

Dirty Ptrace: Exploiting Undocumented Behaviors in Kernel mmap Handlers

*A review of challenges in securing driver
mmap memory for years*

Xingyu Jin and Martijn Bogaard



Android Red Team

Mission

Protect the Android ecosystem through offensive security.

Agenda

Part I: A Review of Known Driver mmap Vulnerabilities



Part II: Less Known mmap Vulnerability: “Dirty Ptrace”



Part III: Exploit “Dirty Ptrace”



Classic Driver mmap Vulnerabilities in the “golden age” of Android Rooting

Classic Driver mmap Vulnerability in Early Years

- CVE-2012-6422 (by alephzain), ...
 - `crw-rw-rw- 1 system graphics [...] /dev/exynos-mem`
 - `mmap /dev/exynos-mem` allows an unprivileged app to **map arbitrary physical pages** back to userspace
- CVE-2017-13164 (thomasking): information leak
- Exploitation is easy in old days
 - Read kernel symbols
 - Patch kernel function
 - Root

Stop Driver mmap Vulnerabilities

- SELinux was introduced and enforced
- OEMs start sanitizing user input
 - Introduce integer overflow / OOB bugs
- Integer overflow (e.g. CVE 2018-8781 by checkpoint, CVE-2019-18675 by deshal3v) in `.mmap` handler:

```
if (vm_pgoff + size > obj.allowed_len) {  
    return -EINVAL;  
}  
[...]  
remap_pfn_range(vma, vm_start, ...)
```

Stop Driver mmap Vulnerabilities

- OOB bug still happens in 2024 :(

```
static vm_fault_t cma_heap_vm_fault(struct vm_fault *vmf)
{
    struct vm_area_struct *vma = vmf->vma;
    struct cma_heap_buffer *buffer = vma->vm_private_data;

    if (vmf->pgoff > buffer->pagecount)
        return VM_FAULT_SIGBUS;

    vmf->page = buffer->pages[vmf->pgoff];
    get_page(vmf->page);

    return 0;
}
```

CVE-2024-46852

```
static vm_fault_t udmabuf_vm_fault(struct vm_fault *vmf)
{
    struct vm_area_struct *vma = vmf->vma;
    struct udmabuf *ubuf = vma->vm_private_data;

    vmf->page = ubuf->pages[vmf->pgoff];
    get_page(vmf->page);
    return 0;
}
```

CVE-2023-2008 (BFS Lab)

VM_PFNMAP is Difficult to Manage Securely

- Driver marks the mmaped **VMA** with the **VM_PFNMAP** flag
 - Basically, the **VM_PFNMAP** flag means it's up to the driver to manage this region of virtual memory.
- Easy to go wrong
 - **Reference count** <- Exynos Driver 0day CVE-2024-44068
 - The implication of the VM flags
 - Race condition
 - VM flag
 - GPU Page UAF
 - ...



In the wild Exynos Oday CVE-2024-44068

<https://googleprojectzero.github.io/0days-in-the-wild/0day-RCA/2024/CVE-2024-44068.html>

CVE-2024-44068: Exynos In the Wild Oday

- CVE-2024-44068 attacks the Exynos device driver `m2m1shot_scaler0` which is for JPEG image scaling and decoding
- Exploit chain hijacks `init` process by overwriting `libc`
 - If kernel major version is 4, gain root & disguise itself as `vendor.samsung.hardware.camera.provider@3.0-service`
 - If the target kernel major version is 5, mount an img file and load a rootkit driver

```
DWORD1(v75) |= 5u;  
ioctl(special_file_fd, LOOP_SET_STATUS64, v74);  
v22 = mount(special_file, "/data/troot_mnt/img", "ext4", 1uLL, &unk_2299);  
__
```

CVE-2024-44068: Exynos In the Wild Oday

- ioctl `M2M1SHOT_IOC_PROCESS` for importing memory pages for I/O operation

```
m2m1shot_task.buf_cap.num_planes = 3;  
m2m1shot_task.buf_cap.type = M2M1SHOT_BUFFER_USERPTR;  
m2m1shot_task.buf_cap.plane[SC_PLANE_Y].userptr = ion_map;  
m2m1shot_task.buf_cap.plane[SC_PLANE_Y].len = 0x8000LL;  
m2m1shot_task.buf_cap.plane[SC_PLANE_CB].userptr = buf_cap_map_2;  
m2m1shot_task.buf_cap.plane[SC_PLANE_CB].len = 0x8000LL;  
m2m1shot_task.buf_cap.plane[SC_PLANE_CR].userptr = buf_cap_map_3;  
m2m1shot_task.buf_cap.plane[SC_PLANE_CR].len = 0x8000LL;  
m2m1shot_task.fmt_out.fmt = V4L2_PIX_FMT_YUV420M;
```

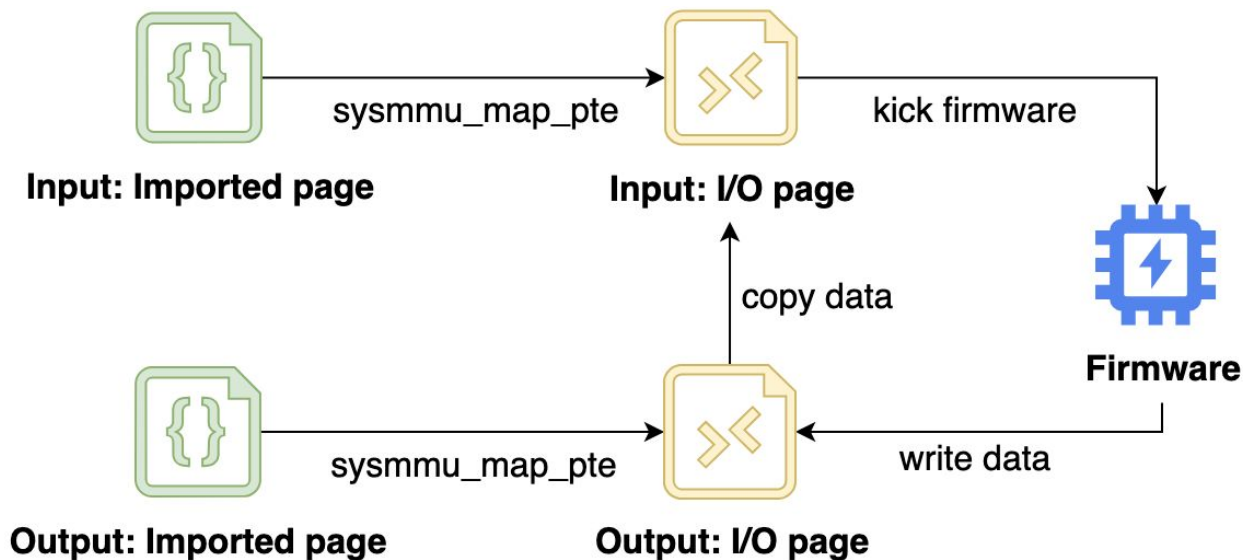
Destination: Imported memory from
userspace `userptr`

Source: Imported memory from
userspace `userptr`

```
m2m1shot_task.buf_out.num_planes = 3;  
m2m1shot_task.buf_out.type = M2M1SHOT_BUFFER_USERPTR;  
m2m1shot_task.buf_out.plane[SC_PLANE_Y].userptr = buf_out_map_1;  
m2m1shot_task.buf_out.plane[SC_PLANE_Y].len = 0x8000LL;  
m2m1shot_task.buf_out.plane[SC_PLANE_CB].userptr = buf_out_map_2;  
m2m1shot_task.buf_out.plane[SC_PLANE_CB].len = 0x8000LL;  
m2m1shot_task.buf_out.plane[SC_PLANE_CR].userptr = buf_out_map_3;  
m2m1shot_task.buf_out.plane[SC_PLANE_CR].len = 0x8000LL;  
m2m1shot_task.op.op = M2M1SHOT_OP_CSC_NARROW;
```

CVE-2024-44068: Exynos In the Wild Oday

- `m2m1shot_dma_addr_map -> exynos_iovmm_map_userptr -> exynos_iommu_map_userptr -> sysmmu_map_pud -> sysmmu_map_pte -> ...`



CVE-2024-44068: Exynos In the Wild Oday

- **sysmmu_map_pte**: Hey it's not my duty to take care of PFNMAP pages

```
static int sysmmu_map_pte(struct mm_struct *mm,
    pmd_t *pmd, unsigned long addr, unsigned long end,
    struct exynos_iommu_domain *domain, sysmmu_iova_t iova, int prot)
{
    ...
    ent = alloc_lv2entry_userptr(domain, iova);
    ...

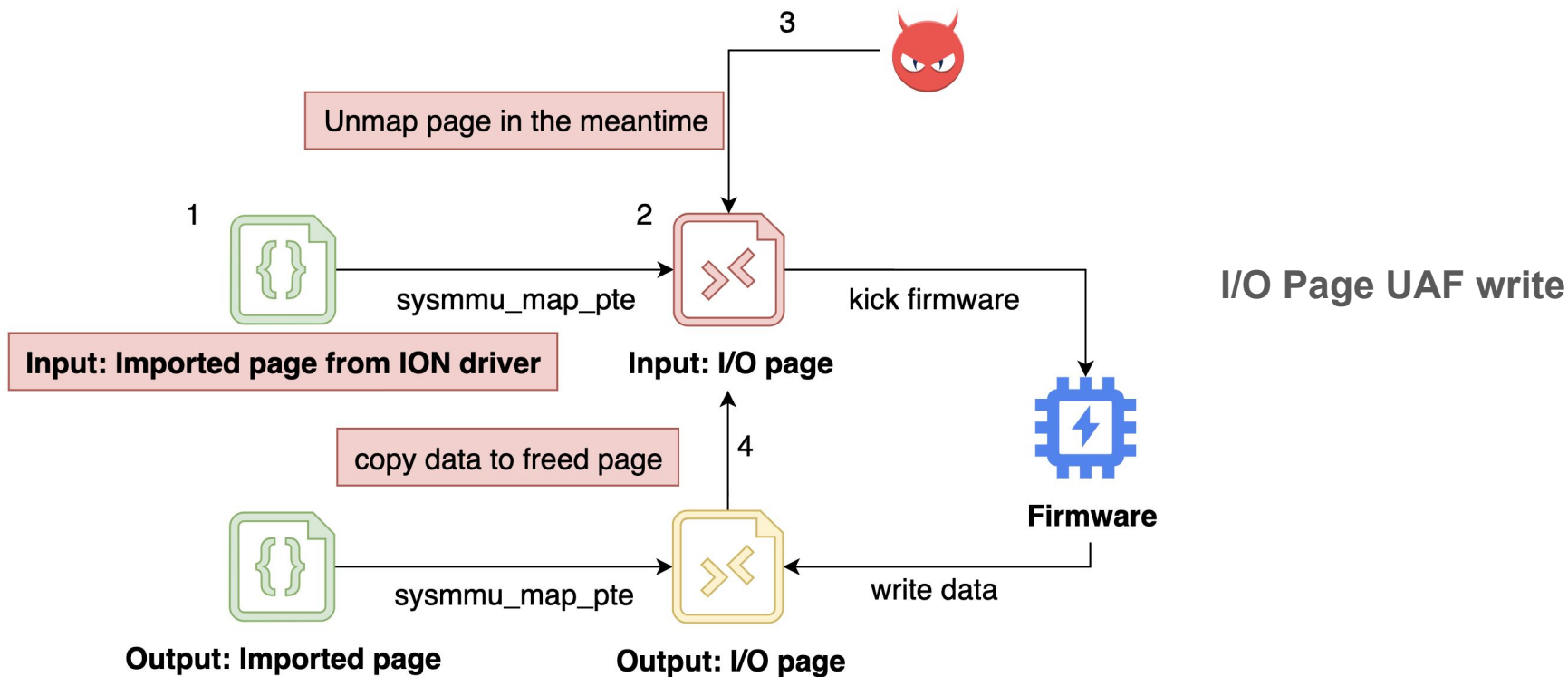
    do {
        ...
        *ent = mk_lv2ent_spage(pte_pfn(*pte) << PAGE_SHIFT);
        if (!pfnmap)
            get_page(pte_page(*pte));
        else
            mk_lv2ent_pfnmap(ent);
        [...]
    } while (pte++, addr += PAGE_SIZE, addr != end);

    pgtable_flush(ent_beg, ent);
err:
    pte_unmap_unlock(pte - 1, ptl);
    return ret;
}
```

The driver expects users to import a normal (non-VM_PFNMAP) memory.

If a malicious user imports a **VM_PFNMAP** pages to the Exynos driver, the page reference count is **not** elevated.

CVE-2024-44068: Exynos In the Wild Oday



CVE-2024-44068: Exynos In the Wild Oday

- Send a stack of user pages, unmap PFNMAP pages in the meantime
 - How to win race condition with better chance?

Stack of Input User Pages



CVE-2024-44068: Exynos In the Wild Oday

- `mincore` checks whether the virtual page's underlying physical page is allocated or not

```
do {  
    mincore(m2m1shot_task.buf_cap.plane[1].userptr, 0x1000, vec);  
} while (!vec[0]);  
munmap(m2m1shot_task.buf_cap.plane[0].userptr, 0x8000);
```

- Exploit Flow
 - Spray page tables
 - Use page UAF write primitive to build KSMA
 - Overwrite kernel code



Challenges in Securing Driver mmap Code

Challenges in Securing Driver mmap code

- Binder CVE-2023-20928: Race Condition (P0)

```
@@ -213,7 +213,7 @@ static int binder_update_page_range(struct binder_alloc *alloc, int allocate,
    mm = alloc->vma_vm_mm;

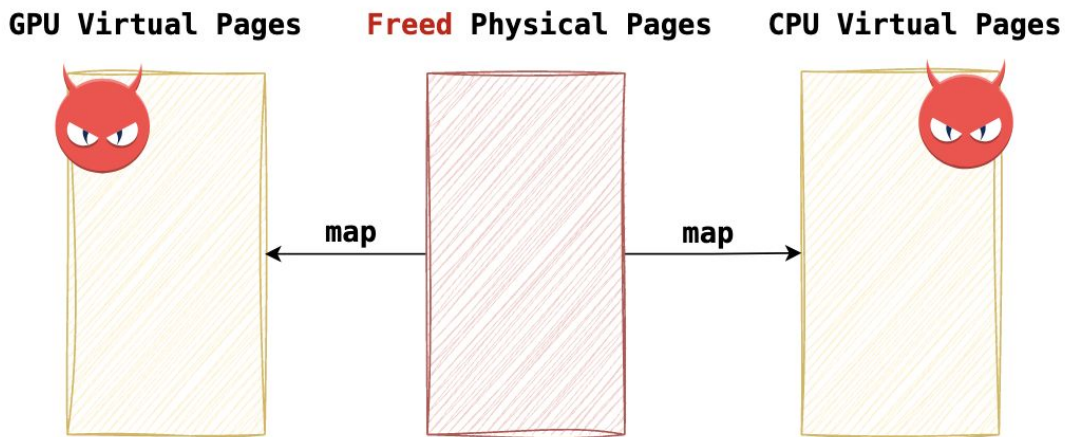
    if (mm) {
-        mmap_read_lock(mm);
+        mmap_write_lock(mm);
        vma = alloc->vma;
    }
```

- CVE-2022-33917: when VM_IO is not present (P0)

```
static void kbase_csf_user_io_pages_vm_open(struct vm_area_struct *vma) {
    WARN(1, "Unexpected attempt to clone private vma\n");
    vma->vm_private_data = NULL;
}
```

Challenges in Securing Driver mmap code

- Create a CPU mapping by **mmap**
- GPU logic bugs
 - Shrinker, Sparse Texture Memory, ...



Challenges in Securing Driver mmap code

- mmap read-only pages != read-only (IMG GPU CVE-2024-34726 P0)
 - No VM_WRITE flag != No VM_MAYWRITE flag
 - `void *ptr = mmap(NULL, <len>, PROT_READ, MAP_SHARED, <fd>, <offset>);`
 - *`mprotect(ptr, <len>, PROT_READ|PROT_WRITE)`*
- mmap read-only pages != read-only
 - **Dirty Ptrace**



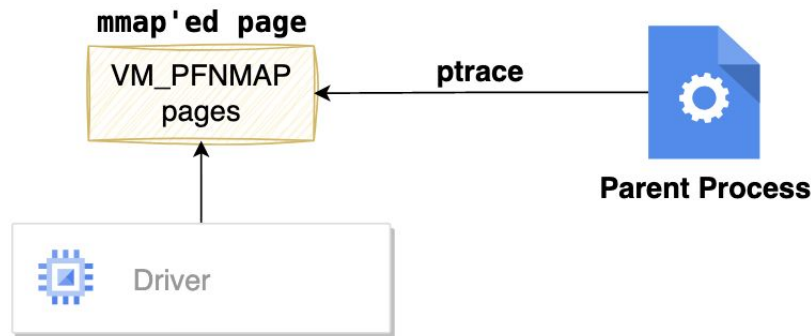
Dirty Ptrace

“Special” Read-Only Memory Overwrite

Dirty Ptrace: Special Read-Only Memory Overwrite

- Ask a (non-root) debugger to write on read-only memory
 - 1) Read-only memory by `mmap` with `MAP_PRIVATE`?
 - 2) Read-only memory by `mmap` with `MAP_SHARED`?

- Any exceptions?



Dirty Ptrace: Special Read-Only Memory Overwrite

- `ptrace(PTRACE_POKEDATA, ...)` to read / write on the PFN pages ❌
- The `access` callback

```
struct vm_area_struct *vma
```

the vma to access

```
unsigned long addr
```

userspace address, not relative offset within **vma**

```
void *buf
```

buffer to read/write

```
int len
```

length of transfer

```
int write
```

set to `FOLL_WRITE` when writing, otherwise reading

```
static int vma_access(struct vm_area_struct *vma, unsigned long addr,
    void *buffer, int length, int write) {
    ...
    if (write) {
        memcpy(kernel_obj_mapping, buffer, length);
    } else {
        memcpy(buffer, kernel_obj_mapping, length);
    }
}
```

Dirty Ptrace: Special Read-Only Memory Overwrite

- `ptrace(PTRACE_POKEDATA, ...)` -> `write_int` -> `access_process_vm` -> `__access_remote_vm`

```
static inline int
write_int(struct task_struct *task, unsigned long addr, int data)
{
    int copied = access_process_vm(task, addr, &data, sizeof(int),
                                   FOLL_FORCE | FOLL_WRITE);
    return (copied == sizeof(int)) ? 0 : -EIO;
}
```

```
/*
 * Access another process' address space.
 * Source/target buffer must be kernel space,
 * Do not walk the page table directly, use get_user_pages
 */
int access_process_vm(struct task_struct *tsk, unsigned long addr,
                     void *buf, int len, unsigned int gup_flags)
{
    struct mm_struct *mm;
    int ret;

    mm = get_task_mm(tsk);
    if (!mm)
        return 0;

    ret = __access_remote_vm(mm, addr, buf, len, gup_flags);

    mmput(mm);

    return ret;
}
```

Dirty Ptrace: Special Read-Only Memory Overwrite

- `__access_remote_vm` calls `get_user_page_vma_remote`.
- For regular VMA, it makes sure it's impossible to write on read-only memory.

```
int __access_remote_vm(struct mm_struct *mm, unsigned long addr, void *buf,
                      int len, unsigned int gup_flags) {
    ...
    while (len) {
        ...
        struct vm_area_struct *vma = NULL;
        struct page *page = get_user_page_vma_remote(mm, addr,
                                                    gup_flags, &vma);

        if (IS_ERR_OR_NULL(page)) {
            /* We might need to expand the stack to access it */
            vma = vma_lookup(mm, addr);
            if (!vma) {
                ...
            }
        }
    }
}
```

Dirty Ptrace: Special Read-Only Memory Overwrite

- `get_user_page_vma_remote` ->
`get_user_pages_remote` ->
`get_user_pages_locked`
- If `VM_PFNMAP` flag is on,
`get_user_page_vma_remote`
returns an error.

```
static long __get_user_pages_locked(struct mm_struct *mm, unsigned long start,
                                   unsigned long nr_pages, struct page **pages,
                                   int *locked, unsigned int foll_flags)
{
    struct vm_area_struct *vma;
    bool must_unlock = false;
    unsigned long vm_flags;
    long i;

    ...

    for (i = 0; i < nr_pages; i++) {
        vma = find_vma(mm, start);
        if (!vma)
            break;

        /* protect what we can, including chardevs */
        if ((vma->vm_flags & (VM_IO | VM_PFNMAP)) ||
            !(vm_flags & vma->vm_flags))
            break;

        ...
    }
    ...
    return i ? : -EFAULT;
}
```

Dirty Ptrace: Special Read-Only Memory Overwrite

- Second chance
 - If the `vm_ops->access` callback is registered, invoke this callback.
- `CONFIG_HAVE_IOREMAP_PROT` is always enabled on Android.

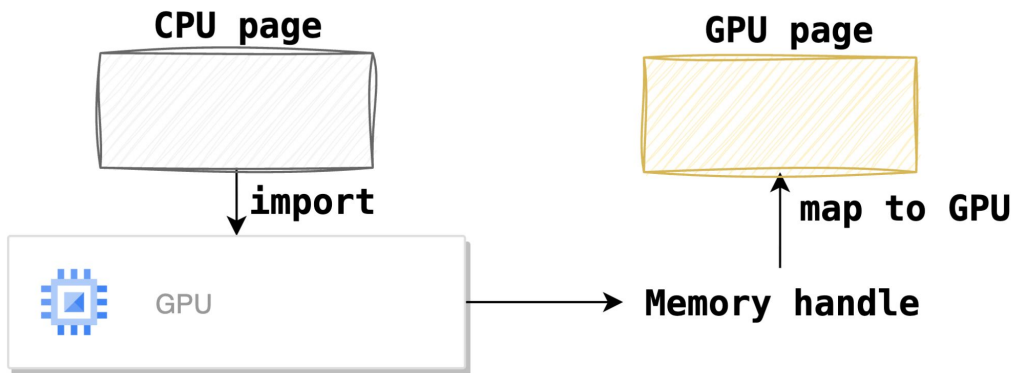
```
/*  
 * Check if this is a VM_IO | VM_PFNMAP VMA, which  
 * we can access using slightly different code.  
 */  
bytes = 0;  
#ifdef CONFIG_HAVE_IOREMAP_PROT  
    if (vma->vm_ops && vma->vm_ops->access)  
        bytes = vma->vm_ops->access(vma, addr, buf,  
                                    len, write);  
#endif  
if (bytes <= 0)  
    break;  
}  
...
```

Dirty Ptrace: Special Read-Only Memory Overwrite

- Linux Kernel Documentation from kernel.org
 - ->access() is called when `get_user_pages()` fails in `access_process_vm()`, typically used to debug a process through `/proc/pid/mem` or `ptrace`. This function is needed only for `VM_IO` | `VM_PFNMAP` VMAs.
- Driver developers assumes the kernel sanitizes the `write` argument
- The Linux kernel assumes the driver checks the `write` argument

Export Read-Only Memory From Drivers

- Drivers (e.g. GPU, TPU, NPU, DSP...) might import user space memory
 - ... and remap the memory back to the user space if possible
 - ***Some drivers may map read-only privileged memory to users.***
- Import texture memory from CPU and map to GPU virtual memory
 - Mali GPU CVE-2021-39793 (anonymous researcher), IMG GPU CVE-2024-43705 allows an unprivileged app to import libc.so and overwrite the read-only memory



Dirty Ptrace: Special Read-Only Memory Overwrite

- A kernel patch was proposed in the late 2024 by Jann Horn from P0 and T.J. from the Android kernel team
 - “It’s really a ... *uncommon* scenario”
 - “*only root user can invoke the **ptrace** syscall*”
 - “*Debugging feature*”

```
--- a/mm/memory.c
+++ b/mm/memory.c
@@ -6591,9 +6591,13 @@ static int __access_remote_vm(struct mm_struct
 *mm, unsigned long addr,
 */
bytes = 0;
#ifdef CONFIG_HAVE_IOREMAP_PROT
-    if (vma->vm_ops && vma->vm_ops->access)
+    if (vma->vm_ops && vma->vm_ops->access) {
+        if (write && !(vma->vm_flags & VM_WRITE))
+            break;
+
        bytes = vma->vm_ops->access(vma, addr, buf,
                                  len, write);
+    }
#endif
if (bytes <= 0)
break;
```



The Impact of “Dirty Ptrace” on Several GPU Drivers

Industry Security Advisory

Dirty Ptrace: Nvidia GPU CVE-2025-23244

- `ioctl -> ... -> RmValidateMmapRequest`
 - Unless you're a ROOT user, the driver will decide the page protection permission for users.
- E.g. If an unprivileged user wants to map the PCIe BAR memory, the driver only allows the user to mmap the memory with the ***read-only*** permission.

```
static NV_STATUS RmValidateMmapRequest(  
    nv_state_t *pNv,  
    NvU64      offset,  
    NvU64      length,  
    NvU32      *pProtection  
)  
{  
    ...  
    if (osIsAdministrator())  
    {  
        *pProtection = NV_PROTECT_READ_WRITE;  
        return NV_OK;  
    }  
  
    params.addressStart = offset;  
    params.addressLength = length;  
  
    status = pRmApi->Control(pRmApi, pNv->rmapi.hClient,  
                             pNv->rmapi.hSubDevice,  
                             NV2080_CTRL_CMD_GPU_VALIDATE_MEM_MAP_REQUEST,  
                             &params, sizeof(params));  
  
    if (status == NV_OK)  
    {  
        *pProtection = params.protection;  
    }  
  
    return status;  
}
```

Dirty Ptrace: Nvidia GPU CVE-2025-23244

- The `nvidia_vma_access` function does not validate the `write` argument passed from the Linux kernel.
- Unprivileged user can overwrite the mmap'ed read-only memory.

```
static int
nvidia_vma_access(
    struct vm_area_struct *vma,
    unsigned long addr,
    void *buffer,
    int length,
    int write
)
{
    nv_alloc_t *at = NULL;
    nv_linux_file_private_t *nvlfp = NV_GET_LINUX_FILE_PRIVATE(NV_VMA_FILE(vma));
    nv_state_t *nv = NV_STATE_PTR(nvlfp->nvptr);
    NvU32 pageIndex, pageOffset;
    void *kernel_mapping;
    const nv_alloc_mapping_context_t *mmap_context = &nvlfp->mmap_context;
    NvU64 offset;
    ...
    if (write)
        memcpy(kernel_mapping, buffer, length);
    else
        memcpy(buffer, kernel_mapping, length);
    ...
    return length;
}
```

Dirty Ptrace: Nvidia GPU CVE-2025-23244

- Unprivileged user could mmap read-only PCIe BAR memory
 - MMIO Registers
 - Nvidia GPU VRAM
 - ...

- Patch: Validate **write**



146 files changed +53946 -54730 lines changed

kernel-open/nvidia/nv-mmap.c

```
143 143         return -EINVAL;
144 144     }
145 145
146 +     if (write && !(mmap_context->prot & NV_PROTECT_WRITEABLE))
147 +     {
148 +         return -EACCES;
149 +     }
150 +
146 151     if (nv->flags & NV_FLAG_CONTROL)
147 152     {
```

Dirty Ptrace: Nvidia GPU CVE-2025-23244

- CWE-863: Confused Deputy
- Are there any other GPUs implement the `.access` callback?
 - ARM and Qualcomm: No `.access` callback
 - AMD, Intel and IMG: Implemented `.access` callback

NVIDIA GPU DISPLAY DRIVER

CVE ID	Description	Vector	Base Score	Severity	CWE	Impacts
CVE-2025-23244	NVIDIA GPU Display Driver for Linux contains a vulnerability which could allow an unprivileged attacker to escalate permissions. A successful exploit of this vulnerability might lead to code execution, denial of service, escalation of privileges, information disclosure, and data tampering.	<u>AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H</u>	7.8	High	<u>CWE-863</u>	Code execution, denial of service, escalation of privileges, information disclosure, and data tampering

Intel GPU: No Impact

- Intel GPU allows users to import read-only memory to the driver.
- Fortunately, `vm_access` function implements the security check.

```
static int
vm_access(struct vm_area_struct *area, unsigned long addr,
          void *buf, int len, int write)
{
    ...
    if (i915_gem_object_is_readonly(obj) && write)
        return -EACCES;

    addr -= area->vm_start;
    if (range_overflows_t(u64, addr, len, obj->base.size))
        return -EINVAL;

    ...

    if (write) {
        memcpy(vaddr + addr, buf, len);
        __i915_gem_object_flush_map(obj, addr, len);
    } else {
        memcpy(buf, vaddr + addr, len);
    }
}
```

AMD GPU: No Impact

- AMD GPU does not allow the driver to re-map the memory originated from an user imported memory.
- A safer GPU memory mapping policy may save the day!

```
/**
 * amdgpu_verify_access - Verify access for a mmap call
 *
 * @bo: The buffer object to map
 * @filp: The file pointer from the process performing the mmap
 *
 * This is called by ttm_bo_mmap() to verify whether a process
 * has the right to mmap a BO to their process space.
 */
static int amdgpu_verify_access(struct ttm_buffer_object *bo, struct file *filp)
{
    struct amdgpu_bo *abo = ttm_to_amdgpu_bo(bo);

    if (amdgpu_ttm_tt_get_usermm(bo->ttm))
        return -EPERM;
    return drm_vma_node_verify_access(&abo->tbo.base.vma_node,
                                     filp->private_data);
}
```

IMG GPU: CVE-2025-8109

- The IMG GPU maps read-only memory to users
 - Shared driver information page
 - Shared shader memory pages
- Android processes access the same shared read-only pages

```
eError = PMR_WriteBytes(psDevInfo->hTQUSCSharedMem, 0, RGXShaderUSCMem(psShaderFW)
if (eError != PVRSRV_OK)
{
    PVR_LOG("%s: Unexpected error from PMR_WriteBytes (%s)",
            __func__,
            PVRSRVGetErrorString(eError));
    goto failed_uscpmr;
}
```

Initialization of the Shared Memory

IMG GPU: CVE-2025-8109

- IMG GPU allows users to ptrace these global shared read-only pages
 - E.g. we can overwrite the shared shader memory
- Android processes trust the shared shader memory
 - Craft OOB access in the com.android.systemui process
 - Memory corruptions can be triggered when process calls

SkSurface::flushAndSubmit

```
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x000000723ecf23d0
x0 0000000000000001 x1 0000000000000000 x2 0000000000000000 x3 0000000000000000
x4 000000709b35017c x5 0000000000000004 x6 0000000000000000 x7 0000000000000004
x8 0000000000200008 x9 00000071bc4ca150 x10 0000000000000000 x11 0000000041414141
x12 0000000020a0a0a0 x13 0000000000000000 x14 0000000000000000 x15 0000000000000000
x16 0000000000000000 x17 0000000000000000 x18 000000709a650000 x19 000000709b353214
x20 000000709b350068 x21 b400007097f2f800 x22 0000000000000038 x23 000000709b3553f0
x24 000000709b350078 x25 000000709b3501e0 x26 000000709b3553f0 x27 000000709b350058
x28 0000000000000000 x29 000000709b34f750
lr 0000007109ed3490 sp 000000709b34f250 pc 0000007109ed35d4 pst 0000000060000000
backtrace:
#00 pc 0000000000485d4 /vendor/lib64/libsrv_um.so (BuildId: eeace5e8935a0cc3be4450bf61018ab2)
#01 pc 0000000000448d0 /vendor/lib64/libsrv_um.so (BuildId: eeace5e8935a0cc3be4450bf61018ab2)
#02 pc 00000000005272c /vendor/lib64/libsrv_um.so (RGQueueTransfer+5300) (BuildId: eeace5e8935a0cc3be4450bf61018ab2)
#03 pc 00000000003c0c0 /vendor/lib64/libIMGegl.so (TQQueueTransfer+1184) (BuildId: b6c4b56facd8bfb491918b03e6c)
#04 pc 000000000013d180 /vendor/lib64/egl/libGLESv2_mtk.so (BuildId: 0e02c8f1e2225ac94e0860c49d05345a)
#05 pc 000000000013e360 /vendor/lib64/egl/libGLESv2_mtk.so (BuildId: 0e02c8f1e2225ac94e0860c49d05345a)
#06 pc 0000000000146fa4 /vendor/lib64/egl/libGLESv2_mtk.so (BuildId: 0e02c8f1e2225ac94e0860c49d05345a)
#07 pc 000000000014ace0 /vendor/lib64/egl/libGLESv2_mtk.so (BuildId: 0e02c8f1e2225ac94e0860c49d05345a)
#08 pc 000000000015d194 /vendor/lib64/egl/libGLESv2_mtk.so (BuildId: 0e02c8f1e2225ac94e0860c49d05345a)
#09 pc 0000000000b1f8c /vendor/lib64/egl/libGLESv2_mtk.so (glDrawArrays+1312) (BuildId: 0e02c8f1e2225ac94e0860c49d05345a)
#10 pc 00000000008fad98 /system/lib64/libhwui.so ((anonymous namespace)::TextureOpImpl::onExecute(GrOpFlushState*)+100) (BuildId: 0e02c8f1e2225ac94e0860c49d05345a)
#11 pc 00000000008e2ad8 /system/lib64/libhwui.so (skgpu::v1::OpsTask::onExecute(GrOpFlushState*)+808) (BuildId: 0e02c8f1e2225ac94e0860c49d05345a)
```

IMG GPU: CVE-2024-49739

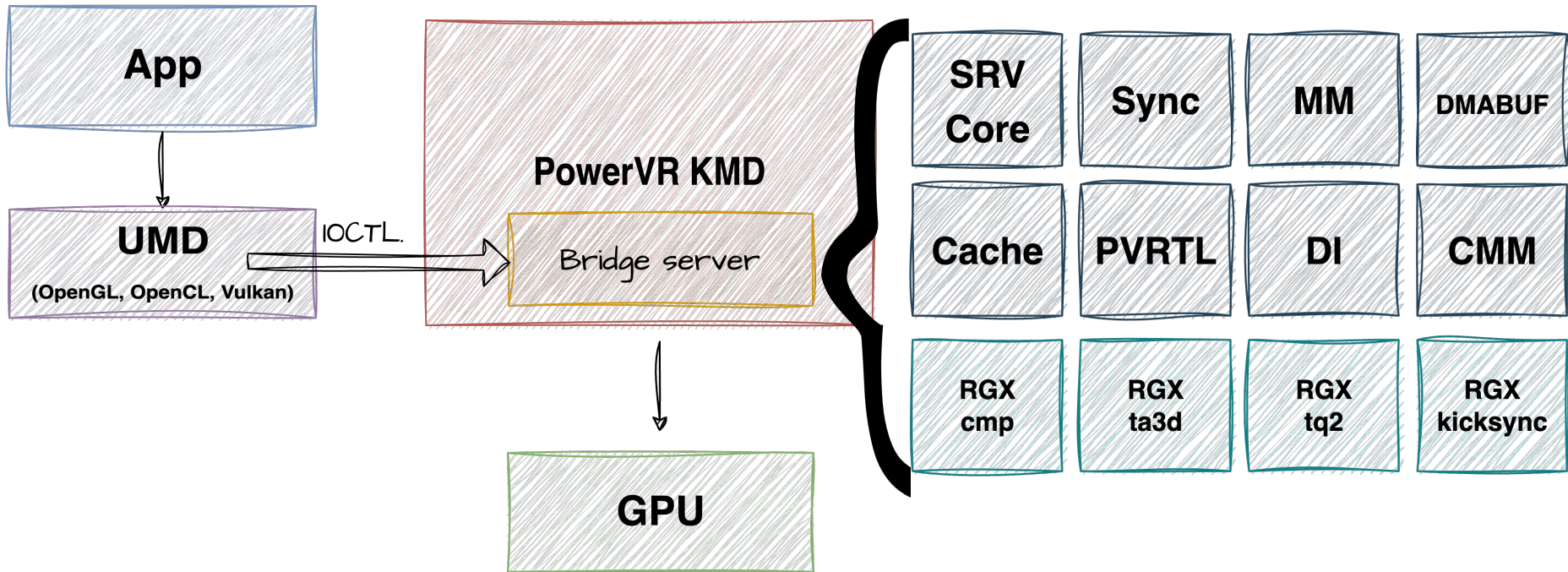
- The IMG GPU allows users to import read-only memory and re-maps the memory to userspace
- No sanitization check against “Dirty Ptrace”

```
// _CopyBytesExtMem is invoked by the .access callback
static PVRSRV_ERROR _CopyBytesExtMem(..., bWrite)
{
    ...
    if (bWrite)
    {
        OSDeviceMemCopy(sKernMapData.pvKernLinAddr + uiCopyOffset,
                        pcBuffer,
                        uiBytesPerLoop);
    }
    else
    {
        OSDeviceMemCopy(pcBuffer,
                        sKernMapData.pvKernLinAddr + uiCopyOffset,
                        uiBytesPerLoop);
    }
}
```

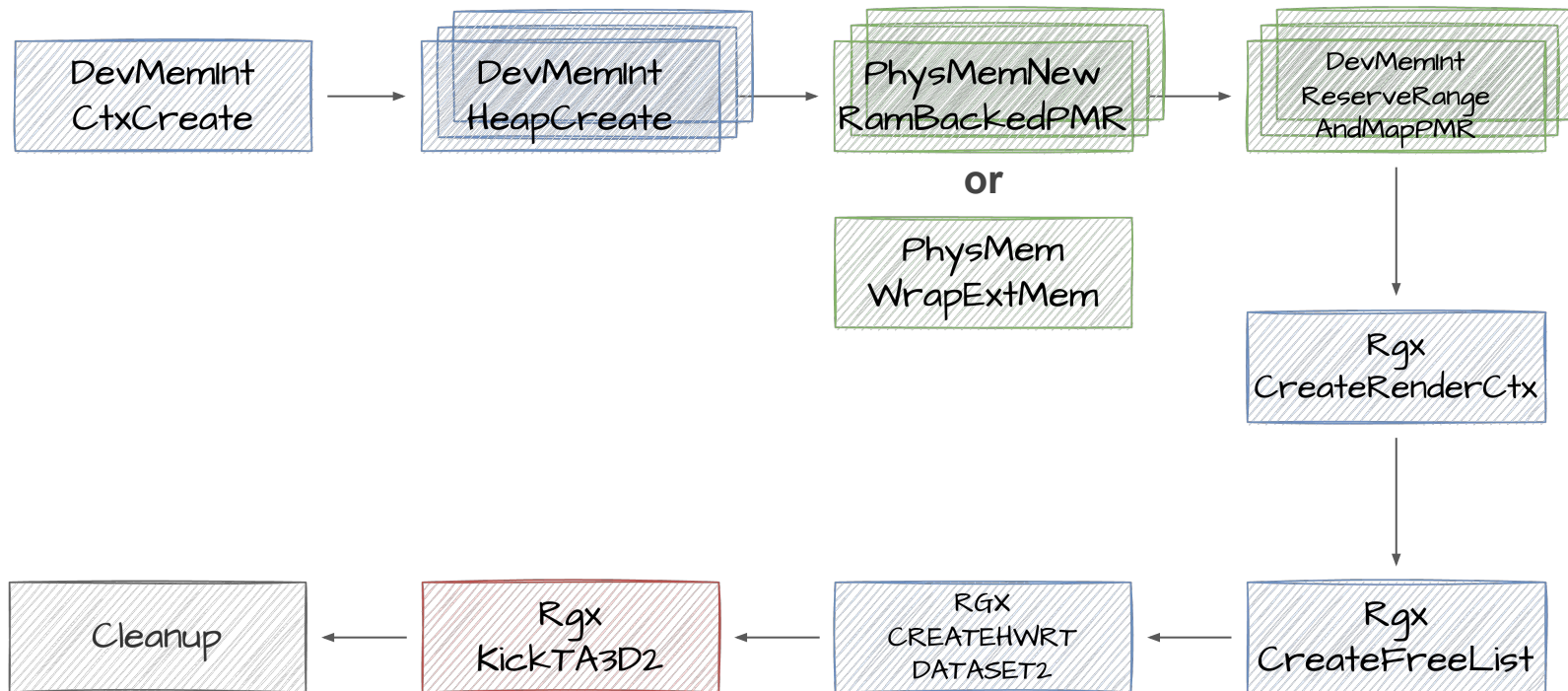


A ~~deep~~ shallow dive into the IMG DXT GPU on Pixel 10

Pixel 10 PowerVR Driver Stack



Pixel 10 PowerVR Driver Stack - Simplified 3D rendering



PhysMemWrapExtMem

In

```
typedef struct PVRSRV_BRIDGE_IN_PHYSMEMWRAPEXTMEM
{
    IMG_UINT64 ui64CpuVAddr;
    IMG_DEVMEM_SIZE_T uiSize;
    PVRSRV_MEMALLOCFLAGS_T uiFlags;
};
```

IOCTL.

Out

```
typedef struct PVRSRV_BRIDGE_OUT_PHYSMEMWRAPEXTMEM
{
    IMG_HANDLE hPMRPtr;
    PVRSRV_ERROR eError;
} __packed;
```


PVRSRV_MEMALLOCFLAG_{GPU_READABLE, GPU_WRITEABLE,
CPU_READABLE, CPU_WRITEABLE}


PMR handle can be mapped using MMAP

PhysMemWrapExtMem

In

```
typedef struct PVRSRV_BRIDGE_IN_PHYSMEMWRAPEXTMEM
{
    IMG_UINT64 ui64CpuVAddr;
    IMG_DEVMEM_SIZE_T uiSize;
    PVRSRV_MEMALLOCFLAGS_T uiFlags;
};
```

PVRSRV_MEMALLOCFLAG_{GPU_READABLE, GPU_WRITEABLE,
CPU_READABLE, CPU_WRITEABLE}

IOCTL.

Out

```
typedef struct PVRSRV_BRIDGE_OUT_PHYSMEMWRAPEXTMEM
{
    IMG_HANDLE hPMRPtr;
    PVRSRV_ERROR eError;
} __packed;
```

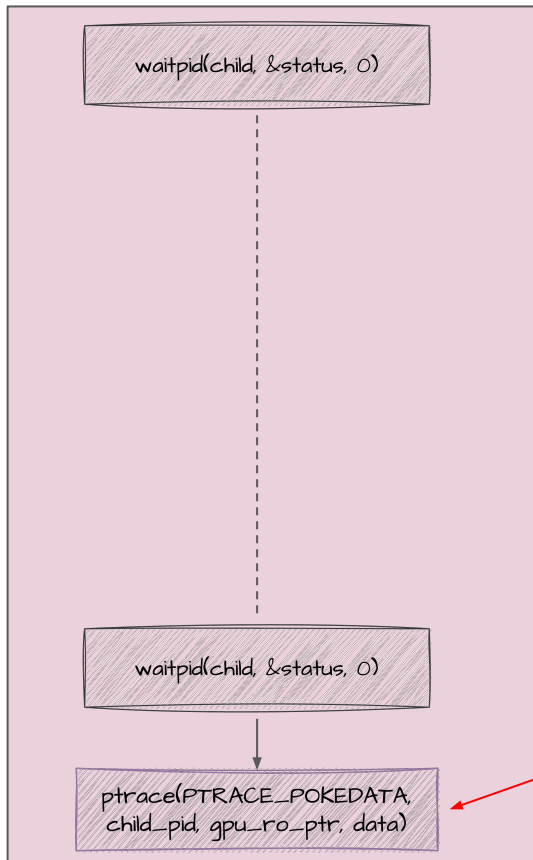
PMR handle can be mapped using MMAP



Exploiting Pixel 10*

Concept

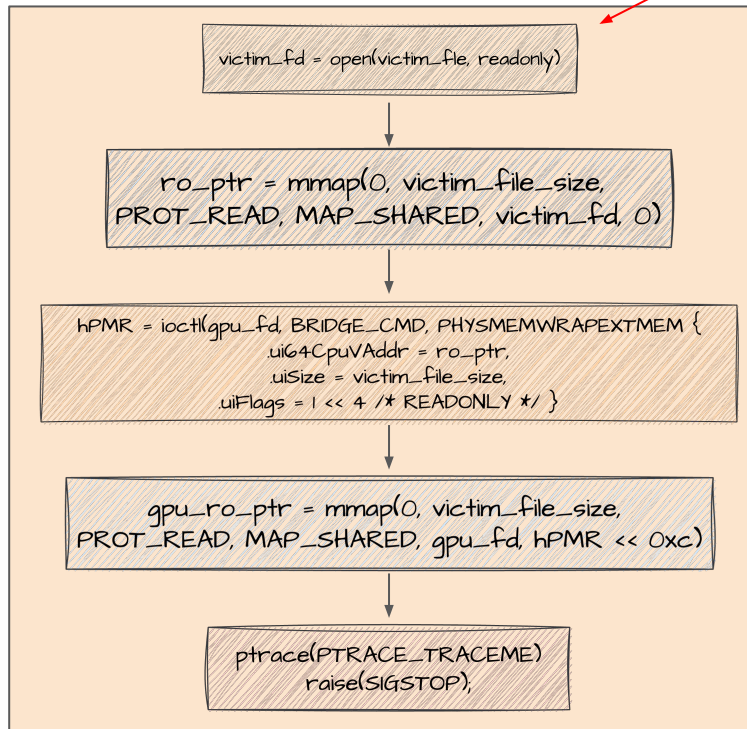
Parent
Process



`child_pid = fork();`

Child
Process

Any readable file...

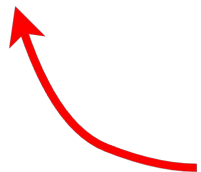


Path to root

What to overwrite?

- Libraries?
- System binaries?

→ Something ***highly*** privileged



Like **init** in DirtyPipe

Path to root

No access



/system/bin/init

```
PropertyServiceThread
while (1) {
    epoll(...)
    if (property_changed) {
        if (ShouldLog()) {
            ...
        }
        ... do work
    }
}
```

Read access



/system/lib64/libbase.so

```
00015770 android::base::ShouldLog(android::base::LogSeverity, char const*) ; ...
00015770 var_s0      = 0
00015770 var_s8      = 8
00015770 var_s10     = 0x10
00015770 var_s18     = 0x18
00015770 PACIASP
00015774 STP        X29, X30, [SP, #-0x20+var_s0]!
00015778 STP        X20, X19, [SP, #var_s10]!
0001577C MOV        X29, SP
00015780 MOV        X19, X1
00015784 MOV        W20, W0
00015788 MOV        W0, #0x1E
0001578C MOV        W1, WZR
00015790 MOV        W2, WZR
00015794 BL         __is05VersionAtLeast
00015798 CBZ        W0, loc_157C4
0001579C ADD        W0, W0, #0
000157A0 CMP        W0, #0
000157A4 MOV        W9, #7
000157A8 MOV        X1, X19
000157AC CSEL        W0, W0, W9, CC
000157B0 MOV        W2, #4
000157B4 BL         __android_log_is_loggable
000157B8 CMP        W0, #0
000157BC CSET        W0, NE
000157C0 B          loc_157D4
000157C4 loc_157C4   ; ...
000157C8 ADRP        X8, #android::base::gMinimumLogSeverity@PAGE
000157CC LDR        W0, [X8, #android::base::gMinimumLogSeverity@PAGEOFF]
000157D0 CMP        W20, W0
000157D4 CSET        W0, GE
000157D8 loc_157D4   ; ...
000157DC LDP        X20, X19, [SP, #var_s10]
000157E0 LDP        X29, X30, [SP+var_s0], #0x20
000157E4 RETAA
```

Payload

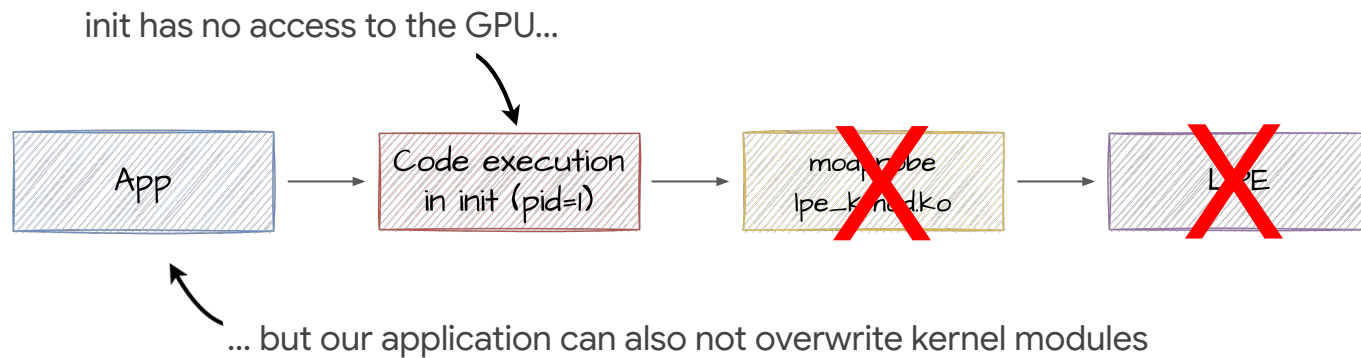
Plan

Step 1: overwrite ShouldLog in libbase.so

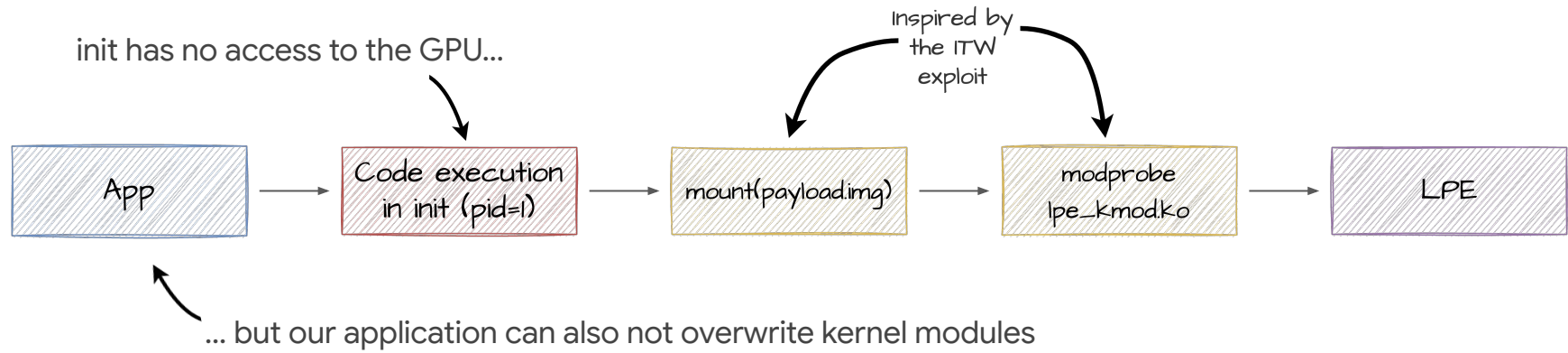
Step 2: ???

Step 3: Becoming root!

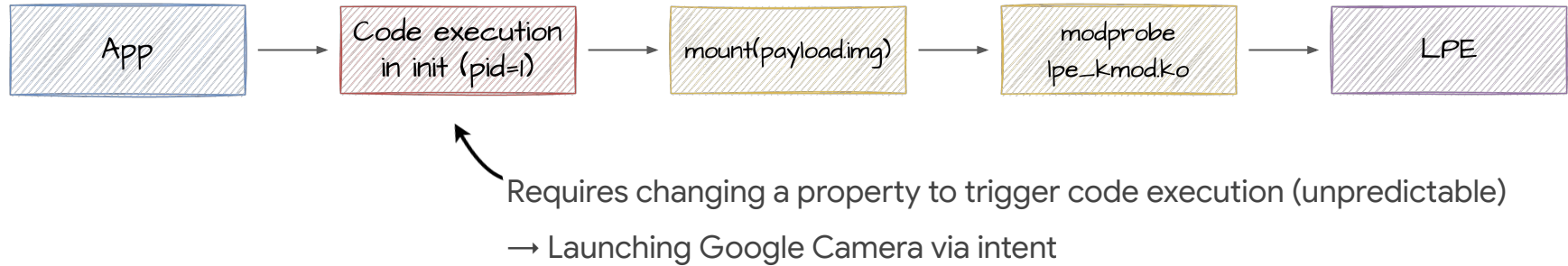
Plan



Plan



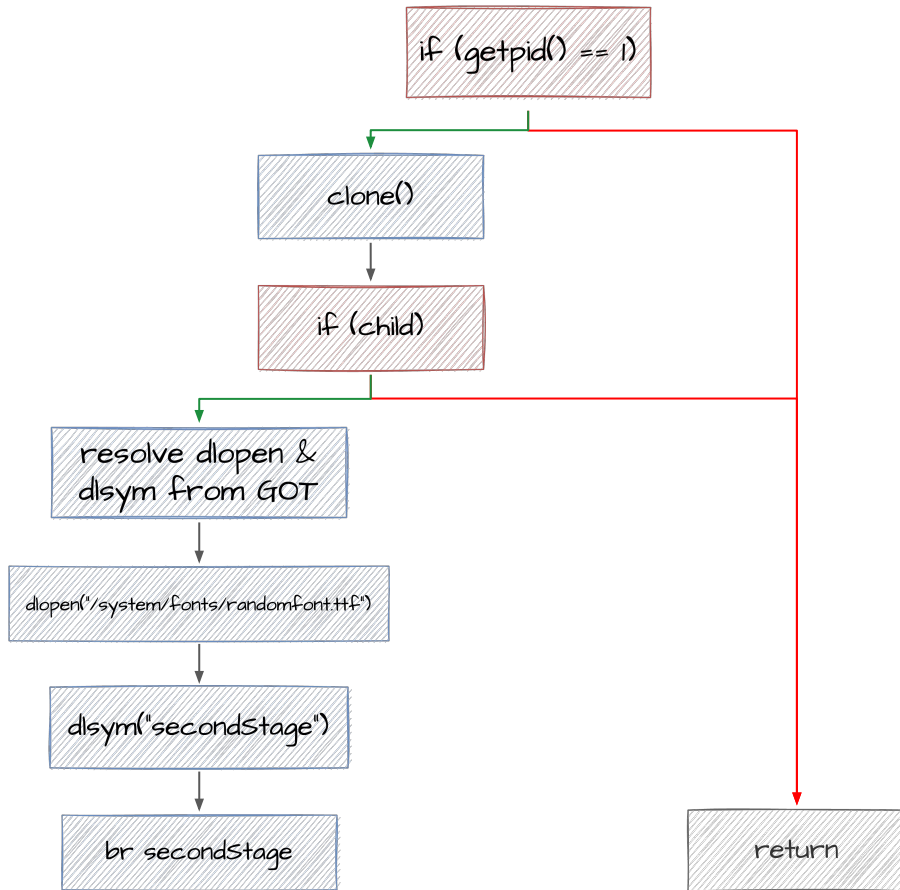
Plan



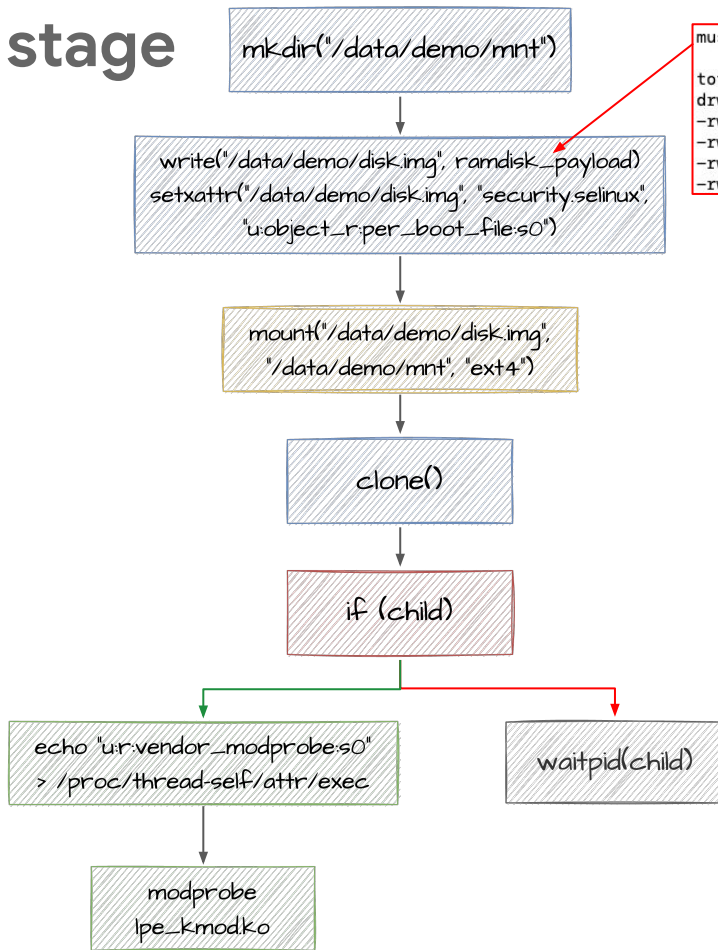
Initial payload

Runs in init (pid = 1, uid = 0)

Any RO system file
also accessible by
init is useable



Second stage



```
mustang:/data/poc2025/img # ls -lZ
```

```
total 208
drwx----- 2 root  root  u:object_r:unlabeled:s0  12288 2025-10-17 22:17 .lost+found
-rwxr-xr-x 1 system system u:object_r:vendor_file:s0 196048 2025-10-17 22:17 lpe_kmod.ko
-rw-r--r-- 1 system system u:object_r:vendor_file:s0 1 2025-10-17 22:17 modules.alias
-rw-r--r-- 1 system system u:object_r:vendor_file:s0 13 2025-10-17 22:17 modules.dep
-rw-r--r-- 1 system system u:object_r:vendor_file:s0 12 2025-10-17 22:17 modules.load
```

`read32(addr), read64(addr),
write32(addr), write64(addr),
lookup_symbol()`

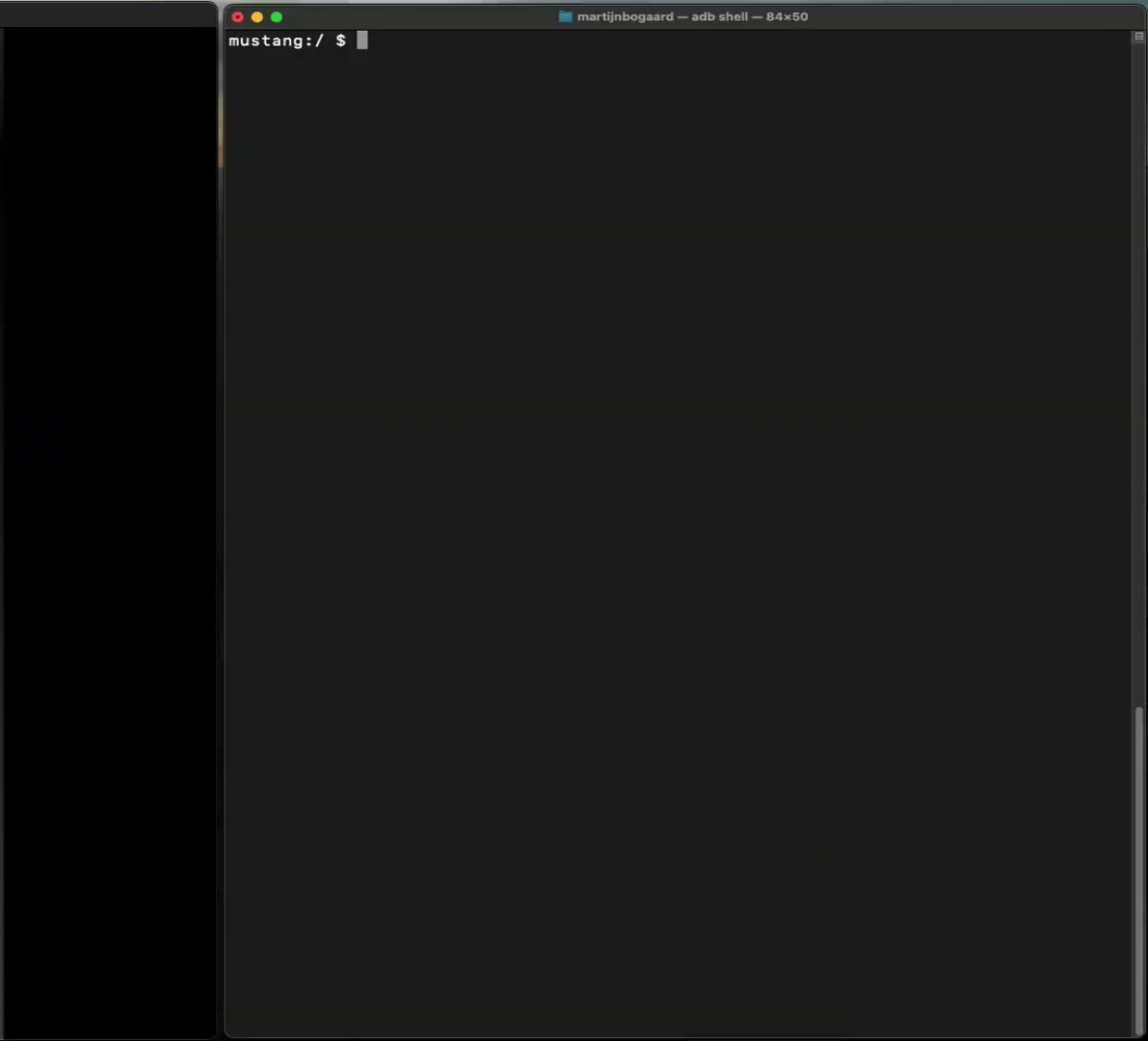
Final steps

Standard LPE:

- `selinux_enforcing = 0`
- Walk task list:
 - `task[sh].cred.uid/gid = 0`
 - `task[sh].cred.cap = 0xffffffff`



Demo





Thank you!



Questions? Read our blogs!

