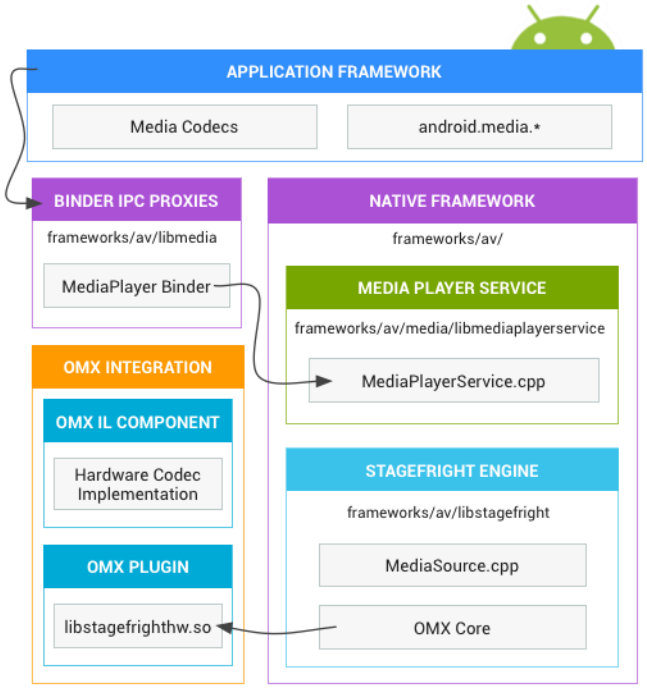
Bypassing the Maginot Line: Remotely Exploit the Hardware Decoder on Smartphone

Xiling Gong of Tencent Blade Team

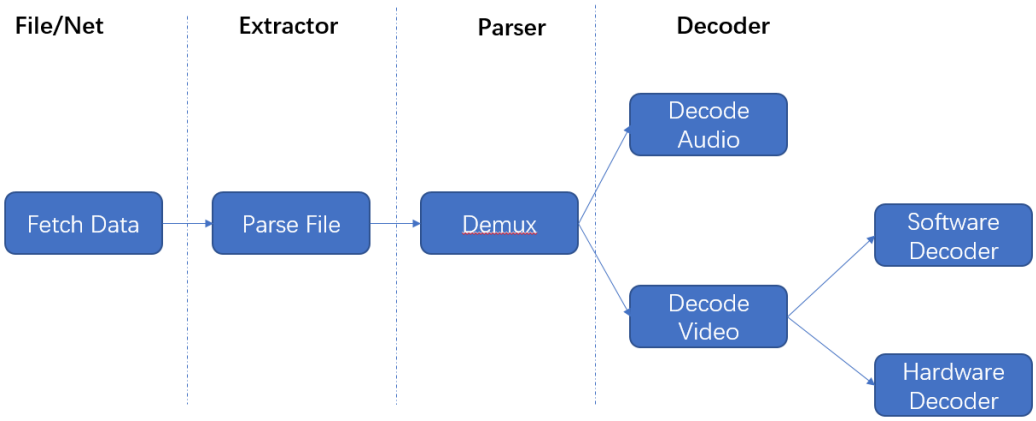
Bypassing the Maginot Line: Remotely Exploit the Hardware Decoder on Smartphone 1
1 Background.....2
2 Qualcomm Venus.....3
3 Debug Venus.....5
4 Reverse Engineering.....6
4.1 OMX Component and Linux Driver.....6
4.2 Venus Firmware.....7
4.2.1 Firmware & Memory Layout.....7
4.2.2 Registers.....8
4.2.3 Firmware Module.....9
5 Vulnerability and Exploitation.....9
5.1 The Vulnerability(CVE-2019-2256).....9
5.2 Exploitation.....10
6 Conclusions and Future Works.....12

1 Background

Let's start with the media architecture of Android platform [1].



From the figure, we can see that, generally when we play a media file on Android, the application will simply call API provided by Application Framework. The framework will then call the MediaPlayerService through IPC. In the MediaPlayerService, the Stagefright Engine will handle all the necessary proces. The whole procedure is quite complex from the source code. I've drew a simplified figure to explain the process.



From the figure, we can see that, there are mainly three components in the Stagefright, that is Extractor, Demuxer, Decoder.

Extractor - The extractor will parse the media file, determinate the file format and extract media data (audio and video data). Other information in the file, such as timing, progress, if interesting and available, the extractor will also handle them well.

Demuxer - In the media file, the Audio and Video data are mixed together and the structure differs from file types. So, we need a Demuxer to separate the audio and video data from the mixed structure.

Decoder – Now the Decoder could receive the pure video or audio data and start decoding. The decoder is an OMX decoder components, which implement the interfaces defined by OMX and the decoder procedure defined by the video or audio format specification. The real implementation of the decoder procedure varies with the vendor.

For example. The available decoders are in the file

/vendor/etc/media_codecs.xml.

You can see from the file, for type=”video/avc”, there are two MediaCodec

<MediaCodec name="OMX.google.h264.decoder" type="video/avc">

<MediaCodec name="OMX.qcom.video.decoder.hevc" type="video/hevc" >

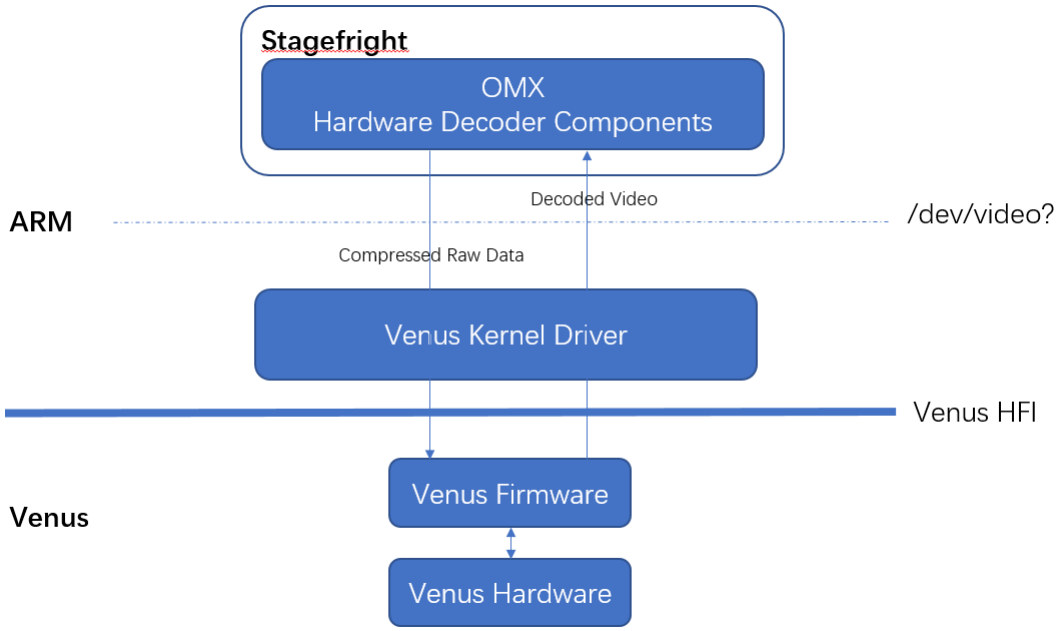
That represent two different implementation of the H264 decoder. One is the software implementation by Google in Android Open Source Project (AOSP) [2]. And one is the hardware implementation by Qualcomm.

For the software implementation in the AOSP, you must be familiar with it, because there are lots of vulnerabilities in the Android Security Bulletin every month, well-known as the Stagefright vulnerabilities [3]. Then how about the hardware decoder? This is talk will discuss about it.

2 Qualcomm Venus

2.1 Overall Architecture of Venus

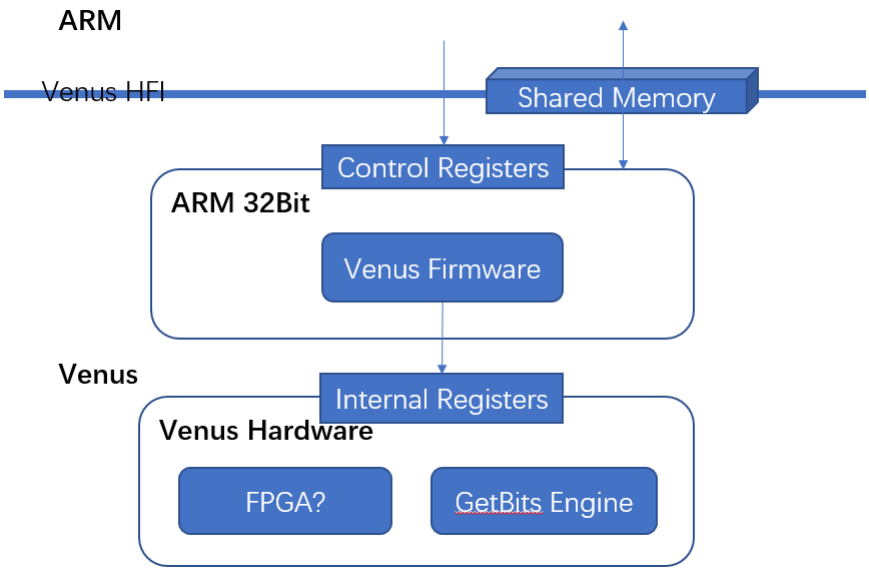
Qualcomm Venus is the dedicated video hardware decoder. I’ve drew a simple architecture of the Venus Hardware Decoder.



From the figure, we can see that the OMX Hardware Decoder Components in the Stagefright will fill the compressed raw data from the video source into Venus. Then Venus

will return the decoded video back to the OMX Hardware Decoder Components. Venus is a dedicated chip separated with the ARM main chip. So there are a Linux Kernel Driver and a inter-processor interface to communicate between Venus and ARM. There are shared memory, and also a simple communication protocol between Venus and ARM.

Now let's dive into Venus itself.

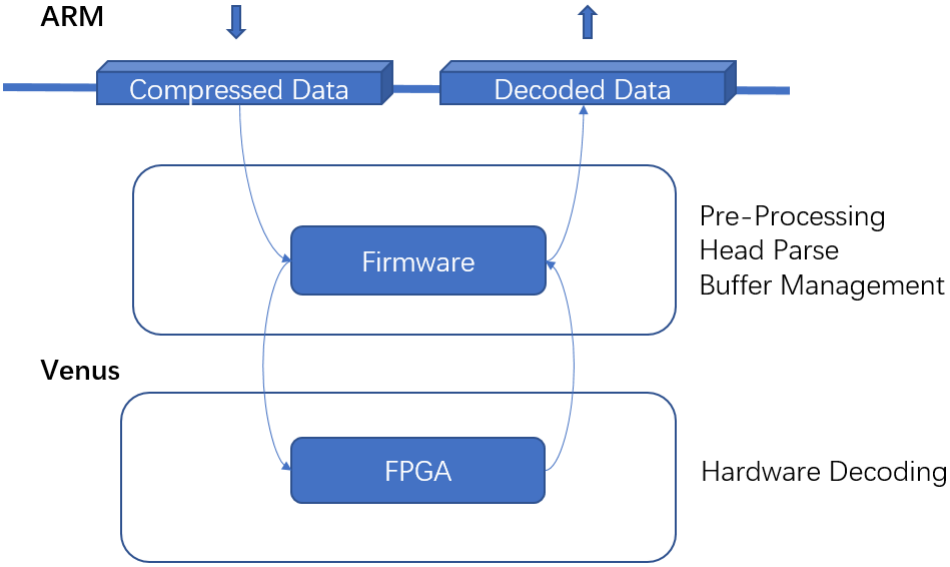


From the figure we can see that, Venus has three main components.

ARM 32Bits - There is also a small arm chip inside Venus, which is 32 bits and used to run the Venus Firmware. We can get the firmware from factory image and reverse engineering it.

Venus Hardware - This hardware is the real hardware, not software. It is something like the FPGA, which is used to decode the compressed video data. Using the FPGA, Venus can accelerate the processing and decrease the power consuming I think. Besides, there is a hardware named GetBits Engine (I named it) used to accelerate the bit operating. Using the GetBits Engine, programmer is able to easily get bits from the compressed video stream.

Registers – There are two groups of registers I think. One is used by the Linux Driver to control Venus and the other is used by Venus firmware to control the hardware (FPGA and GetBits Engine).



Generally, the procedure of Venus decoding video data includes three steps

Step 1 - Stagefright and Linux Driver will fill the raw video data into buffer and send to Venus.

Step 2 - Venus Firmware will receive the buffer and do some pre-processing (mainly parse the head of different types I think) to get some more *pure* data that hardware could handle it easier. Then Venus Firmware will command the FPGA to process the pure data.

Step 3 - Finally the FPGA will decode the video data and fill the decoded data into buffer. The decoded buffer will then be send to Linux by Venus Firmware.

3 Debug Venus

Before we go, let's prepare the debugger for Venus.

As we know, the peripheral firmware such as Modem, WLAN, Venus, is protected by the Secure Boot[4] and unable to be modified. That is, if you modify the Venus firmware, the TrustZone will detect it and refuse to load it into Venus memory.

Generally, there are three methods to debug Venus

Option A - Find a vulnerability in the Secure Boot. The vulnerability may be in boot stage. This is not an easy task. A more practical method is, there are some known issues in the TrustZone, which could be used to compromise the TrustZone in an older version. So utilize such a vulnerability and then disable Secure Boot in TrustZone.

Option B - Find a vulnerability in Venus and compromise Venus locally from Linux and setup the debugger. This is not as difficult as Option A, though still not very easy. When I exploit the Qualcomm WLAN this year, I'm using this method. So this method should also be OK.

Option C - Buy a development board. A Snapdragon development board could help you a lot. No secure boot, no firmware verification, JTAG interface on, etc.

Here we would like to propose another method we are using,

Option D - to buy a phone that already disabled Secure Boot. For product mode, that is the phone we buy from market, the Secure Boot should always be turn on to protect the device. But we are quite lucky, the Pixel we bought is already Secure Boot disabled. So we can directly modify the firmware and load into Venus. Some Chinese vendors may sell phones with Secure Boot disabled (though I'm not familiar with these model), I think you can try.

Since we can modify Venus firmware freely, setup the debugger is not a difficult task. Let's start the reverse engineering.

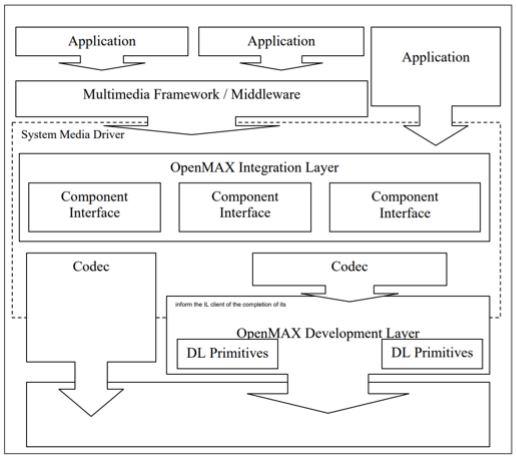
4 Reverse Engineering

Let's do some reverse engineering work for Venus.

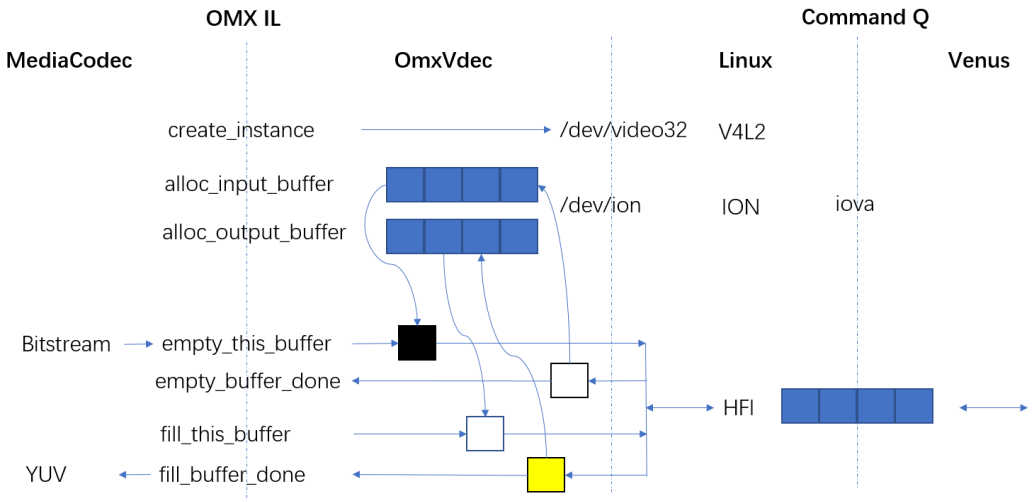
4.1 OMX Component and Linux Driver

Before looking into Venus, let's get a deep comprehension about the Linux side component of Venus, that is the OMX Component and Linux Driver. Understanding the Linux side will help us better understand the Venus firmware side.

The OMX is actually a complex architecture [5] as the following figure.



The important thing we need to know is that, the library libqomx_core.so implements the OMX interface, and libOmxVdec.so is the worker who do the job with Kernel driver. Here is a simple illustration about the OmxVdec.



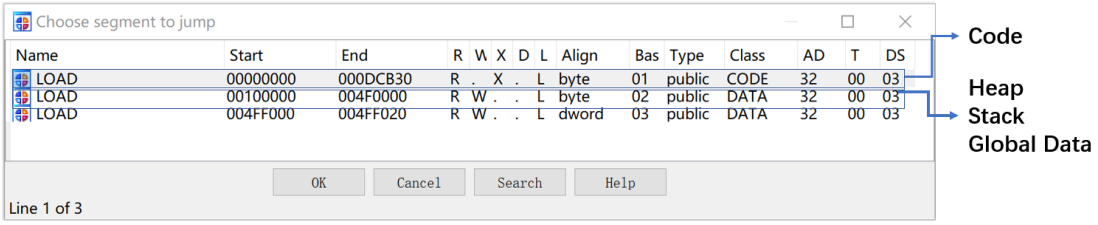
We can see that, OmxVdec will send the input buffer to Kernel driver and receive decoded buffer from Kernel driver. OmxVdec exchanged data with Venus through ION(/dev/ion) to improve the performace. Control message is send through ioctl(/dev/video32) into Kernel and then Kernel relay the command by HFI into Venus.

The important thing in the figure is, in OMX, empty_this_buffer means decode this buffer and fill_this_buffer means fill the decoded output into this buffer. Follow the CommandQ and the data buffer operation into Venus, we can then understand the code flow in Venus.

4.2 Venus Firmware

4.2.1 Firmware & Memory Layout

Let's take a look at the firmware and memory layout of Venus.



Static	E0000000	E00FF000	Register Area
Dynamic	70800000	708F0000	Shared Memory (Message Queue)
Dynamic	70A00000	···	Shared Memory (Input Buffers)
Dynamic	70A00000	···	Shared Memory (Output Buffers)

From the figure, we can see that, the memory of Venus consists of 4 parts

Code - The core of Venus is an ARM chip. Drag the firmware of Venus into IDA, we can directly disassemble it. The first segment is the code segment we'll reverse it later.

Data - The second segment is the data segment. The heap, stack, global data are all in this segment.

Registers - The registers are in the virtual address area 0xE0000000~0xE00FF000.

Dynamic Memory – This memory area is the shared memory area between Linux and Venus. The address of the Message Queue is fixed from 0x70800000 to 0x708F0000. But the input buffers and output buffers seems not fixed, which are mapped into Venus dynamically from 0x70A00000.

4.2.2 Registers

I've to admit that most of the function of the registers are unknown. But I've still managed to find the usage of some registers, which are quite useful.

Control Registers - These registers are used by Linux to control Venus, or used by Venus to share status with Linux. The usage of these registers is documented in vidc_hfi_io.h.

```
#define VIDC_CPU_CS_A2HSOFTINT      (VIDC CPU CS BASE OFFS + 0x18)
#define VIDC_QTBL_ADDR 0x000D2054
```

GetBits Registers - Maybe named CCE internally. These registers are used to control the GetBits engine by Venus. What's GetBits? The specification of video stream contains lots of bit operation. When parsing the stream, the bit count to fetch and the data type vary a lot. One bit, two bits, one byte, unsigned int, signed int, for programmers it's annoying! The GetBits Engine could help this out. Just simply setup the data pointer and length, then set the bits want to fetch, then we'll get the expected result and the read pointer will advance automatically.

The following figure is used to setup the data and length registers.

```
MEMORY[0xE0032014] = 0xC00;
MEMORY[0xE0032030] = 0;
MEMORY[0xE0032034] = 0;
MEMORY[0xE003F240] = v99;                   // CCE Programming : address
MEMORY[0xE003201C] = decoderInstance->length;// filled_len
MEMORY[0xE0032010] = v4;                    // alloc_len
MEMORY[0xE0032018] = v88 - v99;             // (buffer + offset) - buffer
MEMORY[0xE0032020] = 0;
MEMORY[0xE003207C] = 0x73FFF357;
```

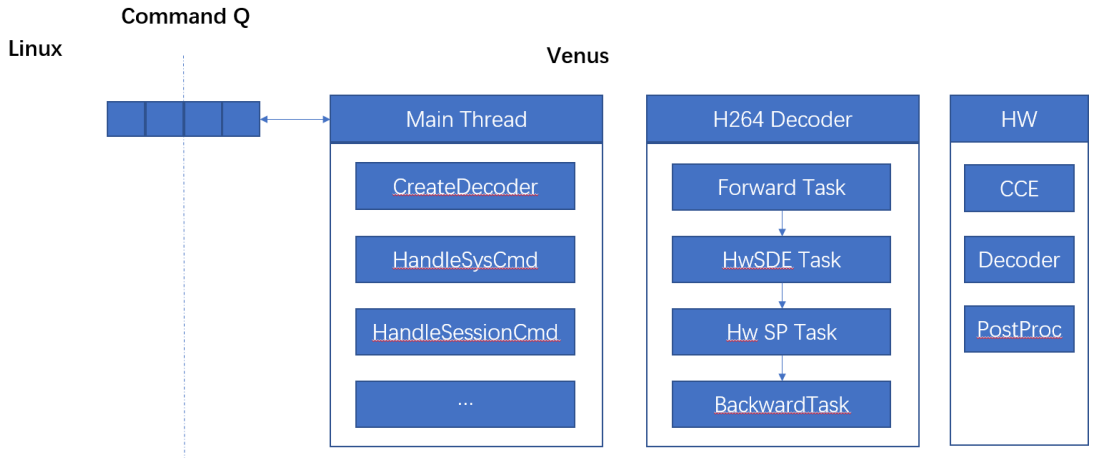
And the following figure using 0xE0032000 to tell the CCE the data type and length wanted. And then read 0xE0032078 to check the result.

```
CCE_Programming(v7, v6, 0, v8);
MEMORY[0xE0032000] = 0x13000000;
while ( !(MEMORY[0xE0032078] & 0x10000401) )
  ;
Log(1, "Done waiting for CCE bits");
v9 = (MEMORY[0xE0032004] & 0x80000000) == 0;
if ( MEMORY[0xE0032004] & 0x80000000 )
  v9 = (MEMORY[0xE0032078] & 0x10000001) == 0;
if ( !v9 )
{
  v10 = 1715;
  v11 = 889791;
  v12 = "%s(%d): error in parsing CCE_DEC_RESULT:\n";
  v13 = 8;
```

This is very useful to understand the code flow of a decoder.

4.2.3 Firmware Module

Now let's dive into the Venus firmware itself.



The firmware mainly consists of these parts

The Main Thread - The Main Thread is used to receive commands from Linux and process the commands. For example, CreateDecoder command to start a new decode session, BufferCommand to decode the compressed video data. You can find the definition of the commands in hfi.h in Linux kernel driver.

Decoders - Each decoder session creates a decoder. Venus supports up to 16 sessions simultaneously. The format Venus support is defined in the media_codecs.xml. Each decoder is a standalone thread, which communicate with the Main Thread through message queue.

Generally, the decoder consists of 4 tasks, the Forward Task, the HwSDE Task, the HwSP Task, and the Backward Task.

The Forward Task, as I know, is used to receive buffer from the Main Thread and do some preprocessing work, such as parse head, extract valid data, etc. The Forward Task is the main attack surface I think.

The HwSDE/HwSP Task, then the Forward Task will pass the *pure* data to HwSDE for further process, maybe hardware decodes using the FPGA.

Backward Task, and the Backward Task is used to send decoded buffer back to Linux.

OK, that's all we need to know about Venus.

5 Vulnerability and Exploitation

5.1 The Vulnerability(CVE-2019-2256)

OK, now we've got some basic sense about Venus. Let's find and exploit the vulnerability.

The vulnerability is CVE-2019-2256 which announced in Android Security Bulletin May 2019[7]. The issue occurs when Venus decode the Sequence Parameter Set(SPS) of H264. Venus doesn't check the parameter number_views and copy as much of

num_non_anchor_refs as possible. So, if we provide a large number of num_non_anchor_refs, then buffer overflow occurs.

Let's check the code in sub_64E08, where the overflow occurs,

```
if ( MEMORY[0xE0032004] )
{
  MEMORY[0xE0032000] = 0x1000000;             // num_views_minus1
  v5 = MEMORY[0xE0032004] + 1;
  v17 = MEMORY[0xE0032004] - 1 < 0;
  *(_DWORD *)(a2 + 1212) = MEMORY[0xE0032004] + 1;
  if ( (unsigned __int8)(v17 ^ __OFSUB__(v5, 2)) | (v5 == 2) )
  {
    v6 = 0;
    while ( *(_DWORD *)(a2 + 1212) > v6 )     // view_id
    {
      MEMORY[0xE0032000] = 0x1000000;
      v7 = a2 + 2 * v6++ + 1024;
      *(_WORD *)(v7 + 192) = MEMORY[0xE0032004];
    }
    v8 = (_WORD *)(a2 + 1024);
```

The issue is quite straightforward, a2+1212 is read from the bitstream (GetBits Engine 0xE0032004) with type unsigned integer, that is the num_views_minus1. And then will read num_views_minus1 of unsigned integer from the bitstream into v7+192, which causes the heap buffer overflow.

The following figure is the simple PoC used to trigger this issue.

```
int num_views_minus1 = 1;
set_ue_golomb(&pb, num_views_minus1);

for (int i = 0; i <= num_views_minus1; i++) {
    set_ue_golomb(&pb, i);
}

for (int i = 1; i <= num_views_minus1; i++) {
    set_ue_golomb(&pb, 0);
    set_ue_golomb(&pb, 0);
}

for (int i = 1; i <= num_views_minus1; i++) {
    set_ue_golomb(&pb, 0);
    set_ue_golomb(&pb, xxx_size / 2);
    for (int j = 0; j < xxx_size; j += 2) {
        unsigned int c = xxx[j];

        c = c + (xxx[j + 1] << 8);
        set_ue_golomb_long(&pb, c);
        addr += 2;
    }
}
```

Please note, construct a valid H264 file and encapsulate it into a mpeg4 from zero is an annoying task. The PoC is written by modify the AVC encoder of the ffmepg[6], that is quite simpler.

5.2 Exploitation

The exploitation is quite simple, since the only mitigations in Venus are the W^X and stack cookie, there are no ALSR, no heap protection.

Mitigation	Status
Heap ASLR	N
Heap Cookie	N
Stack Cookie	Y
Code & Global Data ASLR	N
W^X	Y
CFI	N

For this heap overflow issue, we can simply overflow to a meaningful address and overwrite the content.

Quite lucky, for each decode session, while create the decoder, the decoder instance object is allocated from the heap, and then the decoder function is filled into the instance (so that the decoder task could call into different decoder functions for different file format).

That's quite convenient for us if we could overwrite this object, the decoderInstance, we can control the PC and moreover, when calling the functions, the register R0 contains the pointer of the decoderInstance.

```
decoderInstance = (H264DecodeInstance *)DALSYS_Malloc(7232);
decoderInstance_1 = decoderInstance;
if ( !decoderInstance )
{
  Log(8, "%s(%d): No memory to create the decoder instance.\n", 857513, 568, decoderInstance_1);
  return 0;
}
memset(decoderInstance, 0x1C40u);
```

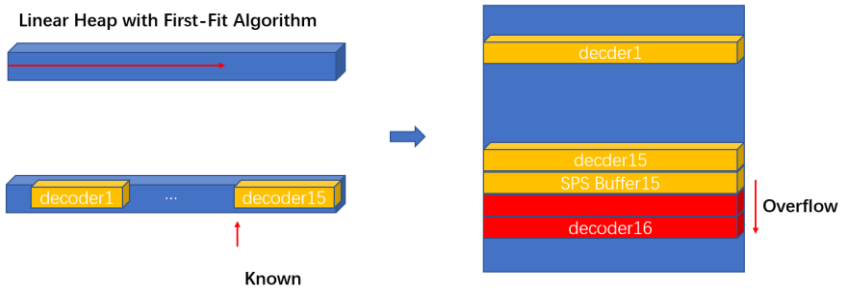
…

```
sub_4082C((_DWORD *)v10 + 13, (int)decoderInstance_1, (int)h264Dec_ProcessInput_0);
sub_4082C((_DWORD *)v10 + 15, (int)decoderInstance_1, (int)h264BackwardHandler);
sub_4082C((_DWORD *)v10 + 17, (int)decoderInstance_1, (int)h264HwSpTask);
sub_4082C((_DWORD *)v10 + 19, (int)decoderInstance_1, (int)h264HwSdeTask);
```

The problem we encounter is that, how to make the overwrite stably? That means,

- The overflow address is stable and predictable
- The target object (decderInstance) address is also stable and predictable

Quite lucky, the heap of Venus is not quite complex. It's a simple linear heap which allocate from the beginning to the end using the Fist-Fit-Algorithm (I guess). We can simply spray lots of decoder session and related variant to make everything predictable. In my experiments, spray about 5 sessions is enough to make the address stable. To make things perfect, in my final exploitation, I've sprayed 16 sessions, that is the maximum decoder count Venus supports. Here is a simple figure to illustrate the heap spray.



Now we've controlled the PC and R0. We can run useful ROP. In my exploitation, I'm simply using the ROP to print out a debug log.

6 Conclusions and Future Works

So now we have remotely compromised Venus. It's a long way. From the top of the Android framework to the Linux Kernel and finally into Venus. Although there are lots of hardening and mitigations in the media framework and Linux Kernel, we've now bypassed then directly. Thus, expose the Linux Kernel to a huge risk!

And what's next? The immediate question is, can we break into Linux from Venus? It's a very interesting question. And another question is how about the other file formats and other vendors? Also, according to the risk, we have to consider how to improve the security status of the hardware decoder. For vendors, there are lots of work could improve the security status. Hope more researchers could look into this area.

References

[1] https://source.android.com/devices/media/

[2]https://android.googlesource.com/platform/frameworks/av/+/refs/heads/master/media/libstagefright/codecs/avcdec/

[3] https://www.blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf

[4] https://www.qualcomm.com/media/documents/files/secure-boot-and-image-authentication-technical-overview.pdf

[5] https://www.khronos.org/openmax/

[6] https://ffmpeg.org/

[7] https://source.android.com/security/bulletin/2019-05-01