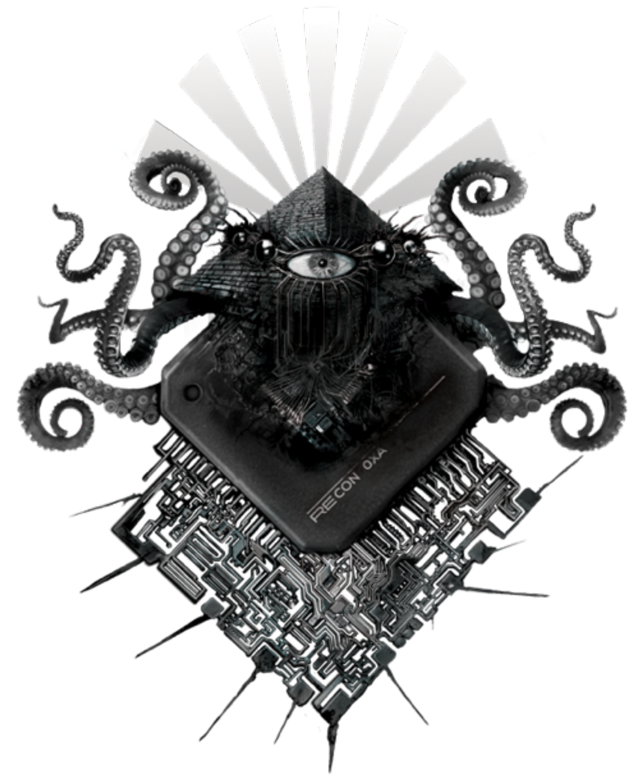
Comsecuris



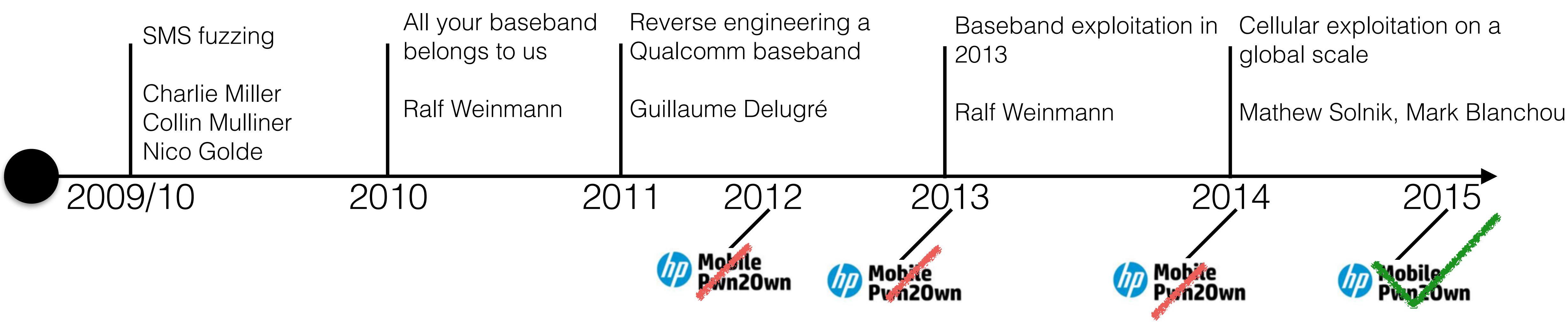
Breaking Band

reverse engineering and exploiting the shannon baseband

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

Comsecuris

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$)

Comsecuris

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

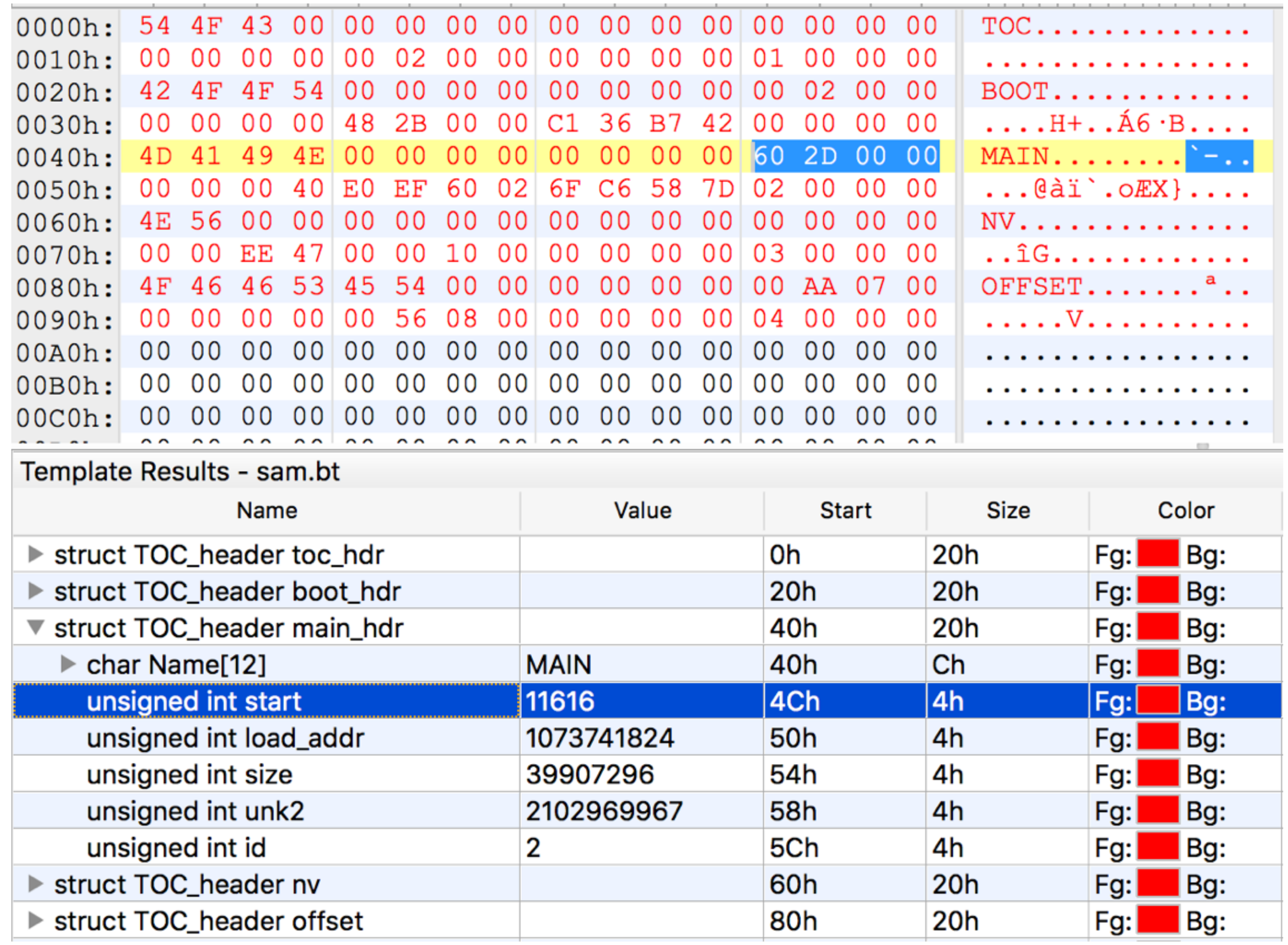
- modem.bin can be obtained from firmware images or Android RADIO device partition

- No luck on the naive approach:

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

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
10024         0x2728          CRC32 polynomial table, little endian
27103337      0x19D9069       MySQL ISAM compressed data file Version 11
```

Identifying Code

- BOOT: baseband bootstrap code
- MAIN: baseband code
- NV: non-volatile memory - likely baseband settings etc
- OFFSET: unknown



```
0000h: 54 4F 43 00 00 00 00 00 00 00 00 00 00 00 00 00  TOC.............
0010h: 00 00 00 00 00 02 00 00 00 00 00 00 01 00 00 00  ................
0020h: 42 4F 4F 54 00 00 00 00 00 00 00 00 00 02 00 00  BOOT............
0030h: 00 00 00 00 48 2B 00 00 C1 36 B7 42 00 00 00 00  ....H+..Á6·B....
0040h: 4D 41 49 4E 00 00 00 00 00 00 00 00 60 2D 00 00  MAIN........`-..
0050h: 00 00 00 40 E0 EF 60 02 6F C6 58 7D 02 00 00 00  ...@àï`.oÆX}....
0060h: 4E 56 00 00 00 00 00 00 00 00 00 00 00 00 00 00  NV..............
0070h: 00 00 EE 47 00 00 10 00 00 00 00 00 03 00 00 00  ..îG............
0080h: 4F 46 46 53 45 54 00 00 00 00 00 00 00 AA 07 00  OFFSET.......ª..
0090h: 00 00 00 00 00 56 08 00 00 00 00 00 04 00 00 00  .....V..........
00A0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00B0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00C0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Template Results - sam.bt

Name	Value	Start	Size	Color
▶ struct TOC_header toc_hdr		0h	20h	Fg: Bg:
▶ struct TOC_header boot_hdr		20h	20h	Fg: Bg:
▼ struct TOC_header main_hdr		40h	20h	Fg: Bg:
▶ char Name[12]	MAIN	40h	Ch	Fg: Bg:
unsigned int start	11616	4Ch	4h	Fg: Bg:
unsigned int load_addr	1073741824	50h	4h	Fg: Bg:
unsigned int size	39907296	54h	4h	Fg: Bg:
unsigned int unk2	2102969967	58h	4h	Fg: Bg:
unsigned int id	2	5Ch	4h	Fg: Bg:
▶ struct TOC_header nv		60h	20h	Fg: Bg:
▶ struct TOC_header offset		80h	20h	Fg: Bg:

- proprietary/undocumented header format
- contains some kind of hash / secure boot

Comsecuris

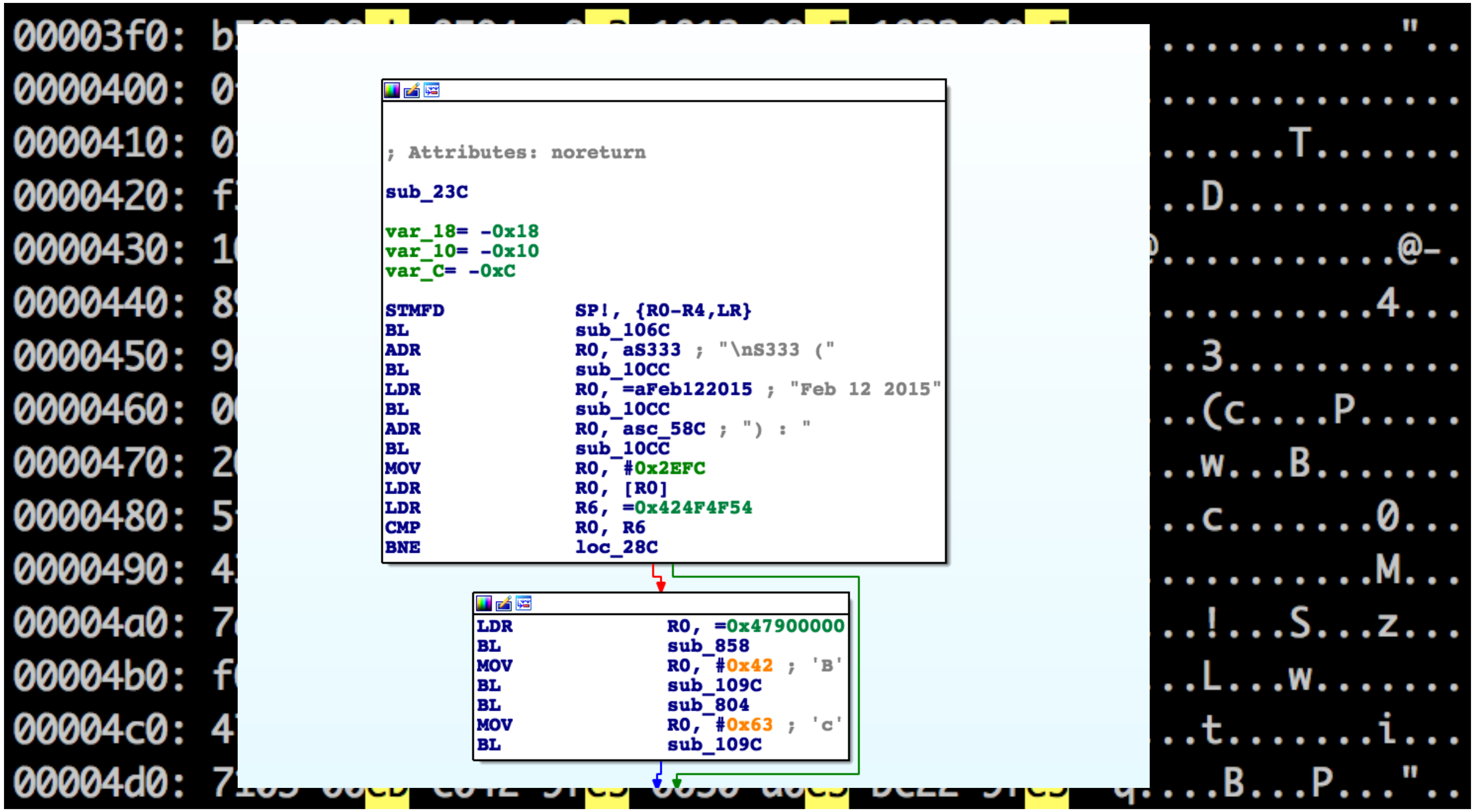
Identifying BOOT Code

```
00003f0: b503 00eb 8504 a0e3 1812 90e5 1822 90e5  ............."..
0000400: 0f10 81e3 0210 81e1 1812 80e5 1215 a0e3  ................
0000410: 0103 a0e3 8c08 00eb 5400 a0e3 9e03 00eb  ........T.......
0000420: f301 00eb 4400 a0e3 9b03 00eb 0c04 00eb  ....D...........
0000430: 1040 bde8 0d0d 8fe2 a303 00ea 1f40 2de9  .@...........@-.
0000440: 8903 00eb cd0f 8fe2 9f03 00eb 3403 9fe5  ............4...
0000450: 9d03 00eb 330e 8fe2 9b03 00eb fc0e 02e3  ....3...........
0000460: 0000 90e5 2863 9fe5 0600 50e1 0600 001a  ....(c....P.....
0000470: 2003 9fe5 7701 00eb 4200 a0e3 8603 00eb   ...w...B.......
0000480: 5f01 00eb 6300 a0e3 8303 00eb 3008 00eb  _...c.......0...
0000490: 4300 a0e3 8003 00eb 1301 00eb 4d00 a0e3  C...........M...
00004a0: 7d03 00eb 2108 00eb 5300 a0e3 7a03 00eb  }...!...S...z...
00004b0: f602 00eb 4c00 a0e3 7703 00eb aaff ffeb  ....L...w.......
00004c0: 4700 a0e3 7403 00eb a602 00eb 6900 a0e3  G...t.......i...
00004d0: 7103 00eb c042 9fe5 0050 a0e3 bc22 9fe5  q....B...P..."..
```

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

Comsecuris

Identifying BOOT Code



- looks like sane ARM code!

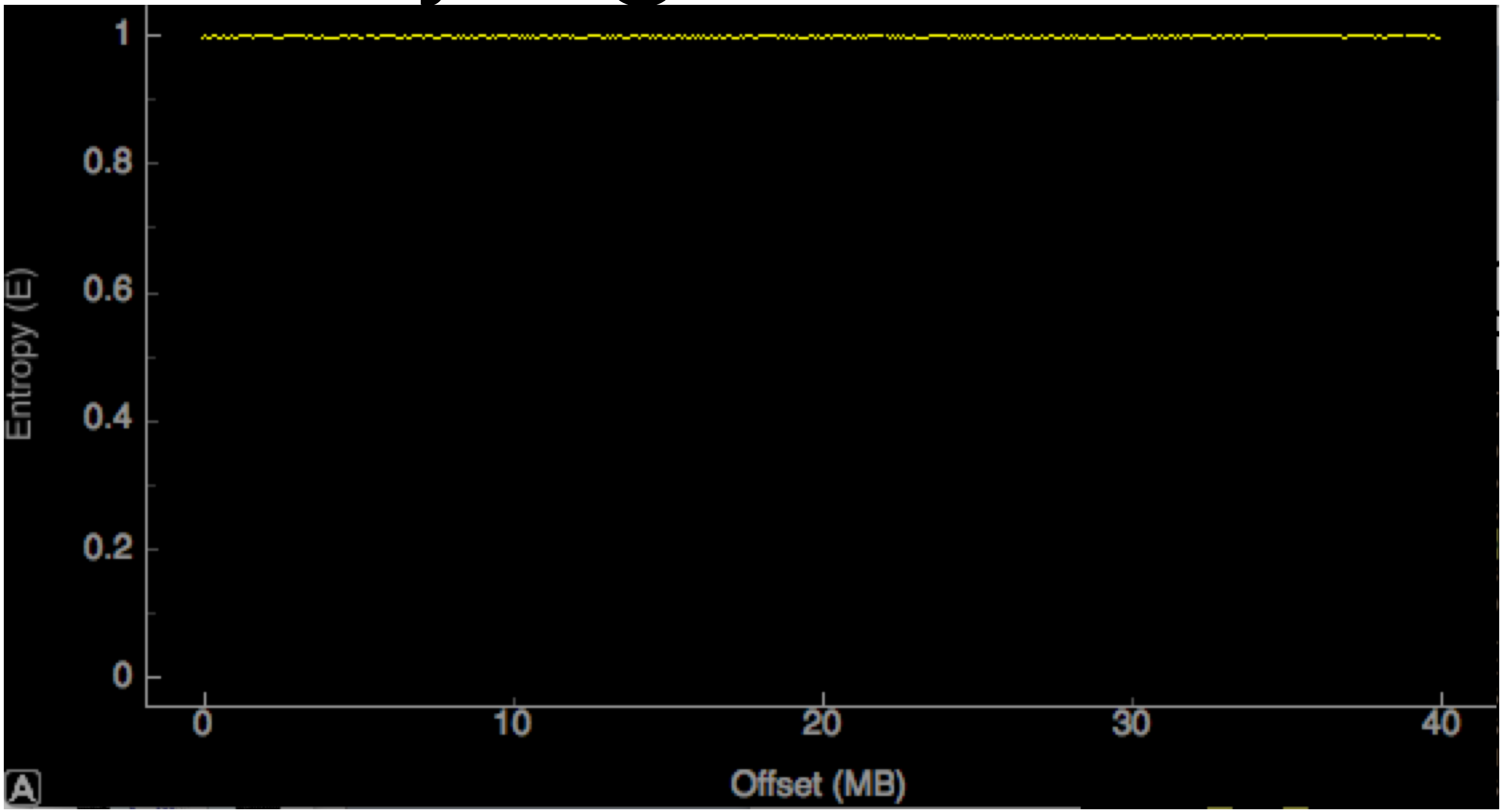
Comsecuris

Identifying MAIN Code

```
0006120: 5fa8 b6d6 dcfe 207a 7e04 7208 65bc e571  _..... z~.r.e..q
0006130: d6e4 e72c e789 c8c8 bd47 f313 4b0a e178  ...,.....G..K..x
0006140: 7d6f 6226 6c46 f8de 66a9 bc96 5d4c cf7d  }ob&lF..f...]L.}
0006150: 86e4 9ebe cd58 49bf 0f5b bb9a fcb7 4f7e  .....XI..[....O~
0006160: 9c84 8504 581c c006 ec55 b8cc 05d2 e130  ....X....U.....0
0006170: df9b 3a75 be5e 42c2 daeb daf8 f6fc 4b0d  ..:u.^B.......K.
0006180: 78da 4e7a 1d4d dd8a 63a3 33e2 12ba 81b8  x.Nz.M..c.3.....
0006190: c264 5f04 d557 08b7 fd89 7f75 ed99 ca0c  .d_..W.....u....
00061a0: 249d 29bd 5001 a4cd e951 0176 f06a 7dca  $.).P....Q.v.j}.
00061b0: 39d1 fc35 8c1e 1d86 25c2 d510 6271 8c47  9..5....%...bq.G
00061c0: 5fad ca73 83ea 802c 5ea6 b79d f45f 7fc5  _..s...,^...._..
00061d0: dc5d 8fcc c994 9e9d bf46 1cf4 cc92 3b6e  .].......F....;n
00061e0: 4b0c 4fba 0781 64c8 d44d 73e3 9a3f 0eba  K.O...d..Ms..?..
00061f0: e903 2b76 a9af b5ba ff66 983e 41f4 0601  ..+v.....f.>A...
0006200: 3ef0 6ce5 8b41 f934 7b2a 7142 dccc 77bf  >.l..A.4{*qB..w.
0006210: b5ef f3d4 fae4 eb48 0482 5585 d0f5 f50e  .......H..U.....
0006220: ba84 4d6e 6416 b84a 0719 9f5d 0597 6b8a  ..Mnd..J...]..k.
0006230: f663 4d51 ce46 3e80 f29a 3f25 13db 634f  .cMQ.F>...?%..cO
0006240: f3fa e47f ddc6 64a2 c61b 6a42 fac0 c2d6  ......d...jB....
```

- ~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/

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)

```
mif: cbd: prepare_boot_args: DEV(/dev/spi_boot_link) opened (fd 10)
mif: cbd: prepare_boot_args: BIN(/dev/block/platform/15570000.ufs/by-name/RADIO) opened (fd 11)
mif: cbd: prepare_boot_args: toc[0].name = TOC
mif: cbd: prepare_boot_args: toc[1].name = BOOT
mif: cbd: prepare_boot_args: toc[2].name = MAIN
mif: cbd: prepare_boot_args: toc[3].name = NV
mif: cbd: prepare_boot_args: NV(/efs/nv_data.bin) opened (fd 12)
```

- 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

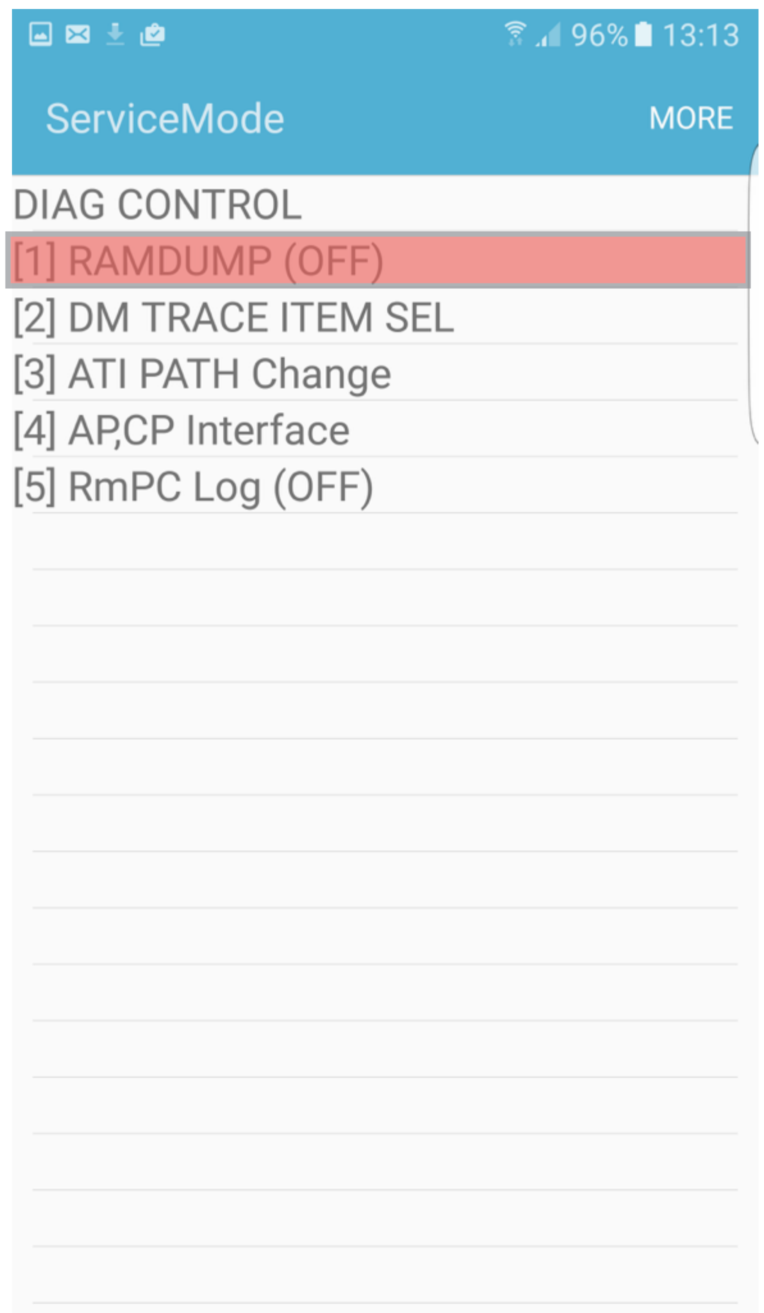
- can be triggered directly from cbd as root via -o u (test/ramdump)

```
root@zerolte:/sdcard/log # ls -l
-rwxrwx---   1 root   sdcard_r   134656256 2015-05-31 20:29 cpcrash_dump_20150531-2050.log
-rwxrwx---   1 root   sdcard_r         512 2015-05-31 20:29 cpcrash_info_ss333_20150531-2050.log
-rwxrwx---   1 root   sdcard_r         685 2015-05-31 20:29 cpcrash_log_20150531-2050.log
-rwxrwx---   1 root   sdcard_r      232312 2015-05-31 20:29 logcat_radio_20150531-2050.log
-rwxrwx---   1 root   sdcard_r     8388608 2015-05-31 20:29 mem_dump_20150531-2050.log
-rwxrwx---   1 root   sdcard_r         432 2015-05-31 20:29 mif_trace_201505312051_17.log
-rwxrwx---   1 root   sdcard_r      528719 2015-05-31 20:29 umts_crash_201505312051_17.log
```

Comsecuris

UI-based RAMDUMPs

- 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

```
dword_2948[0].start_ea = (void *)0x40000000;
dword_2948[0].size = 0x8000000;
dword_2948[1].start_ea = (void *)0x4000000;
dword_2948[1].size = 0x10000;
dword_2948[2].start_ea = (void *)0x4800000;
dword_2948[2].size = 0x4000;
dword_2948[3].start_ea = (void *)0xE0000000;
dword_2948[3].size = 0x57000;
dword_2948[4].start_ea = (void *)0x2F00;
dword_2948[4].size = 0x100;
```

can nicely translate into an IDA loader!

Comsecuris

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

```
==================================================================
          SS333/SS310 DEVELOPMENT PLATFORM
 - ARM Emulation Baseboard | Cortex-R7
 - Software Build Date : %s
 - Software Builder    : %s
 - Compiler Version    : ARM RVCT %d.%d [Build %d]
   Platform Abstraction Layer (PAL) Powered by
              Modem H/W Lab BSP SW Part
==================================================================
```

```
C../../../PSS/StackService/SMS/Code/Src/sms_PduCodec.c
../../../PSS/StackService/SMS/Code/Src/sms_Utilities.c
A../../../PSS/StackService/DS_SMS/Code/Src/ds_sms_Main.c
$A../../../PSS/StackService/DS_SMS/Code/Src/ds_sms_Main.c
C../../../PSS/StackService/DS_SMS/Code/Src/ds_sms_PduCodec.c
../../../PSS/StackService/DS_SMS/Code/Src/ds_sms_Utilities.c
../../../PSS/StackService/GMC/GmcF/code/src/Gmc_Timer.c
../../../PSS/StackService/GMC/GmcF/code/src/Gmc_TraceLog.c
$A../../../PSS/StackService/GMC/GmcF/code/src/Gmc_EventBus.c
../../../PSS/StackService/GMC/GmcF/code/src/Gmc_Main.c
../../../PSS/StackService/GMC/GmcF/code/src/Gmc_Main.c
```

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

Comsecuris

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_*!**

```
GMM_TIMEROUT_ERR
GMM_LOW_LAYER_FAILURE_ERR
GMM_DETACH_BY_THE_NETWORK_ERR
GMM_AUTH_FAIL_ERR
GMM_AUTH_REJ_ERR
GMM_INCORRECT_STATE_ERR
GMM_USER_PLMN_SEL
GMM_SIM_CONSIDERED_INVALID_ERR
GMM_ACCESS_CLASS_NOT_ALLOWED
GMM_EST_REJ_TRY_OTHER_RAT
GMM_NORMAL_RELEASE
```

```
A../../../HEDGE/NASL3/DS_MM/Code/Src/ds_mm_GmmPduCodec.c
../../../HEDGE/DS_GL3/GRR/Code/Src/ds_rr_tim.c
../../../CALPSS/LteL3/LteRrc/Code/src/LteRrc_CommUtil.c
A../../../HEDGE/DS_GL3/GRR/Code/Src/ds_rr_resel.c
A../../../HEDGE/DS_GL3/GRR/Code/Src/ds_rr_resel.c
<B../../../HEDGE/DS_GL3/GRR/Code/Src/ds_rr_plmn.c
../../../HEDGE/DS_GL2/GMAC/Code/Src/ds_mac_util.c
+;A../../../HEDGE/DS_GL2/GMAC/Code/Src/ds_mac_util.c
:;A../../../HEDGE/DS_GL3/GRR/Code/Src/ds_rr_list.c
A../../../HEDGE/GL3/GRR/Code/Src/rr_resel.c
```

Strings->Function Label

"exact" strings

identify handlers with debug info → fatal_error, assert_fatal, free, debug_trace_ → function names, file names, path info (module)

"fuzzy/misc" strings

sanitize remaining strings → > 5 chars, alphanumeric, consonants, vowels

Comsecuris

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

Comsecuris

RTOS Primitive Identification

- In ARM, 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

```
.text:40E1CB74 disable_instruction_cache              ; CODE XREF: initialize_MPU_config+6↑p
.text:40E1CB74                 MRC             p15, 0, R1,c1,c0, 0
.text:40E1CB78                 BIC             R1, R1, #0x1000
.text:40E1CB7C                 MCR             p15, 0, R1,c1,c0, 0 ; Write System Control Regi
.text:40E1CB80                 ISB             SY
.text:40E1CB84                 BX              LR
.text:40E1CB84 ; End of function disable_instruction_cache
.text:40E1CB84
.text:40E1CB88
.text:40E1CB88 ; =============== S U B R O U T I N E =======================================
.text:40E1CB88
.text:40E1CB88
.text:40E1CB88 enable_instruction_cache               ; CODE XREF: initialize_MPU_config+86↑p
.text:40E1CB88                 MOV             R0, #0
.text:40E1CB8C                 MCR             p15, 0, R0,c7,c5, 0 ; Invalidate entire instruc
.text:40E1CB90                 MRC             p15, 0, R0,c1,c0, 0
.text:40E1CB94                 ORR             R0, R0, #0x1000
.text:40E1CB98                 MCR             p15, 0, R0,c1,c0, 0 ; Write System Control Regi
.text:40E1CB9C                 ISB             SY
.text:40E1CBA0                 BX              LR
.text:40E1CBA0 ; End of function enable_instruction_cache
.text:40E1CBA0
.text:40E1CBA4
.text:40E1CBA4 ; =============== S U B R O U T I N E =======================================
.text:40E1CBA4
.text:40E1CBA4
.text:40E1CBA4 enable_instruction_and_data_cache
.text:40E1CBA4                 MRC             p15, 0, R1,c1,c0, 0
.text:40E1CBA8                 ORR             R1, R1, #0x1000
.text:40E1CBAC                 ORR             R1, R1, #4
.text:40E1CBB0                 DSB             SY
.text:40E1CBB4                 MOV             R0, #0
.text:40E1CBB8                 MCR             p15, 0, R0,c7,c10, 1 ; Clean Data Cache Line to
.text:40E1CBBC                 MOV             R0, #0
.text:40E1CBC0                 MCR             p15, 0, R0,c7,c5, 0 ; Invalidate entire instruc
.text:40E1CBC4                 MCR             p15, 0, R1,c1,c0, 0 ; Write System Control Regi
.text:40E1CBC8                 ISB             SY
.text:40E1CBCC                 BX              LR
.text:40E1CBCC ; End of function enable_instruction_and_data_cache
.text:40E1CBCC
```

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)?

Comsecuris

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

```
stack_top: 0x42f05b78 stack_base: 0x42ef5b9c
entry function is LTE_TCPI_task_entry
task name is LTE_SISO
stack_top: 0x42f15b78 stack_base: 0x42f05b9c
entry function is LTE_SISO_task_entry
task name is PacketHa2
stack_top: 0x42d69110 stack_base: 0x42d66934
entry function is
task name is MM
stack_top: 0x42d80880 stack_base: 0x42d7d8a4
entry function is MM_task_entry
task name is LLC
stack_top: 0x42e917f0 stack_base: 0x42e90814
entry function is LLC_task_entry
task name is recMailT
stack_top: 0x42f6d378 stack_base: 0x42f6cf9c
entry function is recMailT_task_entry
task name is DS_MM
stack_top: 0x42d8bc38 stack_base: 0x42d88c5c
entry function is DS_MM_task_entry
task name is DS_LLC
stack_top: 0x42e979b4 stack_base: 0x42e969d8
entry function is DS_LLC_task_entry
task name is LteRrc
stack_top: 0x42dfbff0 stack_base: 0x42dec014
entry function is
task name is REG_SAP
stack_top: 0x42eb9b78 stack_base: 0x42eb8b9c
entry function is REG_SAP_task_entry
task name is SIM_SAP
stack_top: 0x42ebc378 stack_base: 0x42ebbb9c
entry function is SIM_SAP_task_entry
task name is DS_REG_S
stack_top: 0x42ebe378 stack_base: 0x42ebd39c
entry function is DS_REG_S_task_entry
task name is Default
stack_top: 0x42d55910 stack_base: 0x42d55534
entry function is Default_task_entry
task name is CC
stack_top: 0x42d7d880 stack_base: 0x42d798a4
entry function is CC_task_entry
task name is SAEL3
stack_top: 0x42dabff0 stack_base: 0x42d9c014
```

Comsecuris

Task Message Queuing

```
while ( 1 )
{
  v12 = get_incoming_msg_from_queue_struct(23, &ptr, &res, 1);// msg queue API, shared across all tasks
  v44 = &unk_41CC7250;
  v45 = 262213;
  dbg_trace_args_something((unsigned __int64 *)&v44, -20071784);
  ++num_cc_in_messages;
  v44 = &unk_41CC7544;
  v45 = 262212;
  dbg_trace_args_something((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_HEDGE_NASL3_CC_cc_EctManagement_something__3((unsigned int)ptr);// sg related to timer expiry
    }
    else
    {
      v44 = &unk_41CC726C;
      v45 = 262208;
      dbg_trace_args_something((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 )
    {
      CC_process_msg();                       // process incoming message
      j_free(&curr_cc_msg, "../../../HEDGE/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_something((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;
```

Comsecuris

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.

```
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

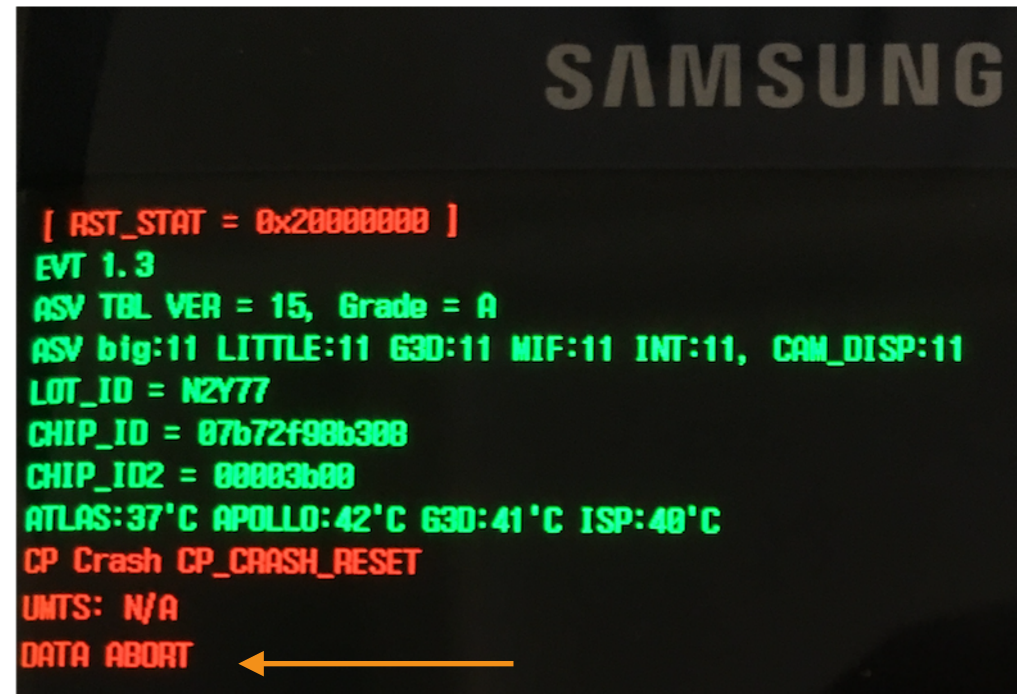
```
signed int initialize_MPU_config()
{
  int v0; // r0@1
  signed int v1; // r4@1
  int i; // r4@3
  _DWORD *v3; // r1@4
  unsigned int v4; // r0@5
  _BOOL1 v5; // nf@5
  unsigned __int8 v6; // vf@5
  int v7; // r0@8

  disable_instruction_cache();
  sub_40E1CAF0();
  sub_40E1CCAC(v0);
  dword_2F0C = 19506;
  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 )
  {
    v4 = MPU_region_configs[10 * i];
    v6 = __OFSUB__(v4, 255);
    v5 = ((v4 - 255) & 0x80000000) != 0;
    if ( v4 != 255 )
    {
      v6 = __OFSUB__(i, 14);
      v5 = i - 14 < 0;
    }
    if ( !((unsigned __int8)v5 ^ v6) )
      break;
    v3 = &MPU_region_configs[10 * i];
    configure_MPU_wrapper(
      v4,
      v3[1],
      v3[2],
      v3[3],
      *((_QWORD *)v3 + 2),
      *((_QWORD *)v3 + 2) >> 32,
      v3[6],
      v3[7],
      v3[8],
      v3[9]);
  }
  enable_instruction_cache();
  v7 = invalidate_data_cache();
  sub_40E1CCA8(v7);
  return sub_40335E50();
}
```

```
NEXT REGION
Region num: 1
DRBAR (R0): 0x04000000
DRSR (R1): 0x0000001f
DRACR (R2): 0x00000608
DRNR (R3): 0x00000001
                Region addr:            0        -0x04010000
                Region size:            6        000)
                Region enabled:         1
                Disabled subregions:    [0       0L, 0L, 0L, 0L, 0L, 0L]
                Region share-able:      0
                Region XN:              0
                Region AP:              Privileged: Read Only User: Read Only
                Region TEX,C,B:         Outer and Inner Non-cachable Normal
NEXT REGION
Region num: 2
DRBAR (R0): 0x04800000
DRSR (R1): 0x0000001b
DRACR (R2): 0x00001308
DRNR (R3): 0x00000002
                Region addr:            0x04800000-0x04804000
                Region size:            16 KB (0x00004000)
                Region enabled:         1
                Disabled subregions:    [0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L]
                Region share-able:      0
                Region XN:              1
                Region AP:              Pri        d/Write User: Read/Write
                Region TEX,C,B:         Oute        er Non-cachable Normal
```

Comsecuris

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




```
_BYTE *dump_reg_values()
{
  signed int v0; // r5@1
  _DWORD *v1; // r4@1
  _BYTE *result; // r0@4

  v0 = 0;
  print_0("iLine    : %d \n", dword_432BEFD4);
  print_0("szFile   : %s \n", dword_432BEFD8);
  print_0("szError  : %s \n", error_status_ptr);
  print_0("r0       : 0x%08X \n", dword_432BF1E8);
  print_0("r1       : 0x%08X \n", dword_432BF1EC);
  print_0("r2       : 0x%08X \n", dword_432BF1F0);
  print_0("r3       : 0x%08X \n", dword_432BF1F4);
  print_0("r4       : 0x%08X \n", dword_432BF1F8);
  print_0("r5       : 0x%08X \n", dword_432BF1FC);
  print_0("r6       : 0x%08X \n", dword_432BF1FC);
  print_0("r7       : 0x%08X \n", dword_432BF204);
  print_0("r8       : 0x%08X \n", dword_432BF208);
  print_0("r9       : 0x%08X \n", dword_432BF20C);
  print_0("r10      : 0x%08X \n", dword_432BF210);
  print_0("r11      : 0x%08X \n", dword_432BF214);
```

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

```
.data:432BF1E4 dword_432BF1E4  DCD 0x42D7D780
.data:432BF1E8 r0              DCD 0x42521D4C
.data:432BF1EC r1              DCD 0xFFFFFFFF
.data:432BF1F0 r2              DCD 0xFFFF
.data:432BF1F4 r3              DCD 3
.data:432BF1F8 r4              DCD 0x40041
.data:432BF1FC r5              DCD 0
.data:432BF1FC
.data:432BF200 r6              DCB    0
.data:432BF201                 DCB    0
.data:432BF202                 DCB    0
.data:432BF203                 DCB    0
.data:432BF204 r7              DCD 0xFECDBA98
.data:432BF208 r8              DCD 0x40045
.data:432BF20C r9              DCD 0xFE
.data:432BF210 r10             DCD 1
.data:432BF214 r11             DCD 0x42521D4C
.data:432BF218 r12             DCD 0
.data:432BF21C r15_pc          DCD 0x4049FA04
.data:432BF220 cpsr            DCD 0x20000033
.data:432BF224 r13_sp_usr      DCD 0x4803540
.data:432BF228 r14_lr_usr      DCD 0
.data:432BF22C                 DCB 0x13
.data:432BF22D                 DCB    0
.data:432BF22E                 DCB    0
.data:432BF22F                 DCB 0x80 ; �
.data:432BF230 r13_sp_svc      DCD 0x42D7D780
.data:432BF230
.data:432BF234 r14_lr_svc      DCD 0x404DFA25
.data:432BF238 dword_432BF238  DCD 0xDEADDEAD
.data:432BF23C spsr_abt        DCD 0x4803580
.data:432BF240 r13_sp_abt      DCD 0xDEADDEAD
.data:432BF244 r14_lr_abt      DCD 0x2C05CFF9
.data:432BF248 spsr_und        DCD 0x48035C0
.data:432BF24C r13_sp_und      DCD 0
.data:432BF250 spsr_irq        DCD 0x60000033
.data:432BF254 r13_sp_irq      DCD 0x48036E0
.data:432BF258 r14_lr_irq      DCD 0x400000C4
```

Live Debugging

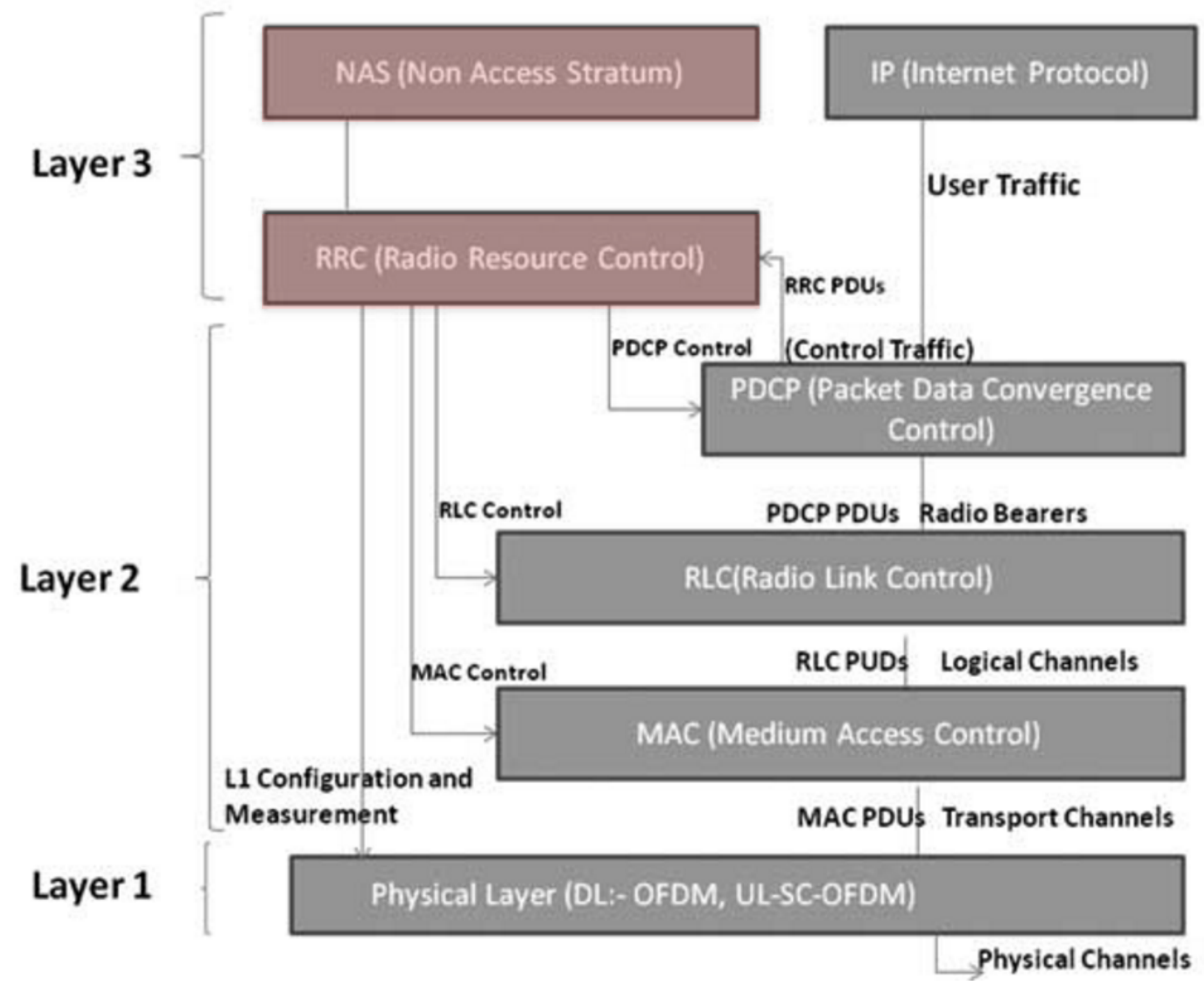
- SVE-2016-5301* mentioned ability to unlock device via AT command
- AT command situation far worse than what authors released! (try AT+CLAC)
- modem read/write memory via AT commands among other things
- could also build a full debugger now… but we skipped that

```
Terminal ready
AT+HREGREAD=41422158
0x41422158=0xffffffff

OK
AT+HREGWRITE=41422158=42
OK
AT+HREGREAD=41422158
0x41422158=0x42

OK
```

*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

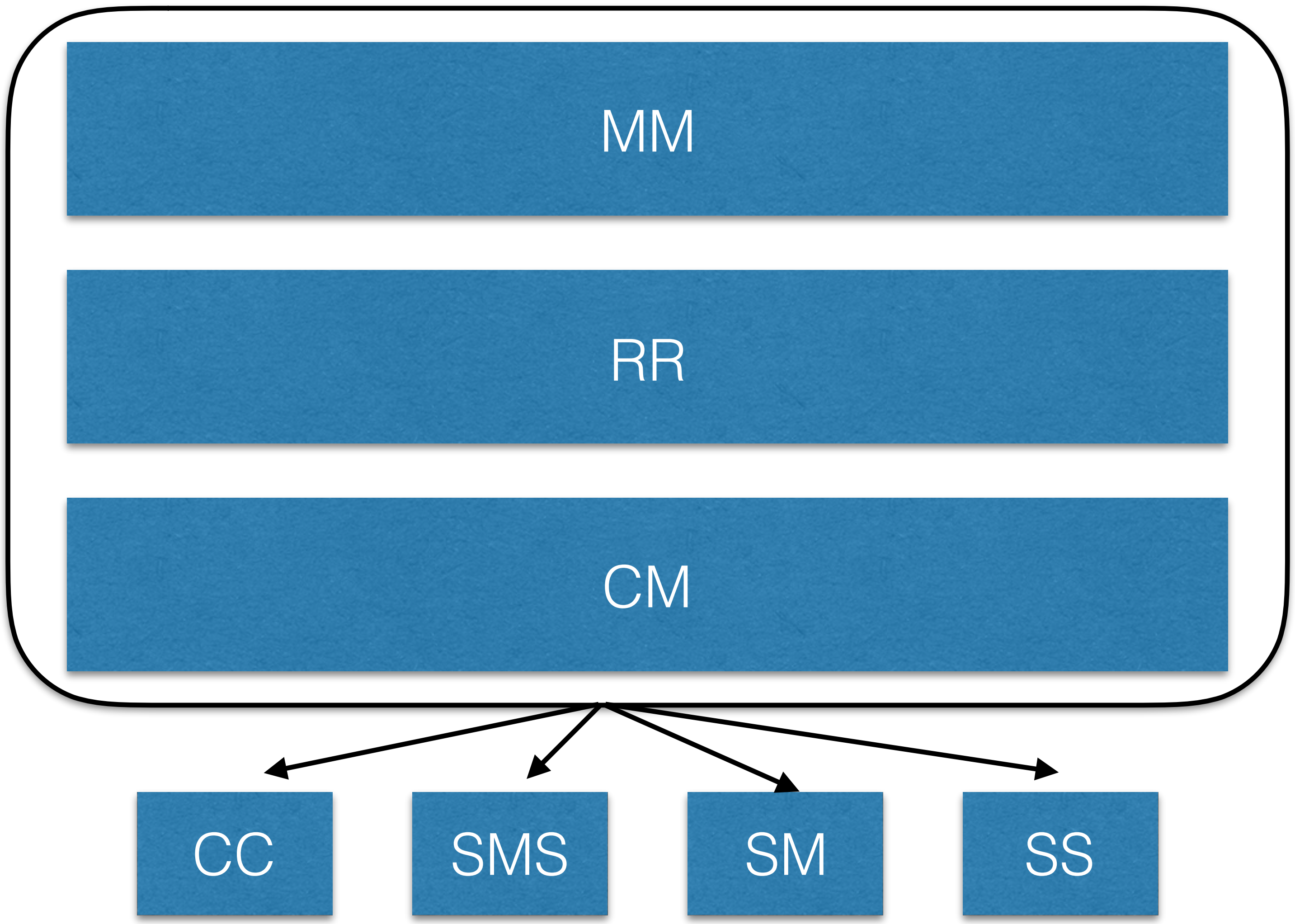


Vulnerability Hunting / NAS

(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)**

simplified LTE Layer 3



*also see 3GPP TS 24.007/24.008

Vulnerability Hunting / NAS

(non-GPRS)

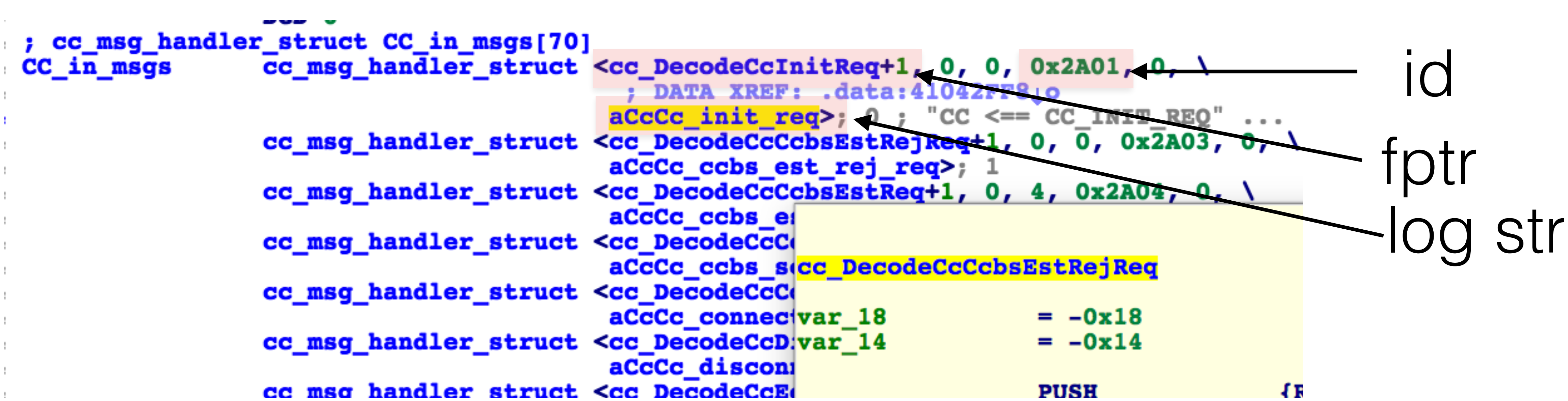
- two approaches:
  - try to associate spec understanding with collected strings / IE parsing
  - **identify message processing in L3 stack**
- Example L3/Call Control (CC) task loop:
  - dequeue message
  - CC_process_msg() -> parse IEs -> trigger callback (-> generate OTA response)
  - free message

*also see 3GPP TS 24.007/24.008

Comsecuris

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; LI; is_present>
- dispatches handler from global array based on message id (**useful for exploitation as well!**)
  - handlers work on IE representation array content



Comsecuris

CC_process_msg()

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

```
41301A4C aCcRadioMsgAlert_ind DCB "CC <== <RADIO MSG> ALERT_IND",0
41301A4C                                        ; DATA XREF: .data:CC_in_msgs↑o
41301A69                ALIGN 4
41301A6C aCcRadioMsgModify_ind DCB "CC <== <RADIO MSG> MODIFY_IND",0
41301A6C                                        ; DATA XREF: .data:CC_in_msgs↑o
41301A8A                ALIGN 4
41301A8C aCcRadioMsgNotify_ind DCB "CC <== <RADIO MSG> NOTIFY_IND",0
41301A8C                                        ; DATA XREF: .data:CC_in_msgs↑o
41301AAA                ALIGN 4
41301AAC aCcRadioMsgFacility_ind DCB "CC <== <RADIO MSG> FACILITY_IND",0
41301AAC                                        ; DATA XREF: .data:CC_in_msgs↑o
41301ACC aCcVcg_callestablish_cnf DCB "CC <== VCG_CALLESTABLISH_CNF",0
41301ACC                                        ; DATA XREF: .data:CC_in_msgs↑o
41301AE9                ALIGN 4
41301AEC aCcVcg_altercodec_cnf DCB "CC <== VCG_ALTERCODEC_CNF",0
41301AEC                                        ; 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 <u>OTA live</u> 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

Comsecuris

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

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

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.

Comsecuris

Example CVE-2015-8546

CC_decodeProgressInd

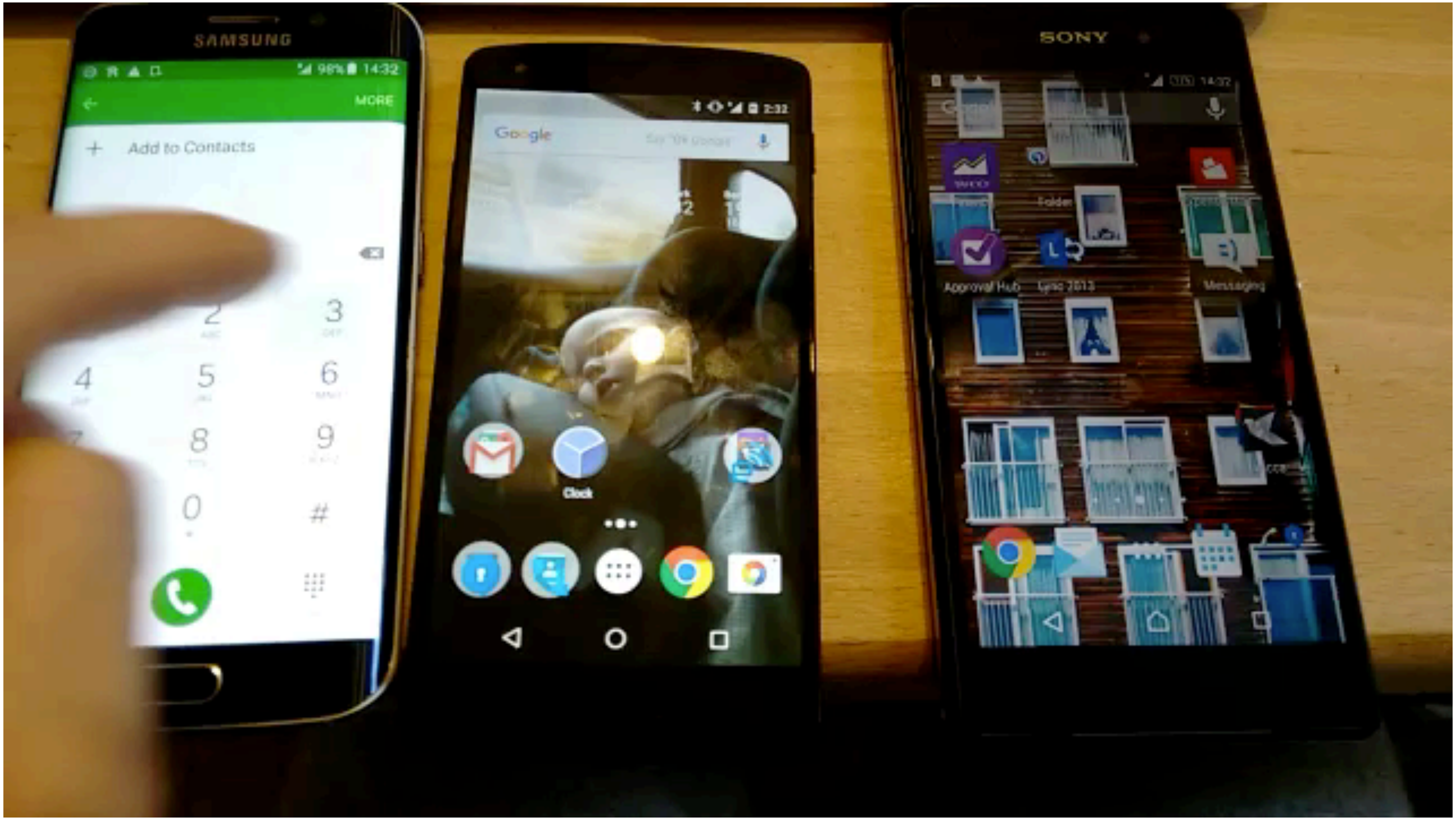
```
{
  sub_404EAEF4((char *)(unsigned __int8)in
  if ( is_progress_ind_set() == 1 )
  {
    copy_progress_ind((int)&v24);


    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!

Comsecuris

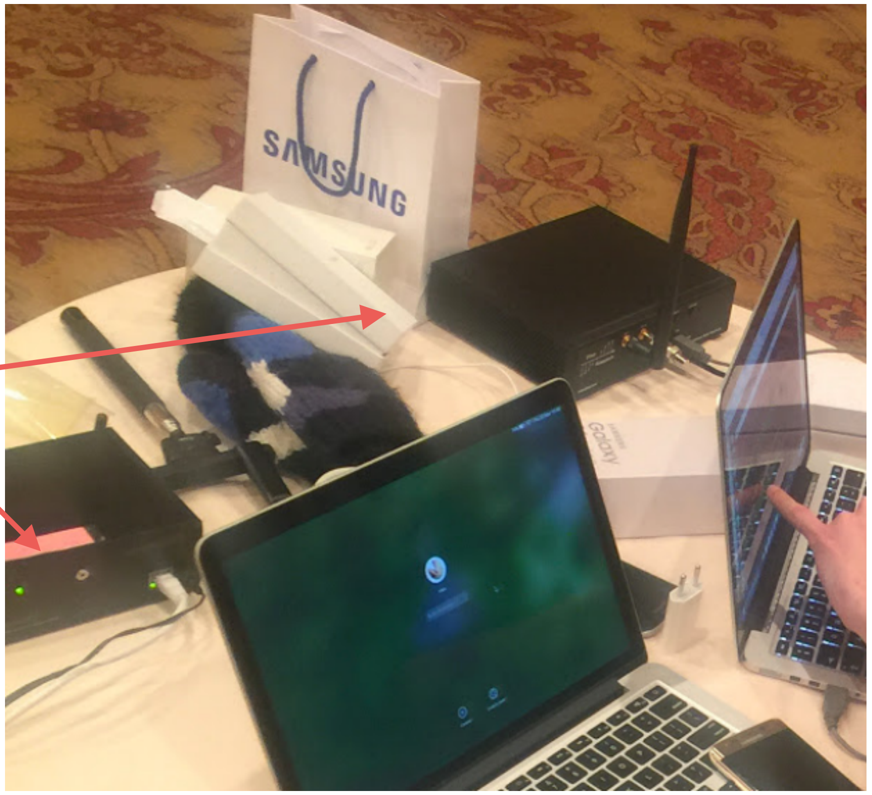
DEMO



Comsecuris

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 $



Comsecuris

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

Comsecuris

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.)

Comsecuris

Exploit Payloads

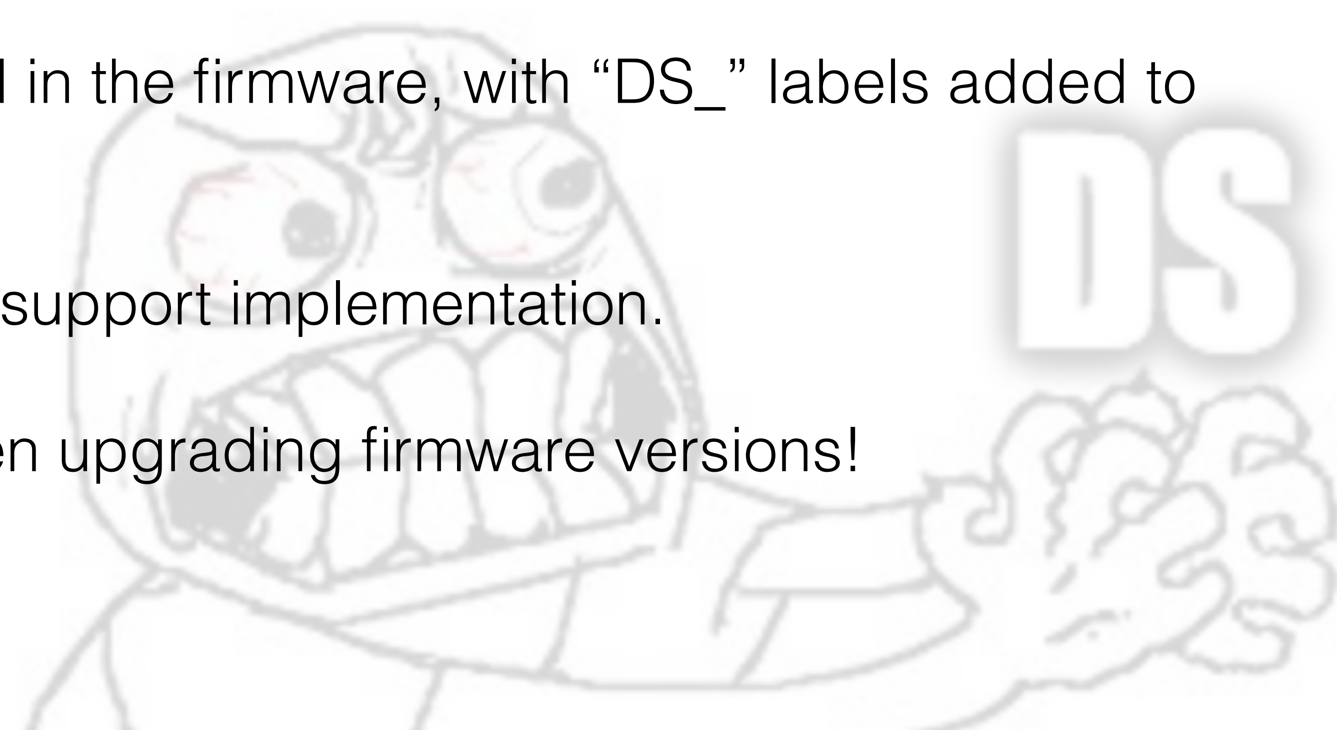
- baseband code execution has limited functionality
  - **not** the master over application processor/memory (these days), but loaded by apps processor! (pls get this right in public debates)
- baseband sees all* data/signaling exchanged with cellular networks though (calls, text messages, data)
- typical 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!

Comsecuris

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!



Comsecuris

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 ;)

Comsecuris

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

Comsecuris

Tools Release

- 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.

Comsecuris

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)