How to design a Baseband debugger

SSTIC 2020

June the 3rd
Synacktiv
David Berard, Vincent Fargues

Table of Contents

Table of Contents

Who are we

- David Berard
- Vincent Fargues

- Synacktiv
  - Offensive security company created in 2012
  - Soon 74 ninjas !
  - 3 poles : pentest, reverse engineering, development
  - 4 sites : Toulouse, Paris, Lyon, Rennes

Table of Contents

Context

Smartphones

- Most targeted devices
- Many entry points
  - Browser
  - SMS/MMS
  - Instant messaging
  - Wifi/Bluetooth
  - Baseband

Target

- Samsung phone baseband : Shannon

Context

Smartphones

- Most targeted devices
- Many entry points
  - Browser
  - SMS/MMS
  - Instant messaging
  - Wifi/Bluetooth
  - Baseband

Target

- Samsung phone baseband : Shannon
- Galaxy s7

Table of Contents

Related Work

Previous work - Baseband exploitation

- Breaking Band – reverse engineering and exploiting the Shannon Baseband - *Nico Golde and Daniel Komaromy*
- A walk with Shannon - *Amat Cama*

Previous work - Baseband debugging

- Rétroconception et débogage d'un Baseband Qualcomm - *Guillaume Delugré*

Table of Contents

Table of Contents

Dedicated ARM Processor

Shannon Communication Processor

- Developed by Samsung
- Arm Cortex R7
- Used by all non-US Samsung phones

Firmware

- The file modem.bin is the code that runs on the Baseband
- Provided in Samsung Firmware
- Easy to load in IDA

Table of Contents

Shannon operating system - Operating system

Operating system

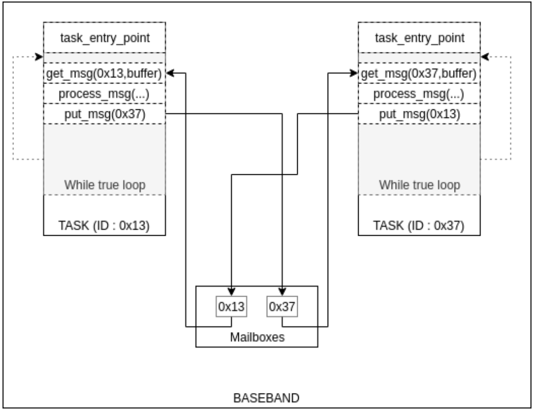
- Real Time OS
- Many tasks
  - Full mobile phone stack 2G - 5G
  - Communication with Application Processor
  - Communication with SIM cards
  - Remote File System

Shannon operating system - Tasks

Tasks

- Each task has the same structure
- Initialization
- While loop waiting for messages
- Communication between tasks using a mailbox system

Shannon operating system - Tasks



Mailboxes used for inter-tasks communications

Table of Contents

SYNACKTIV DIGITAL SECURITY

AP-CP Communications(1/2)

Communications

- Communications are done with shared memories and mailboxes

Mailboxes

- Each mailbox is used for one way communication
- Mailbox notifies the other processor with an interrupt
- 20 mailboxes are used by the GS7

AP-CP Communications(2/2)

SIPC5

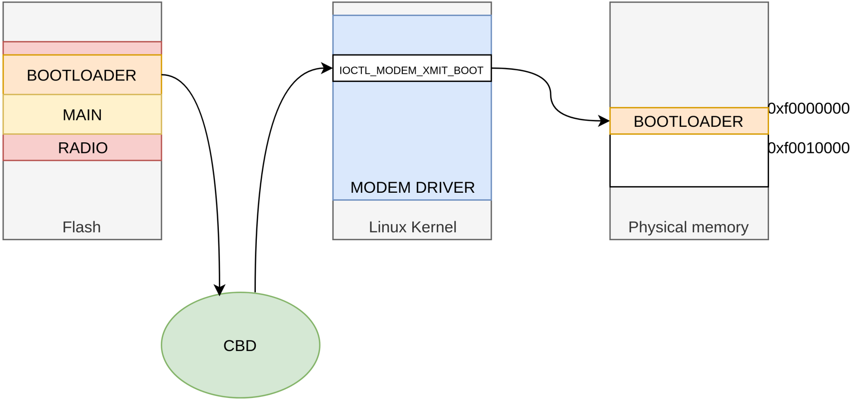
- CP and Linux Kernel communicate with a custom protocol called SIPC5
- Linux Kernel dispatches to user-space programs with char devices

Userland binaries

- Most of communications are done by 2 binaries
- cbd : boot and initialization of the Baseband firmware
- rild : Baseband communications, remote file system, OEM functionalities

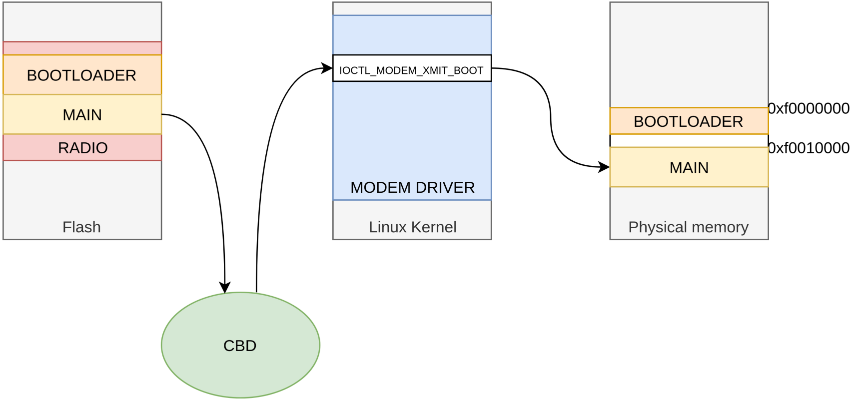
Table of Contents

Boot(1/6) - Copy of BOOTLOADER part from flash to physical memory



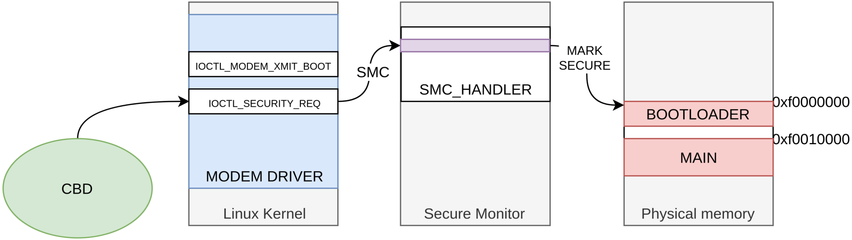
Copy of BOOTLOADER part from flash to physical memory

Boot(2/6) - Copy of MAIN part from flash to physical memory



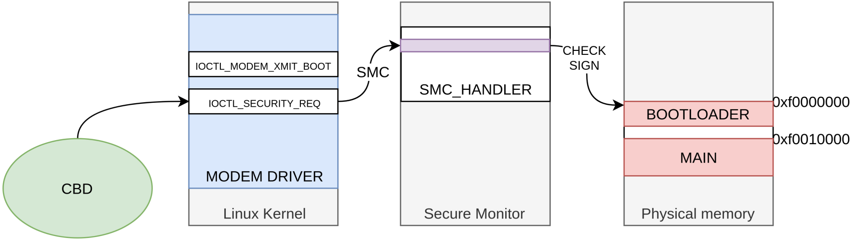
Copy of MAIN part from flash to physical memory

Boot(3/6) - Tag memory as Secure



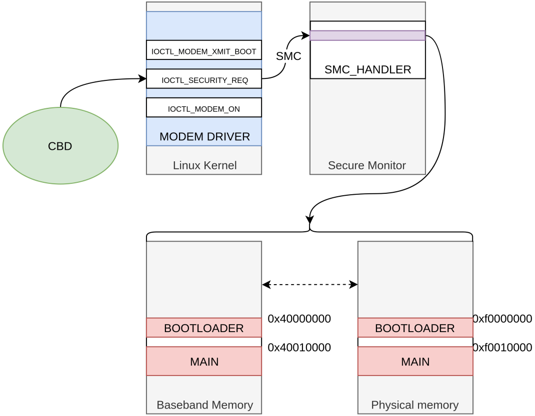
Tag memory as Secure

Boot(4/6) - Verify signature



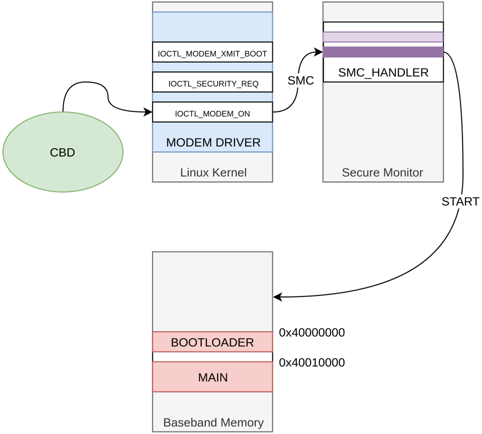
Verify signature

Boot(5/6) - Configure Baseband memory



Configure Baseband memory

Boot(6/6) - Start Baseband Processor



Start Baseband Processor

Table of Contents

Memory Map

- After the Baseband start, some code is copied and the memory layout is as follows

Memory Layout

- 0x00000000 - Bootloader
- 0x40010000 - Main (Shared with application Processor)
- 0x04000000 - Tightly Coupled Memory (Not shared)

Table of Contents

Table of Contents

Exploit a 1-day vulnerability : Secure World

Baseband Boot

- Baseband `MAIN` memory is marked as secure memory by the EL3 monitor
- Baseband firmware signature is checked by the EL3 monitor

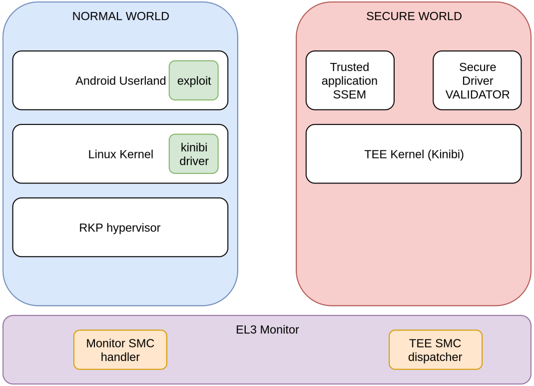
Required vulnerability

- A vulnerability in the Baseband itself : but cannot be used to debug the Baseband initialization and is Baseband firmware dependant.
- A vulnerability in the Secure World which allows to bypass the code signature. This kind of vulnerability is not Baseband firmware dependant, and permits to load newer and older Baseband versions on a vulnerable phone.

Available vulnerabilities

- Quarkslab has presented a chain of vulnerabilities at Blackhat-US 2019 that allows to gain code execution at the highest privilege on the AP : EL3 monitor => perfect chain for our objective.

Exploit a 1-day vulnerability : Exploit chain



Samsung S7 TrustZone software architecture

Exploit a 1-day vulnerability : Step 1 Trusted Application

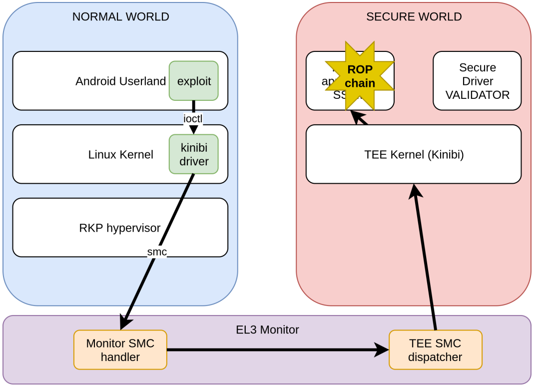
Trivial stack buffer overflow in `SSEM` Trusted Application

- Can be reached from Android Userland (need a favorable SElinux context / rooted phone)
- TEE Kernel does not implement ASLR
- This Trusted Application is built without stack cookies

Communication between TA and Secure Drivers

- Trusted Application can communicate with driver with IPC
- Driver may implement a whitelist of allowed TA

Exploit a 1-day vulnerability : Step 1 Trusted Application



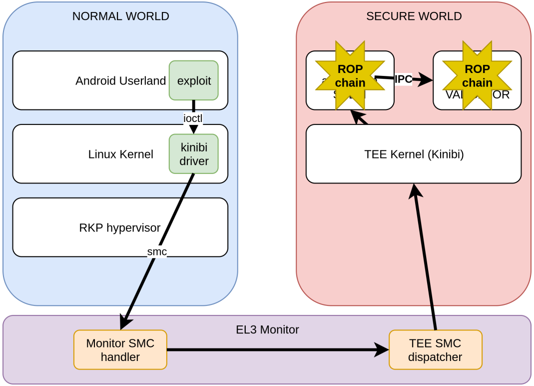
Gaining code execution in the `SSEM` TA.

Exploit a 1-day vulnerability : Step 2 Secure Driver

Trivial stack buffer overflow in `VALIDATOR` Secure Driver

- Can be reached from the `SSEM` Trusted Application
- Drivers are just Trusted Application with access to an extended API
- TEE kernel does not implement ASLR
- This Secure Driver is built without stack cookies

Exploit a 1-day vulnerability : Step 2 Secure Driver



Gaining code execution in the `VALIDATOR` Secure Driver.

Exploit a 1-day vulnerability : Step 3 TEE Kernel

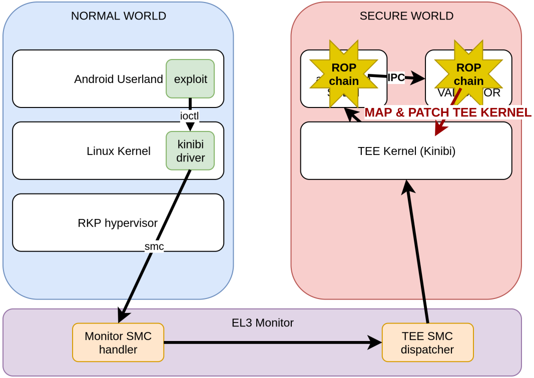
Driver API

- Driver can map physical addresses in their address space
- TEE Kernel deny some address ranges to be mapped

Physical memory access

- TEE Kernel forgot to denies itself to be mapped by Secure Drivers ....
- Exploit in `VALIDATOR` driver can map the TEE Kernel R/W
- TEE Kernel code can be live patched from the driver
  - adresses verification in the `map` syscall is NOP'ed
  - Driver can now map everything

Exploit a 1-day vulnerability : Step 3 TEE Kernel



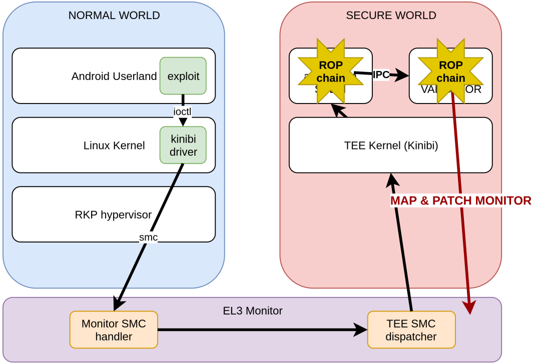
Patching TEE kernel from `VALIDATOR` driver

Exploit a 1-day vulnerability : Step 4 Secure Monitor

Secure Monitor patching

- After TEE Kernel patch, driver can map everything
- Signature check is disabled
- Function responsible of marking the Baseband memory secure is NOP’ed

Exploit a 1-day vulnerability : Step 4 Secure Monitor



Patching Secure Monitor from `VALIDATOR` driver

Exploit a 1-day vulnerability : Patched but still exploitable

Anti-rollback mechanism

- Vulnerable TA and Secure Driver are still loadable (no anti-rollback)
- TEE Kernel is still vulnerable on the latest firmwares (Galaxy S7)

Table of Contents

Patch the firmware : format

Firmware Format : TOC

The firmware starts with a header that contains information for memory segments :

- address where the section will be copied in the Baseband memory
- offset in the firmware file
- segment size

Segments in original firmware

- `BOOT`
- `MAIN`
- `NV`
- `OFFSET`

Patch the firmware : add a segment

Patches

- All the debugger code is injected in a new segment (after the `MAIN` segment)
- The debugger code starts with an interruption vector table
- `MAIN` segment is modified to change interruption handler addresses
- **This patched firmware can be loaded since Secure Monitor has been patched to remove the signature check**

Load the patched firmware

```
# stop the CBD service
setprop ctl.stop cpboot-daemon

# start a new one
cbd -d -tss310 -bm -mm -P ../../data/local/tmp/modem.bin
```

Table of Contents

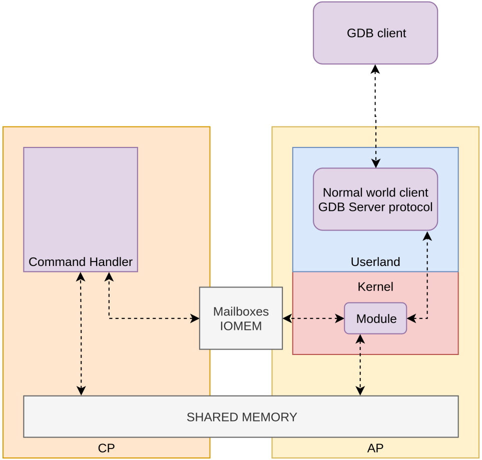
Table of Contents

Architecture

3 main components

- Debugger server that runs on the Android Userland
- A Linux driver that provides the debugger server to communicate with the Baseband
- Injected code in the Baseband that handles exceptions and communications from/to the debugger server.

Architecture - Overview



Communication between different components

Table of Contents

Catched Interrupts

Vector table offset	Name	Mode	catched / why
0x00	Reset	Supervisor	no
0x04	Undefined instruction	Undefined	**yes** (catch undefined instructions in debugger)
0x08	Software interrupt	Supervisor	**yes** (used for asserts by the Baseband, need to be catched)
0x0C	Prefetch Abort	Abort	**yes** (catch breakpoint and instruction fetch issue)
0x10	Data Abort	Abort	**yes** (catch memory issues)
0x18	IRQ	IRQ	**yes** (used for Communication)
0x1C	FIQ	FIQ	no

```
MAIN:40010084 18 01 01 40 off_40010084     DCD handle_reset
MAIN:40010088                ; int (*off_40010088)(void)
MAIN:40010088 A4 00 01 40 off_40010088     DCD handle_und_inst
MAIN:4001008C B0 00 01 40 off_4001008C     DCD handle_sw_intr
MAIN:40010090 BC 00 01 40 off_40010090     DCD handle_prefetch_abort
MAIN:40010094 CC 00 01 40 off_40010094     DCD vector_data_abort_0
MAIN:40010098 DC 00 01 40 off_40010098     DCD handle_reserved
MAIN:4001009C                ; int (*off_4001009C)(void)
MAIN:4001009C E0 00 01 40 off_4001009C     DCD handle_irq_0
MAIN:400100A0 F4 00 01 40 off_400100A0     DCD handle_fiq
MAIN:400100A4
```

Table of Contents

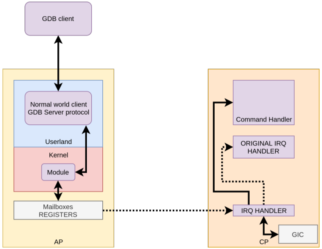
IRQ & Mailboxes

Communication mechanism

The debugger injected code uses the same mechanisms as the standard communication between AP and CP :

- A dedicated mailbox for CP->AP communications (Interrupts handled in Linux driver)
- A dedicated mailbox for AP->CP communications (Interrupts handled in IRQ handler)
- A dedicated shared memory area

IRQ & Mailboxes



IRQ handler

- Debugger server in the Android Userland uses IOCTL to write a command inside a mailbox
- Driver writes into the mailbox control register, an interrupt is generated on the CP side
- Injected code handles the IRQ, checks the current interrupt ID on the `GIC`
- Injected code handles the mailbox interrupt and redirect other interrupt to the original IRQ handler

IRQ & Mailboxes

Debugger commands

- Mailbox interrupt allows receiving commands from the debugger server
- The CP->AP mailbox is used to acknowledges the command

Signal notification

- Breakpoints / data abort / asserts / undefined instructions are handled in their respective interruption handler
- The CP->AP mailbox is used to notify the debugger server

Shared memory

Shared memory area

- A shared memory area is dedicated to communications between injected code and the debugger server.
- A zone already in the CP-AP shared memory is used (not used by the Baseband)

Memory synchronisation

- AP side : Cache flushes / sync with dedicated ARMv8 instructions
- CP side :
  - CP uses a PL310 cache controler, need to read/write some registers to perform cache flush/sync operations

Shared memory

Shared memory area

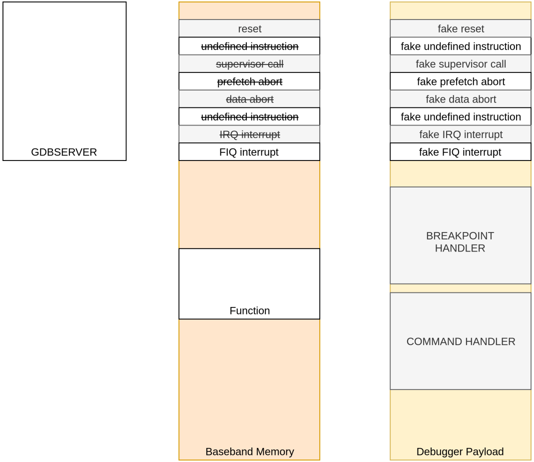
- A shared memory area is dedicated to communications between injected code and the debugger server.
- A zone already in the CP-AP shared memory is used (not used by the Baseband)

Memory synchronisation

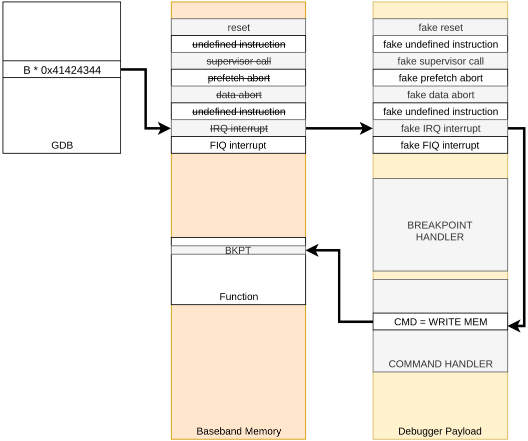
- AP side : Cache flushes / sync with dedicated ARMv8 instructions
- CP side :
  - CP uses a PL310 cache controler, need to read/write some registers to perform cache flush/sync operations
  - Cortex-R7 cache management instructions do not affect these caches

Table of Contents

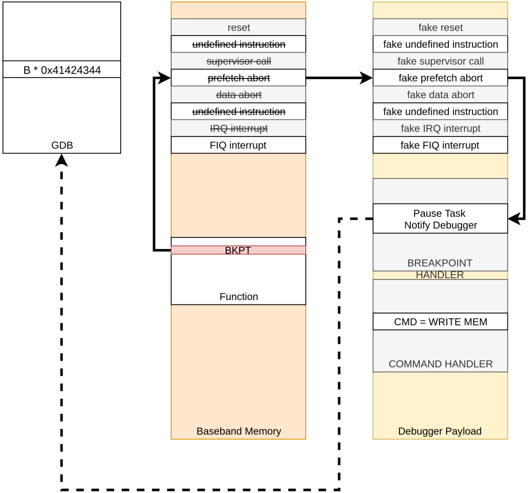
Insert Breakpoint



Insert Breakpoint



Handle Breakpoint



Continue Execution

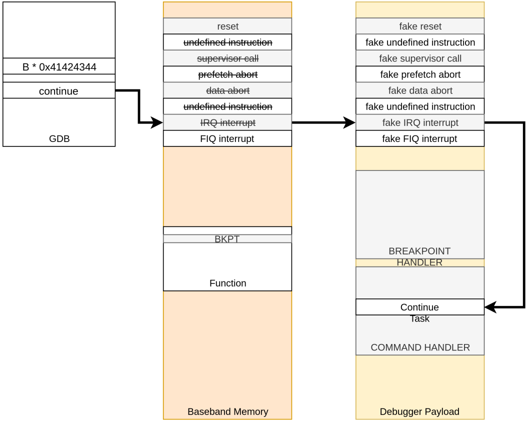


Table of Contents

GDBServer

- A binary is developed in Userland which implements the GDB Remote Serial Protocol

Some functionnalities are in the Baseband payload

- Read / Write memory
- Read / Write registers
- Stop Target
- Resume Target
- Continue

GDBServer - Stop/Non-Stop mode

2 modes for GDB

- Choosen by gdb client with command set non-stop on/off
- Both modes are handled by the gdbserver provided

Stop mode

- The whole Baseband is debugged as one single program
- Problems with inter-task communication and watchdog

Non-stop mode

- Each Baseband task is a thread for gdb
- Each task is handled separately
- All stops reply by the gdb server are asynchronous

Table of Contents

Improvements

Work in progress

- Handle multiple breakpoints in multiple tasks
- Watchpoints
- Watchdogs
- Performance

Support

- EL3 patching is done for version `G930FXXS6ESJ2`
- Older and newer CP version can be load on this version
- Some offset to adjust to support another version

Table of Contents

Table of Contents

Basic debugger functionning

Functionnality

- Read/Write mem
- List tasks
- Insert breakpoint
- Backtrace

Basic debugger functionning - Demo

Table of Contents

Table of Contents

Modification of a Nas packet

Breakpoint

- Function responsible for sending GMM : `mm_SendGmmMessage`
- Modify the content of the message buffer
- Continue Execution

Modification of a Nas packet - GDB Script

```
target remote :1337
b * 0x40CC7010
commands
    set $type = *(unsigned char *) ($r0+0x1+4)
    printf "[i] GMM type : 0x%02x\n", $type
    if($type==0x16)
        printf "[i] Modifying identity response ...\n"
        set *(unsigned long long*) ($r0+4+3) = 0x32344b43414e5953
        printf "[+] modified IMSI : SYNACK42\n"
        del br 1
    end

    continue
end
continue
```

Modification of a Nas packet - Demo

Table of Contents

Table of Contents

Conclusion

Conclusion

- Now Shannon Baseband can be debugged
- Still work to do for a full featured debugger
- Code provided allows to execute code as the Baseband
- Code will be available on Synacktiv's github

QUESTIONS ?

Contact :

- david.berard@synacktiv.com
- vincent.fargues@synacktiv.com

MERCI DE VOTRE ATTENTION