École Polytechnique Fédérale de Lausanne

Leveraging Bootloader Exploits on COTS Android Devices for Security Research

by Károly Artúr Papp

Semester Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Dr. Marcel Busch
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 12, 2023

Abstract

Modern smartphones make use of a bootloader to load the kernel, the trusted execution environment (TEE) and verify integrity of the necessary partitions before passing execution to the kernel. Bootloaders are mostly proprietary and not accessible to the general public. Because of this, security researchers must first gain access to the source code, in order to discover vulnerabilities that may affect the security of the whole booting sequence. Since the bootloader is read-only, these vulnerabilities are near impossible to fix, and can only be rectified when the vendor brings out a new chipset. In this project we analysed already known bootloader vulnerabilities on chipsets from two vendors, Qualcomm and HiSilicon. On Qualcomm devices, building on existing research, we broadened the scope to the BQ Aquaris X device, and automatized the process. On HiSilicon devices, we examined four devices, the Huawei P9 Lite, the Huawei P20 Lite, the Huawei Psmart and the Huawei P40 Pro. We were able to load modified components on the P9 Lite and the P20 Lite.

Introduction

Mobile phones have become indispensable tools in modern society, being used for communication, financial transactions, and to access many services. However, due to their importance, security flaws in mobile phones have heightened consequences, potentially affecting millions. In this context, security research plays an important role in identifying vulnerabilities and developing mitigations. The bootloader, a crucial component of a mobile phone's firmware, is responsible for initiating the device's operating system. As such, it presents a critical point of entry for security research. By leveraging bootloader exploits, researchers can access deeper layers of a mobile phone's firmware, allowing for the discovery and analysis of even more potential security flaws. The most useful bootloader exploits are those that can be used to break the chain of trust, and load insecure components.

The two vendors we examined during this project, Qualcomm and HiSilicon, shipped 29% of SoCs for mobile phones in Q1 2023. [9] Qualcomm works with a number of mobile phone vendors, such as Samsung, Xiaomi and other Chinese companies. HiSilicon is owned by Huawei and only ships to them. We examined HiSilicon because of interesting previous research being made on bootloader exploits. For both vendors, the exploits involved a special download mode, which is normally used for repairing bricked phones.

For Qualcomm, we based our research on an article by Bjoern Kerler [6] in which he details how to leverage a vulnerability in Qualcomm chips to load an unverified kernel on a BQ Aquaris X Pro device. Some corrections and modifications were needed to fit the exploit to the BQ Aquaris X, and the whole process was automatized.

For Huawei, the bootloader vulnerability we used was described in a paper by the TASZK Security Labs. [7] The specific Huawei P40 Lite device we had at our disposition was used in previous research. [3] It is unclear if due to this, or other circumstances, but we were only able to reproduce the vulnerability described in the paper to a certain extent. The PSmart device we used showed signs of external and internal damage. Despite our numerous different attempts, we were not able to make the phone boot completely. We were more successful with the Huawei P9 Lite and the Huawei P20 Lite, where we were able to load modified Fastboot images.

In light of this, further analysis and research could be done, both on HiSilicon devices and other vendors. It is yet unclear whether controlling the Fastboot image on the P9 and P20 devices can be used to load unsafe kernels and/or trusted components; our attempts and conclusions in the matter shall be presented at the end of this report.

Qualcomm

Background and previous research

First we familiarize ourselves with a high-level description of the bootchain used on the Aquaris X device. [2]

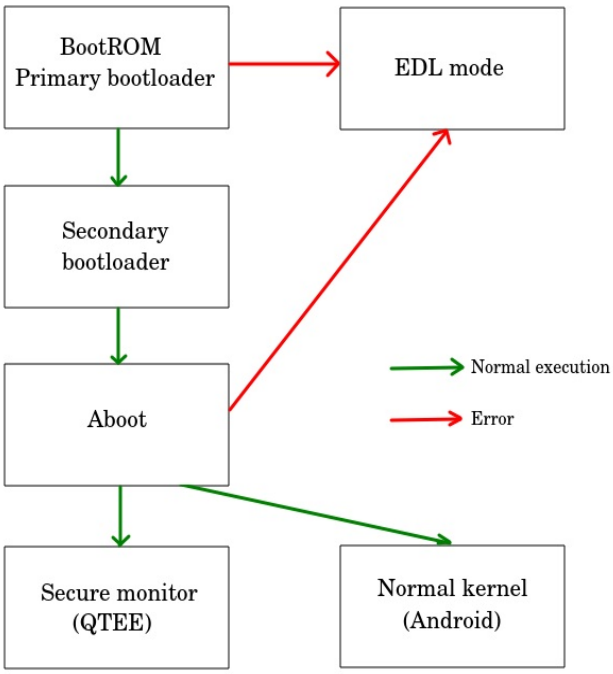

Figure 1: Description of Qualcomm bootchain on non-Samsung devices. As we see, the execution goes towards booting the kernel, unless other circumstances arise.

As seen in Figure 1, in case of an error the booting process goes in to Emergency Download mode (EDL). This can be triggered by:

- an error inside the Secondary bootloader
- in the case of the Aquaris X, using the buttons, or rebooting into EDL mode from the command line
- in the case of other Qualcomm devices, shorting a test pin may be necessary

In case there is no error, or indication to boot into EDL mode, the bootchain works as intended, with the Secure monitor being booted at EL3 (highest privilege) and the kernel at EL1 (normal privileges).

The security researchers at Aleph Research have published an extensive series of articles on bootloader vulnerabilities in Qualcomm chipsets, [5] and the exploit demonstrated by Kerler builds upon these vulnerabilities. [6] To summarize, Hay and Hadad built a tool upon the existing Firehose protocol that takes advantage of insufficient rollback protection, and a powerful peek and poke primitive that allows for arbitrary writes. Kerler then uses this tool on the Aquaris X Pro to:

- modify the stock kernel to remove xpu restrictions and logging of SuperVisor Calls (SVC) in the trusted components

- add a fake root to the kernel and sign it
- patch the TrustZone partition with a shellcode to allow code injection on-the-fly
- patch the aboot partition to allow custom ramdisks

Discussion

The exploit proposed by Kerler was not immediately adaptable to the Aquaris X. There were numerous faults that had to be corrected, which required combing over the toolchain. Also, since some tools have been updated, it was necessary to figure out how to get all the components to cooperate. Since the firmware versions were different, we had to find the right image, since updating to one that is too recent could cause problems with the toolchain and the exploits. We even tried loading the kernel used in the X Pro onto the X, but this resulted in bus errors when trying to write to the appropriate places. The exploit was also updated for Android 10, and contrary to what was stated, it was not backwards compatible.

The more interesting part of the research was figuring out, how the memory layout of the Aquaris X differs from that of the Aquaris X Pro and updating the exploits accordingly. Even after writing the shellcode to the updated address, running the code made the phone crash. After some research we tried seeing if the X does indeed have the same XPUs as the X Pro, since the exploit tries to disable all of these. Through trial-and-error we were able to figure out that trying to disable a non-existant HWIO_OCIMEM_MPU_XPU was indeed what caused the problem. After adding all the necessary elements and files to a docker container, we have a tool that can automatically root and setup a COTS Aquaris X device for TrustZone debugging in about 5 minutes. (see Figure 2)

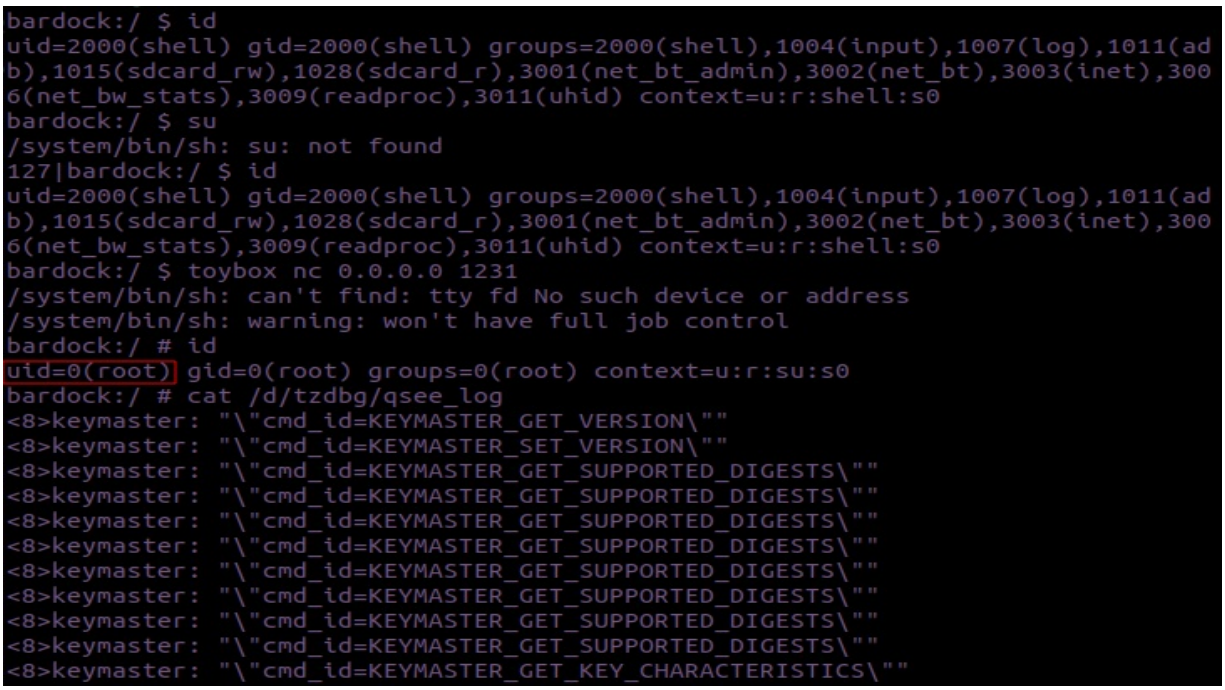

Figure 2: Huawei P9 Lite with modified Fastboot image

In terms of further steps, given that our research showed which points of the exploit are variable, one could construct a program, that adapts this knowledge to various devices, broadening the use of the tool.

HiSilicon

Background and previous research

In Figure 3 we can see the bootchain of the Kirin 650, 659a, 710 (Figure 3(left)) and the Kirin 990 (Figure 3(right)) SoCs. [3]

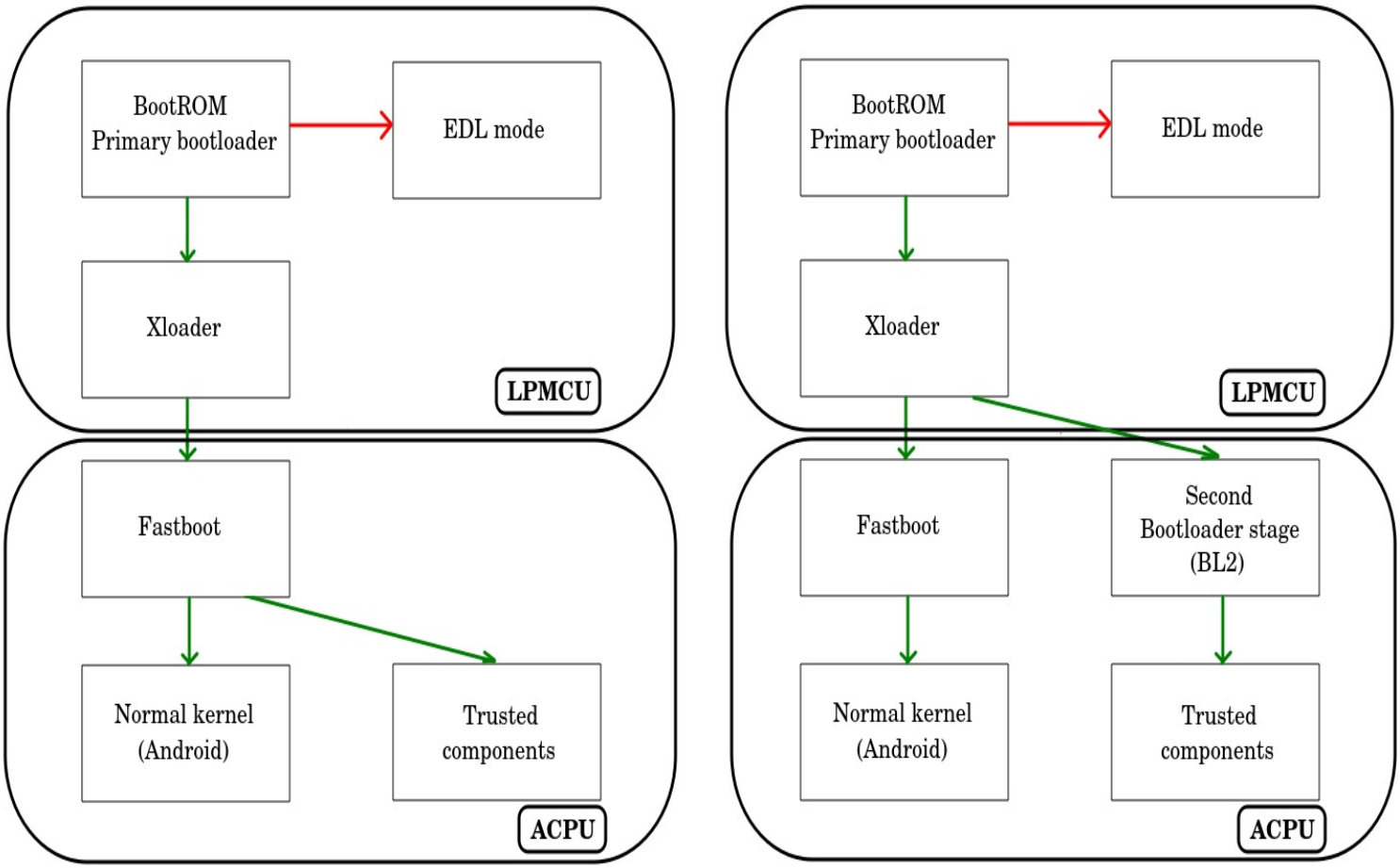

Figure 3: High-level description of HiSilicon bootchain for the researched devices. LPMCU stands for Low-Power Microcontroller Unit, a smaller chip, only used for boot. APCU stands for Application Central Processing Unit, this is the main processor.

For these devices triggering the EDL mode requires shorting a test point, and connecting the phone to a computer via USB. The phone then shows up as a USB serial device. In EDL mode, which as stated earlier is designed to repair bricked phones, the phone expects a xloader image, which in turn expects a fastboot image. Once in fastboot, we can flash correct images to repair the phone, or modified images for more nefarious purposes.As we can see in Figure 2 both the Kirin 990 and the earlier chipsets have a similar bootchain. The components verify their successors, so if we manage to break the chain of trust, we can take over the phone.

Since the xmodem protocol plays an important role in the vulnerabilities that we exploited, here is a brief overview of said protocol. [7] When using xmodem, the file to be sent is broken into blocks, and a crc checksum is added to help detect transmission errors. The device we are sending to replies

to each block sent, with either:

- 0xAA (ACK)
- 0x55 (NACK)
- 0x07 (Error)

There are four different types of blocks:

- Head (command: 0xFE)
- Data (command: 0xDA)
- Tail (command: 0xED)
- Inquiry (command: 0xCD)

The head blocks communicate where the file should be written, and what size it is. The tail block ends the transmission. The inquiry block, read a status code from a given address that varies per chipset. This can also be misused to leak data from the phone. The vulnerability discovered by Komaromy and Szabo uses a programming error in the xmodem implementation in the BootROM. It also takes advantage of the fact that the device does not erase unverifiable data that it receives, but instead places it at the given address, verifies the data, fails, and starts again. The implementation of xmodem in the devices we looked at verifies once a head block is sent, that the address we're trying to write to is legitimate. Unfortunately if we send a second head block, the address encoded in this second block is not verified, thus we are able to write to any page in the SRAM. This gives us the possibility to load and run code on the phones. It is interesting to note that HiSilicon/Huawei chose not to fix the head resend vulnerability, and instead took security measures like encrypting the various components.

Discussion

The first step in our research was to reproduce the exploit used for dumping the BootROM of the four devices. This involves using the head resend bug to send a small shellcode to the phone. The phone keeps executing the shellcode and leeking 1 or 4 bytes to an external device via the inquiry address in the xmodem protocol. We were able to dump the BootROM of the P9, the P20 and the P40. The PSmart device we used was heavily damaged, and shall be discussed later. It is interesting to note that a previous student had dumped the BootROM of the P40, and the result differed in 4 bytes to the one we managed to dump. In order to rule out any accidental errors, we dumped the BootROM again, and found the same 4 byte discrepency, at the same address. This unfortunately lowers confidence in any code that we dumped via the inquiry byte(s). Since each dump takes about 6 hours, we decided to leave the task of verifying this for future researchers.

Having dumped the BootROM, and thus gaining better understanding of the code and the head-resend bug, we set out to load a modified xloader image onto the P9. There were two possible

paths to do this: either use the head resend bug to send a modified xloader and get it executed, or reverse engineer the verification process in the BootROM and try to construct a modified image that still can be verified. At first, the head resend bug did not work. In retrospect, this was probably due to an altered USB connection on my laptop. It is also important to note that the xmodem protocol, in this particular instance, is not deterministic, and may produce errors or status codes that differ from one execution to another. The next attempt was to reverse engineer the verification process. This consists of two main functions in the case of the P9, one that verifies the header page (0x1000) and one that verifies the rest of the code. Unfortunately these functions are fairly convoluted, hard to reproduce, and involve values/addresses to which we have no access. Nevertheless, the insights gained here would come in handy while reverse engineering the xloader image, since this has a similar xmodem protocol and verification code. The third attempt was to use the Flash Patch and Breakpoint Engine (FPB). This is a tool designed to patch read-only memory, used for OTA updates. In short, one can setup a breakpoint at a certain line in the bootROM code, and when this line is hit, code execution jumps into a debug handler. If we are able to setup the debug handler ourselves, this opens up a whole new array of possibilities. Unfortunately, triggering the breakpoint never worked, so we had to give up on this angle. But once again, the knowledge gained here would prove useful later, while working on the P40 device. Ultimately, we turned back to our original idea of using the head resend bug. Until now, we had been using a script from a previous project, as well as the xmodem script provided for the HiKey 620 development board. [1] First we re-implemented and cleaned the script running on the computer. This involved risking an infinite loop in order to control more closely the return value of the xmodem protocol. Finally this worked, and we were able to send a modified xloader to the phone, without detection.

Figure 4a

```
int verify(undefined4 param_1,undefined4 param_2,undefined4 param_3){
  int iVar1;
  int iVar2;

  iVar2 = DAT_00021214 + 0x211b6;
  iVar1 = FUN_00027438(0x10000000,DAT_00021208,DAT_0002120c,param_1,iVar2,param_2,param_3);
  if (iVar1 != -1) {
    if (iVar1 == -0xf) {
      iVar1 = 0;
    }
    else {
      memcpy(DAT_00021210,0x10000000,0x1000);
      memcpy(0x10000000,0x10001000,DAT_0002120c);
      iVar1 = inner_verify(DAT_00021210,0x10000000,param_1,0xf,DAT_00021218 + 0x211f2,iVar2);
    }
  }
  return iVar1;
}
```

Figure 4b

```
int verify(undefined4 param_1,undefined4 param_2,undefined4 param_3){
  int iVar1;

  iVar1 = FUN_00027438(0x10000000,DAT_00021208,DAT_0002120c,param_1,DAT_00021214 + 0x211b6,param_2,
                       param_3);
  if (iVar1 != -1) {
    if (iVar1 == -0xf) {
      iVar1 = 0;
    }
    else {
      memcpy(DAT_00021210,0x10000000,0x1000);
      memcpy(0x10000000,0x10001000,DAT_0002120c);
      iVar1 = 0;
    }
  }
  return iVar1;
}
```

Figure 4: Decompiled verify function from the xloader image, 4a is the standard version, 4b is with the verification bypass

Next, we had to reverse engineer the xloader image, in order to figure out to how to bypass or annul the verification and load a modified fastboot. Simply not entering the verify function shown in Figure 4 is not a solution, since the function also take care of copying the loaded fastboot code and placing it in the right address for the APCU. The solution was to avoid going into the inner_verify function, and setting the return value to 0, as this was the expected value in order to continue execution. This changed the disassembled code to what we see in Figure 4b.

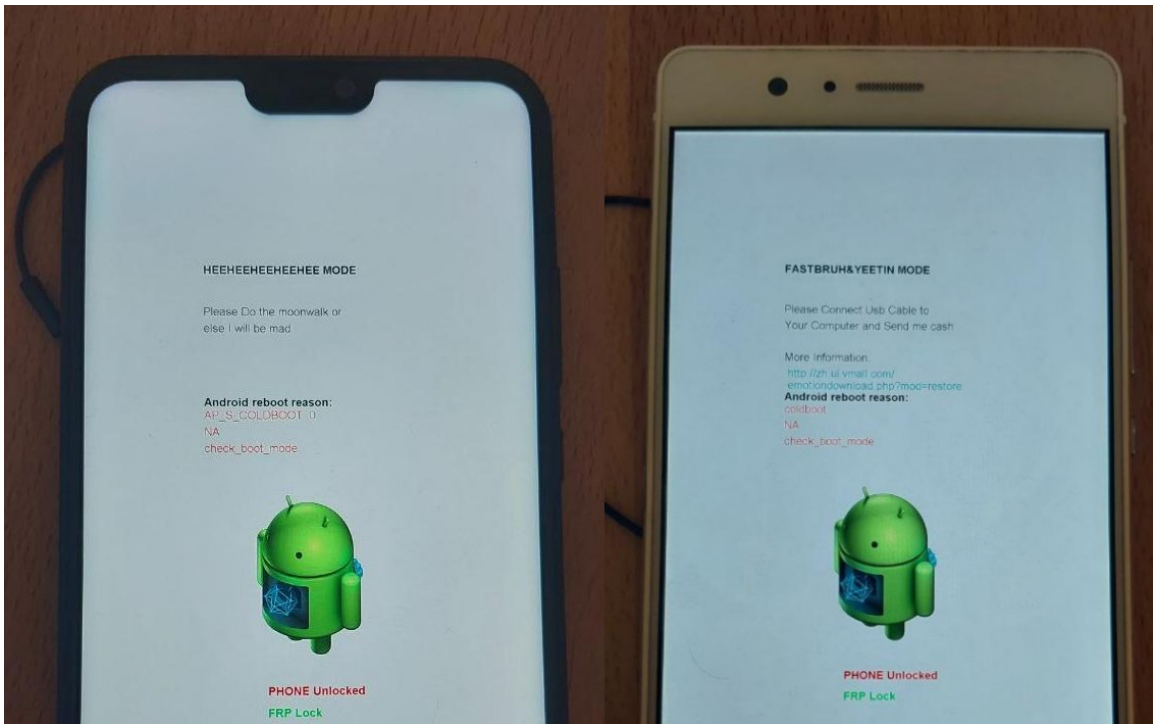

Figure 5: P20 Lite (left) and P9 Lite (right) with modified fastboot images

In Figure 5 we can see that indeed we managed to load a modified fastboot image. Since the chipset of the P20 is very similar to that of the P9, adapting the exploit was easy.

Deciding to take a wide, rather than a deep approach, we set out to load modified xloader/fastboot images on other devices. First we attempted on the PSmart phone which has a Kirin 710 chip. This phone was heavily damaged, but we attempted nonetheless to use it, since it showed moderate, but confusing signs of life (i.e., it show up as a CD-ROM player when plugging into the computer). To make sure that we were on the right track, we tried dumping the BootROM, but we kept getting 00 bytes. Having ruled out a problem with a faulty USB connection, we set out to "be creative, and try some more stuff". Unfortunately, while trying to boot into a legitimate fastboot, the device kept

hanging and becoming unresponsive. This routinely occurred during the xmodem protocol, after having sent 0x5000 bytes of the xloader, and always at the same address. In an attempt to gain more information as to the nature of the problem, we tried playing with the timing of the sends.

	Expected	Actual with no modifications	3 sec delay	Disable re-enumerating
Payload	0xAA	0xAA	0xAA	0xAA
	0xAA	0xAA	0xAA	0xAA
	0xAA	0xAA	0xAA	0xAA
Head resend	0xAA	0xAA	0xAA	0xAA
	0x07	0x07	0x07	0x55
	0xAA	0x07	0xAA	0x55
	0xAA	hangs	0xAA	0x55
Dump	BootROM		hangs	useless bytes

Figure 6: P20 Lite (left) and P9 Lite (right) with modified fastboot images

As it can be seen in Figure 6, a delay of 3 seconds lets us send one block more, but nothing else. We tried not re-enumerating the device after each file, but this produced NACK return values, which never occur in successful executions. It is important to note, that according to Komaromy and Szabo, [7] the sequence OxAA, Ox55, Ox55 actually means that the device is not vulnerable to the head resend bug, but given the damages the phone had sustained, we cannot concur that this is really the case.

We then decided to check if the address values we were using were perhaps wrong, since these came from single, unofficial sources. However, after writing to a large number of addresses in SRAM, we were not able to conclude anything in this regard. The addresses we used were the following:

- xmodem base address: 0x22000 (this implies executing code from 0x23000)
- rip address of xmodem function: 0x49BC8, this must be overwritten to jump to the unverified code we previous sent
- top of stack: 0x49BFC, used to assess where we can store code/data without disrupting the execution flow

Next, we moved on to the P40. After successfully dumping the BootROM, we tried working on the xloader. This device presents a particular difficulty, since this is the first phone, where the xloader itself is encrypted. To circumvent this, Komaromy and Szabo propose to use the phone as a decryption oracle, loading the xloader, waiting for it to get decoded and then dumping it. [7] They draw the outlines for the payload with which this should be possible, and they claim that they managed to dump the decrypted xloader. Unfortunately their payload did not work on the P40, but this could be due to the fact that the used a Mate 30 Pro device, even though both devices use the Kirin 990 chipset. I had access to the work of a previous student, who claims to have been able to dump the xloader from the same P40 device I had access to, but even with access to his payload the exploit did not work. Since this was the first payload that made use of the FPB, we used our previously gained knowledge to try and debug the payload. For the sake of sheer amusement, here are some of the things we experienced:

- Commenting out a seemingly vital line of code while setting up the breakpoint has no effect
- The xmodem protocol randomly refusing payloads (if the payload was of a certain length, it got refused 19 out of 20 times)
- Going into an infinite loop even though the code would not warrant this

Wanting to make sure the faults didn't occur because of hard faults, we took the time to test the boundaries of the SRAM. Based on our experience it ranges from 0x20000 to 0x67000. Thus, this did not seem to be the problem. A more complete list of the tried strategies is available on request.

Forced to renege on our strategy to go wide, we next attempted to make use of our control over the fastboot image on the P9 device, in order to try and load a modified kernel. For this we had to reverse engineer the fastboot, which proved more challenging than the xloader image. Using Ghidra to disassemble the file, we found that many strings were never referenced, even though they are clearly used and many functions appear to not be called. Since the Fastboot is the first component loader by the ACPU, we were also uncertain where to set the base address. Having flashed unsafe images onto the phone using the Fastboot protocol, we tried if any of the possible Fastboot commands could help us boot up the phone, (i.e., fastboot continue, or fastboot boot <modified_boot.img>. However we quickly discovered that Huawei didn't actually support these commands. According to our current understanding, it is necessary to figure out how the xloader calls into the fastboot (with which parameters, global variables, etc.). A few attempts at modifiying an xloader image to boot straight up were unsuccessful, and it is unclear at this time, whether this is even possible. Overall there are two fronts where one could continue in the direction of this project. First, it must be further investigated if it is possible to boot into a modified kernel, from the fastboot under our control. If yes, then it would be interesting to broaden the research to other models of Huawei phones.

Related Work

The most important related scientific work has been previously stated, as our project builds on them. There is a BootROM exploit for MediaTek devices, detailed in this article. [10] This might be a topic of further research. Regarding scientific papers the last few years have been rather stale in regards of this particular topic, further highlighting the need for more research. Of course there have been papers on TEEs, and other trusted components, [8], [4] but none of them deal in-depth with the effect on security research, or attempt to look at more than one vendor. One will note, that other than the occasional whitepaper, all relevant sources are either online blog posts, articles or GitHub repositories. Unfortunately, most of these lack rigor and precision, and as such information stated in these sources should be taken with a grain of salt. To produce work of scientific quality, one must test the claims made in these sources, and not build upon them blindly.

Conclusion

This project has two main results. Firstly, automatizing a debugging and rooting setup for select Qualcomm devices, and noting the parameters necessary to broaden the scope of tool to other devices. Secondly, attempting to reproduce and test exploits from the TASZK security lab on HiSilicon devices, and in case they worked, use them to load unverified components. A script framework has been setup, in order to do this more efficiently. It is clear that further research is need to determine if a full unverified kernel can be loaded, and if the encryption Huawei does on some of the components makes the just harder or impossible.

Bibliography

[1] 96boards. *Xmodem script for HiKey 620 development board*. URL: https://raw.githubusercontent.com/96boards/burn-boot/master/hisi-idt.py.

[2] Elouan Appere. *Analysis of Qualcomm Secure Boot Chains*. 2019. URL: https://blog.quarkslab.com/analysis-of-qualcomm-secure-boot-chains.html.

[3] Dominik Braun. "On the Dynamic Analysis of COTS Mobile Device Bootloaders". MA thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2022.

[4] Marcel Busch, Johannes Westphal, and Tilo Mueller. "Unearthing the TrustedCore: A Critical Review on Huawei's Trusted Execution Environment". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. URL: https://www.usenix.org/conference/woot20/presentation/busch.

[5] R. Hay and N. Hadad. *Exploiting Qualcomm EDL Programmers*. 2018. URL: https://alephsecurity.com/2018/01/22/qualcomm-edl-1.

[6] Bjoern Kerler. *Bring Light To The Darkness P3*. 2020. URL: https://bkerler.github.io/2020/08/03/bring-light-to-the-darkness-p3.

[7] D. Komaromy and L. Szabo. *How To Tame Your Unicorn: Exploring And Exploiting Zero-Click Remote Interfaces of Huawei Smartphones*. Tech. rep. 2021. URL: https://i.blackhat.com/USA21/Wednesday-Handouts/US-21-Komaromy-How-To-Tame-Your-Unicorn-wp.pdf.

[8] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "Armageddon: Cache attacks on mobile devices." In: *USENIX Security Symposium*. 2016, pp. 549–564.

[9] Counterpoint Research. *Global Smartphone AP (Application Processor) Shipments Market Share*. 2023. URL: https://www.counterpointresearch.com/global-smartphone-ap-market-share/.

[10] tinyhack.com. *Dissecting a MediaTek BootROM exploit*. URL: https://tinyhack.com/2021/01/31/dissecting-a-mediatek-bootrom-exploit/.