How to design a Baseband debugger

SSTIC 2020
# Table of Contents

1. Introduction
2. Shannon architecture
3. Debugger injection
4. Debugger development
5. Examples of use
6. Conclusion
Table of Contents

1 Introduction

- Who are we
  - Context
  - Related Work
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

1 Introduction

- Who are we
- Context
- Related Work
### 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
## 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>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Shannon architecture</td>
</tr>
<tr>
<td>3</td>
<td>Debugger injection</td>
</tr>
<tr>
<td>4</td>
<td>Debugger development</td>
</tr>
<tr>
<td>5</td>
<td>Examples of use</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
</tr>
</tbody>
</table>
Table of Contents

2 Shannon architecture

- Dedicated ARM Processor

- Shannon operating system
- AP-CP Communications
- Boot
- Memory Map
**Dedicated ARM Processor**

<table>
<thead>
<tr>
<th>Shannon Communication Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Developed by Samsung</td>
</tr>
<tr>
<td>- Arm Cortex R7</td>
</tr>
<tr>
<td>- Used by all non-US Samsung phones</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Firmware</th>
</tr>
</thead>
<tbody>
<tr>
<td>- The file modem.bin is the code that runs on the Baseband</td>
</tr>
<tr>
<td>- Provided in Samsung Firmware</td>
</tr>
<tr>
<td>- Easy to load in IDA</td>
</tr>
</tbody>
</table>
Table of Contents

2 Shannon architecture
   ■ Dedicated ARM Processor

■ Shannon operating system
   ■ AP-CP Communications
   ■ Boot
   ■ Memory Map
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**

<table>
<thead>
<tr>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each task has the same structure</td>
</tr>
<tr>
<td>Initialization</td>
</tr>
<tr>
<td>While loop waiting for messages</td>
</tr>
<tr>
<td>Communication between tasks using a mailbox system</td>
</tr>
</tbody>
</table>
Shannon operating system - Tasks

Mailboxes used for inter-tasks communications
Table of Contents

2 Shannon architecture
   ■ Dedicated ARM Processor

■ Shannon operating system
■ AP-CP Communications
■ Boot
■ Memory Map
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
### 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>
<thead>
<tr>
<th></th>
<th>Shannon architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Dedicated ARM Processor</td>
</tr>
</tbody>
</table>

- Shannon operating system
- AP-CP Communications
- Boot
- Memory Map
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
Boot(4/6) - Verify signature
Boot(5/6) - Configure Baseband memory

Configure Baseband memory
Boot(6/6) - Start Baseband Processor

Start Baseband Processor
Table of Contents

2 Shannon architecture

- Dedicated ARM Processor

- Shannon operating system
- AP-CP Communications
- Boot
- 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

1. Introduction  
2. Shannon architecture  
3. Debugger injection  
4. Debugger development  
5. Examples of use  
6. Conclusion
Table of Contents

3 Debugger injection

- Exploit a 1-day vulnerability
- Baseband code injection
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
  - addresses 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

3 Debugger injection

- Exploit a 1-day vulnerability
- Baseband code injection
**Patch the firmware : format**

<table>
<thead>
<tr>
<th>Firmware Format : TOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>The firmware starts with a header that contains information for memory segments :</td>
</tr>
<tr>
<td>- address where the section will be copied in the Baseband memory</td>
</tr>
<tr>
<td>- offset in the firmware file</td>
</tr>
<tr>
<td>- segment size</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Segments in original firmware</th>
</tr>
</thead>
<tbody>
<tr>
<td>- BOOT</td>
</tr>
<tr>
<td>- MAIN</td>
</tr>
<tr>
<td>- NV</td>
</tr>
<tr>
<td>- OFFSET</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th></th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Shannon architecture</td>
</tr>
<tr>
<td>3</td>
<td>Debugger injection</td>
</tr>
<tr>
<td>4</td>
<td>Debugger development</td>
</tr>
<tr>
<td>5</td>
<td>Examples of use</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
</tr>
</tbody>
</table>
Table of Contents

Debugger development
  ■ Architecture
  ■ Interrupts Handler

Communication
  ■ Breakpoint handling
  ■ GDBServer
  ■ Improvements
## 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

4 Debugger development

- Architecture
- Interrupts Handler

- Communication
- Breakpoint handling
- GDBServer
- Improvements
## Caught Interrupts

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

```
MAIN:40010004 18 01 01 40 off_40010004  DCD handle_reset
MAIN:40010005 ; int (*off_40010005)(void)
MAIN:40010005 A4 00 01 40 off_40010005  DCD handle_und_inst
MAIN:4001000C E0 00 01 40 off_4001000C  DCD handle_sw_intr
MAIN:4001009A DC 00 01 40 off_4001009A  DCD vector_data_abort_0
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
```
Table of Contents

4 Debugger development
  - Architecture
  - Interrupts Handler

- Communication
  - Breakpoint handling
  - GDBServer
  - Improvements
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

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 controller, need to read/write some registers to perform cache flush/sync operations
### 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 controller, 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

4 Debugger development
   ▪ Architecture
   ▪ Interrupts Handler

   ▪ Communication
   ▪ Breakpoint handling
   ▪ GDBServer
   ▪ Improvements
Insert Breakpoint

**GDB**

- reset
- undefined instruction
- supervisor-call
- prefetch abort
- data-abort
- undefined instruction
- IRQ-interrupt
- FIQ interrupt

**Baseband Memory**

**BREAKPOINT HANDLER**

- CMD = WRITE MEM
- COMMAND HANDLER

**Debugger Payload**

- fake reset
- fake undefined instruction
- fake supervisor call
- fake prefetch abort
- fake data abort
- fake undefined instruction
- fake IRQ interrupt
- fake FIQ interrupt

**B * 0x41424344**
Handle Breakpoint
Continue Execution
<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Debugger development</td>
</tr>
<tr>
<td>- Architecture</td>
</tr>
<tr>
<td>- Interrupts Handler</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
A binary is developed in Userland which implements the GDB Remote Serial Protocol

Some functionalities are in the Baseband payload

- Read / Write memory
- Read / Write registers
- Stop Target
- Resume Target
- Continue
**GDBServer - Stop/Non-Stop mode**

<table>
<thead>
<tr>
<th>2 modes for GDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Chosen by gdb client with command set non-stop on/off</td>
</tr>
<tr>
<td>- Both modes are handled by the gdbserver provided</td>
</tr>
</tbody>
</table>

**Stop mode**

<table>
<thead>
<tr>
<th>Stop mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>- The whole Baseband is debugged as one single program</td>
</tr>
<tr>
<td>- Problems with inter-task communication and watchdog</td>
</tr>
</tbody>
</table>

**Non-stop mode**

<table>
<thead>
<tr>
<th>Non-stop mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Each Baseband task is a thread for gdb</td>
</tr>
<tr>
<td>- Each task is handled separately</td>
</tr>
<tr>
<td>- All stops reply by the gdb server are asynchronous</td>
</tr>
</tbody>
</table>
# Table of Contents

4 Debugger development
- Architecture
- Interrupts Handler

Communication
- Breakpoint handling
- GDBServer
- Improvements
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

1 Introduction
2 Shannon architecture
3 Debugger injection
4 Debugger development
5 Examples of use
6 Conclusion
Table of Contents

Examples of use

- Basic debugger functioning
- Logs enabling
- Modification of a Nas packet
Basic debugger functioning

**Functionality**
- Read/Write mem
- List tasks
- Insert breakpoint
- Backtrace
Basic debugger functioning - Demo
## Table of Contents

5 Examples of use

- Basic debugger functioning
- Logs enabling
- Modification of a Nas packet
Table of Contents

5 Examples of use

- Basic debugger functioning
- Logs enabling
- Modification of a Nas packet
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 ...
"
    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

1. Introduction
2. Shannon architecture
3. Debugger injection
4. Debugger development
5. Examples of use
6. Conclusion
# Table of Contents

- Conclusion

6 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