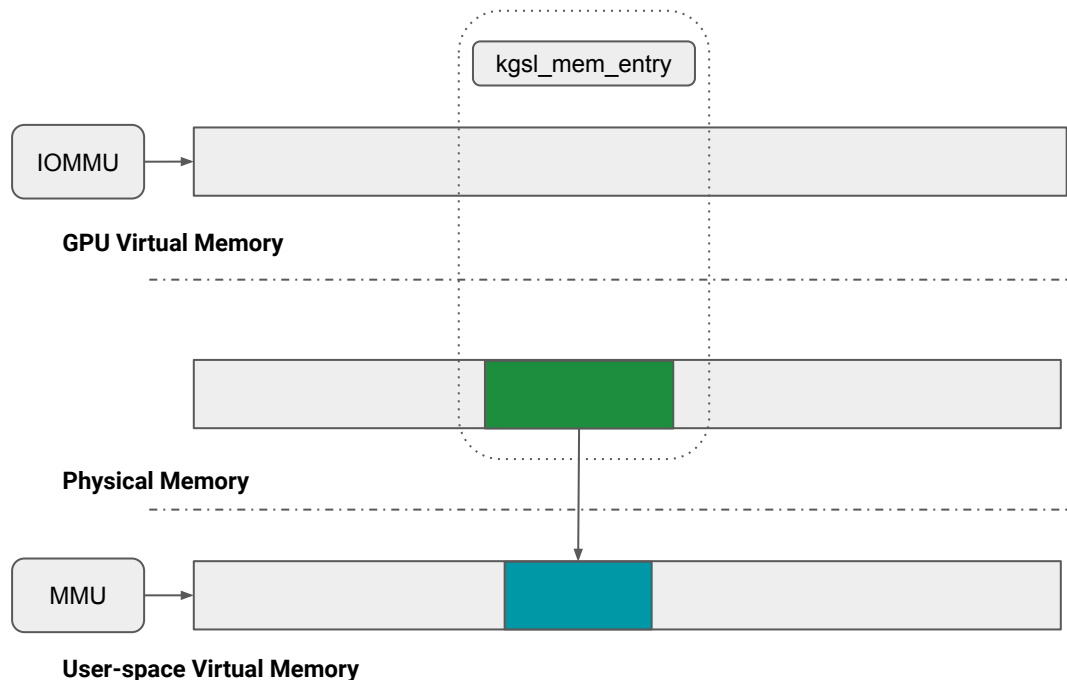


Importing User-space Memory Object

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

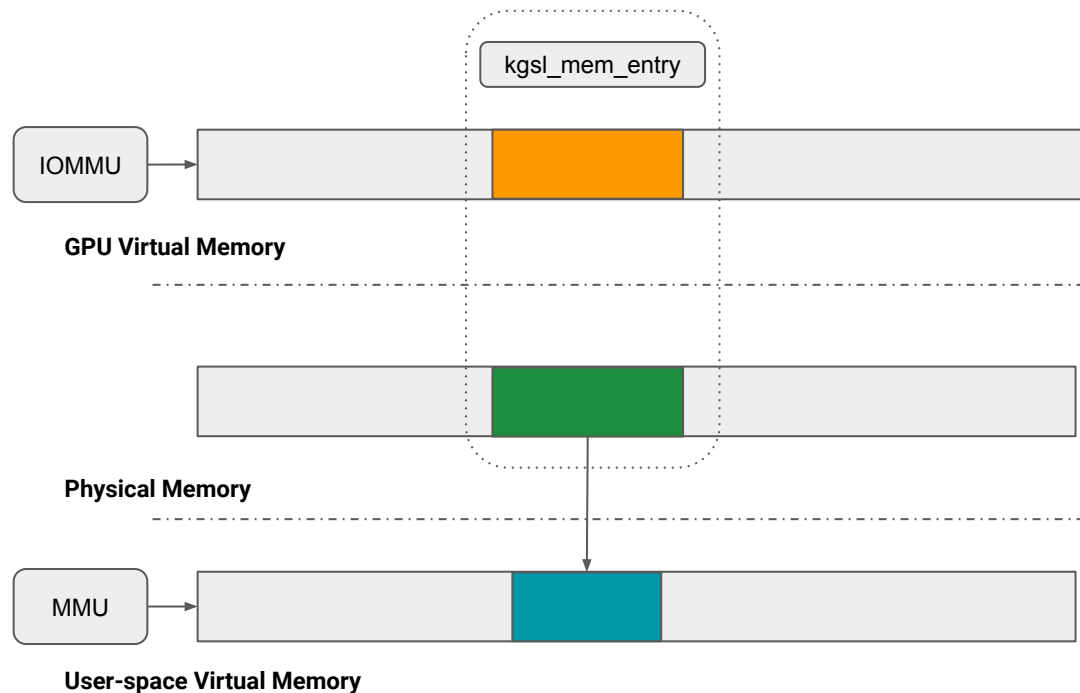


Importing User-space Memory Object

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**

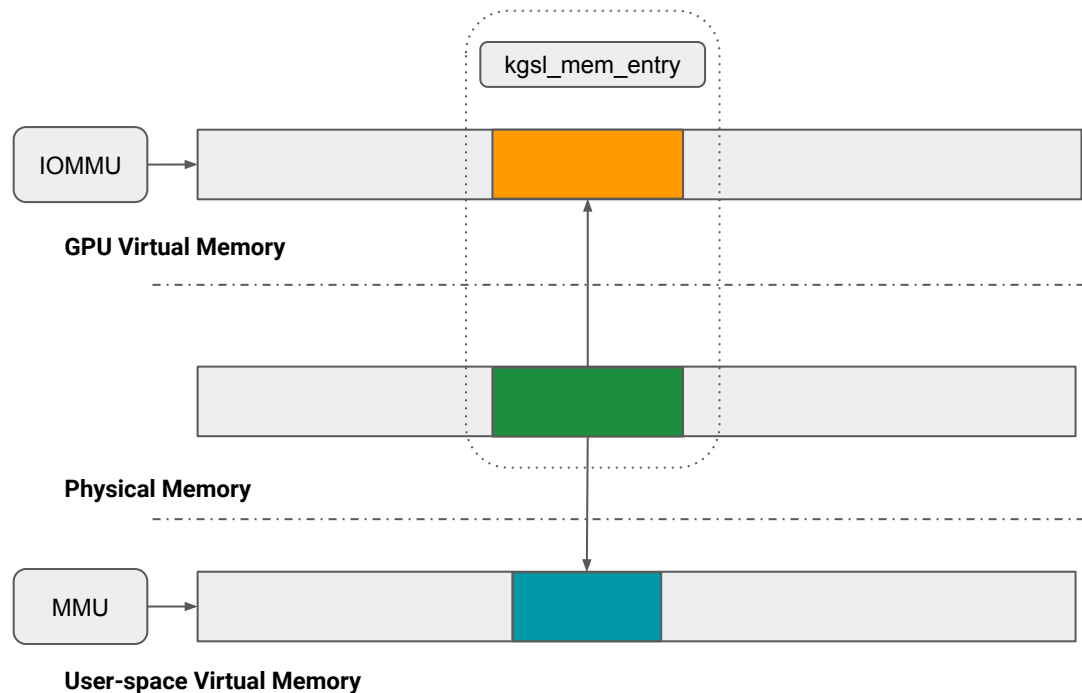


Importing User-space Memory Object

Initial state: allocated user-space memory object

`IOCTL_KGSL_GPUOBJ_IMPORT` ioctl:

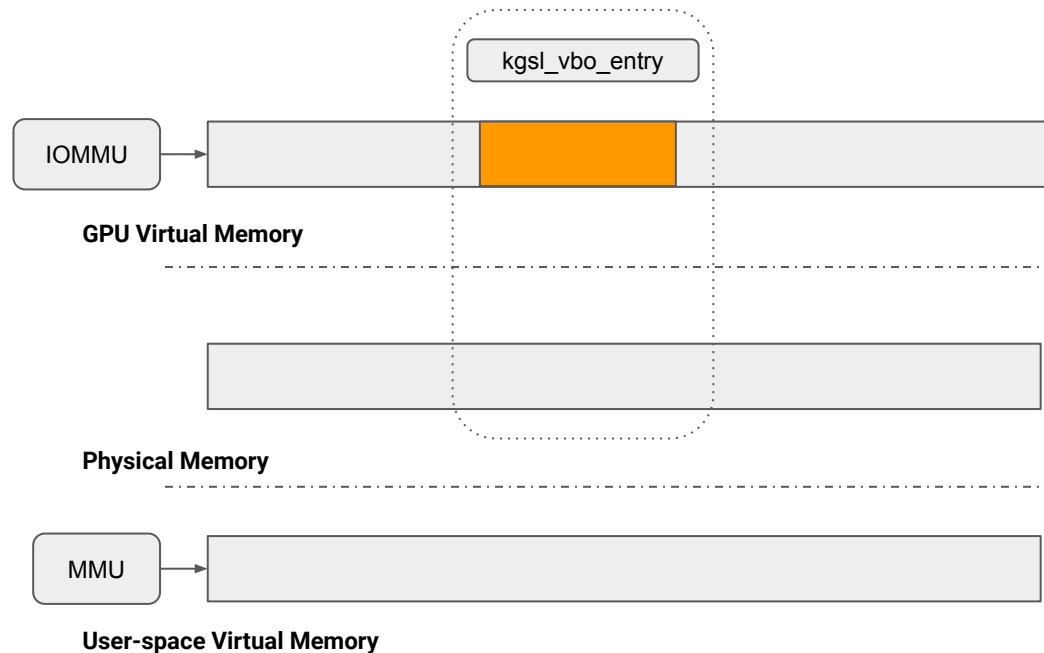
1. Get scatter list of physical pages corresponding to the user-space memory object
2. Allocate address range in GPU virtual address space
3. **Map physical pages in the scatter list to the allocated GPU virtual address range**



Virtual Buffer Object (VBO) Allocation

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

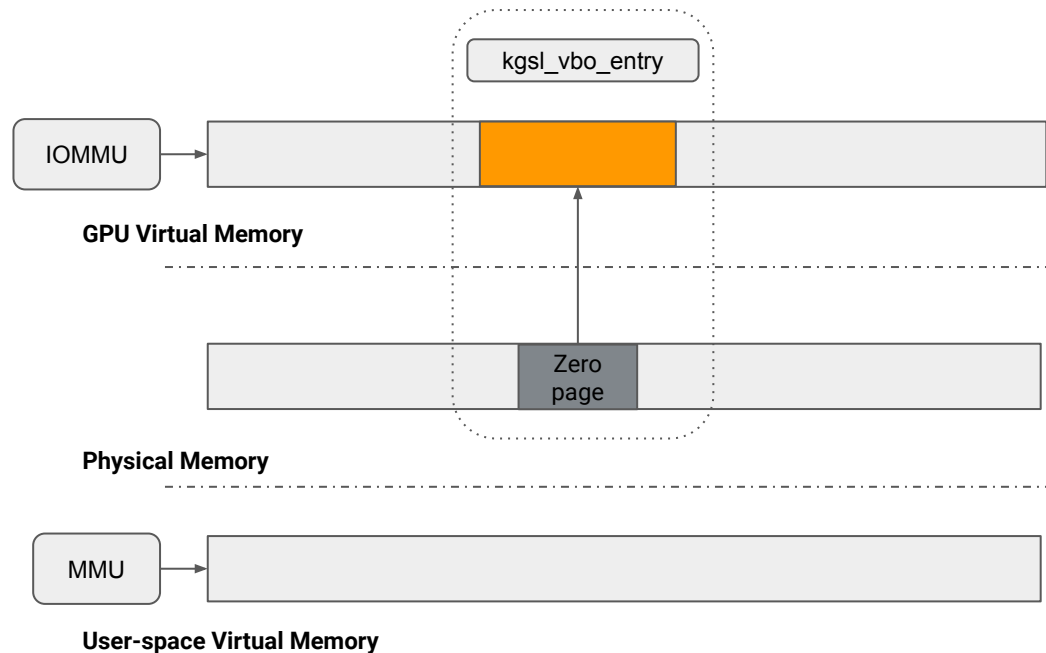
1. **Allocate address range in GPU virtual address space**



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**

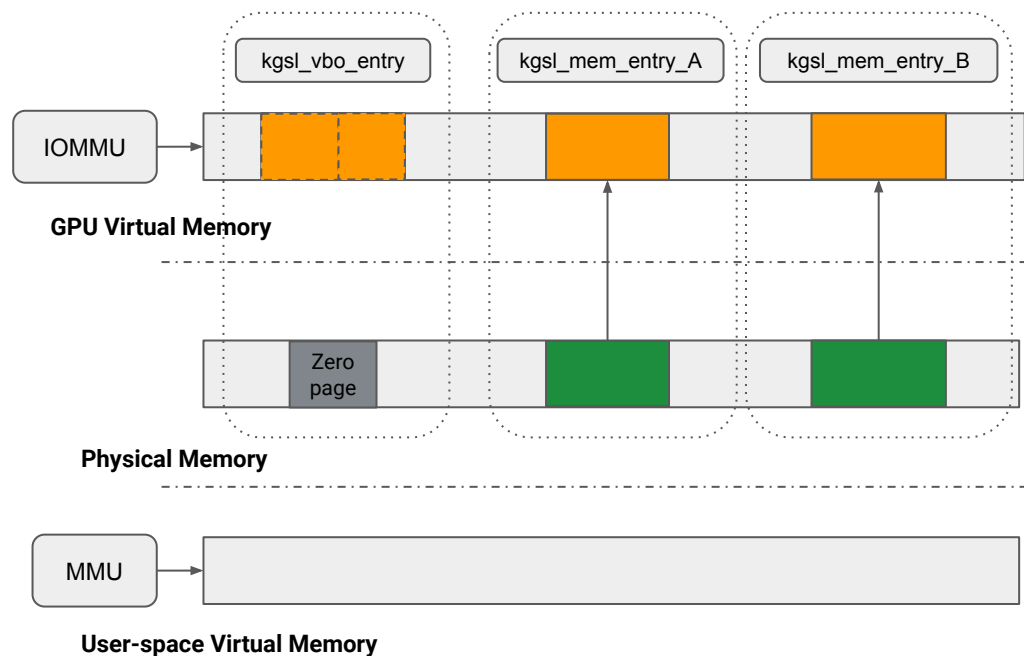


VBO: Binding Basic Memory Object

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

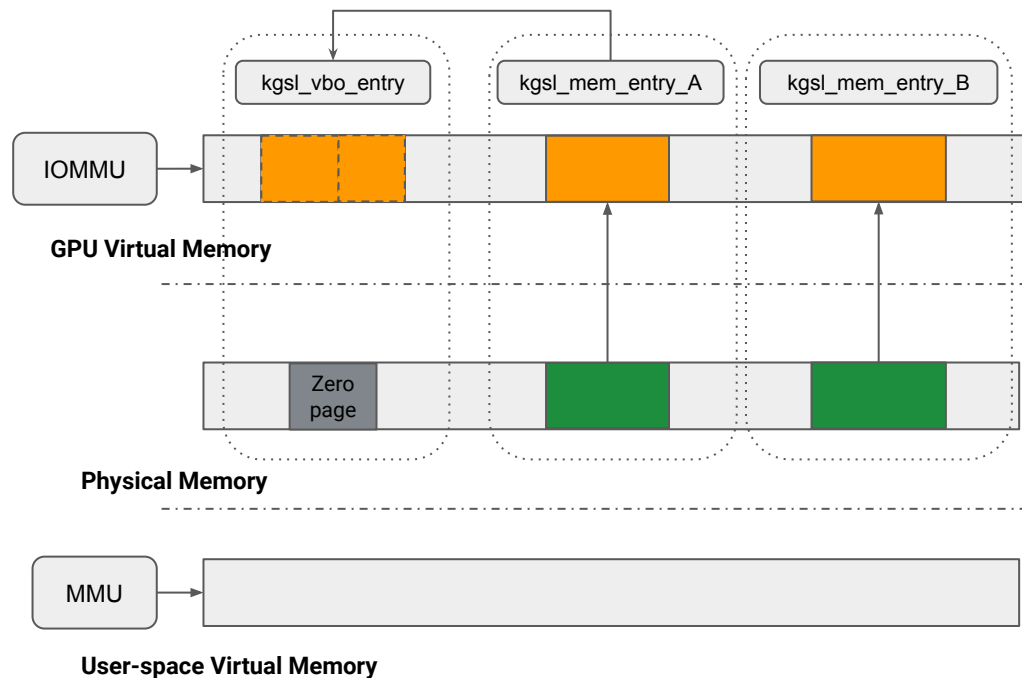


VBO: Binding Basic Memory Object

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**

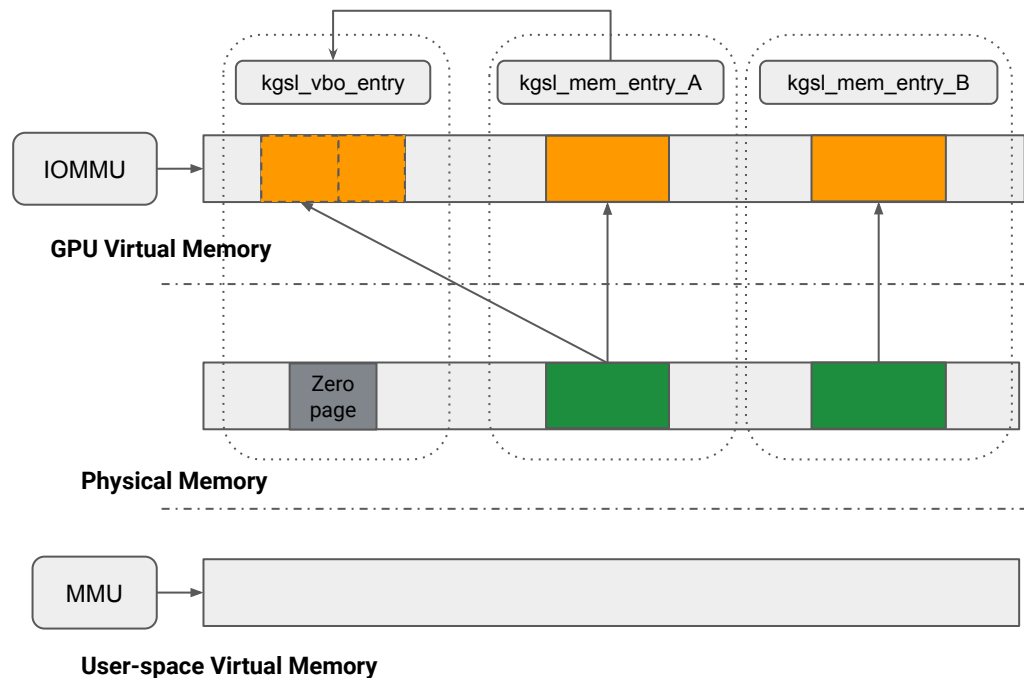


VBO: Binding Basic Memory Object

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**

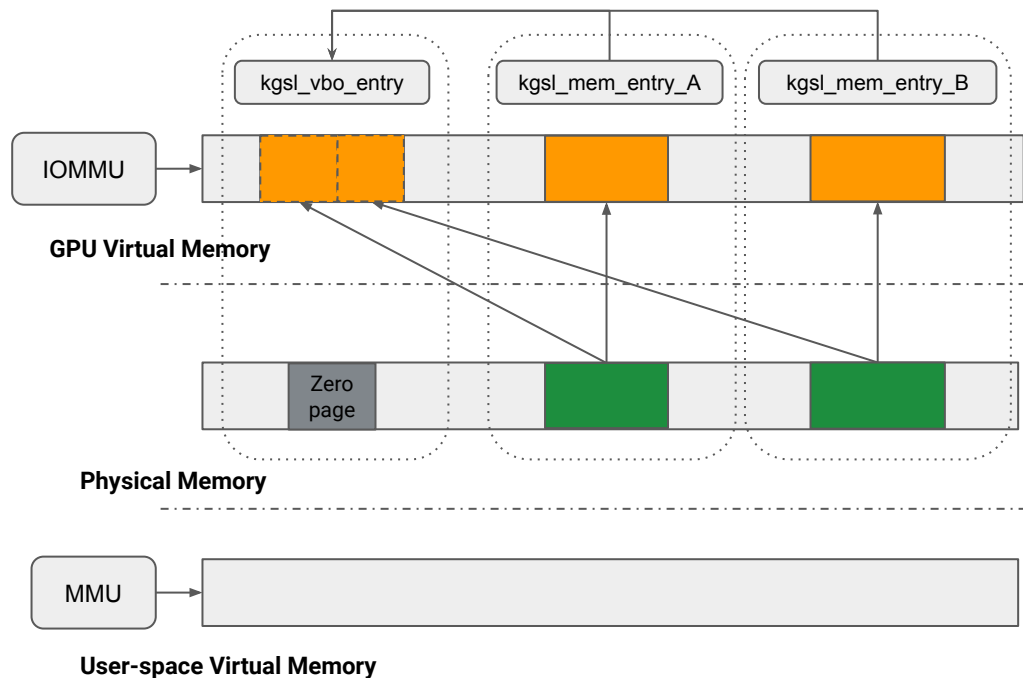


VBO: Binding Basic Memory Object

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**

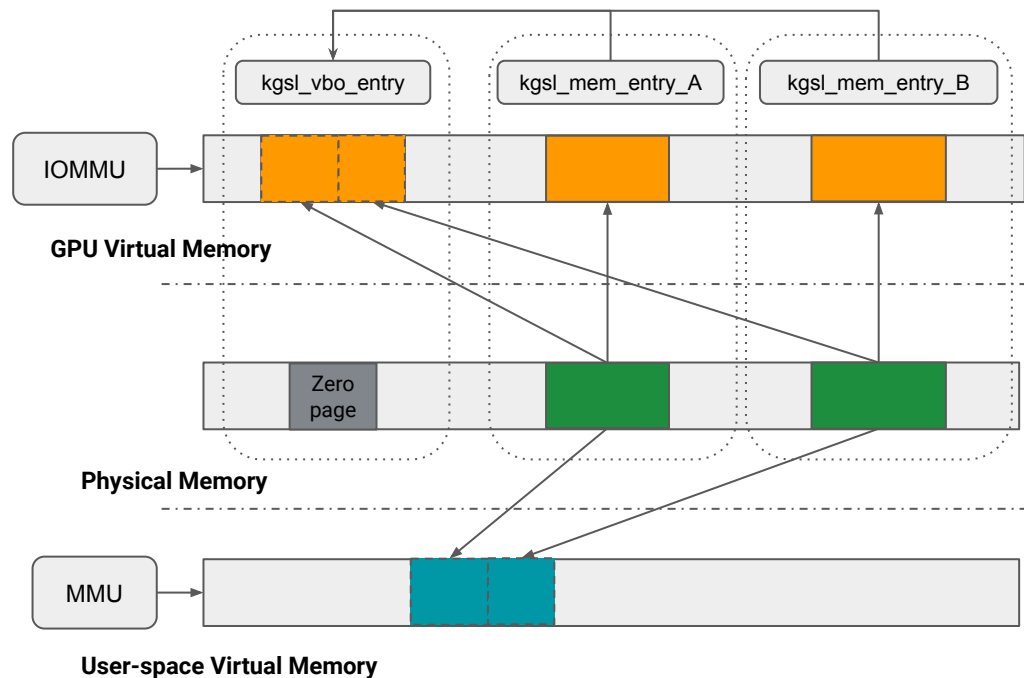


VBO: Binding Basic Memory Object

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 kgs1_memdesc_add_range(struct kgs1_mem_entry *target,
                                u64 start, u64 last, struct kgs1_mem_entry *entry, u64 offset)
{
    struct kgs1_memdesc *memdesc = &target->memdesc;

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

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

    trace_kgs1_mem_add_bind_range(target, range->range.start,
                                  range->entry, bind_range_len(range));
    mutex_unlock(&memdesc->ranges_lock);

    return kgs1_mmu_map_child(memdesc->pagetable, memdesc, start,
                              &entry->memdesc, offset, last - start + 1);
}
```

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

```
static int kgs1_memdesc_add_range(struct kgs1_mem_entry *target,
                                u64 start, u64 last, struct kgs1_mem_entry *entry, u64 offset)
{
    struct kgs1_memdesc *memdesc = &target->memdesc;

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

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

    trace_kgs1_mem_add_bind_range(target, range->range.start,
                                  range->entry, bind_range_len(range));
    mutex_unlock(&memdesc->ranges_lock);

    return kgs1_mmu_map_child(memdesc->pagetable, memdesc, start,
                              &entry->memdesc, offset, last - start + 1);
}
```

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

```
static int kgs1_memdesc_add_range(struct kgs1_mem_entry *target,
                                u64 start, u64 last, struct kgs1_mem_entry *entry, u64 offset)
{
    struct kgs1_memdesc *memdesc = &target->memdesc;

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

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

    trace_kgs1_mem_add_bind_range(target, range->range.start,
                                  range->entry, bind_range_len(range));
    mutex_unlock(&memdesc->ranges_lock);

    return kgs1_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, `kgs1_mmu_map_child` isn't protected by the mutex!
 - thus, basic memory object **may be unbound** by the time `kgs1_mmu_map_child` is invoked

```
static int kgs1_memdesc_add_range(struct kgs1_mem_entry *target,
                                u64 start, u64 last, struct kgs1_mem_entry *entry, u64 offset)
{
    struct kgs1_memdesc *memdesc = &target->memdesc;

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

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

    trace_kgs1_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 kgs1_mmu_map_child(memdesc->pagetable, memdesc, start,
                              &entry->memdesc, offset, last - start + 1);
}
```

CVE-2024-23380: The fix

```
125 static int kgs_l_memdesc_add_range(struct kgs_l_mem_entry *target,  
126     u64 start, u64 last, struct kgs_l_mem_entry *entry, u64 offset)  
127 {  
128     struct interval_tree_node *node, *next;  
129     struct kgs_l_memdesc *memdesc = &target->memdesc;  
130     struct kgs_l_memdesc_bind_range *range =  
131         bind_range_create(start, last, entry);  
132     int ret = 0;  
133  
134     if (IS_ERR(range))  
135         return PTR_ERR(range);  
136  
137     mutex_lock(&memdesc->ranges_lock);  
238     }  
239 }  
240
```

```
241 /* Add the new range */  
242 interval_tree_insert(&range->range, &memdesc->ranges);  
243  
244 trace_kgs_l_mem_add_bind_range(target, range->range.start,  
245     range->entry, bind_range_len(range));  
246 mutex_unlock(&memdesc->ranges_lock);  
247  
248- return kgs_l_mmu_map_child(memdesc->pagetable, memdesc, start,  
249-     &entry->memdesc, offset, last - start + 1);  
250
```

```
125 static int kgs_l_memdesc_add_range(struct kgs_l_mem_entry *target,  
126     u64 start, u64 last, struct kgs_l_mem_entry *entry, u64 offset)  
127 {  
128     struct interval_tree_node *node, *next;  
129     struct kgs_l_memdesc *memdesc = &target->memdesc;  
130     struct kgs_l_memdesc_bind_range *range =  
131         bind_range_create(start, last, entry);  
132     int ret = 0;  
133  
134     if (IS_ERR(range))  
135         return PTR_ERR(range);  
136  
137     mutex_lock(&memdesc->ranges_lock);  
238     }  
239 }  
240
```

```
241+ ret = kgs_l_mmu_map_child(memdesc->pagetable, memdesc, start,  
242+     &entry->memdesc, offset, last - start + 1);  
243+ if (ret)  
244+     goto error;  
245+  
246 /* Add the new range */  
247 interval_tree_insert(&range->range, &memdesc->ranges);  
248  
249 trace_kgs_l_mem_add_bind_range(target, range->range.start,  
250     range->entry, bind_range_len(range));  
251 mutex_unlock(&memdesc->ranges_lock);  
252  
253+ return ret;  
254
```

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

- GPU IOMMU misconfiguration can be achieved by invoking `kgs_l_memdesc_add_range` and `kgs_l_memdesc_remove_range` concurrently in two threads.

```
static int kgs_l_memdesc_add_range(struct kgs_l_mem_entry *target,
                                   u64 start, u64 last, struct kgs_l_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 kgs_l_mmu_map_child(memdesc->pagetable, memdesc, start,
                               &entry->memdesc, offset, last - start + 1);
}
```

```
static void kgs_l_memdesc_remove_range(struct kgs_l_mem_entry *target,
                                       u64 start, u64 last, struct kgs_l_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 (kgs_l_mmu_unmap_range(memdesc->pagetable,
                                     memdesc, range->range.start, bind_range_len(range)))
                continue;
            interval_tree_remove(node, &memdesc->ranges);
            kgs_l_mem_entry_put(range->entry);
            kfree(range);
        }
    }

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

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

Thread A

```
static int kgs_l_memdesc_add_range(struct kgs_l_mem_entry *target,  
    u64 start, u64 last, struct kgs_l_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);
```

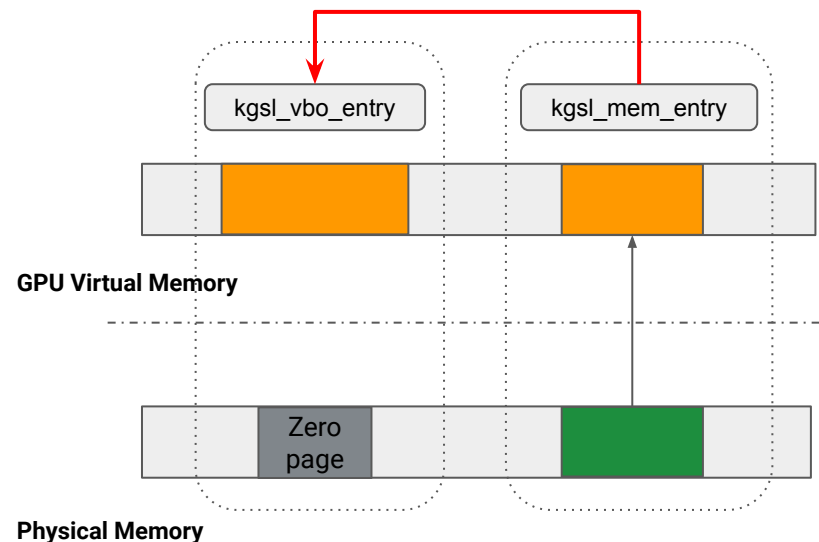
```
return kgs_l_mmu_map_child(memdesc->pagetable, memdesc, start,  
    &entry->memdesc, offset, last - start + 1);
```

Thread B

```
static void kgs_l_memdesc_remove_range(struct kgs_l_mem_entry *target,  
    u64 start, u64 last, struct kgs_l_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 (kgs_l_mmu_unmap_range(memdesc->pagetable,  
                memdesc, range->range.start, bind_range_len(range)))  
                continue;  
            interval_tree_remove(node, &memdesc->ranges);  
            ...  
            kgs_l_mem_entry_put(range->entry);  
            kfree(range);  
        }  
    }  
    mutex_unlock(&memdesc->ranges_lock);  
}
```

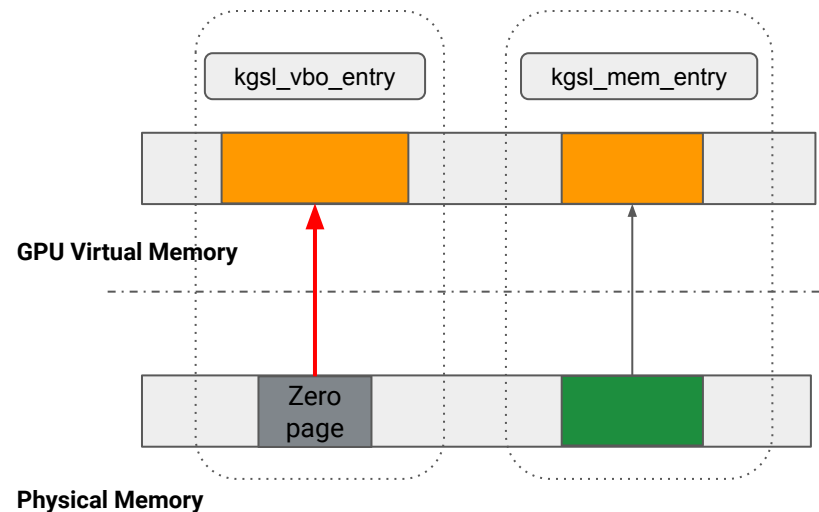
CVE-2024-23380: Triggering the vulnerability (3)

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	



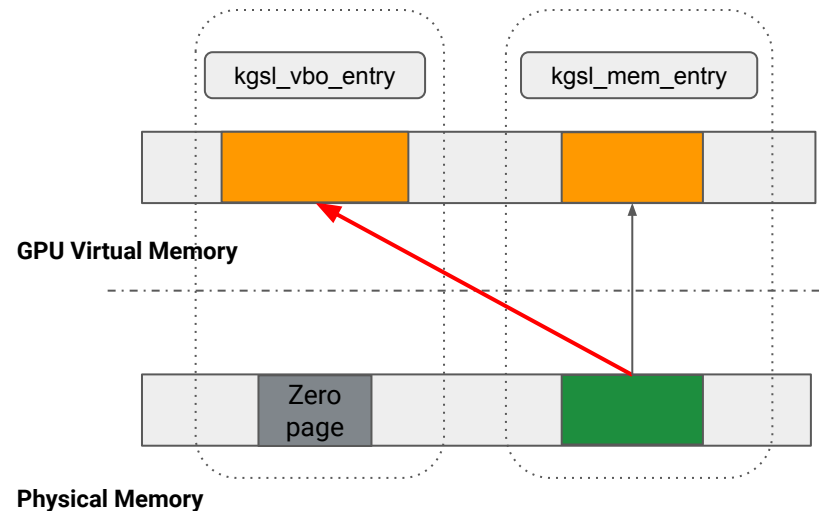
CVE-2024-23380: Triggering the vulnerability (4)

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	



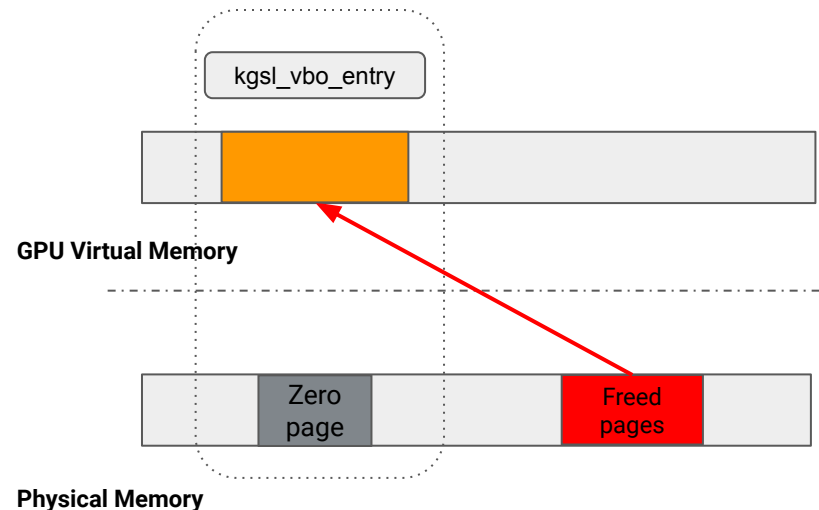
CVE-2024-23380: Triggering the vulnerability (5)

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	



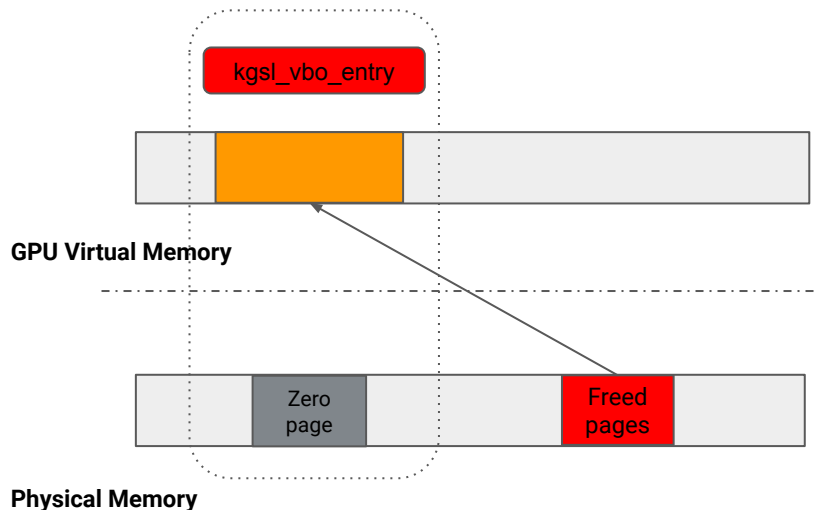
CVE-2024-23380: Triggering the vulnerability (6)

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 to get the status of whether the issue 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” attack¹

[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

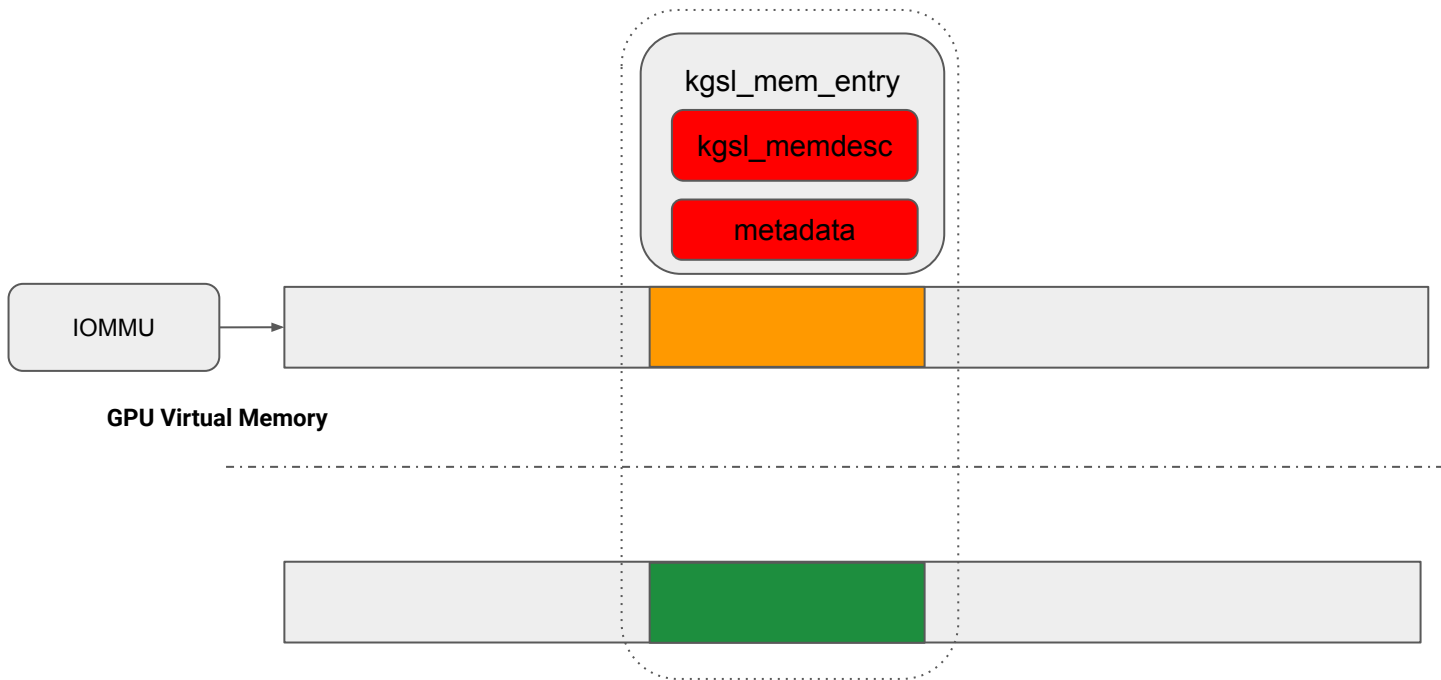
Overwrite contents of struct `kgs_l_memdesc`

- **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

```
struct kgs_l_mem_entry {  
    struct kref refcount;  
    struct kgs_l_memdesc memdesc;  
    void *priv_data;  
    struct rb_node node;  
    unsigned int id;  
    struct kgs_l_process_private *priv;  
    int pending_free;  
    char metadata[KGSL_GPUOBJ_ALLOC_METADATA_MAX + 1];  
    struct work_struct work;  
};
```

```
struct kgs_l_memdesc {  
    struct kgs_l_pagetable *pagetable;  
    void *hostptr;  
    unsigned int hostptr_count;  
    uint64_t qpuaddr;  
    phys_addr_t physaddr;  
    uint64_t size;  
    unsigned int priv;  
    struct sg_table *sgt;  
    const struct kgs_l_memdesc_ops *ops;  
};
```

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



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

```
struct kgs_l_memdesc {  
    struct kgs_l_pagetable *pagetable;  
    void *hostptr;  
    unsigned int hostptr_count;  
    uint64_t gpuaddr;  
    phys_addr_t physaddr;  
    uint64_t size;  
    unsigned int priv;  
    struct sq_table *sq;  
    const struct kgs_l_memdesc_ops *ops;  
};
```

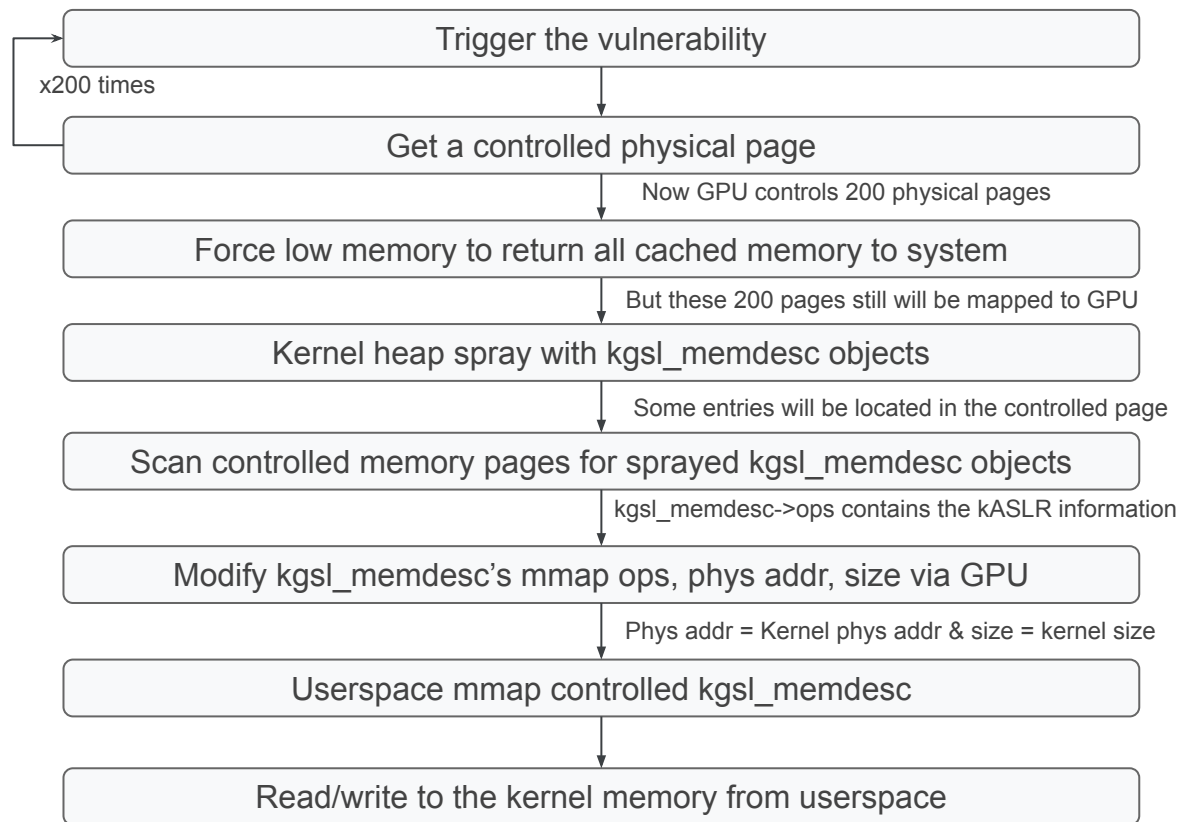
```
static const struct kgs_l_memdesc_ops kgs_l_page_ops = {  
    .free = kgs_l_free_pages,  
    .vmflags = VM_DONTDUMP | VM_DONTEXPAND | VM_DONTCOPY,  
    .vmfault = kgs_l_paged_vmfault,  
    .map_kernel = kgs_l_paged_map_kernel,  
    .unmap_kernel = kgs_l_paged_unmap_kernel,  
    .put_gpuaddr = kgs_l_unmap_and_put_gpuaddr,  
};
```

```
static const struct kgs_l_memdesc_ops kgs_l_contiguous_ops = {  
    .free = kgs_l_contiguous_free,  
    .vmflags = VM_DONTDUMP | VM_PFNMAP | VM_DONTEXPAND | VM_DONTCOPY,  
    .vmfault = kgs_l_contiguous_vmfault,  
    .put_gpuaddr = kgs_l_unmap_and_put_gpuaddr,  
};
```

```
static vm_fault_t kgs_l_contiguous_vmfault(struct kgs_l_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);  
}
```

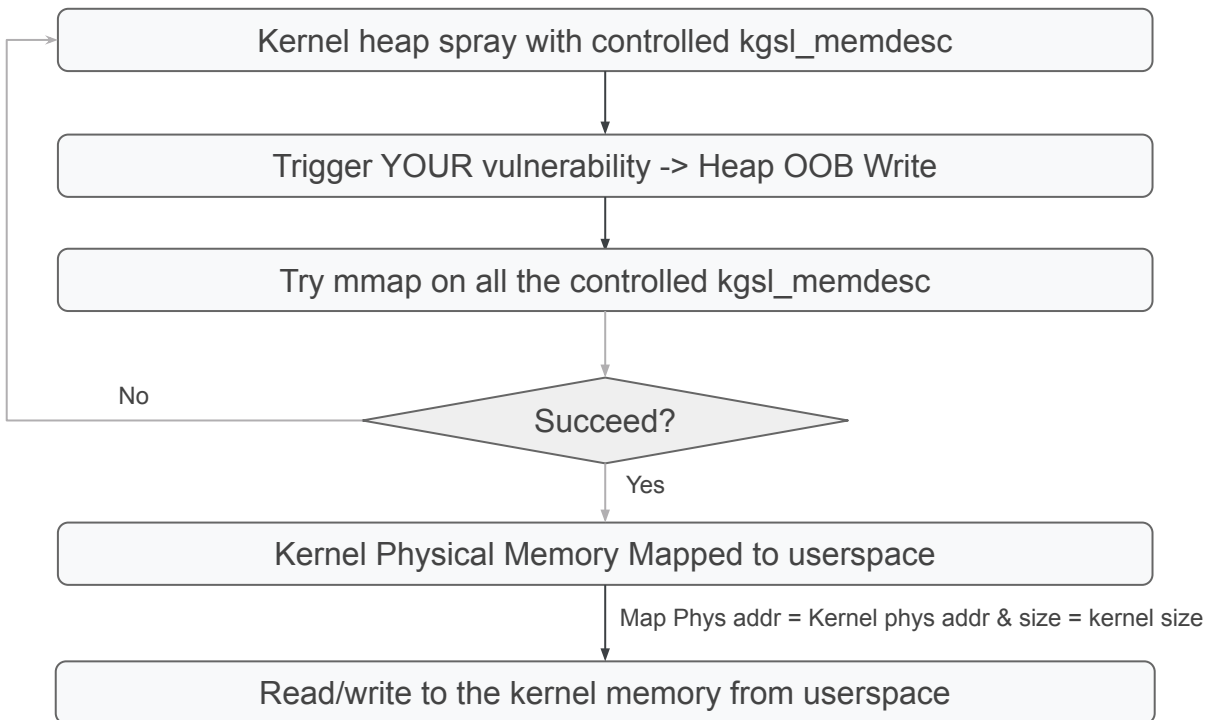
1. Do mmap on this kgs_l_memdesc
2. When userspace probes the mmap'ed memory, it will trigger vmfault and setup the mapping for memdesc->physaddr

CVE-2024-23380: Exploitation



```
struct kgs_l_memdesc {  
    struct kgs_l_pagetable *pagetable;  
    void *hostptr;  
    unsigned int hostptr_count;  
    uint64_t qpuaddr;  
    phys_addr_t physaddr;  
    uint64_t size;  
    unsigned int priv;  
    struct sg_table *sgt;  
    const struct kgs_l_memdesc_ops *ops;  
};
```

Exploitation: Expanding to Generic Cases



Heap

Overflow

```
struct kgs_l_memdesc {  
    struct kgs_l_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 kgs_l_memdesc_ops *ops;  
};
```

Exploitation: bypassing kASLR

Method A: Direct kASLR bypass

```

[ ]:/ # cat /proc/kallsyms | grep "ffffffdd39066f"
ffffffdd39066f60 r kgs_l_contiguous_ops [msm_kgs_l]
ffffffdd39066fc0 r kgs_l_secure_page_ops [msm_kgs_l]
ffffffdd39066f90 r kgs_l_secure_system_ops [msm_kgs_l]
ffffffdd39066ff0 r kgs_l_system_ops [msm_kgs_l]

```

Ops=kgs_l_secure_page_ops

0C	F6 60 93 DD FF FF FF
----	----------------------

→kgs_l_contiguous_ops

06	F6 60 93 DD FF FF FF
----	----------------------

Known The Same

Method B: Brute Force 16 trials

```

ffffffdd39066f60 r kgs_l_contiguous_ops [msm_kgs_l]
ffffffdd39067020 r kgs_l_page_ops [msm_kgs_l]

```

Ops=kgs_l_page_ops

02 07	60 93 DD FF FF FF
-------	-------------------

→kgs_l_contiguous_ops

06 F6	60 93 DD FF FF FF
-------	-------------------

Guess these 4 bits = 16 trials

Heap

Overflow

```

struct kgs_l memdesc {
    struct kgs_l_pagetable *pagetable;
    void *hostptr;
    unsigned int hostptr_count; Not used
    uint64_t gpuaddr;
    phys_addr_t physaddr;
    uint64_t size;
    unsigned int priv;
    struct sg_table *sgt;
    const struct kgs_l_memdesc_ops *ops;
}

```

Demo

<https://youtu.be/qhY9cbvuuAU>

Methodology & Bug Discovery

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

IOMMU Use-After-Free

CVE-2024-21471 (May, 2024) - Qualcomm Internal

Memory corruption when IOMMU unmap of a GPU buffer, the memory entries are getting release.

CVE-2024-23373 (July, 2024) - Man Yue Mo

Memory corruption when IOMMU unmap operation fails, the DMA and anon buffers are getting released.

CVE-2024-????? - Xiling Gong

Will be public in the Security Bulletin of August 2024

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

Future Work

- Fuzzing
 - Especially for race-condition issues
- Firmware Emulation
- Static analysis



Thank You! Questions?

<https://androidoffsec.withgoogle.com>

androidoffsec-external@google.com