Emulating Samsung’s Baseband for Security Testing

Grant Hernandez & Marius Muench

Tyler Tucker, Hunter Searle, Weidong Zhu
Patrick Traynor & Kevin Butler
# Intros

Grant Hernandez
- Recent Ph.D. graduate (University of Florida)
- Works on Android security and firmware analysis
- Just joined Qualcomm’s product security team

Marius Muench
- Ph.D. graduate (EURECOM)
- By now PostDoc @ VUSec
- Builds tooling for better dynamic analysis of embedded systems
# Outline

- Motivation
- Previous baseband work
- Initial journey in baseband vulnerability research
- Reverse engineering Samsung’s baseband firmware
- Building a baseband emulator
- Emulator based fuzzing with AFL
- n-day and 0-day vulnerability discovery
- Crashing basebands over-the-air
- Conclusions
What is a baseband processor?

- A dedicated device that implements the 2G - 5G cellular protocols
- Seamlessly routes mobile data, calls, SMS, and more while on the go
- The “phone” part of your smartphone
- Each vendor below implements their own baseband
- Typically runs embedded firmware using a Real Time Operating System (RTOS)
- Separate CPU core(s) from the application processor
# Why basebands?

- Large attack surface due to multi-mode support of complicated standards
- Some components you can expect to find in a baseband:
  - Custom DSPs and radio drivers
  - Multiple ASN.1 decoders
  - Custom IP stack
  - Multiple voice & video codecs
  - X.509 parsing
  - DNS parsing
  - ...and plenty more obscure protocols unique to cellular
- Remote baseband attacks can be devastating
  - Silent full call & data MITM, independent of operating system
  - Escalation to application processor
# Samsung’s “Shannon” Baseband

- Present on Samsung smartphones with the EXYNOS chipset (non US phones)
- 2G - 5G “multi-mode” baseband
- Uses ARM Cortex-R ISA
- An interesting vulnerability research target
  - Previously demonstrated over-the-air attacks
  - Less exploit mitigations than application processor
- 2016: Breaking Band (Nico Golde, Daniel Komaromy - REcon)
- 2018: A Walk with Shannon (Amat Cama - Infiltrate)
- 2020: How to Design a Baseband Debugger (David Berard, Vincent Fargues - SSTIC)
Other baseband exploit work

- 2009: Fuzzing the Phone in Your Phone (Collin Mulliner, Charlie Miller - BHUSA)
- 2011: SMS of Death (Collin Mulliner, Nico Golde, J.P. Seifert - USENIX SEC)
- 2011: Baseband Attacks (Ralf-Phillip Weinman - WOOT)
- 2017: Path of Least Resistance: Cellular Baseband to Application Processor Escalation on Mediatek Devices (György Miru - Comsecuris Blog)
- 2018: Exploitation of a Modern Baseband (Marco Grassi, et al. - BHUSA)
- 2018: There is life in the Old Dog yet (Nico Golde - Comsecuris Blog)
- 2019: Touching the Untouchables (LTEFuzz) (Kim et al. (KAIST) - IEEE S&P)
- 2019: Exploiting Qualcomm WLAN and Modem OTA (QualPwn) (Tencent Blade - BHUSA)
- 2020: BaseSAFE: Baseband SAnitized Fuzzing through Emulation (Maier et al. - WiSec)
- ...and more -- but not much more
# Introducing ShannonEE

- An Emulation Environment for the Shannon baseband
- Executes baseband firmware directly - no pre-processing required
- No physical devices required, scalable to as many cores as you have available
- Cold boots baseband and brings up most RTOS tasks

Core Features:
- Python API for prototyping
- ModKit & FFI for extending & exploring the baseband
- Integrated coverage-guided fuzzing with system-level AFL
- GDB for triaging crashes
- Support for multiple SoC versions
Quick Demo
How did we get here?
Early research attempts (May 2019)

- Static vs. dynamic analysis
- We started on a project to fuzz basebands over-the-air
  - No experience with SDR / cellular, but lots of reversing and exploitation knowledge
  - How hard could it be?
- Cellular protocols standards are pretty much one big recursive acronym
- Wrestled with radios and base station code for months before finally getting a working setup

- Test setup:
  - Modified a 2G base station’s GSM / GPRS downlink (L2 -> L1) packet paths to randomly bitflip packets
  - Targeted a mix of phones, created an ADB logcat monitor for crashes
# Crashes without a clue

We got crashes, but no modem dumps due to root!
Next to impossible to identify root cause
Crashes without a clue

We got crashes, but no modem dumps due to root!
Next to impossible to identify root cause
So you want to fuzz basebands?

- **We don't** recommend **OTA live** fuzzing at all!
- Researchers developed fuzzers and found bugs, but:
  - basebands are more fragile than you think: hangs and weird behavior are normal during test
  - often implement spec loosely or only subset
  - state machines are complex, especially in error/repetition cases
  - a significant amount of corruptions do not result in good crashes

We got crashes, but no modem dumps due to root! Next to impossible to identify root cause
Modem stack trace is suppressed if device is unlocked (OS_Fatal_Error)

To get modem dumps we need root, but rooting suppresses modem logs
How do we find & debug memory corruptions?

- Modem anti-debug makes on-target difficult without other exploits
- Manual reverse engineering is an option, but basebands are huge (doesn’t scale)
  - See [https://github.com/grant-h/shannon_s5000/](https://github.com/grant-h/shannon_s5000/)
- Ideal setup:
  - GDB
  - Coverage guided fuzzing
  - Snapshotting
- The most viable approach for scaling and automating our research is emulation
Initial Firmware Reversing
# Extracting modem.bin

- Firmware available at SamMobile
- Purchased Samsung S10 for research and targeted SM-G973F firmware

- Previous researchers and papers mentioned that modem.bin is encrypted
  - None of our images were \(\_\_\_\_\_\_\_\_\\)\

- Well reverse engineered already: [https://github.com/Comsecuris/shannonRE](https://github.com/Comsecuris/shannonRE)
<table>
<thead>
<tr>
<th>Entry Name</th>
<th>File Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOC</td>
<td></td>
</tr>
<tr>
<td>...@</td>
<td>W ....</td>
</tr>
<tr>
<td>BOOT</td>
<td></td>
</tr>
<tr>
<td>...@@</td>
<td>.W ....</td>
</tr>
<tr>
<td>MAIN</td>
<td>`&quot; ....</td>
</tr>
<tr>
<td><a href="mailto:...@.yT">...@.yT</a>.?</td>
<td>. ....</td>
</tr>
<tr>
<td>VSS</td>
<td>.T ....</td>
</tr>
<tr>
<td>...G`.]...)</td>
<td></td>
</tr>
<tr>
<td>NV</td>
<td>. ....</td>
</tr>
<tr>
<td>OFFSET</td>
<td>...V ....</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Load Address</th>
<th>Size</th>
<th>CRC</th>
<th>Count/Entry ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>544f 4300</td>
<td>0000</td>
<td>0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0080 0040</td>
<td>1004</td>
<td>0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>4d41 494e</td>
<td>0000</td>
<td>0000</td>
<td>0000 6022 0000</td>
</tr>
<tr>
<td>0000 0140</td>
<td>a079</td>
<td>5402</td>
<td>3fb1 20ef 0200 0000</td>
</tr>
<tr>
<td>5653 5300</td>
<td>0000</td>
<td>0000</td>
<td>0000 0000 009c 5402</td>
</tr>
<tr>
<td>0000 8047</td>
<td>60f6</td>
<td>5d00</td>
<td>04e5 2907 0300 0000</td>
</tr>
<tr>
<td>4e56 0000</td>
<td>0000</td>
<td>0000</td>
<td>0000 0000 0000 0000</td>
</tr>
<tr>
<td>0000 6045</td>
<td>0000</td>
<td>1000</td>
<td>0000 0000 0400 0000</td>
</tr>
<tr>
<td>4f46 4653</td>
<td>4554</td>
<td>0000</td>
<td>0000 0000 00aa 0700</td>
</tr>
<tr>
<td>0000 0056</td>
<td>0800</td>
<td>0000</td>
<td>0000 0500 0000</td>
</tr>
<tr>
<td>Entry Name</td>
<td>Load Address</td>
<td>Size</td>
<td>CRC</td>
</tr>
<tr>
<td>------------</td>
<td>--------------</td>
<td>------</td>
<td>-----</td>
</tr>
<tr>
<td>544f 4300</td>
<td>0080 0040</td>
<td>1004</td>
<td>0000</td>
</tr>
<tr>
<td>424f 4f54</td>
<td>0000 0140</td>
<td>401e</td>
<td>d597</td>
</tr>
<tr>
<td>4d41 494e</td>
<td>0000 5140</td>
<td>5a65</td>
<td>3fb1</td>
</tr>
<tr>
<td>5653 5300</td>
<td>0000 8047</td>
<td>60f6</td>
<td>04e5</td>
</tr>
<tr>
<td>4e56 0000</td>
<td>0000 6045</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>4f46 4653</td>
<td>0000 0000</td>
<td>0056</td>
<td>0800</td>
</tr>
</tbody>
</table>
Looks like ARM “always” conditionals!
Let’s disassemble from here :)

```asm
...           ...           ...
00000420: 3c00 00ea d8f1 9fe5 d8f1 9fe5 d8f1 9fe5 < ............
00000430: d8f1 9fe5 feff ffea d4f1 9fe5 d4f1 9fe5 ........$
00000440: f81b 0000 fc1b 0000 241c 0000 0000 0000 .LBT ........
00000450: 004c 4254 0000 0000 0000 0000 0000 0000 ........
00000460: 0000 0000 0000 0000 0000 0000 0000 0000 ........
```

...
# Exception Vectors

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Destination</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>40000000</td>
<td>3c 00 00 ea</td>
<td>b</td>
<td>boot_RESET</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Flow Override:</strong> CALL_RETURN (CALL_TERMINATOR)</td>
</tr>
<tr>
<td>LAB_40000004</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40000004</td>
<td>d8 f1 9f e5</td>
<td>ldr</td>
<td>pc=&gt;boot_UDI,[DAT_400001e4]</td>
</tr>
<tr>
<td>40000008</td>
<td>d8 f1 9f e5</td>
<td>ldr</td>
<td>pc=&gt;boot_SWI,[DAT_400001e8]</td>
</tr>
<tr>
<td>4000000c</td>
<td>d8 f1 9f e5</td>
<td>ldr</td>
<td>pc=&gt;boot_PREFETCH,[DAT_400001ec]</td>
</tr>
<tr>
<td>40000010</td>
<td>d8 f1 9f e5</td>
<td>ldr</td>
<td>pc=&gt;boot_DATA_ABORT,[DAT_400001f0]</td>
</tr>
<tr>
<td>boot_NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40000014</td>
<td>fe ff ff ea</td>
<td>b</td>
<td>boot_NA</td>
</tr>
<tr>
<td>40000018</td>
<td>d4 f1 9f e5</td>
<td>ldr</td>
<td>pc=&gt;boot_IRQ,[DAT_400001f4]</td>
</tr>
<tr>
<td>4000001c</td>
<td>d4 f1 9f e5</td>
<td>ldr</td>
<td>pc=&gt;boot_FIQ,[DAT_400001f8]</td>
</tr>
</tbody>
</table>
# Shannon Boot Mode

- Bootloader debug prints using UART
  - Saved to early boot log, can be dumped from kernel as well
- DUMP mode activated during crash dump
- BOOT mode for normal startup
- Who signals these boot modes?

```c
if (boot_mode == DUMP_MODE) {
  boot_unk_common_setup();
  uart_putchar('#');
  boot_crash_or_dump();
  boot_unk_crash();
  uart_putchar(s_Done_40000550);
  FUN_40000d84(&DAT_00002e00);
}
else {
  if (boot_mode == BOOT_MODE) {
    boot_unk_common_setup();
    uart_putchar('#');
    boot_prepare_mpu_next_ram();
    uart_putchar(s_Boot_40000568);
    nextFnPointer = boot_comm_ap();
    if ((void*)0x1000000 < nextFnPointer) {
      boot_stage2(nextFnPointer);
      goto LAB_400004f0;
    }
    r0 = s_Xxx!_40000570;
  }
  else {
    r0 = s_Unknown_4000055c;
  }
  uart_putchar(r0);
}
```
A side-quest to the Samsung Kernel

- Examined the Samsung S10's kernel sources to better understand the modem's boot
  - Samsung S10 Kernel - https://github.com/grant-h/SM-G973F-Kernel
- drivers/misc/modem_v1
  - modem_io_device.c - dev node ioctl handler (for booting), read/write for commands
  - link_device_memory.h - Contains structs on shared memory regions
- Cellular Boot Daemon (CBD) communicates with /dev/umts_boot0 (modem_v1)
  - IOCTL_MODEM_RESET - Reset and await new boot
  - IOCTL_SECURITY_REQUEST - Establish modem bootloader secure boot
  - IOCTL_MODEM_ON, IOCTL_MODEM_BOOT_ON - Start boot process
  - IOCTL_MODEM_DL_START - Download code to modem memory
  - IOCTL_MODEM_BOOT_OFF - Finalizes boot process
Modem driver in Linux maps shared memory region between CP and AP for DMA-based IPC (ring buffers)

Commands are queued in a full-duplex fashion

Commands come from Samsung RIL

```
struct __packed shmem_4mb_phys_map {
    u32 magic;
    u32 access;

    u32 fmt_tx_head, fmt_tx_tail;
    u32 fmt_rx_head, fmt_rx_tail;

    u32 raw_tx_head, raw_tx_tail;
    u32 raw_rx_head, raw_rx_tail;

    ... char fmt_tx_buff[SHM_4M_FMT_TX_BUFF_SZ];
    ... char fmt_rx_buff[SHM_4M_FMT_RX_BUFF_SZ];
}
```
Peripherals

ShannonOS

Host Interface (HIF)

GSM UMTS LTE

RFCTRL

CP (Cortex-R)

Interprocessor Interrupts

SMC (BP Secure Boot)

Samsung EXYNOS Platform Services

MARCONI DSP

RFIC

USIM

UART

USI (I2C/SPI)

SIPC MBOX

Shared Memory Link (SHM)

Modem Interface (MIF)

RIL CBD RFS

Android OS

AP (Cortex-A Series)

Linux
Emulation: The first steps
# Choose your weapon

avatar² + QEMU
Displaying the Boot UART

```python
class UARTPeripheral(ShannonPeripheral):
    def hw_read(self, offset, size):
        if offset == 0x18:
            return self.status
        return 0
    def hw_write(self, offset, size, value):
        if offset == 0:
            sys.stderr.write(chr(value & 0xff))
            sys.stderr.flush()
        else:
            self.log_write(value, size, "UART")
        return True
    def __init__(self, name, address, size, **kwargs):
        super(UARTPeripheral, self).__init__(name, address, size)
        self.status = 0
        self.write_handler[0:size] = self.hw_write
        self.read_handler[0:size] = self.hw_read

self.create_peripheral(UARTPeripheral, 0x84000000, 0x1000, name='boot_uart')
```
Early Boot Support

https://drive.google.com/file/d/1hrKDB6m0LnSYJ4WzmgpYBqvmlebcyJcw/view?usp=sharing
Snapshot support

- Piece of cake with Avatar’s remote QMP protocol support
- Care needed to snapshot Avatar PyPeripherals alongside QEMU memory

```python
def snapshot(self, snapshot_name):
    monitor = self.qemu.protocols.monitor

    log.info("Performing snapshot " + snapshot_name)

    peripherals = {}
    for mem in self.avatar.memory_ranges:
        if hasattr(mem.data, 'python_peripheral'):
            per = mem.data.python_peripheral
            log.info("Snapshotting " + str(per))
            peripherals[mem.begin] = per

    with open('avatar-snapshot-%s' % snapshot_name, 'wb') as fp:
        pickle.dump(peripherals, fp)

    log.info("Snapshotting qemu state...")
    result = monitor.execute_command('human-monitor-command',
        {"command-line": "savevm %s" % snapshot_name})
    log.info("Snapshot result: " + result)

    self.qemu.cont(blocking=False)
```
The tooling so far

Avatar <-> QEMU IPC is slow
GDB hooks and remote memory are the primary bottleneck
*hacking montage*
Moving to Panda

- Platform for Architecture-Neutral Dynamic Analysis
- First release in 2013
- Modified QEMU
  - Record & Replay Infrastructure
  - Plugin infrastructure
  - Callbacks to QEMU runtime state
- Already integrated in avatar2-infrastructure
- Still has the same performance issues!
# Enter PyPanda

- import panda -> QEMU as a thread
  - No more IPC!
  - Single python interpreter with FFI for PANDA C functions
  - Can replace all GDB hooks with native panda hooks in Python
- HUGE speedup!
Emulating Peripherals: pal_init1()

- A huge monolith function that starts all modem subsystems and tasks
  - Activates malloc heap
  - Loads NV items
  - Starts timers
  - Initializes DSP(s)
  - Starts all tasks
- Iteratively emulated peripherals by watching for crash strings or infinite loops
- MMIO monitoring used to see which peripherals needed more modeling
- Simple automated cyclic-bit pattern heuristic
- Partially emulated: PMIC, CLK, DSP, SOC, SIPC/SHM, TIMER, GIC

```
00000040: 4d41 494e 0000 0000 0000 0000 6022 0000
00000050: 0000 0140 a079 5402 3fb1 20ef 0200 0000
```

MAIN........`"...
...@.yT.?.....
We reached the banner!

[INFO] OS_Create_Event_Group(0x4177b710, ACPM_PMIC_RX_EVENT)

[INFO] OS_Create_Event_Group(0x4177b710, ACPM_PMIC_RX_EVENT)

DEVELOPMENT PLATFORM
- ARM Emulation Baseboard | Cortex-R7
- Software Build Date: Jul 19 2019 05:03:07
- Software Builder:
- Compiler Version: ARM RVCT 50.6 [Build 422]

Platform Abstraction Layer (PAL) Powered by CP Platform Part
Fuzz’n’Roll
Enter Triforce-AFL

- Originally developed in 2017
- Invokes AFL from within the guest via hypercalls
- We combined the patch-set with some afl++ additions
  - Better coverage-collection
  - Persistent mode
- Fuzzing from within using the AFL Task
Using the ModKit for the AFL Task

- Wrote our own ModKit
- Inject custom tasks inside the emulated baseband
- C-based compiled modules
- Dynamic baseband symbol resolution for a nice FFI
  - Provides access to recovered Shannon symbols
- Mods dynamically linked into RWX page in baseband memory
# Task Targeting

- Target the task message queues
- Increases accuracy as significant processing before radio payloads
  - No need for manual rehosting of functions, just need the right message IDs
- Inter-task messaging
  - `pal_SendMsg(QID, msg)`
  - `pal_RecvMsg(QID)`

```c
struct qitem_header {
    union {
        struct {
            uint16_t src; // source queue
            uint16_t dst; // destination queue
        }
        uint32_t dir;
    }
    uint16_t size; // size of the payload
    uint16_t msgId; // [msgGroup:8][msgNumber:8]
} PACKED;
```
const char TASK_NAME[] = "AFL_GSM_SM\0";
static pal_qid_t qid;
// called once before forkserv
int fuzz_single_setup()
{
    // dynamically look up the QID
    qid = queuename2id("SM");

    struct qitem_sm * init = malloc(sizeof(struct qitem_sm));

    init->header.size = 0;
    // 0x3407 SMREG_INIT_REQ
    init->header.msgId = 0x3407;
    pal_SendMsg(qid, init);

    return 1;
}
// called for each test case
void fuzz_single()
{
    uint32_t input_size;
    uint16_t size;

    struct qitem_sm * item =
        malloc(sizeof(struct qitem_sm));
    char * pdu = malloc(AFL_MAX_INPUT);
    item->pdu = pdu;

    char * buf = getWork(&input_size);

    item->header.msgGroup = 0x3414;
    item->header.size = size;

    memcpy(item->pdu, buf, size);

    // memory range to collect coverage
    startWork(0, 0xffffffff);

    // immediately reschedules to target task
    pal_SendMsg(qid, item);
    // returns here when task done processing
    doneWork(0);
}
<table>
<thead>
<tr>
<th>Process Timing</th>
<th>Overall Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run time: 0 days, 0 hrs, 0 min, 52 sec</td>
<td>Cycles done: 0</td>
</tr>
<tr>
<td>Last new path: 0 days, 0 hrs, 0 min, 0 sec</td>
<td>Total paths: 93</td>
</tr>
<tr>
<td>Last uniq crash: none seen yet</td>
<td>Unique crashes: 0</td>
</tr>
<tr>
<td>Last uniq hang: 0 days, 0 hrs, 0 min, 27 sec</td>
<td>Unique hangs: 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle Progress</th>
<th>Map Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now processing: 1 (1.08%)</td>
<td>Map density: 942 (0.04%)</td>
</tr>
<tr>
<td>Paths timed out: 0 (0.00%)</td>
<td>Count coverage: 1.64 bits/tuple</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stage Progress</th>
<th>Findings in Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now trying: calibration</td>
<td>Favored paths: 11 (11.83%)</td>
</tr>
<tr>
<td>Stage execs: 38/40 (95.00%)</td>
<td>New edges on: 52 (55.91%)</td>
</tr>
<tr>
<td>Total execs: 18.8k</td>
<td>Total crashes: 0 (0 unique)</td>
</tr>
<tr>
<td>Exec speed: 960.0/sec</td>
<td>Total hangs: 28 (2 unique)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fuzzing Strategy Yields</th>
<th>Path Geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit flips: n/a, n/a, n/a</td>
<td>Levels: 2</td>
</tr>
<tr>
<td>Byte flips: n/a, n/a, n/a</td>
<td>Pending: 93</td>
</tr>
<tr>
<td>Arithmetics: n/a, n/a, n/a</td>
<td>Pend fav: 11</td>
</tr>
<tr>
<td>Known ints: n/a, n/a, n/a</td>
<td>Own finds: 80</td>
</tr>
<tr>
<td>Dictionary: n/a, n/a, n/a</td>
<td>Imported: 0</td>
</tr>
<tr>
<td>Havoc: 0/0, 0/0</td>
<td>Variable: 91</td>
</tr>
<tr>
<td>Trim: 0.00%/4, n/a</td>
<td></td>
</tr>
</tbody>
</table>

[https://drive.google.com/file/d/1zGFQmlkJ7VuT-K-wDInJo4nhwJl7GpvL/view?usp=sharing](https://drive.google.com/file/d/1zGFQmlkJ7VuT-K-wDInJo4nhwJl7GpvL/view?usp=sharing)
Bugs, Bugs, Bugs!
N-day Rediscovery

- Let’s rediscover an n-day in Shannon
- Targeted ~2017 Galaxy S7 firmware (Amat’s research environment)
- Built harness for GSM & GPRS radio packet handlers
  - Call control (CC)
  - Mobility Management (MM)
  - Session management (SM)

- AFL automatically rediscovered the GPRS PDP_NETWORK_ACCEPT crash from “A Walk with Shannon”!
# 0–day Fuzzing

- **Target**: the GSM and GPRS protocol stacks (2 and 2.5G)
- **Why 2G?**
  - Lowest hanging fruit in baseband,
  - Trivial fake base station based attack model (no mutual authentication in 2G)
- Coverage guided fuzzing generated test cases (blank initial seeds)
- Coverage debugging using PANDA and plotting using GHIDRA’s Dragondance

- Ran across 30 CPU cores for 5 days of CPU time (highly scalable)
- **Rediscovered two n-days (S7 Edge) and one 0-day (S10) vulnerability!**
  - In disclosure process with Samsung
- LTE is next on our list :)

In disclosure process with Samsung
“Call of Death” n-day

- On the S7 we discovered a previously unknown overflow in the parsing of the call setup bearer Information Element (IE) (3GPP 24.008 - 5.2.2)
- This packet is incorrectly parsed by the baseband when a call is incoming
- Heap based buffer overflow
- We confirmed that this vulnerability is no longer!

Bounds checking has been added in the latest modem versions.
ShannonEE Triage

[2.17512][AFL] 0x4500013d pal_MsgSendTo(CC (23))

[2.17539][CC] 0x40d3e3c7 0b101: [cc_Main.c] - CC TASK

[2.17570][CC] 0x40d3de45 0b10: [cc_Main.c] - cc_UpdStackId :CcCurrentStackId: 0

[2.17612][CC] 0x4088dcb9 0b101: [cc_MsgDescription.c] - cc_MapSubTypeToMessageNum SubType = 5

[2.17990][CC] 0x40d3dfe5 0b100: [cc_Main.c] - [StackNo] 0

[2.18022][CC] 0x40d3e051 0b100: [cc_Main.c] - CC <= <RADIO MSG> SETUP_IND

[2.18059][CC] 0x40d3e245 0b1010: [cc_Main.c] - PRIVACY! MT message : SETUP

..snip...

[2.18775][CC] 0x40d3a7d5 0b100: [cc_PduCodec.c] - ----------- Displaying Information Elements -----------

[2.18814][CC] 0x40d39beb 0b100: [cc_PduCodec.c] - Received SETUP From Network

[2.18846][CC] 0x40d39c05 0b100: [cc_PduCodec.c] - 2 Mandatory IE -> ...[24.008]-9.3.23.2

[2.18870][CC] 0x40d39c21 0b100: [cc_PduCodec.c] - BearerCapabilityOne -> ...

[2.19013][CC] 0x408775b 0b101: [cc_MtCallEstablishment.c] - Entering cc_DecodeSetupIndMsg....[24.008] - 5.2.2...

[2.19066][CC] 0x408a7841 0b10: [cc_MtCallEstablishment.c] - TransactionId -> 11

[2.19164][CC] 0x40d407a7 0b100: [cc_Main.c] - Bearer 1 Capability -> ....

[2.19230][CC] 0x40d40997 0b100: [cc_Main.c] - Network Transfer Capability -> CC_NTWK_SPEECH_BEARER

..snip...

[2.20447][CC] 0x4103e933 0b100: [bc_utilities.c] - Setting BC_LENGTH_OF_BEARER (0) = 0x82

[ERROR] FATAL ERROR (CC): from 0x40fc1e69 [pal_PlatformMisc.c:146 - Fatal error:PAL_MEM_GUARD_CORRUPTION

pal_MemInterface.c Line 886]
```plaintext
gef> shell xxd -e ./cc-crash-min

00000000: 30303030 30303030 30303030 30303030 30303030 30303030 30303030 30303030
00000010: 00000000 30303030 30303030 30303030 30303030 30303030 30303030 30303030
00000020: 80040533 30303030 30303030 30303030 30303030 30303030 30303030 30303030
00000030: 00000040 30303030 30303030 30303030 30303030 30303030 30303030 30303030
00000040: 00000050 30303030 30303030 30303030 30303030 30303030 30303030 30303030
00000050: 00000060 30303030 30303030 30303030 30303030 30303030 30303030 30303030
00000060: 00000070 30303030 30303030 30303030 30303030 30303030 30303030 30303030
00000070: 00000080 30303030 30303030 30303030 30303030 30303030 30303030 30303030
00000080: 00000090 30303030 30303030 30303030 30303030 30303030 30303030 30303030
```
Over-the-air Reproduction
# Experimental Setup

- **SDR**: bladeRF x40 ($420)
- **Base station**: YateBTS (modified with exploit payload)
- **Target device**: Samsung S10 / S7
- **SIM Card**: Osmocom USIM (for easy phone connection)
- Thanks to Tyler Tucker for recording the demo and managing the test setup!
Crash Triggered!

Output of `adb logcat` during S7 crash
Let’s wrap it up
Remaining Challenges & Future Work

- Cellular-tailored stateful fuzzing is a requirement to get deeper code coverage

- The physical layer is ignored - what are we missing by not supporting it?
  - DSPs are attack surface too

- A holistic analysis of baseband attack surface
  - What has been covered and what remains
# Takeaways

- We need scalable tools to keep up with the ever increasing amount of cellular protocol implementations and attack surface, especially with widespread 5G on the horizon.

- Building a baseband emulator, even with undocumented hardware configurations can be done. We can bring this methodology to other basebands.

- We MUST emulate basebands for the level of introspection required to debug memory corruptions. Over-the-air testing alone is not sufficient.
# Releases

**https://github.com/grant-h/ShannonBaseband**

- **GHIDRA Tools**
  - Shannon firmware loader
  - Debug annotation script
  - Auto-renamer

- **Shannon reversing details**
  - Export of 2017 Shannon image (reversed)
  - On-device log parser (.BTL) and file format info

- **Firmware samples**

**https://github.com/grant-h/ShannonEE**

- **ShannonEE** (will be released after disclosures + QA)
Acknowledgements

NSF/SRC CNS–1815883

NWO 628.001.030 ("TROPICS")

AFOSR FA8560–18–S–1201
AFRL/AFOSR–2018–003

ONR OTA–N00014–20–1–2205
Thanks!

@Digital_Cold

@nSinusR
Backup Slides
CPU Support
# Shannon CPU Support

- ARM Cortex-R7 (identified from strings and previous work)
- QEMU doesn’t support this ARM variant
  - This seems like it could derail things before they even get started
- QEMU does support the R5
- **What would be required to get the R7 working?**
  - Time to read the processor documentation

Taken from [https://developer.arm.com/ip-products/processors/cortex-r/cortex-r7](https://developer.arm.com/ip-products/processors/cortex-r/cortex-r7)
diff --git a/target/arm/cpu.c b/target/arm/cpu.c
index 04b062c..9f6d05e 100644
--- a/target/arm/cpu.c
+++ b/target/arm/cpu.c
@@ -1132,6 +1132,15 @@ static void cortex_r5_initfn(Object *obj)
     define_arm_cp_regs(cpu, cortexr5_cp_reginfo);
 }

+static void cortex_r7_initfn(Object *obj)
+{
+    ARMCPU *cpu = ARM_CPU(obj);
+    cortex_r5_initfn(obj);
+    // granth: not sure how many there actually are
+    cpu->pmsav7_dregion = 32;
+}
+
+static const ARMCPRegInfo cortexa8_cp_reginfo[] = {
+    { .name = "L2LOCKDOWN", .cp = 15, .crn = 9, .crm = 0, .opc1 = 1, .opc2 = 0,
+        .access = PL1_RW, .type = ARM_CP_CONST, .resetvalue = 0 },
@@ -1573,6 +1582,7 @@ static const ARMCPUInfo arm_cpus[] = {
+    { .name = "cortex-a7", .initfn = cortex_a7_initfn },
+    { .name = "cortex-r5", .initfn = cortex_r5_initfn },
+    { .name = "cortex-r7", .initfn = cortex_r7_initfn },
+    { .name = "cortex-a7", .initfn = cortex_a7_initfn },
+Not much after all :)
# ShannonLoader for GHIDRA

- Automates the extraction of TOC entries into memory regions
- Extracts MPU tables for extra RAM/MMIO regions, allowing for XREFs
- Resolves overlapping memory regions and applies MPU permissions (RO/RW, X)
- Useful for batch loading of modem firmware (e.g. when BinDiffing)

Name: ShannonLoader  
Description: A loader for Samsung's Shannon modem binaries.  
Author: Grant Hernandez  
Created-on: 3/1/2020  
Version: 9.1.2
Cross-version Support
Automated memory map extraction

- MPU enables coarse-grained permission control (rwx) over memory regions
- ShannonEE parses MPU configurations to automatically create regions in Panda
- Uses binary patterns ([0-255] based regex) to reliably find MPU signature

```
00000000 - 00002e40 TOC_BOOT_LOW (rwx)
00002e40 - 00008000 SPLIT_2e40_51c0 (r-x)
00100000 - 00120000 MPU22_00100000 (rw-)
04000000 - 04020000 MPU1_04000000 (r-x)
04800000 - 04804000 MPU2_04800000 (rw-)
40000000 - 40002e40 TOC_BOOT (rwx)
40002e40 - 40010000 SPLIT_40002e40_d1c0 (r-x)
40010000 - 43010000 TOC_MAIN (rwx)
43010000 - 45600000 SPLIT_43010000_25f0000 (rwx)
45600000 - 45700000 NV (rw-)
45700000 - 46000000 MPU3_45700000 (rwx)
... 
47300000 - 47382000 SPLIT_47300000_82000 (rw-)
47382000 - 47382100 DSPPeripheral (<class 'hw.DSPPeripheral.DSPPeripheral'>)
... 
```
class S355AP(ShannonSOC):
    peripherals = [
        SOCPeripheral(DSPPeripheral, 0x4751b000, 0x100, name="DSPPeripheral", sync=[137, 292]),
        SOCPeripheral(PMICPeripheral, 0x96450000, 0x1000, name="PMIC"),
        SOCPeripheral(S355DSPBufferPeripheral, 0x47504000, 0x1000, name="DSPB"),
    ]

CHIP_ID = 0x03550000
SIPC_BASE = 0x95b40000
SHM_BASE = 0x48000000
SOC_BASE = 0x83000000
SOC_CLK_BASE = 0x83002000
CLK_PERIPHERAL = S355APClkPeripheral
TIMER_BASE = SOC_BASE + 0xc000

name = "S355AP"
Backtrace Logging
# Modem Backtrace Logging (BTL)

- Dumped on crash, but can be manually extracted (.BTL files)
- Reverse engineered BTL task
- LZ4 encoded log ring buffer in baseband memory
  - Log entries are pointers to format strings in `modem_debug.bin`
  - Varargs are pointers or literals, depending on format specifier
- Not 100% reverse engineered, but good enough to compare ShannonEE with real modem logging