A 0-click exploit chain for the Pixel 9 Part 1: Decoding Dolby

2026-JAN-14 Natalie Silvanovich

Over the past few years, several AI-powered features have been added to mobile phones that allow users to better search and understand their messages. One effect of this change is increased 0-click attack surface, as efficient analysis often requires message media to be decoded before the message is opened by the user. One such feature is audio transcription. Incoming SMS and RCS audio attachments received by Google Messages are now automatically decoded with no user interaction. As a result, audio decoders are now in the 0-click attack surface of most Android phones.

I've spent a fair bit of time investigating these decoders, first reporting CVE-2025-49415 (https://project-zero.issues.chromium.org/issues/368695689) in the Monkey's Audio (https://www.monkeysaudio.com/) codec on Samsung devices. Based on this research, the team reviewed the Dolby Unified Decoder, and Ivan Fratric and I reported CVE-2025-54957 (https://project-zero.issues.chromium.org/issues/428075495) . This vulnerability is likely in the 0-click attack surface of most Android devices in use today. In parallel, Seth Jenkins investigated a driver accessible from the sandbox the decoder runs in on a Pixel 9, and reported CVE-2025-36934.

As I've shared this research, vendors as well as members of the security community have questioned whether such vulnerabilities are exploitable, as well as whether 0-click exploits are possible for all but the most well-resourced attackers in the modern Android Security environment. We were also asked whether code execution in the context of a media decoder is practically useful to an attacker and how platforms can reduce the risks such a capability presents to users.

To answer these questions, Project Zero wrote a 0-click exploit chain targeting the Pixel 9. We hope this research will help defenders better understand how these attacks work in the wild, the strengths and weaknesses of Android's security features with regards to preventing such attacks, and the importance of remediating media and driver vulnerabilities on mobile devices.

The exploit will be detailed in three blog posts.

Part 1 of this series will describe how we exploited CVE-2025-54957 (https://project-zero.issues.chromium.org/issues/428075495) to gain arbitrary code execution in the mediacodec context of a Google Pixel 9.

Part 2 of this series will describe how we exploited CVE-2025-36934 (https://project-zero.issues.chromium.org/issues/426567975) to escalate privileges from mediacodec to kernel on this device.

Part 3 will discuss lessons learned and recommendations for preventing similar exploits on mobile devices.

The vulnerabilities discussed in these posts were fixed as of January 5, 2026.

The Dolby Unified Decoder

The Dolby Unified Decoder component (UDC) is a library that provides support for the Dolby Digital (DD) and Dolby Digital Plus (DD+) audio formats. These formats are also known as AC-3 and EAC-3 respectively. A public specification (https://www.etsi.org/deliver/etsi_ts/102300_102399/102366/01.04.01_60/ts_102366v010401p.pdf) is available for these formats. The UDC is integrated into a variety of hardware and platforms, including Android, iOS, Windows and media streaming devices. It is shipped to most OEMs as a binary 'blob' with limited symbols, which is then statically linked into a shared library. On the Pixel 9, the UDC is integrated into `/vendor/lib64/libcodec2_soft_ddpdec.so`.

The Bug

DD+ audio is processed from a bitstream, which consists of independently decodable syncframes, each representing a series of audio samples. During normal operation, the UDC consecutively decodes each syncframe from the bitstream.

One element of a syncframe is the audio block which, according to the specification, can contain the following fields. A syncframe can contain up to 6 audio blocks.

Syntax	Number of bits
`skiple`	1
`if(skiple)`	
`skipl`	9
`skipfld`	9 * 8
`}`	

This means the decoder can copy up to 0x1FF (`skipl`) bytes per audio block from the bitstream into a buffer we'll call the 'skip buffer'.

The skip buffer contains data in a format called Extensible Metadata Delivery Format (EMDF). This format is synchronized, meaning that the UDC looks for a specific series of bytes in the skip buffer, then processes the data afterwards as EMDF. The EMDF in a single syncframe is called an 'EMDF container'. This is represented in the specifications as:

Syntax	Number of bits
`emdf_sync(){`	
`syncword`	16
`emdf_container_length`	16

Syntax	Number of bits
}	

The EMDF syncword is ‘X8’.

An EMDF container is defined as follows:

Syntax	Number of bits
emdf_container() {	
emdf_version	2
if (emdf_version == 3) {	
emdf_version += variable_bits(2)	
}	
key_id	3
if (key_id == 7) {	
key_id += variable_bits(3)	
}	
while (**emdf_payload_id** != 0x0) {	5
if (emdf_payload_id == 0x1F) {	
emdf_payload_id += variable_bits(5)	
}	
}	
emdf_payload_config()	
emdf_payload_size	variable_bits(8)
for (i = 0; i < payload_size; i++) {	
emdf_payload_byte	8
}	
emdf_protection()	
}	

variable_bits is defined as:

Syntax	Number of bits
variable_bits (n_bits) {	
value = 0;	
do {	
value += **read**	n_bits

Syntax	Number of bits
`read_more`	1
`if (read_more) {`	
`value <<= n_bits;`	
`value += (1<<n_bits);`	
`}`	
`}`	
`while (read_more);`	
`return value`	
`}`	

If you've spent time looking for vulnerabilities in this type of specification, a problem might already be apparent. There is no stated limit for the size of `emdf_payload_size`, meanwhile the output of `variable_bits` could be very large, essentially any numeric value.

Indeed, this is the root of the problem Ivan Fratric found while analyzing the Android UDC binary. In pseudo-code, it reads the EMDF payload into a custom 'evo' heap as follows:

```
result = read_variable_bits(this, 8, &payload_length);
if ( !result )
{
  if ( evo_heap )
  {
    buffer = ddp_udc_int_evo_malloc(evo_heap, payload_length, param.extra_len);
    outstruct.buf = buffer;
    if ( !buffer )
      return 2;
    if ( payload_length )
    {
      index = 0;
      while ( !ddp_udc_int_evo_brw_read(this, 8, &byte_read) )
      {
        outstruct.buf[index++] = byte_read;
        if ( index >= payload_length )
          goto ERROR;
      }
      return 10;
    }
  }
```

So, memory is allocated, then the bytes of the payload are copied into the allocated memory. How does this allocation work?

```
void ddp_udc_int_evo_malloc(heap *h, size_t alloc_size, size_t extra)
{
  size_t total_size;
  unsigned __int8 *mem;

  total_size = alloc_size + extra;
  if ( alloc_size + extra < alloc_size )
    return 0;
  if ( total_size % 8 )
    total_size += (8 - total_size) % total_size;
  if ( total_size > heap->remaining )
```

```
    return 0;
  mem = heap->curr_mem;
  heap->remaining -= total_size;
  heap->curr_mem += total_size;
  return mem;
}
```

The evo heap is a single slab, with a single tracking pointer that is incremented when memory is allocated. There is no way to free memory on the evo heap. It is only used to process EMDF payloads for a single syncframe (the specification provides no limit on the number of payloads a syncframe can contain, outside of limits on the size of the skip buffer), and once that frame is processed, the entire evo heap is cleared and re-used for the next frame, with no persistence between syncframes.

While `evo_malloc` performs a fair number of length checks on allocations, this check is flawed, as it lacks an integer overflow check:

```
if ( total_size % 8 )
   total_size += (8 - total_size) % total_size;
```

If total allocation size on a 64-bit platform is between 0xFFFFFFFFFFFFFFF9 and 0xFFFFFFFFFFFFFFFF, the value of `total_size` will wrap, leading to a small allocation, meanwhile, the loop that writes to the buffer uses the original `payload_length` as its bounds.

Integer overflow bugs are often challenging to exploit because they perform very large writes, but this code has a feature that makes this not the case. Each byte that is written is read from the skip buffer using `ddp_udc_int_evo_brw_read`, and that function checks read bounds based on `emdf_container_length`, which is also read from the skip buffer. If the read bounds check fails, the loop exits, and no more data is written to the buffer allocated by `evo_malloc`. This means that the size of the overflow is controllable, as are the values of the bytes written out of bounds, to the limit of the size of `skipl` (0x1FF * 6 audio blocks).

This is a powerful primitive that I will refer to as the ‘buffer overrun capability’ of this vulnerability. But if you look closely, this bug also contains a leak.

EMDF content is written to the skip buffer with length `skipl`, but the EMDF container also has a size, `emdf_container_length`. What happens when `emdf_container_length` is larger than `skipl`?

```
 if ( skipflde && ... )
 {
        int skip_copy_len = 0;
        for ( int block_num = 0; block_num < total_blocks; ++block_num )
{
        if ( skiple )
        {
                        ...
                        for ( skip_copy_len; skip_copy_len < skipl; skip_copy_len++ )
                        {
                                b = read_byte_from_syncframe();
                                skip_buffer[skip_copy_len] = b;
                        }
        }
}
int i = 0;
for (i = 0; i < skip_copy_len; i+=2 )
```

```
{
        int16_t word = skip_buffer[i] | skip_buffer[i+1]);
        if ( word == "X8" )
        {
                        has_syncword = 1;
                        break;
        }
}
if ( has_syncword )
{
                …
emdf_container_length = skip_buffer[i + 1] | ( skip_buffer[i] << 8);
        bit_reader.size = emdf_container_length;
                bit_reader.data = skip_buffer[i + 2];
        }
}
```

So while the skip buffer data is written based on `skipl`, the bit reader used to process the EMDF container has its length set to `emdf_container_length`. This means that EMDF data can be read outside of the initialized skip buffer. I will refer to this as the 'leak capability' of this vulnerability going forward.

We didn't report the leak capability is a separate vulnerability from CVE-2025-54957, as it doesn't have a security impact independent of the bug. The skip buffer is initialized to all zeros when the decoder starts, and afterwards, only syncframe data (i.e. the contents of the media being processed) is written to it. So in normal circumstances, an attacker couldn't use the leak capability to leak anything they don't already know. Only when combined with the buffer overrun capability of the vulnerability, does the leak capability become useful.

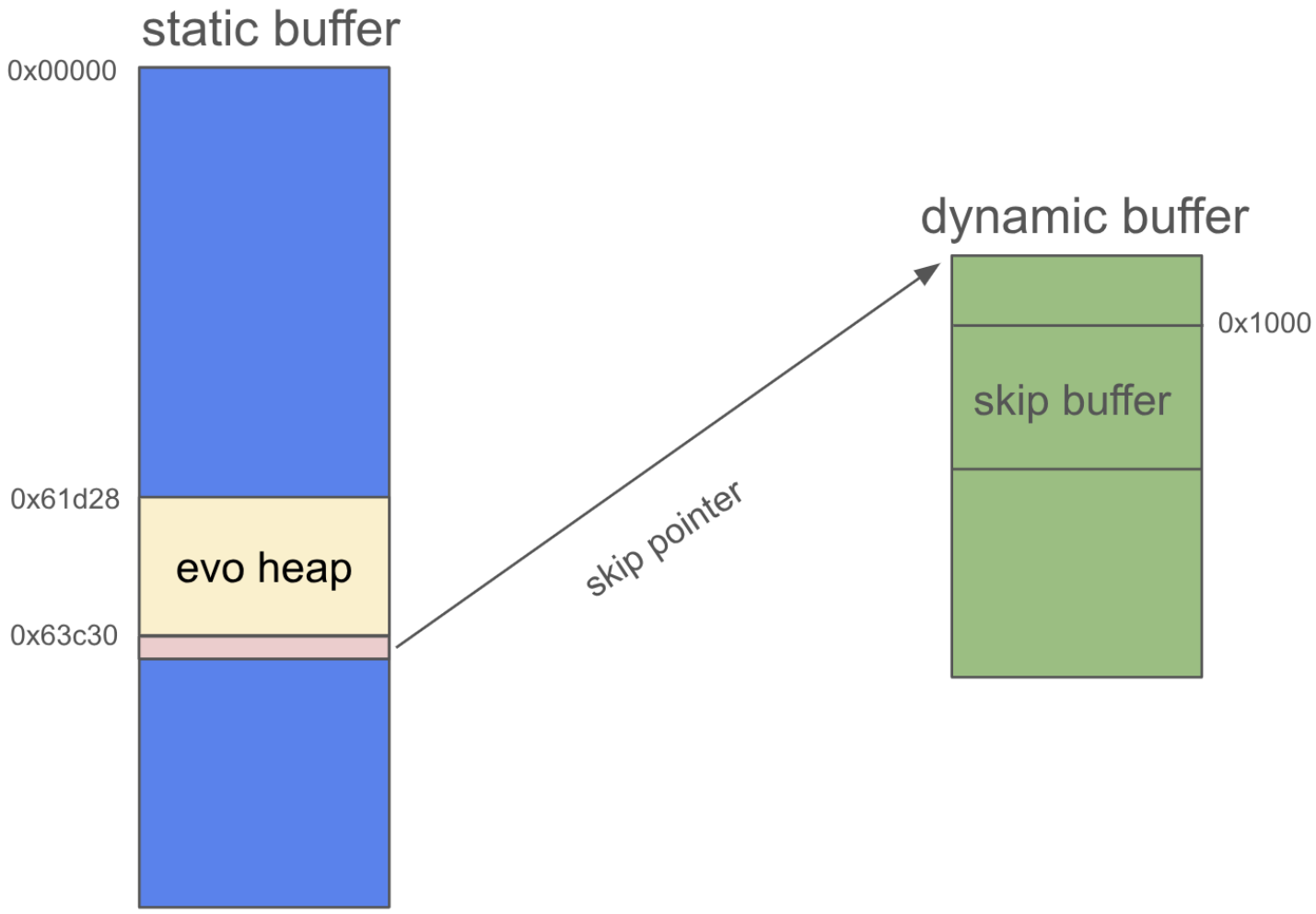
Decoder Memory Layout

The next step in exploiting this bug was understanding what structures in memory it can overwrite. This required understanding the memory layout of the UDC. The UDC performs a total of four system heap allocations when decoding DD+ audio, all occurring when the decoder is created, before any syncframes are processed. These allocations are freed and re-allocated between processing each media file. This is fairly typical of media decoders, as system heap allocations have non-deterministic timing, which can cause lag when the media is played.

One buffer that is allocated is the 'static buffer'. This buffer contains a large struct, which supports all the functionality of the decoder. The evo heap is part of this buffer. On Android, the size of the static buffer is 692855. Another buffer that is allocated is the 'dynamic buffer'. This buffer is used as 'scratch space' for a variety of calculations, and is also the location of the skip buffer. It is 85827 bytes long. The other two allocations are for input parameters and output data, and aren't relevant to this exploit.

The terms 'static buffer' and 'dynamic buffer' are somewhat confusing, as there are other static and dynamic buffers used by the decoder, and both buffers are dynamically allocated. However, these are the names used by Android when integrating the UDC. Throughout this post, the term 'static buffer' will always refer to the 692855-byte buffer allocated by the UDC on initialization, and the term 'dynamic buffer' will always refer to the 85827-byte buffer allocated by the UDC on initialization, and no other static or dynamic buffers.

The following diagram shows where the skip buffer and evo heap are located in relation to these buffers:



The evo heap is located at offset 0x61d28 in the static buffer, and immediately afterwards is the pointer used to write to the skip buffer when processing EMDF, which I will call the 'skip pointer'. It points 0x1000 below the skip buffer, and 0x1000 is added to its value to calculate the address that skip data (`skipfld`) is written to each time a syncframe is processed.

This means the vulnerability has the potential to overwrite a pointer that is later written to with attacker-controllable content, the skip data of the next syncframe. Unfortunately, this is not as simple as using the buffer overrun capability to overwrite the pointer, as the evo heap is 0x1f08 bytes long, and the maximum value of `skipl` is 3066 (0xbfa = 0x1ff * 6 audio blocks), meaning that the value the skip pointer would be overwritten with is not immediately controllable by simply decoding an EMDF payload that contains the bug.

This behavior is demonstrated by the original proof-of-concept (https://project-zero.issues.chromium.org/428075495#attachment67118327) attached to CVE-2025-54957. This file causes the buffer overrun to occur, but because the skip pointer is more than 3066 bytes away from the evo heap allocation that is overwritten, data is copied from outside the skip buffer. Since this memory is always zero, the skip pointer is overwritten with 0, and a null pointer crash occurs when the skip data from the next syncframe is written.

To get around this, the buffer overrun needs to be triggered on an evo heap allocation when the heap is partially filled. Fortunately, an EMDF container can contain multiple EMDF payloads, and parsing each payload allocates memory on the evo heap. Analyzing `ddp_udc_int_evo_parse_bitstream`, the function that performs this parsing and allocation, the smallest possible payload consumes 19 bits from the skip buffer. Meanwhile, every EMDF payload processed causes 96 bytes to be allocated on the evo heap. This means it

would take roughly 99 payloads to fill up the evo heap, which translates to 235 bytes of skip data. This is well within the available skip data space. Using this technique, it was possible to overwrite the skip pointer with a controllable absolute value, then write arbitrary data to it.

Write what where?

While this is a useful primitive, its utility is limited by ASLR, as an attacker would need to know the absolute value of a pointer to write to, which is unlikely in a 0-click context. Another possibility is partially overwriting the skip pointer, for example, 0x7AAAAA00A0 could be overwritten to be 0x7AAAAA1234. Since the skip pointer originally points to the dynamic buffer, this allows most of the dynamic buffer to be overwritten. Unfortunately, the dynamic buffer is only used to store temporary numeric data and does not contain any pointers or other structures that would be helpful for exploitation, but there is one useful aspect of this primitive. Normally, only 3066 bytes of skip data can be written to the skip buffer, but it can allow an attacker to write more.

For example, imagine the following series of syncframes:

1. Sets skip pointer to 0x7XXXXX4000
2. Writes 3066 bytes of skip data to skip pointer
3. Sets skip pointer to 0x7XXXXX3800
4. Writes 0x800+ bytes of skip data to skip pointer

Now the length of the available data in the skip buffer is 3066 + 0x800, and this can be chained with more syncframes to write up to 0xFFFF bytes into the dynamic buffer. This isn't on its own a path to exploitation, but it is a primitive that will become useful later. I will refer to it as WRITE DYNAMIC in future sections.

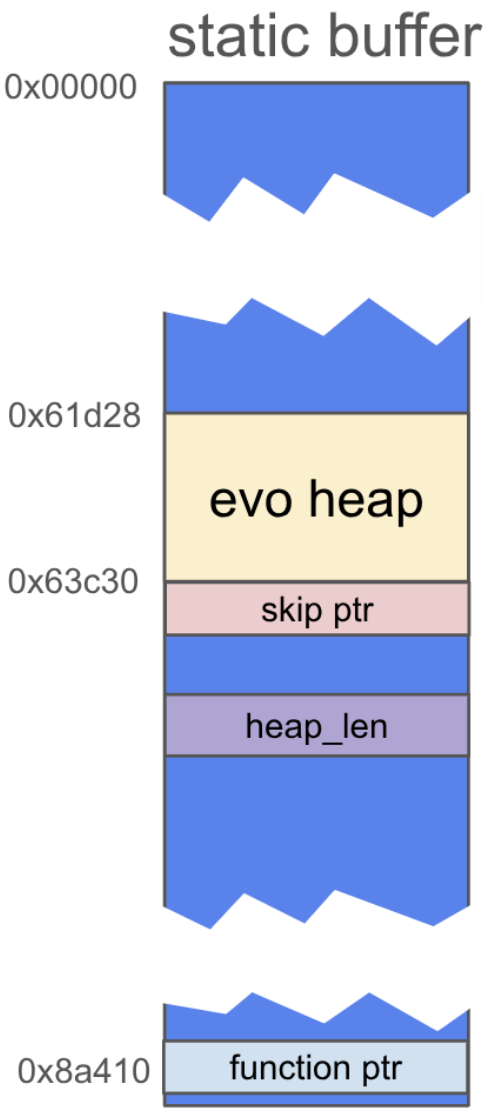
There is one subtlety that is important to notice. Why does syncframe 3 only move the skip pointer back 0x800 (2048) bytes when it could move it back 3066 bytes? This is because setting the skip pointer overwrites the data in the skip buffer. So syncframe 2 writes 3066 bytes, but syncframe 3 overwrites, for example, 200 bytes of that, then syncframe 4 needs to write 0x800+200 bytes to 'fix' the overwritten data. So to accurately write a long buffer to the dynamic buffer, the memory overwritten by each syncframe needs to overlap. But never fear, with enough syncframes, it is possible to fill almost the entire dynamic buffer with attacker controlled data. It is also possible to set the skip pointer to process the written data without modifying it by setting the skip pointer to the start of the data to be processed in one syncframe, then processing a second syncframe with `skipl` of 2, which will only write the syncword ('X8'). The skip data will then be processed based on the `emdf_container_length` already written.

Regardless, the WRITE DYNAMIC primitive was clearly not sufficient for exploitation, so I decided to take a step back and figure out what memory I could overwrite to gain code execution, even if I didn't have an immediate strategy for overwriting it. Analyzing the static buffer, I learned that my options were fairly limited. There are only two function pointers in the entire static buffer, called very frequently by the function `DLB_CLqmf_analysisL`, at offsets 0x8a410 and 0x8a438. This appears to be the only dynamically allocated memory used by the UDC that contains any function pointers.

Note that 0x8a410 and 0x8a438 are absolutely gargantuan offsets. They are more than 0x20000 bytes from the end of the evo heap, at address 0x63c30. A typical exploitation approach might be to directly overflow the heap to overwrite one of these pointers, but this offset is far too large. Even if the above primitive was used to fill the entire dynamic buffer (writable length 0xFFFF) with EMDF container data, it would still not be enough data to overwrite these pointers.

Extending the evo heap

A different approach was needed, so I revisited the static buffer, looking for other fields I could overflow near the end of the evo_heap. One looked interesting:



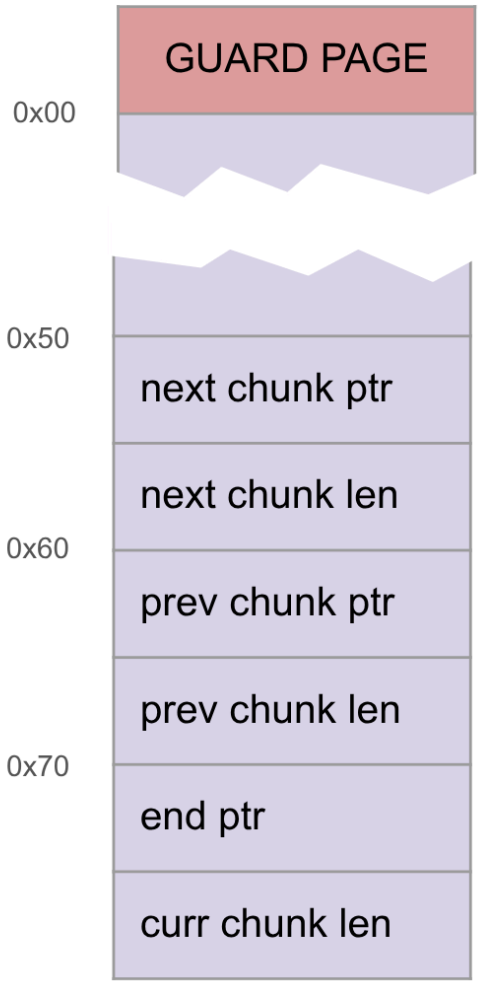
The `heap_len` is used to set the allocation limit of the evo heap during the processing of each syncframe. If it could be overwritten, it would be possible for the evo heap to allocate memory outside of its original bounds.This was a very promising possibility, as it had the potential to enable a primitive that would allow relative writes within the static buffer. For example, if I overwrote the heap length with a very large value, then allocated 0x286e8 bytes, since the evo heap starts at offset 0x61d28 and I am able to allocate and write to evo heap memory, would I then be able to write to offset 0x61d28 + 0x286e8 = 0x8a410?

Of course, this is still limited by the available size of the skip data, which is now 0xFFFF due to the WRITE DYNAMIC primitive. But since payloads use skip buffer memory at a ratio of 19 bits to 90 bytes, the function pointer could theoretically be overwritten using 0x286e8 / 90 * 19 / 8 = ~ 0xa000 bytes of skip data, which is smaller than the available 0xFFFF bytes.

Overwriting `heap_len` presents a challenge, though, as a write that reaches it will also overwrite the skip pointer, and if the skip pointer is invalid, it will cause a crash before the new value of `heap_len` is processed. One way to get around this would be to know the absolute value of a writable pointer and include it in the data that overwrites the memory, but without an information leak, this isn't practical on a Pixel. Another would be if there was a valid pointer in the dynamic buffer, as using the leak capability, it would be possible to embed it in the skip data for a frame and use it for the overwrite, but the dynamic buffer only contains numeric data.

Then I realized that the dynamic buffer does contain pointers. Not in the allocated portion, but in the contiguous metadata included in the allocation by Android's scudo allocator. Inspecting the dynamic buffer in a debugger, the pointer always has the address format 0x000000XXXXXXX0A0. The offset of 0xa0 leaves space for the heap header.

The heap header of the dynamic buffer is as follows:



The memory between offset 0x00 and 0x50 is unused by the scudo heap because this is a secondary (large) allocation, but unfortunately, there is a guard page before the header, and 0x50 bytes is not enough space for

the EMDF container needed to overwrite the skip pointer and heap length, so I investigated ways to increase the unused memory between the guard page and allocation header. I discovered:

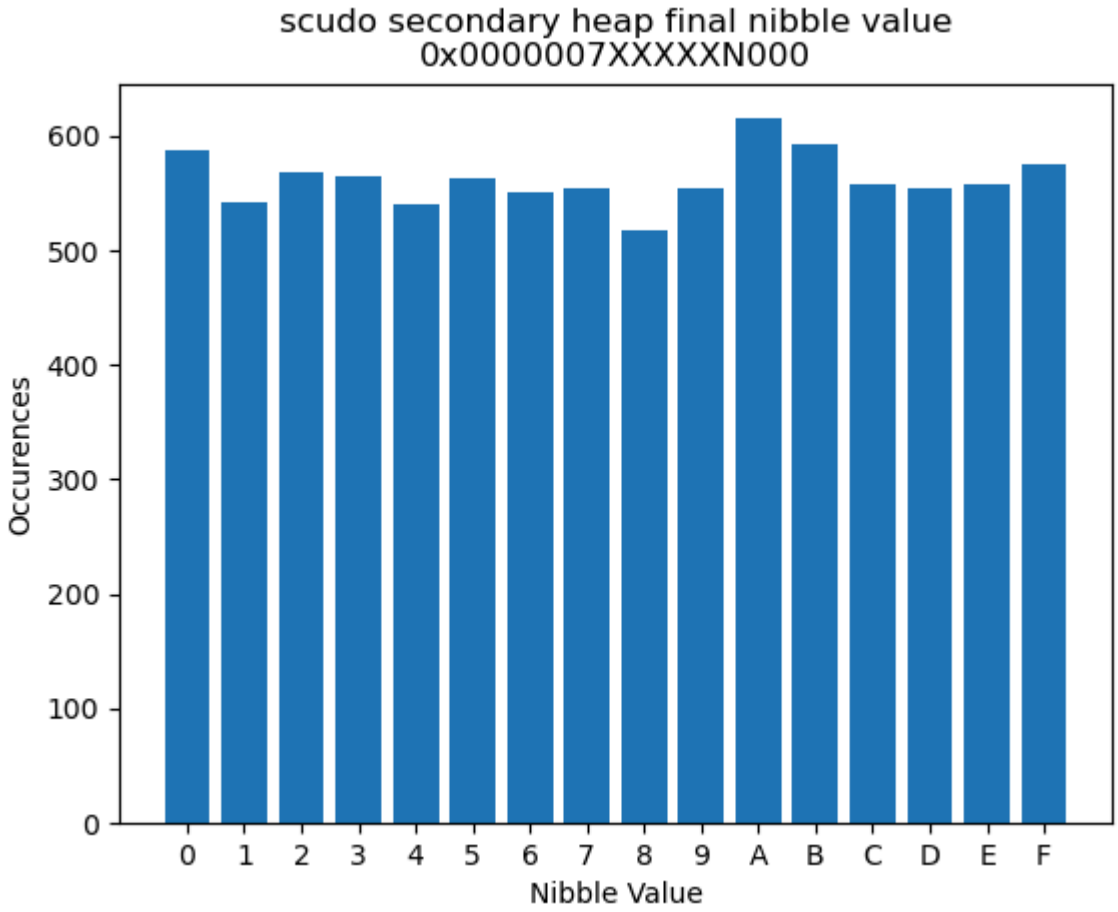
- If a secondary allocation is freed, and a chunk that is up to 0x2000 bytes smaller is then allocated, the freed chunk will be reallocated to satisfy the request. More importantly, the heap header will be shifted upwards. For example, if a heap chunk of size 0x17000 is allocated at 0x7f00000000 then freed, and then an allocation of size 0x15000 is made, then the chunk will be reused, but the heap header will now be at 0x7f00002000.
- When a secondary chunk is freed, scudo determines the size entirely based on the "curr chunk len" field shown above

It's also important to note that the dynamic and static buffers are such large allocations with such unusual sizes that scudo always allocates them in the same location in a specific process, allocating the memory when the decoder is initialized and freeing it when it is uninitialized, as once the chunks are created by the heap, they are the only suitable existing chunks to fulfill an allocation request of that size. (Note that the UDC runs in a separate process from other codecs on Android.)

Putting this all together, it is possible to point the skip pointer to the 'curr chunk len' field of the dynamic buffer's header, then overwrite it, so the chunk's length is 0x17000 instead of 0x15000. Then, when the decoder is reset (i.e. when a new file is played), the buffer will be reallocated, with an extra 0x2000 bytes of writable space before the heap header. This means the exploit will require decoding multiple files, but that isn't a problem when exploiting this bug via transcription, as multiple audio attachments to a single message are decoded in sequence.

There is a small ASLR problem with this step. As mentioned above, the dynamic buffer is allocated at a pointer with the format 0x000000XXXXXXY0a0, with X and Y being bits randomized by ASLR. The desired value to be written to is 0x000000XXXXXXY065. But remember, the skip buffer is actually at an offset of 0x1000 from the address the skip pointer references. So to perform the write, the skip pointer needs to be set to 0x000000XXXXXXZ065, where Z is one less than Y. This means the exploit needs to overwrite the nibble Y, and therefore know the value of Y, which is randomized by ASLR.

I did an experiment on a Pixel to see how this value was randomized and it seemed fairly even.



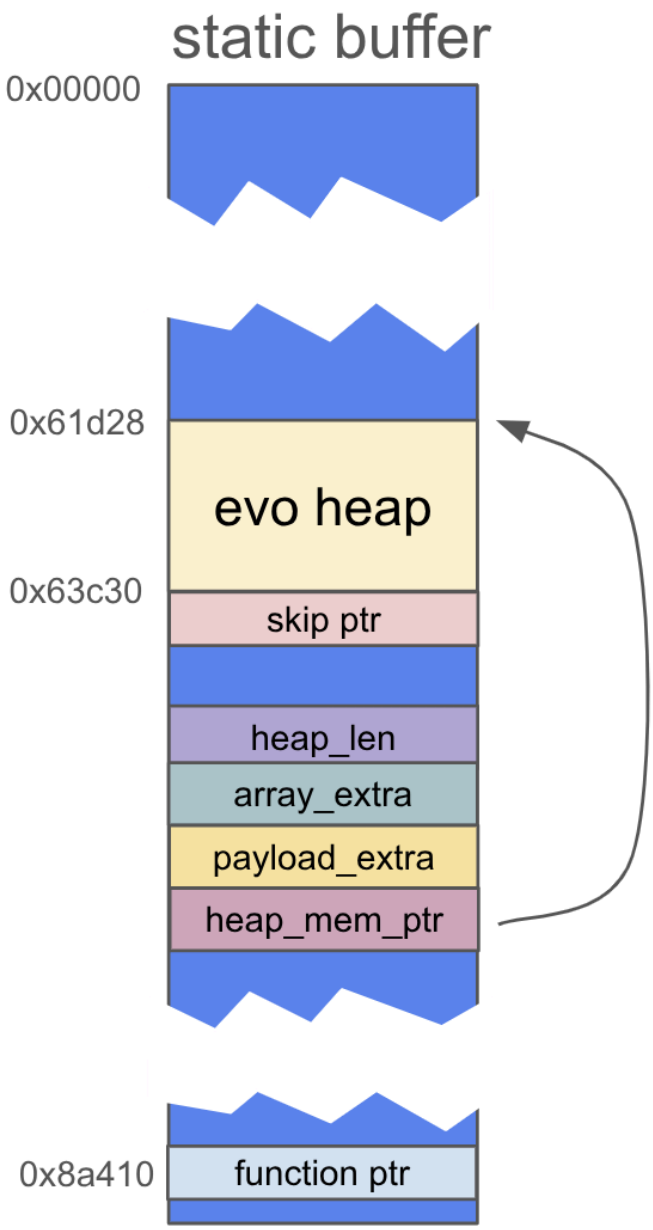
So the only option here is to guess this value, which means this exploit would work 1 out of every 16 times. This isn't prohibitive, though, as an attacker could send the exploit repeatedly until it works, and if the heap nibble value is wrong, the decoding process crashes and respawns after roughly three seconds, which means the exploit would succeed on average in 24 seconds.

My exploit assumes the nibble value is 3. With this, and the shifting of the scudo heap header described above, it's possible to insert an EMDF container before the heap header and use the leak capability of the bug to copy it over the skip pointer, then continue the copy to set the heap length. The heap length ends up being overwritten by audio data from early in the dynamic buffer (bit allocation pointers to be specific), which for the syncframe I used, is a value of 0x77007700770077.

Controlling PC

Now everything is ready to go: we can write and EMDF container with roughly 2070 EMDF payloads into the dynamic buffer, and when its processed ~0x28000 bytes of the evo heap gets allocated, then the final payload overwrites the function pointer at 0x8a410. Unfortunately, this didn't work.

It turns out that there are some other fields after the heap length in the static buffer.



To understand what these are, and why they are causing problems, we need to look more closely at how evo memory is allocated when EMDF payloads are processed. In highly simplified pseudocode, it works something like this.

```
int num_payloads = 0;

while(true){

        int error = evo_parse_payload_id(&reader, &payload_id);

        if(payload_id == 0 || error)
                break;

        num_payloads++;
error = evo_parse_payload(reader, payload_id, 0, 0, &payload, 0); //allocates no memory
        if(error)
                break;
}

void** payload_array = evo_malloc(evo_heap, 8 * num_payloads, 8 *  array_extra);

for (int i = 0; i < num_payload; i++){
        payload_array[i] = evo_alloc(88, 0);
}
```

```
reader.seek(0);

for (int i = 0; i < num_payload; i++){

        int error = evo_parse_payload_id(&reader, &payload_id);

        if(payload_id == 0 || error)
                break;

error = evo_parse_payload(reader, payload_id, evo_heap, 0, payload_array[i], 0);

        if(error)
                break;
}
```

Within the second call to `evo_parse_payload`, a single allocation (the same one which can overflow when the bug occurs) is performed as follows:

```
void* payload_mem = evo_alloc(payload_size, payload_extra);
```

On a high level, this code counts the number of EMDF payloads, then allocates an array of that size to hold pointers to a struct for each payload, then allocates a struct to represent each payload, and sets the corresponding pointer in the array to the struct allocation, then reads each EMDF object into its payload struct, optionally allocating payload memory if it contains payload bytes.

Two fields from the static buffer are marked in bold in the code above. `array_extra` and `payload_extra` are both integrator-configurable parameters that cause specific calls to `evo_alloc` to allocate extra memory.

So why does this cause my attempt to overwrite the function pointer in the static buffer to fail? When the decoder processes the EMDF container with a large number of payloads, it starts to allocate memory outside of the evo heap, because the heap length was overwritten with a very large size. The first evo heap memory allocated is the `payload_array`, an array of pointers that are later set to 88-byte evo heap allocations, one for each payload. With 2070 EMDF containers, this array is very large, 0x40B0 bytes. It overlaps `payload_extra`, and many other fields in the static buffer, setting them to pointer values. For fields that are interpreted as integers, like `payload_extra`, the end result is that they now contain numeric values that are very large.

Soon after `payload_extra` is overwritten, `evo_parse_payload` is called, which attempts the allocation:

```
void* payload_mem = evo_alloc(payload_size, payload_extra);
```

The allocation size is calculated by adding `payload_size + payload_extra` (with an integer overflow check) before the buggy addition of alignment padding that leads to the vulnerability padding occurs. Since pointers are tagged on Android, this will end up being something like:

```
total_size = payload_size + 0xB400007XXXXXXXXX;
```

Meanwhile, the heap length was overwritten to be 0x77007700770077, which is always smaller than `total_size`, so this allocation fails. Even worse, the overwritten `payload_extra` persists across syncframes,

meaning that no `payload_mem` allocation will ever succeed again. This prevents the bug from ever triggering again, as it requires a successful allocation, so there is no possibility of correcting these values in the static buffer.

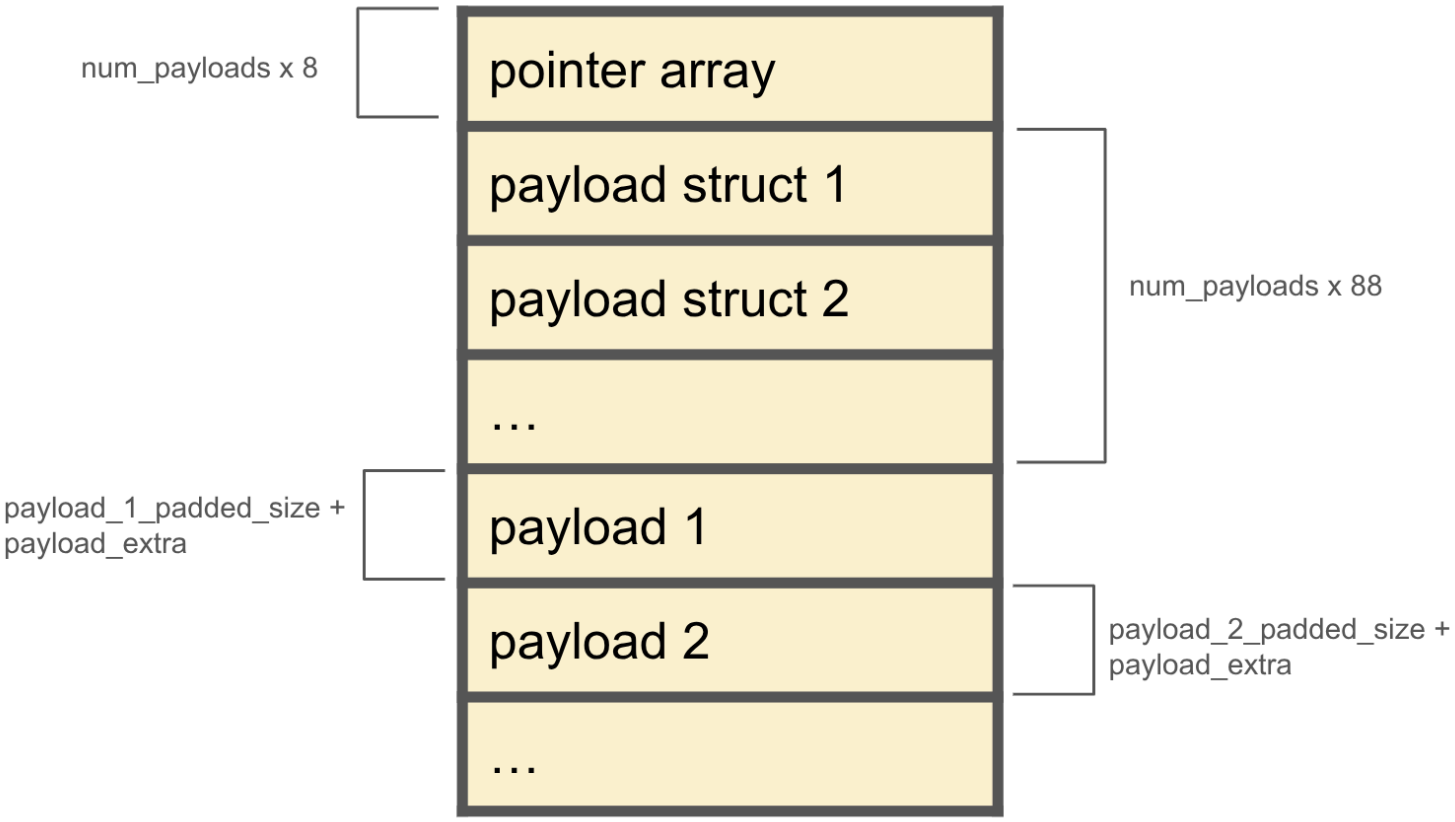
But maybe it isn't necessary to ever trigger the bug again, as the skip pointer is one of the many fields that gets overwritten by the huge `payload_array` allocation, causing it to point into the static buffer, above the evo heap. I'm going to skip over some details here, because I ended up not using this strategy in the final exploit, but by writing data to the altered skip pointer, it was possible to overwrite the function pointer, which demonstrated that this vulnerability could set the program counter!

Non-contiguous Overwrites

Controlling the PC showed this bug has excellent exploitability, but the above strategy had a serious downside: it prevented the bug from being triggered again, so I could only perform one overwrite, which would make achieving shellcode execution challenging. So my next step was to find a way to perform multiple non-contiguous writes to the static buffer.

When setting the PC, the unavoidable corruption of `payload_extra` prevented future overwrites, but I eventually realized that I could use the ability to set this field to my advantage.

The layout of allocations on the evo heap is as follows:



If an EMDF container contained two EMDF payloads, the data for the second payload would be allocated at `num_payloads` × 96 + `payload_1_size` + `payload_extra`. This allows payload_extra bytes to be allocated in the static buffer, but not overwritten by the payload. Since the length and contents of payload data is controllable by the attacker, it would be possible to write basically any data at any relative location in the static buffer if I could find some way to overwrite `payload_extra` with controlled data. The fact that

payload_1_size is also set from syncframe data makes this even more convenient. Since all the writes this exploit requires are fairly close to each other in memory, payload_extra only needs to be written once, so heap_base + num_payloads × 96 + payload_1_size + payload_extra is equal to the X0 parameter of DLB_CLqmf_analysisL (more on why this is a good choice later.) Then, by modifying the size of payload_1_size, the address of individual writes can be shifted by that many bytes. For example, if payload_1_size is 14 × 8, the function pointer in the static buffer discussed above will be overwritten.

Overwriting payload_extra

Unfortunately, the method used for overwriting the heap length is not sufficient to overwrite payload_extra as well, and the corruption that occurred while gaining PC control did not provide adequate control of the values overwriting payload_extra to perform the steps above. Remember, the heap length was overwritten by audio data in the dynamic buffer that happened to be written at an address soon after the static buffers's scudo heap header, and payload_extra was overwritten by a pointer. For just extending the heap length, setting the value to 'random garbage' was enough, but for multiple overwrites via payload_extra, a specific value is needed.

A simple solution would be to use WRITE DYNAMIC to write the data after the heap header to the needed value, but this isn't possible, because this address is written by the decoder while decoding a portion of the audio blocks called bit allocation pointers (baps), between when attacker-controlled data is written and when it is processed by the next syncframe. So even if the needed values are written with WRITE DYNAMIC, they are overwritten before they can be used to set payload_extra and nearby fields. I tried stopping the write from happening by including erroneous data in the syncframe that prevented baps from being written, but this also stopped EMDF data from being processed. I also tried altering an audio block to write controlled data in this location, but the possible values of baps are fairly limited, only low 16-bit integers.

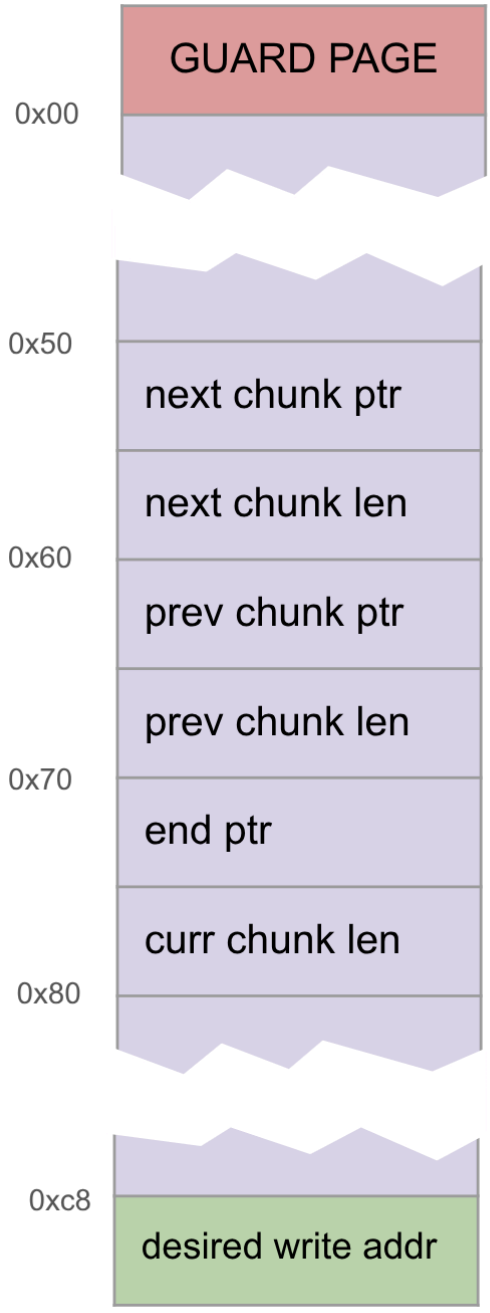
I eventually wondered if it would be possible to get the scudo heap to write an 'inactive' header, i.e. one that contains pointer values, but isn't currently in use. I experimented with scudo, and discovered that if a secondary chunk is the first one of that size ever allocated by a process (like the dynamic buffer is), its previous pointer will point to itself, and if the previous pointer is partially overwritten (for example, so the last two bytes are 0x5000 instead of 0x3000), the next time the chunk is allocated, the address returned by the allocator will be at the 0x5000 address, but the scudo header at 0x3000 will not be cleared. This only works because the dynamic buffer is the only buffer anywhere near its size that is allocated by the process, otherwise, there would be a risk that this buffer would be allocated again, leading to memory corruption that could cause a crash before the exploit is finished running.

Since the decoder needs to be reset to cause the dynamic buffer to be reallocated, implementing this required adding a third media file to the exploit, but this isn't a big cost in a fully-remote exploit, as three attachments can easily be added to the same SMS or RCS message. Now the exploit has three files:

- first.mp4 -- Using WRITE DYNAMIC, writes dynamic_base + 0x3061 to 0x48, causing the dynamic buffer to be reallocated at dynamic_base + 0x4800 when second.mp4 is loaded
- second.mp4 -- Using WRITE DYNAMIC, writes dynamic_base + 0x4861 to 0x50, causing the dynamic buffer to be reallocated at dynamic_base + 0x5000 when third.mp4 is loaded
- third.mp4 -- contains the rest of the exploit

Note that `dynamic_base` is the location of the dynamic buffer with the lower two bytes cleared, i.e. `dynamic_buffer && 0xFFFFFFFFFFFF0000` When the ASLR state needed for the exploit to work is correct, the dynamic buffer is at `dynamic_base + 0x3000`.

Now, there is a scudo heap header at `dynamic_base + 0x4800` that is not actively in use and does not get overwritten by baps that can be used to create an EMDF container that will overwrite `payload_extra`. But there is one problem. I explained earlier that, when filling a buffer using DYNAMIC WRITE, the exploit needs to perform overlapping writes downwards, because the next EMDF container, which is needed to move the skip pointer for the next step, overwrites some data at the start of the write. This doesn't matter when writing a long page of data, because the next write can fix the previous one, but it does in this case. The layout of the heap header is as follows:



I needed to write specific data to exactly offset 0xc8, but couldn't corrupt the 'prev chunk ptr' because it was needed to overwrite the skip pointer during the copy. There's 0x60 bytes between these, which is not enough for a payload that moves the skip pointer.

So I needed a new primitive. Thankfully, the way the decoder handles the EMDF syncword provides this. Basically, once skip data is copied into the skip buffer, the buffer is searched for the syncword ('X8'), and EMDF container parsing starts after the syncword. So it is possible to put some data before the syncword, and that gets written to the skip pointer, then put the container that moves the skip pointer after that. This allows the data to be written to the skip pointer, and then then skip pointer to be moved in a single syncframe, so that data doesn't get corrupted by a future skip pointer write. I will call this primitive WRITE DYNAMIC FAST. There's two downsides of this primitive compared to WRITE DYNAMIC. One is that since the EMDF container that moves the skip pointer and the data written are in the same syncframe, a smaller amount of data can be written. The other is that it is more difficult to debug. In a WRITE DYNAMIC syncframe, the address written to is always at the same offset, so it is easy to visually inspect many syncframes and determine where they are writing, but this is not the case with WRITE DYNAMIC FAST. So, my exploit uses WRITE DYNAMIC wherever possible, and only uses WRITE DYNAMIC FAST for writes that can't be accomplished with WRITE DYNAMIC.

With this primitive, I could create a syncframe that overwrites the skip pointer with a valid pointer to the dynamic buffer, then overwrites the heap length and `payload_extra`. This created a new primitive, which I will call WRITE STATIC. This allows a write to any offset in the static buffer larger than 0x63c30 relative to the static buffer's base!

Calling Controllable Functions

Now that I had the ability to perform multiple writes to the static buffer, it was time to figure out a path to shellcode execution. This required analyzing how the function pointers in the static buffer are called. It happens in the following function:

```
void* DLB_CLqmf_analysisL(void **static_buffer, __int64 *output_index, __int64 in_param)
{

//static_buffer is static buffer at offset 0x8a3c8
…

int loop_times = *(int*)static_buffer + 5);
int index = *(_DWORD *)static_buffer;
do
{
    index_val = *output_index++;
    param_X0 = static_buffer[12];
    param_val = param_X0 + 8 * index;
    (static_buffer[14])(
        param_X0,
        static_buffer[5],
        static_buffer[1],
        static_buffer[7],
        in_param);

    result = dlb_forwardModulationComplex(
        param_X0,
        index_val,
        param_val,
        *static_buffer,
        static_buffer[13],
        static_buffer[8],
        static_buffer[9]);

    index = *(unsigned int *)static_buffer;
```

```
    --loop_times;
    …
    }
    while ( loop_times );
    return result;
}
```

The function `dlb_forwardModulationComplex` contains the following condition:

```
if ( a7 )
{
  result = (__int64 (__fastcall *)(__int64, __int64, _QWORD))(*a7)(a3, a1, a4);
}
```

This function's behavior is extremely promising with regards to exploitation. It reads a function pointer and parameters out of memory that can be written with WRITE STATIC, then calls the function pointer with those parameters. There is also an option to make an indirect function call using `dlb_forwardModulationComplex`, if there happens to be a situation where a pointer to a function pointer is available instead of the function pointer itself. Finally, the call is repeated a specific number of times, based on a controllable value read out of the static buffer. Combining `DLB_CLqmf_analysisL` with WRITE STATIC, I could partially overwrite function pointers to run ROP with controllable parameters.

What's the plan, (Seth and) Jann?

As I developed this exploit, Jann Horn asked several times how I was planning to get from ROP to code execution in the `mediacodec` context, as Android has several security features intended to make this step difficult. I put this off as a 'future problem', but now was at a point where this needed to be solved.

Normally, my strategy would be to write a shared library to the filesystem, then call `dlopen` on it. Or write shellcode to a buffer and call a `mprotect` with ROP to make it executable. SELinux prevented both of these. It turns out the mediacodec SELinux context does not have any allow rule that allows it to open and write the same file, so `dlopen` was a non-starter. Additionally, mediacodec does not have execmem permissions, so making memory executable was also out. Making matters worse, `libcodec2_soft_ddpdec.so` makes limited calls to libc. So not very many functions were available for ROP purposes. For example, the library imports `fopen` and `fread`, but not `fwrite` or `fseek`.

Eventually, I got together with Jann Horn and Seth Jenkins to figure out a strategy to get from ROP to arbitrary instruction execution. Jann had the idea to write to `/proc/self/mem` This ProcFS file allows for any memory in a process to be overwritten for debugging purposes (i.e. to support software breakpoints), and could potentially be used to overwrite a function, and then execute it.

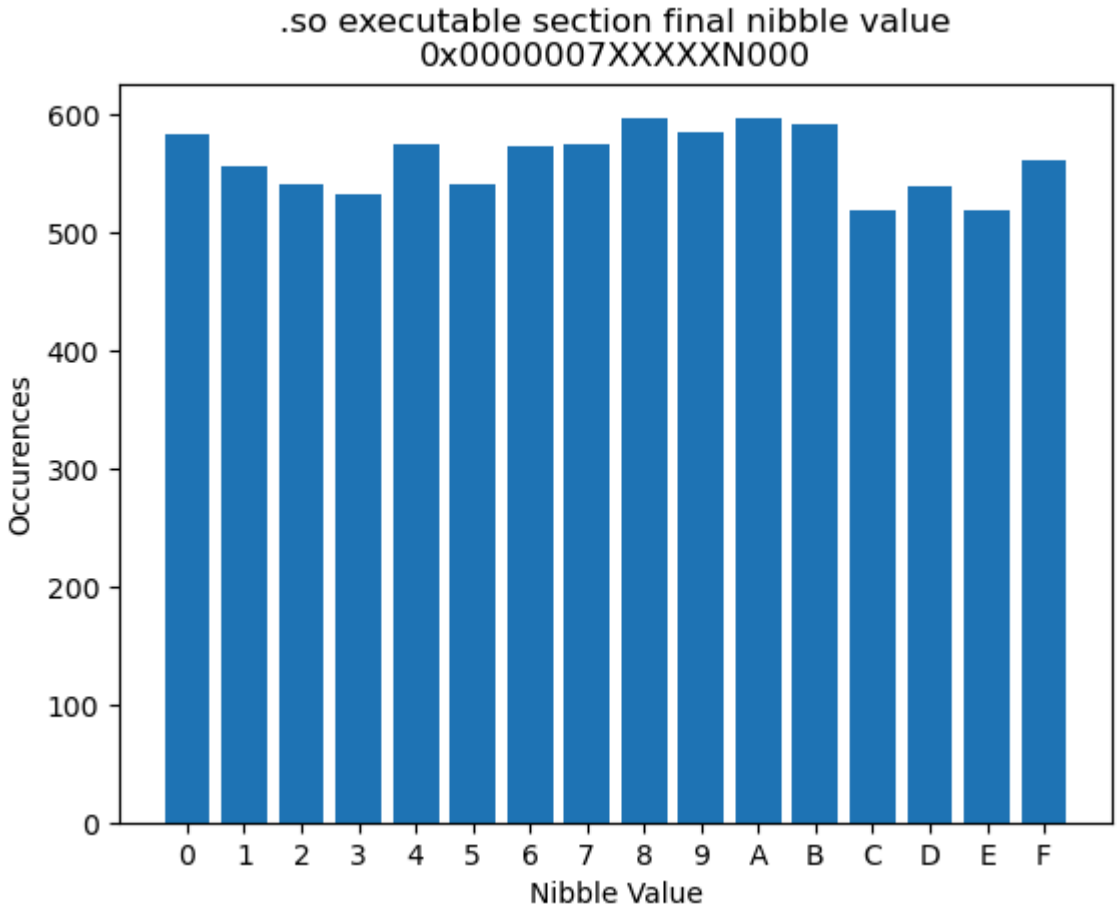
After investigating the `mediacodec` context's permissions, we came up with the following strategy:

1. Map shellcode into memory using WRITE DYNAMIC
2. Call `fopen` on `/proc/self/mem` many times, so a file descriptor number associated with `/proc/self/mem` can be easily guessed

3. Call `pwrite` to write the shellcode to a function that can later be executed. (Note that `pwrite` is not imported by `libcodec2_soft_ddpdec.so`, but nothing else that can write to a file handle is either).

Translating this sequence into ROP calls made by WRITE STATIC was more difficult than expected. One problem was that partially overwriting the function pointers in `DLB_CLqmf_analysisL` provided less functionality than I'd imagined. If you recall, `DLB_CLqmf_analysisL` makes two function calls that can be overwritten. The first is a direct call to `analysisPolyphaseFiltering_P4` at 0x26BDEC (note this isn't symbolized in the Android version of the library). The second is an indirect call to `DLB_r8_fft_64` via a pointer at offset 0x2A7B60.

The upper nibble of the second byte of where these functions are loaded is randomized by ASLR on Android. I tested this, and saw the behavior below, which is fairly uniform.



So my only options were to use ROP gadgets that involve only overwriting the first byte of the function pointers, or add additional unreliability to the exploit. The available gadgets weren't promising, so I decided to just guess this offset in my exploit, which adds another 1/16 probability, meaning the exploit will work one out of 256 times total. Considering the decoder process takes three seconds to respawn, this means the exploit would take on average around six minutes to succeed, which isn't prohibitive.

Guessing this nibble expands the available ROP gadgets to a span of 0xFFFF bytes, and it's possible to shift this span somewhat, depending on what value the exploit guesses this nibble to be. Still, this is only about 5% of the 1.3 MB of code in `libcodec2_soft_ddpdec.so`. For the indirect call, 0xFFFF spans almost the entire export table, as well as the global offset table (GOT), so there's some options there, but the library exports only about 40 functions from libc.

But it wasn't hopeless. For one, it is possible to call `memcpy` with these limitations, and if the parameters are unmodified, `dst` is a location in the dynamic buffer, and `src` is a location in the static buffer. Also, there was a promising ROP gadget in the accessible range:

```
0x000000000026ae38 :

ldr w8, [x1]
add w8, w8, #0x157
str w8, [x1]
ret
```

I will call this the “increment gadget”.

With this, I had a plan:

1. Change the indirect call to the `fopen` pointer in the GOT, and call it several times on `/proc/self/mem`
2. Change the indirect call to `memcpy`, and copy the `fopen` GOT entry to the dynamic buffer
3. Set the `dst` parameter of `memcpy` to the location of the GOT pointer in the dynamic buffer and call it again, causing a pointer to the `fopen` function in libc to be copied to the dynamic buffer
4. Use DYNAMIC WRITE to overwrite the last byte of the function pointer, so the distance between the pointer and `pwrite` is a multiple of 0x157
5. Call the increment gadget over and over to increment the function pointer in the dynamic buffer by 0x157 until its value is `pwrite`
6. Call `pwrite`
7. Profit?

This plan obviously glosses over a lot, most of which will be explained in the next section, but it is the plan I wrote up at the time.

One immediate question is “does the math work”? It seems to. In the version of the library I looked at, `fopen` is at 0x92E90 and `pwrite` is 0xDD6C0. A one-byte overwrite could change a `fopen` pointer to 0x92E4A, then:

```
0x157 × 890 + 0x92E4A = 0xDD6C0
```

Another question is whether this math would work generally, even on devices that have libc compiled with different offsets. I believe it would. In each version of libc, there are at least four call locations that will end up calling `pwrite`: `pwrite`, `pwrite`’s PLT, `pwrite64` and `pwrite64`’s PLT. If those don’t work, there’s combinations of `seek` and `write` or `fseek` and `fwrite`. Worst case, the exploit could change the GOT entry that’s read, so the math starts with a different function pointer than `fopen`. There are a very large number of possibilities and more than one is likely to work on every libc compilation.

The Exploit

Now, it was time to write the third file of the exploit. This turned out to be fairly complicated, with some unexpected problems. In order to explain these, this section will go through the third file of the exploit, one syncframe at a time. You can follow along here (https://project-

zero.issues.chromium.org/428075495#attachment72535031). Note that filenames that begin with numbers, for example, `10_write_x0` contain the actual syncframe data for that syncframe, meanwhile files with names like `make_10_write_x0.py` contain Python that generates the frame, often created with Gemini. Files with no corresponding Python were either handforged or exact copies of previous syncframes. Files appended with the suffix `_special` were generated with the corresponding Python, then altered by hand. The syncframes can be combined into a single MP4 file with correct checksums by running `combine_frames.py`.

longmem

The third exploit MP4 starts with the 36 syncframes in the longmem directory, containing the shellcode that the exploit eventually runs. The shellcode is copied to the dynamic buffer at descending addresses using DYNAMIC WRITE. As the exploit progresses, it performs actions that break DYNAMIC WRITE, so it's easiest to get this into memory now.

1_adjust_write_heap

This syncframe sets the skip pointer to `dynamic_base + 0xF000`.

2_adjust_write_heap_special

This syncframe uses DYNAMIC WRITE FAST to write 'wb' and "`/proc/self/mem`" to the address above, so they are available as parameters for a future `fopen` call, then moves the skip pointer to `dynamic_base + 0xD000`, so they aren't immediately corrupted.

3_adjust_write_heap

This syncframe sets the skip pointer to `dynamic_base + 0x48c8`, an offset that will correspond to the evo heap length and `payload_extra` once the memory is copied. (In hindsight, this could have been done in the previous frame, but too late now.)

4_adjust_write_heap_special

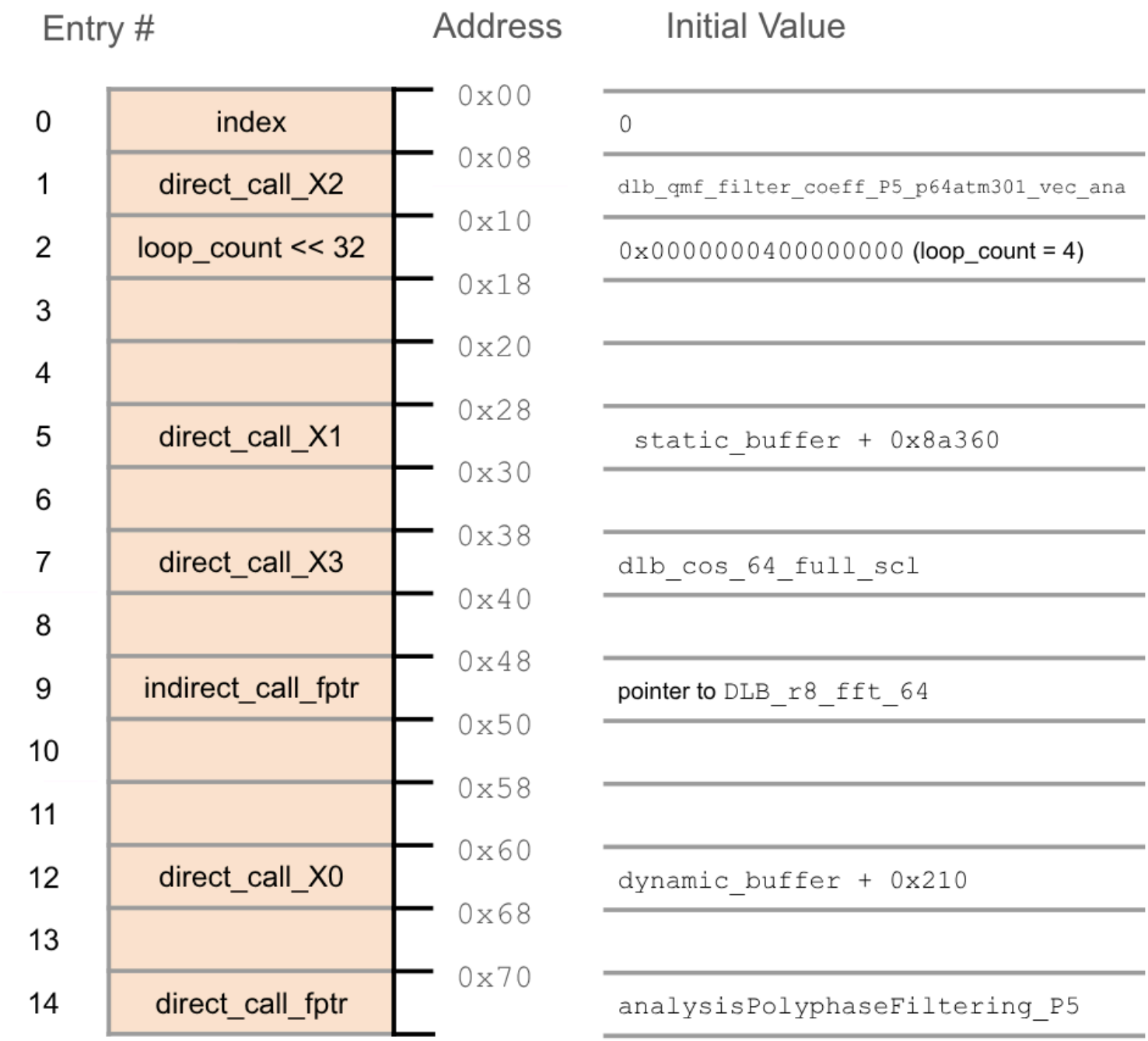
This synframe uses DYNAMIC WRITE FAST to write the memory at the offset corresponding to the evo heap length to 0xFFFFFFFFFFFFFFFF and the offset corresponding to `payload_extra` to 0x28530. It then sets the skip pointer to `dynamic_base + 0x473a`.

5_do_heap_write

This syncframe writes the start of an EMDF container to the address set in the previous frame, so that the data written by `3_adjust_write_heap`, `4_adjust_write_heap_special` and this syncframe together form a valid EMDF container, which is then parsed, triggering the bug and setting the heap length to 0xFFFFFFFFFFFFFFFF and `payload_extra` to 0x28530. This makes the WRITE STATIC primitive available, but also makes WRITE DYNAMIC and DYNAMIC WRITE FAST no longer function, as evo heap allocations no longer take up the same amount of space on the heap.

6_write_pc

To understand this and future syncframes, it's important to understand the functionality of WRITE STATIC in a bit more detail. The memory this primitive can write, which is eventually the X0 parameter to `DLB_CLqmf_analysisL` is laid out as follows:



The function pointer for the direct call is available to be overwritten, as are its parameters, ARM64 registers X0 through X3. The indirect function parameters are also calculated from values in this structure, which I will explain in more detail later.

Each 64-bit slot can be considered an 'entry' that needs to be individually overwritten in order to do non-contiguous partial overwrites. WRITE STATIC can alter a single entry per syncframe. Unfortunately, `DLB_CLqmf_analysisL` also executes once per syncframe, which can cause crashes or undesired behavior if the exploit is in the process of setting parameters when the call occurs.

This syncframe sets `direct_call_fptr` at entry 14 to a gadget that contains only the instruction `ret`, by doing a partial overwrite of the existing pointer. This prevents the direct function call from causing unexpected behavior.

7_garbage

Executing any frame with a valid EMDF header caused a crash after the previous frame, due to an out-of-bounds `memset`. Based on its parameters, this call is obviously intended to zero the evo heap, but since the heap length is now larger than the static buffer, it writes out of bounds. I performed a minimal analysis of what triggers this call and discovered that it requires processing two syncframes containing EMDF containers in a row, so I added in a syncframe that contains random invalid data to reset this. This 'garbage' syncframe is now required after every valid syncframe to avoid crashes. I will omit it as I continue through the exploit, but note that every future frame is even-numbered, because all the odd-numbered frames are 'garbage'.

8_write_str_str

Similar to syncframe 6, it is necessary to overwrite the indirect function pointer at entry 9 to avoid crashes as parameters are set, however, it is not possible to use ROP, as the entry needs to be set to a pointer to a function pointer. This syncframe sets entry 9 to the GOT entry pointing to `strstr` by doing a partial overwrite. While this isn't ideal, for the time being, `X0` and `X1` of the indirect call will always be pointers, and `strstr` doesn't modify any memory, so running it repeatedly won't cause crashes or other problems.

10_write_x0

This syncframe prepares the X0 parameter for the indirect call to `fopen`. For this call, X0's value will be the pointer at entry 12 (`direct_call_X0`) plus an offset calculated from entry 0 (`index`). The entire calculation is:

```
indirect_call_x0 = direct_call_X0 + 8 *  index;
```

In syncframe 1, "`/proc/self/mem`" was already loaded into the dynamic buffer, and this syncframe sets `index` to 1, so X0 references this string, 8 bytes away from the string 'wb'.

12_write_x1

This syncframe partially overwrites entry 10, which is currently a pointer to the dynamic buffer so that its value is `dynamic_base + 0xF000`, making it point to the string 'wb'.

14_write_fopen

This syncframe partially overwrites entry 9, so the indirect function pointer now references `fopen`. `fopen` will immediately be called four times, the default value of `loop_count`.

16_garbage to 23_garbage

The exploit now processes a few garbage syncframes to run `fopen` repeatedly to 'spray' the file handle so it can be guessed. This works because the UDC process opens very few files, so the handles are predictable over a certain number.

24_write_str_str

Returns entry 9 (the indirect function pointer) to `strstr`, so `fopen` stops being called.

26_write_x2

This syncframe sets `direct_call_X2` (entry 1) to 0xb8 in preparation for a call to `memcpy`.

28_write_x0

This syncframe partially overwrites the dynamic buffer pointer in `direct_call_X0` (entry 12) to `dynamic_base + 0xEC00`, in preparation for a call to `memcpy`.

30_loop_count

This syncframe sets the `loop_count` in entry 2 to 1, so future function calls do not execute multiple times per sycframe.

32_memcpy

This syncframe sets the direct function pointer (entry 14) to a `memcpy` gadget at 0x26cc2c, which is then called, causing the static buffer to be copied to the dynamic buffer, including an indirect pointer to `strstr`, set at entry 9 above. Note that the copy will occur every syncframe until entry 14 is overwritten again.

34_write_x0

The previously-set value of direct_call_X0 was a dummy value, to keep the copy away from skip buffer while the previous, especially large, EMDF container was being processed. This syncframe sets it to the actual copy destination, dynamic_base + 0x5F83.

36_zero_page and 38_copy_x1_special

The next two syncframes copy the newly written `strstr` GOT entry pointer to `direct_call_X1` using the leak capability of the vulnerability, so it can be the `src` parameter of the next memcpy.

`36_zero_page` writes zeros, followed by the end of an EMDF container to the skip pointer.

The `memcpy` then occurs, copying the GOT pointer into the middle of the EMDF container.

`38_copy_x1_special` writes the head of the EMDF container to the skip pointer, then the container is parsed, causing `direct_call_X1` (entry 5) to be set to the GOT pointer.

40_write_x0 and 42_write_x0

Syncframe 40 sets `direct_call_X0` (entry 12) to `dynamic_base + 0xEF00`. `memcpy` is then called, causing a direct pointer to `strstr` to be copied to that address. Syncframe 42 sets it to `dynamic_base + 0x6043`, so the copied memory doesn't get corrupted, and to set up the next `memcpy` call.

44_write_x2, 46_write_scf, 48_zero_page and 50_write_x3_special

Though it wasn't strictly necessary at this point, I wanted to set `direct_call_X3` to `strstr`, so it would be available as `offset`, the fourth parameter to the eventual `pwrite` call. This made sense because the pointer

was currently available in the dynamic buffer, and all other direct calls needed by the exploit had fewer than four parameters. Flash forward to the future: this was a bad idea.
The `offset` parameter specifies the location `pwrite` writes to, which for `/proc/self/mem` in this exploit is the address of a function that will be overwritten with shellcode. `strstr` seemed perfect, because I could already make controlled calls to it, and it otherwise doesn't get called a lot, but when I ran the finished exploit, it didn't work, because `getpid`, `munlock` and several other frequently-called functions were located immediately after it in libc. They usually got called first, causing the exploit to jump into the middle of the shellcode.

It was easiest just to use `memcpy` to copy a different function pointer, and after some testing, I selected `__stack_chk_fail`, as it doesn't get called during normal operation and the functions after it in libc aren't used by the UDC either. So this combination of syncframes uses the same trick as was used to copy the `strstr` GOT into `direct_call_X1` to copy a pointer to `__stack_chk_fail` into `direct_call_X3`. Note that this only takes one 'round' of using the leak capability to copy a pointer, versus two for `strstr`, because I was able to partially overwrite the pointer to the `strstr` GOT entry in `direct_call_X1` to so it pointed to the `__stack_chk_fail` GOT entry, so didn't need to copy the static buffer a second time.

52_set_pc_back

This syncframe sets the direct function call back to the ret gadget, so it stops calling `memcpy`.

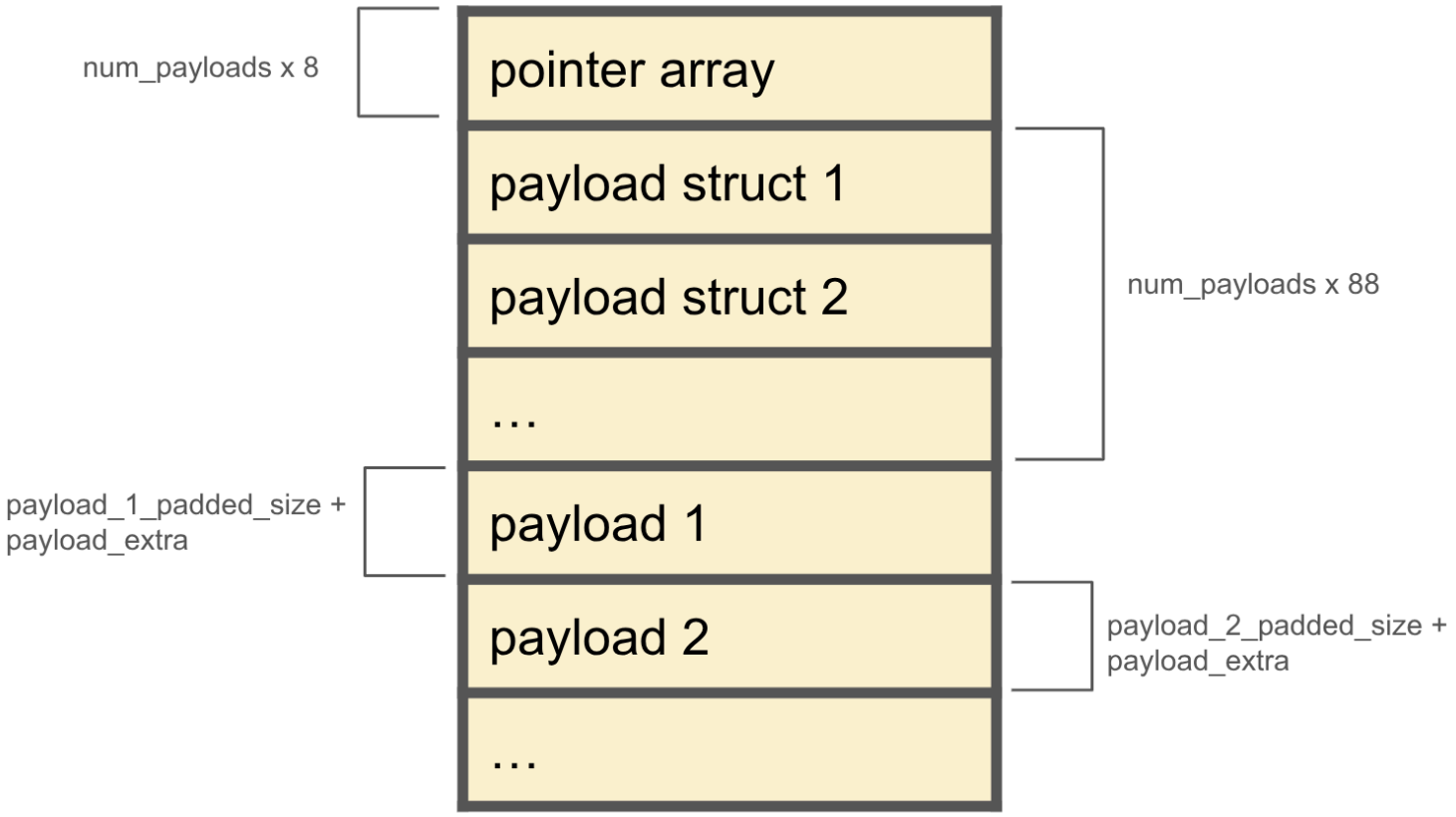
54_write_skip, 56_write_x1_end_special and 58_write_x1_start_special

When starting this exploit, I genuinely believed it would be possible to get shellcode execution without WRITE DYNAMIC once WRITE STATIC was unlocked. This turned out to be wrong. In the plan I wrote up for the exploit, I missed the fact that `direct_call_X1` was set to the GOT at this point in the exploit, but needed to be set to the dynamic buffer.

Some nice pointers to the dynamic buffer were already in the dynamic buffer from when I had copied the static buffer there to get the address of the GOT, and I could use the same trick to copy one to `direct_call_X1` that I'd used to copy the other pointers, but I'd need to move and write to the skip pointer to their address. I decided at this point the easiest path forward would be to regain the WRITE DYNAMIC primitive.

This was really just a math problem: the original WRITE DYNAMIC primitive would allocate a lot of EMDF payloads to exhaust the heap, then trigger the buffer overwrite capability to alter the skip pointer, meanwhile, with `payload_extra` overwritten, this would fail due to an integer overflow check failing when it is added to the payload size. But it's not actually necessary to trigger the vulnerability once the heap length is overwritten, as the evo heap no longer accurately checks whether heap writes are out of bounds.

As a refresher, the evo heap is laid out as follows:



The new WRITE DYNAMIC allocates the perfect number of payloads so that the allocation size of the pointer array plus the payload structs is exactly even with the skip pointer, and then the first payload's data overlaps with the pointer, and can be used to overwrite it.

These syncframes use a series of WRITE DYNAMIC and WRITE DYNAMIC FAST calls to set `direct_call_X1` to the dynamic buffer.

60_write_skip, 62_write_single_byte and 64_move_skip

The first two syncframes use DYNAMIC WRITE to overwrite the final byte of the pointer to `strstr`, so it is a multiple of 0x157 away from `pwrite`. The final syncframe moves the skip pointer to another address so it doesn't write the byte a second time.

66_write_index

The exploit is about to call the increment gadget a large number of times, which will also increment the variable index at entry 0 in `DLB_CLqmf_analysisL`. This syncframe sets its value to zero, so that these future increments don't lead to reads out of bounds.

68_loop_count

This syncframe sets the loop_count in entry 2 to 0x7B, so that the increment gadget runs the correct number of times. Note that `DLB_CLqmf_analysisL` will run twice, causing the gadget to run 0xF6 times.

70_write_x1

`direct_call_X1` currently points somewhere in the dynamic buffer. This syncframe makes it point exactly to the modified pointer to `strstr`.

72_inc_157

This syncframe sets the direct function pointer to the increment gadget, which is then called 0xf6 times, causing the function pointer in the dynamic buffer to point to `pwrite`.

74_set_pc_back

Sets the direct call pointer back to the ret gadget, so incrementing stops.

76_set_malloc

The indirect function pointer is currently set to `strstr`. This will become a problem as its parameters are prepared for calling `pwrite`, as `pwrite`'s first parameter is a file handle (i.e. an integer), which will crash as the first parameter of `strstr`. This syncframe sets the indirect function pointer to `malloc`, as its GOT entry is within range and the call will succeed with a single integer parameter.

78_write_x0

This syncframe writes `direct_call_X0` with 40, the estimated handle to `/proc/self/mem`.

80_write_x1

This syncframe partially overwrites `direct_call_X1` so it points to the shellcode in the dynamic buffer.

82_write_x2

This syncframe writes `direct_call_X2` with the integer length of the shellcode.

84_write_end_special and 86_write_start_special

These syncframes copy the pointer to `pwrite` to the direct_call_fptr (entry 14), using the same method as other pointer copies from the dynamic buffer. `pwrite` is immediately called, overwriting `__stack_chk_fail` with the shellcode.

88_write_scf

This syncframe partially overwrites the indirect call register, so it points to the GOT entry for `_stack_chk_fail`. `__stack_chk_fail` immediately executes, running the shellcode!

How reliable is this exploit?

Due to ASLR guessing, this exploit works roughly 1 in 255 times. There is one other source of unreliability. Occasionally, binder performs a secondary allocation while the exploit is running, in which case, header checks fail and it crashes. This happens a lot when the debugger is attached, but I observed it less than 10% of the time when the process is running normally.

Another question is whether the exploit could be made more reliable. I have two ideas in this regard, both which would require substantial development effort.

To remove the 1/16 probability when guessing the dynamic buffer location, it might be possible to overwrite the second lowest byte of the prev pointer in the dynamic buffer allocation before exploitation starts. As discussed previously, this causes the buffer to be reallocated at that address, so this would have the end result of moving the allocation to a consistent offset from the `dynamic_base` before the exploit runs.

The challenge here would be to find a way to write to the header of the dynamic buffer while only overwriting the lowest byte of the pointer, as this is the only byte that can be overwritten without knowing the ASLR bits. One possibility is using the bap write feature of the decoder, as it writes data close to the skip pointer, but very limited data can be written. The `evod_process` function also writes to low addresses of the skip buffer after the EMDF container is parsed, so it might be possible to use this write as well.

This strategy would not make determining the dynamic buffer allocation 100% reliable, because the location where the dynamic buffer is reallocated needs to be mapped. For example, if an allocation at `dynamic_base + 0x3000` has its prev pointer overwritten to be `dynamic_base + 0xF000`, it will be shifted to that address, but if an allocation at `dynamic_base + 0xF000` is overwritten to be `dynamic_base + 0x3000`, it will crash when scudo attempts to write a heap header to the lower address, because that memory is not mapped. Overwriting the prev pointer to `dynamic_base + 0xF000` would theoretically always work, but that would limit DYNAMIC WRITE to addresses between `dynamic_base + 0xF000` and `dynamic_base + 0xFFFF`, because the primitive can only overwrite bytes in the address it writes to, it cannot increment the third lowest byte to extend this range. So this strategy would require reducing the amount of memory in the dynamic buffer that the exploit needs, but if that's possible, it could potentially remove the unreliability caused by the second nibble randomization of the dynamic buffer.

To remove the 1/16 probability when guessing the load address of `libcodec2_soft_ddpdec.so`, if it was possible to copy a pointer to the dynamic buffer, it would then be possible to use the second nibble of that pointer as the `emdf_container_length` of a syncframe. For most lengths, it's then possible to craft an EMDF container that would not trigger the bug if the length is too short, because the bytes triggering the bug aren't processed, and not trigger the bug if the length is too long (as `evo_parse_payload` is called twice, triggering the bug on the second call, so an invalid payload occurs after the trigger, it prevents the trigger from running). Then, a series of syncframes that work with all 16 possible library offsets could be crafted, and only the correct ones would be processed.

The real challenge here would be copying from the static buffer to the dynamic buffer without guessing the library location, as both the direct and indirect calls available are quite limited. But if this was possible, the unreliability due to not knowing the library load address could be avoided, at the cost of substantial development effort.

Overall, I suspect it's possible to substantially improve the reliability of this exploit, though it would likely require several months more development effort.

Reflections on Mitigations

My progress writing this exploit was impeded by several Android platform mitigations, meanwhile others were not as effective as I expected, so I want to take this chance to reflect on what worked and can be improved.

ASLR was by far the most challenging mitigation to bypass, this exploit would have been substantially easier to write without it. Partially overwriting pointers to bypass ASLR is a common exploit strategy, and I was surprised by how much more difficult randomization of low bits of the pointer made it. While it's also important that pointers have enough overall randomization that they can't be guessed, my takeaway from this is that randomization at low address bits does a lot more to increase exploit development time than randomization at high bits.

I also performed a lot of testing of Android ASLR, and I did not find any areas that were not randomized enough to prevent exploitation. This has not always been true of Android in the past, and I was pleased to see that Android ASLR appears to be well implemented and tested.

SELinux also made exploitation more difficult, as a lot of 'classic' techniques for running shellcode didn't work, and I was lucky to have access to experts like Seth and Jann who could help me understand the restrictions on the system and how to get around them. That said, that is likely a one-time cost for attackers: once they learn strategies for bypassing SELinux, they will work for multiple exploits.

The `mediacodec` context usually has seccomp rules that prevent a process from executing syscalls that aren't needed for its normal functionality. A policy is implemented (https://android.googlesource.com/platform/frameworks/av/+/22c571b/services/mediacodec/minijail/seccomp_policy/mediacodec-seccomp-arm.policy) in AOSP, and I tested that the Samsung S24 enforces this policy on its media decoding processes. However, this was somehow left out of the Pixel 9. A seccomp policy similar to Samsung's would have prevented the call to `pwrite` used by the exploit. This wouldn't have prevented exploitation, as every syscall needed to access the BigWave vulnerability this exploit chains into must be callable by the decoder process for decoding to function correctly, but it likely would have forced the exploit to be written entirely in ROP, versus jumping to shellcode. This would have added at least a few more weeks of exploit development effort.

Likewise, the accessibility of `/proc/self/mem` was a big shortcut to exploitation. Since it is only used during debugging, I wonder if it is possible to implement some sort of mitigation that makes it inaccessible when a device is not being debugged.

scudo also lacked mitigations that could have made this exploit much more difficult, or even impossible. It was surprisingly easy to modify secondary headers to 'trick' the allocator into moving an allocation, meanwhile, in the primary partition, this would have been prevented by checksums. While vulnerabilities that allow a scudo secondary header to be modified are fairly rare, as every scudo secondary allocation is preceded by a guard page, the performance cost of adding checksums to the secondary partition would likely be limited, as in most applications, there are far fewer secondary allocations compared to primary allocations.

It's also important to note that part of why this vulnerability was exploitable in a 0-click context was because it is an exceptionally high quality bug. It contained both the ability to leak memory and to overwrite it, provided a high level of control over each and the structures that could be corrupted by the overwrite were unusually fortuitous. That said, the memory layout that enabled this isn't unusual among media decoders. For example, the H264 decoder that I reported this 2022 vulnerability (https://project-zero.issues.chromium.org/issues/42451420) in has a similar layout, with large structs, and could potentially be prone to similar exploitation techniques involving overflows between struct members.

On Mac and iOS devices we tested, the UDC is compiled with -fbounds-safety, a compiler mitigation which injects bounds checks into a compiled binary, including the bounds of arrays within C structs. We believe CVE-2025-54957 is not exploitable on binaries compiled with this mitigation. While there is a performance cost, compiling all media libraries with this flag would greatly reduce the number of exploitable vulnerabilities of this type. Even in situations where this is not practical in production, testing and fuzzing media libraries with -fbounds-safety enabled could make it easier to find and fix this type of exceptionally exploitable vulnerability.

The Next Step

Now that we've gained code execution in the mediacodec context, it is time to escalate to kernel! Stay tuned for Part 2: Cracking the Sandbox with a Big Wave (/2026/01/pixel-0-click-part-2.html) .■

make zeroday hard.