Beyond Root

Custom Firmware For Embedded Mobile Chipsets
Biography

Christopher Wade

Security Consultant at Pen Test Partners

@Iskuri1
https://github.com/lskuri

https://www.pentestpartners.com
Project Origin

Smartphones contain a huge amount of closed firmware

This limits the capabilities of even rooted devices

By breaking firmware protections and reverse engineering embedded chipsets, smartphones can be used as attack tools
Wi-Fi Monitor Mode

Many smartphones support Wi-Fi Monitor Mode

Activated in Snapdragon chipsets via:

    echo 4 > /sys/module/wlan/parameters/con_mode

Broadcom chipsets can utilise custom firmware

Well known, implemented in modern mobile testing tools
USB Device Emulation

Linux Kernel supports emulating USB devices via GadgetFS

This can be used to emulate any standard USB device

Rarely used, but very effective
Debian Chroot

A full Debian Root Filesystem can be generated with qemu-debootstrap.

A simple script can provide hardware access and direct SSH connectivity:

```bash
mount -o remount,rw /data
mount --bind /proc /data/debian_arm64/proc
mount --bind /sys /data/debian_arm64/sys
mount --bind /dev /data/debian_arm64/dev
mount devpts /data/debian_arm64/dev/pts -t devpts
chroot /data/debian_arm64/ /bin/bash --login -c /usr/sbin/sshd &
```
NFC on Android is restricted to very specific features:

- **Generic Reader Modes**
- **Mobile Payments**
- **NDEF Communication**
- **Host-Card Emulation**
NFC On Android – Unsupported Functionality

Desired features for an NFC attack tool:

Reader Based Attacks

Raw Tag Emulation

Passive Sniffing
Target Device

Samsung S6 - SM-G920F

Older smartphone – readily available

Allows for OEM unlocking and deployment of Custom ROMs

Found to use a proprietary Samsung Semiconductor NFC Controller in non-US versions
NFC Controller – S3FWRN5

Custom chip developed by Samsung Semiconductor

Utilised in non-US Samsung S6, and Note 4 devices

Boasts the ability to securely update firmware

Utilises ARM SC000 SecurCore architecture

Communicated with via I2C and GPIO on phone
Basic Communication – Hardware On Android

Smartphones are essentially embedded Linux devices

GPIO and I2C communication can be performed via files in “/dev/i2c-*” and “/dev/gpio*”

Samsung’s Kernel abstracts these to custom driver, accessed using device file “/dev/sec-nfc”

File reads, writes and IOCTLs control the chip
NCI Communication

NFC chips communicate via a standard protocol

This abstracts and restricts NFC functionality, to simplify the process

Send and receive packets consist of the following:

GID – Byte containing identifier of functionality group (Core, RF, Vendor Specific)
OID – Byte containing identifier of specific operation
Length – Byte containing the length of parameters
Payload – Data related to the operation
NCI – Non Standard Functionality

Vendor GID (0xf) allows for any non-standard functionality to be implemented

Vendor operations from 0x00-0xff can be enumerated by checking error responses

Vendor defined operations are most likely to contain actionable weaknesses

In addition, configuration and mode operations allow for non-standard functionality
# S3FWRN5 – Firmware Updates

S3FWRN5 chip supports firmware updates via I2C

Firmware updates are never implemented via NCI, a custom bootloader is used

Loaded from firmware files are found in vendor partition

| 00000000 | 32 30 31 35 31 31 31 38 | 31 32 30 34 ff ff ff ff | 00000010 | Z2 00 00 02 00 00 00 00 | 00 00 00 00 00 00 00 00 |
| 00000020 | 1d 00 00 00 ac 00 00 00 | 00 00 00 00 00 00 00 00 |
| 00000030 | 1e 63 31 91 18 c7 bc b4 | 05 4d 38 e1 00 0e be c0 03 |
| 00000040 | 4f 0f 75 3c 09 09 02 41 | 42 eb 7c cb 10 11 7b 24 |
| 00000050 | dc 0b a9 72 f8 a8 7c 82 | 8c 08 00 ed 00 00 00 00 |
| 00000060 | 85 3f c6 82 f1 7a 85 6b | 21 03 01 09 55 51 dc 83 |
| 00000070 | 85 d1 a0 12 bf ca 06 0b | be 5b ad 75 4d 74 dd d2 |
| 00000080 | 40 82 88 6c 4f 68 44 99 | 41 69 ca d6 13 e8 2a 1c |
| 00000090 | 78 f3 87 00 00 f7 39 5f | a9 2b 5b e9 45 76 3c de |
| 000000a0 | a2 31 75 49 76 2e 9f 3f | 3d 3b f1 6f 41 09 6f 65 |
| 000000b0 | e1 eb d0 54 22 f4 7c 96 | fa 4a f7 41 64 5a 46 97 |
| 000000c0 | e9 88 f1 b0 37 eb 1c a2 | e1 54 16 63 e3 55 12 0a |
| 000000d0 | de 8d 89 58 07 07 bf c1 | 4e 9c bf f1 03 0f 44 8e |
| 000000e0 | ff f4 13 8c 9d 1d 32 af | 49 8c 9a 4b 63 22 11 |
| 000000f0 | c7 6a 89 e2 ff d7 10 24 | a4 6a 4e 65 5a 35 b0 12 |
| 00001000 | 43 6e 7b 0a db 70 54 09 | f7 a5 3c df e8 59 9e 62 |
| 00001010 | ac 9e 97 61 01 67 e1 dc | 91 15 a6 64 0e 0f a9 e9 |
| 00001020 | 9b 66 6b 9d f8 5a 48 ed | d8 0d a7 0a 0f ff ff ff |
| 00001030 | 22 00 00 00 02 00 ff ff ff | 21 30 00 00 00 30 00 00 |
| 00001040 | ff ff ff ff ff ff ff ff | ff ff ff ff 04 48 80 f3 |
Enabling Debug Mode

*.rc configs can be modified in /system/

Debug and forced firmware updates can be enabled

Traces can be pulled from Logcat
Analysis Of Firmware Update Protocol

Update traces can be pulled from Logcat

Utilises four byte header followed by payload:

- **0x00**: Command type
- **0x01**: Command
- **0x02-0x03**: Payload size
- **0x04-0x100**: Payload data

0x80 is added to first byte on alternating sends
Firmware Update Files

Firmware and configs can be found in Android Filesystem

Depending on device version, can be in main system image or hidden Vendor partition

Usually available from publicly available Android images

```
./system/etc/nfc/THL/sec_s3fwrn5p_rfreg.bin
./system/etc/nfc/sec_s3fwrn5p_rfreg.bin
./system/vendor/firmware/nfc/sec_s3fwrn5p_firmware.bin
./system/vendor/firmware/sec_s3fwrn5p_firmware.bin
```
S3FWRN5 Firmware File Analysis

Basic format: metadata, signature, and full firmware

Payload provides size information about internal memory of device
Firmware Update Files – Identifying Architecture

Simple mnemonics can be used to identify chip architectures

Thumb’s “BX LR” operation translates in hex to “0x70 0x47”, and in ASCII to “pG”

A high number of instances of this imply Thumb code in use

This was identified in the firmware
Implementing Firmware Updates

Dump the Firmware Update protocol command sequence

Send dumped IOCTL and commands in sequence

Compare received values for each command

Header files from Open Source Kernel drivers can aid this: “sec_nfc.h”
Firmware Update Protocol and Sequence

Utilises numbered commands for firmware updates:

0: Reset
1: Boot Info
2: Begin Update
4: Update Sector
5: Complete Update

A numbered command is missing from the sequence

This heavily implied additional hidden commands
Identifying Hidden Bootloader Commands

Commands only work at certain stages of update process

Chip returns error 2 if command is not valid at that stage

Chip returns error 9 if the payload is too small

This can be brute forced through the firmware update protocol
Hidden Bootloader Command 3

Same functionality as command 4

 Writes 512-byte blocks instead of 4096

No actionable weaknesses
Hidden Bootloader Command 6

Takes eight bytes of parameters, two 32-bit values

Individual bits were set in parameters and responses were checked

Testing showed this allowed for reading of arbitrary memory – address and size

This allows for dumping of RAM, the firmware and the secure bootloader
Dumping The Bootloader

Memory can be stitched from hidden command 6

This showed a standard Cortex-M firmware format starting at address 0x00000000 (vector table followed by code), with a size of 8KB

This allowed for static analysis and emulation

The firmware contained no strings, drastically increasing time to analyse
Analysing Bootloader Binary

Loaded into IDA as ARM Little-endian

Memory Layout:

0x00000000 – Flash Memory
0x20000000 – RAM
0x40000000/0x50000000 – Hardware Peripherals
0xE0000000 – System
Analysing Bootloader Binary
Bootloader Artefacts

On start-up, the bootloader checks for a magic number at address 0x3000:

0x5AF00FA5

This magic number is only written if the signature is valid during upgrade.

Attempts to manually write the value were unsuccessful – first block must start with 0xFFFFFFFF.
Bootloader Artefacts

Bootloader commands can be swiftly identified for analysis
Bootloader Artefacts

RSA Public Key can be found in memory

0x80 high entropy bytes followed by “00 01 00 01” – 65537 as exponent
Identifying Memory Corruption

Fuzzing any embedded firmware could irreparably damage the chip

Only one phone was available for testing

Debugging and analysis via I2C would be difficult

Emulation of the bootloader was attempted
Emulating Embedded Firmware With Unicorn Engine

Library for emulating architectures and hooking all functionality

Can define architecture, memory mapping, and hardware integration
Emulating Embedded Firmware With Unicorn Engine

Bootloader was loaded at address 0x00000000

Program Counter was set to value in reset vector (0x000002BD)

Memory was mapped for flash, RAM and hardware registers

```
uc_mem_map(uc, 0x0, 0x3000, UC_PROT_ALL);
uc_mem_map(uc, 0x3000, 0x1000, UC_PROT_ALL);
uc_mem_map(uc, 0x00400000, 0x2000, UC_PROT_ALL);
uc_mem_map(uc, 0x20000000, 0x2000, UC_PROT_ALL);
uc_mem_map(uc, 0x40000000, 0x4000, UC_PROT_ALL);
uc_mem_map(uc, 0x50000000, 0x4000, UC_PROT_ALL);
uc_mem_map(uc, 0xe0000000, 0x4000, UC_PROT_ALL);
```
Emulating Embedded Firmware With Unicorn Engine

Commands are received in infinite loop in main thread, with no interrupts

This meant that emulation would be a simpler task

```assembly
; START OF FUNCTION CHUNK FOR sub_330

loc_13EA          ; CODE XREF: sub_330+2C+j
    LDR           R4, =byte_200000C4

loc_13EC          ; CODE XREF: sub_330+10C2+j
    LDR           R1, [R4,#(dword_200000CC - 0x200000C4)]
    LDR           R0, [R4,#(dword_200000D0 - 0x200000C4)]
    BLX           R1
    B             loc_13EC

; END OF FUNCTION CHUNK FOR sub_330
```
Emulating Embedded Firmware With Unicorn Engine

Execution was found to cause device resets when accessing hardware registers during configuration.

The bootloader image was patched to bypass hardware initialisation.

Static hardware register values were dumped from the chip and loaded into Unicorn.
Emulating Embedded Firmware With Unicorn Engine

The firmware was allowed to run, until it hit a hardware register

This was a read at address 0x40022030

The disassembly showed specific bits were checked

This implied it was a status register for I2C

The read was overridden to return random data
Emulating Embedded Firmware With Unicorn Engine

Next, the firmware continually read bytes from a single address - 0x40022038

This implied it was the I2C FIFO buffer

Firmware update commands were sent via this register

Responses to commands were sent to address 0x40022034

This constituted full emulation of the I2C communication
Memory Corruption Opportunities

Randomised fuzzing would now be viable

Commands have 16-bit sizes – larger than entire contents of RAM

Some commands send additional data in chunks

Size of hash and signature are defined in initialisation command

{0x80, 0x02, 0x04, 0x00, 0x14, 0x00, 0x80, 0x00}
Bypassing Signature Checks

Manipulation of the hash and signature sizes allowed for more data to be sent in chunks

Analysis in Unicorn showed that this caused out of bounds memory access

Further analysis showed that this overwrote the stack
Bypassing Signature Checks

Overwriting the stack allowed for manipulation of Program Counter

SC000 chipsets cannot execute from RAM

Stack was too small for complex ROP exploits

Program Counter was set to just after signature check:

0x016d (PC + 1 for Thumb code)
Bypassing Signature Checks

The exploit was performed on the physical chip

This booted the main firmware without power cycling

The firmware was started and could be run, bypassing signature checking

This would allow for custom firmware to be developed

The vulnerability was disclosed to Samsung
Bypassing Signature Checks – Remediation Methods

Method 1:

Patch the bootloader from the main firmware, removing the buffer overflow

This could brick the chip, as the core bootloader would be overwritten

Method 2:

Patch the Kernel to disallow large hashes and signatures

Trivially bypassed by kernel modification or direct I2C access
Further Research - Samsung Semiconductor NFC Chips

Multiple NFC chips outlined on company website

<table>
<thead>
<tr>
<th>Core</th>
<th>Flash</th>
<th>RAM</th>
<th>Interface</th>
<th>Crypto</th>
<th>Product Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC000</td>
<td>160K</td>
<td>14K</td>
<td>3 SWP, PC, SPI</td>
<td>NFC/FeliCa</td>
<td>Samples Available</td>
</tr>
<tr>
<td></td>
<td>128K</td>
<td></td>
<td>3 SWP, PC</td>
<td>NFC</td>
<td>Mass Production</td>
</tr>
<tr>
<td></td>
<td>12K</td>
<td></td>
<td>3 SWP, PC</td>
<td>NFC</td>
<td>Mass Production</td>
</tr>
<tr>
<td></td>
<td>10K</td>
<td></td>
<td>3 SWP, PC</td>
<td>NFC</td>
<td>Mass Production</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Part Number</th>
<th>Core</th>
<th>Flash</th>
<th>RAM</th>
<th>Interface</th>
<th>Crypto</th>
<th>Product Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3NRN74</td>
<td>SC000</td>
<td>160K</td>
<td>14K</td>
<td>3 SWP, PC, SPI</td>
<td>NFC</td>
<td>Samples Available</td>
</tr>
<tr>
<td>S3NRN81</td>
<td>SC000</td>
<td>128K</td>
<td>10K</td>
<td>3 SWP, PC</td>
<td>NFC</td>
<td>Mass Production</td>
</tr>
<tr>
<td>S3NRN82</td>
<td>SC000</td>
<td>160K</td>
<td>12K</td>
<td>3 SWP, PC, SPI</td>
<td>NFC</td>
<td>Mass Production</td>
</tr>
<tr>
<td>SEN82AB</td>
<td>SC000</td>
<td>160K</td>
<td>12K</td>
<td>3 SWP, PC, SPI</td>
<td>NFC/FeliCa</td>
<td>Mass Production</td>
</tr>
</tbody>
</table>
Samsung Semiconductor NFC Chips – Identification In Phones

Device specifications do not always contain NFC chipsets

It is more accurate to identify the firmware filenames in Android images

Android images can be downloaded directly from online archives

The /vendor directory contains these firmware files

Occasionally, this is a separate partition
The /vendor directory always contains these firmware files

Occasionally, this is a separate partition
Further Research – S3NRN82

S3NRN82 was selected as the next target – latest available chipset

Multiple chip firmware revisions available

Found in Samsung Galaxy S9

S9 was purchased, and rooted using OEM unlocking and a Custom ROM
S3NRN82 – Firmware File

Same format as S3FWRN5

Initial Stack Pointer larger – more RAM

Reset Vector lower – smaller bootloader

Firmware size 32kB larger
Further Research – Replicating Vulnerability

Commands 3 and 6 were removed

A new command, 7, was identified to reboot the chip

New bootloader size implied that it had been modified

Lack of memory readout would force any exploitation to be blind

Signatures checks utilising SHA-1 were found to fail
Further Research – Replicating Vulnerability

I2C communication was no longer provided by Logcat.

A /proc/nfclog file was found which contained the sizes of commands in sequence.

From this, the change from SHA-1 to SHA-256 could be deduced.

This was verified by modifying the firmware update tool.
Further Research – Replicating Vulnerability

Nature of device crashes allowed for analysis of stack size

The entire stack could be overwritten with pointers into code memory
Further Research – Exploiting Vulnerability

Stack was filled an initial value (0x0001) via buffer overflow

NCI initialisation commands were sent to chip

If an NCI response was received, the exploit worked

If NCI response failed, the device was reset and the initial value incremented

Signature bypass succeeded at address 0x0165
Demo

starite:/data/local/s3rm82_bypass_example #./run
Further Research – Disclosure

Vulnerability was disclosed to Samsung

The vulnerability was patched on newly manufactured chipsets from April 2020

All future chipsets will not be vulnerable

Custom Firmware would still be viable for older devices
Patching Existing Firmware

Custom firmware could be written for any of these chips

An initial goal was to dump the S3NRN82 bootloader

The only method for accessing data would be via I2C

This would also facilitate debugging
Patching Existing Firmware

Unreferenced/blank memory in firmware can be used to store new code.

Compiled machine code can be patched in.

The oldest available firmware was found, and used as a base – found in a Galaxy S8 ROM.
Patching Existing Firmware

C functions can be compiled as a raw binary using "gcc -c"

Stack handling is performed as with normal compilation

Function relocation is not performed

No standard C libraries can be included
Patching Existing Firmware

In C, function calls are generated as Branch and Link Instructions

These can be directly patched in order to implement different functionality

This can completely override intended functionality
Branch And Link uses two’s complement relative addresses.

Using the function address and current address can allow for creation of new BL functions.

This can be directly patched over original BL functions.
Patching Existing Firmware

A build application for linking and relocation was developed, which directly patched firmware.

```c
int main()
{
    // perform relocations;
    // return 0;

    printf("Starting firmware build\n");

    printf("Doing test branch\n");
    // generate LL function(0x0,0x0000);
    // generate LL function(0,0x37a0);
    // exit(0);

    // get original firmware
    char *fp = open("/sec3/wrap_firmware_unmodified_firmware.bin", O_RDONLY);
    int readsize = read(fd, baData, baSize);
    printf("Original firmware size: %d\n", readSize);
    close(fd);

    // patched functions
    int ff = open("/sec3/wrap_firmware_modifed_firmware.bin", O_RDONLY);
    readSize = read(fd, baData, CUSTOM_FUNCTIONS_OFFSET + ReadSize);
    printf("Additional firmware size: %d\n", readSize);
    close(fd);

    for(int i = 0; i < readSize; i++)
    {
        // printf("\n\n");
        // relocate function calls
        performRelocations();
        // generate symbol pointers
        generateSymbolPointers();
```
Patching Existing firmware

The vendor-specific NCI command “2F 24” was selected for modification.

Its response was found by searching for “MOVS.*#0x24”

sub_11A76 was overridden to the new “getArbitraryMemory” function.

Writing of new firmware took ~20 seconds.

The new function could be expanded as needed.
Patching Existing firmware

To receive parameters, location of command in RAM must be found

A crafted NCI request was generated: 2F 24 04 FA CE FA CE

The parameters were searched through RAM, and address set in response payload

This could allow for parameters to be used in readout

```c
for(int i = 0x20000000 ; i < 0x20002000 ; i++) {
    uint8_t* ptr = i;
    if(ptr[0] == 0xfa && ptr[1] == 0xce && ptr[2] == 0xfa && ptr[3] == 0xce) {
        r0[0] = i&0xff;
        r0[1] = (i>>8)&0xff;
        r0[2] = (i>>16)&0xff;
        r0[3] = (i>>24)&0xff;
        break;
    }
}
```
S3NRN82 Bootloader

The patched firmware allowed for dumping of arbitrary memory

With this, the new bootloader was downloaded

This allowed for analysis of how the initial exploit worked at 0x0165

Exploit was modified to point to 0x0173
Custom Firmware – Tag Emulation

The hardware of the chip supports multiple protocols:
ISO14443a, ISO14443b and more

Access to hardware registers allow for arbitrary communication

A goal was to emulate a Mifare Classic tag in its entirety on the S9

A Proxmark was used for debugging
Custom Firmware – Tag Emulation

NCI commands to initialise device were dumped from phone and replayed

Unnecessary commands were removed

The NCI RF Discover command was modified to only act as ISO14443a tag
Custom Firmware – Tag Emulation

Initial reversing requires knowledge of functions and hardware in depth

Lack of any strings means that this would require inferring the purpose of functions manually

To begin, the ISO14443A SELECT command (0x93) was searched for in IDA: “CMP.*#0x93”

The first result provided immediate information:
Custom Firmware – Tag Emulation

Placing the phone on a reader allowed this to be verified

It was possible to use the patched I2C function to dump the entire hardware configuration

This corroborated the results from IDA

Reader commands could be read

Access to these registers would also allow for passive sniffing
ISO14443a enumeration occurs using the following information:

- ATQA – defined by NCI
- SAK – defined by NCI
- UID – randomised on phones, first byte always 0x08

These define tag type and unique identifier.

Via NCI, ATQA and SAK values are restricted to specific values.

Due to their purpose, these values were stored in individual hardware registers.
Custom Firmware – Tag Emulation - Enumeration

Via NCI, SAK and ATQA values were sent to the chip

Using the patched I2C command, a RAM dump was taken

The SAK and ATQA values were identified in RAM, and compared with IDA

This lead to a single function referencing hardware registers
Custom Firmware – Tag Emulation - Enumeration

This function was overridden, then called within the new function

Custom SAK, ATQA and UID values were added via hardware to replace initial values

Confirmation of this patch was performed using a Proxmark as a reader

```c
uint32_t potentialMemorySetup(uint32_t r0) {
    uint32_t (*setupFunction)(uint32_t) = (uint32_t (*)(uint32_t))*0x8847;
    uint32_t vol = setupFunction(r0);

    // override atqa and uid
    uint32_t* uidPtr = 0x400020034;
    uint32_t* atqaptr = 0x40002003c;
    uint32_t* sakPtr = 0x400020048;

    struct TagState* tagState = TAG_STATE_OFFSET;

    memcpy(sakPtr[0], tagState->tagHeader[0], 8);
    atqaptr[0] = 0x44000000;
    uidPtr[0] = 0x00000000;

    return vol;
}
```
Custom Firmware – Tag Emulation - Enumeration

Analysis via the Proxmark demonstrated that this was successful

This would allow for modification of enumeration information, but not full communication

```Shell
Architecture Identifier: AT91SAM7Sxx Series
Nonvolatile Program Memory Type: Embedded Flash Memory
proxmark3> hf 14a reader
iso14443a card select failed
proxmark3> hf 14a reader
uid : 67 c6 f4 a7 20 14 a7
ATQA : 00 44
SAK : 09 [2]
Field dropped.
proxmark3> hf 14a reader
uid : 67 c6 f4 a7 20 14 a7
ATQA : 00 44
SAK : 09 [2]
Field dropped.
proxmark3> hf 14a reader
uid : 67 c6 f4 a7 20 14 a7
ATQA : 00 44
SAK : 09 [2]
Field dropped.
```
Custom Firmware – Tag Emulation – Full Communication

Chip was known to respond to commands 0x50 (HALT) and 0xE0 (RATS)

RATS was searched via: “CMP.*#0xe0”

Four results were found, and analysed individually

This lead to finding the state machine functions

Additional valid commands were noted
Further tracing from RATS found the function which sent responses

This was found to set a buffer, size, and some configuration information

The written registers were copied and added to a new function
Custom Firmware – Tag Emulation – Full Communication

A basic read command was first implemented: 30 XX + CRC

This was configured to return unencrypted memory blocks

This could later be extended to include appropriate encryption

```
proxmark3> hf 14a raw -sc 30 00
Card selected. UID[7]:
67 C6 F4 A7 20 14 A7
received 18 bytes:
67 C6 F4 A7 20 14 A7 89 44 00 C2 00 00 00 00 00 00 80 50
```
Custom Firmware – Tag Emulation – Full Communication

The state machine function was overridden

A switch statement was used to respond to Mifare commands

Analysis showed that the HALT command affected the internal state machine

This function was called from the new state machine

Non-standard debugging commands were also added
Custom Firmware – Tag Emulation – Full Communication

With full control, any ISO14443a tag could be emulated

Mifare Classic’s Crypto-1 authentication and access mechanisms were implemented

While this worked with a Proxmark, it would not work on a legitimate reader
Custom Firmware – Tag Emulation – Restrictions

Mifare Classic encrypted communication overrides the parity bit of each communicated byte

The chip hardware was configured to auto-generate this parity bit

It was possible that a hardware register setting may allow for modifying parity bits

Each register was modified in turn, while responses were checked on a Proxmark
Custom Firmware – Tag Emulation – Restrictions

The parity register was found at address 0x40020004, by setting bit 0x4000

With this set, parity could be modified

This required adding additional bits to the buffer, and increasing the length set by one bit per byte

With this in place, a Mifare Classic tag could be fully emulated
Custom Firmware – Tag Emulation – Dumping Writes

Writes to tags were hooked to send I2C messages

This allowed for persistent modification of tags

```c
} else if((tagState->setupState == State_AwaitingWriteBlock) { 
    tagState->setupState = State_Selected; 
    tmemcpy(&tagState->TagHeader[tagState->blockwrite][0x16],&cmd[0],16); 
    sendack(tagState); 
    // writes data pack 
    void (*setupResponseHeader)(uint32_t,uint32_t) = (void (*)(uint32_t,uint32_t))0x10888; 
    setupResponseHeader(0x0f,0x09); 
    unsigned char* 12cBlock = 0x20000024; 
    12cBlock[2] = 0x11; 
    12cBlock[3] = tagState->blockwrite; 
    tmemcpy(&12cBlock[1],&cmd[0],15); 
    void (*sendKraftedNickResponse)(void) = (void (*)(void))0x13985; 
    sendKraftedNickResponse(); 
    // do write blocking here if needed 
} else if((tagState->setupState == State_AwaitingAuth) { 
```
Tag emulation allows for spoofing of 13.56MHz access control cards, as well as more esoteric uses

All other NFC functionality works as normal, despite patching

More subtle than a dedicated attack tool

Expansion of this functionality could allow for offline cracking attacks

The same emulation could be performed on any supported protocol

Now framework is in place, easy to develop for
Conclusion

All outlined vulnerabilities were patched by Samsung as of April 2020

The vulnerability required root access, but fully compromised the chip

Phones are exploitable embedded devices, and should be treated as such

Bootloader vulnerabilities are more common than you think, especially in phones

Developing custom firmware for proprietary chips is challenging, but rewarding

If an undisclosed vulnerability is found in an old chip, it’ll likely be in the new one