TrustZone TEEs

An Attacker's Perspective

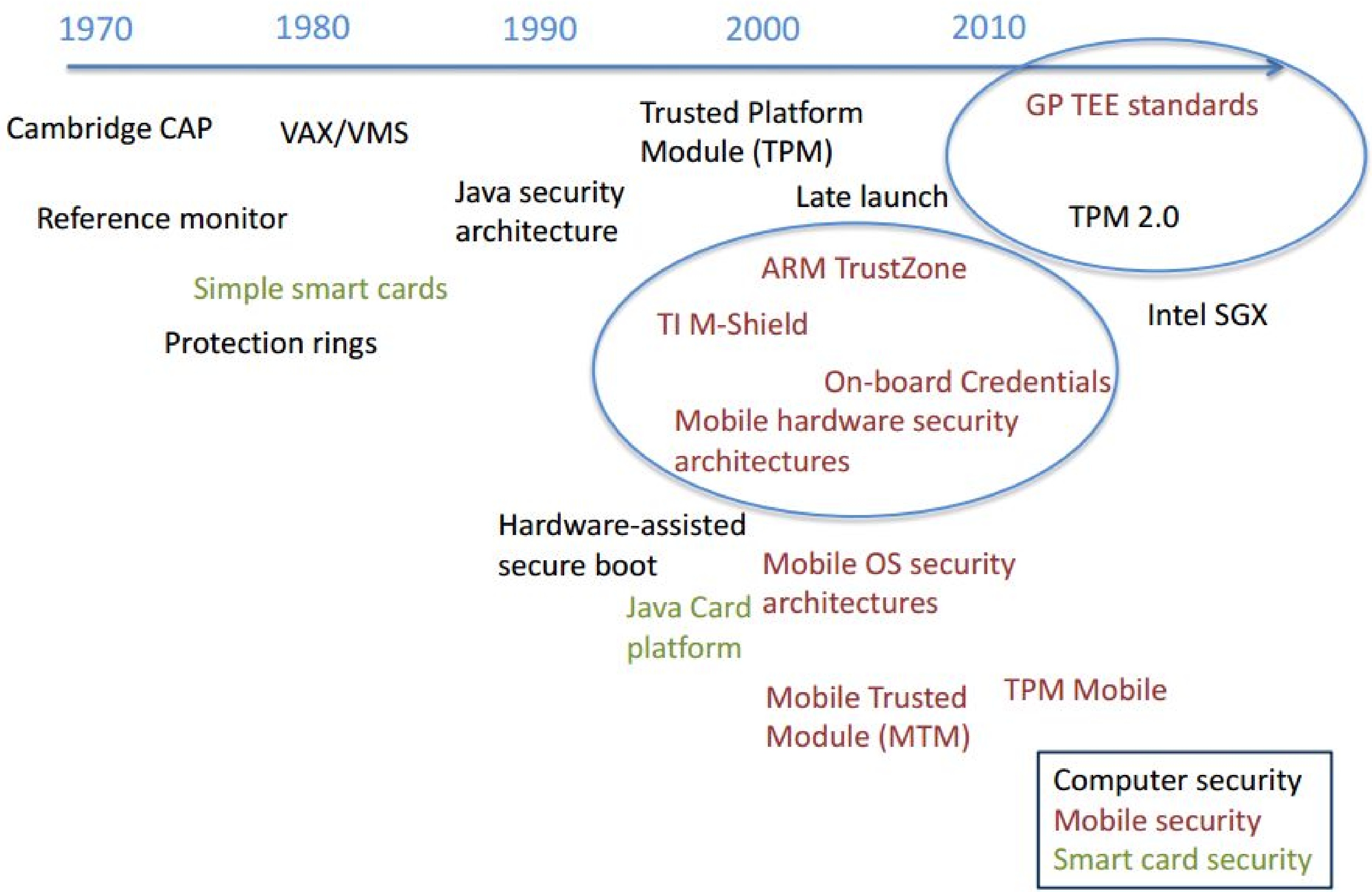
Gal Beniamini

What is a Trusted Execution Environment?

- A designated “secure” area of the application processor
    - Aims to provide **isolation** using a variety of hardware features
    - Guarantees **confidentiality** of data processed within the environment
    - Ensures the **integrity** of all code running within the TEE

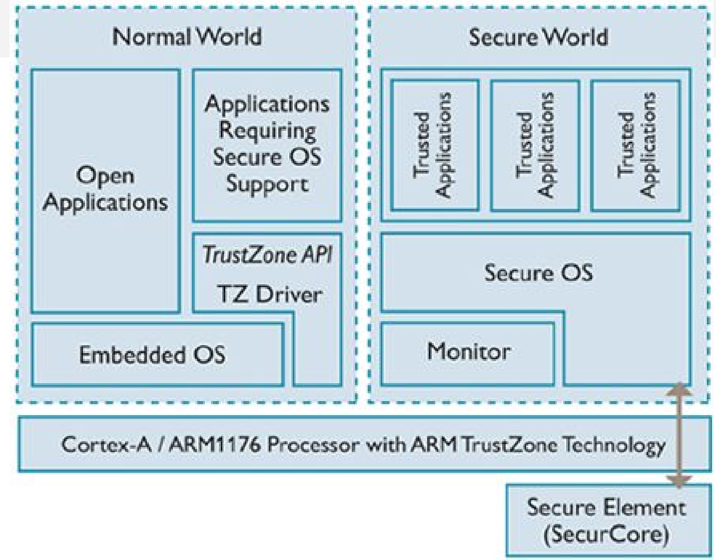
- In this talk, we’ll focus on TrustZone-based TEE solutions
    - Mainly, QSEE (Qualcomm) and some MobiCore (Trustonic)
    - Specifically, QSEE has been present in nearly all Nexus devices

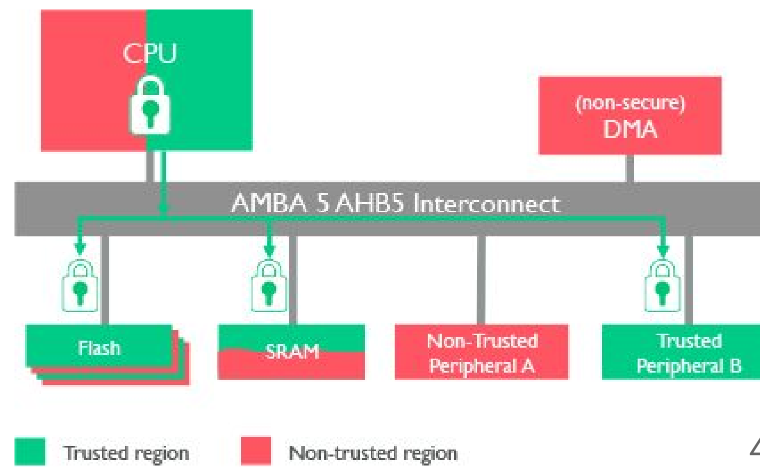
Historical Perspective on Mobile Hardware Security



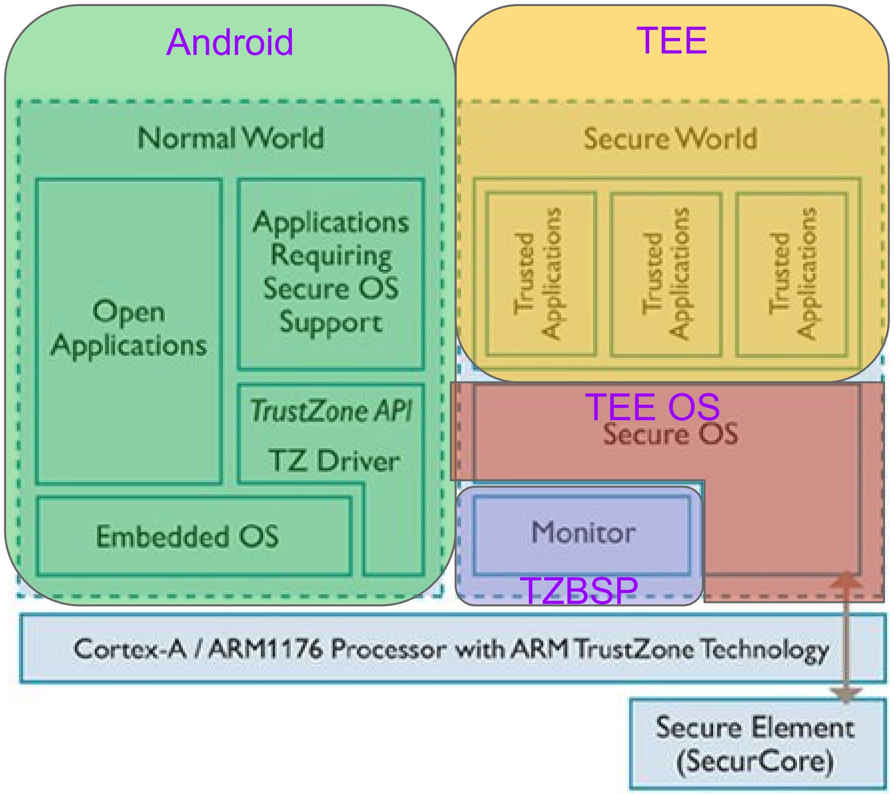
What is TrustZone?

- A hardware architecture designed by ARM, introduced in ARMv6
- Specifies two “Virtual Processors”, backed by hardware
  - One for the “Secure World”, one for the “Normal World”
  - The current “world” is denoted by the *NS* bit
- Peripherals can also be marked as “Secure” or “Non-Secure”
  - These peripherals can access the AMBA AXI bus (AXPROT, AWPROT, etc.)
  - Allows fine-grained memory controllers to prevent illegal non-secure access
  - For example, this allows for separation of memory into “Secure” and “Non-Secure” regions



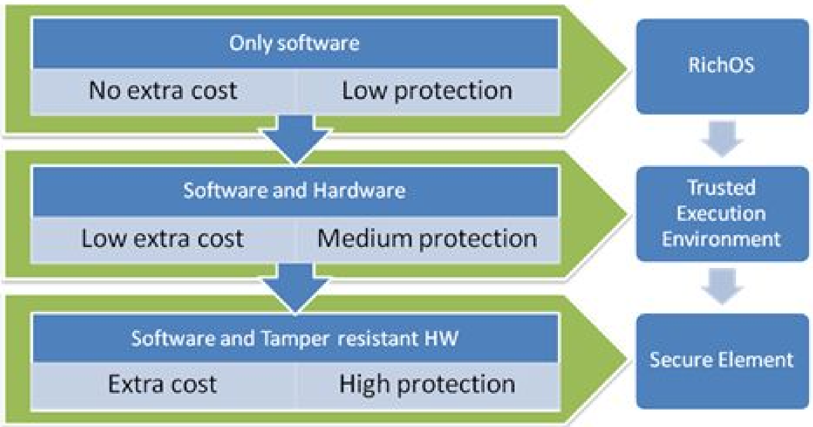


Typical TrustZone-based TEEs



TEEs vs. Secure Elements

- A Secure Element is a tamper-resistant platform
  - Capable of hosting applications
  - Secure storage of cryptographic material
  - Normally implemented using a separate chip
- Being discrete components, SEs can offer better security guarantees
  - In fact, they're already used by some Android devices
  - For example, Samsung KNOX utilizes NXP SEs
- But... Secure Elements are slow in comparison to TEEs
  - Remember - TEEs run on the application processor!



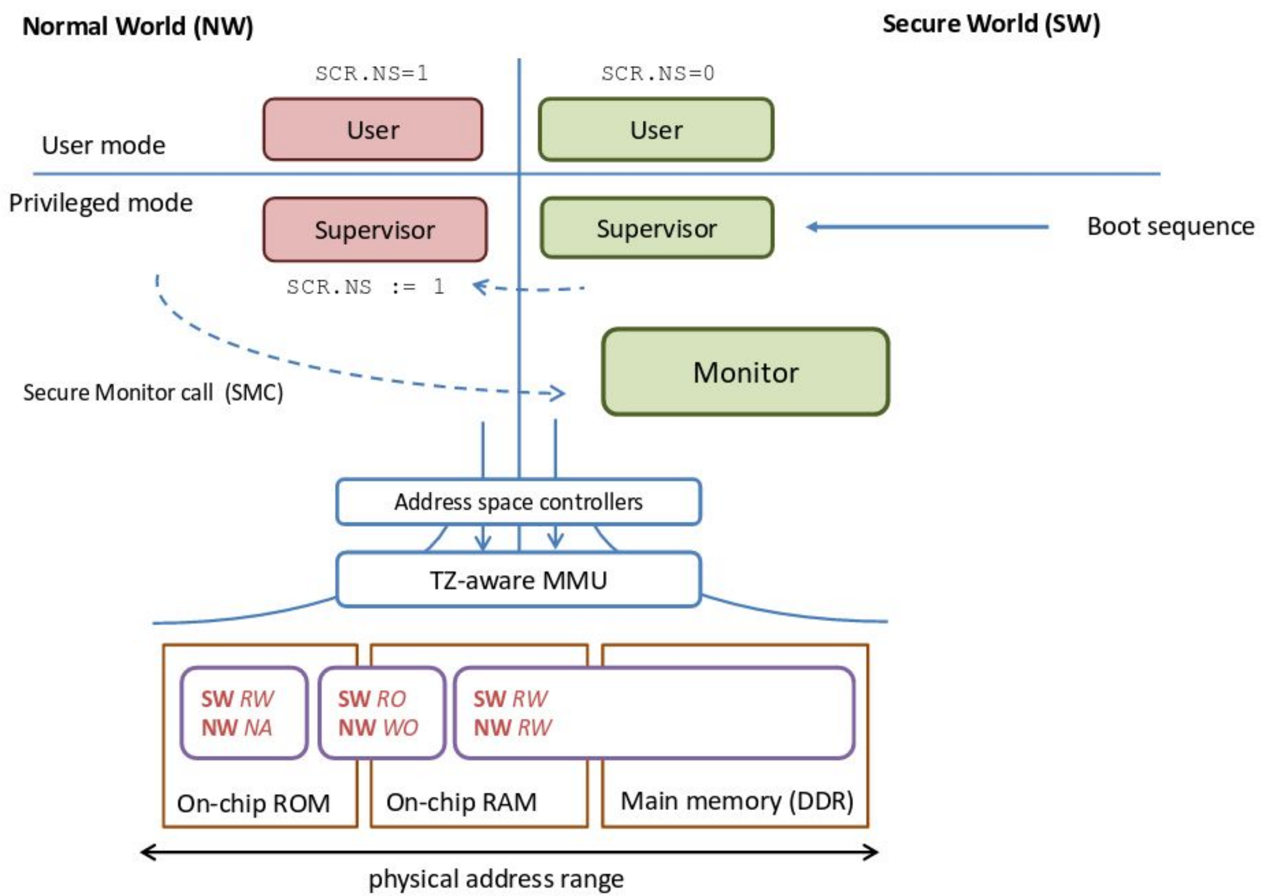
TEE Memory Isolation

- Usually, information processed within the TEE is highly sensitive
  - Can include payment information (for systems without a SE)
  - Encryption and/or HMAC keys (such as the KeyMaster implementation)
  - For most devices, there’s also biometric data - e.g. the image from the fingerprint sensor
    - In fact, the TA simply runs (software-based) SIFT to perform the matching
- So how can we keep this information from an attacker?
  - The security-model assumes that the attacker has supervisor-mode code execution on the application processor
  - This (classically) implies full access to the DRAM

TEE Memory Isolation

- This is where some extra hardware comes in handy!
- Different SoCs implement this isolation in a variety of ways...
- … But essentially, it boils down to this:
  - “Sacrifice” pre-defined regions of the DRAM for TrustZone (/TEE)
  - Guard against access to these regions using TrustZone-aware memory protection units (recall that peripherals can access the NS-bit)
- In Qualcomm’s case, these units are called XPUs
  - They are configured by the TrustZone kernel during boot
  - XPUs also prevent disallowed access by the Secure World (overriding the ARM MMU)

TEE Memory Isolation



Looking at TEEs from an “adversarial” PoV

- We’ve seen some fortifications of TEEs which aim to make them more secure
  - Memory Isolation
  - Cryptographic verification of all loaded trustlets
  - Trustlets are isolated from one another the the TEE OS
  - The TEE is a small TCB, which should be easier to verify than a “rich” OS

- So it seems like the existence of a TEE is an overall security benefit
  - Or is it?

The State of Android Security

- Android security is getting quite good!
- There's a vast (and ever-expanding) set of security mechanisms:
  - SEAndroid
  - App sandboxing via the Linux Kernel (running under different User-IDs)
  - Android permission enforcement
  - (Nearly) full ASLR, non-executable heap & stack, EXECMEM, stack cookies
  - Selective additional hardening by compiling with UBSan
  - etc.
- Most importantly: open-source
  - Builds on many years of security improvements and wide-spread auditing

TrustZone: The soft underbelly of Android devices

- "Feature creep" has gradually expanded the TCB of TEE OSes
- The TEE OS must support many TA use-cases:
    - TAs that interact with the "Non-Secure World"
        - Samsung's TIMA PKM, LKMAUTH
    - TAs that perform cryptographic operations
        - KeyMaster
    - TAs for Trusted User-Interface (e.g., trusted keypad)
        - Samsung's KNOX TUI
    - TAs for processing biometric information
    - TAs that interact with one another

“Small” TCB

```
-rwxrwxrwx 1 1000 1000  37133 2008-12-31  00060308060501020000000000000000.tlbin
-rwxrwxrwx 1 1000 1000  25733 2008-12-31  07010000000000000000000000000000.tlbin
-rwxrwxrwx 1 1000 1000  10237 2008-12-31  07060000000000000000000000000000.tlbin
-rwxrwxrwx 1 1000 1000   6077 2008-12-31  08130000000000000000000000000000.tlbin
-rwxrwxrwx 1 1000 1000 238025 2008-12-31  ffffffff000000000000000000000004.tlbin
-rwxrwxrwx 1 1000 1000  58129 2008-12-31  ffffffff000000000000000000000005.tlbin
-rwxrwxrwx 1 1000 1000  35725 2008-12-31  ffffffff00000000000000000000000a.tlbin
-rwxrwxrwx 1 1000 1000   8933 2008-12-31  ffffffff00000000000000000000000b.tlbin
-rwxrwxrwx 1 1000 1000 144737 2008-12-31  ffffffff00000000000000000000000c.tlbin
-rwxrwxrwx 1 1000 1000 112477 2008-12-31  ffffffff00000000000000000000000d.tlbin
-rwxrwxrwx 1 1000 1000 562713 2008-12-31  ffffffff00000000000000000000000e.tlbin
-rwxrwxrwx 1 1000 1000  33821 2008-12-31  ffffffff00000000000000000000000f.tlbin
-rwxrwxrwx 1 1000 1000 456729 2008-12-31  ffffffff000000000000000000000012.tlbin
-rwxrwxrwx 1 1000 1000  62945 2008-12-31  ffffffff000000000000000000000013.tlbin
-rwxrwxrwx 1 1000 1000 490981 2008-12-31  ffffffff000000000000000000000014.tlbin
-rwxrwxrwx 1 1000 1000  27053 2008-12-31  ffffffff000000000000000000000016.tlbin
-rwxrwxrwx 1 1000 1000 102317 2008-12-31  ffffffff000000000000000000000017.tlbin
-rwxrwxrwx 1 1000 1000  12361 2008-12-31  ffffffff000000000000000000000019.tlbin
-rwxrwxrwx 1 1000 1000 638177 2008-12-31  ffffffff00000000000000000000001f.tlbin
-rwxrwxrwx 1 1000 1000 280677 2008-12-31  ffffffff00000000000000000000002e.tlbin
-rwxrwxrwx 1 1000 1000 444141 2008-12-31  ffffffff000000000000000000000038.tlbin
-rwxrwxrwx 1 1000 1000 463281 2008-12-31  ffffffff00000000000000000000003e.tlbin
-rwxrwxrwx 1 1000 1000  11045 2008-12-31  ffffffff000000000000000000000041.tlbin
-rwxrwxrwx 1 1000 1000   2797 2008-12-31  ffffffff000000000000000000000042.tlbin
-rwxrwxrwx 1 1000 1000 127929 2008-12-31  ffffffffd00000000000000000000004.tlbin
-rwxrwxrwx 1 1000 1000 329769 2008-12-31  ffffffffd0000000000000000000000a.tlbin
-rwxrwxrwx 1 1000 1000  16817 2008-12-31  ffffffffd0000000000000000000000e.tlbin
-rwxrwxrwx 1 1000 1000  27677 2008-12-31  ffffffffd00000000000000000000014.tlbin
-rwxrwxrwx 1 1000 1000  19797 2008-12-31  ffffffffd00000000000000000000016.tlbin
-rwxrwxrwx 1 1000 1000  13129 2008-12-31  ffffffffd00000000000000000000017.tlbin
-rwxrwxrwx 1 1000 1000 233777 2008-12-31  fffffffff0000000000000000000001b.tlbin
-rwxrwxrwx 1 1000 1000  85285 2008-12-31  fffffffff0000000000000000000001e.tlbin
```

TEE for Two

- Some OEMs, such as Samsung, rely on features which aren't present in all TEEs
  - For example, the KNOX TUI is only supported by the MobiCore TEE
- On the other hand, Samsung ships Qualcomm and Exynos variants for most devices
- In order to work-around this shortcoming, some devices ship with two TEEs
  - In Samsung's case, this means both QSEE and MobiCore
- This significantly complicates the TEE OS, adding even more potential attack surface
  - How do applications communicate cross-TEE?
  - How does cross TEE isolation work?
  - Is the TEE API precisely emulated for all TAs?
  - etc.

State of TEE OS Security

- Nearly no public research has been done on TEE OSes
- The implementation is completely proprietary
  - Ergo, the only way to gain insight into TEEs is by reverse-engineering
- Luckily, there aren’t too many TEEs around
  - QSEE and MobiCore account for all Qualcomm and Exynos devices
  - MobiCore (trustonic) is also present on MediaTek chips
- So… let’s start by surveying the security mechanisms in the TEE itself
  - Surely TEEs are developed with security in mind
  - Hopefully we’ll get to see some great security architecture

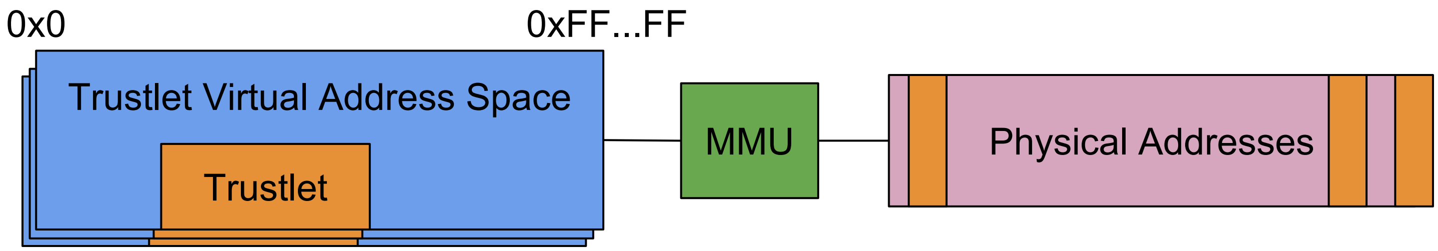
QSEE Trustlets - Memory Protections

- All QSEE trustlets are loaded into a “secure” memory region - “secapp-region”
- The region is XPU-protected, meaning it can’t be accessed by the “Non-Secure World”
- The QSEOS loader loads trustlets into a randomly chosen address within “secapp”
    - But the trustlets’ translation table is flat!
    - This means that each trustlet views only the physical memory region
    - Ergo, the number of (virtual) base addresses is very limited, resulting in ~9 bits of entropy

```
qcom,qseecom@fe806000 {
        compatible = "qcom,qseecom";
        reg = <0x7f00000 0x500000>;
        reg-names = "secapp-region";
        qcom,disk-encrypt-pipe-pair = <2>;
        qcom,file-encrypt-pipe-pair = <0>;
        qcom,hlos-ce-hw-instance = <1>;
        qcom,qsee-ce-hw-instance = <0>;
        qcom,support-fde;
        qcom,support-pfe;
        qcom,msm_bus,name = "qseecom-noc";
        qcom,msm_bus,num_cases = <4>;
        qcom,msm_bus,active_only = <0>;
        qcom,msm_bus,num_paths = <1>;
        qcom,no-clock-support;
        qcom,msm_bus,vectors =
                <55 512 0 0>,
                <55 512 3936000000 393600000>,
                <55 512 3936000000 393600000>,
                <55 512 3936000000 393600000>;
};
```

MobiCore TAs - Memory Protections

- Luckily - MobiCore decided to use the entire VAS for TAs!
- ...Unluckily - there is no form of ASLR at all
  - All TAs are loaded into a fixed address specified in the MCLF header
  - The “support libraries” are also loaded into predefined addresses
- This means that not only can a local attacker brute-force the loading address
  - But also any TA vulnerability is trivially remotely exploitable
  - No need to find information disclosure vulnerabilities



MobiCore TAs - Memory Protections

```
/**
 * Version 2 MCLF header.
 */
typedef struct {
    mclfIntro_t             intro;          /**< MCLF header start with the mandatory intro. */
    uint32_t                flags;          /**< Service flags. */
    memType_t               memType;        /**< Type of memory the service must be executed from. */
    serviceType_t           serviceType;    /**< Type of service. */

    uint32_t                numInstances;   /**< Number of instances which can be run simultaneously. */
    mcUuid_t                uuid;           /**< Loadable service unique identifier (UUID). */
    mcDriverId_t            driverId;       /**< If the serviceType is SERVICE_TYPE_DRIVER the Driver ID is used. */
    uint32_t                numThreads;     /**<
                                             * <pre>
                                             * <br>Number of threads (N) in a service depending on service type.<br>
                                             *
                                             *   SERVICE_TYPE_SP_TRUSTLET: N = 1
                                             *   SERVICE_TYPE_SYSTEM_TRUSTLET: N = 1
                                             *   SERVICE_TYPE_DRIVER: N >= 1
                                             * </pre>
                                             */
    segmentDescriptor_t     text;          /**< Virtual text segment. */
    segmentDescriptor_t     data;          /**< Virtual data segment. */
    uint32_t                bssLen;        /**< Length of the BSS segment in bytes. MUST be at least 8 byte. */
    addr_t                  entry;         /**< Virtual start address of service code. */
    uint32_t                serviceVersion; /**< Version of the interface the driver exports. */
} mclfHeaderV2_t, *mclfHeaderV2_ptr;
```

QSEE Trustlets - Memory Corruption Mitigations

- QSEE trustlets use a “stack cookie” in order to prevent exploitation of stack-overflow vulnerabilities
  - The cookie itself is generated using the TZ kernel RNG
  - The cookie is re-generated after each QSEE call
- However…
  - Many QSEE applications use BSS-allocated buffers
  - These buffers are not protected by a random “cookie”
- Moreover, the trustlet’s stack resides directly after its BSS
  - There is no guard page (the BSS, heap and stack are carved out of a single segment)
  - This means that every BSS or heap overflow gives direct control over the stack, and therefore full code execution

MobiCore TAs - Memory Corruption Mitigations

- There is no stack cookie mitigation on MobiCore
  - Every stack-overflow vulnerability is trivially exploitable
- Coupling the complete lack of ASLR on MobiCore with no stack cookie:
  - Renders every stack-overflow trivially *remotely* exploitable
  - Removes the need for information leaks or position-independent exploits
- MobiCore TAs also load the “support library” into the address space of each TA
  - The loading address is fixed (part of the TA header)
  - The large code-base allows for comfortable and generic ROP gadgets (which are cross-TA)

QSEOS Trustlet Isolation

- As we’ve seen before, QSEE trustlets are isolated from one another
- Trustlets cannot access the memory of other loaded trustlets
    - Even if they know their loading address within “secapp”
- However, QSEOS is able to access all trustlet memory (just like any other OS)
    - Setting the DACR in the ARM MMU allows full TA access to the kernel-context of a single trustlet, which prevents the need to “mess” with the translation table
    - Setting the DACR also enables QSEOS to write (and execute) code within a trustlet
- Therefore, the trustlet isolation is only “as strong” as the TrustZone kernel
    - Finding a vulnerability in the TZ kernel breaks all isolation guarantees

QSEOS Trustlet Isolation

- QSEOS provides a substantial amount (>70) of system calls to QSEE trustlets
  - Memory management syscalls (e.g., flushing the I/D caches)
  - Creation of cryptographic handles for various crypto primitives
  - Querying the state of the SoC (e.g., reading SW or HW fuses)

```
FE81BC68 syscall_table_6 DCD 1                    ; DATA XREF: seg003:FE81BDC4↓o
FE81BC6C                 DCD Invalidate_entire_Unified_TLB_Inner_Shareable_0
FE81BC70                 DCD 2
FE81BC74                 DCD invalidate_inst_tlb
FE81BC78                 DCD 3
FE81BC7C                 DCD invalidate_data_tlb
FE81BC80                 DCD 4
FE81BC84                 DCD invalidate_mmu_cache_and_icache
FE81BC88                 DCD 5
FE81BC8C                 DCD _armv7_mmu_cache_flush_1
FE81BC90                 DCD 6
FE81BC94                 DCD _armv7_mmu_cache_flush
FE81BC98                 DCD 7
FE81BC9C                 DCD _armv7_mmu_cache_flush_0
FE81BCA0                 DCD 8
FE81BCA4                 DCD invalidate_data_cache
FE81BCA8                 DCD 9
FE81BCAC                 DCD flush_data_cache
FE81BCB0                 DCD 0xA
```

QSEOS Trustlet Isolation

- As QSEOS is proprietary, no prior public research has been done into it...
- Auditing the QSEOS syscall implementations revealed the embarrassing truth (CVE-2016-2431):
  - Some syscalls receive pointers from QSEE (e.g., the location at which to allocate a cryptographic object)
  - However, QSEOS made no validations in order to make sure that these addresses indeed reside in the QSEE region for that specific trustlet
  - Therefore, passing a pointer to QSEOS within a syscall would result in corruption of the TrustZone kernel memory
  - This could be leveraged to enable full code execution in the TrustZone kernel

QSEOS Trustlet Isolation

- Since all syscalls were affected, finding a comfortable exploitation primitive wasn't too difficult
    - The kernel-mode context for the trustlet did not have translations for the memory addresses of other trustlets
    - However, it did contain translations for the entire TZ kernel
- This meant an attack could trivially overwrite malleable data in the TZ kernel in order to achieve code execution
    - Recall, however, that most of the TZ kernel is XPU-protected
        - Luckily most ≠ all; indeed some TZ code segments were left unprotected
    - On the other hand, once in the TZ kernel, we can disable the XPUs

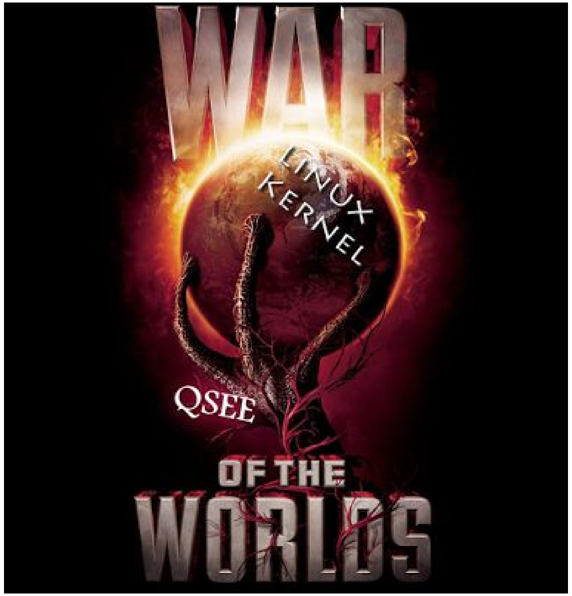
```
signed int __fastcall qsee_cipher_get_param(void *cipher, unsigned int param_type, void *output, int unknown)
{
  signed int res; // r4@1
  int v6; // r0@7
  unsigned int v7; // r1@7
  int v9; // [sp+0h] [bp-18h]@1

  res = 0;
  v9 = 1;
  if ( cipher && output && unknown )
  {
    if ( param_type < 8 )
    {
      switch ( param_type )
      {
        case 0u:
          v6 = *((_DWORD *)cipher + 1);
          v7 = 2;
          goto LABEL_10;
        case 1u:
          v6 = *((_DWORD *)cipher + 1);
          v7 = 3;
          goto LABEL_10;
        case 7u:
          v6 = *((_DWORD *)cipher + 1);
          v7 = 8;
          goto LABEL_10;
        case 4u:
          v6 = *((_DWORD *)cipher + 1);
          v7 = 5;
          goto LABEL_10;
        case 6u:
          v6 = *((_DWORD *)cipher + 1);
          v7 = 7;
          goto LABEL_10;
        case 5u:
          v6 = *((_DWORD *)cipher + 1);
          v7 = 6;
LABEL_10:
          if ( sub_FE868EC0(v6, v7, (int)output, (int *)unknown) )
            res = -1;
          return res;
        case 2u:
          if ( sub_FE868EC0(*((_DWORD *)cipher + 1), 4u, (int)output, &v9) )
            res = -1;
          if ( (unsigned int)*(_BYTE *)output <= 5 )
            return res;
          return -4;
        case 3u:
          *(_BYTE *)output = *((_BYTE *)cipher + 8);
          return res;
        case 2u:
          if ( sub_FE868EC0(*((_DWORD *)cipher + 1), 4u, (int)output, &v9) )
            res = -1;
          if ( (unsigned int)*(_BYTE *)output <= 5 )
            return res;
          return -4;
        default:
          break;
      }
    }
    res = -4;
  }
  else
  {
    res = -16;
  }
  return res;
}
```

TEEs as a High-Value Target

- As we’ve seen, the security mechanisms currently employed by TEEs are awful
    - For MobiCore - no ASLR and no stack cookie
    - For QSEE - ~9 bit ASLR, and the stack is after the BSS, with no guard page
- Also, the trustlets themselves are proprietary, along with the TEE OSes
    - For QSEOS, this has allowed a trivial vulnerability to persist for many years
    - No doubt the same is true also for MobiCore (more research on the way!)
- ...But what about elevating the TEEs as a means of attacking the HLOS?
    - What access controls are placed by the TEE OS to prevent abuse by TAs?
    - Can the TEE itself be used to mount an attack on the “Non-Secure World”?

A storm in a TEEpot

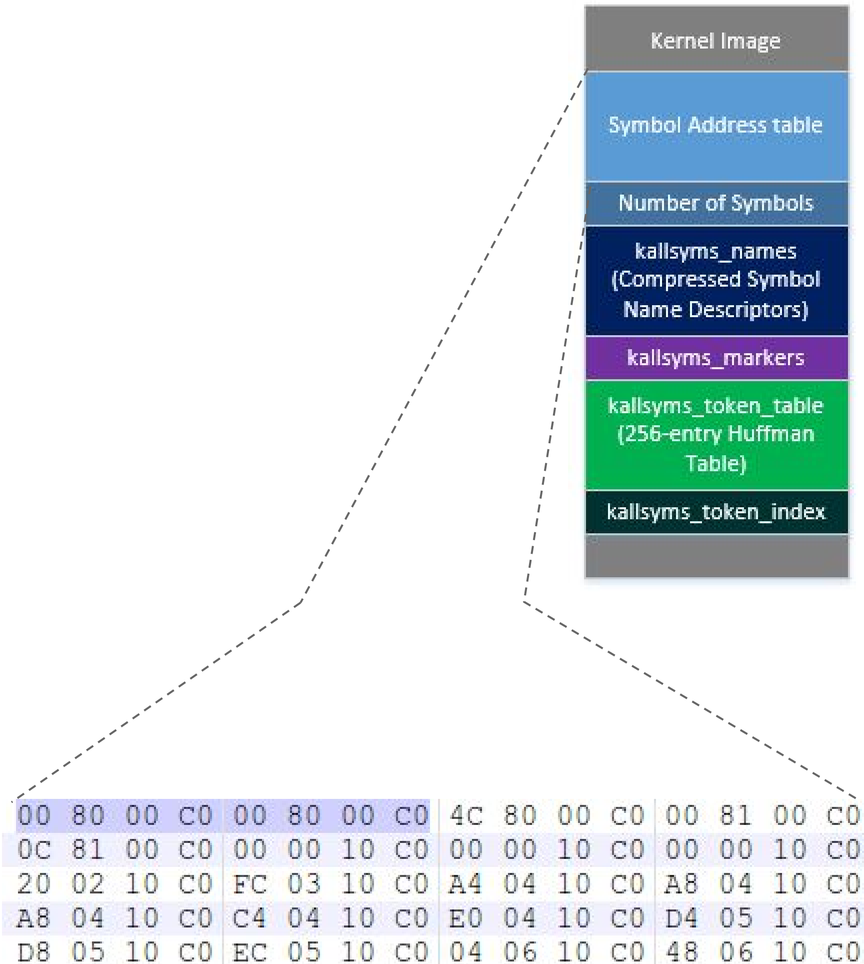
- Recall that some OEMs use TAs to provide "Normal-World" kernel attestation
    - e.g., TIMA PKM
- This implies TAs must have some way of acquiring or measuring memory in the "Non-Secure World"
- Digging deeper reveals that this functionality is provided by TEE OS syscalls
    - Any trustlet can request the TEE OS to map in any physical memory belonging to the "Non-Secure World" (with read-write permissions!)
    - As such, code-execution in any TA allows full code execution in the "Non-Secure World"

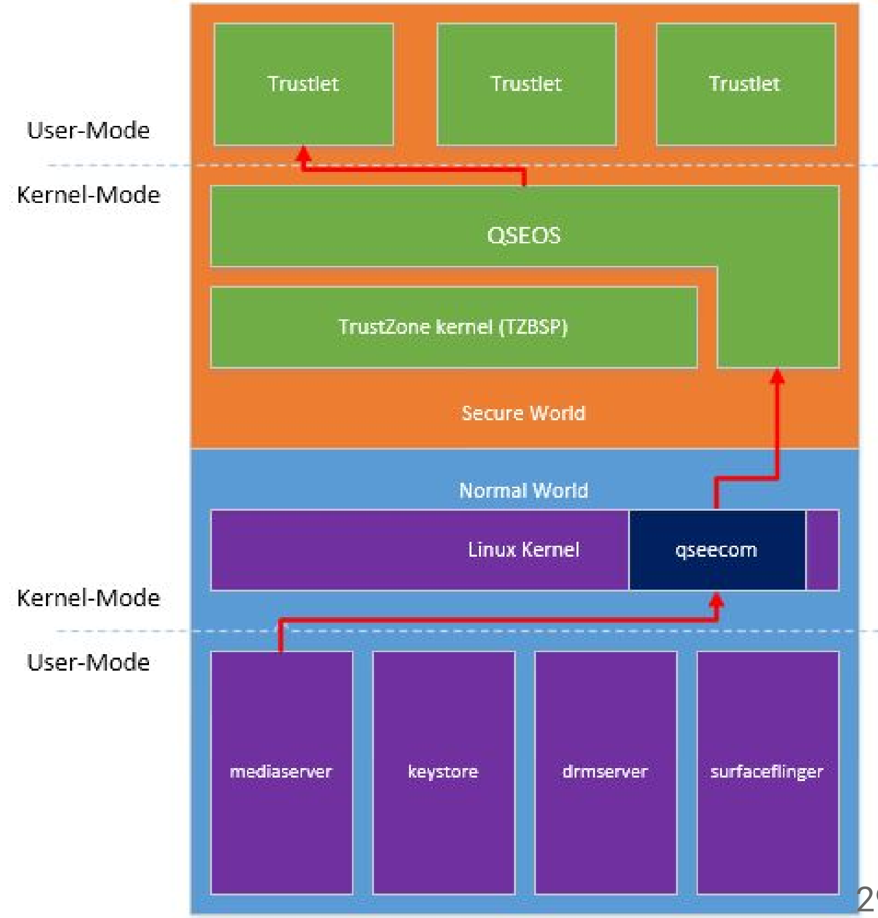


A storm in a TEEpot

```
signed int __fastcall qsee_register_shared_buffer(unsigned int buf_addr, int buf_len)
{

  //Validity checks to make sure there are no overflows, etc.
  <...SNIP...>

  //Checking for the specially allowed ranges in the "Secure World"
  if ( (is_ns_disallowed_range(buf_addr, buf_len) ||
        !is_ns_allowed_range(dword_FE824920, buf_addr, end_addr - 1))
    && !qsee_is_tag_area(1, buf_addr, buf_addr + buf_len)
    && !qsee_is_tag_area(2, buf_addr, buf_addr + buf_len)
    && !qsee_is_tag_area(3, buf_addr, buf_addr + buf_len)
    && !qsee_is_tag_area(4, buf_addr, buf_addr + buf_len)
    && !qsee_is_tag_area(6, buf_addr, buf_addr + buf_len)
    && !qsee_is_tag_area(5, buf_addr, buf_addr + buf_len) )
  {
    log(5, "{%x:%x %x}", -54, buf_addr, buf_len);
    return -1;
  }

  //Inserting the entry into the mapped buffers list
  <...SNIP...>

  //Mapping the buffer into QSEE!
  qsee_map_region(buf_addr, buf_addr, buf_len, 6, 32773, 1);
  return 0;
}
```

A storm in a TEEpot





TA Over Exposure

- Although TEEs lack modern mitigations, some vendors expose them to directly unprivileged users
    - This means any unprivileged attacker can attack the TEE
    - Successfully doing so results in bypassing all the protections enforced by Android
- For example, Samsung exposes many TAs with no required permissions
    - This is done by creating Android service which proxy arbitrary commands to TAs
    - ...Sadly, these TAs sometimes contain trivial memory corruptions
    - For example, to OTP TA was exposed to unprivileged attackers
        - https://bugs.chromium.org/p/project-zero/issues/detail?id=938
        - https://bugs.chromium.org/p/project-zero/issues/detail?id=939

Another High-Value Target - Android Full Disk Encryption

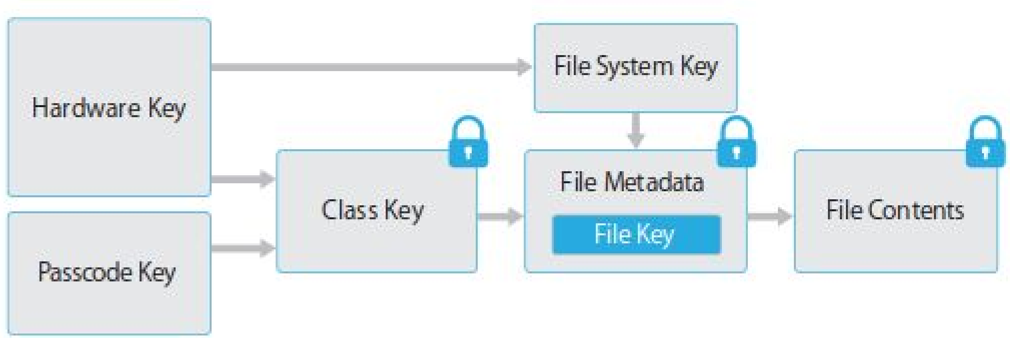
- Android supports encryption mechanisms in order to protect personal data,
  - Full Disk Encryption (FDE) has been enabled by default since Android 6.0
  - File Based Encryption (FBE) has been introduced in Android 7.0
- In the coming slides we'll see how Android's FDE scheme relies on the TEE
  - The underlying defects that we're about to see are still relevant for FBE
  - However, as the original research was done before the release of FBE, we'll focus on the FDE scheme instead

Full Disk Encryption - Real World Example

- Recall the case of Apple vs. FBI (the San Bernardino terrorist attack)
  - Sayed Farook's work phone was seized by the FBI after the terrorist attack
  - The FBI did not know the unlock passcode for the device
  - The device had full disk encryption enabled, preventing access to the stored data
- So why not just brute-force the passcode?
  - Mobile passphrases can be expected to be relatively weak (e.g., 4 digit PIN)
  - Let's assume that the FBI can also acquire the flash of the device
  - What's stopping them from brute-forcing off the device?

Apple’s Full Disk Encryption - Hardware Binding

- Apple defends against off-device brute forcing by “tangling” the key to the hardware
  - The iPhone FDE KDF is bound to a hardware 256-bit key, called the UID key
  - The UID key is randomly generated and fused in the factory
  - The UID key is not software or firmware accessible in any way (it can only be selected as the input key for the AES Crypto Engine)
  - For more information, see Apple’s security guide: https://www.apple.com/business/docs/iOS_Security_Guide.pdf



Apple's Full Disk Encryption - On-Device Brute Force Protections

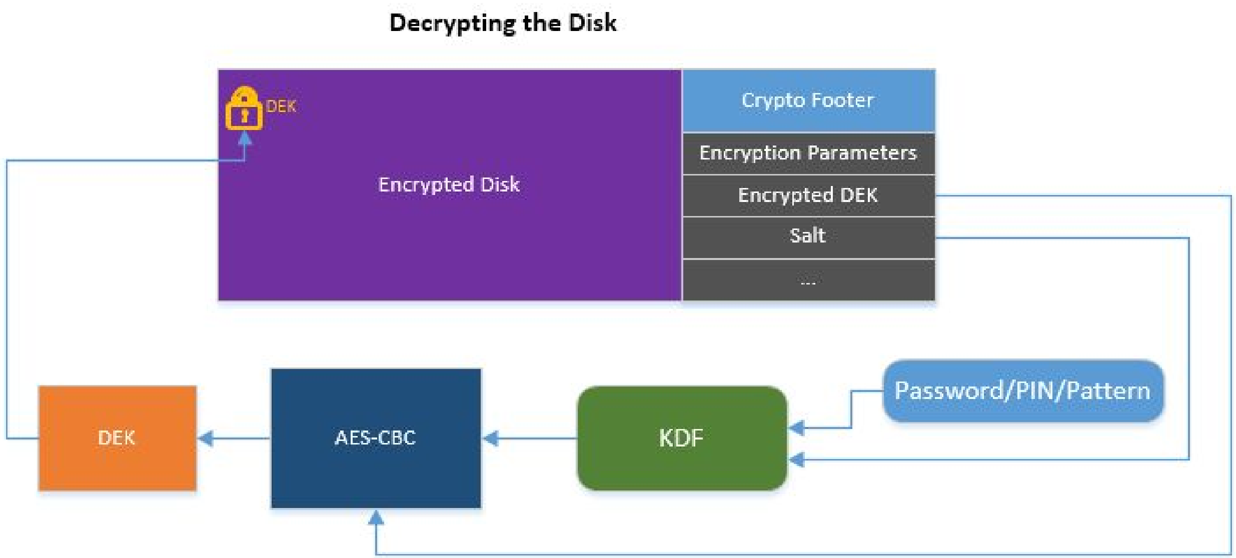
- Binding the KDF to the hardware implies brute force attacks must occur on-device
  - That is, barring hardware attacks or errors in cryptographic design
- So how can Apple dissuade on-device brute-force attacks?
  - The KDF is tuned to require ~80 milliseconds to execute on the device
    - This works out as ~2 weeks for a 4-character alphanumeric password
  - The software itself can introduce a maximal number of unlock attempts
    - However, software protections can more easily be subverted

Delays between passcode attempts

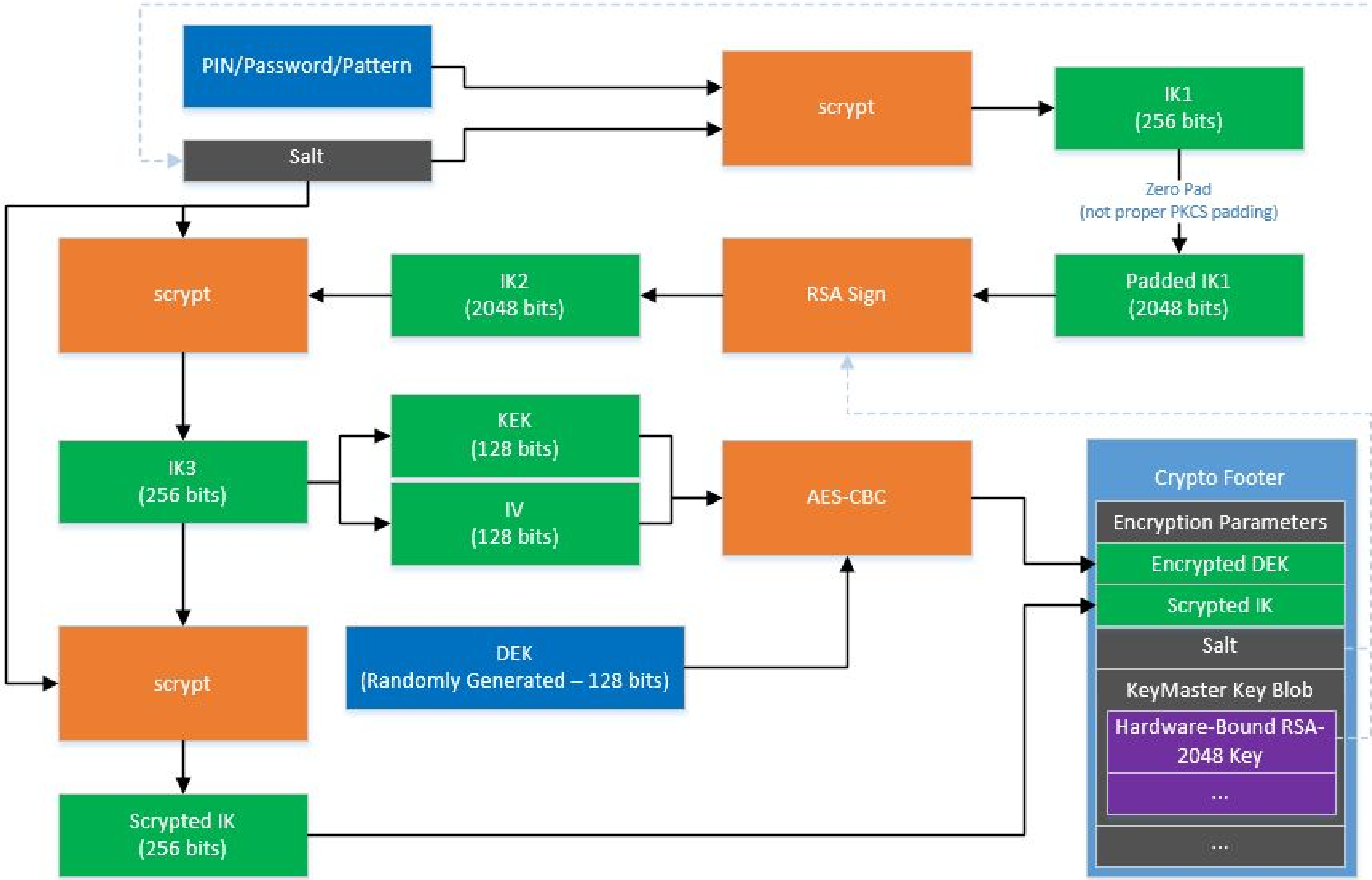
Attempts	Delay Enforced
1-4	none
5	1 minute
6	5 minutes
7-8	15 minutes
9	1 hour

Android's Full Disk Encryption

- The encryption scheme itself is based on the Linux Kernel Subsystem - dm-crypt
    - *dm-crypt* is a widely deployed and researched scheme
- However, the scheme itself still doesn't cover the actual FDE KDF
    - How is the encryption key generated and verified?
    - How does the hardware binding take place?

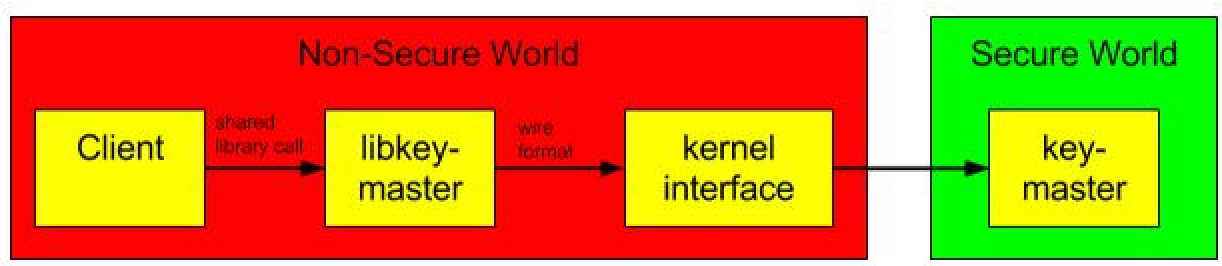


Android FDE KDF



Android's Full Disk Encryption - Hardware Binding

- The KeyMaster TA is used in order to "bind" the KDF to the hardware of the device
  - This is "comfortable" since KeyMaster key blobs are meant to be hardware bound
  - All devices with hardware-backed credentials storage support KeyMaster off-the-bat
- ...However...
  - *Specifying* hardware binding is one thing, but unlike Apple's FDE, there's no mention of the actual mechanism by which the binding takes place
  - The only way to make sure is to reverse-engineer the closed-source KeyMaster TA



Android's Full Disk Encryption - KeyMaster Trustlet

- Unfortunately the KeyMaster trustlet is proprietary
  - However, using the tools mentioned previously, we can load the TA in IDA
- The actual logic behind the KeyMaster TA is relatively simple
  - KeyMaster can generate a key blob, which is supposedly "hardware-bound"
  - KeyMaster may produce RSA signatures on user data using a supplied key blob
- So how does KeyMaster actually "use" the encapsulated key blob?
  - Either KeyMaster is somehow able to "decapsulate" the key
  - Or perhaps the key blob is used as a handle to retrieve the key from some cryptographic key storage (i.e., TPM)

```
struct  qcom_km_key_blob {
  uint32_t magic_num;
  uint32_t version_num;
  uint8_t  modulus[KM_KEY_SIZE_MAX];
  uint32_t modulus_size;
  uint8_t  public_exponent[KM_KEY_SIZE_MAX];
  uint32_t public_exponent_size;
  uint8_t  iv[KM_IV_LENGTH];
  uint8_t  encrypted_private_exponent[KM_KEY_SIZE_MAX];
  uint32_t encrypted_private_exponent_size;
  uint8_t  hmac[KM_HMAC_LENGTH];
};
typedef struct  qcom_km_key_blob qcom_km_key_blob_t;
```

Android's Full Disk Encryption - KeyMaster Trustlet

```
if ( key_blob->magic_num == 'KMKB' )
{
  buffer_0 = get_some_kind_of_buffer(0);
  if ( buffer_0 )
  {
    buffer_1 = get_some_kind_of_buffer(1);
    if ( buffer_1 )
    {
      res = qsee_hmac(2, (int)key_blob, 0x624, buffer_1, 32, (int)&hmac_result);
      if ( !res )
      {
        if ( timesafe_compare((int)&hmac_result, (int)key_blob->hmac, 0x20u) )
        {
          res = -20;
        }
        else
        {
          res = do_something_with_keyblob(key_blob, buffer_0, 16);
          if ( !res )
          {
            *(_DWORD *)output_size_ptr = output_len;
            res = sign_data_to_output(key_blob, data, datalen, output_ptr, output_size_ptr);
```

Android's Full Disk Encryption - KeyMaster Trustlet

```
int __fastcall do_something_with_keyblob(qcom_km_key_blob_t *keyblob, int key, int key_length)
{
  int result; // r0@1
  char mode; // [sp+8h] [bp-18h]@4
  int cipher; // [sp+Ch] [bp-14h]@1

  cipher = 0;
  qsee_cipher_init(0, (int)&cipher);
  if ( !result )
  {
    qsee_cipher_set_param(cipher, 0, key, key_length);// set key
    if ( !result )
    {
      qsee_cipher_set_param(cipher, 1, (int)keyblob->iv, 16);// set IV
      if ( !result )
      {
        mode = 1;
        qsee_cipher_set_param(cipher, 2, (int)&mode, 1);// set mode
        if ( !result )
        {
          result = qsee_cipher_decrypt(
                     cipher,
                     (int)keyblob->encrypted_private_exponent,
                     keyblob->encrypted_private_exponent_size,
                     (int)keyblob->encrypted_private_exponent,
                     (int)&keyblob->encrypted_private_exponent_size);
          if ( !result )
            cipher_destroy_probably(cipher);
        }
      }
    }
  }
  if ( result > 0 )
    result = -result;
  return result;
}
```

Android's Full Disk Encryption - KeyMaster Trustlet

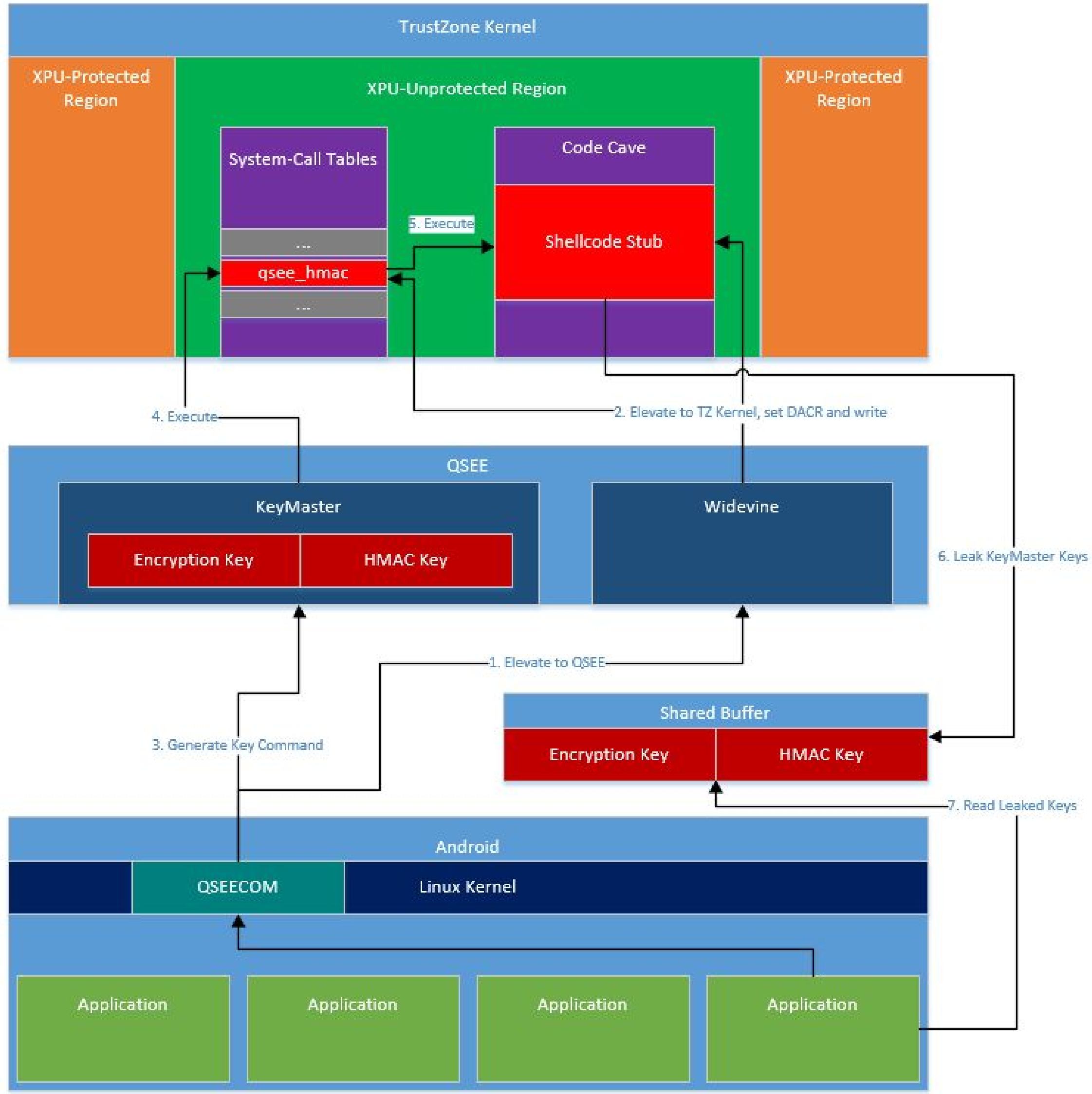
```
int __fastcall get_enc_key_or_hmac_key(int request_type)
{
  int global_buffer; // r9@0
  int res; // r4@1
  int _strlen1; // r5@4
  int _strlen2; // r0@4
  int strlen1; // r5@6
  int strlen2; // r0@6

  res = 0;
  if ( request_type )
  {
    if ( request_type == 1 )                    // HMAC key
    {
      strlen1 = strlen(global_buffer + 103);    // KM HMAC HW Crypto key derived from SHK
      strlen2 = strlen(global_buffer + 73);     // KM HMAC HW Crypto Derived key
      if ( !some_kind_of_kdf(0, 16, global_buffer + 73, strlen2, global_buffer + 103, strlen1, global_buffer + 224, 32) )
        res = global_buffer + 224;
    }
  }
  else                                          // Encryption Key
  {
    _strlen1 = strlen(global_buffer + 34);      // KM CPHR HW Crypto Derived key
    _strlen2 = strlen(global_buffer + 4);       // KM CPHR HW Crypto key derived from SHK
    if ( !some_kind_of_kdf(0, 16, global_buffer + 4, _strlen2, global_buffer + 34, _strlen1, global_buffer + 208, 16) )
      res = global_buffer + 208;
  }
  return res;
}
```

Android's Full Disk Encryption - Hardware Binding?

- As we've seen, the encryption key protecting the KeyMaster blobs is TEE-accessible
  - This means that gaining access to the TEE would allow us to leak the key
  - Once the key is leaked, the KDF is no longer hardware bound!
- The key is derived from a hardware-fused key (SHK) and a pair of constant strings
  - Therefore, once the key is leaked, it can no longer be modified!
  - Moreover, OEMs may be coerced into signing a TA which leaks the key (Apple vs. FBI)
  - Rolling back a device to a vulnerable version would allow an attacker to leak the key
    - This means that devices with no rollback prevention (e.g., Nexuses) may still be attacked using "patched" vulnerabilities

Breaking Android's Full Disk Encryption

- As we've seen, Android's FDE KDF hardware binding is only as strong as the TEE
  - We've also seen that QSEOS's trustlet isolation is weak
  - Moreover, the protection mechanisms for TAs in QSEE are insufficient
    - ~9 bit ASLR and a stack carved from the same segment as the BSS
- This means we simply need to:
  - Find a vulnerability in any TA
  - Break out of the TA into the TEE OS
  - Take over the KeyMaster TA from the TEE OS kernel
  - Leak the encryption key from the KeyMaster TA back to the "Non-Secure World"


TrustZone Kernel
XPU-Protected Region
XPU-Unprotected Region
XPU-Protected Region
System-Call Tables
...
qsee_hmac
...
Code Cave
Shellcode Stub
5. Execute
4. Execute
2. Elevate to TZ Kernel, set DACR and write
QSEE
KeyMaster
Encryption Key
HMAC Key
Widevine
6. Leak KeyMaster Keys
1. Elevate to QSEE
Shared Buffer
Encryption Key
HMAC Key
3. Generate Key Command
7. Read Leaked Keys
Android
QSEECOM
Linux Kernel
Application
Application
Application
Application

Breaking Android’s Full Disk Encryption

- Once the key is extracted, we can decrypt the KeyMaster key blob
  - The RSA private key in the key blob can be used to compute the “hardware-bound” step
  - This means the entire KDF can now be calculated off the device
- The Android FDE “crypto footer” also contains an “*scrypt*-ed intermediate key” field
  - This value is derived by applying scrypt to the result of the FDE KDF
  - An attacker may use this value to check the validity of each brute force attempt

```
[+] Enabled all domain permissions
[+] Writing shellcode to code cave
[+] Writing the function arguments
[+] Writing the function arguments
[+] Overwriting qsee_hmac function pointer
[+] Generated encrypted keypair blob!
-----------------------------------------------
[+] Leaked KeyMaster Keys!
[+] KeyMaster Key Encryption Key (KEK): 08DF57BED3F2396BACB6719444A308F2
[+] KeyMaster HMAC Key: C983041E84C04C1DFAA707763C31993D3FEA235AD54D7C2F3EE162CD380EEA30
[+] Widevine unload res: 0
```

Breaking Android's Full Disk Encryption

- If you want to play with the exploit, I've open sourced all the required parts
  - You can get the exploit chain to leak the KeyMaster keys here:
    - https://github.com/laginimaineb/ExtractKeyMaster
  - You can get the python script which to brute-force the FDE passphrase using the aforementioned leaked keys here:
    - https://github.com/laginimaineb/android_fde_bruteforce

```
$ python fde_bruteforce.py \
        metadata.bin \
        08DF57BED3F2396BACB6719444A308F2 \
        C983041E84C04C1DFAA707763C31993D3FEA235AD54D7C2F3EE162CD380EEA30 \
        wordlist.txt
[+] HMAC match!
[+] Key is valid!
[+] pow(pow(0x1337, e, N), d, N) == 0x1337
[+] Trying password: password
[+] Trying password: dadada
[+] Trying password: secret
------------------------------------------
[+] Found Full Disk Encryption Passphrase!
[+] Passphrase: secret
[+] Intermediate Key: 8dd12c8d9f1f9ead18873f0f7363f880ce65502baaca94a81b5af5bb6eb5d57e
------------------------------------------
```

Additional Thoughts on Android's FDE KDF

- The current implementation can't be considered "hardware-bound"
  - A software attack was enough to leak the encryption key and break the binding
- There are other flaws in the KDF design as well
  - For example, "raw" RSA is used to produce a signature of IK1
  - An attacker may provide an unpadded blob which would reveal the private key (without attacking the TEE at all!)
    - This has been fixed in newer versions of KeyMaster (v1) which support padded RSA
  - The RSA signature is not iterated in the KDF
    - Allows for a relatively fast brute force attack on the device (the "scrypt" calculations can be done off-device and pipelined)
    - Gatekeeper seeks to prevent such attacks, but can be subverted by attacking the TEE

Fixing Android’s FDE KDF

- Can the FDE KDF be fixed in current-gen hardware?
    - According the QC, the SHK can’t be used directly as it is used to generate TA secrets
    - Moreover, the SHK cannot be modified, as it is fused into the device’s hardware
- Perhaps we can think of a temporary “patch”?
    - There are hardware crypto engines (Qualcomm CE) which allow crypto operations using hardware fused keys
        - HMAC-SHA256 can be viewed as a PRF, replacing the RSA signature by the TEE
    - IK1 can be used as additional input to the SHK KDF
        - Provides binding between each passcode attempt and the derived encryption key
        - Relies on the “strength” of the SHK KDF, which is unknown

Android’s FDE KDF - Better Hardware Binding

- A more robust approach would be to allow actual hardware binding
  - Either by using a hardware-bound encryption key (ala Apple KDF)
  - Or by using a Secure Element in order to store the key or produce the signature
- The problem was (and remains) the fragmentation of Android devices
  - Many OEMs, many SoCs
  - The same KeyMaster design is currently used by all SoCs
  - Many devices (e.g., Samsung flagship phones) already have Secure Elements, which can be leveraged to achieve a higher level of security

TEEs as an Attack Surface

- We've seen some fortifications of TEEs which aim to make them more secure
  - ~~Memory Isolation~~
  - ~~Exploit mitigations to safeguard trustlets from attacks~~
  - ~~Trustlets are isolated from one another the the TEE OS~~
  - ~~The TEE is a small TCB, which should be easier to verify than a "rich" OS~~
- In reality, TEEs offer great attack surface!
  - Getting code execution in the TEE allows full control over the "Non-Secure" world
  - The TEEs is "weaker" than the HLOS
  - More and more OEMs are exposing large portions of the TEE to non-privileged users in the "Non-Secure World"

Conclusions

- TEEs have nearly no exploit mitigations, making them an easy target
  - Either no ASLR or insufficient entropy
  - Lack of stack cookie (MobiCore)
  - Lack of stack guard page, stack placed after BSS (QSEE)
- TEEs don't follow the principle of least-privilege, making them a valuable target
  - Huge (and expanding) TCB
  - Direct control over "Non-Secure World" memory
- The only way to guarantee the safety of TEEs is to audit them
  - The current situation leaves a huge proprietary code base in charge
  - If TEEs cannot be open-sourced, they should at least by audited by OEMs

Q & A