How to design a baseband debugger

David Berard and Vincent Fargues
david.berard@synacktiv.com
vincent.fargues@synacktiv.com

Synacktiv

Abstract. Modern basebands are an interesting topic for reverse engineers. However, the lack of debugger for these components makes this work harder.

This article presents how a 1-day vulnerability in Samsung Trustzone can be used to rewrite the Shannon baseband memory and install a debugger on a Galaxy S7 phone. The details of the debugger development are explained and a demonstration will be done by using specific breakpoints to extract interesting informations.

1 Introduction

In 2020, smartphones are used by everyone and have become ones of the most targeted devices. However, phone manufacturers put a lot of effort into securing them by hardening kernel, browsers, and every binary running on the application processor.

As a consequence, researchers began looking for vulnerabilities in other components such as Wi-Fi firmware, Bluetooth firmware, or basebands [2, 4, 5] which became easier targets.

This presentation focuses on the baseband component which is used by a modern phone to connect to cellular networks (2G, 3G, 4G and even 5G).

Another motivation for a researcher may have been the fact that several smartphone's basebands are a target for the mobile *pwn2own* [1] competition. During the last 3 years, the team *Fluoroacetate* has been able to exploit a vulnerability in the Samsung baseband called Shannon.

The Shannon's real-time operating system is relatively big, and many tasks are involved to provide a connection to the cellular network. The functioning is quite complex to understand, and no debugger is available to do dynamic analysis of the OS/Tasks.

This presentation covers the Shannon's architecture, and how a debugger can be implemented on top of that.

1. https://www.zerodayinitiative.com/Pwn2OwnTokyo2019Rules.html

2 Related work

In 2012, Guillaume Delugre has presented a debugger of Qualcomm's baseband running on a 3G USB stick [3]. The debugger code is available online [2] and has been used as inspiration for doing our research.

Comsecuris published some script to perform static analysis on Shannon Firmware image [3] [4].

3 Shannon architecture

The component studied in this paper is the Communication Processor (CP) developed by Samsung and known as Shannon. The firmware provided by Samsung runs on a dedicated processor which is based on ARM Cortex-R7. It is used on all non-US Samsung phones (US-Phones use Qualcomm chips).

The code running on the baseband processor is responsible for handling the full mobile phone stack: 2G-5G, communication with the Application Processor (AP), communication with SIM cards.

The file *modem.bin* provided in Samsung's firmwares is the code that runs on the baseband. It can be easily loaded in IDA in order to start reverse engineering. Previously, this file was stored encrypted and decrypted at runtime but this is not the case anymore, recent firmwares being in cleartext.

3.1 Shannon operating system

The baseband runs a Real Time OS that switches between tasks. Each task is dedicated to a particular function such as communicating with the application processor, handling a radio layer, etc.

Each task has the same structure. Every necessary component is initialized first and then runs a message loop waiting for events from other tasks. The communication between tasks is done using a mailbox system which allows reading or writing based on a mailbox ID. The figure 1 shows a simplified example of communication between tasks.

Tasks related to radio messages are interesting when looking for vulnerabilities. Indeed; they can easily be found in the firmware by following the strings *<RADIO MSG>*.

2. `https://code.google.com/archive/p/qcombbdbg/`
3. `https://github.com/Comsecuris/shannonRE`

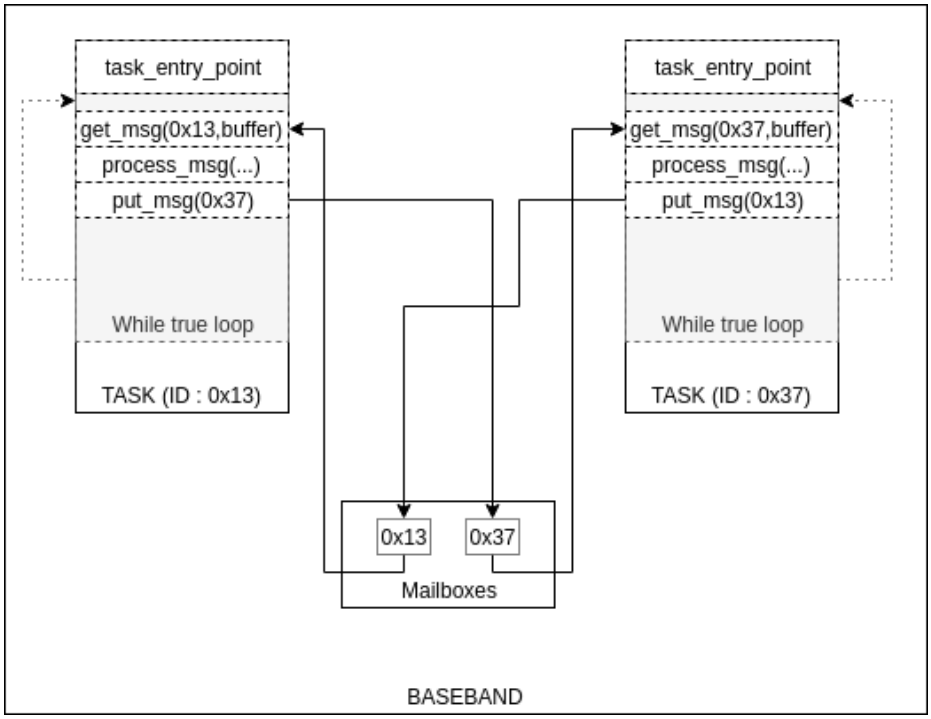

Fig. 1. Mailboxes used for inter tasks communications

3.2 Tasks scheduling

In order to debug a stand alone task running in the RTOS, the mechanism responsible for scheduling tasks as been reversed engineered.

The task structure has been studied and the following offsets of interest are used by the developer.

```
00000000 next_task        DCD ?
00000004 prev_task        DCD ?
00000008 task_magic       DCB 4 dup(?)
0000000C id_plus_1        DCW ?
0000000E id               DCW ?
00000010 field_10         DCW ?
[...]
00000018 sched_grp        DCD ?
0000001C sched_id         DCD ?
00000020 sched_grp_ptr    DCD ?
00000024 task_start_fn_ptr DCD ?
00000028 stack_top        DCD ?
0000002C stack_base       DCD ?
00000030 stack_ptr        DCD ?
00000034 running_or_not   DCD ?
[...]
```

```
0000004C saved_stack_ptr DCD ?
[...]
0000005C task_name       DCB 8 dup(?)
[...]
```

Listing 1. Example of logs

A list of tasks is saved by the RTOS at the address *0x04802654* and the id of the current task is saved at the address *0x04800EE8*.

All these informations are used to pass on tasks related information to the gdbserver stub and to enable thread debugging.

3.3 AP-CP Communications

Communications between the application processor and the baseband processor are done using shared memories and mailboxes. Mailboxes are used for one-way communications, some of them are used for CP to AP communications while some others are used for AP to CP communications.

Mailboxes notify (using an interrupt) the other processor and send a 32-bit value. Sixty-four mailboxes are available, but only twenty are used in the Galaxy S7.

The baseband and the Linux kernel use a protocol called *SIPC5* to[2] communicate. The protocol has multiple channels; the Linux driver acts as a demultiplexer to dispatch these channels to user-space programs. Samsung publishes the kernel source code, it is therefore simple to study.

Most communications are handled by 2 processes on the AP:

— *cbd*: this process is responsible for the boot and initialization of the baseband firmware.

— *rild*: this process handles baseband communications after the baseband starts.

3.4 Boot

The baseband boot is driven by the *cbd* process and operates as follows (simplified):

— The *MODEM* firmware image is read from the internal flash memory. This image is parsed to get two major parts:*BOOTLOADER* and *MAIN*.

— The *BOOTLOADER* part is sent to the kernel with the *IOCTL_MODEM_XMIT_BOOT* ioctl. This part is copied in the future baseband bootloader physical memory address (*0xF0000000* [4] on the GS7).

4. All physical addresses can be found in the Linux Device-Tree.

— The *MAIN* part is sent to the kernel. This part is copied in the future baseband physical memory address (*0xF0000000 + 0x10000* on the GS7).
— *cbd* emits the *IOCTL_SECURITY_REQ* ioctl. This ioctl is used to emit an *SMC* call to the Secure Monitor. The Secure Monitor is the most privilegied code running in the AP.
— The Secure Monitor marks the baseband memory zone (*BOOTLOADER* and *MAIN*) as secure memory in order to prevent modification from the non-secure world (i.e Linux Kernel).
— The Secure Monitor verifies the signature of the *BOOTLOADER* and *MAIN* parts of the baseband firmware.
— The Secure Monitor configures the baseband processor. As part of this configuration, the physical memory reserved for the baseband (*0xF0000000* on GS7) is set to be the baseband main memory.
— If all the previous steps succeeded the *cbd* emits the *IOCTL_MODEM_ON* ioctl to start the baseband processor.

During its initialization, the baseband firmware configures the Cortex-R7 Memory Protection Unit (MPU) to restrict access to memory zones.

The Cortex-R7 core does not provide advanced memory management features like address translation.

4 Debugger injection

All the injection steps are performed by exploiting a vulnerability in the application processor which allows to write the memory shared between AP and CP. This memory is used by the baseband to store the *MAIN* part of its firmware.

4.1 Exploit a 1-day vulnerability

As seen in the baseband boot section, the bootloader memory is signed by Samsung, and the memory is marked secure before checking the signature.

To be able to modify the baseband code, two methods can be explored:
— Find a vulnerability in the baseband itself and try to bypass the MPU from here in order to patch the code. This method is baseband's firmware dependant.
— Find a vulnerability in the application processor software to gain the ability to modify the secure memory. This method depends on the AP software version, but not on the baseband's firmware version.

The second method was chosen as a 1-day vulnerability can be used and the debugger will not depend on the baseband version.

Quarkslab recently disclosed a chain of vulnerabilities in the Trusted Execution Environment which allows gaining code execution in a secure driver on the Samsung Galaxy S7 Exynos [6]. Samsung did not implement correctly the anti-rollback mechanism, thus the exploit chain is still applicable to up-to-date phone by loading the old driver and trusted applications.

Trusted Execution Environment The Galaxy S7 (G930F) uses Kinibi as Trusted Execution Environment (TEE). Many good descriptions of its internals can be found online, so only the required knowledge is covered by this document. Readers are strongly encouraged to read the full explanation given by Quarkslab on the TEE internals/exploitation in their presentation.

Kinibi is a small operating system built by Trustonic [5] running in Secure World which provides security functionalities to the Normal World OS and applications. The segmentation between secure and non-secure world relies on ARM TrustZone technology as seen on figure 2.

Kinibi is composed of multiple components:

— The microkernel (MTK), which runs at the S-EL1 exception level;
— The main task (RTM), which runs at the S-EL0 exception level;
— Secure drivers are running at the S-EL0 exception level;
— Trusted Applications are running at the S-EL0 exception level;

The microkernel provides a set of system calls to userland applications (RTM, TAs, drivers). These system calls are filtered depending on which components perform the call.

Trusted Applications can only call a limited part of these syscall: basically no communication with underlying components (no SMC), no direct memory mapping. Only basic syscalls and IPCs to secure drivers are allowed.

TA Trusted Applications can be reached from Android allowed applications through a Linux driver and a set of SMCs forwarded by the Secure Monitor to the TEE.

TAs implement a loop for incoming messages which waits for the notification from Normal World (NW), handles the messages, and notifies the NW that a response is available.

5. `https://www.trustonic.com/`

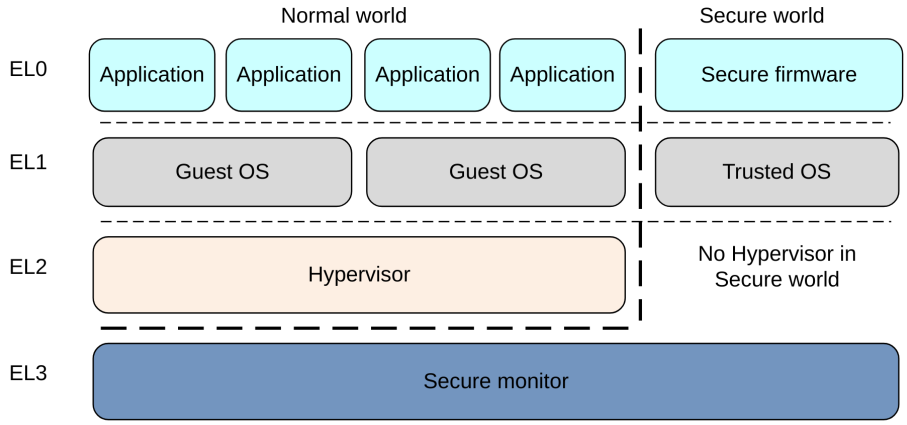

Fig. 2. Aarch64 exception levels

Messages are exchanged through a shared memory.

The SSEM Trusted Application contains a trivial vulnerability inside one of the command handler, a stack-based buffer overflow.

This TA is built without stack cookies and TEE does not provide security mechanisms like ASLR / PIE, therefore the exploitation is quite straightforward.

Since Trusted applications cannot change attributes on memory pages to mark them as executable, the exploitation is done with a ROP chain.

A ROP chain is used to gain the ability to perform arbitrary IPC calls to secure drivers.

Secure Driver Like TA, Secure Drivers implement a loop for incoming IPC messages. Messages are memory pages shared between the TA and the driver. When a message arrives, the driver maps the TA message memory in its own memory virtual space.

Like the SSEM TA, the VALIDATOR driver contains a trivial vulnerability in one of the IPC message handler, a stack-based buffer overflow.

This driver is built without stack cookies, so the exploitation is quite straightforward.

Secure Drivers are way more privileged than Trusted Application; they can for example invoke syscall to:

— Map physical memory (inside a whitelist)
— Call other components by performing Secure Monitor Calls

Secure Monitor The Secure Monitor is the most privileged piece of software in the ARM TrustZone architecture. It runs at the EL3 exception level, and it is responsible for handling Secure Monitor Calls and for the switch from Normal World to Secure World.

From the secure driver, there are lot of ways to gain the ability to patch the Secure Monitor/baseband (mmap of S-EL1 components, SMC calls reserved to secure world, ...). There are no public vulnerability and exploit to achieve this part.

In order to modify the baseband code from the application processor, two Secure Monitor patches are applied:

— Signature check is disabled in order to load a modified baseband bootloader;
— When the baseband is copied, the memory is not marked as secure.

4.2 Baseband code injection

Baseband memory zones The baseband memory layout is composed of:

— The bootloader part is mapped at address *0*;
— The main code is mapped at the address *0x40000000* and is hosted on a physical memory shared with the application processor;
— The cortex-R7 provides a Tightly Coupled Memory (TCM). This memory zone is not shared with the application processor, and is used for low latency and time predictability. The baseband uses this memory, and copies the code from the main part when the baseband is being initialized.

Injection Since the monitor has been patched to remove integrity/authenticity check of the baseband image, a patched baseband image can be loaded into the communication processor.

This image includes the debugger host code and the modified interrupt handlers.

After the baseband starts, all of the required memory patches are then applied from the debugger host code (this allows for example modifying TCM memory which is not available to the application processor).

The baseband debugger code can also be injected after the baseband starts, but since there is no debugger code prior to that point to perform cache eviction operations, this may cause issues with the cortex-R7 caches. Modified interrupts handlers are not taken immediately after the modification.

5 Debugger Development

This section covers the development of the debugger. Two main components are involved: a payload that runs on the baseband itself and a server that runs on the application processor (AP) which provides a gdb-server interface.

The baseband payload is compiled to run in the Cortex-R7 processor (CP). It is responsible for communications with the server, handling all the interrupts and providing useful primitives in order to read and write memory and registers.

5.1 Interrupts Handler

To handle breakpoints, invalid memory accesses, invalid instructions etc., the debugger must be able to handle all the interruptions.

In ARM architecture, there is a vector table at the entry point of the firmware responsible for handling all exceptions. The list of vectors can be seen in figure 3.

Normal Vector offset	High vector address	Exception
0x0	0xFFFF0000	Not used
0x4	0xFFFF0004	UNDEFINED instruction
0x8	0xFFFF0008	Supervisor Call
0xC	0xFFFF000C	Prefetch Abort
0x10	0xFFFF0010	Data Abort
0x14	0xFFFF0014	Not used
0x18	0xFFFF0018	IRQ interrupt
0x1C	0xFFFF001C	FIQ interrupt

Fig. 3. Arm exception vector table

When installing the debugger, the aim is to rewrite all the handlers to redirect the execution flow to the code of the debugger to handle all exceptions.

An example of the use of these handlers is presented in Figure 4 regarding how breakpoints are handled.

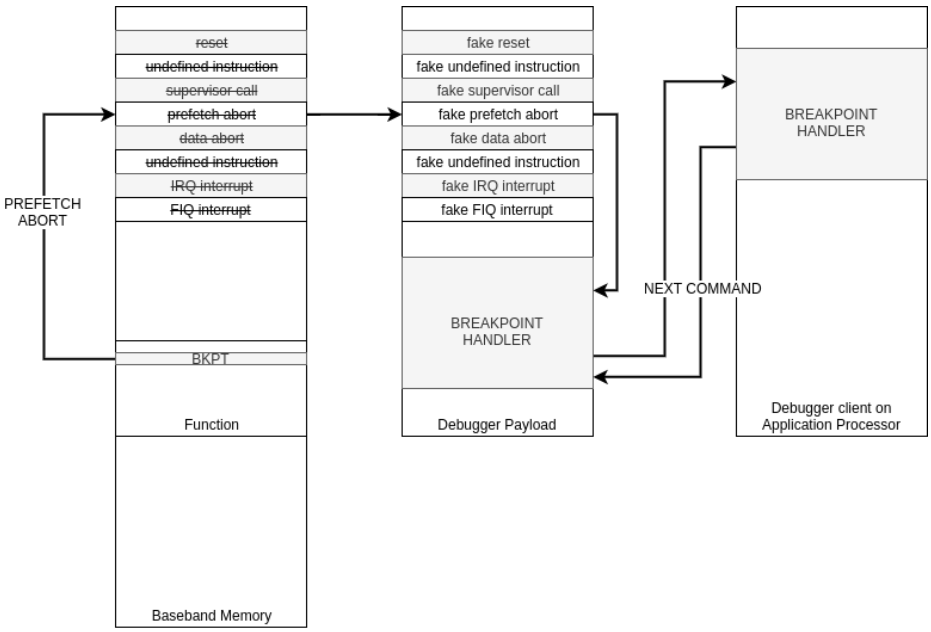

Fig. 4. Breakpoint handling

5.2 Communication

The communication between the injected code in the baseband and the debugger server in Android userland relies on the same mechanisms used by the Linux kernel to communicate: shared memories and mailboxes.

A Linux kernel module allows reading and writing on shared memories from Linux userland. A set of *ioctls* is used to send messages through mailboxes.

Mailboxes are an easy way to trigger an interrupt in the baseband side. The IRQ interruption handler allows jumping on the injected code, and getting commands from the debugger server.

The IRQ generated by the mailbox write is handled by a modified IRQ handler. This handler uses the Generic Interrupt Controller (GIC) to

know which IRQ is active. If the current interruption is the mailbox IRQ (86) the handler uses the *EXYNOS_MCU_IPC_INTMSR1* registers to know which mailbox interruption is active, and jumps in the debugger command handler if the mailbox interruption dedicated to the debugger is active. In other cases, the modified handler jumps in the original IRQ handler.

After handling the interruption, the modified handler acknowledges it to the GIC and calls a function inside the modem firmware to perform the end of exception routine (rescheduling, etc.).

Thanks to this mechanism based on mailbox IRQ, the debugger server running on the AP is able to interrupt the execution of the CP, and send commands to the injected code.

Commands handled here allow the debugger server to read and write the CP memory, resume a task after a breakpoint, and stop and resume the full CP OS.

In the other direction (CP->AP), the same mechanism is used. The Linux driver registers a mailbox IRQ handler for the dedicated mailbox interruption. Mailbox interrupts received by this handler can be read by a userland application through a char device.

5.3 Shared memory synchronization

CP and AP share some memory ranges, but each CPU has its own memory cache system. To be sure that the data is written to the memory before interrupting the other CPU, the cache has to be flushed / synced.

The cache management in the AP is well known. It's a standard ARMv8 cache managed with dedicated instructions. Before generating the mailbox interruption, the cache is flushed.

The CP cache management is less known. Reverse engineering the IPC system in the CP firmware allows to understand the cache mechanisms. The CP uses an external PL310 cache controller [1]. Cache sync and flush requires some I/O mapped registers to be written/read. A cache flush is done before each call to custom interruption handlers, the shared memory range is flushed on the controller and after each call a cache sync operation is requested to the controller.

5.4 GDB Server

The debugger server in userland implements the specification of gdb-server in order to be able to connect a gdb-client. All the required func-

tionalities such as read or write memory are implemented in the baseband payload. The Figure 5 describes the flow.

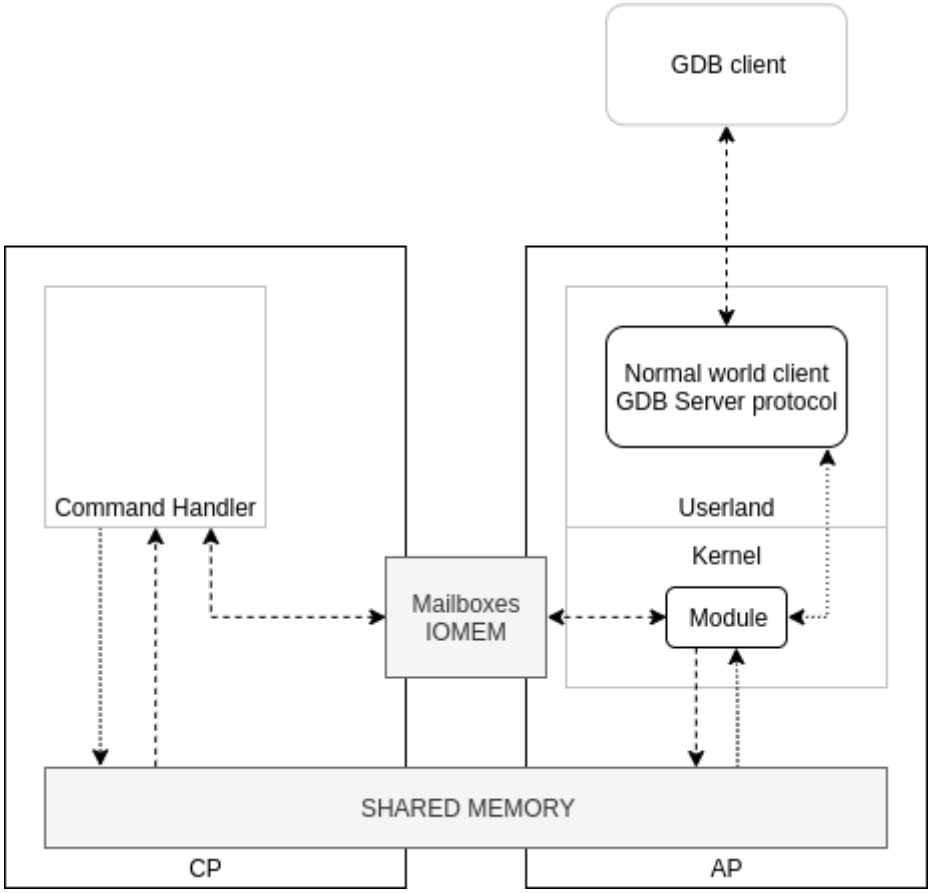

Fig. 5. Communication between different components

6 Examples of use

6.1 Logs enabling

While reversing the baseband's firmware, a function responsible for printing logs has been identified. However, this function is not enabled on production devices.

As a demonstration of the debugger, a **breakpoint** has been set on this function and a command has been written to print the logs when the breakpoint is reached.

```
# ./debug_server
[+] ****************************** GMC RX EVENT (1-24)
    ************************************
[+] GET int@HISR :81
[+] xdma is not running
[+] ShmTIMER KICK @ShmTask
[+] gmc_MsgPreProcessNoWait
[+] ####Processing the incoming message####
[+] GET int@HISR :81
[+] xdma is not running
[+] ShmTIMER KICK @ShmTask
```

Listing 2. Example of logs

6.2 Modification of a NAS packet

A breakpoint can be set on a function responsible for sending NAS[6] packet to modify the content sent to the core network. This can be used to test or fuzz this kind of equipment from a controlled device.

Here the example on 6 demonstrates the injection of the string *SYNA* in the field p-tmsi of a *GMM-Attach-Request* packet.

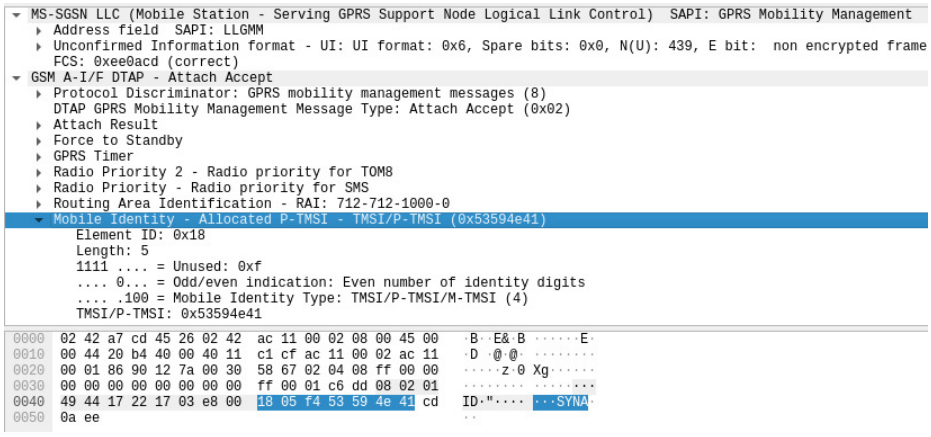

Fig. 6. Injection in a GMM packet

6. https://en.wikipedia.org/wiki/Non-access_stratum

7 Conclusion

Due to to vulnerabilities on the GS7 phone, it is possible to write in secure memory and to inject a custom payload on the Shannon Baseband.

A debugger has been written in order to debug potential crashes, enable hidden functionalities such as logs or inject GPRS traffic.

This tool is provided for the Galaxy S7 but can be adapted to a more recent Samsung Phone if a public vulnerability allows to writing in secure memory.

The source code is available on Github.

References

1. ARM. PL310 cache controller – technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0246a/DDI0246A_l2cc_pl310_r0p0_trm.pdf.
2. Amat Cama. A walk with Shannon. https://downloads.immunityinc.com/infiltrate2018-slidepacks/amat-cama-a-walk-with-shannon/presentation.pdf, Infiltrate 2018.
3. Guillaume Delugre. Rétroconception et débogage d'un baseband Qualcomm. https://www.sstic.org/media/SSTIC2012/SSTIC-actes/rtroconception_et_dbogage_dun_baseband_qualcomm/SSTIC2012-Article-rtroconception_et_dbogage_dun_baseband_qualcomm-delugre_1.pdf, SSTIC 2012.
4. Nico Golde and Daniel Komaromy. Breaking Band – reverse engineering and exploiting the shannon baseband. https://comsecuris.com/slides/recon2016-breaking_band.pdf, Recon 2016.
5. Marco Grassi. Exploitation of a modern smartphone baseband. https://i.blackhat.com/us-18/Thu-August-9/us-18-Grassi-Exploitation-of-a-Modern-Smartphone-Baseband-wp.pdf, Black Hat USA 2018.
6. Maxime Peterlin, Alexandre Adamski, and Joffrey Guilbon. Breaking Samsung's ARM TrustZone. https://i.blackhat.com/USA-19/Thursday/us-19-Peterlin-Breaking-Samsungs-ARM-TrustZone.pdf, Black Hat USA 2019.