March 10, 2021

One day short of a full chain: Part 1 - Android Kernel arbitrary code execution

Man Yue Mo

In this series of posts, I'll exploit three bugs that I reported last year: a use-after-free in the renderer of Chrome, a Chromium sandbox escape that was reported and fixed while it was still in beta, and a use-after-free in the Qualcomm msm kernel. Together, these three bugs form an exploit chain that allows remote kernel code execution by visiting a malicious website in the beta version of Chrome. While the full chain itself only affects beta version of Chrome, both the renderer RCE and kernel code execution existed in stable versions of the respective software. All of these bugs had been patched for quite some time, with the last one patched on the first of January.

Vulnerabilities used in the series

The three vulnerabilities that I'm going to use are the following. To achieve arbitrary kernel code execution from a compromised beta version of Chrome, I'll use CVE-2020-11239, which is a use-after-free in the kgsl driver in the Qualcomm msm kernel. This vulnerability was reported in July 2020 to the Android security team as A-161544755 (GHSL-2020-375) and was patched in the Januray Bulletin. In the security bulletin, this bug was mistakenly associated with A-168722551, although the Android security team has since confirmed to acknowledge me as the original reporter of the issue. (However, the acknowledgement page had not been updated to reflect this at the time of writing.) For compromising Chrome, I'll use CVE-2020-15972, a use-after-free in web audio to trigger a renderer RCE. This is a duplicate bug, for which an anonymous researcher reported about three weeks before I reported it as 1125635 (GHSL-2020-167). To escape the Chrome sandbox and gain control of the browser process, I'll use CVE-2020-16045, which was reported as 1125614 (GHSL-2020-165). While the exploit uses a component that was only enabled in the beta version of Chrome, the bug would probably have made it to the stable version and be exploitable if it weren't reported. Interestingly, the renderer bug CVE-2020-15972 was fixed in version 86.0.4240.75, the same version where the sandbox escape bug would have made into stable version of Chrome (if not reported), so these two bugs literally missed each other by one day to form a stable full chain.

Qualcomm kernel vulnerability

The vulnerability used in this post is a use-after-free in the kernel graphics support layer (kgsl) driver. This driver is used to provide an interface for apps in the userland to communicate with the Adreno gpu (the gpu that is used on Qualcomm's snapdragon chipset). As it is necessary for apps to access this driver to render themselves, this is one of the few drivers that can be reached from third-party applications on all phones that use Qualcomm chipsets. The vulnerability itself can be triggered on all of these phones that have a kernel version 4.14 or above, which should be the case for many mid-high end phones released after late 2019, for example, Pixel 4, Samsung Galaxy S10, S20, and A71. The exploit in this post, however, could not be launched directly from a third party App on Pixel 4 due to further SELinux restrictions, but it can be launched from third party Apps on Samsung phones and possibly some others as well. The exploit in this post is largely developed with a Pixel 4 running AOSP built from source and then adapted to a Samsung Galaxy A71. With some adjustments of parameters, it should probably also work on flagship models like Samsung Galaxy S10 and S20 (Snapdragon version), although I don't have those phones and have not tried it out myself.

The vulnerability here concerns the ioctl calls `IOCTL_KGSL_GPUOBJ_IMPORT` and `IOCTL_KGSL_MAP_USER_MEM`. These calls are used by apps to create shared memory between itself and the kgsl driver.

When using these calls, the caller specifies a user space address in their process, the size of the shared memory, as well as the type of memory objects to create. After making the ioctl call successfully, the kgsl driver would map the user supplied memory into the gpu's memory space and be able to access the user supplied memory. Depending on the type of the memory specified in the ioctl call parameter, different mechanisms are used by the kernel to map and access the user space memory.

The two different types of memory are `KGSL_USER_MEM_TYPE_ADDR`, which would ask kgsl to pin the user memory supplied and perform direct I/O on those memory (see, for example, `Performing Direct I/O` section here). The caller can also specify the memory type to be `KGSL_USER_MEM_TYPE_ION`, which would use a direct memory access (DMA) buffer (for example, `Direct Memory Access` section here) allocated by the ion allocator to allow the gpu to access the DMA buffer directly. We'll look at the DMA buffer a bit more later as it is important to both the vulnerability and the exploit, but for now, we just need to know that there are two different types of memory objects that can be created from these ioctl calls. When using these ioctl, a `kgsl_mem_entry` object will first be created, and then the type of memory is checked to make sure that the `kgsl_mem_entry` is correctly populated. In a way, these ioctl calls act like constructors of `kgsl_mem_entry`:

```
long kgsl_ioctl_gpuobj_import(struct kgsl_device_private *dev_priv,
                unsigned int cmd, void *data)
{
    ...
    entry = kgsl_mem_entry_create();
    ...
        if (param->type == KGSL_USER_MEM_TYPE_ADDR)
                ret = _gpuobj_map_useraddr(dev_priv->device, private->pagetable,
                        entry, param);
    //KGSL_USER_MEM_TYPE_ION is translated to KGSL_USER_MEM_TYPE_DMABUF
        else if (param->type == KGSL_USER_MEM_TYPE_DMABUF)
                ret = _gpuobj_map_dma_buf(dev_priv->device, private->pagetable,
                        entry, param, &fd);
        else
                ret = -ENOTSUPP;
```

In particular, when creating a `kgsl_mem_entry` with DMA type memory, the user supplied DMA buffer will be "attached" to the gpu, which will then allow the gpu to share the DMA buffer. The process of sharing a DMA buffer with a device on Android generally looks like this (see this for the general process of sharing a DMA buffer with a device):

1. The user creates a DMA buffer using the ion allocator. On Android, ion is the concrete implementation of DMA buffers, so sometimes the terms are used interchangeably, as in the kgsl code here, in which `KGSL_USER_MEM_TYPE_DMABUF` and `KGSL_USER_MEM_TYPE_ION` refers to the same thing.

2. The ion allocator will then allocate memory from the ion heap, which is a special region of memory seperated from the heap used by the `kmalloc` family of calls. I'll cover more about the ion heap later in the post.
3. The ion allocator will return a file descriptor to the user, which is used as a handle to the DMA buffer.
4. The user can then pass this file descriptor to the device via an appropriate ioctl call.
5. The device then obtains the DMA buffer from the file descriptor via `dma_buf_get` and uses `dma_buf_attach` to attach it to itself.
6. The device uses `dma_buf_map_attachment` to obtain the `sg_table` of the DMA buffer, which contains the locations and sizes of the backing stores of the DMA buffer. It can then use it to access the buffer.
7. After this, both the device and the user can access the DMA buffer. This means that the buffer can now be modified by both the cpu (by the user) and the device. So care must be taken to synchronize the cpu view of the buffer and the device view of the buffer. (For example, the cpu may cache the content of the DMA buffer and then the device modified its content, resulting in stale data in the cpu (user) view) To do this, the user can use `DMA_BUF_IOCTL_SYNC` call of the DMA buffer to synchronize the different views of the buffer before and after accessing it.

When the device is done with the shared buffer, it is important to call the functions `dma_buf_unmap_attachment`, `dma_buf_detach`, and `dma_buf_put` to perform the appropriate clean up.

In the case of sharing DMA buffer with the kgsl driver, the `sg_table` that belongs to the DMA buffer will be stored in the `kgsl_mem_entry` as the field `sgt`:

```
static int kgsl_setup_dma_buf(struct kgsl_device *device,
                                struct kgsl_pagetable *pagetable,
                                struct kgsl_mem_entry *entry,
                                struct dma_buf *dmabuf)
{
    ...
        sg_table = dma_buf_map_attachment(attach, DMA_TO_DEVICE);
    ...
        meta->table = sg_table;
        entry->priv_data = meta;
        entry->memdesc.sgt = sg_table;
```

On the other hand, in the case of a `MAP_USER_MEM` type memory object, the `sg_table` in `memdesc.sgt` is created and owned by the `kgsl_mem_entry`:

```
static int memdesc_sg_virt(struct kgsl_memdesc *memdesc, struct file *vmfile)
{
    ...
    //Creates an sg_table and stores it in memdesc->sgt
        ret = sg_alloc_table_from_pages(memdesc->sgt, pages, npages,
                                        0, memdesc->size, GFP_KERNEL);
```

As such, care must be taken with the ownership of `memdesc->sgt` when `kgsl_mem_entry` is destroyed. If the ioctl call somehow failed, then the memory object that is created will have to be destroyed. Depending on the type of the memory, the clean up logic will be different:

```
unmap:
        if (param->type == KGSL_USER_MEM_TYPE_DMABUF) {
                kgsl_destroy_ion(entry->priv_data);
                entry->memdesc.sgt = NULL;
        }

        kgsl_sharedmem_free(&entry->memdesc);
```

If we created an ION type memory object, then apart from the extra clean up that detaches the gpu from the DMA buffer, `entry->memdesc.sgt` is set to `NULL` before entering `kgsl_sharedmem_free`, which will free `entry->memdesc.sgt`:

```
void kgsl_sharedmem_free(struct kgsl_memdesc *memdesc)
{
    ...
        if (memdesc->sgt) {
                sg_free_table(memdesc->sgt);
                kfree(memdesc->sgt);
        }

        if (memdesc->pages)
                kgsl_free(memdesc->pages);
}
```

So far, so good, everything is taken care of, but a closer look reveals that, when creating a `KGSL_USER_MEM_TYPE_ADDR` object, the code would first check if the user supplied address is allocated by the ion allocator, if so, it will create an ION type memory object instead.

```
static int kgsl_setup_useraddr(struct kgsl_device *device,
                struct kgsl_pagetable *pagetable,
                struct kgsl_mem_entry *entry,
                unsigned long hostptr, size_t offset, size_t size)
{
    ...
        /* Try to set up a dmabuf - if it returns -ENODEV assume anonymous */
        ret = kgsl_setup_dmabuf_useraddr(device, pagetable, entry, hostptr);
        if (ret != -ENODEV)
                return ret;

        /* Okay - lets go legacy */
        return kgsl_setup_anon_useraddr(pagetable, entry,
                hostptr, offset, size);
}
```

While there is nothing wrong with using a DMA mapping when the user supplied memory is actually a dma buffer (allocated by ion), if something goes wrong during the ioctl call, the clean up logic will be wrong and `memdesc->sgt` will be incorrectly deleted. Fortunately, before the ION ABI change introduced in the 4.12 kernel, the now freed `sg_table` cannot be reached again. However, after this change, the `sg_table` gets added to the `dma_buf_attachment` when a DMA buffer is attached to a device, and the `dma_buf_attachment` is then stored in the DMA buffer.

```
static int ion_dma_buf_attach(struct dma_buf *dmabuf, struct device *dev,
                                struct dma_buf_attachment *attachment)
```

```
{
    ...
        table = dup_sg_table(buffer->sg_table);
    ...
        a->table = table;                          //<---- c. duplicated table stored in attachment, which is the output of dma_buf_attach in
    ...
        mutex_lock(&buffer->lock);
        list_add(&a->list, &buffer->attachments);  //<---- d. attachment got added to dma_buf::attachments
        mutex_unlock(&buffer->lock);
        return 0;
}
```

This will normally be removed when the DMA buffer is detached from the device. However, because of the wrong clean up logic, the DMA buffer will never be detached in this case, (`kgsl_destroy_ion` is not called) meaning that after the ioctl call failed, the user supplied DMA buffer will end up with an attachment that contains a free'd `sg_table`. This `sg_table` will then be used any time when the `DMA_BUF_IOCTL_SYNC` call is used on the buffer:

```
static int __ion_dma_buf_begin_cpu_access(struct dma_buf *dmabuf,
                                          enum dma_data_direction direction,
                                          bool sync_only_mapped)
{
    ...
        list_for_each_entry(a, &buffer->attachments, list) {
        ...
                if (sync_only_mapped)
                        tmp = ion_sgl_sync_mapped(a->dev, a->table->sgl,        //<--- use-after-free of a->table
                                                  a->table->nents,
                                                  &buffer->vmas,
                                                  direction, true);
                else
                        dma_sync_sg_for_cpu(a->dev, a->table->sgl,              //<--- use-after-free of a->table
                                            a->table->nents, direction);
            ...
                }
        }
...
}
```

There are actually multiple paths in this ioctl that can lead to the use of the `sg_table` in different ways.

Getting a free'd object with a fake out-of-memory error

While this looks like a very good use-after-free that allows me to hold onto a free'd object and use it at any convenient time, as well as in different ways, to trigger it, I first need to cause the `IOCTL_KGSL_GPUOBJ_IMPORT` or `IOCTL_KGSL_MAP_USER_MEM` to fail and to fail at the right place. The only place where a use-after-free can happen in the `IOCTL_KGSL_GPUOBJ_IMPORT` call is when it fails at `kgsl_mem_entry_attach_process`:

```
long kgsl_ioctl_gpuobj_import(struct kgsl_device_private *dev_priv,
                unsigned int cmd, void *data)
{
    ...
        kgsl_memdesc_init(dev_priv->device, &entry->memdesc, param->flags);
        if (param->type == KGSL_USER_MEM_TYPE_ADDR)
                ret = _gpuobj_map_useraddr(dev_priv->device, private->pagetable,
                        entry, param);
        else if (param->type == KGSL_USER_MEM_TYPE_DMABUF)
                ret = _gpuobj_map_dma_buf(dev_priv->device, private->pagetable,
                        entry, param, &fd);
        else
                ret = -ENOTSUPP;

        if (ret)
                goto out;

    ...
        ret = kgsl_mem_entry_attach_process(dev_priv->device, private, entry);
        if (ret)
                goto unmap;
```

This is the last point where the call can fail. Any earlier failure will also not result in `kgsl_sharedmem_free` being called. One way that this can fail is if `kgsl_mem_entry_track_gpuaddr` failed to reserve memory in the gpu due to out-of-memory error:

```
static int kgsl_mem_entry_attach_process(struct kgsl_device *device,
                struct kgsl_process_private *process,
                struct kgsl_mem_entry *entry)
{
    ...
        ret = kgsl_mem_entry_track_gpuaddr(device, process, entry);
        if (ret) {
                kgsl_process_private_put(process);
                return ret;
        }
```

Of course, to actually cause an out-of-memory error would be rather difficult and unreliable, as well as risking to crash the device by exhausting the memory.

If we look at how a user provided address is mapped to gpu address in `kgsl_iommu_get_gpuaddr`, (which is called by `kgsl_mem_entry_track_gpuaddr`, note that these are actually user space gpu address in the sense that they are used by the gpu with a user process specific pagetable to resolve the actual addresses, so different processes can have the same gpu addresses that resolved to different actual locations, in the same way that user space addresses can be the same in different processes but resolved to different locations) then we see that an alignment parameter is taken from the `flags` of the `kgsl_memdesc`:

```
static int kgsl_iommu_get_gpuaddr(struct kgsl_pagetable *pagetable,
                struct kgsl_memdesc *memdesc)
{
    ...
        unsigned int align;
    ...
    //Uses `memdesc->flags` to compute the alignment parameter
        align = max_t(uint64_t, 1 << kgsl_memdesc_get_align(memdesc),
                        memdesc->pad_to);
...
```

and the flags of `memdesc` is taken from the `flags` parameter when the ioctl is called:

```
long kgsl_ioctl_gpuobj_import(struct kgsl_device_private *dev_priv,
                unsigned int cmd, void *data)
{
    ...
        kgsl_memdesc_init(dev_priv->device, &entry->memdesc, param->flags);
```

When mapping memory to the gpu, this `align` value will be used to ensure that the memory address is mapped to a value that is aligned (i.e. multiples of) to `align`. In particular, the gpu address will be the next multiple of `align` that is not already occupied. If no such value exist, then an out-of-memory error will occur. So by using a large `align` value in the ioctl call, I can easily use up all the addresses that are aligned with the value that I specified. For example, if I set `align` to be `1 << 31`, then there will only be two addresses that aligns with `align` (`0` and `1 << 31`). So after just mapping one memory object (which can be as small as `4096` bytes), I'll get an out-of-memory error the next time I use the ioctl call. This will then give me a free'd `sg_table` in the DMA buffer. By allocating another object of similar size in the kernel, I can then replace this `sg_table` with an object that I control. I'll go through the details of how to do this later, but for now, let's assume I am able to do this and have complete control of all the fields in this `sg_table` and see what this bug potentially allows me to do.

The primitives of the vulnerability

As mentioned before, there are different ways to use the free'd `sg_table` via the `DMA_BUF_IOCTL_SYNC` ioctl call:

```
static long dma_buf_ioctl(struct file *file,
                          unsigned int cmd, unsigned long arg)
{
    ...
        switch (cmd) {
        case DMA_BUF_IOCTL_SYNC:
        ...
                if (sync.flags & DMA_BUF_SYNC_END)
                        if (sync.flags & DMA_BUF_SYNC_USER_MAPPED)
                                ret = dma_buf_end_cpu_access_umapped(dmabuf,
                                                                     dir);
                        else
                                ret = dma_buf_end_cpu_access(dmabuf, dir);
                else
                        if (sync.flags & DMA_BUF_SYNC_USER_MAPPED)
                                ret = dma_buf_begin_cpu_access_umapped(dmabuf,
                                                                       dir);
                        else
                                ret = dma_buf_begin_cpu_access(dmabuf, dir);

                return ret;
```

These will ended up calling the functions `__ion_dma_buf_begin_cpu_access` or `__ion_dma_buf_end_cpu_access` that provide the concrete implemenations.

As explained before, the `DMA_BUF_IOCTL_SYNC` call is meant to synchronize the cpu view of the DMA buffer and the device (in this case, gpu) view of the DMA buffer. For the kgsl device, the synchronization is implemented in `lib/swiotlb.c`. The various different ways of syncing the buffer will more or less follow a code path like this:

1. The `scatterlist` in the free'd `sg_table` is iterated in a loop;
2. In each iteration, the `dma_address` and `dma_length` of the `scatterlist` is used to identify the location and size of the memory for synchronization.
3. The function `swiotlb_sync_single` is called to perform the actual synchronization of the memory.

So what does `swiotlb_sync_single` do? It first checks whether the `dma_address` (`dev_addr`, `dma_to_phys` for `kgsl` is just the identity function) in the `scatterlist` is an address of a `swiotlb_buffer` using the `is_swiotlb_buffer` function, if so, it calls `swiotlb_tlb_sync_single`, otherwise, it will call `dma_mark_clean`.

```
static void
swiotlb_sync_single(struct device *hwdev, dma_addr_t dev_addr,
                    size_t size, enum dma_data_direction dir,
                    enum dma_sync_target target)
{
        phys_addr_t paddr = dma_to_phys(hwdev, dev_addr);

        BUG_ON(dir == DMA_NONE);

        if (is_swiotlb_buffer(paddr)) {
                swiotlb_tbl_sync_single(hwdev, paddr, size, dir, target);
                return;
        }

        if (dir != DMA_FROM_DEVICE)
                return;

        dma_mark_clean(phys_to_virt(paddr), size);
}
```

The function `dma_mark_clean` simply flushes the cpu cache that corresponds to `dev_addr` and keeps the cpu cache in sync with the actual memory. I wasn't able to exploit this path and so I'll concentrate on the `swiotlb_tbl_sync_single` path.

```
void swiotlb_tbl_sync_single(struct device *hwdev, phys_addr_t tlb_addr,
                             size_t size, enum dma_data_direction dir,
                             enum dma_sync_target target)
{
        int index = (tlb_addr - io_tlb_start) >> IO_TLB_SHIFT;
        phys_addr_t orig_addr = io_tlb_orig_addr[index];
        if (orig_addr == INVALID_PHYS_ADDR)                            //<--------- a. checks address valid
                return;
        orig_addr += (unsigned long)tlb_addr & ((1 << IO_TLB_SHIFT) - 1);

        switch (target) {
        case SYNC_FOR_CPU:
                if (likely(dir == DMA_FROM_DEVICE || dir == DMA_BIDIRECTIONAL))
                        swiotlb_bounce(orig_addr, tlb_addr,
                                       size, DMA_FROM_DEVICE);
    ...
}
```

After a further check of the address (`tlb_addr`) against an array `io_tlb_orig_addr`, the function `swiotlb_bounce` is called.

```
static void swiotlb_bounce(phys_addr_t orig_addr, phys_addr_t tlb_addr,
                           size_t size, enum dma_data_direction dir)
{
    ...
        unsigned char *vaddr = phys_to_virt(tlb_addr);
        if (PageHighMem(pfn_to_page(pfn))) {
        ...

                while (size) {
                        sz = min_t(size_t, PAGE_SIZE - offset, size);

                        local_irq_save(flags);
                        buffer = kmap_atomic(pfn_to_page(pfn));
                        if (dir == DMA_TO_DEVICE)
                                memcpy(vaddr, buffer + offset, sz);
                        else
                                memcpy(buffer + offset, vaddr, sz);
            ...
                }
        } else if (dir == DMA_TO_DEVICE) {
                memcpy(vaddr, phys_to_virt(orig_addr), size);
        } else {
                memcpy(phys_to_virt(orig_addr), vaddr, size);
        }
}
```

As `tlb_addr` and `size` comes from a `scatterlist` in the free'd `sg_table`, it becomes clear that I may be able to call a `memcpy` with a partially controlled source/destination (`tlb_addr` comes from `scatterlist` but is constrained as it needs to pass some checks, while `size` is unchecked). This could potentially give me a very strong relative read/write primitive. The questions are:

1. What is the `swiotlb_buffer` and is it possible to pass the `is_swiotlb_buffer` check without a seperate info leak?
2. What is the `io_tlb_orig_addr` and how to pass that test?
3. How much control do I have with the `orig_addr`, which comes from `io_tlb_orig_addr`?

The Software Input Output Translation Lookaside Buffer

The Software Input Output Translation Lookaside Buffer (SWIOTLB), sometimes known as the bounce buffer, is a memory region with physical address smaller than 32 bits. It seems to be very rarely used in modern Android phones and as far as I can gather, there are two main uses of it:

1. It is used when a DMA buffer that has a physical address higher than 32 bits is attached to a device that can only access 32 bit addresses. In this case, the SWIOTLB is used as a proxy of the DMA buffer to allow access of it from the device. This is the code path that we have been looking at. As this would mean an extra read/write operation between the DMA buffer and the SWIOTLB every time a synchronization between the device and DMA buffer happens, it is not an ideal scenario but is rather only used as a last resort.
2. To use as a layer of protection to avoid untrusted usb devices from accessing DMA memory directly (See here)

As the second usage is likely to involve plugging a usb device to a phone and thus requires physical access. I'll only cover the first usage here, which will also answer the three questions in the previous section.

To begin with, let's take a look at the location of the SWIOTLB. This is used by the check `is_swiotlb_buffer` to determine whether a physical address belongs to the SWIOTLB:

```
int is_swiotlb_buffer(phys_addr_t paddr)
{
        return paddr >= io_tlb_start && paddr < io_tlb_end;
}
```

The global variables `io_tlb_start` and `io_tlb_end` marks the range of the SWIOTLB. As mentioned before, the SWIOTLB needs to be an address smaller than 32 bits. How does the kernel guarantee this? By allocating the SWIOTLB very early during boot. From a rooted device, we can see that the SWIOTLB is allocated nearly right at the start of the boot. This is an excerpt of the kernel log during the early stage of booting a Pixel 4:

```
...
[    0.000000] c0      0 software IO TLB: swiotlb init: 00000000f3800000
[    0.000000] c0      0 software IO TLB: mapped [mem 0xf3800000-0xf3c00000] (4MB)
...
```

Here we see that `io_tlb_start` is `0xf3800000` while `io_tlb_end` is `0xf3c00000`.

While allocating the SWIOTLB early makes sure that the its address is below 32 bits, it also makes it predictable. In fact, the address only seems to depend on the amount of memory configured for the SWIOTLB, which is passed as the `swiotlb` boot parameter. For Pixel 4, this is `swiotlb=2048` (which seems to be a common parameter and is the same for Galaxy S10 and S20) and will allocate 4MB of SWIOTLB (`allocation size = swiotlb * 2048`) For the Samsung Galaxy A71, the parameter is set to `swiotlb=1`, which allocates the minimum amount of SWIOTLB (`0x40000` bytes)

```
[    0.000000] software IO TLB: mapped [mem 0xfffbf000-0xfffff000] (0MB)
```

The SWIOTLB will be at the same location when changing `swiotlb` to `1` on Pixel 4.

This provides us with a predicable location for the SWIOTLB to pass the `is_swiotlb_buffer` test.

Let’s take a look at `io_tlb_orig_addr` next. This is an array used for storing addresses of DMA buffers that are attached to devices with addresses that are too high for the device to access:

```
int
swiotlb_map_sg_attrs(struct device *hwdev, struct scatterlist *sgl, int nelems,
                     enum dma_data_direction dir, unsigned long attrs)
{
    ...
        for_each_sg(sgl, sg, nelems, i) {
                phys_addr_t paddr = sg_phys(sg);
                dma_addr_t dev_addr = phys_to_dma(hwdev, paddr);

                if (swiotlb_force == SWIOTLB_FORCE ||
                    !dma_capable(hwdev, dev_addr, sg->length)) {
            //device cannot access dev_addr, so use SWIOTLB as a proxy
                        phys_addr_t map = map_single(hwdev, sg_phys(sg),
                                                     sg->length, dir, attrs);
             ...
}
```

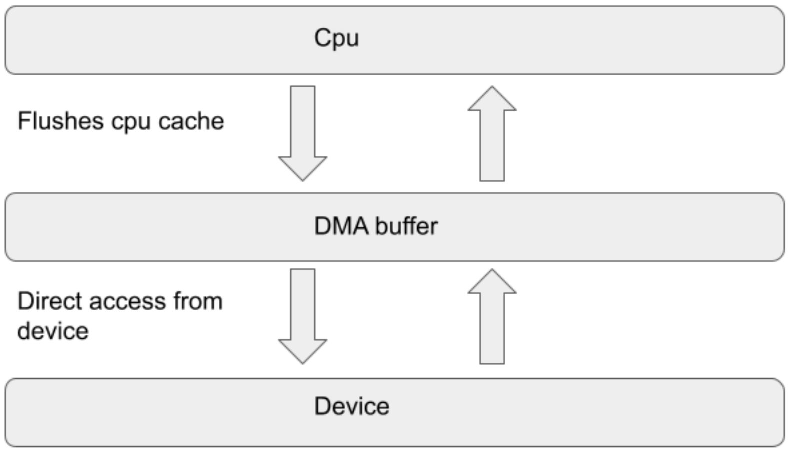
In this case, `map_single` will store the address of the DMA buffer (`dev_addr`) in the `io_tlb_orig_addr`. This means that if I can cause a SWIOTLB mapping to happen by attaching a DMA buffer with high address to a device that cannot access it (`!dma_capable`), then the `orig_addr` in `memcpy` of `swiotlb_bounce` will point to a DMA buffer that I control, which means I can read and write its content with complete control.

```
static void swiotlb_bounce(phys_addr_t orig_addr, phys_addr_t tlb_addr,
                           size_t size, enum dma_data_direction dir)
{
    ...
    //orig_addr is the address of a DMA buffer uses the SWIOTLB mapping
        } else if (dir == DMA_TO_DEVICE) {
                memcpy(vaddr, phys_to_virt(orig_addr), size);
        } else {
                memcpy(phys_to_virt(orig_addr), vaddr, size);
        }
}
```

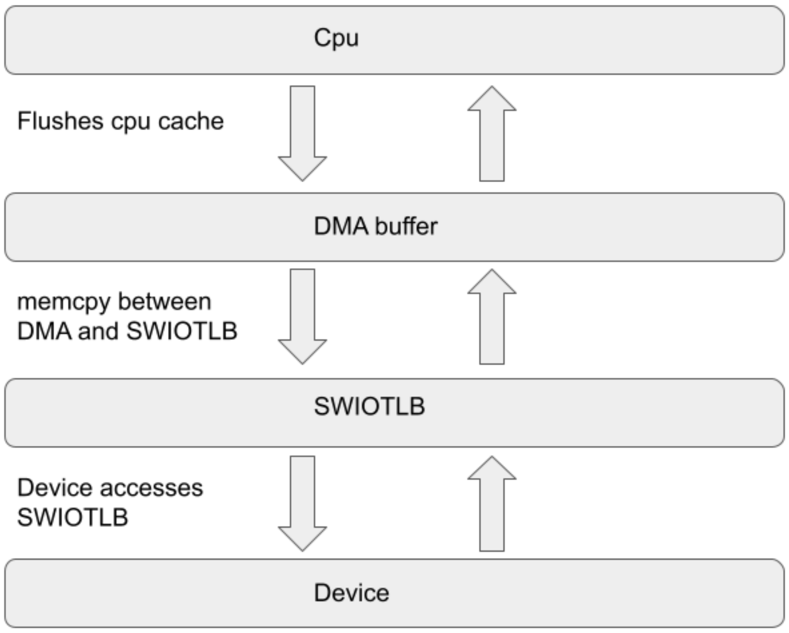
It now becomes clear that, if I can allocate a SWIOTLB, then I will be able to perform both read and write of a region behind the SWIOTLB region with arbitrary size (and completely controlled content in the case of write). In what follows, this is what I’m going to use for the exploit.

To summarize, this is how synchronization works for DMA buffer shared with the implementation in `/lib/swiotlb.c`.

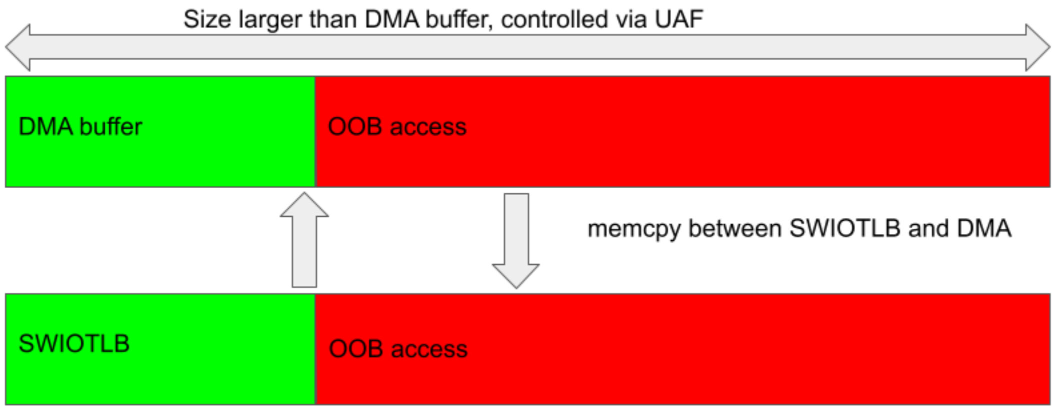
When the device is capable of accessing the DMA buffer’s address, synchronization will involve flushing the cpu cache:



When the device cannot access the DMA buffer directly, a SWIOTLB is created as an intermediate buffer to allow device access. In this case, the `io_tlb_orig_addr` array is served as a look up table to locate the DMA buffer from the SWIOTLB.



In the use-after-free scenario, I can control the size of the `memcpy` between the DMA buffer and SWIOTLB in the above figure and that turns into a read/write primitive:


Size larger than DMA buffer, controlled via UAF
DMA buffer
OOB access
memcpy between SWIOTLB and DMA
SWIOTLB
OOB access

Provided I can control the `scatterlist` that specifies the location and size of the SWIOTLB, I can specify the size to be larger than the original DMA buffer to cause an out-of-bounds access (I still need to point to the SWIOTLB to pass the checks). Of course, it is no good to just cause out-of-bounds access, I need to be able to read back the out-of-bounds data in the case of a read access and control the data that I write in the case of a write access. This issue will be addressed in the next section.

Allocating a Software Input Output Translation Lookaside Buffer

As it turns out, the SWIOTLB is actually very rarely used. For one or another reason, either because most devices are capable of reading 64 bit addresses, or that the DMA buffer synchronization is implemented with `arm_smmu` rather than `swiotlb`, I only managed to allocate a SWIOTLB using the `adsprpc` driver. The `adsprpc` driver is used for communicating with the DSP (digital signal processor), which is a seperate processor on Qualcomm's snapdragon chipset. The DSP and the `adsprpc` itself is a very vast topic that had many security implications, and it is out of the scope of this post.

Roughly speaking, the DSP is a specialized chip that is optimized for certain computationally intensive tasks such as image, video, audio processing and machine learning. The cpu can offload these tasks to the DSP to improve overall performance. However, as the DSP is a different processor altogether, an RPC mechanism is needed to pass data and instructions between the cpu and the DSP. This is what the `adsprpc` driver is for. It allows the kernel to communicate with the DSP (which is running on a separate kernel and OS altogether, so this is truly "remote") to invoke functions, allocate memory and retrieve results from the DSP.

While access to the `adsprpc` driver from third-party apps is not granted in the default SELinux settings and as such, I'm unable to use it on Google's Pixel phones, it is still enabled on many different phones running Qualcomm's snapdragon SoC (system on chip). For example, Samsung phones allow accesses of `adsprpc` from third party Apps, which allows the exploit in this post to be launched directly from a third party App or from a compromised beta version of Chrome (or any other compromised App). On phones which `adsprpc` accesses is not allowed, such as the Pixel 4, an additional bug that compromises a service that can access `adsprpc` is required to launch this exploit. There are various services that can access the `adsprpc` driver and reachable directly from third party Apps, such as the `hal_neuralnetworks`, which is implemented as a closed source service in `android.hardware.neuralnetworks@1.x-service-qti`. I did not investigate this path, so I'll assume Samsung phones in the rest of this post.

With `adsprpc`, the most obvious ioctl to use for allocating SWIOTLB is the `FASTRPC_IOCTL_MMAP`, which calls `fastrpc_mmap_create` to attach a DMA buffer that I supplied:

```
static int fastrpc_mmap_create(struct fastrpc_file *fl, int fd,
        unsigned int attr, uintptr_t va, size_t len, int mflags,
        struct fastrpc_mmap **ppmap)
{
    ...
        } else if (mflags == FASTRPC_DMAHANDLE_NOMAP) {
                VERIFY(err, !IS_ERR_OR_NULL(map->buf = dma_buf_get(fd)));
                if (err)
                        goto bail;
                VERIFY(err, !dma_buf_get_flags(map->buf, &flags));
        ...
                map->attach->dma_map_attrs |= DMA_ATTR_SKIP_CPU_SYNC;
        ...
```

However, the call seems to always fail when `fastrpc_mmap_on_dsp` is called, which will then detach the DMA buffer from the `adsprpc` driver and remove the SWIOTLB that was just allocated. While it is possible to work with a temporary buffer like this by racing with multiple threads, it would be better if I can allocate a permanent SWIOTLB.

Another possibility is to use the `get_args` function, which will also invoke `fastrpc_mmap_create`:

```
static int get_args(uint32_t kernel, struct smq_invoke_ctx *ctx)
{
    ...
        for (i = bufs; i < bufs + handles; i++) {
        ...
                if (ctx->attrs && (ctx->attrs[i] & FASTRPC_ATTR_NOMAP))
                        dmaflags = FASTRPC_DMAHANDLE_NOMAP;
                VERIFY(err, !fastrpc_mmap_create(ctx->fl, ctx->fds[i],
                                FASTRPC_ATTR_NOVA, 0, 0, dmaflags,
                                &ctx->maps[i]));
        ...
        }
```

The `get_args` function is used in the various `FASTRPC_IOCTL_INVOKE_*` calls for passing arguments to invoke functions on the DSP. Under normal circumstances, a corresponding `put_args` will be called to detach the DMA buffer from the `adsprpc` driver. However, if the remote invocation failed, the call to `put_args` will be skipped and the clean up will be deferred to the time when the `adsprpc` file is close:

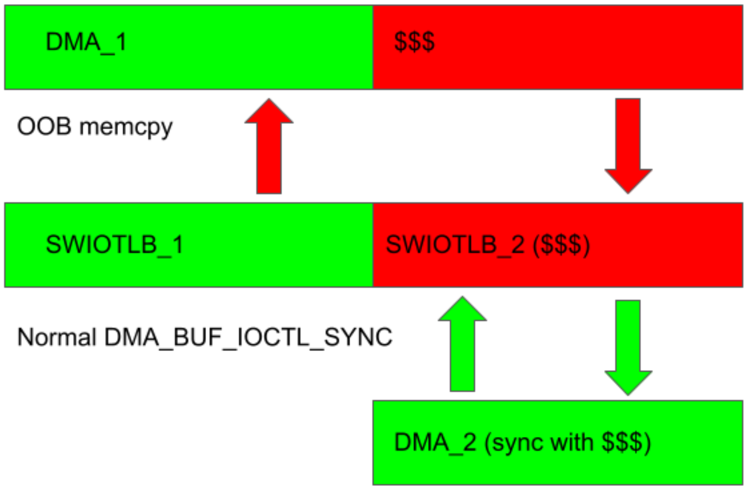
```
static int fastrpc_internal_invoke(struct fastrpc_file *fl, uint32_t mode,
                                   uint32_t kernel,
                                   struct fastrpc_ioctl_invoke_crc *inv)
{
    ...
        if (REMOTE_SCALARS_LENGTH(ctx->sc)) {
                PERF(fl->profile, GET_COUNTER(perf_counter, PERF_GETARGS),
                VERIFY(err, 0 == get_args(kernel, ctx));                  //<----- get_args
                PERF_END);
                if (err)
                        goto bail;
        }
    ...
 wait:
        if (kernel) {
      ....
        } else {
                interrupted = wait_for_completion_interruptible(&ctx->work);
                VERIFY(err, 0 == (err = interrupted));
                if (err)
                        goto bail;                                        //<----- invocation failed and jump to bail directly
        }
    ...
        VERIFY(err, 0 == put_args(kernel, ctx, invoke->pra));    //<------ detach the arguments
        PERF_END);
    ...
 bail:
    ...
        return err;
}
```

So by using `FASTRPC_IOCTL_INVOKE_*` with an invalid remote function, it is easy to allocate and keep the SWIOTLB alive until I choose to close the `/dev/adsprpc-smd` file that is used to make the ioctl call. This is the only part that the `adsprpc` driver is needed and we're now set up to start writing the exploit.

Now that I can allocate SWIOTLB that maps to DMA buffers that I created, I can do the following to exploit the out-of-bounds read/write primitive from the previous section.

1. First allocate a number of DMA buffers. By manipulating the ion heap, (which I'll go through later in this post), I can place some useful data behind one of these DMA buffers. I will call this buffer `DMA_1`.
2. Use the `adsprpc` driver to allocate SWIOTLB buffers associated with these DMA buffers. I'll arrange it so that the `DMA_1` occupies the first SWIOTLB (which means all other SWIOTLB will be allocated behind it), call this `SWIOTLB_1`. This can be done easily as SWIOTLB are simply allocated as a contiguous array.
3. Use the read/write primitive in the previous section to trigger out-of-bounds read/write on `DMA_1`. This will either write the memory behind `DMA_1` to the SWIOTLB behind `SWIOTLB_1`, or vice versa.
4. As the SWIOTLB behind `SWIOTLB_1` are mapped to the other DMA buffers that I controlled, I can use the `DMA_BUF_IOCTL_SYNC` ioctl of these DMA buffers to either read data from these SWIOTLB or write data to them. This translates into arbitrary read/write of memory behind `DMA_1`.

The following figure illustrates this with a simplified case of two DMA buffers.



Replacing the `sg_table`

So far, I planned an exploitation strategy based on the assumption that I already have control of the `scatterlist sgl` of the free'd `sg_table`. In order to actually gain control of it, I need to replace the free'd `sg_table` with a suitable object. This turns out to be the most complicated part of the exploit. While there are well-known kernel heap spraying techniques that allows us to replace a free'd object with controlled data (for example the [`sendmsg`] and [`setxattr`]), they cannot be applied directly here as the `sgl` of the free'd `sg_table` here needs to be a valid pointer that points to controlled data. Without a way to leak a heap address, I'll not be able to use these heap spraying techniques to construct a valid object. With this bug alone, there is almost no hope of getting an info leak at this stage. The other alternative is to look for other suitable objects to replace the `sg_table`. A CodeQL query can be used to help looking for suitable objects:

```
from FunctionCall fc, Type t, Variable v, Field f, Type t2
where (fc.getTarget().hasName("kmalloc") or
       fc.getTarget().hasName("kzalloc") or
       fc.getTarget().hasName("kcalloc"))
      and
      exists(Assignment assign | assign.getRValue() = fc and
             assign.getLValue() = v.getAnAccess() and
             v.getType().(PointerType).refersToDirectly(t)) and
      t.getSize() < 128 and t.fromSource() and
      f.getDeclaringType() = t and
      (f.getType().(PointerType).refersTo(t2) and t2.getSize() <= 8) and
      f.getByteOffset() = 0
select fc, t, fc.getLocation()
```

In this query, I look for objects created via `kmalloc`, `kzalloc` or `kcalloc` that are of size smaller than 128 (same bucket as `sg_table`) and have a pointer field as the first field. However, I wasn't able to find a suitable object, although `filename` allocated in [`getname_flags`] came close:

```
struct filename *
getname_flags(const char __user *filename, int flags, int *empty)
{
        struct filename *result;
    ...
        if (unlikely(len == EMBEDDED_NAME_MAX)) {
        ...
                result = kzalloc(size, GFP_KERNEL);
                if (unlikely(!result)) {
                        __putname(kname);
                        return ERR_PTR(-ENOMEM);
                }
                result->name = kname;
                len = strncpy_from_user(kname, filename, PATH_MAX);
        ...
```

with `name` points to a user controlled string and can be reached using, for example, the `mknod` syscall. However, not being able to use null character turns out to be too much of a restriction here.

Just-in-time object replacement

Let's take a look at how the free'd `sg_table` is used, say, in `__ion_dma_buf_begin_cpu_access`, it seems that at some point in the execution, the `sgl` field is taken from the `sgl_table`, and the `sgl_table` itself will no longer be used, but only the cached pointer value of `sgl` is used:

```
static int __ion_dma_buf_begin_cpu_access(struct dma_buf *dmabuf,
                                          enum dma_data_direction direction,
                                          bool sync_only_mapped)
{

                if (sync_only_mapped)
                        tmp = ion_sgl_sync_mapped(a->dev, a->table->sgl,    //<------- `sgl` got passed, and `table` never used again
                                                  a->table->nents,
                                                  &buffer->vmas,
                                                  direction, true);
                else
                        dma_sync_sg_for_cpu(a->dev, a->table->sgl,
                                            a->table->nents, direction);
```

While source code could be misleading as auto function inlining is common in kernel code (in fact `ion_sgl_sync_mapped` is inlined), the bottom line is that, at some point, the value of `sgl` will be cached in registry and the state of the original `sg_table` will not affect the code path anymore. So if I am able to first replace `a->table` with another `sg_table`, then deleting this new `sg_table` using [`sg_free_table`] will also cause the `sgl` to be deleted, but of course, there will be clean up logic that sets `sgl` to `NULL`. But what if I set up another thread to delete this new `sg_table` after `sgl` had already been cached in the registry? Then it won't matter if `sgl` is set to `NULL`, because the value in the registry will still be pointing to the original `scatterlist`, and as this `scatterlist` is now free'd, this means I will now get a use-after-free of `sgl` directly in `ion_sgl_sync_mapped`. I can then use the [`sendmsg`] to replace it with controlled data. There is one major problem with this though, as the time between `sgl` being cached in registry and the time where it is used again is very short, it is normally not possible to fit the entire `sg_table` replacement sequence within such a short time frame, even if I race the slowest cpu core against the fastest.

To resolve this, I'll use a technique by Jann Horn in [Exploiting race conditions on [ancient] Linux], which turns out to still work like a charm on modern Android.

To ensure that each task(thread or process) has a fair share of the cpu time, the linux kernel scheduler can interrupt a running task and put it on hold, so that another task can be run. This kind of interruption and stopping of a task is called preemption (where the interrupted task is preempted). A task can also put itself on hold to allow other task to run, such as when it is waiting for some I/O input, or when it calls `sched_yield()`. In this case, we say that the task is voluntarily preempted. Preemption can happen inside syscalls such as ioctl calls as well, and on Android, tasks can be preempted except in some critical regions (e.g. holding a spinlock). This means that a thread running the `DMA_BUF_IOCTL_SYNC` ioctl call can be interrupted after the `sgl` field is cached in the registry and be put on hold. The default behavior, however, will not normally give us much control over when the preemption happens, nor how long the task is put on hold.

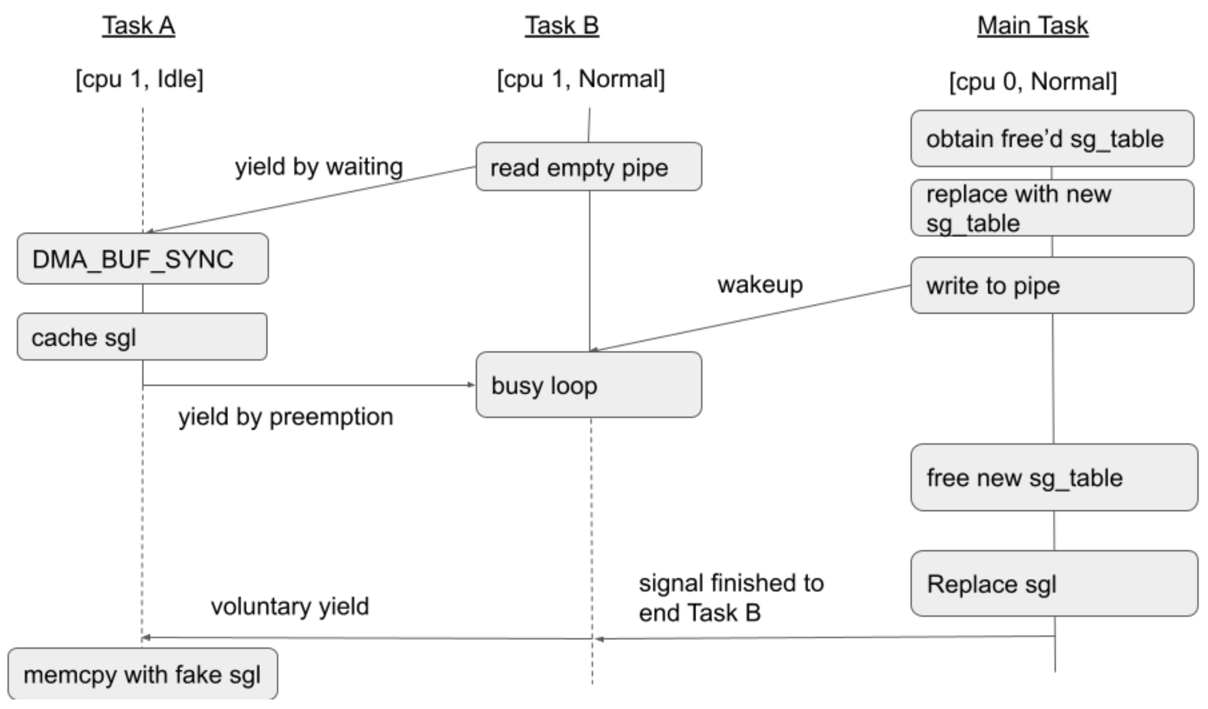
To gain better control in both these areas, cpu affinity and task priorities can be used. By default, a task is run with the priority `SCHED_NORMAL`, but a lower priority `SCHED_IDLE`, can also be set using the `sched_setscheduler` call (or `pthread_setschedparam` for threads). Furthermore, it can also be pinned to a cpu with `sched_setaffinity`, which would only allow it to run on a specific cpu. By pinning two tasks, one with `SCHED_NORMAL` priority and the other with `SCHED_IDLE` priority to the same cpu, it is possible to control the timing of the preemption as follows.

1. First have the `SCHED_NORMAL` task perform a syscall that would cause it to pause and wait. For example, it can read from a pipe with no data coming in from the other end, then it would wait for more data and voluntarily preempt itself, so that the `SCHED_IDLE` task can run;
2. As the `SCHED_IDLE` task is running, send some data to the pipe that the `SCHED_NORMAL` task had been waiting on. This will wake up the `SCHED_NORMAL` task and cause it to preempt the `SCHED_IDLE` task, and because of the task priority, the `SCHED_IDLE` task will be preempted and put on hold.
3. The `SCHED_NORMAL` task can then run a busy loop to keep the `SCHED_IDLE` task from waking up.

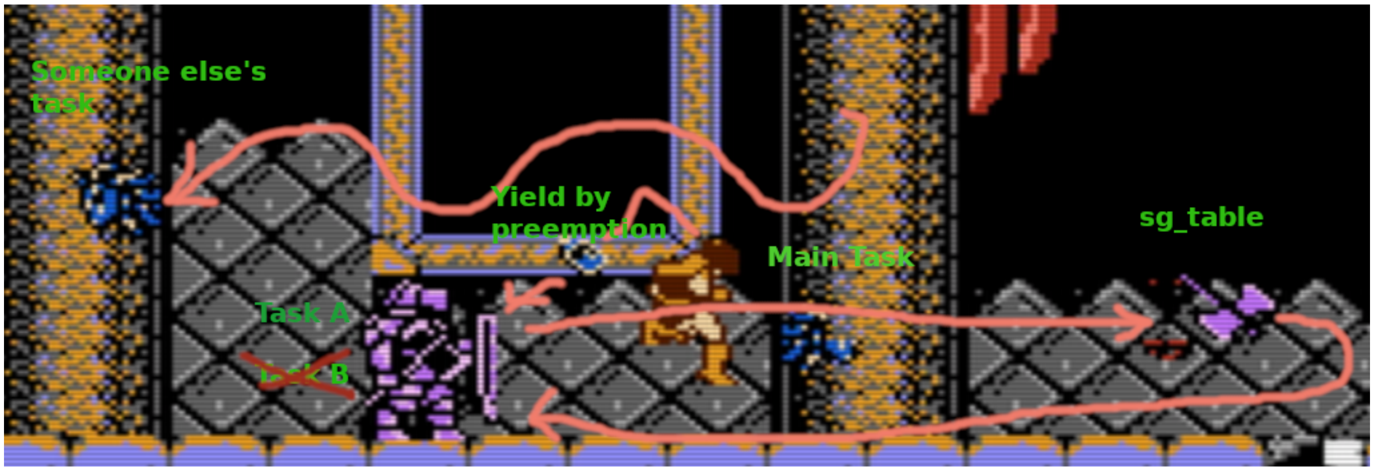
In our case, the object replacement sequence goes as follows:

1. Obtain a free'd `sg_table` in a DMA buffer using the method in the section [Getting a free'd object with a fake out-of-memory error].
2. First replace this free'd `sg_table` with another one that I can free easily, for example, making another call to `IOCTL_KGSL_GPUOBJ_IMPORT` will give me a handle to a `kgsl_mem_entry` object, which allocates and owns a `sg_table`. Making this call immediately after step one will ensure that the newly created `sg_table` replaces the one that was free'd in step one. To free this new `sg_table`, I can call `IOCTL_KGSL_GPUMEM_FREE_ID` with the handle of the `kgsl_mem_entry`, which will free the `kgsl_mem_entry` and in turn frees the `sg_table`. In practice, a little bit more heap manipulation is needed as `IOCTL_KGSL_GPUOBJ_IMPORT` will allocate another object of similar size before allocating a `sg_table`.
3. Set up a `SCHED_NORMAL` task on, say, `cpu_1` that is listening to an empty pipe.
4. Set up a `SCHED_IDLE` task on the same cpu and have it wait until I signal it to run `DMA_BUF_IOCTL_SYNC` on the DMA buffer that contains the `sg_table` in step two.
5. The main task signals the `SCHED_IDLE` task to run `DMA_BUF_IOCTL_SYNC`.
6. The main task waits a suitable amount of time until `sgl` is cached in registry, then send data to the pipe that the `SCHED_NORMAL` task is waiting on.
7. Once it receives data, the `SCHED_NORMAL` task goes into a busy loop to keep the `DMA_BUF_IOCTL_SYNC` task from continuing.
8. The main task then calls `IOCTL_KGSL_GPUMEM_FREE_ID` to free up the `sg_table`, which will also free the object pointed to by `sgl` that is now cached in the registry. The main task then replaces this object by controlled data using `sendmsg` heap spraying. This gives control of both `dma_address` and `dma_length` in `sgl`, which are used as arguments to `memcpy`.
9. The main task signals the `SCHED_NORMAL` task on `cpu_1` to stop so that the `DMA_BUF_IOCTL_SYNC` task can resume.

The following figure illustrates what happens in an ideal world.



The following figure illustrates what happens in the real world.



Crazy as it seems, the race can actually be won almost every time, and the same parameters that control the timing would even work on both the Galaxy A71 and Pixel 4. Even when the race failed, it does not result in a crash. It can, however, crash, if the `SCHED_IDLE` task resumes too quickly. For some reason, I only managed to hold that task for about 10-20ms, and sometimes this is not long enough for the object replacement to complete.

The ion heap

Now that I'm able to replace the `scatterlist` with controlled data and make use of the read/write primitives in the section Allocating a SWIOTLB, it is time to think about what data I can place behind the DMA buffers.

To allocate DMA buffers, I need to use the ion allocator, which will allocate from the ion heap. There are different types of ion heaps, but not all of them are suitable, because I need one that would allocate buffers with addresses greater than 32 bit. The locations of various ion heap can be seen from the kernel log during a boot, the following is from Galaxy A71:

```
[    0.626370] ION heap system created
[    0.626497] ION heap qsecom created at 0x000000009e400000 with size 2400000
[    0.626515] ION heap qsecom_ta created at 0x00000000fac00000 with size 2000000
[    0.626524] ION heap spss created at 0x00000000f4800000 with size 800000
[    0.626531] ION heap secure_display created at 0x00000000f5000000 with size 5c00000
[    0.631648] platform soc:qcom,ion:qcom,ion-heap@14: ion_secure_carveout: creating heap@0xa4000000, size 0xc00000
[    0.631655] ION heap secure_carveout created
[    0.631669] ION heap secure_heap created
[    0.634265] cleancache enabled for rbin cleancache
[    0.634512] ION heap camera_preview created at 0x00000000c2000000 with size 25800000
```

As we can see, some ion heap are created at fixed locations with fixed sizes. The addresses of these heaps are also smaller than 32 bits. However, there are other ion heaps, such as the system heap, that does not have a fixed address. These are the heaps that have addresses higher than 32 bits. For the exploit, I'll use the system heap.

DMA buffers allocated on the system heap is allocated via the `ion_system_heap_allocate` function call. It'll first try to allocate a buffer from a preallocated memory pool. If the pool is full, then it'll allocate more pages using `alloc_pages`:

```
static void *ion_page_pool_alloc_pages(struct ion_page_pool *pool)
{
        struct page *page = alloc_pages(pool->gfp_mask, pool->order);
    ...
        return page;
}
```

and recycle the pages back to the pool after the buffer is freed.

This later case is more interesting because if the memory is allocated from the initial pool, then any out-of-bounds read/write are likely to just be reading and writing other ion buffers, which is only going to be user space data. So let's take a look at `alloc_pages`.

The function `alloc_pages` allocates memory with page granularity using the buddy allocator. When using `alloc_pages`, the second parameter `order` specifies the size of the requested memory and the allocator will return a memory block consisting of `2^order` contiguous pages. In order to exploit overflow of memory allocated by the buddy allocator, (a DMA buffer from the system heap), I'll use the results from Exploiting the Linux kernel via packet sockets by Andrey Konovalov. The key point is that, while objects allocated from `kmalloc` and co. (i.e. `kmalloc`, `kzalloc` and `kcalloc`) are allocated via the slab allocator, which uses preallocated memory blocks (slabs) to allocate small objects, when the slabs run out, the slab allocator will use the buddy allocator to allocate a new slab. The output of `proc/slabinfo` gives an indication of the size of slabs in pages.

```
kmalloc-8192        1036   1036   8192    4    8 : tunables    0    0    0 : slabdata    262    262      0
...
kmalloc-128       378675 384000    128   32    1 : tunables    0    0    0 : slabdata  12000  12000      0
```

In the above, the `5th` column indicates the size of the slabs in pages. So for example, if the size 8192 bucket runs out, the slab allocator will ask the buddy allocator for a memory block of 8 pages, which is order 3 (`2^3=8`), to use as a new slab. So by exhausting the slab, I can cause a new slab to be allocated in the same region as the ion system heap, which could allow me to over read/write kernel objects allocated via `kmalloc` and co.

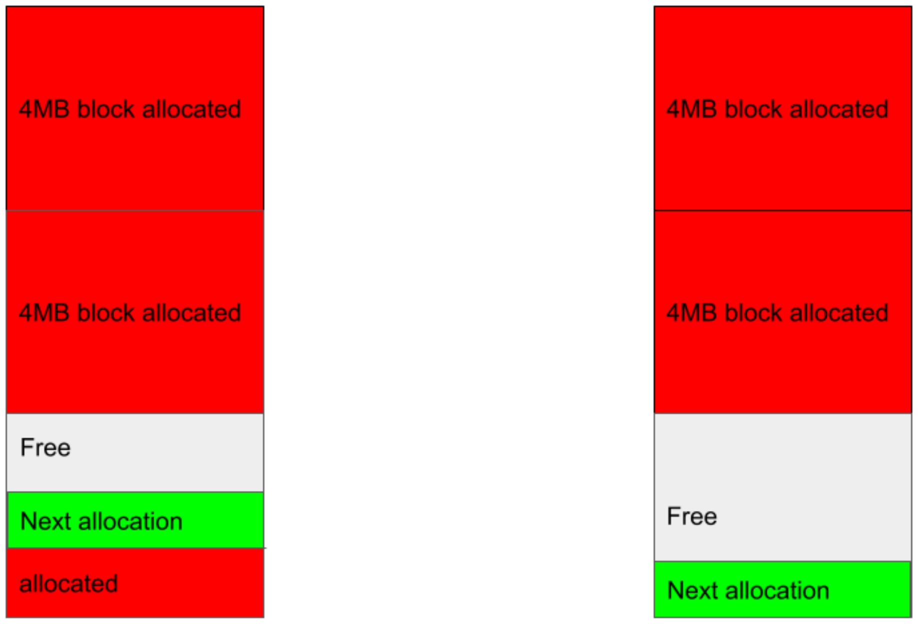
Manipulating the buddy allocator heap

As mentioned in Exploiting the Linux kernel via packet sockets, for each order, the buddy allocator maintains a freelist and use it to allocate memory of the appropriate order. When a certain order (`n`) runs out of memory, it'll try to look for free blocks in the next order up, split it in half and add it to the freelist in the requested order. If the next order is also full, it'll try to find space in the next higher up order, and so on. So by keep allocating pages of order `2^n`, eventually the freelist will be exhausted and larger blocks will be broken up, which means that consecutive allocations will be adjacent to each other.

In fact, after some experimentation on Pixel 4, it seems that after allocating a certain amount of DMA buffers from the ion system heap, the allocation will follow a very predicatble pattern.

1. The addresses of the allocated buffers are grouped in blocks of 4MB, which corresponds to order 10, the highest order block on Android.
2. Within each block, a new allocations will be adjacent to the previous one, with a higher address.
3. When a 4MB block is filled, allocations will start in the beginning of the next block, which is right below the current 4MB block.

The following figure illustrates this pattern.



So by simply creating a large amount of DMA buffers in the ion system heap, the likelihood would be that the last allocated buffer will be allocated in front of a “hole” of free memory, and the next allocation from the buddy allocator is likely to be inside this hole, provided the requested number of pages fits in this hole.

The heap spraying strategy is then very simple. First allocate a sufficient amount of DMA buffers in the ion heap to cause larger blocks to break up, then allocate a large amount of objects using `kmalloc` and co. to cause a new slab to be created. This new slab is then likely to fall into the hole behind the last allocated DMA buffer. Using the use-after-free to overflow this buffer then allows me to gain arbitrary read/write of the newly created slab.

Defeating KASLR and leaking address to DMA buffer

Initially, I was experimenting with the `binder_open` call, as it is easy to reach (just do `open("/dev/binder")`) and will allocate a `binder_proc` struct:

```
static int binder_open(struct inode *nodp, struct file *filp)
{
    ...
        proc = kzalloc(sizeof(*proc), GFP_KERNEL);
        if (proc == NULL)
```

which is of size 560 and will persist until the `/dev/binder` file is closed. So it should be relatively easy to exhaust the `kmalloc-1024` slab with this. However, after dumping the results of the out-of-bounds read, I noticed that a recurring memory pattern:

```
00011020: 68b2 8e68 c1ff ffff 08af 5109 80ff ffff  h..h......Q.....
00011030: 0000 0000 0000 0000 0100 0000 0000 0000  ................
00011040: 0000 0200 1d00 0000 0000 0000 0000 0000  ................
...
```

The `08af 5109 80ff ffff` in the second part of the first line looks like an address that corresponds to kernel code. Indeed, it is the address of `binder_fops`:

```
# echo 0 > /proc/sys/kernel/kptr_restrict
# cat /proc/kallsyms | grep ffffff800951af08
ffffff800951af08 r binder_fops
```

So looks like these are `file struct` of the binder files that I opened and what I’m seeing here is the field `f_ops` that points to the `binder_fops`. Moreover, the 32 bytes after `f_ops` are the same for every file struct of the same type, which offers a pattern to identify these files. So by reading the memory behind the DMA buffer and looking for this pattern, I can locate the `file` structs that belong to the binder devices that I opened.

Moreover, the `file` struct contains a `mutex` `f_pos_lock`, which contains a field `wait_list`:

```
struct mutex {
        atomic_long_t           owner;
        spinlock_t              wait_lock;
    ...
        struct list_head        wait_list;
...
};
```

which is a standard doubly linked list in linux:

```
struct list_head {
        struct list_head *next, *prev;
};
```

When `wait_list` is initialized, the head of the list will just be a pointer to itself, which means that by reading the `next` or `prev` pointer of the `wait_list`, I can obtain the address of the `file` struct itself. This will then allow me to work out the address of the DMA buffer which I can control because the offset between the `file` struct and the buffer is known. (By looking at its offset in the data that I dumped, in this example, the offset is `0x11020`).

By using the address of `binder_fops`, it is easy to work out the KASLR slide and defeat KASLR, and by knowing the address of a controlled DMA buffer, I can use it to store a fake `file_operations` ("vtable" of `file` struct) and overwrite `f_ops` of my `file` struct to point to it. The path to arbitrary code execution is now clear.

1. Use the out-of-bounds read primitive gained from the use-after-free to dump memory behind a DMA buffer that I controlled.
2. Search for binder `file` structs within the memory using the predictable pattern and get the offset of the `file` struct.
3. Use the identified `file` struct to obtain the address of `binder_fops` and the address of the `file` struct itself from the `wait_list` field.
4. Use the `binder_fops` address to work out the KASLR slide and use the address of the `file` struct, together with the offset identified in step two to work out the address of the DMA buffer.
5. Use the out-of-bounds write primitive gained from the use-after-free to overwrite the `f_ops` pointer to the file that corresponds to this `file` struct (which I owned), so that it now points to a fake `file_operation` struct stored in my DMA buffer. Using file operations on this file will then execute functions of my choice.

Since there is nothing special about `binder` files, in the actual exploit, I used `/dev/null` instead of `/dev/binder`, but the idea is the same. I'll now explain how to do the last step in the above to gain arbitrary kernel code execution.

Getting arbitrary kernel code execution

To complete the exploit, I'll use "the ultimate ROP gadget" that was used in An iOS hacker tries Android of Brandon Azad (and I in fact stole a large chunk of code from his exploit). As explained in that post, the function `__bpf_prog_run32` can be used to invoke eBPF bytecode supplied through the second argument:

```
unsigned int __bpf_prog_run32(const void *ctx, const bpf_insn *insn)
```

to invoke eBPF bytecode, I need to set the second argument to point to the location of the bytecode. As I already know the address of a DMA buffer that I control, I can simply store the bytecode in the buffer and use its address as the second argument to this call. This would allow us to perform arbitrary memory load/store and call arbitrary kernel functions with up to five arguments and a 64 bit return value.

There is, however one more detail that needs taking care of. Samsung devices implement an extra protection mechanism called the Realtime Kernel Protection (RKP), which is part of Samsung KNOX. Research on the topic is widely available, for example, Lifting the (Hyper) Visor: Bypassing Samsung's Real-Time Kernel Protection by Gal Beniamini and Defeating Samsung KNOX with zero privilege by Di Shen.

For the purpose of our exploit, the more recent A Samsung RKP Compendium by Alexandre Adamski and KNOX Kernel Mitigation Byapsses by Dong-Hoon You are relevant. In particular, A Samsung RKP Compendium offers a thorough and comprehensive description of various aspects of RKP.

Without going into much details about RKP, the two parts that are relevant to our situation are:

1. RKP implements a form of CFI (control flow integrity) check to make sure that all function calls can only jump to the beginning of another function (JOPP, jump-oriented programming prevention).
2. RKP protects important data structure such as the credentials of a process so they are effectively read only.

Point one means that even though I can hijack the `f_ops` pointer of my file struct, I cannot jump to an arbitrary location. However, it is still possible to jump to the start of any function. The practical implication is that while I can control the function that I call, I may not be able to control call arguments. Point two means that the usual shortcut of overwriting credentials of my own process to that of a root process would not work. There are other post-exploitation techniques that can be used to overcome this, which I'll briefly explain later, but for the exploit of this post, I'll just stop short at arbitrary kernel code execution.

In our situation, point one is actually not a big obstacle. The fact that I am able to hijack the `file_operations`, which contains a whole array of possible calls that are thin wrappers of various syscalls means that it is likely to find a file operation with a 64 bit second argument which I can control by making the appropriate syscall. In fact, I don't even need to look that far. The first operation, `llseek` fits the bill:

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
    ...
```

This function takes a 64 bit integer, `loff_t` as the second argument and can be invoked by calling the `lseek64` syscall:

```
off_t lseek64(int fd, off_t offset, int whence);
```

where `offset` translates directly into `loff_t` in `llseek`. So by overwriting the `f_ops` pointer of my file to have its `llseek` field point to `__bpf_prog_run32`, I can invoke any eBPF program of my choice any time I call `lseek64`, without even the need to trigger the bug again. This gives me arbitrary kernel memory read/write and code execution.

As explained before, because of RKP, it is not possible to simply overwrite process credentials to become root even with arbitrary kernel code execution. However, as pointed out in Mitigations are attack surface, too by Jann Horn, once we have arbitrary kernel memory read and write, all the userspace data and processes are essentially under control and there are many ways to gain control over privileged user processes, such as those with system privilege to effectively gain system privileges. Apart from the concrete technique mentioned in that post for accessing sensitive data, another concrete technique mentioned in Galaxy's Meltdown - Exploiting SVE-2020-18610 is to overwrite the kernel stack of privileged processes to gain arbitrary kernel code execution as a privileged process. In short, there are many post exploitation techniques available at this stage to effectively root the phone.

Conclusion

In this post I looked at a use-after-free bug in the Qualcomm kgsl driver. The bug was a result of a mismatch between the user supplied memory type and the actual type of the memory object created by the kernel, which led to incorrect clean up logic being applied when an error happens. In this case, two common software errors, the ambiguity in the role of a type, and incorrect handling of errors, played together to cause a serious security issue that can be exploited to gain arbitrary kernel code execution from a third-party app.

While great progress has been made in sandboxing the userspace services in Android, the kernel, in particular vendor drivers, remain a dangerous attack surface. A successful exploit of a memory corruption issue in a kernel driver can escalate to gain the full power of the kernel, which often result in a much shorter exploit bug chain.

The full exploit can be found here with some set up notes.

Next week I'll be going through the exploit of Chrome issue 1125614 (GHSL-2020-165) to escape the Chrome sandbox from a beta version of Chrome.