Reflections on Trusting TrustZone

Dan Rosenberg

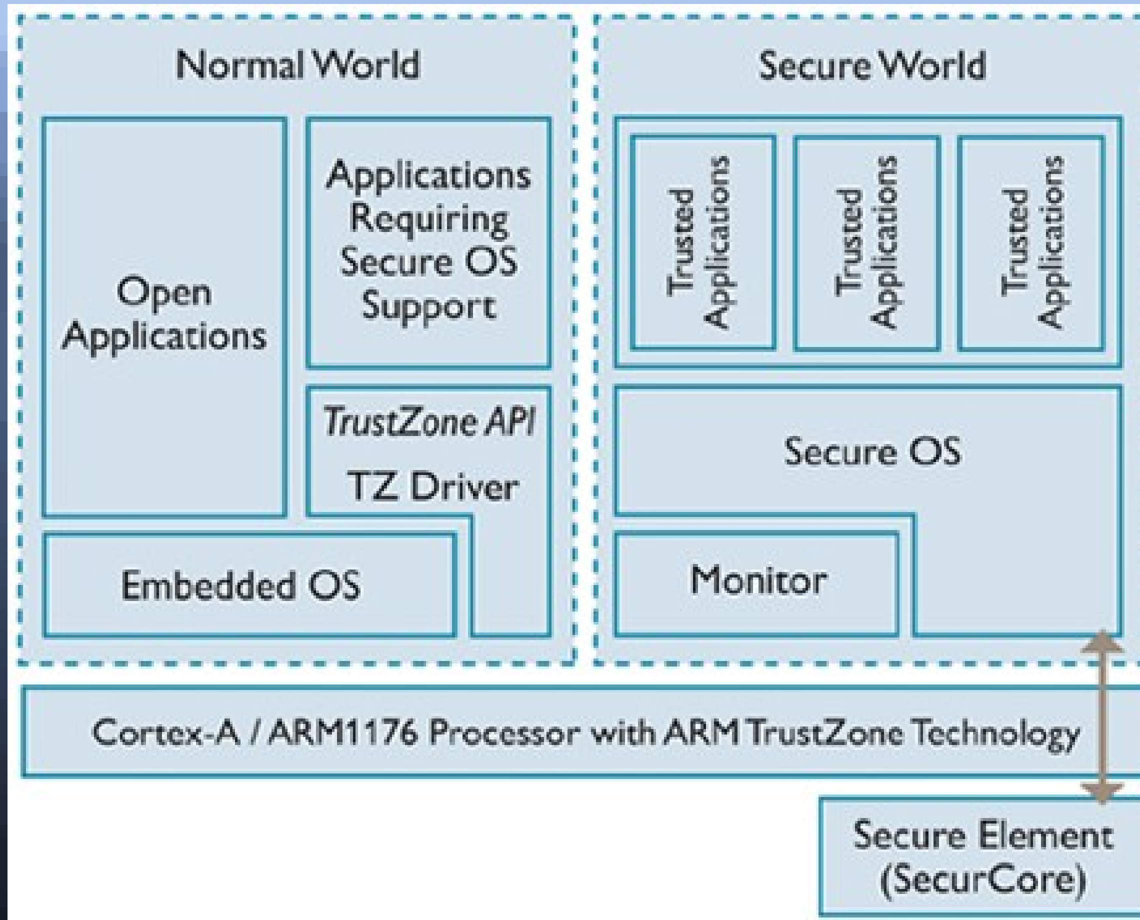
azimuth SECURITY

What is TrustZone?

- "ARM® TrustZone® technology is a system-wide approach to security for a wide array of client and server computing platforms, including handsets, tablets, wearable devices and enterprise systems. Applications enabled by the technology are extremely varied but include payment protection technology, digital rights management, BYOD, and a host of secured enterprise solutions."

azimuth SECURITY

TrustZone Architecture



* Image courtesy ARM Ltd.

azimuth SECURITY

Real-World Uses

- DRM (WideVine, PlayReady, DTCP-IP)
- Secure key storage (dm-verify)
- Mobile payments
- Protected hardware (framebuffer, PIN entry)
- Management of secure boot (via QFuses)
- Kernel integrity monitoring (TIMA)

Prior Work

- "Next Generation Mobile Rootkits" (Thomas Roth, 2013)
- "Unlocking the Motorola Bootloader" (Azimuth Security, 2013)
- Maybe a few HTC S-OFF exploits (undocumented)

Motivation

- High value target
- Very little public research/scrutiny

Target

- Qualcomm Secure Execution Environment (QSEE)
- Majority market share among mid/high-end Android phones
  - Samsung GS4/GS5/Note3, LG Nexus 4/Nexus 5/G2/G3, Moto X, HTC One series...

Toolchain

- TrustZone images included in firmware available online or pulled from devices
- IDA Pro
- Qualcomm loader for earlier TZ, now it's ELF

azimuth SECURITY

Attack Surface

- Software exceptions: Secure Monitor Call (SMC) interface
- Hardware exceptions: IRQ, FIQ, external abort
- Shared memory interface (mostly MobiCore)
- eMMC flash (e.g. secure boot)
- Trustlet-specific calls

Attacker Assumptions

- Arbitrary code execution on device
  - Extremely minimal remote attack surface
- Kernel privileges
  - Ability to issue SMC instructions
  - Otherwise, practically no ability to interact with TrustZone directly
- Crashes/DoS bugs are not security relevant
  - The kernel can already bring down the device

azimuth SECURITY

QSEE SCM Interface

- Code in Qualcomm trees (CAF) at *arch/arm/mach-msm/scm.c*
- Not a typo: Qualcomm chose SCM ("Secure Channel Manager") as name for Linux kernel driver that interacts with QSEE via SMC
- Two calling conventions: call-by-register, or request/response structures

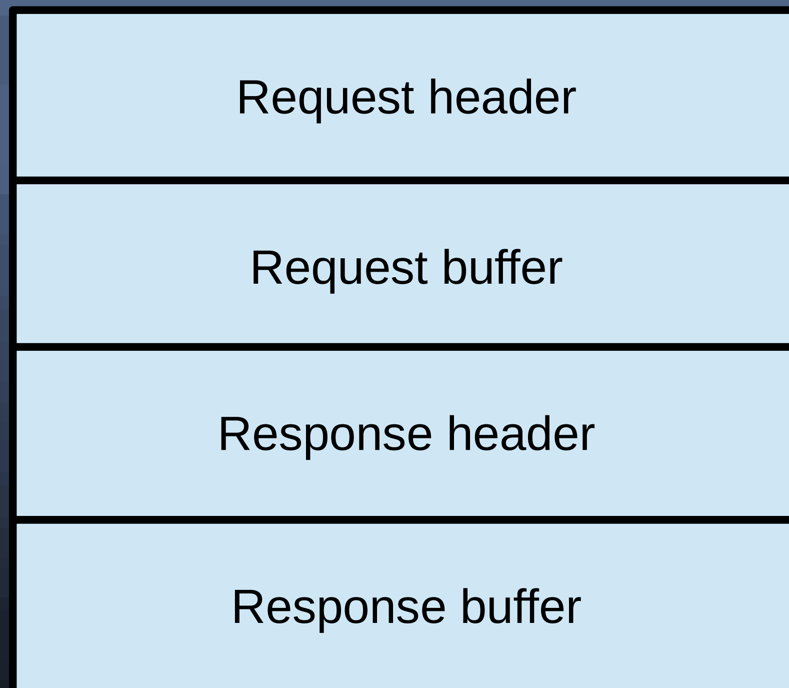
SCM Call-by-Register Convention

- Load *r0* with OR'd value containing SMC command number, flags, and number of arguments:

```
#define SCM_ATOMIC(svc, cmd, n) (((((svc) << 10)|((cmd) & 0x3ff)) << 12) | \
                SCM_CLASS_REGISTER | \
                SCM_MASK_IRQS | \
                (n & 0xf))
```

- Arguments go in *r1,r2,...,rN*

SCM Command Structures




```
struct scm_command {
   u32 len;
   u32 buf_offset;
   u32 resp_hdr_offset;
   u32 id;
   u32 buf[0];
};
```

```
struct scm_response {
    u32 len;
    u32 buf_offset;
    u32 is_complete;
};
```

azimuth SECURITY

Structure Sanity Checking

1. cmd.len >= 16
   (Command length is greater than size of request header)
2. cmd.buf_offset < cmd.len
   (Start of request buffer resides inside command buffer)
3. cmd.buf_offset >= 16
   (Request buffer does not overlap with request header)
4. cmd.resp_hdr_offset <= cmd.len – 12
   (Entire response header resides inside command buffer)
5. **qsee_is_ns_memory(cmd, cmd.len) returns true**
   (Entire command buffer resides in non-secure memory)

Secure Memory Checking

- Series of functions to check if memory is "protected"
- Hard-coded list of regions with flags to indicate memory attributes
- Analogous to Linux kernel's access_ok() checks
  - “Is this memory is safe for TZ to operate on?”

Integer Overflow Vulnerability

- Take another look at the invocation of secure memory checking in validating the SCM command structure:

```
qsee_is_ns_memory(cmd, cmd.len)
```

- What if (cmd + cmd.len) overflows 32-bit integer?

azimuth
SECURITY

Secure Memory Checking Pseudocode

```
int qsee_is_ns_memory(long addr, long size)
{
    return qsee_range_not_in_region(qsee_region_list, addr, addr+size);
}

int qsee_range_not_in_region(void *region_list, long start, long end)
{
    long tmp;

    if (end < start) {
        tmp = start;
        start = end;
        end = tmp;
    }

    /* Validate that start to end doesn't overlap
     * secure list */
    ...
}
```

azimuth SECURITY

Pathological Command Buffer

```
cmd.len = 0xfffff000
cmd.buf_offset = 0xffffe000
cmd.resp_hdr_offset =
 arbitrary value < 0xfffff000
```

1. cmd.len >= 16
2. cmd.buf_offset < cmd.len
3. cmd.buf_offset >= 16
4. cmd.resp_hdr_offset <= cmd.len – 16
5. qsee_is_nonsecure_memory(cmd, cmd.len) returns true

What is Written to Response Output?

- Hard-coded response buffer for all requests that receive output:

```
rsp.len = 12;
rsp.buf_offset = 12;
rsp.is_complete = 1;
```

Result: Arbitrary Secure Memory Write Primitive

- By crafting SMC request to exploit integer overflow, possible to cause QSEE to write three words (0x0000000c 0x0000000c 0x00000001) to response structure, which can reside in arbitrary secure memory!

- Can we achieve arbitrary secure code execution?

How Can This Be Exploited?

- Memory layout of QSEE is known
  - Image resides unencrypted on eMMC flash
  - Loaded at known physical address
- Most of RAM is non-secure memory
- Can't we just clobber part of a function pointer in secure memory to point to our non-secure payload and trigger?
  - e.g. 0xdeadbeef → 0xcadbeef

Sorcery!

- This doesn't appear to work
- Suspect QSEE has mechanism to prevent TZ execution from non-secure pages
- Undocumented black hole
- Any Qualcomm or ARM employees in the audience?

Building Better Primitives

- A 12-byte uncontrolled write makes exploitation somewhat difficult
  - Unaligned writes clobber extra words, potentially unstable
  - Minimal options for redirecting pointers to non-secure memory
- How can we use this to build a more flexible primitive?

Region Lists Revisited

- List of protected memory regions composed of structures similar to the following:

```
struct qsee_memory_region {
        int id;
        int flags;
        unsigned long start;
        unsigned long end;
}
```

azimuth SECURITY

Region List Corruption

- Use 12-byte write to clobber flags, start, and end addresses for entry corresponding to the QSEE image
- Result: all checks intended to ensure safety of user-provided output pointers pass
- Now we can write arbitrary secure memory with any value written as output by QSEE!

Choosing A New Write Primitive

- Enumerate SMC handlers
- Eliminate those that don't write any output
- Choose best option based on task at hand
- But then what?

SMC Handler Table

- In QSEE, SMC table entries are variable length:

```
struct smc_entry {
        unsigned int smc_num;
        char *handler_name;
        unsigned int flags;
        int (*smc_handler)();
        unsigned int num_args;
        unsigned int arg_lens[];
}
```

- Iterates through table using *num_args* to calculate entry length, matching against *smc_num*

azimuth SECURITY

SMC Table Extension Attack

- Use arbitrary secure memory write to modify *num_args* field of SMC table entry
- Expand size of entry so iterator jumps to supposed next entry in attacker-controlled non-secure memory
- Create fake entry to call arbitrary QSEE functions with arbitrary arguments!

Arbitrary TZ Code Execution

- Find *memcpy*, copy all of secure memory to a non-secure buffer, break all DRM/secure key storage
- Disable TIMA
- Invoke OEM-specific functionality to e.g. unlock the bootloader permanently :-)

DEMO!

azimuth SECURITY

Lessons Learned

- Analysis/exploitation is made much easier due to lack of encryption of TZ image
  - Compare to iOS
- Parsing of complex data structures is an obvious likely point of failure
- As a vuln researcher, learn to trust your gut
  - If it looks sketchy, it probably is

azimuth SECURITY

Lessons Learned

- Single points of failure are a bad idea
  - Compare Motorola's secure boot to Samsung's
- Improving security by minimizing attack surface is a good idea, but feature creep will eliminate this advantage entirely
- Marketing does not eliminate software bugs

Questions?