Satellite Baseband firmware modifications

Muñiz, Sebastian
smuniz@groundworkstech.com

Ortega, Alfredo
aortega@groundworkstech.com

Buenos Aires, Argentina

Abstract

In this article we show the analysis and modification of the firmware of the Analog Devices AD6900 (LeMans) Baseband processor used in the Inmarsat IsatPhone Pro satellite terminal. Techniques for code execution in both the CPU and the DSP are documented, the main result being the instrumentation of functions inside the blackfin DSP with the objective of control, monitoring and emission of Layer-1 GEO-Mobile Radio [9] interface packets.

Contents

Figure 1: IsatPhone Pro

1 Introduction

Modern wireless handset radio systems are composed of a general-purpose CPU handling the user interface, a small analogue front-end and a dedicated CPU or DSP called Baseband processor, that handles the signal processing in effect making it a software defined radio [6].

Previous work exists about wireless Baseband security [1] and the same ideas/concepts can be used against baseband as used on satellite phones. Modifications to the firmware are usually done to remove SIM restrictions (unlocking) of their wireless handsets because SIM management is implemented inside the baseband processor. This is often done as a local security violation of the device using physical access (JTAG or Serial) or vulnerabilities triggered from a previously compromised general-purpose CPU on the system.

Also recent work was published that focuses on the exploitability of security flaws inside the baseband of several popular chipsets using security bugs in the low-level protocol implementation like GSM [2].

Finally, a recent article about static analysis of satellite baseband firmware is available [3] that focus on the cipher used in several satellite terminals, including the GMR-2-P protocol used in the IsatPhone Pro.

A way to execute customized code was found in the CME (User-interface)

general-purpose CPU. This allowed in turn to execute code in the DSP processor, and to instrument low-level communication routines. In all cases, the code was written and run from RAM - in no circumstances we modified the device's firmware or hardware, and hence, all modifications are transient and will disappear once the device is powered off.

It is important to stress that all techniques presented in this article require physical access to the device, thus security implications are low for the user, but could be high for the complete system, that may not be designed to allow a customized device to interact with the satellite network.

2 Firmware extraction

IsatPhone Pro firmware images provided by Inmarsat are available to download here [7]. The downloaded file with .fpk extension is a proprietary file format. Upon execution of the upgrade tool, the path to the .fpk file must be provided. At that moment, the upgrade utility splits the file content into the several inner files and temporarily saves then into the current Operative System temp directory until the upgrade process is finished. At this moment the files can be copied to a working directory as any other regular file and then close the upgrade utility without continuing with the upgrade process.

It contains three internal files named File1.bin, File2.bin and File3.bin. The first file contains the main Operative System on the ARM CPU and can be loaded into IDA Pro for further analysis. The second file contains the Blackfin DSP Firmware that can also be analysed with IDA and the third file is used in a Device Firmware Upgrade (DFU) protocol [3] and was not analysed.

3 General purpose CPU (Application) processor code execution

No code-signing structure was detected in the firmware itself, so code execution using a custom firmware update may be trivial. For malicious purposes code patching often uses this alternative as persistence is desirable. However a better alternative for analysis and development is to patch code in-memory, as non-persistent modifications are faster and safer for the device. An important characteristic of both techniques is that they require physical access,

so the security risk exposed by both techniques is low. Attack surface of this device is limited as no complex software like browsers are exposed to malicious interaction.

3.1 Architecture

The IsatPhone Pro satellite baseband chipset is actually an adapted GSM chipset from Analog Devices, the LeMans AD6900 [5] platform consisting on a 260 MHZ ARM 926EJ-S CPU and a 260 MHZ Blackfin DSP (ADSP-BF532 compatible). The block diagram showing the chipset architecture can be seen in figure 2.
The embedded OS used on the ARM CPU is a derivative of AMX 4-Thumb Kernel from Kadak Inc. [10] called *AOS*. It manages the user interface and applications. In the Blackfin DSP the software uses a custom framework from Analog Devices specially designed for the AD6900, called VDK (Visual DSP++ Kernel)[11]. It is referred in the debug strings as *WIOS*.

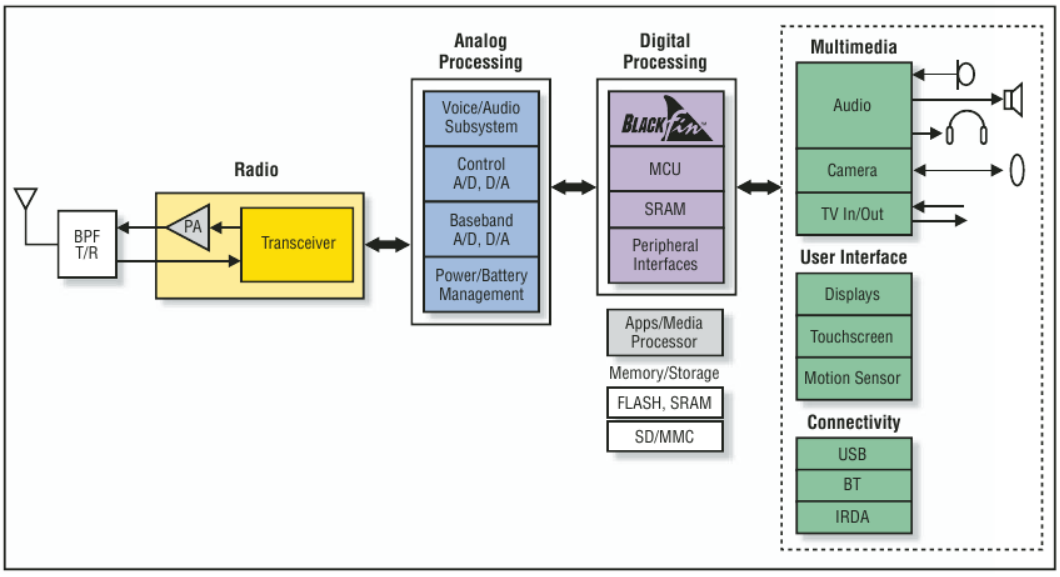

Figure 2: AD6900 chipset architecture

The memory map is detailed in figure 3. Interestingly, both the CPU and DSP share the data/address bus, as a result firmware and memory are accessible from either CPU. The only protection is a read-only mapping

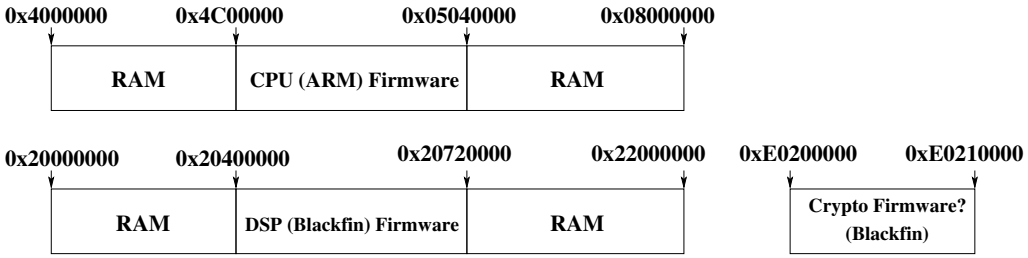

Figure 3: Memory map

of the DSP firmware (memory range 0x20400000-0x20720000) that can be easily bypassed. It is important to note that the curent firmware version at the moment of performing this research for the IsatPhone Pro was 4.0.0.

3.2 Main CPU control

The first and most difficult step is to gain code execution in any CPU. As both are connected, once one of them is under control, the other should be easily controlled. After plugging the micro-USB connector to the PC, the phone presents itself as a USB serial device. When connecting to it with any terminal emulator[1] utility (minicom, screen, etc.) it offers an AT-command interface. This is one of the most commonly exploited attack surfaces on phones:

```
AT
 OK

+SKMSI: 1,2

+SKGPSPOSI: 0

+SKGPSACQ: 0

+CREG: 2
```

[1]Being an emulated serial, bauds settings are not important.

3.2.1 Serial console

Firmware was analysed to discover undocumented commands. Although no proper symbols were found in any firmware file, reverse-engineering was greatly simplified thanks to consistent error-reporting structures present in most functions that printed the name and the source code path of the current function. With this available information, AT command registering functions were trivially found. Apart from standard GSM AT command set, the ISatPhone Pro firmware supports AT "Hooks", that is, additional vendor-specific commands than can be invoked via the AT interface.

Two functions, the first:

```
1 RegisterATCmdHook(char *prefix, void *handler)
```

located at offset 0x04e65768 and the second:

```
1 RegisterPendingATCmdHook(char *prefix, void *handler)
```

located at offset 0x04E0CA30 [2], are used to register customized AT command handlers. Each handler in turn can implement several sub-commands. By looking for all calls to these functions and analysing the parameters used to call these functions (remember one of the parameters is the command string), a complete list of the custom commands can be obtained, for example:

[2] All offsets in this documents are absolute 32-bit CPU address space offsets

RegisterPendingATCmdHook()	RegisterATCmdHook()
@XSH=	@ECHO
@COPS	+CFUN=
@CCLK	+CMSS
@CBC	+CMGS
@CSQ	+COPS=2
@RDIAL	+CPIN=
+CIMI	@sleep
	@wakeup
	@dtr
	@CLOCK
	@pending
	@PMGVER
	@LANG
	...

Several standard GSM AT commands are implemented in this way, for example "+CIMI" : "Request international mobile subscriber identity". However all commands starting with "@" are non-standard.

3.2.2 Xshell

An interesting command to further analyse is the "@XSH=" command. As the name implies, there is an internal shell-like process running concurrently with the AT command processor that can be accessed using the $AT@XSH =$ "XXX" syntax.

Issuing the following command:

```
at@xsh="help"

'help', 'xlog', 'mpoint', 'PmVpm', 'AT', 'XGS', 'FillDB', 'tapi',
'SMS_CNF', 'SMS_QUOTA', 'profiles', 'settings', 'ppm', 'XTime',
'regprint', 'regset', 'xsys', 'xfs_pwd', 'xfs_cd', 'xfs_dir',
'xfs_mkdir', 'xfs_del', 'xfs_info', 'xfs_format', 'xfs_get',
'xfs_fcs', 'xfs_put', 'xfs_getinfo', 'xfs_memput', 'xfs_memget',
'xfs_file', 'xfs_drv', 'xfs_copy', 'xfs_move', 'xfs_attr', 'xosev',
'echo', 'sendback'
OK
```

We can observe that many commands available in the xshell.

Also help can be invoked with an individual command as a parameter to show further information:

```
at@xsh="help xlog"

xlog [p[olicy] | f[iltering] | o[verflow] | ts[tamp] | tr[igger]}]
     [-list |<param>]

OK
```

We can see several commands tasked with the internal configuration database (firmware contains an embedded SQLite 3 DB), internal file system commands ("xfs_*" commands). Also interesting is the "xsys" command that shows the modules or processes running currently in the system:

```
at@xsh="xsys list"

Modules listed with xsys:
[0x14],  (XSH_MODULE)        [0x109], (DATACHANNELS_MODULE)
[0x108], (LOG_MODULE)        [0x05],  (LOG_MODULE)
[0x87],  (Audio_Manager)     [0x67],  (MMF_MODULE)
[0x11a], (BootScreen)        [0x0e],  (MIB_MODULE)
[0x63],  (XTIME_MODULE)      [0x75],  (XMC_MODULE)
[0x81],  (XFS_MODULE)        [0x0a],  (REGISTRY_MODULE)
[0x139], (FDL_MODULE)        [0x101], (XSYSUTILS_MODULE)
[0x117], (USB_MODULE)        [0x53],  (XDB_MODULE)
[0x138], (NVM_MODULE)        [0x09],  (PMGSYNC_MODULE)
[0x64],  (SCHD_MODULE)       [0x11f], (XGS_MODULE)
[0x121], (CAR_MODULE)        [0x4a],  (MAPI_MODULE)
[0x57],  (TAPI_MODULE)       [0x5a],  (MAPI_SMS_MODULE)
[0x58],  (TAPI_MODULE)       [0x4c],  (CNT_MODULE)
[0x61],  (WCLK_MODULE)       [0x83],  (XPROF Module)
[0x12b], (SETTINGS_MODULE)         [0x65],  (ACLK_MODULE)
[0x66],  (CAL_MODULE)            [0x79],  (MIME_MODULE)
[0x5f],  (SYNCML_DS_MODULE)          [0x7c],  (CallHist_MODULE)
[0x113], (CBS_MODULE)
OK
```

3.2.3 Hidden commands

Analysis indicated that the shell supported additional commands that were hidden from the "help" command. The next step is to discover all the commands that can be issued. This is done by first locating the function that register commands into this shell. The function in charge of command registration is located at offset 0x4E99090 and it follows this C prototype:

```
xshell_register_command_(int p_elements_list_,
                         void *pfnCmdHandler,
                            char *pszCommand,
                           char *pszHelpText)
```

The "help" command showed 32 different commands. However we found 80 references to the xshell_register_command() function, meaning that many additional shell commands exists but are hidden from "help", i.e. the registration of the command "reldate":

```
.text:04EFF2EE MOVS  R4, R0
.text:04EFF2F0 ADR   R2, aGetReleaseDate ; "Get Release
                                         ;  Date"
.text:04EFF2F2 ADR   R1, aReldate ; "reldate"
.text:04EFF2F4 LDR   R0, =0x214CE770
.text:04EFF2F6 ADDS  R0, #0x20 ; iUnkAddr
.text:04EFF2F8 LDR   R3, =(cmd_reldate+1) ; pszHelpText
.text:04EFF2FA BL    xshell_register_command_
```

After finding this command, we can execute it from the console in this way:

```
at@xsh="reldate"

Release Date: Mar 21 2011, Extn :

OK
```

3.2.4 Code execution

Hidden commands include "GPS", "sqlshell" and "poke". This last command, "poke", allows for an easy way to modify arbitrary memory locations:

```
at@xsh="help poke"

poke <8|16|32> <hex address> <hex val>

OK
```

Using this command it is possible to modify any position of RAM memory. However, while the ARM OS lacks any form of inter-process memory protection, it has the ability to map areas of code as execute-only. This protection limits the *at@xsh =" poke"* command to modify only non-protected RAM area in the device. Nevertheless arbitrary code execution in the main ARM CPU is possible by injecting native code into an unused memory position and then modifying the execution flow of the CPU into the injected code. This is done usually by modifying a C function pointer and forcing a call to it or modifying a C-stack return address, both structures stored generally in RAM.

A good candidate to hook and modify the execution flow are the AT command pointers. The *xshell*() registration function must store command information into RAM, as it's executed in run-time. As command registration is always done in the same order, offsets and content of data structures stored in RAM are deterministic and do not change between reboots of the device. Analysis of the xshell_register_command() function revealed that the "xshell" commands are stored in a simple double-linked list in RAM, defined as:

```
typedef struct {
    s_command *next;
    s_command *prev;
    char commandstr[16];
    char *helpstr;
    void *(*commandfunc)(int *);
} s_command;
```

For example, the "echo" command structure is always located at offset 0x2148BB28 in RAM, and the commandfunc() pointer is at offset 0x2148BB44. If we execute the next commands:

```
at@xsh="poke 32 0x2148BB44 0x41414141"

writing 41414141 at address 2148bb44


OK
at@xsh="echo"
*phone crashes*
```

The IsatPhone Pro terminal main CPU program counter will jump to address 0x41414141 and reboot as no memory is mapped at that area. We can then use the same "poke" command to write ARM code into free memory and redirect any command to this position, in this way executing arbitrary native code.

The problem of finding a suitable memory position that is not used by the phone Operative System is solved using another useful hidden command, "xmm", whose syntax follows:

```
xmm [conf | list_pools |info [<pool_num>|all] | owners [info|reset]
    | malloc <size> | jrnl [start [<name>] | stop]]
```

The "xmm" command allow access to the memory manager of the Operative System, we can use the "malloc" subcommand to allocate a memory block of arbitrary size (up to the memory size limit of the device) that will not be disturbed by the operative system and can be used to store any custom code or data. In figure 4 steps required for ARM code execution are detailed.

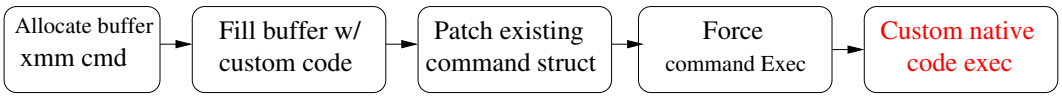

Figure 4: ARM code execution steps

4 DSP (Baseband) processor code execution

Native code-execution in the main CPU of the chipset can be used to insert malicious code and steal data. As this requires physical access, the security impact of this attack is low. However, low-level GMR-2 protocol parsing and modulation/demodulation is not performed in this CPU, but in the Blackfin DSP that shares the memory address space with the main ARM CPU.

A trivial way to execute code would be to use the "poke" command to directly modify the Blackfin firmware located at address 0x20400000 (see fig 3), however the ARM CPU protects this memory area, setting it read-only. Any write attempt in the DSP firmware memory range will cause an exception and reboot the phone.

4.1 Memory protection bypass

As the memory protection is implemented using the standard ARM V5 MMU, we can disable it by simply executing custom code in the ARM processor. Examples of code to disable the MMU are provided by ARM Inc. [8]. The code sequence to modify DSP firmware is:

```
1     MRC p15, 0, r1, c1, c0, 0     ; read CP15 Register 1
2     BIC r1, r1, #0x1
3     MCR p15, 0, r1, c1, c0, 0     ; disabled
4     NOP ;Fetch flat
5     NOP ;Fetch flat
6     NOP ;Fetch flat
7     NOP ;Fetch flat
8
9     ; Modify DSP firmware here
10    ; I.E. write to 0x20400000-0x20700000
11    ;
12
13    MRC p15, 0, r1, c1, c0, 0     ; read CP15 Register 1
14    ORR r1, r1, #0x1
15    MCR p15, 0, r1, c1, c0, 0     ; enable MMUs
16    NOP ;Fetch translated
17    NOP ;Fetch translated
18    NOP ;Fetch translated
19    NOP ;Fetch translated
```

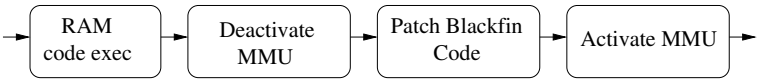

Figure 5: Blackfin code execution steps

This modification (illustrated in figure 5) is performed in-memory and is not persistent between reboots. It allows complete access to all DSP firmware where functions to monitoring and injection of the GMR-2 protocols were implemented.

4.2 Generic function hooking

In sections 3.2.4 and 4 we presented a method to insert and execute code into AT commands. However a more generic hooking technique is necessary to modify behaviour of all functions. The method (implemented in utility "isat_hook_bf_call.py") is to redirect any subroutine call inside the target function into the hook code by overwriting only the original CALL instruction. As figure 6 shows, the hook code saves all registers, executes, restore registers and executes the original call before return (or simply jump to the original function). This allows to cleanly hook any function in the firmware. This method can be used for ARM and Blackfin, however in this research only Blackfin code was hooked in this way.

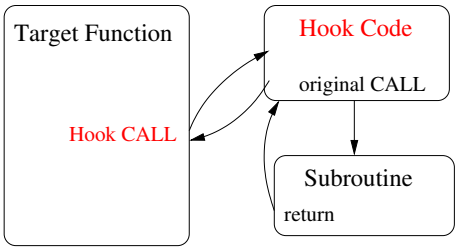

Figure 6: Hooking existing code

5 GMR-2 monitoring

GMR-2 is a complex family of specifications, most of them are freely available as the ETSI TS 101-377 standard [3]. It specifies the first three OSI layers of radio interface. It is composed of 7 series of documents, each containing several articles. In this paper we will focus in the 05-series, the Radio interface physical layer specifications. The ETSI TS 101 377-5-3 document details the format of all the low-level channels. According to Annex A of document, there are 28 different control channel types, and 8 traffic channel types[4]. Each channel type has its own rate, packet size and error-correction (ECC) algorithms[4]. The communication stack follows a standard architecture of an inner block-code (Reed-Solomon or fire-code), outer Convolutional code (punctured or unpunctured, decoded using the Viterbi algorithm) and a variable interleaver (see Fig. 7 for an example). Additionally, Walsh-Hadamard codes are used when the channels are combined using a CDMA algorithm, however most channels share the medium as TDMA. Block sizes can be as small as 28 bits and up to 240 bits. Encoded block sizes are much larger due to parity added. Our first objective is to dump raw packets from the downstream channels.

5.1 Dispatcher

There is a function shared by most channel implementations as we can see in figure 8 that acts like a dispatcher. We call this function DecodePacket(), located at offset 0x206b8130 [5]. This function receives the ECC-encoded data packet as an argument and decodes it using different error correction algorithms according to the channel type. This is the last transformation that the data suffers in the communication stack before data is available as plain text.

This particular function was hooked to dump any received packet into a memory-resident circular buffer. As the packet buffer is an argument of the dispatcher function, the pointer is always available as a DSP register or also can be read directly from the stack. As Blackfin firmware cannot access the serial console directly, the current way to dump big amounts of data is to

[3] specification for the cipher are not public but were reverse-engineered in http://gmr.crypto.rub.de

[4] Not all channel types are implemented in the current version of GMR-2

[5] This is Blackfin firmware

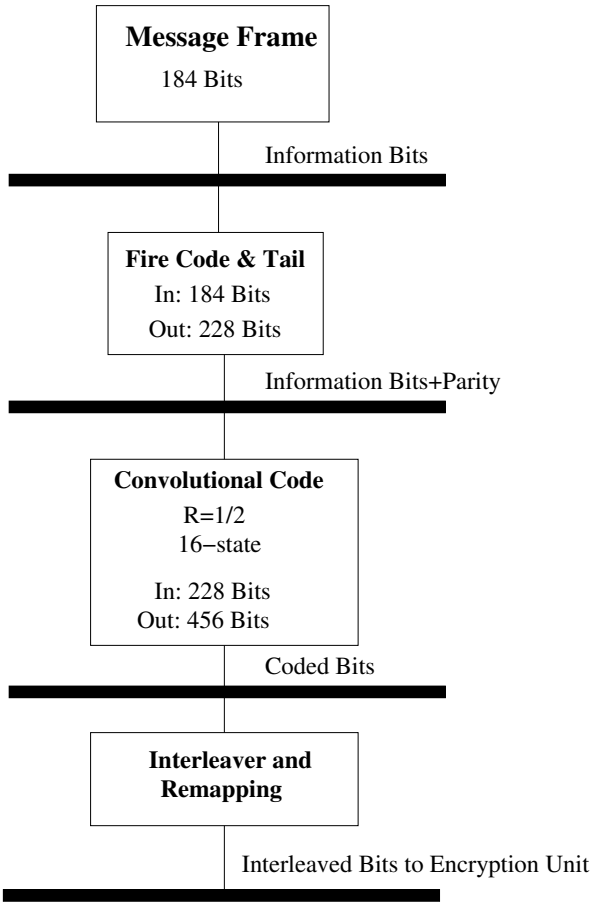

Figure 7: Satellite Broadcast Control Channel (S-BCCH), Paging Channel (S-PCH) and Access Grant Channel S-AGCH communication stack

copy it to an unused memory buffer shared with the ARM CPU and then dump it from an ARM routine, that does have access to the serial console.

Once the packet start accumulating in the circular buffer, it is dumped to the serial console using custom code in the ARM side in the form of an AT command. Then a python script in the host computer periodically executes this custom AT command that polls the circular packet buffer, and dumps any new packet into a TUN/TAP device to simulate a network device that contains all satellite transmitted packets.

This network device uses the GSMTAP protocol so it can be easily read

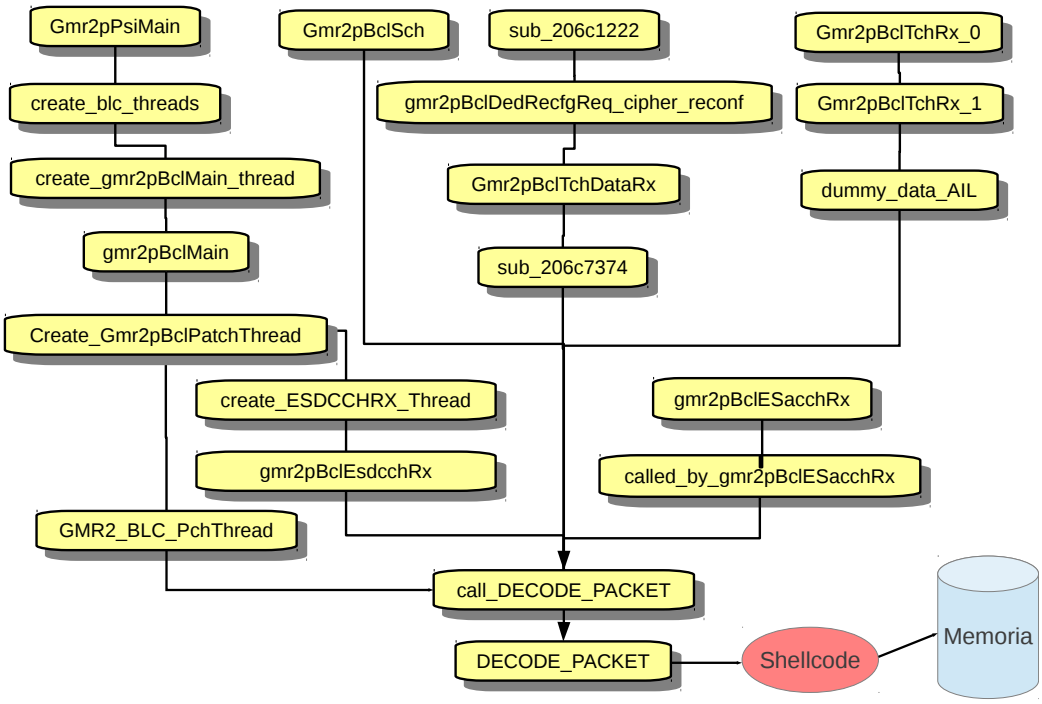

Figure 8: Decode Packet function dispatcher

by popular sniffers such as Wireshark [6] In this way, one can use available packet monitoring software to analyse all incoming packets, as Fig. 9 shows.

A similar technique can be used to hook the function EncodeECC() at 0x206b817A, that is in charge of encoding data on all outgoing channels with an error correction algorithm that depends on the channel type. By copying outgoing packets to the same circular buffer, we can monitor both outgoing/incoming packets of most or all channels in the terminal.

Before GMR2 parsing is implemented in protocol analysers like wireshark, a preliminary solution was a custom packet decoder implemented in python that assemble messages from different channels in real-time as they are received from the downlink. In Fig. 11 decoding of a balance enquiry response USSD message can be seen.

[6]Wireshark v17 has GSMTAP included but incomplete support for GMR-1, and no support for GMR-2, however the protocols share many structures and definitions

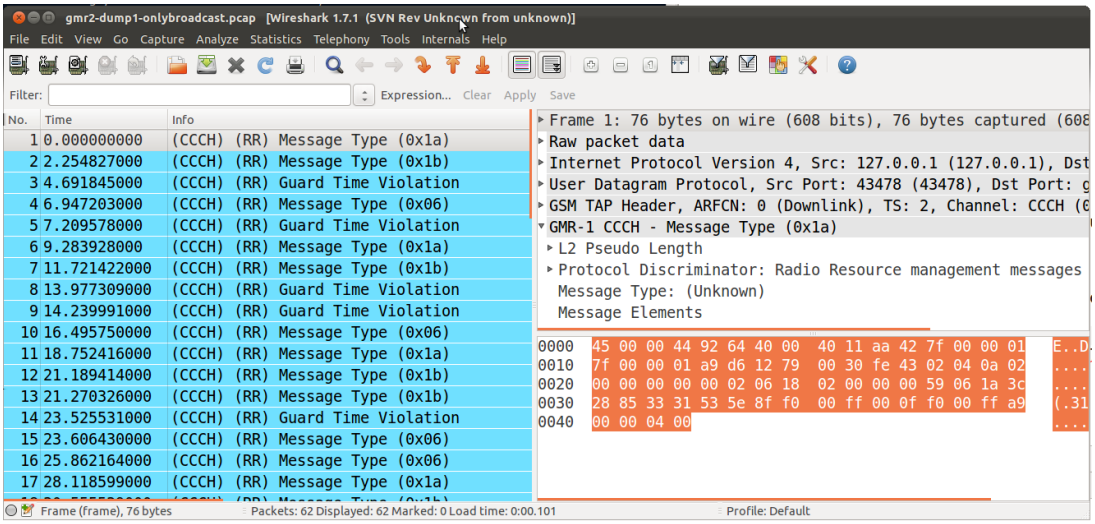

Figure 9: Capturing GMR2p broadcast packets with wireshark

6 GMR-2 injection

An interesting possibility of the instrumentation of the EncodeECC() function is the ability to modify original data just before the first data transformation, the ECC algorithm.

Being EncodeECC() a low-level function, all of the packet is available, including headers and meta-data. All this data can be modified but we focus on modification of harmless data like the body of a SMS message. Modification of headers or meta-data while possible, incurs on a risk of the satellite provider as it is data it probably do not expect inside the satellite network (see 7). Nevertheless injection of arbitrary packets before ECC and encryption is possible, a diagram of the technique used can be seen in figure 10.

6.1 SMS Transmission

There are several services in this terminal that uses raw-text message packets: SMS, Email and USSD messages[7]. Most of the message is encrypted however we are able to insert modifications before encryption. Messages are separated

[7]Like pre-paid credit management

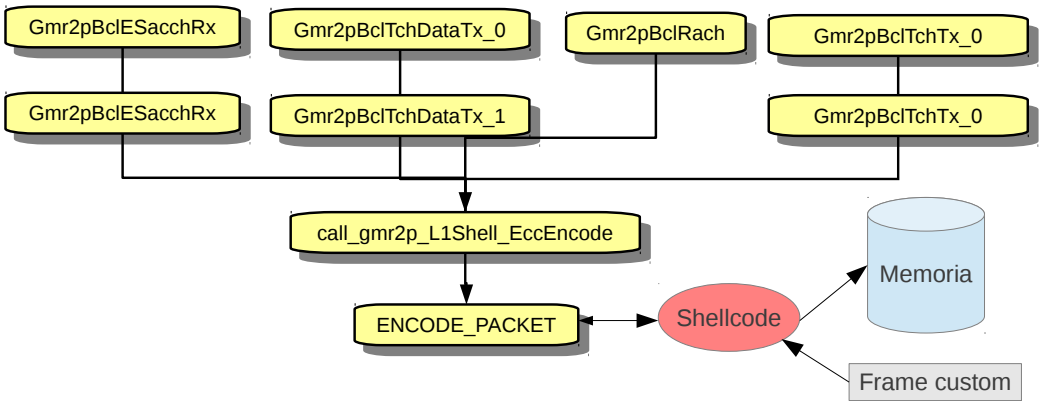

Figure 10: Injecting a custom frame by hooking EncodeECC()

into several 24-byte packets, with a small 4 bytes header (Packet type and offset) and the rest is the message segment encoded as 7-bit data.

To inject any data into this message the first step is to encode it in 7-bit characters, and then insert the encoded bits into the proper output buffer by hooking the EncodeECC() function and inspecting, for example, all channels of type 0x3 (SMS). Figure 12 shows a successfully SMS message being intercepted and modified by our custom code hooking the EncodeECC() function inside the Blackfin baseband.

7 Theoretical attacks

As any complex protocol, simple attacks can be performed on the remote-end protocol or data structure parsers. For example, this is an original MES-to-Satellite hexadecimal bytes of the frame encoding the SMS message *"Hola desde la Eko"* with 7-bits per character. The bytes marked in blue are the modification of the message, changing the original message to "**PWND** desde la eko" (Note that we do not need to calculate any additional checksum after the modification):

0D 02 53 F8 00 00 FF 14 D0 AB 93 08 22 97 E7 E4

However we can see that the length of the message (20 characters in this case) is also encoded at the beginning of the message, here marked in red:

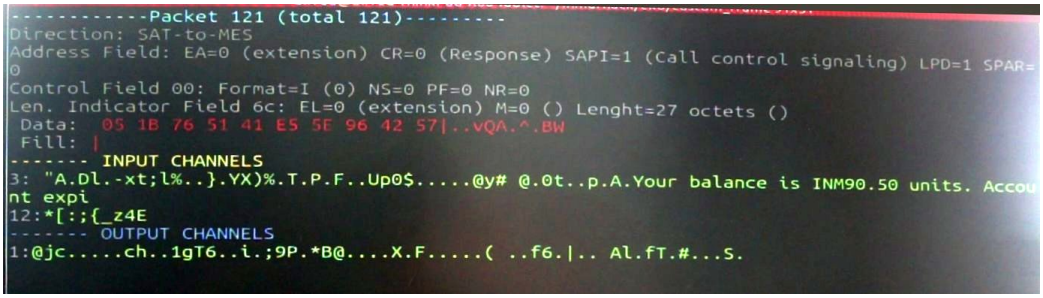

Figure 11: Python GMR-2 packet decoding of a balance enquiry USSD resp. message

0D 02 63 F8 00 00 FF 14 D0 AB 93 08 22 97 E7 E4

By changing only this byte, a malformed length message can be sent into the satellite network. Buffer overflows are commonly triggered by parsers incorrectly accepting invalid length messages, and this modification and other similar in the headers of the GMR frames could theoretically trigger bugs in the remote protocol parsers.

Another theoretical attack could be causing a resource exhaustion by requesting physicals channels and never releasing them. However this kind of bugs are less common and easily detected.

Also attacks can be done by forging the GPS position transmitted to the satellite network. Until recently, due to patent suits different areas of the world had different service pricing, for example pre-pay service was not available inside USA. By simulating a GPS position just outside continental USA it could be possible to use pre-pay service inside the country [8].

Finally, by having direct access to monitoring and modification of the encryption routines, theoretical attacks like [3] now are practical.

8 Closing remarks

In this article we described a method for easy code injection in a popular satellite phone platform. Low-risk of hardware destruction and fast in-memory firmware modification was achieved. While the security implications

[8] At the publication of this article the Inmarsat pre-pay service is available inside USA

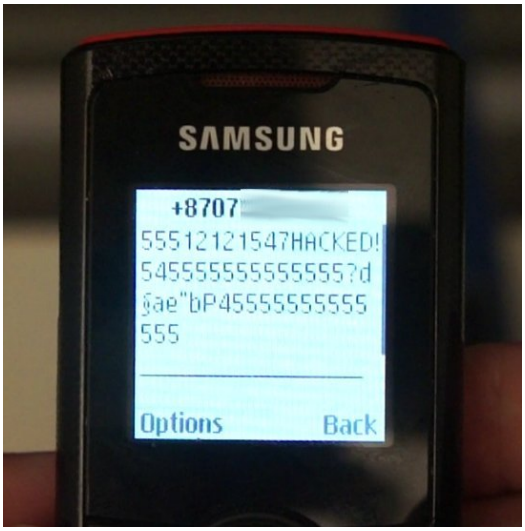

Figure 12: "HACKED" string inserted in SMS sent by Inmarsat IsatPhone Pro

for the end-user are low as physical access is still required, there is a risk for the satellite network as it may not be designed for malformed and/or malicious low-level signal transmission by end-users. Compared to an SDR system reimplementation, advantages include completeness of protocol, TX/RX capacity and a very good amplification/antenna system, components often missing or badly implemented in Software Radios. More work is needed to turn the AD6900 chipset in a proper SDR and even if it's possible it will posses a very limited bandwidth. However for GMR/GMR-2 study and a easy and low-cost platform for RX/TX GMR-2 baseband experimentation, the platform is ideal.

References

[1] Luis Miras, *Baseband playground*. Ekoparty, 7th Edition, 2011.

[2] R.-P. Weinmann, *The Baseband Apocalypse*. Chaos Communication Congress (27C3), 2011.

[3] Driessen - Hund - Willems - Paar - Holz, *Don't Trust Satellite Phones: A Security Analysis of Two Satphone Standards*, IEEE Symposium on Security and Privacy (128-142), 2012.

[4] T. K. Moon., *Error Correction Coding: Mathematical Methods and Algorithms.*, John Wiley & Sons, New York, USA, 2005.

[5] LeMans AD6900, *http://www.mediatek.com/en/Products/product_content.php?sn=12*. Mediatek

[6] Software Defined Radio, *http://en.wikipedia.org/wiki/Software_defined_radio*, Wikipedia, retrieved september 2012.

[7] ISatPhone Pro Support, *http://www.isatphonelive.com/support*. Inmarsat

[8] ARM MMU Disable, *http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344f/Cihfgdif.htm* ARM Inc.

[9] GEO Mobile Radio Interface, *http://en.wikipedia.org/wiki/GEO-Mobile_Radio_Interface*. Wikipedia, retrieved september 2012.

[10] Kadak AMX rtos, *http://www.kadak.com/rtos/rtos.htm*, Kadak Inc.

[11] Visual DSP++, *http://www.analog.com/en/processors-dsp/blackfin/vdsp-bf-sh-ts/products/product.html*, Analog Devices.