7 Patching Kosher Firmware for Nokia 2720

by Assaf Nativ
`D7 90 D7 A1 D7 A3 D7 A0 D7 AA D7 99 D7 91`
in collaboration with two anonymous coworkers.

This fun little article will introduce you to methods for patching firmware of the Nokia 2720 and related feature phones. We'll abuse a handy little bug in a child function called by the verification routine. This modification to the child function that we can modify allows us to bypass the parent function that we cannot modify. Isn't that nifty?

A modern feature phone can make phone calls, send SMS or MMS messages, manage a calendar, listen to FM radio, and play Snake. Its web browser is dysfunctional, but it can load a few websites over GPRS or 3G. It supports Bluetooth, those fancy ringtones that no one ever buys, and a calculator. It can also take ugly low-resolution photos and set them as the background.

Not content with those unnecessary features, the higher end of modern feature phones such as the Nokia 208.4 support Twitter, WhatsApp, and a limited Facebook client. How are the faithful to study their scripture with so many distractions?

A Kosher phone would be a feature phone adapted to the unique needs of a particular community of the Orthodox Jews. The general idea is that they don't want to be bothered by the outside world in any way, but they still want a means to communicate between themselves without breaking the strict boundaries they made. They wanted a phone that could make phone calls or calculate, but that only supported a limited list of Hasidic ringtones and only used Bluetooth for headphones. They would be extra happy if a few extra features could be added, such as a Jewish calendar or a prayer time table. While Pastor Laphroaig just wants a phone that doesn't ring (except maybe when heralding new PoC), frowns on Facebook, and banishes Tweety-boxes at the dinner table, this community goes a lot further and wants no Facebook, Twitter, or suchlike altogether. This strikes the Pastor as a bit extreme, but good fences make good neighbors, and who's to tell a neighbor how tall a fence he ought to build? So this is the story of a neigbor who got paid to build such a fence.[5]

– – – – – – – – – – – – – – – – – – – – – – – – – – – –

I started with a Nokia phone, as they are cost effective for hardware quality and stability. From Nokia I got no objection to the project, but also no help whatsoever. They said I was welcome to do whatever helps me sell their phones, but this target group was too small for them to spend any development time on. And so this is how my quest for the Kosher phone began.

During my journey I had the pleasure of developing five generations of the Kosher phone. These were built around the Nokia 1208, Nokia 2680, Nokia 2720, Samsung E1195, and the Nokia 208.4. There were a few models in between that didn't get to the final stage either because I failed in making a Kosher firmware for them or because of other reasons that were beyond my control.

I won't describe all of the tricks I've used during the development, because these phones still account for a fair bit of my income. However, I think the time has come for me to share some of the knowledge I've collected during this project.

It would be too long to cover all of the phones in a single article, so I will start with just one of them, and just a single part that I find most interesting.

Nokia has quite a few series of phones differ in the firmware structure and firmware protection. SIM-locking has been prohibited in the Israeli market since 2010, but these protections also exist to keep neighbors from playing with baseband firmware modifications, as that might ruin the GSM network.

Nokia phones are divided into a number of baseband series. The oldest, DCT1, works with the old analog networks. DCT3, DCT4 and DCT4+ work with 2G GSM. BB5 is sometimes 2G and sometimes 3G, so far as I know. And anything that comes after, such as Asha S40, is 3G. It is important to understand that there are different generations of phones because vulnerabilities and firmware seem to work for all devices within a family. Devices in different families require different firmware.

[5] Disclaimer: No one forces this phone on them; they choose to have it of their own will. No government or agency is involved in this, and the only motivation that drives customers to use this kind of phone is the community they live in.

I'll start with a DCT4+ phone, the Nokia 1208. Nowadays there are quite a few people out there who know how to patch DCT4+ firmware, but the solution is still not out in the open. One would have to collect lots of small pieces of information from many forum posts in order to get a full solution. Well, not anymore, because I'm going to present here that solution in all of its glory.

A DCT4+ phone has two regions of executable code, a flashable part and a non-flashable secured part, which is most likely mask ROM. The flashable memory contains a number of important regions.

- The Operating System, which Nokia calls the MCUSW. (Read on to learn how they came up with this name.)
- Strings and localization strings, which Nokia calls the PPM.
- General purpose file system in a FAT16 format. This part contains configuration files, user files, pictures, ringtones, and more. This is where Nokia puts phone provider customizations, and this part is a lot less protected. It is usually referred to as the CNT or IMAGE.

All of this data is accessible for the software as one flat memory module, meaning that code that runs on the device can access almost anything that it knows how to locate.

`0x0084_0000` – `0x0090_0000`	Secured Rom
`0x0100_0000` – `0x01CE_0000`	MCUSW and PPM
`0x0218_0000` – `0x02FC_0000`	Image
`0x0300_0000` – `0x0400_0000`	External RAM
`0x0500_0000` – `0x0510_0000`	API RAM

At this point I focused on the operating system, in my attempt to patch it to make the phone Kosher. The operating system contains nearly all of the code that operates the phone, including the user interface, menus, web browser, SMS, and anything else the phone does. The only things that are not part of the OS are the code for performing the flashing, the code for protecting the flash, and some of the baseband code. These are all found in the ROM part. The CNT part contains only third party apps, such as games.

Obtaining a copy of the firmware is not hard. It's available for download from many websites, and also directly from Nokia's own servers. These firmware images can be flashed using Nokia's flashing tool, Phoenix Service Software, or with NaviFirm+. The operating system portion comes with a `.mcu` or `.mcusw` extension, which stands for MicroController Unit SoftWare.

This file starts with the byte `0xA2` that marks the version of the file. The is a simple Tag-Length-Value format. From offset `0xE6` everything that follows is encoded as follows:

- 1 Byte: Type, which is always `0x14`.
- 1 Dword: Address
- 3 Bytes: Length
- 1 Byte: Unknown
- 1 Byte: Xor checksum

Combining all of the data chunks, starting at the address `0x100_0000` we'll see something like this:

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000_0000	AD	7E	B6	1A	1B	BE	0B	E2	7D	58	6B	E4	DB	EE	65	14
0000_0010	42	30	95	44	99	18	18	38	DB	00	FF	FF	FF	FF	FF	FF
0000_0020	FF	FF	FF	FF	F8	1F	8B	22	50	65	61	4B	FF	FF	FF	FF
0000_0030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0000_0040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0000_0050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0000_0060	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	F8	C4	AA	C3
0000_0070	85	CF	C6	E7	00	04	8A	5F	01	00	01	00	00	00	00	00
0000_0080	00	00	00	00												

Note that some of these `0xFF` bytes are just missing data because of the way it is encoded. The first data chunk belongs to address `0x0100_0000`, but it's just `0x2C` bytes long, and the next data chunk starts at `0x0100_0064`. The data that follows byte `0x0100_0084` is encrypted, and is auto decrypted by hardware.

I know that decryption is done at the hardware level, because I can sniff to see what bytes are actually sent to the phone during flashing. Further, there are a few places in memory, such as the bytes from `0x0100_0000` to `0x0100_0084`, that are not encrypted. After I managed to analyze the encryption, I later found that in some places in the code these bytes are accessed simply by adding `0x0800_0000` to the address, which is a flag to the CPU that says that this data is not encrypted, so it shouldn't be decrypted.

Now an interesting question that comes next is what the encryption is, and how I can reverse it to patch the code. My answer is going to disappoint you, but I found out how the encryption works by gluing together pieces of information that are published on the Internet.

If you wonder how the fine folks on the Internet found the encryption, I'm wondering the same thing. Perhaps someone leaked it from Nokia, or perhaps it was reverse engineered from the silicon. It's possible, but unlikely, that the encryption was implemented in ARM code in the unflashable region of memory, then recovered by a method that I'll explain later in this article.

It's also possible that the encryption was reversed mathematically from samples. I think the mechanism has a problem in that some plaintext, when repeated in the same pattern and at the same distance from each other, is encrypted to the same ciphertext.

The ROM contains a rather small amount of code, but as it isn't included in the firmware updates, I don't have a copy. The only thing I care about from this code is how the first megabyte of MCU code is validated. If and only if that validation succeeds, the baseband is activated to begin GSM communications.

If something in the first megabyte of the MCU code were patched, the validation found in the ROM would fail, and the phone would refuse to communicate with anything. This won't interrupt anything else, as the phone would still need to boot in order to display an appropriate error message. The validation function in the ROM is invoked from the MCU code, so that function call could be patched out, but again, the GSM baseband would not be activated, and the phone wouldn't be able to make any calls. It might sound as if this is what the customer is looking for, but it's not, as phone calls are still Kosher six days a week. Note that Bluetooth still works when baseband doesn't, and can be a handy communication channel for diagnostics.

Another validation found in the MCU code is a common 16 bit checksum, which is done not for security reasons but rather to check the phone's flash memory for corruption. The right checksum value is found somewhere in the first `0x100` bytes of the MCU. This checksum is easily fixed with any hex editor. If the check fails, the phone will show a "Contact Service" message, then shut down.

At this point I didn't know much about what kind of validation is performed on the first megabyte, but I had a number of samples of official firmware that pass the validation. Every sample has a function that resides in that megabyte of code and validates the rest of the code. If that function fails, meaning that I patched something in the code coming after the first megabyte, it immediately reboots the phone. The funny thing is that the CPU is so slow that I can get a few seconds to play with the phone before the reboot takes place. Unfortunately, patching out this check still leaves me with no baseband, and thus no product.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000_0000	AD	7E	B6	1B	23	10	03	40	C6	05	E4	01	20	A2	00	00
0000_0010	00	00	00	00	00	00	00	00	00	00	00	FF	FF	FF	FF	FF
0000_0020	FF	FF	FF	FF	F8	1F	**AA**	**02**	50	65	61	4B	FF	FF	FF	FF
0000_0030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0000_0040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0000_0050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0000_0060	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	*C0*	*52*	*90*	*D4*
0000_0070	*4A*	*E4*	*5C*	*8F*	*00*	*02*	*00*	*00*	*01*	*00*	*01*	*00*	*00*	*00*	*00*	*00*
0000_0080	*00*	*00*	*00*	*00*	FF	FF	FF	FF	FF	FF	FF	FF	01	CE	00	00
0000_0090	03	00	00	00	00	04	CC	A2	00	04	CC	A3	FF	FF	FF	FF
0000_00A0	00	00	F1	EF	89	33	EB	2D	1F	09	3B	DA	C7	C0	3D	9F
0000_00B0	BB	D3	29	98	01	C8	BC	B0	06	6E	A8	11	0E	D1	69	67
0000_00C0	A4	A3	9A	A5	BF	7B	27	5A	E6	C7	61	2D	F7	B8	70	9C
0000_00D0	D4	1C	09	96	AF	5B	F2	05	20	92	49	DF	D5	0B	FC	DE
0000_00E0	A8	30	B7	39	34	59	13	7D	E7	BD	72	3F	C7	CF	B3	5A
0000_00F0	60	2C	5E	7D	63	17	56	C4	9F	6C	C5	1A	01	BF	**B5**	**CF**
0000_0100	**EA**	**01**	**FF**	**BE**	**00**	**FE**	**6A**	**84**	**EA**	**50**	**20**	**20**	**20**	**20**	**6A**	**04**
0000_0110	**2D**	**CF**	**20**	**20**	**20**	**20**	**6A**	**01**	**9D**	**7C**	**20**	**20**	**20**	**20**	**6A**	**01**
0000_0120	**B3**	**C8**	**20**	**20**	**20**	**20**	**6A**	**01**	**A5**	**C2**	**20**	**20**	**20**	**20**	**6A**	**04**

16 bit checksum. If this fails, the phone shows "Contact Service" message and shuts down.
If changed, the baseband fails to start and the phone shows no signal.
`These bytes can be freely changed.  They are likely version info and a public key.`

To attack this protection I had to better understand the integrity checks. I didn't have a dump of the code that checks the first megabyte, so I reversed the check performed on the rest of the binary in an attempt to find some mistake. Using the FindCrypt IDA script, I found a few implementations of SHA1, MD5, and other hashing functions that could be used—and should be used!—to check binary integrity.

Most importantly, I found a function that takes arguments of the hash type, data's starting address, and length, and returns a digest of that data. Following the cross references of that function brought me to the following code:

```
FLASH:01086266 loc_1086266                          ; CODE XREF: SHA1_check+1F6
FLASH:01086266                                      ; SHA1_check+1FC
FLASH:01086266              LDR     R2, =0x300C8D2
FLASH:01086268              MOVS    R1, #0x1C
FLASH:0108626A              LDRB    R0, [R2,R0]
FLASH:0108626C              MULS    R1, R0
FLASH:0108626E              LDR     R0, =SHA1_check_related
FLASH:01086270              SUBS    R0, #0x80
FLASH:01086272              ADDS    R0, R1, R0
FLASH:01086274              MOVS    R4, R0
FLASH:01086276              ADDS    R0, #0x80
FLASH:01086278 R1 = Start
FLASH:01086278              LDR     R1, [R0,#0xC]
FLASH:0108627A              LDR     R2, [R0,#0x10]
FLASH:0108627C              LDR     R0, [R0,#0xC]
FLASH:0108627E DataLength = DataStart - DataEnd;
FLASH:0108627E              SUBS    R3, R2, R0
FLASH:01086280              ADD     R2, SP, #0x38+hashLength
FLASH:01086282              STR     R2, [SP,#0x38+hashLengthCopy]
FLASH:01086284              LDRB    R0, [R6,#8]
FLASH:01086286 DataLength += 1;
FLASH:01086286              ADDS    R3, R3, #1
FLASH:01086288              ADDS    R7, R7, R3
```

```
FLASH:0108628A R2 = DataLength;
FLASH:0108628A                  MOVS    R2, R3
FLASH:0108628C                  ADD     R3, SP, #0x38+hashToCompare
FLASH:0108628E                  BL      hashInitUpdateNDigest_j
FLASH:0108628E
FLASH:01086292                  CMP     R0, #0
FLASH:01086294                  BNE     loc_10862A4
FLASH:01086294
FLASH:01086296                  LDR     R0, =hashRelatedVar
FLASH:01086298                  MOVS    R1, #1
FLASH:0108629A                  BL      MONServerRelated_over1
FLASH:0108629A
FLASH:0108629E                  MOVS    R0, #4
FLASH:010862A0                  BL      reset
```

The digest function is `hashInitUpdateNDigest_j`, of course. The `SHA1_check_related` address had the following data in it:

```
FLASH:01089DD4 SHA1_check_related DCD 0xB5213665       ; DATA XREF: SHA1_check:loc_108616A
FLASH:01089DD4                                          ; SHA1_check+9E ...
FLASH:01089DD8                  DCD 3
FLASH:01089DDC SHA1_check_info  DCD 0x200400AA          ; DATA XREF: SHA1_check+44
FLASH:01089DE0 #1
FLASH:01089DE0                  DCD loc_1100100         ; Start
FLASH:01089DE4                  DCD loc_13AFFFE+1       ; End
FLASH:01089DE8                  DCD 0xEE41347A          ; \
FLASH:01089DEC                  DCD 0x8C88F02F          ;  \
FLASH:01089DF0                  DCD 0x563BB973          ;   = SHA1SUM
FLASH:01089DF4                  DCD 0x040E1233          ;  /
FLASH:01089DF8                  DCD 0x8C03AFFA          ; /
FLASH:01089DFC #2
FLASH:01089DFC                  DCD loc_13B0000
FLASH:01089E00                  DCD loc_165FFFE+1
FLASH:01089E04                  DCD 0xCC29F881
FLASH:01089E08                  DCD 0xA441D8CD
FLASH:01089E0C                  DCD 0x7CEF5FEF
FLASH:01089E10                  DCD 0xC35FE703
FLASH:01089E14                  DCD 0x8BD3D4D6
FLASH:01089E18 #3
FLASH:01089E18                  DCD loc_1660000
FLASH:01089E1C                  DCD loc_190FFFC+3
FLASH:01089E20                  DCD 0x77439E9B
FLASH:01089E24                  DCD 0x530F0029
FLASH:01089E28                  DCD 0xA7490D5B
FLASH:01089E2C                  DCD 0x4E621094
FLASH:01089E30                  DCD 0xC7844FE3
FLASH:01089E34 #4
FLASH:01089E34                  DCD loc_1910000
FLASH:01089E38                  DCD dword_1BFB5C8+7
FLASH:01089E3C                  DCD 0xA87ABFB7
FLASH:01089E40                  DCD 0xFB44D95E
FLASH:01089E44                  DCD 0xC3E95DCA
FLASH:01089E48                  DCD 0xE190ECCA
FLASH:01089E4C                  DCD 0x9D100390
FLASH:01089E50                  DCD 0
FLASH:01089E54                  DCD 0
```

This is SHA1 digest of other arrays of binary, in chunks of about `0x002B_0000` bytes. All of the data from 0x0100_0100 to 0x0110_0100 is protected by the ROM. The data from 0x0110_0100 to 0x013A_FFFF digest to EE41347A8C88F02F563BB973040E12338C03AFFA under SHA1. So I guessed that this function is the validation function that uses SHA1 to check the rest of the binary.

Later on in the same function I found the following code.

```
FLASH:010862E0 for( i = 0; i < hashLength; ++i ) {
FLASH:010862E0
FLASH:010862E0 loc_10862E0                               ; CODE XREF: SHA1_check+1CC
FLASH:010862E0                 ADDS    R3, R4, R0
FLASH:010862E2                 ADDS    R3, #0x80
FLASH:010862E4                 ADD     R2, SP, #0x38+hashToCompare
FLASH:010862E6                 LDRB    R2, [R2,R0]
FLASH:010862E8                 LDRB    R3, [R3,#0x14]
FLASH:010862EA    if (hash[i] != hashToCompare[i]) {
FLASH:010862EA       return False;
FLASH:010862EA    }
FLASH:010862EA                 CMP     R2, R3
FLASH:010862EC                 BEQ     loc_10862F0
FLASH:010862EC
FLASH:010862EE                 MOVS    R5, #1
FLASH:010862EE
FLASH:010862F0
FLASH:010862F0 loc_10862F0                               ; CODE XREF: SHA1_check+1C4
FLASH:010862F0                 ADDS    R0, R0, #1
FLASH:010862F0
FLASH:010862F2
FLASH:010862F2 loop                                      ; CODE XREF: SHA1_check+1B6
FLASH:010862F2                 CMP     R0, R1
FLASH:010862F4 }
FLASH:010862F4                 BCC     loc_10862E0
FLASH:010862F4
FLASH:010862F6                 CMP     R5, #1
FLASH:010862F8 // Patch here to 0xe006
FLASH:010862F8
FLASH:010862F8                 BNE     loc_1086308
FLASH:010862F8
FLASH:010862FA                 LDR     R0, =0x7D0005
FLASH:010862FC                 BL      HashMismatch
FLASH:010862FC
FLASH:01086300                 MOVS    R0, #4
FLASH:01086302                 BL      reset
FLASH:01086302
FLASH:01086306                 B       loc_1086310
```

This function performs the comparison of the calculated hash to the one in the table, and, should that fail to match, it calls the HashMismatch() function and then the reset function with Error Code 4.

The HashMismatch() function looks a bit like this.

```
FLASH:01085320 ; Attributes: thunk
FLASH:01085320
FLASH:01085320 HashMismatch                              ; CODE XREF: sub_1084232+38
FLASH:01085320                                           ; sub_1085B6C+6C ...
FLASH:01085320                 BX      PC
FLASH:01085320
FLASH:01085320 ; ---------------------------------------------------------------------------
FLASH:01085322                 ALIGN 4
FLASH:01085322 ; End of function HashMismatch
```

```
FLASH:01085322
FLASH:01085324                 CODE32
FLASH:01085324
FLASH:01085324 ; =============== S U B R O U T I N E =======================================
FLASH:01085324
FLASH:01085324
FLASH:01085324 sub_1085324                              ; CODE XREF: HashMismatch
FLASH:01085324                 LDR     R12, =(sub_1453178+1)
FLASH:01085328                 BX      R12 ; sub_1453178
FLASH:01085328
FLASH:01085328 ; End of function sub_1085324
FLASH:01085328
FLASH:01085328 ; ---------------------------------------------------------------------------
FLASH:0108532C off_108532C     DCD sub_1453178+1        ; DATA XREF: sub_1085324
FLASH:01085330                 CODE16
FLASH:01085330
FLASH:01085330 ; =============== S U B R O U T I N E =======================================
FLASH:01085330
FLASH:01085330 ; Attributes: thunk
FLASH:01085330
FLASH:01085330 sub_1085330                              ; CODE XREF: sub_10836E6+86
FLASH:01085330                                          ; sub_10874BA+3C ...
FLASH:01085330                 BX      PC
FLASH:01085330
FLASH:01085330 ; ---------------------------------------------------------------------------
FLASH:01085332                 ALIGN 4
FLASH:01085332 ; End of function sub_1085330
FLASH:01085332
FLASH:01085334                 CODE32
```

Please recall that ARM has two different instruction sets, the 32-bit wide ARM instructions and the more efficient, but less powerful, variable-length Thumb instructions. Then note that ARM code is used for a far jump, which Thumb cannot do directly.

Therefore what I have is code that is secured and is well checked by the ROM, which implements a SHA1 hash on the rest of the code. When the check fails, it uses the code that it just failed to verify to alert the user that there is a problem with the binary! It's right there at `0x0145_3178`, in the fifth megabyte of the binary.

From here writing a bypass was as simple as writing a small patch that fixes the Binary Mismatch flag and jumps back to place right after the check. Ain't that clever?

How could such a vulnerability happen to a big company like Nokia? Well, beyond speculation, it's a common problem that high level programmers don't pay attention to the lower layers of abstraction. Perhaps the linking scripts weren't carefully reviewed, or they were changed after the secure bootloader was written.

It could be that they really wanted to give the user some indication about the problem, or that they had to invoke some cleanup function before shutdown, and by mistake, the relevant code was in another library that got linked into higher addresses, and no one thought about it.

Anyhow, this is my favorite method for patching the flash. It doesn't allow me to patch the first megabyte directly, but I can accomplish all that I need by patching the later megabytes of firmware.

However, if that's not enough, some neighbors reversed the first megabyte check for some of the phones and made it public. Alas, the function they published is only good for some modules, and not for the entire series.

How did they manage to do it, you ask? Well, it's possible that it was silicon reverse engineering, but another method is rumored to exist. The rumor has it that with JTAG debugging, one could single-step through the program and spy on the Instruction Fetch stage of the pipeline in order to recover the instructions from mask ROM. Replacing those instructions with a NOP before they reach the WriteBack stage of the pipeline would linearize the code and allow the entire ROM to be read by the debugger while the CPU sees it as one long NOP sled. As I've not tried this technique myself, I'd appreciate any concrete details on how exactly it might be done.

Now that I had a way to patch the firmware, I could go on to creating a patched version to make this phone Kosher. I had to reverse the menu functions entirely, which was quite a pain. I also had to reverse the methods for loading strings in order to have a better way to find my way around this big binary file.

Some of the patching was a bit smoother than others. For instance, after removing Internet options from all of the menus, I wanted to be extra careful in case I missed a secret menu option.

To disable the Internet access, one might suggest searching for the TCP implementation, but that would be too much work, and as a side effect it might harm IPC. One can also suggest searching for things like the default gateway and set it to something that would never work, but again that would be too much work. So I searched for all the places where the word "GET" in all capitals was found in the binary. Luckily I had just one match, and I patched it to "BET", so from now on, no standard HTTP server would ever answer requests. Moreover, to be on the extra, extra safe side I've also patched "POST" to "MOST". Lets see them downloading porn with that!

Be sure to read my next article for some fancy tricks involving the filesystem of the phone.