Breaking Band

reverse engineering and exploiting the shannon baseband

Nico Golde <nico@comsecuris.com> @iamnion
Daniel Komaromy <daniel@comsecuris.com> @kutyacica
Motivation

- little concrete/reproducible work on analyzing and exploiting cellular basebands
- lots of protocol research: Benoit Michau, Ravi Borgaonkar, SRLabs, Osmocom,..
- everyone keeps talking about this / lots of FUD (hi OSnews!)
- highest payout at mobile pwn2own historically (100-150k$)
Motivation cont.

- Most research focused on Qualcomm basebands (AMSS)
- But we worked for Qualcomm :)
- QC lost significant market share with release of Samsung Galaxy S6/Edge
- S6* became pwn2own target
- Shannon: how hard can it be?

This is our story from 0 to 0-day
Talk Structure

• Steps to reverse engineer the RTOS, find vulns, and write a full RCE exploit

• We try to reconstruct our path, including both successes and fails

• We release all our custom-built RE tools \o/
Shannon Background

- Samsung's own(?) cellular processor (CP)/modem/baseband implementation
- entire mobile phone stack (2-4G, SIM, IPC with application processor OS, ...)
- **not new** at all
  - Galaxy S5 mini, Galaxy Note 4, various Samsung USB LTE sticks (e.g. GT-B3740)
- **non-Samsung** devices
  - e.g. some Meizu smartphone models
- ... and **still used by Samsung**!
  - most non-US Galaxy S7 devices
Taking a Peek at Firmware

- modem.bin can be obtained from firmware images or Android RADIO device partition

- No luck on the naive approach:

```sh
$ file modem.bin
modem.bin: TOC sound file
$ binwalk modem.bin
```

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>HEXADECIMAL</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>10024</td>
<td>0x2728</td>
<td>CRC32 polynomial table, little endian</td>
</tr>
<tr>
<td>27103337</td>
<td>0x19D9069</td>
<td>MySQL ISAM compressed data file Version 11</td>
</tr>
</tbody>
</table>
Identifying Code

- **BOOT**: baseband bootstrap code
- **MAIN**: baseband code
- **NV**: non-volatile memory - likely baseband settings etc
- **OFFSET**: unknown
- proprietary/undocumented header format
- contains some kind of hash / secure boot
Identifying BOOT Code

- E* often tied to ARM condition codes -> actual code?
Identifying BOOT Code

• looks like sane ARM code!
Identifying MAIN Code

• ~38 MB binary
• no such luck as before, no idea what this is
• Galaxy S6 image the first to feature this
Identifying MAIN Code

- constant high/flat entropy, likely encryption
- no silly xor encryption as far as we can tell

also see [http://www.devttys0.com/2013/06/differentiate-encryption-from-compression-using-math/](http://www.devttys0.com/2013/06/differentiate-encryption-from-compression-using-math/)
MAIN Code: Remaining Options

- BOOT: tight copy/replace loops with hardware-assisted memory mapped-io -> hard

- TEE/TrustZone: Trustlets potentially involved in decryption -> dead end

- Android kernel/user space involvement (/sbin/cbd): CP Boot Daemon / Cellular Baseband Daemon -> dead end
CP Boot Daemon (cbd)

- started at boot:
  - parses modem image TOC
  - sends modem via SPI* for loading
  - kernel driver assistance (see drivers/misc/modem_v1/modem_io_device.c)
  - no relevant unpacking/decrypting of image though

*yo HexRays, we would have appreciated that ARM64 decompiler plugin 6 months earlier ;(
Generating live RAMDUMPs

• cbd/kernel code have code for ramdumps via:

/dev/umts_ramdump0
IOCTL_MODEM_RAMDUMP_START

• can be triggered directly from cbd as root via -o u (test/ramdump)
UI-based RAMDUMP

- non-root (as we found later)

*#9090*

*#9900*
Interpreting RAMDUMP

- 130mb dump: containing code, but not continuous in memory -> analysis in IDA will be broken

- cbd<->boot knowledge brought us to ramdump handler in boot

```c
8    dword_2948[0].start_ea = (void *)0x40000000;
9    dword_2948[0].size = 0x8000000;
10   dword_2948[1].start_ea = (void *)0x4000000;
11   dword_2948[1].size = 0x10000;
12   dword_2948[2].start_ea = (void *)0x4800000;
13   dword_2948[2].size = 0x4000;
14   dword_2948[3].start_ea = (void *)0xE000000;
15   dword_2948[3].size = 0x57000;
16   dword_2948[4].start_ea = (void *)0x2F00;
17   dword_2948[4].size = 0x100;
```

can nicely translate into an IDA loader!
Reverse Engineering Shannon

- 130MB ramdump (~38 code)
- ~70k functions
- stripped, but fairly verbose on strings
- ARM Cortex R7

**Goal:**
- identify RTOS primitives
- identify cellular stack layers (Layer2/3 GSM, UMTS, LTE)
- find way to debug
- find exploitable OTA issues
Sugar-coating MAIN Code

• We got the MAIN code, but:
  • significant amount of unidentified code
  • tons of strings to make use of
  • RTOS identification cumbersome with stock IDA functionality
  • debug capability needed for actual verification
Assisting Function Detection

• IDA's 2 pass analysis is decent, but still misses lots of functions, confuses code/data segments

• Simple script to find function prologues improves upon IDA's results by thousands of functions

• False positives definitely exist, but hurt very little
Making Use of Strings

- ~100k usable strings (common in basebands due to debug tools, e.g. Samsung DM)
- state strings
- file paths (hierarchical info)
- function names

- any automatic labeling is better than sub_*!
Strings->Function Label

"exact" strings
identify handlers with debug info

fatal_error
assert_fatal
free
default_trace

function names
file names
path info (module)

"fuzzy/misc" strings
sanitize remaining strings

> 5 chars
alphanumeric
c consonants
vowels
Applying Labels

• For each function:
  • calls known API? -> trace back arguments -> label
  • part of known directory structure? -> sanitize path -> partial label
  • contains file name -> sanitize file -> sub module / partial label
  • uses only fuzzy string? -> label

• reuse names for labeling callers of these functions -> "calls_..."

• rinse and repeat every now and then

IDApython yields ~20k named functions
RTOS Primitive Identification

- In ARM, a lot of RTOS primitives are implemented via system control co-processor instructions (MCR/MRC)
- IDA doesn’t parse these
- Scripted MCR annotation: ARM R7, ARM9, and ARM11
RTOS Baseline

• What privilege level are we running at?

• How to find/enumerate the tasks of the OS?

• How are tasks handled in this OS? Start-up, communication, separation?

• Memory management of tasks (heaps&stacks, MMU/MPU)?

• How to identify most interesting tasks (3GPP Layer3 components doing message (IE) parsing)?
Execution Mode

- Expected: typical OS with kernel+user space: many SVC calls in user-space code, complex SVC handlers and RETs in kernel code.

- Few SVC handlers implemented, mostly ramdumping and resets

- System registers indicate supervisor

- Preliminary conclusion*: all supervisor, all the time :)

* ultimately verified by issuing privileged instructions once we had RCE
Task Identification

- tasks in ramdump make use of their stack frames
- find stacks in ramdump by stackframe analysis
- heuristic of a stack: dword == instr+1, instr follows a BL
- backtrace frames → common task init function → initialization routine fills in task struct, kept on linked lists
- taskscan.py walks linked list structure: #101 tasks
Task Message Queuing

```c
while (1) {
    v12 = getIncoming_msg_from_queue_struct(23, &ptr, &res, 1); // msg queue API, shared across all tasks
    v44 = &unk_41CC7250;
    v45 = 262213;
    dbg_trace_args_somethings((unsigned __int64 *)&v44, -20071784);
    ++num_cc_in_messages;
    v44 = &unk_41CC7544;
    v45 = 262212;
    dbg_trace_args_somethings((unsigned __int64 *)&v44);
    if (v12)
        break;
    if ((unsigned __int8)res != 2) {
        if ((unsigned __int8)res == 3) // res == 3 seems to mean that it is a timer expiry event, res == 3 that it is a new
            // calls_NEDGE_NASL3_CC_cc_ExcManagement_somethings((unsigned int.ptr); // sg related to timer expiry
        } else {
            v44 = &unk_41CC726C;
            v45 = 262208;
            dbg_trace_args_somethings((unsigned __int64 *)&v44);
        }
    curr_cc_msg = 0;
    goto LABEL_14;
}

v13 = ptr;
curr_cc_msg = (int)ptr;
while (1) // now process each stored message, if any
    {
        if (v13)
            {
                // process incoming message
                j_free(&curr_cc_msg, "././././NEDGE/NASL3/CC/Code/Src/cc_Main.c", (void *)__0x1E2);
            }
    }

LABEL_14: v14 = 0;
while (1) {
    v15 = (char **)stored_cc_msgs[2 * v14]; // some messages get stored away for later processing
    v16 = *v15;
    if (*v15)
        break;
    v14 = (unsigned __int8)(v14 + 1);
    if (v14 >= 7)
        goto LABEL_18;
    }
    *v15 = 0;
    v44 = &unk_41CC7560;
    v45 = 262212;
    dbg_trace_args_somethings((unsigned __int64 *)&v44, -20071784, stored_cc_msgs);

LABEL_18: curr_cc_msg = (int)v16;
if (v16)
    break; // rest of the outer while loop is timer and state mgmt
v13 = v16;
```
RTOS Memory Management

- Task stacks:
  - found easily from task structs
  - static locations, always packed one after the other. Each stackframe’s top includes two DEADBEEF markers.

- Heaps:
  - y = malloc(x); memcpy(y, z, x) is a very frequent pattern. relatively easy to spot. free, realloc found from there
  - custom implementation. tl;dr: slot-based allocator for various sizes, with look-aside doubly-linked free lists
Memory Configuration/*PU?

- The ARM R7 has an MPU only (no MMU).
- MPU configured via MCR instructions; reuse scripting
- This yields a static struct in memory -> get segment permission values. Wrote another script to automate all that.
- Result: we know the permissions and type of every segment precisely now.

```plaintext
configure_MPU
MCR p15, 0, R3,c6,c2, 0 ; Write MPU Region Number Register
MCR p15, 0, R0,c6,c1, 0 ; Write MPU Region Base Address Register
MCR p15, 0, R1,c6,c1, 2 ; Write MPU Region Size and Enable Register
MCR p15, 0, R2,c6,c1, 4 ; Write MPU Region Access Control Register
BX LR
```

main code regions start@0x04000000 and 0x40000000
Memory Management

```c
signed int initialize_MPU_config()
{
    int v0; // r081
    signed int v1; // r481
    int i; // r481
    DWORD *v3; // r104
    unsigned int v4; // r085
    _DWORDL v5; // mf085
    unsigned int v6; // v085
    int7; // r088
    disable_instruction_cache();
    sub_4031CA90();
    sub_4031CCAC(v0);
    dword_2F0C = 19505;
    v1 = 0;
    do
    {
        configure_MPU_wrapper(v1++, 0x80000000, 0x3A, 0x10, 0x300, 0x1000, 0, 0, 0, 0);
            // address 0x80000000
            // size 0x3A
            // permissions: 0x10, 0x300, 0x1000
            // enable bit: 0
    } while (v1 < 14);
    for (i = 0; i < 14; i++)
    {
        v4 = MPU_region_configs[10 * i];
        v6 = __OFSUB__(v4, 255);
        v5 = ((v4 - 255) & 0x80000000) >> 32;
        if (v5 > 255)
        {
            v6 = __OFSUB__(i, 14);
            v5 = i - 14 < 0;
        }
        if (!(unsigned int)v5 ^ v6)
            break;
        v3 = &MPU_region_configs[10 * i];
        configure_MPU_wrapper(v4,
                v3[1],
                v3[2],
                v3[3],
                *(DWORD *)v3 + 2,
                *(DWORD *)v3 + 2) >> 32,
                v3[6],
                v3[7],
                v3[8],
                v3[9]);
    } enable_instruction_cache();
    v7 = invalidate_data_cache();
    return sub_40335850();
}
```
Debugging Crashes

- screen shows crash information, including crash type. mildly useful.
- found register map structure in memory
- following the interrupt vector/exception table we got really lucky here
- exception handling fills global register map
Debugging Crashes

• screen shows crash information, including crash type. mildly useful.

• found register map structure in memory

• following the interrupt vector/exception table we got really lucky here

• exception handling fills global register map

almost proper crash debugging
Live Debugging

• SVE-2016-5301* mentioned ability to unlock device via AT command

• AT command situation far worse than what authors released! (try AT+CLAC)

• modem read/write memory via AT commands among other things

• could also build a full debugger now…. but we skipped that

*Roberto Paleari and Aristide
Vulnerability Hunting

- implementation errors, **exploitable memory corruptions**
- "higher-level" involving parsing of messages we can send from a fake BTS/network
- NAS most fruitful, RRC short signaling messages
Vulnerability Hunting / NAS (non-GPRS)

- NAS responsibilities:
  - Mobility Management (MM)
  - Radio Resource Management (RR)
  - Connection Management
- CM parses/processes/establishes
  - calls (CC)
  - short messages (SMS)
  - USSD (SS)
- messages chain Information Elements (IEs)
- **IE represents** LV/TLV (0-255) and LV-E/TLV-E (0-65535)

*also see 3GPP TS 24.007/24.008*
Vulnerability Hunting / NAS

(non-GPRS)

• two approaches:
  • try to associate spec understanding with collected strings / IE parsing
  • identify message processing in L3 stack

• Example L3/Call Control (CC) task loop:
  • dequeue message
  • CC_process_msg() -> parse IEs -> trigger callback (-> generate OTA response)
  • free message

*also see 3GPP TS 24.007/24.008
CC_process_msg()

• CC_process_msg() operates on raw OTA Layer 3 message

• calls central parse_IEs():
  
  • parses IEs based on global IE definition arrays (type, IEI, min_size, size)
  
  • encapsulates messages into IE representation array <V_ptr; Ll; is_present>

• dispatches handler from global array based on message id (useful for exploitation as well!)

  • handlers work on IE representation array content
CC_process_msg()

- 3GPP spec -> actual handler is trivial
- message ids are not 3GPP ids, but
- everything that contains "<RADIO MSG>" is one essentially

```assembly
41301A4C   aCcRadioMsgAlert_ind DCB "CC <= <RADIO MSG> ALERT_IND",0
            ; DATA XREF: .data:CC_in_msgs↑o
41301A69   ALIGN 4
41301A6C   aCcRadioMsgModify_ind DCB "CC <= <RADIO MSG> MODIFY_IND",0
            ; DATA XREF: .data:CC_in_msgs↑o
41301A6C   ALIGN 4
41301A8A   aCcRadioMsgNotify_ind DCB "CC <= <RADIO MSG> NOTIFY_IND",0
            ; DATA XREF: .data:CC_in_msgs↑o
41301A8C   ALIGN 4
41301AAA   aCcRadioMsgFacility_ind DCB "CC <= <RADIO MSG> FACILITY_IND",0
            ; DATA XREF: .data:CC_in_msgs↑o
41301A9C   aCcVcg_calleestablish_cnf DCB "CC <= VCG_CALLESTABLISH_CNF",0
            ; DATA XREF: .data:CC_in_msgs↑o
41301ACC   ALIGN 4
41301AE9   aCcVcg_altercodec_cnf DCB "CC <= VCG_ALTERCODEC_CNF",0
            ; DATA XREF: .data:stru_41042CB4↑o
41301B06   ALIGN 4
41301B08   aCcCc_alert_ind DCB "CC ==> CC_ALERT_IND",0 ; DATA XREF: .data:CC_out_msgs↑o
41301B1C   aCcCc_aoc_ind DCB "CC ==> CC_AOC_IND",0 ; DATA XREF: .data:stru_41042CD0↑o
41301B2E   ALIGN 0x10
```
Finding Exploitable Bugs

- At this point we know:
  - all OTA handlers
  - structure of incoming payloads; tainted values (payload,len with the constraints)

- Further vulnerability hunting options:
  - manual handler analysis and IDA scripting, looking for tainted length in memcpy etc.
  - bjoern, decompiler+joern, ...

- Can't estimate how "buggy" this code is: we found a winner quickly, weren't forced to do more vuln hunting
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
Example CVE-2015-8546

**SVE-2015-5123: Samsung Galaxy Edge baseband process vulnerability**

Severity: Critical
Affected versions: Selected models including Galaxy S6/S6 Edge, Galaxy S6 Edge+, and Galaxy Note5 with Shannon333 chipset
Reported on: November 12, 2015
 Disclosure status: This issue is publicly known. (CVE-2015-8546)
A vulnerability generating a stack overflow enables an attacker to run remote codes on the vulnerable devices by pushing a malicious code from a fake base station.
The supplied patch prevents a stack overflow problem.
Example CVE-2015-8546

**CVE-2015-5123: Samsung Galaxy Edge baseband process vulnerability**

Severity: Critical
Affected versions: Selected models including Galaxy S6/S6 Edge, Galaxy S6 Edge+, and Galaxy Note5 with Shannon333 chipset
Reported on: November 12, 2015
Disclosure status: This issue is publicly known (CVE-2015-8546)
A vulnerability generating a stack overflow enables an attacker to run remote codes on the vulnerable devices by pushing a malicious code from a fake base station.
The supplied patch prevents a stack overflow problem.
Example CVE-2015-8546

Description:
As described in 3GPP TS 24.008, the serving cellular network can send a "PROGRESS" message (see 9.3.17) to the UE. The standard makes it mandatory to include a "Progress Indicator" Information Element (IE) within this message. This IE is a length/value element, which is specified in 10.5.4.21. From the specification: "The purpose of the progress indicator information element is to describe an event which has occurred during the life of a call."

When the cellular baseband (CP) is parsing this message, it is not properly guarding against a stack-based buffer overflow when copying Progress Indicator elements to a local stack buffer. This can result in memory corruption and as a result, yield to arbitrary code execution by an adjacent attacker who runs the serving network.
Example CVE-2015-8546

CC_decodeProgressInd

```c
sub_404EAEF4((char *) (unsigned __int8) in
if (is_progress_ind_set() == 1)
{
  copy_progress_ind((int) &v24);

  v15 = return_progress_ind_len();
  v12 = v15;
  v16 = v25 & 0x7F;

  int __fastcall copy_progress_ind(int a1)
  {
    return memcpy_8(a1, Progress_Ind_IE_repr.V_ptr, (unsigned __int16) Progress_Ind_IE_repr.LI);
  }
```

literally a text book stack-based buffer overflow over-the-air!
Exploitation / Setup

- OpenBSC provides FOSS network stack (GSM)
  - stuff messages into gsm48_conn_sendmsg()
- many options for Base Transceiver Station (BTS) side:
  - nanoBTS,
  - sysmoBTS
  - SDR (USRP,..)
  - ...
  - <500 $
Exploit Mitigations

- **Existing mitigations/stability improvements**
  - stack overflows are checked (verifies the deadbeef markers during task scheduling switches)
  - heap guard words exist
  - R7 supports XN and is configured for certain regions by the MPU

- **Lack of baseline mitigations**
  - stack/heap guards static, no heap hardening (safe unlinking, ...)
  - no stack canaries
  - no randomization / static unprotected function pointers

- **Broken mitigations**:
  - the XN region configuration is broken/incomplete: e.g. stack/heap not one of them
Exploit Primitives

- **Content at static or less fluctuant address** (some):
  - short-term subscriber identity/TMSI -> known dword
  - network name (long/short) -> alphanumeric ARM shellcode (also uncached!)

- **Payload size restrictions:** bypass via staged CC/L3 handler hooking

- **Clean state returns:** L3 state machines are simple loops -> jump to the beginning automatically processes next message (assuming registers are setup correctly)

- **Persistence:** clean return survives flight mode toggle; potential path for real persistence may exist (e.g. exploiting nv item parsing issues etc.)
Exploit Payloads

- baseband code execution has limited functionality
  - **not** the master over application processor/memory (these days), but loaded by apps processor! (pls get this right in public debates)
- baseband sees all* data/signaling exchanged with cellular networks though (calls, text messages, data)
- typical payloads would alter/eavesdrop/inject/drop these
- for our demo we have chosen to reroute calls (e.g. for MitM): simple payload that changes signaling data (<100 bytes); implanted via patching callback code

* that's why you should use E2E crypto!
Exploitation Fails

• **Caching**
  • making RX code RWX via MPU config works …but actual patching works unreliably; somehow cache flushing MCRs don't work as expected (maybe LLI related?)
  • eventually went for patching data, not code

• **Dual-Sim code snafus**
  • almost the entire L3 code is duplicated in the firmware, with “DS_” labels added to names
  • we suspect this is a primitive dual-sim support implementation.
  • tl;dr: verify bindiff results with care when upgrading firmware versions!
Application Processor Escalation

- **modifying application processor data traffic:**
  - inject JS into HTML or relay traffic to attacker controlled site -> browser pwn or exploit an unsecured update process (e.g. SwiftKey Keyboard, …)

- **IPC channels:**
  - shared memory IPC implementation (parsing, range checking, ..)
  - DMA capable peripherals (data moving)
  - services built on top of this (e.g. RILD*)

- **IPC/LLI message debugging** on Android via `/d/svnet/mem_dump`
  - full baseband<->apps IPC traces, including your seen networks, called numbers, etc
  - yes, this is available to unprivileged applications on Galaxy devices!

* the old remoteFS directory traversal bug discussed by Replicant seems fixed ;)*
Final Remarks

• 2 people / part time effort; 3-6 months

• basebands are also "just" embedded systems, no mad ninja skills required

• still a lot of space for research, especially on exploitation:
  • target identification (device/firmware)
  • application processor escalation
Tools Release

- [github.com/comsecuris/shannon](https://github.com/comsecuris/shannon) (release imminent :)
- 010 Editor templates
- IDA loaders
- RAMDUMP scripts
- idapython: scanning tasks, naming functions, MPU configuration, register dumps, read/write memory, unpack modem binaries, naming of message handlers etc.
Questions

contact@comsecuris.com
Backup - Relaying of Calls / Impact

• Essentially enables interception/MitM of calls

• Attacker would just need to know original number to initiate new call and proxy

• Options:
  • append original number to caller and extract on attacker side
  • 3GPP provides "called party subaddress" field to denote extensions
  • no visible behavior difference from user side (network can see this though)