A 0-click exploit chain for the Pixel 9 Part 2: Cracking the Sandbox with a Big Wave

2026-JAN-14 Seth Jenkins

With the advent of a potential Dolby Unified Decoder RCE exploit, it seemed prudent to see what kind of Linux kernel drivers might be accessible from the resulting userland context, the `mediacodec` context. As per the AOSP documentation (https://source.android.com/docs/core/media/framework-hardening?authuser=1#mediacodecservice_changes) , the `mediacodec` SELinux context is intended to be a constrained (a.k.a sandboxed) context where non-secure software decoders are utilized. Nevertheless, using my DriverCartographer tool, I discovered an interesting device driver, `/dev/bigwave` that was accessible from the `mediacodec` SELinux context. BigWave is hardware present on the Pixel SOC that accelerates AV1 decoding tasks, which explains why it is accessible from the `mediacodec` context. As (https://googleprojectzero.blogspot.com/2024/12/qualcomm-dsp-driver-unexpectedly-excavating-exploit.html) previous (https://googleprojectzero.blogspot.com/2024/06/driving-forward-in-android-drivers.html) research (https://googleprojectzero.blogspot.com/2023/09/analyzing-modern-in-wild-android-exploit.html) has (https://project-zero.issues.chromium.org/issues/380081941) copiously (https://project-zero.issues.chromium.org/issues/42451599) affirmed (https://project-zero.issues.chromium.org/issues/389724938) , Android drivers for hardware devices are prime places to find powerful local privilege escalation bugs. The BigWave driver was no exception - across a couple hours of auditing the code, I discovered three separate bugs, including one that was powerful enough to escape the `mediacodec` sandbox and get kernel arbitrary read/write on the Pixel 9.

The (Very Short) Bug Hunt

The first bug I found (https://project-zero.issues.chromium.org/u/1/issues/425917200) was a duplicate that was originally reported in February of 2024 but remained unfixed at the time of re-discovery in June of 2025, over a year later, despite the bugfix being a transposition of two lines of code. The second bug (https://project-zero.issues.chromium.org/u/1/issues/426548270) presented a really fascinating bug-class that is analogous to the double-free kmalloc exploitation primitive - but with a different linked list entirely. However it was the third bug (https://project-zero.issues.chromium.org/issues/426567975) I discovered that created the nicest exploitation primitive. Fixes were made available for all three bugs on January 5, 2026.

The Nicest Bug

Every time the `/dev/bigwave` device is opened, the driver allocates a new kernel struct called `inst` which is stored in the `private_data` field of the `fd`. Within the `inst` is a sub-struct called `job`, which tracks the register values and status associated with an individual invocation of the BigWave hardware to perform a task. In order to submit some work to the bigo hardware, a process uses the ioctl `BIGO_IOCX_PROCESS`, which fetches Bigwave register values from the ioctl caller in AP userland, and places the `job` on a queue that gets picked up and used by a separate thread, the bigo worker thread. That means that an object whose lifetime is inherently bound to a file descriptor is transiently accessed on a separate kernel thread that isn't explicitly synced to the existence of that file descriptor. During `BIGO_IOCX_PROCESS` ioctl handling, after submitting a `job` to get executed on `bigo_worker_thread`, the ioctl call enters `wait_for_completion_timeout` with a timeout of 16 seconds waiting for `bigo_worker_thread` to complete the job. After those 16 seconds, if `bigo_worker_thread` has not signaled job completion, the timeout period ends and the ioctl dequeues the `job` from the priority queue. However, if a sufficient number of previous jobs were stacked onto the `bigo_worker_thread`, it is possible that `bigo_worker_thread` was so delayed that it has only just dequeued and is concurrently processing the very `job` that the ioctl has considered to have timed out and is trying to dequeue. The syscall context in this case simply returns back to userland, and if at this point userland closes the `fd` associated with the BigWave instance, the `inst` (and thusly the `job`) is destroyed while `bigo_worker_thread` continues to reference the `job`.

The highlights indicate any accesses to the UAF'd object:

```
static int bigo_worker_thread(void *data)
{
        ...

        while(1) {
                rc = wait_event_timeout(core->worker,
                        dequeue_prioq(core, &job, &should_stop),
                        msecs_to_jiffies(BIGO_IDLE_TIMEOUT_MS)); //The job is fetched from
the queue
                ...

                inst = container_of(job, struct bigo_inst, job); //The job is an inline
struct inside of the inst which gets UAF'd

                ...

                rc = bigo_run_job(core, job);

                ...
                job->status = rc;
                complete(&inst->job_comp);
        }
        return 0;
}

...

static int bigo_run_job(struct bigo_core *core, struct bigo_job *job)
{
        ...

        inst = container_of(job, struct bigo_inst, job);
        bigo_bypass_ssmt_pid(core, inst->is_decoder_usage);
        bigo_push_regs(core, job->regs); //The register values of the bigwave processor are
set (defined by userland)
        bigo_core_enable(core);
```

```
        ret = wait_for_completion_timeout(&core->frame_done,
                        msecs_to_jiffies(core->debugfs.timeout)); //pause for 1 second
        ...
        //At this point inst/job have been freed
        bigo_pull_regs(core, job->regs); //A pointer is taken directly from the freed object
        *(u32 *)(job->regs + BIGO_REG_STAT) = status;
        if (rc || ret)
                rc = -ETIMEDOUT;
        return rc;
}


void bigo_pull_regs(struct bigo_core *core, void *regs)
{
        memcpy_fromio(regs, core->base, core->regs_size); //And the current register values
of the bigwave processor are written to that location
}
```

By spraying attacker-controlled `kmalloc` allocations (for example via Unix Domain Socket messages) we can control the underlying UAF pointer `job->regs`, so we can control the destination of our write. Additionally since we set the registers at the beginning of execution, by setting the registers in such a way that the BigWave processor does not execute at all, we can ensure that the end register state is nearly identical to the original register state - hence we can control what is written as well. And just like that, we have a half decent 2144-byte arbitrary write! And all without leaking the KASLR slide!

Defeating KASLR (by doing nothing at all)

Exploiting this issue with KASLR enabled would normally involve reallocating some other object over the bigo `inst` with a pointer at the location of `inst->job.regs`, leading to memory corruption of the object pointed to by that overlapped pointer. That would require finding some allocatable object with a pointer at that location, and also finding a way to take advantage of being able to overwrite the sub-object. Finding such an object is difficult but not impossible, especially if you consider cross-cache attacks. It is, however, quite tedious and is

not really my idea of a fun time. Thankfully I found a much simpler strategy which essentially allows the generic bypass of KASLR on Pixel in its entirety, the details of which you can read about in [my previous blog post (https://googleprojectzero.blogspot.com/2025/11/defeating-kaslr-by-doing-nothing-at-all.html)](). The end-result of that sidequest is the discovery that instead of needing to leak the KASLR base, you can just use `0xffffff8000010000` instead, particularly when it comes to overwriting .data in the kernel. This dramatically simplifies the exploit, and substantially improves the exploit's potential reliability.

Creating an arbitrary read/write

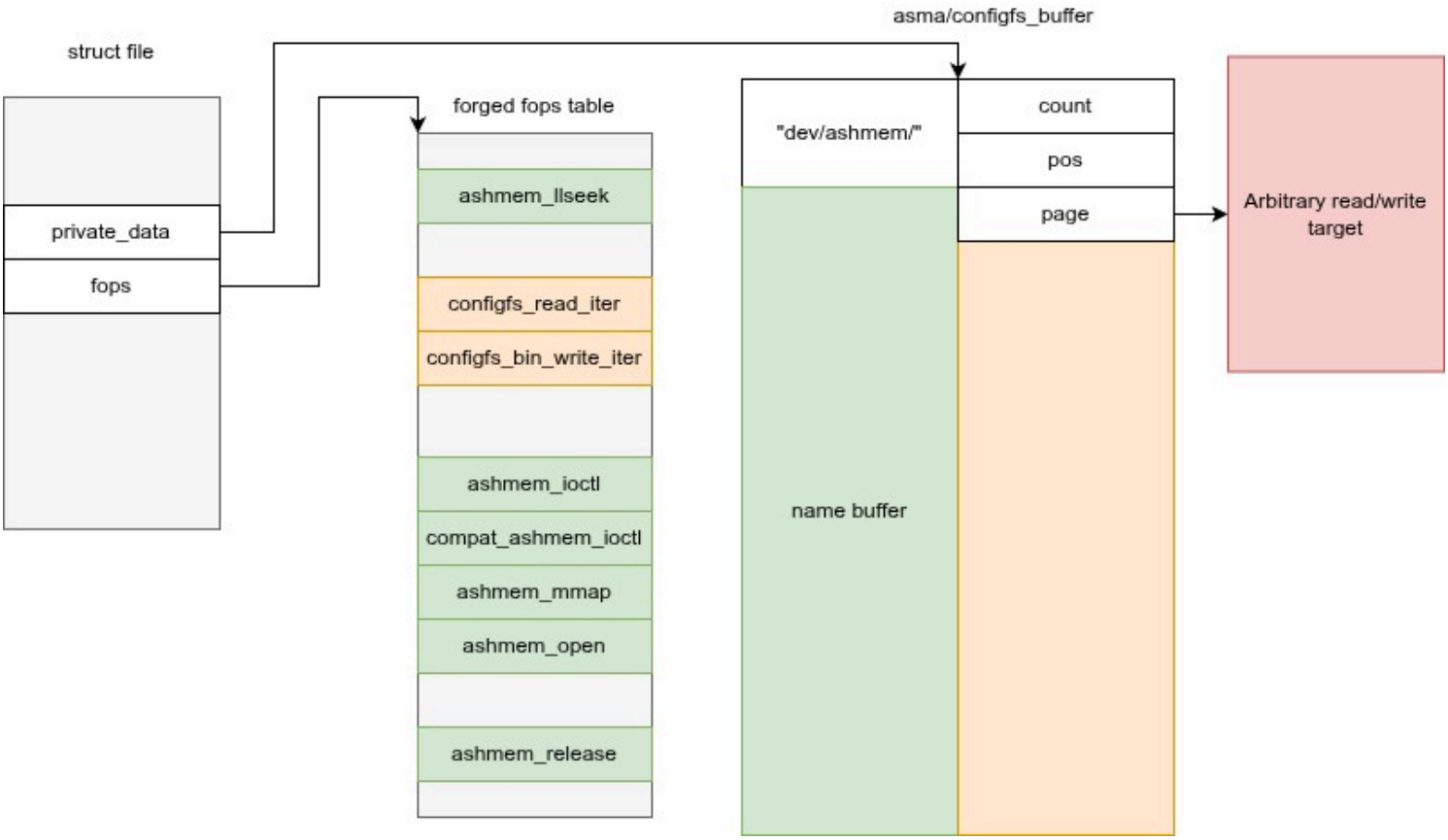
At this point, I have a mostly-arbitrary write primitive anywhere in kernel .data - I have an aliased location for, and can modify, any kernel globals I want. However the `complete` call at the end of the `bigo_worker_thread` job execution loop serves to complicate exploitation a little bit. `complete` calls `swake_up_locked` which performs a set of list operations on a `list_head` node inside of the bigo `inst`:

```
static inline int list_empty(const struct list_head *head)
{
return READ_ONCE(head->next) == head;
}

void swake_up_locked(struct swait_queue_head *q) //The q is located at &inst->job_comp.wait
(so attacker controlled)
{
        struct swait_queue *curr;

        if (list_empty(&q->task_list))
                return;

        curr = list_first_entry(&q->task_list, typeof(*curr), task_list);
        wake_up_process(curr->task);
        list_del_init(&curr->task_list);
}
```

While the first `list_empty` call would be the simplest to forge, it would also require knowing the location of the `inst` in kernel memory as `q` is an inline struct inside of `inst`. Unfortunately, our KASLR bypass does not give us this, nor is it particularly easy to acquire, as the `inst` is in kernel heap, not kernel .data. That means we need to instead forge a valid list entry for the `q` to point to as well as know the location of a task to pass to `wake_up_process()`. Finally we need to actually forge enough of a list to survive a `list_del_init` on an entry in the `q->task_list`, which involves list nodes, and second list nodes that point to the first list node. This might sound quite difficult to forge given the limitation we've previously noted about our KASLR bypass, but in fact, it's not so bad, since our arbitrary write has already happened by this point - so we know the location of memory that we control somewhere in kernel .data. This means we can forge arbitrary list nodes within that space in .data, and we can place pointers to those future forged list nodes in the original heap spray we use to replace the `inst`. We ALSO know the location of a single task struct in the kernel virtual address space - the `init` task! `init`'s task struct is in the kernel .data, so we can reference it through the linear map. A spurious `wake_up_process` on the `init_task` will be entirely inconsequential while avoiding a crash. You can see the code to set up these linked list nodes in `setup_linked_list` in the exploit.

With that roadblock resolved, it's time to figure out what in .data to target with our arbitrary write. Our goal is to change our unreliable arbitrary write of 2144 bytes to a reliable arbitrary read/write that causes significantly less

collateral damage to the memory around it. I decided to try reimplementing the strategy I reversed from an ITW exploit a couple years ago (https://googleprojectzero.blogspot.com/2023/09/analyzing-modern-in-wild-android-exploit.html#:~:text=Stabilizing%20The%20Arbitrary%20Write) . This technique involves creating a type-confusion by replacing some of the VFS/fops handlers in the `ashmem_misc` data structure with other VFS handlers for other file types. In fact, because of CFI you cannot replace the handler function pointers with pointers to just any location in the kernel .text. You must replace the VFS handlers with other VFS handlers. Rather conveniently however, I can use configfs VFS handlers for my exploit, just like the ITW exploit. The final layout of the fops table and `private_data` of the `struct file` look like this:



The fops handlers in green will access the `private_data` structure as a `struct ashmem_area`, or `asma`, while the fops handlers in yellow access the same `private_data` structure as a `configfs` buffer. For the `configfs` fops handlers, the memory pointed to by `page` will be accessed - that is where we will want our arbitrary read/write to read or write. We will set our target using the `ASHMEM_SET_NAME` ioctl.

One additional complication however, is that the linear mapping of the kernel .text is not executable, so I can't use .text region linear map addresses to the VFS handlers when forging my `ashmem_misc` data structure. In practice, it's not particularly difficult to leak the actual KASLR slide. Before targeting `ashmem_misc`, I first use my arbitrary write to target the `sel_fs_type` object in the kernel .data. This structure has a string, name, that is printed when reading `/proc/self/mounts`. By replacing that string pointer using my arbitrary write, and then reading `/proc/self/mounts`, I can turn my unreliable arbitrary write into an arbitrary read instead! Using this arbitrary read, I can read the `ashmem_fops` structure (also through the linear map) which gives me pointers at an offset from the kernel base, allowing me to calculate the KASLR slide.

I then perform my arbitrary write again to overwrite the `ashmem_misc` structure with a pointer to a new forged `ashmem_fops` table that I construct at the same time - such is the perk of overwriting far more data than I need.

However, the astute among you may have realized that this massive 2144 byte arbitrary write has a major drawback too, as such a large write will clobber all of the data surrounding whatever I'm actually targeting with the write - this could lead to all sorts of extraneous crashes and kernel panics. In practice, spurious crashing can occur, but the phone is surprisingly quite stable. My experience was that it seemed to crash upon toggling the wifi on/off - but otherwise the phone seems to work mostly fine.

Once the forged `ashmem_misc` structure has been inserted, we now have a perfectly reliable arbitrary read/write, albeit with the phone extraneously crashing sometimes. Upon getting arb read/write, I set SELinux to permissive (just flip the flag in the `selinux_state` kernel object), fork off a new process, then use my arb read/write to point the new process's task creds to `init_cred`. At this point, I now have a process with root credentials, and SELinux disabled.

Integrating into the Dolby exploit

Combining two exploits into one chain requires a fair amount of engineering effort from both exploits. The Dolby exploit will be delivering the Bigwave exploit as a shellcode payload, (patched into the process using `/proc/self/mem`) so I need to convert my exploit to work as a binary blob. It also needs to be much smaller than my static compilation environment supported. The lowest hanging fruit was to remove the static libc requirement and have the exploit include wrappers for all the syscalls and libc functions it needs. When I set about to complete this rather tedious task, I realized that this is something an LLM would probably be quite good at. So instead of implementing the sycall wrappers myself, I simply copy-pasted my source code into Gemini and asked it to create the needed header file of syscall wrappers for me. Naturally the AI-generated header file caused many compilation errors (as it surely would have if I had tried to do it too). I took those compilation errors, gave them back to the same Gemini window, and asked it to amend the header file to resolve those errors. The amended header file caused gcc to emit whole new and exciting compilation failures - but the errors looked different than before, so I simply repeated the process. After 4 or 5 attempts, Gemini was able to generate a header file that not only compiled - it worked perfectly. This provides some insight into how attackers might be able to use (or more likely are already using) LLMs to make their exploit process more efficient.

This effort results in a much smaller ELF than before (7 KB instead of 500 KB) but just an ELF is not enough - I need the generated blob to work if the dolby exploit simply starts executing from the top of the shellcode. The good news however is that my exploit can operate entirely without a linker - all that is necessary is to prepend a jump to the ELF that sets the PC to the entrypoint. I also include "-mcmodel=tiny -fPIC -pie" in the gcc arguments so that the generated code will work agnostic to the shellcode's location or alignment in memory.

Finalizing the exploit

Kernel arbitrary read/write is motivating enough as a security researcher to demonstrate the impact of the vulnerability, but it seemed incumbent to create some more accessible demo in order to demonstrate impact more broadly. I added code so that the exploit executed an included shell script, then wrote a shell script that took a picture and sent that picture back to an arbitrary IP address.

In the [final part (/2026/01/pixel-0-click-part-3.html)](/2026/01/pixel-0-click-part-3.html) of this blog series, we will discuss what lessons we learned from this research.■

make zeroday hard.