6 Burning a Phone

by Josh "@m0nk" Thomas

Earlier this year, I spent a couple months exploring exactly how power routing and battery charging work in Android phones for the DARPA Cyber Fast Track program. I wanted to see if I could physically break phones beyond repair using nothing more than simple software tricks and I also wanted to share the path to my outcomes with the community. I'm sure I will talk at some point about the entire project and its specific targets, but tonight I want to simply walk through breaking a phone, see what it learns us and maybe spur some interesting follow on work in the process.

Because it's my personal happy place, our excursion into kinetic breakage will be contained to the pseudo Linux kernel that runs in all Android devices. More importantly, we will focus the arch/arm/mach-msm subsystem and direct our curiosity towards breaking the commonplace NAND Flash and SD Card hardware components. A neighbor specifically directed me not to include background information in this write-up, but we have to start somewhere prior to frying and disabling hardware internals and in my mind the logical starting point is the common power regulation framework.

The Linux power regulation framework is surprisingly well documented, so I will simply point a curious reader to the kernel's documentation at Documentation/power/regulator/overview.txt. For the purpose of breaking devices, all we really need to understand at the onset are these three things.

- The framework defines voltage parameters for specific hardware connected to the PCB.
- The framework regulates PMIC and other control devices to ensure specific hardware is given the correct voltages.
- The framework directly interacts with both the kernel and the physical PCB, as one would expect from a (meta) driver

It's also worth noting that the PCB has some (albeit surprisingly limited) hardwired protections against voltage manipulations. Further, the kernel has a fairly robust framework to detect thermal issues and controls to shut down the system when temperature thresholds are exceeded.

So, in essence, we have a system with a collection of logical rules that keep the device safe. This makes sense.

Glancing back at our target for attack, we should quickly consider end result potentials. Do we want to simply over volt the NAND chip to the point of frying all the data or do we want something a little more subtle? To me, subtle is sexy..., so let's walk though simply trying to ensure that any NAND writes or reads corrupt any data in transit or storage.

On the Sony Xperia Z platform, all NAND Flash and all SD-Card interactions are actually controlled by the Qualcomm MSM 7X00A SDCC hardware. Given we RTFM'd the docs above, we simply need to implement a slight patch to the kernel:

```
project kernel/sony/apq8064/
diff --git a/arch/arm/mach-msm/board-sony_yuga-regulator.c
          b/arch/arm/mach-msm/board-sony_yuga-regulator.c

-- RPM_LDO(L5, 0, 1, 0, 2950000, 2950000, NULL, 0, 0),
++ RPM_LDO(L5, 0, 1, 0, 5900000, 5900000, NULL, 0, 0),

-- RPM_LDO(L6, 0, 1, 0, 2950000, 2950000, NULL, 0, 0),
++ RPM_LDO(L6, 0, 1, 0, 5900000, 5900000, NULL, 0, 0),
```

Wow that was oddly easy, we simply upped the voltage supplied to the 7X00A from 2.95V to 5.9V. What did it do? Well, given this specific hardware is unprotected from manipulation across the power band at the PCB layer and at the internal silicon layer, we just ensured that all voltage pushed to the NAND or

SD-Card during read / write operations is well above the defined specification. The internal battery can't actually deliver 5.9V, but the PMIC we just talked to will sure as hell try and our end result is a NAND Flash chip that corrupts nearly every block of storage it attempts to write or read. Sometimes the data comes back from a read request normal, but most of the time it is corrupted beyond recognition. Our writes simply corrupt the data in transit and in some cases bleed over and corrupt neighbor data on storage.

Overall, with two small values changed in the code base of the kernel we have ensured that all persistent data is basically unusable and untrustworthy. Given the PMIC devices on the phone retain the last valid setting they've used, even rebooting the device doesn't fix this problem. Rather, it actually makes it much worse by corrupting large swaths of the resident codebase on disk during the read operation. Simply, we just bricked a phone and corrupted all data storage beyond repair or recovery.

If instead of permanently breaking the embedded storage hardware we wanted to force the NAND to hold all resident data unscathed and ensure that the system could not boot or clean itself, we simply need to under-volt the controller instead of upping the values.

If you find this interesting, look forward to my release of a longer variant of this technique that targets all hardware soldered in the phone PCB in paper form on github soon.