A foray into Linux kernel exploitation on Android

In November of 2020, I decided to dive into the world of Android, more specifically the linux kernel. I did this because earlier in the year, around February, I broke my old phone during a skiing trip and hastily bought a cheap android phone, the Alcatel 1S 2019.

Coming from a background of Windows research I had no idea how to start. My first intuition was that in order to do anything I'd need to root my phone. Thankfully the Alcatel hosts a UNISOC SC9863A, a chipset made by Spreadtrum, who fortunately also provide flashing tools, making the entire process extremely easy, I'll skip this part because it's not really relevant and varies from phone to phone.

The next goal was to find the source code, and due to licensing requiremnts the linux kernel in use has to be open source. A quick search on google for "Alcatel linux source code" netted me a link to their sourceforge where I found all the zips of each phone titled by version (5024 for the Alcatel 1S 2019).

At this point I had to start assessing where I think there would be any poorly written code, I knew I wouldn't be auditing shared linux kernel code. My first guess to figure out where I could find some targets would be to check all the devices and loaded kernel modules. In the loaded devices (`/dev`) I didn't find anything unique to the phone, so I decided to move on and check the linux kernel modules, and these were the results.

```
Module                  Size  Used by
sprd_fm                57344  0
sprdbt_tty             24576  0
sprd_vibrator          16384  0
sunwave_fp             20480  1
leds_sprd_bltc_rgb     20480  0
pvrsrvkm             1454080  105
```

I didn't want to explore any uncharted territory, so I ended up doing some research if there had been any previous work done on any of these modules, I found Di Shens video on exploiting the pvrsrvkm, which is the PowerVR Kernel Module for PowerVR gpus. While the linux kernel source provided by Alcatel had no mentions on pvrsrvkm, the driver is opensource in the chromiumOS source code and can be found here

The PowerVR kernel module contains only 1 `ioctl` that is of any importance and its purpose is to dispatch to relevant functions, you send in a package (described by the struct below), the `bridge_id` describing the subsystem to which the function belongs, and the `bridge_func_id` is an identifier for each function, the rest of the variables are for transporting data.

```
typedef struct drm_pvr_srvkm_cmd {
        __u32 bridge_id;
        __u32 bridge_func_id;
        __u64 in_data_ptr;
        __u64 out_data_ptr;
        __u32 in_data_size;
        __u32 out_data_size;
} ioctl_package;
```

The Bug hunting process

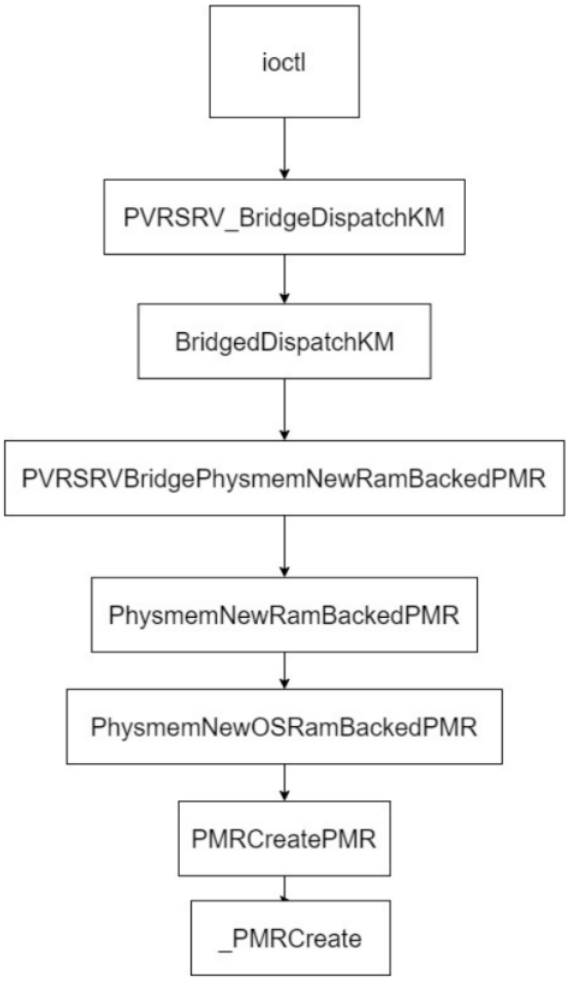
Looking through the driver, there was about 200000 lines of code in total, so I went around looking at subsystems, the most interesting seemed to be memory management.

I went through the functions registered to the memory management bridge, manually auditing them. For debugging, I couldn't connect any sort of kernel debugger to my phone, so instead I wrote a kernel module that would hook functions and intercept them to log values, while crude it helped immensely in understanding. Thankfully there was already a lot of logging in the PowerVR module, easily obtainable with `dmesg`

That's when I found it.

The Bug

Here is the call-stack (backtrace) or code flow to reach the bug.



The function that I am targetting (`PhysmemNewRamBackedPMR`) is responsible for creating shared memory, it has certain options for whether the memory should be GPU, OS, etc… The one constraint being that this memory is actually backed by physical memory (`PMR = PhysicalMemoryResource`), with the choice given to the user on how to map virtual to physical addresses.

This is the structure that is passed in to PRVSRVBridge

```
typedef struct PVRSRV_BRIDGE_IN_PHYSMEMNEWRAMBACKEDPMR_TAG
{
        uint64_t uiSize;
        uint64_t uiChunkSize;
        __u32 ui32NumPhysChunks;
        __u32 ui32NumVirtChunks;
        __u32 * pui32MappingTable;
        __u32 ui32Log2PageSize;
        __u32 uiFlags;
        __u32 ui32AnnotationLength;
        const char * puiAnnotation;
        __u32 ui32PID;
} __attribute__((packed))
PVRSRV_BRIDGE_IN_PHYSMEMNEWRAMBACKEDPMR;
```

It describes many things like the name of the mapping, how many chunks, and certain flags, the mapping table for virtual to physical.

the final function in the chain `_PMRCreate` is responsible for creating a physical memory resource and actually handling the translation/bookkeeping between virtual and physical addresses, the bug occurs in the initialization of this bookkeeping.

```
        pvPMRLinAddr = OSAllocMem(sizeof(*psPMR) +
sizeof(*psMappingTable) + sizeof(IMG_UINT32) *
ui32NumVirtChunks);

        if (pvPMRLinAddr == NULL)
        {
                return PVRSRV_ERROR_OUT_OF_MEMORY;
        }

        psPMR = (PMR *) pvPMRLinAddr;
        psMappingTable = (PMR_MAPPING_TABLE *) (((IMG_CHAR
*) pvPMRLinAddr) + sizeof(*psPMR));

        eError = OSLockCreate(&psPMR->hLock,
LOCK_TYPE_PASSIVE);
        if (eError != PVRSRV_OK)
        {
                OSFreeMem(psPMR);
                return eError;
        }
        psMappingTable->uiChunkSize = uiChunkSize;
        psMappingTable->ui32NumVirtChunks =
ui32NumVirtChunks;
        psMappingTable->ui32NumPhysChunks =
ui32NumPhysChunks;
        OSCachedMemSet(&psMappingTable-
>aui32Translation[0], 0xFF, sizeof(psMappingTable-
>aui32Translation[0])*
                        ui32NumVirtChunks);
        for (i=0; i<ui32NumPhysChunks; i++)
        {
                ui32Temp = pui32MappingTable[i];
                psMappingTable->aui32Translation[ui32Temp]
= ui32Temp;
        }
```

In this code, `pui32MappingTable` is a copied over usermode table that describes linear translations from virtual to physical addresses. With -1 being the default value meaning that no translation was provided. The issue being that the `aui32Translation` table is only the size of the number of Virtual Chunks provided.

This on its own is not an issue, because in order to reach this area there is a constraint, the number of virtual chunks has to be equal to the number of physical checks. The actual issue occurs in the bounds checking when setting up these linear chunks, the index `ui32Temp` is a user controlled variable because it comes from the unverified `pui32MappingTable` and indexs into an array which is meant to hold the number of chunks requested, this gives us a heap overwrite primitive in any cache larger than or equal to `kmalloc-256`, due to the fact that we can control how large `psMappingTable` is, by controlling `ui32NumVirtChunks`

Exploitation

I limited myself to write an exploit with PXN/PAN and ASLR exploitation mitigations in mind, the first step would be to leak an kernel address of any kind, heap or ideally a function address.

Since `aui32Translation` is a 32-bit pointer, I cannot do any freelist shenanigans because realistically the values that I can set and groom are from 0-1000, larger than that I start to edit memory 1000s of bytes away from the corrupted block, making it very unreliable

My next thought was that I have a 4 byte arbitrary overwrite with values 0-1000 which meant I had realistically 2 options, find a structure with a reference counter and cause a UaF or corrupt a length integer.

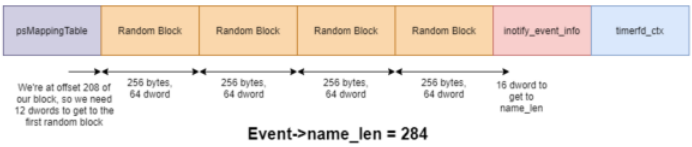
Upon searching for previous research on usable linux structures, I found this great talk talking about elastic objects in the linux kernel, specifically a chart of objects, where I found the suitable inotify_event_info, The idea here being corrupting the length of the inotify_event_info and grooming another object infront of the inotify_event_info and reading it.

What is inotify_event_info and how do we use it? Linux has mechanisms to notify programs when files are changed/modified/deleted etc…. inotify_event_info is the structure used to store the events, the name, the type of operation, a cookie.

essentially it works by giving you a file descriptor which you create with certain options, then when you read it, if there are any events you can easily retrieve them by reading the fd. When an event occurs, it is allocated to the kmalloc-256 cache and when you read it is destroyed.

the gameplan being, fill the heap with inotify_event_infos then, free an inotify_event_info, and trigger the bug overwriting a length value, then as you read the inotify_event_infos you start cluttering the remaining free spaces with a structure whose content you want to leak.

in my case the ideal structure for leaking a function address is a timer, which contains a function address in the first 30 bytes and is very easy to use.



I wrote the PoC, adjusted values and figured out and I kept on crashing, but what was happening??? Here were the panic logs

```
[<ffffff80081df04c>] __check_object_size+0x54/0x220
[<ffffff80082296c4>] inotify_read+0x314/0x39c
[<ffffff80081e2f14>] vfs_read+0x84/0x168
[<ffffff80081e39c4>] sys_read+0x50/0xb0
[<ffffff8008085af0>] el0_svc_naked+0x24/0x28
```

and this line was also in the panic logs

```
<0>[ 5522.162040] c7 14180 usercopy: kernel memory
exposure attempt detected from ffffffc0f50d912c (kmalloc-
256) (284 bytes)
```

a little bit of research led me to, CONFIG_HARDENED_USERCOPY, a protection that stops you copying out of your allocation, meaning that this wouldn't work.

Revelation

In my search to find a good object whose `kref` I could corrupt, while learning about linux I discovered a little thing known as slab merging, in order to optimize and reduce fragmenting in the kernel memory allocator, linux will merge slabs into one slab. The linux kernel source comes with a little program called slabinfo (not to be confused with `/proc/slabinfo`) that can show you merged slabs, I thought of the structures I could use to corrupt and while there were many in the kmalloc-256+ caches, I couldn't find something that was easy to use and groom.

I decided to check what slabs were merged on my phone by running slabinfo with the `-a` flag,

```
:t-0000256   <- ip_dst_cache kmalloc-256 filp
nf_conntrack_expect skbuff_head_cache pool_workqueue
biovec-16 bio-0 sgpool-8
:t-0000360   <- blkdev_requests dm_clone_request
:t-0000512   <- xfrm_dst_cache kmalloc-512 sgpool-16
skbuff_fclone_cache
```

This was my output, I noticed that of the 256 byte caches, filp and kmalloc-256 were merged, this meant that I could corrupt file structures, let's see what we can corrupt in the file structures.

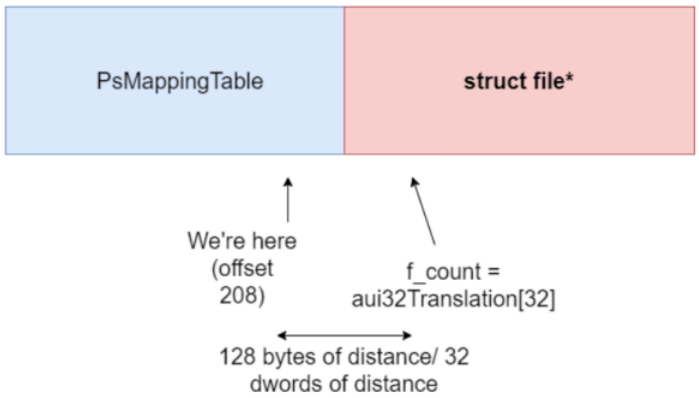
```
struct file {
        union {
                struct llist_node       fu_llist;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct path             f_path;
        struct inode            *f_inode;       /* cached
value */
        const struct file_operations    *f_op;

        /*
         * Protects f_ep, f_flags.
         * Must not be taken from IRQ context.
         */
        spinlock_t              f_lock;
        enum rw_hint            f_write_hint;
        atomic_long_t           f_count;
        unsigned int            f_flags;
        fmode_t                 f_mode;
        struct mutex            f_pos_lock;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        const struct cred       *f_cred;
        struct file_ra_state    f_ra;

        u64                     f_version;
#ifdef CONFIG_SECURITY
        void                    *f_security;
#endif
        /* needed for tty driver, and maybe others */
        void                    *private_data;

#ifdef CONFIG_EPOLL
        /* Used by fs/eventpoll.c to link all the hooks to
this file */
        struct hlist_head       *f_ep;
#endif /* #ifdef CONFIG_EPOLL */
        struct address_space    *f_mapping;
        errseq_t                f_wb_err;
        errseq_t                f_sb_err; /* for syncfs */
} __randomize_layout
  __attribute__((aligned(4)));  /* lest something weird
decides that 2 is OK */
```

Fortunately for us, there's an atomic integer that holds a reference count `f_count`, my idea is that we can open a file many times and increment the counter then we can trigger the heap overwrite to set the `f_count` to a number that is smaller than the current number of references, and then we can close it until, we have handles in our processes file descriptor table that point to `file*` that are non-existent, then from there on we can see what we can do, whether it be manipulating data structures or other things.



The smallest number we can set the refcount to is 32.

One of the nice things about being able to mess with file structures, is that there are very easy paths to LPE, if you can replace file structures and keep handles on them, then you might be able to get access to a file that you shouldn't have access to e.g. imagine SUID executables, giving you root and more…

I also took some inspiration from Jann Horns video, where he mentions how he exploits a file refcount overdecrement, using `FUSE`, and using vectored i/o he was able to replace `file*` by stalling a write operation once access checks were done, and was able to write contents of a file he did not have write access to. Unfortunately, `FUSE` is not accessible in android, so the second best option I had was trying to slow down the processor with interrupts and scheduler tricks.

Reliability

Another syscall he used to check that his UaF was reliable was `kcmp`, sadly `kcmp` isn't implemented in android, but I found an alternative. Using inotify events I can see when a file has been closed, and since only the file with the altered refcount will be closed I'll know whether and when I've closed the corrupted file. As for checking whether a file has been successfully replaced, I can use lseek on new `struct file`s and set the position to some magic number, then I can check my dangling handles and if the seek position has been changed to that magic number we can be sure that our dangling file has been replaced, :P

This is needed because once we find our replaced file, we can still have multiple handles pointing to it and still cause another refcount overflow but this time we know which handle it is, making it a lot more deterministic.

The New Gameplan

1) Create an inotify event for a specific directory, for when files are closed

2) Create/open a bunch files, increment their refcount to a suitable number (to any number above 32), this can be done using the `dup` syscall

3) deallocate a file, to make holes in the block, allocate a `psMappingTable` in the hole.

4) Trigger the bug causing a heap overwrite and hopefully adjusting the refcount of a `struct file` infront of it to 32

5) Close the `dup` fds, 32 times.

6) if we can read an event from inotify, then we've closed a corrupted file meaning that its refcount has been altered.

7) start spraying files and seeking their position to a suitable magic number, check the dangling handles seek positions, if they have changed we've found the corrupted file and we have spare handles to a `file*` with refcount = 1 and probably more dangling handles depending on how many we started with

10) write to the file using the vectored i/o, and try the replacement trick that Jann Horn used, using timing delays to make the race condition easier to hit

11) hopefully replace the file* during the vectored i/o with a much more important file (I used the `runas` file which has suid permissions, which I can open as readable)

12) run modified suid file with LPE payload inside.

The Unfortunate Conclusion

Since I didn't have `FUSE`, I was unable to ever manage to even get the race condition to hit, but the idea was still there. I used scheduler tricks, even tried some interrupt tricks from another talk. Though it seemed impossible to ever get.

Possible alternatives

Thanks to `nspace`, who recommended an alternative exploitation strategy for this bug. The idea was that I somehow communicate with priviledged services that open priviledged files, and that they'd replace the corrupted `file*` struct and I'd have spare handles to a priviledged file that I could edit. Unfortunately I wasn't able to find any easy or useful services that open important priviledged files that I could repeatedly spray reliably.

If anyone has some ideas feel free to message me, I'd love to hear them
ayazmammadov@hotmail.co.uk

I'm looking for employment opportunies
Written on March 31, 2021