JSOF

Racing Against the Lock: Exploiting Spinlock UAF in the Android Kernel

Moshe Kol

OffensiveCon 23

JSOF

You've found a use-after-free on **obj**:

```
spin_lock(&obj->lock);
// Nothing touches obj here.
spin_unlock(&obj->lock);
```

JSOF

You've found a use-after-free on **obj**:

```
spin_lock(&obj->lock);
// Nothing touches obj here.
spin_unlock(&obj->lock);
```

Can it be exploited?
Reliably? Generically?

About Me

- Moshe Kol (@0xkol), Security Researcher at Paragon;
  Former Security Researcher at JSOF

- Started with embedded security;

  Now focusing on the Android kernel

- M.Sc. in Computer Science from the Hebrew University

JSOF

1. CVE-2022-20421 (“Bad Spin”)
2. The Exploitation Technique
3. Demo Video

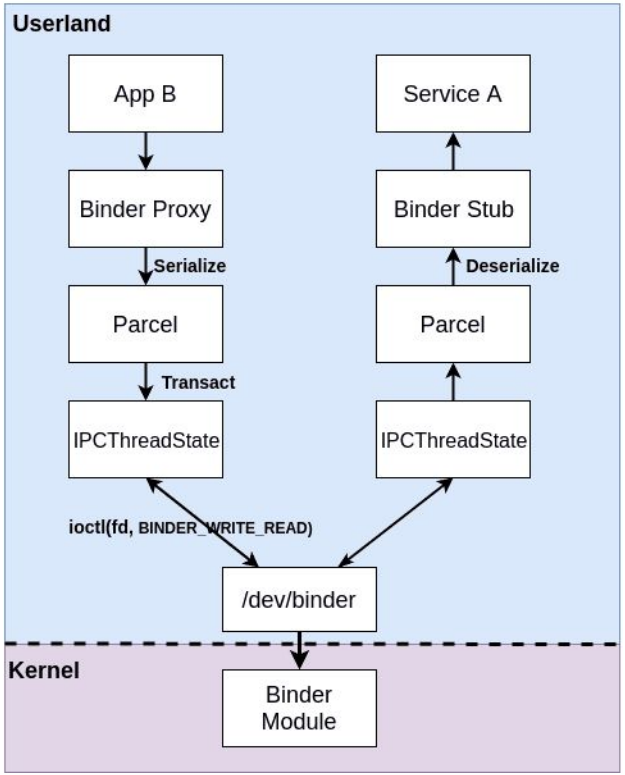
CVE-2022-20421 (“Bad Spin”)

- Race condition in Binder leading to UAF on `binder_proc`
- Reachable from `untrusted_app` SELinux context
- Closed on Android’s Security Bulletin of October 2022

CVE-2022-20421	A-239630375 Upstream kernel	EoP	High	Binder driver

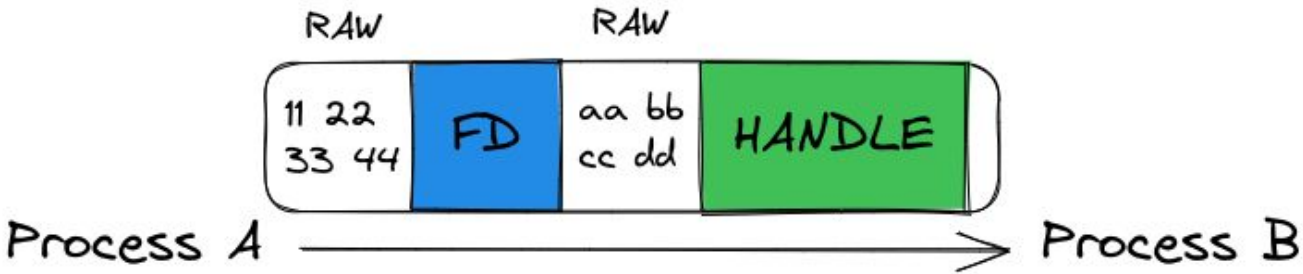
Binder

- IPC for applications and services on Android
- Advantageous in terms of security & performance
- Binder device (`/dev/binder` or `/dev/hwbinder`) is **accessible by all applications**



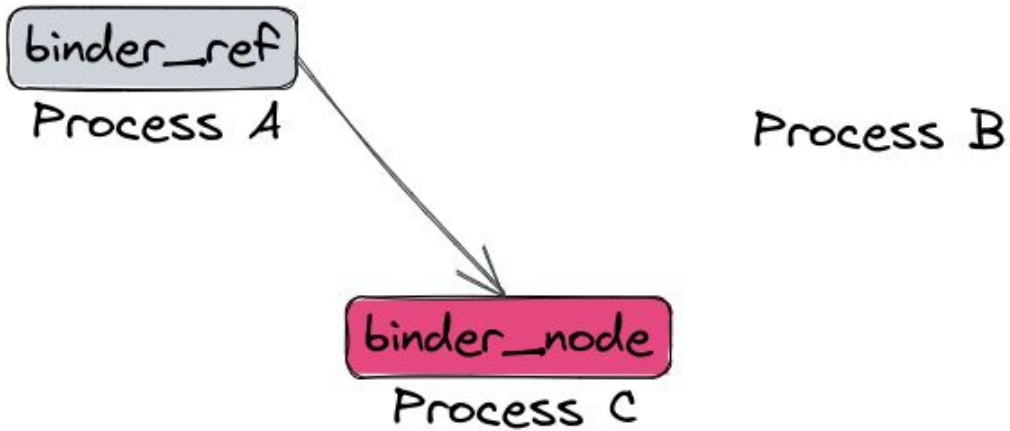
Binder Transactions

- In Binder, processes exchange **transactions**
- Transactions contain **raw data and objects**
- Kernel driver **translates** each object



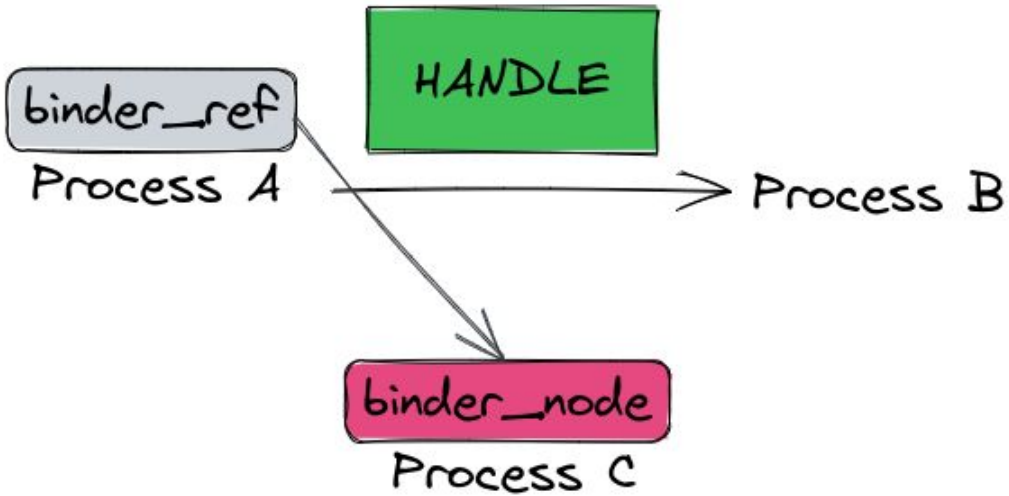
Binder Handles

- Allowing a process to share a handle with a peer



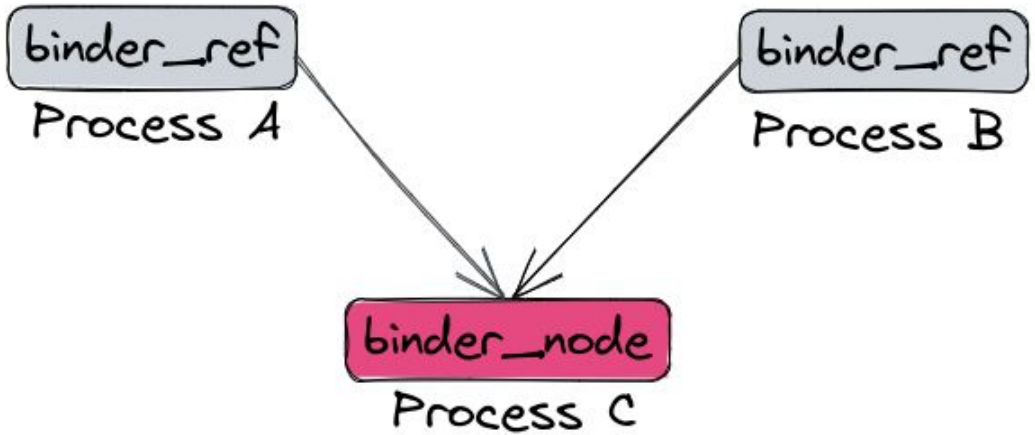
Binder Handles

- Allowing a process to share a handle with a peer



Binder Handles

- Allowing a process to share a handle with a peer



Translating Handles

Carried out in two steps:

1. Create a new Binder reference in the target process (if not already exists)

```
static int binder_inc_ref_for_node(struct binder_proc *proc,
                        struct binder_node *node,
                        bool strong,
                        struct list_head *target_list,
                        struct binder_ref_data *rdata)
{
        struct binder_ref *ref;
        struct binder_ref *new_ref = NULL;
        int ret = 0;

        binder_proc_lock(proc);
        ref = binder_get_ref_for_node_olocked(proc, node, NULL);
        if (!ref) {
                binder_proc_unlock(proc);
                new_ref = kzalloc(sizeof(*ref), GFP_KERNEL);
                if (!new_ref)
                        return -ENOMEM;
                binder_proc_lock(proc);
                ref = binder_get_ref_for_node_olocked(proc, node, new_ref);
        }
        ret = binder_inc_ref_olocked(ref, strong, target_list);
        *rdata = ref->data;
        binder_proc_unlock(proc);
        if (new_ref && ref != new_ref)
                /*
                 * Another thread created the ref first so
                 * free the one we allocated
                 */
                kfree(new_ref);
        return ret;
}
```

Translating Handles

Carried out in two steps:

1. Create a new Binder reference in the target process (if not already exists)
2. Taking a refcount on its Binder node

```
static int binder_inc_ref_for_node(struct binder_proc *proc,
                        struct binder_node *node,
                        bool strong,
                        struct list_head *target_list,
                        struct binder_ref_data *rdata)
{
        struct binder_ref *ref;
        struct binder_ref *new_ref = NULL;
        int ret = 0;

        binder_proc_lock(proc);
        ref = binder_get_ref_for_node_olocked(proc, node, NULL);
        if (!ref) {
                binder_proc_unlock(proc);
                new_ref = kzalloc(sizeof(*ref), GFP_KERNEL);
                if (!new_ref)
                        return -ENOMEM;
                binder_proc_lock(proc);
                ref = binder_get_ref_for_node_olocked(proc, node, new_ref);
        }
        ret = binder_inc_ref_olocked(ref, strong, target_list);
        *rdata = ref->data;
        binder_proc_unlock(proc);
        if (new_ref && ref != new_ref)
                /*
                 * Another thread created the ref first so
                 * free the one we allocated
                 */
                kfree(new_ref);
        return ret;
}
```

Translating Handles

Carried out in two steps:

1. Create a new Binder reference in the target process (if not already exists)
2. Taking a refcount on its Binder node

Bug: If the 2nd step fails, the new reference is not cleaned-up

```
static int binder_inc_ref_for_node(struct binder_proc *proc,
                        struct binder_node *node,
                        bool strong,
                        struct list_head *target_list,
                        struct binder_ref_data *rdata)
{
        struct binder_ref *ref;
        struct binder_ref *new_ref = NULL;
        int ret = 0;

        binder_proc_lock(proc);
        ref = binder_get_ref_for_node_olocked(proc, node, NULL);
        if (!ref) {
                binder_proc_unlock(proc);
                new_ref = kzalloc(sizeof(*ref), GFP_KERNEL);
                if (!new_ref)
                        return -ENOMEM;
                binder_proc_lock(proc);
                ref = binder_get_ref_for_node_olocked(proc, node, new_ref);
        }
        ret = binder_inc_ref_olocked(ref, strong, target_list);
        *rdata = ref->data;
        binder_proc_unlock(proc);
        if (new_ref && ref != new_ref)
                /*
                 * Another thread created the ref first so
                 * free the one we allocated
                 */
                kfree(new_ref);
        return ret;
}
```

Translating Handles

Carried out in two steps:

1. Create a new Binder reference in the target process (if not already exists)
2. Taking a refcount on its Binder node

Bug: If the 2nd step fails, the new reference is not cleaned-up

Normally, not a security issue – reference will be cleaned up in process exit

```
static int binder_inc_ref_for_node(struct binder_proc *proc,
                        struct binder_node *node,
                        bool strong,
                        struct list_head *target_list,
                        struct binder_ref_data *rdata)
{
        struct binder_ref *ref;
        struct binder_ref *new_ref = NULL;
        int ret = 0;

        binder_proc_lock(proc);
        ref = binder_get_ref_for_node_olocked(proc, node, NULL);
        if (!ref) {
                binder_proc_unlock(proc);
                new_ref = kzalloc(sizeof(*ref), GFP_KERNEL);
                if (!new_ref)
                        return -ENOMEM;
                binder_proc_lock(proc);
                ref = binder_get_ref_for_node_olocked(proc, node, new_ref);
        }
        ret = binder_inc_ref_olocked(ref, strong, target_list);
        *rdata = ref->data;
        binder_proc_unlock(proc);
        if (new_ref && ref != new_ref)
                /*
                 * Another thread created the ref first so
                 * free the one we allocated
                 */
                kfree(new_ref);
        return ret;
}
```

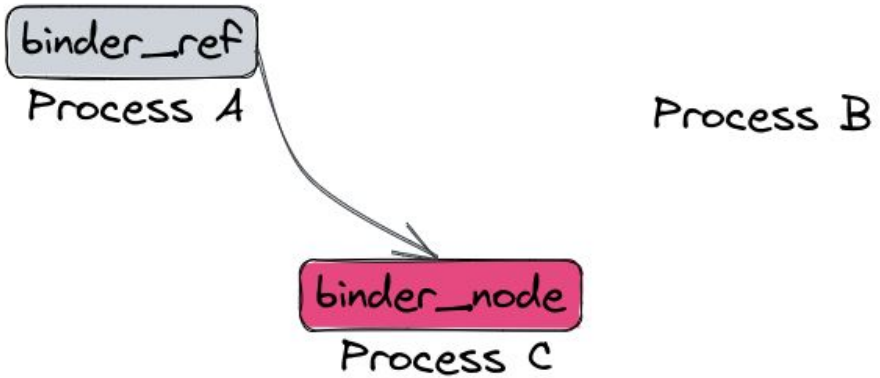
The Race

```
/* A */

int binder_transaction(...) {

     Allocate buffer in target process

     Copy transaction data




     Translate each object

     Schedule transaction

}
```

```
/* B */




exit(0)

> binder_deferred_release()

==> Cleanup all references
```

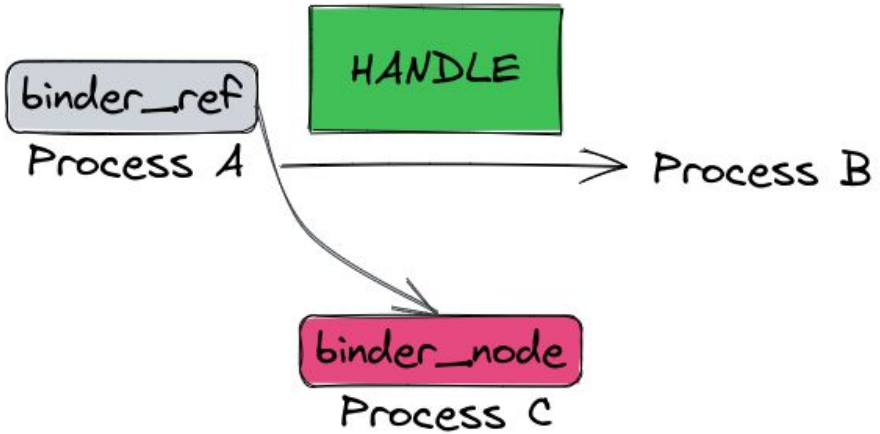
The UAF

0. Setup: ***A*** has reference to ***C***



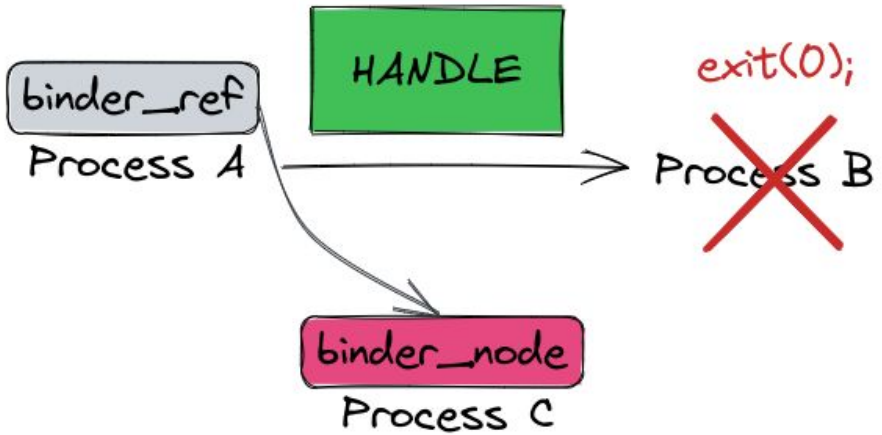
The UAF

0. Setup: ***A*** has reference to ***C***
1. ***A*** shares this reference with ***B***



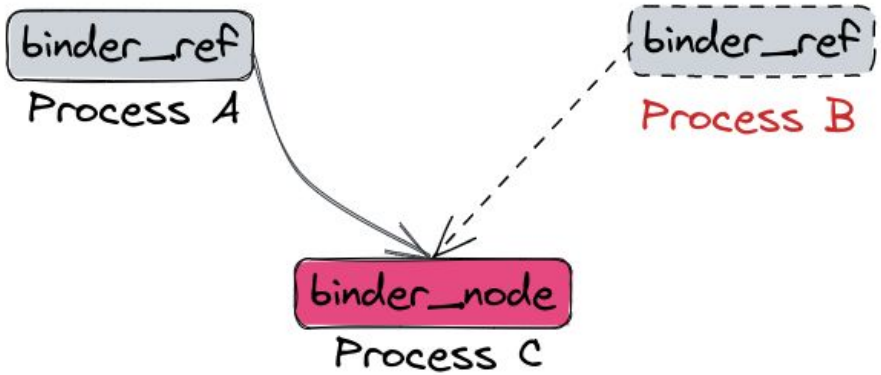
The UAF

0. Setup: ***A*** has reference to ***C***
1. ***A*** shares this reference with ***B***
2. ***B*** exit() in the middle



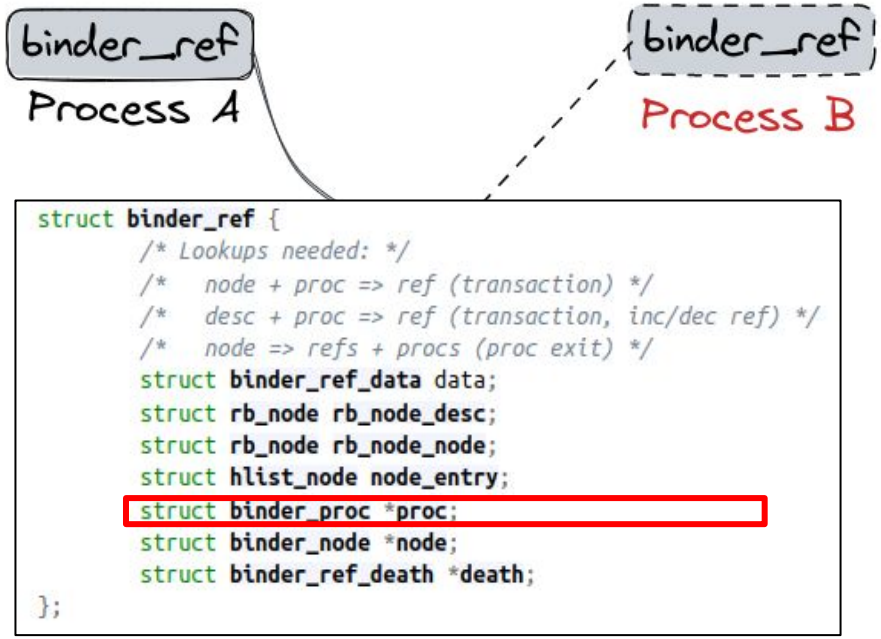
The UAF

0. Setup: ***A*** has reference to ***C***
1. ***A*** shares this reference with ***B***
2. ***B*** exit() in the middle
3. New reference to ***C*** is inserted to ***B***



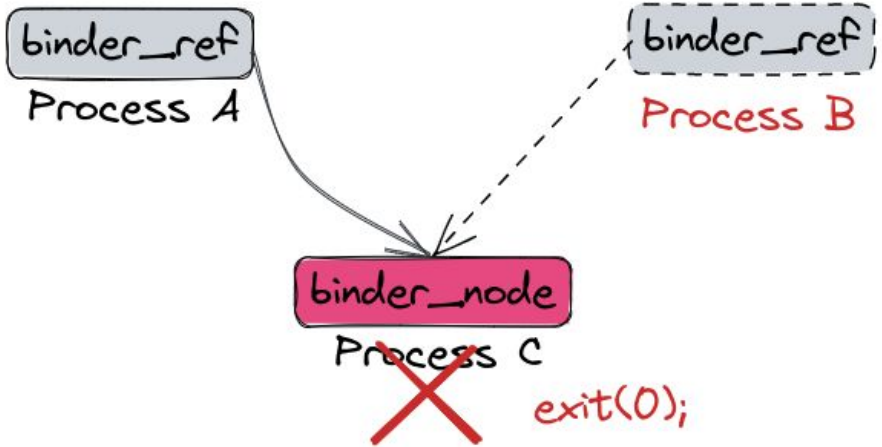
The UAF

0. Setup: ***A*** has reference to ***C***
1. ***A*** shares this reference with ***B***
2. ***B*** exit() in the middle
3. New reference to ***C*** is inserted to ***B***
    3.1. New reference contains a pointer to the owning process (B), which is now freed



The UAF

0. Setup: ***A*** has reference to ***C***
1. ***A*** shares this reference with ***B***
2. ***B*** exit() in the middle
3. New reference to ***C*** is inserted to ***B***
4. ***C*** exit(), anytime



The UAF

0. Setup: ***A*** has reference to ***C***
1. ***A*** shares this reference with ***B***
2. ***B*** exit() in the middle
3. New reference to ***C*** is inserted to ***B***
4. ***C*** exit(), anytime
    4.1. Releases all nodes

```
static void binder_deferred_release(struct binder_proc *proc)
{
    nodes = 0;
    incoming_refs = 0;
    while ((n = rb_first(&proc->nodes))) {
            struct binder_node *node;

            node = rb_entry(n, struct binder_node, rb_node);
            nodes++;
            /*
             * take a temporary ref on the node before
             * calling binder_node_release() which will either
             * kfree() the node or call binder_put_node()
             */
            binder_inc_node_tmpref_ilocked(node);
            rb_erase(&node->rb_node, &proc->nodes);
            binder_inner_proc_unlock(proc);
            incoming_refs = binder_node_release(node, incoming_refs);
            binder_inner_proc_lock(proc);
    }
    binder_inner_proc_unlock(proc);
```

The UAF

0. Setup: ***A*** has reference to ***C***
1. ***A*** shares this reference with ***B***
2. ***B*** exit() in the middle
3. New reference to ***C*** is inserted to ***B***
4. ***C*** exit(), anytime
    - 4.1. Releases all nodes
    - 4.2. Iterates over all references

```
static int binder_node_release(struct binder_node *node, int refs)
{
        hlist_for_each_entry(ref, &node->refs, node_entry) {
                refs++;
                /*
                 * Need the node lock to synchronize
                 * with new notification requests and the
                 * inner lock to synchronize with queued
                 * death notifications.
                 */
                binder_inner_proc_lock(ref->proc);
                if (!ref->death) {
                        binder_inner_proc_unlock(ref->proc);
                        continue;
                }

                death++;

                BUG_ON(!list_empty(&ref->death->work.entry));
                ref->death->work.type = BINDER_WORK_DEAD_BINDER;
                binder_enqueue_work_ilocked(&ref->death->work,
                                            &ref->proc->todo);
                binder_wakeup_proc_ilocked(ref->proc);
                binder_inner_proc_unlock(ref->proc);
        }
```

The UAF

0. Setup: ***A*** has reference to ***C***
1. ***A*** shares this reference with ***B***
2. ***B*** exit() in the middle
3. New reference to ***C*** is inserted to ***B***
4. ***C*** exit(), anytime
    - 4.1. Releases all nodes
    - 4.2. Iterates over all references
    - 4.3. UAF on ***B***'s `binder_proc`

```
static int binder_node_release(struct binder_node *node, int refs)
{
        hlist_for_each_entry(ref, &node->refs, node_entry) {
                refs++;
                /*
                 * Need the node lock to synchronize
                 * with new notification requests and the
                 * inner lock to synchronize with queued
                 * death notifications.
                 */
                binder_inner_proc_lock(ref->proc);
                if (!ref->death) {
                        binder_inner_proc_unlock(ref->proc);
                        continue;
                }

                death++;

                BUG_ON(!list_empty(&ref->death->work.entry));
                ref->death->work.type = BINDER_WORK_DEAD_BINDER;
                binder_enqueue_work_ilocked(&ref->death->work,
                                            &ref->proc->todo);
                binder_wakeup_proc_ilocked(ref->proc);
                binder_inner_proc_unlock(ref->proc);
        }
```

The UAF

0. Setup: ***A*** has reference to ***C***
1. ***A*** shares this reference with ***B***
2. ***B*** exit() in the middle
3. New reference to ***C*** is inserted to ***B***
4. ***C*** exit(), anytime
   - 4.1. Releases all nodes
   - 4.2. Iterates over all references
   - 4.3. UAF on ***B***'s `binder_proc`

Unreachable!

```
static int binder_node_release(struct binder_node *node, int refs)
{
        hlist_for_each_entry(ref, &node->refs, node_entry) {
                refs++;
                /*
                 * Need the node lock to synchronize
                 * with new notification requests and the
                 * inner lock to synchronize with queued
                 * death notifications.
                 */
                binder_inner_proc_lock(ref->proc);
                if (!ref->death) {
                        binder_inner_proc_unlock(ref->proc);
                        continue;
                }

                death++;

                BUG_ON(!list_empty(&ref->death->work.entry));
                ref->death->work.type = BINDER_WORK_DEAD_BINDER;
                binder_enqueue_work_ilocked(&ref->death->work,
                                            &ref->proc->todo);
                binder_wakeup_proc_ilocked(ref->proc);
                binder_inner_proc_unlock(ref->proc);
        }
```

The UAF

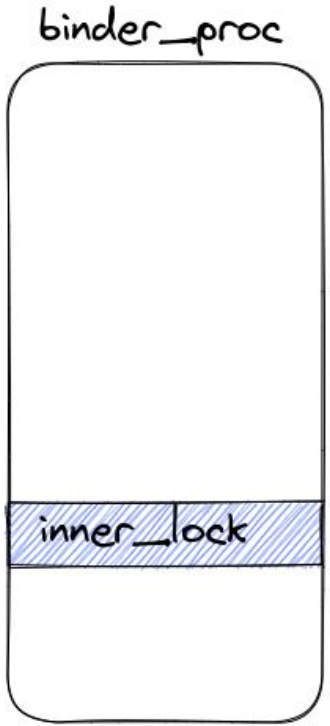
0. Setup: ***A*** has reference to ***C***
1. ***A*** shares this reference with ***B***
2. ***B*** exit() in the middle
3. New reference to ***C*** is inserted to ***B***
4. ***C*** exit(), anytime
   - 4.1. Releases all nodes
   - 4.2. Iterates over all references
   - 4.3. UAF on ***B***'s `binder_proc`
   - 4.4. `inner_lock` is a spinlock

```
static int binder_node_release(struct binder_node *node, int refs)
{
        hlist_for_each_entry(ref, &node->refs, node_entry) {
                refs++;
                /*
                 * Need the node lock to synchronize
                 * with new notification requests and the
                 * inner lock to synchronize with queued
                 * death notifications.
                 */
                binder_inner_proc_lock(ref->proc);
                if (!ref->death) {
                        binder_inner_proc_unlock(ref->proc);
                        continue;
                }
```

```
#define binder_inner_proc_lock(proc) _binder_inner_proc_lock(proc, __LINE__)
static void
_binder_inner_proc_lock(struct binder_proc *proc, int line)
        __acquires(&proc->inner_lock)
{
        binder_debug(BINDER_DEBUG_SPINLOCKS,
                     "%s: line=%d\n", __func__, line);
        spin_lock(&proc->inner_lock);
}
```

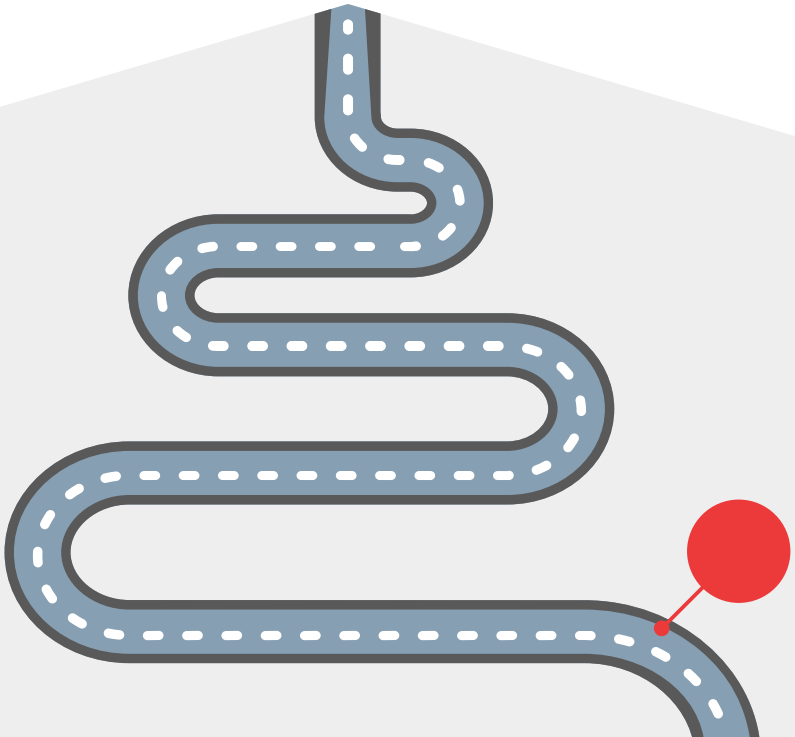
“Bad Spin” Summary

- Race condition that leads to **UAF on `binder_proc`**
  - We win it 100%
- `binder_proc` is allocated in **`kmalloc-1k`**
  - Relatively quiet
- We control the timing of the UAF
  - Since we control process ***C***



JSOF

Kernel R/W
+ kASLR bypass



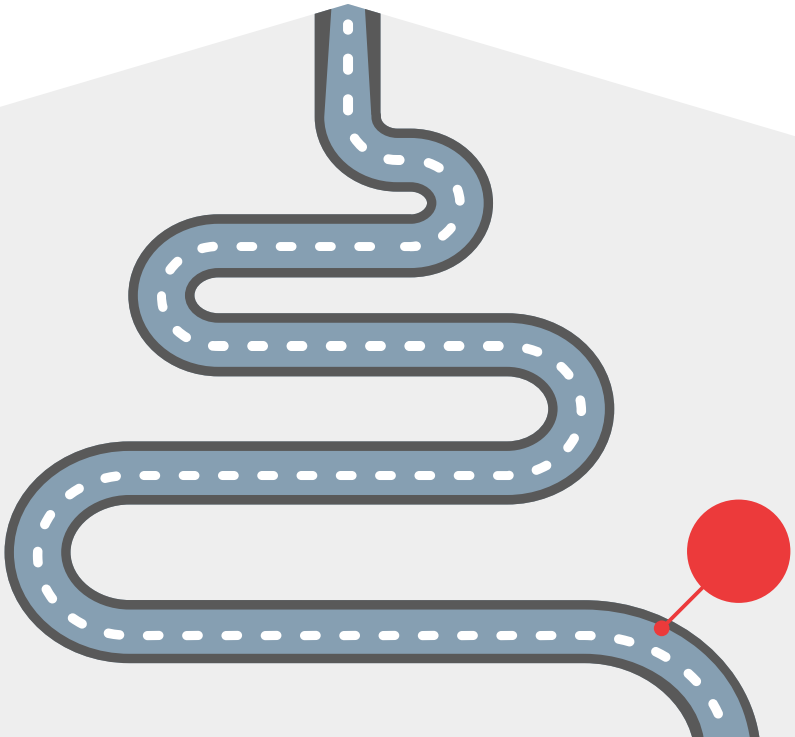
UAF Bug

```
spin_lock()
spin_unlock()
```

JSOF

Kernel R/W
+ kASLR bypass

+ Generic
+ Reliable



UAF Bug

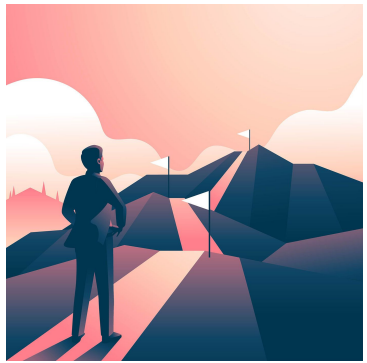
```
spin_lock()
spin_unlock()
```

Challenges

- Seems like a **weak primitive**: flipping a bit from 0 to 1 for a short moment

- Spinlocks **disable kernel preemption**, so winning the race is more difficult

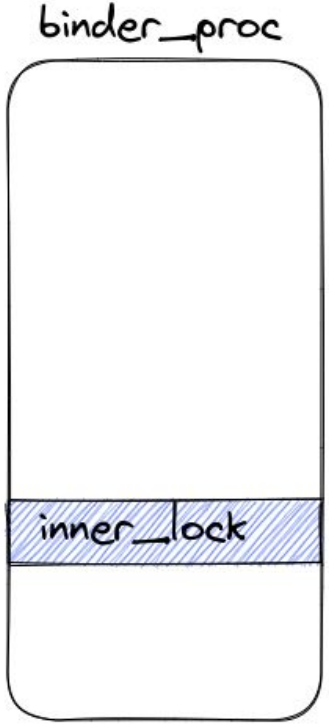
- `inner_lock` **offset varies between devices**:
  - 544 (Samsung S21 Ultra)
  - 576 (Samsung S22 and Google Pixel 6)
  - 584 (Google Pixel 6, Android 13)



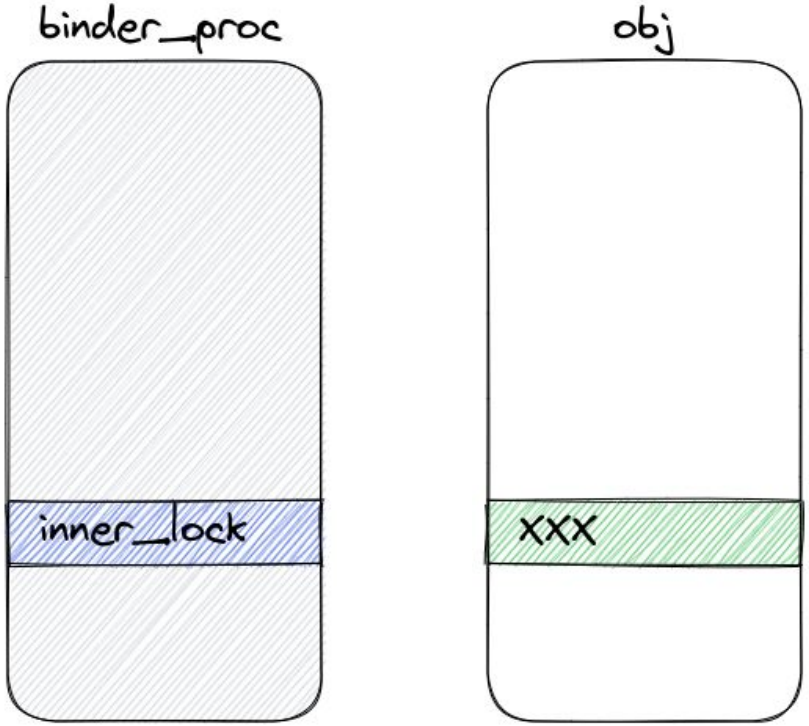
Our (Major) Assumptions

- Queued spinlock implementation (default since kernel 4.19)
- Lock offset divisible by 8
- `GFP_KERNEL` & `GFP_KERNEL_ACCOUNT` served from the same cache
  - By design on 5.10 kernels
  - For older kernels, true if kernel is booted with `cgroup.memory=nokmem`

Extracting Primitives

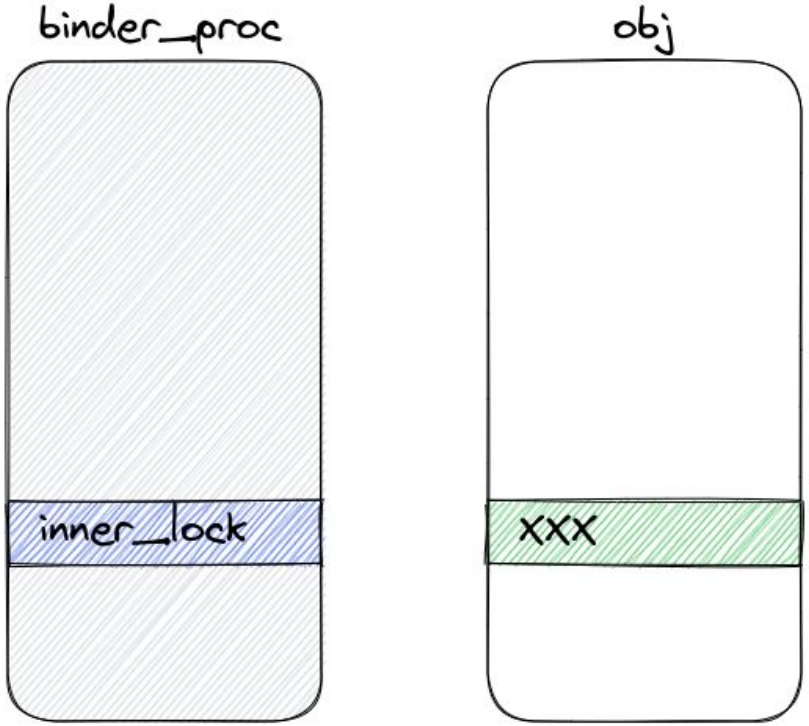


Extracting Primitives



Free `binder_proc` and reallocate as “obj”

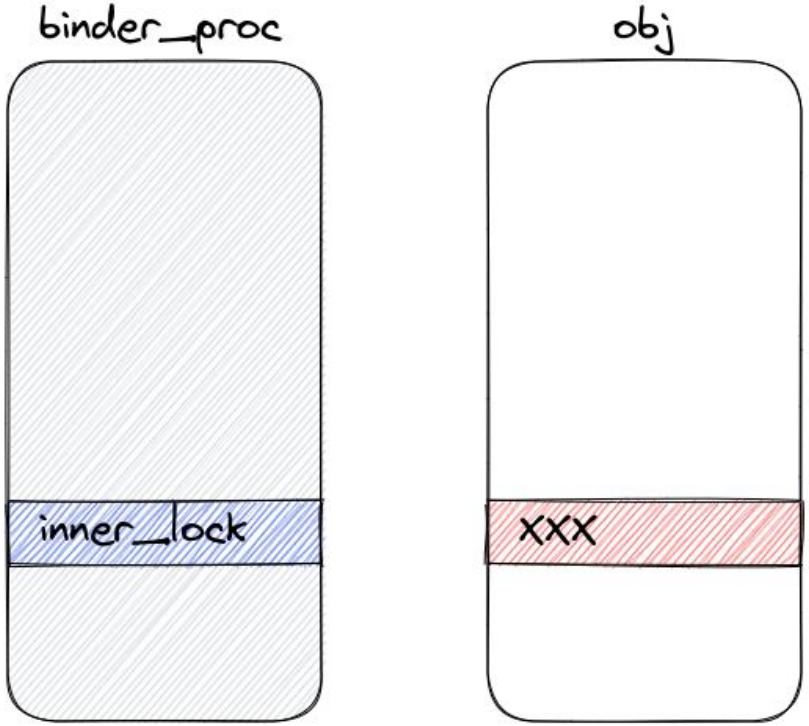
Extracting Primitives



Trigger UAF:

```
spin_lock(&proc->inner_lock)
spin_unlock(&proc->inner_lock)
```

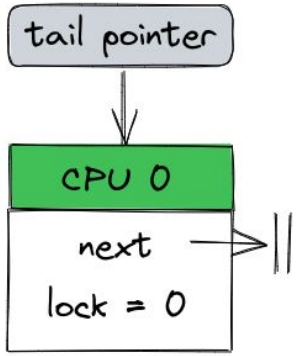
Extracting Primitives



`obj->xxx` changes in some interesting way

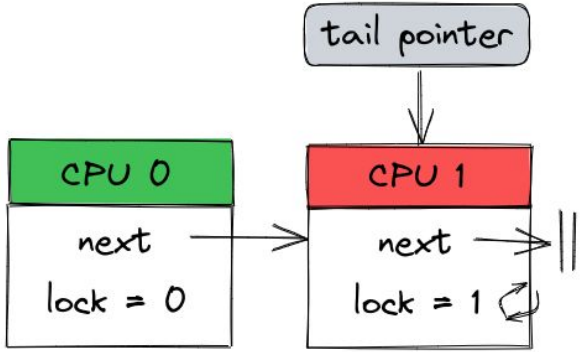
How Spinlocks are Implemented?

- **More complex** than you might think….
- “Queued Spinlock”: Designed to avoid starvation and improve cache utilization
- The lock maintains a queue, each CPU spins on its own accessible value



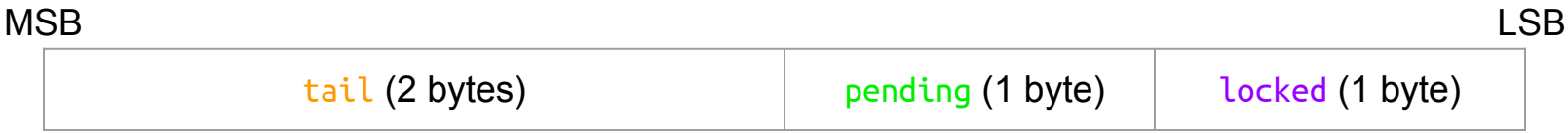
How Spinlocks are Implemented?

- **More complex** than you might think….
- “Queued Spinlock”: Designed to avoid starvation and improve cache utilization
- The lock maintains a queue, each CPU spins on its own accessible value



Implementation Details

- Spinlocks are 4-byte wide, broken as (`tail`, `pending`, `locked`)



- (0, 0, 0) means lock is released (ready to be acquired)

- `locked != 0` means other thread holds the lock

Implementation Details

```
void spin_lock(struct qspinlock *lock) {
     if (*(u32 *)lock == 0) {
          lock->locked = 1;
          return;
     }

     if (lock->tail != 0 || lock->pending != 0)
          goto queue;

     lock->pending = 1;
     while (lock->locked != 0);
     lock->pending = 0;
     lock->locked = 1;
     return;

queue: ...
}
void spin_unlock(struct qspinlock *lock) {
     lock->locked = 0;
}
```

```
void foo(struct foo *f){
     spin_lock(&f->lock);
     ... critical section ...
     spin_unlock(&f->lock);
}
```

foo.c

CPU 0	CPU 1

f->lock: 0x00000000

Implementation Details

```
void spin_lock(struct qspinlock *lock) {
      if (*(u32 *)lock == 0) {
            lock->locked = 1;
            return;
      }

      if (lock->tail != 0 || lock->pending != 0)
            goto queue;

      lock->pending = 1;
      while (lock->locked != 0);
      lock->pending = 0;
      lock->locked = 1;
      return;

queue: ...
}
void spin_unlock(struct qspinlock *lock) {
      lock->locked = 0;
}
```

```
void foo(struct foo *f){
      spin_lock(&f->lock);
      ... critical section ...
      spin_unlock(&f->lock);
}
```

foo.c

CPU 0	CPU 1
spin_lock(&f->lock)	
.. critical section ..	

f->lock: 0x00000001

Implementation Details

```
void spin_lock(struct qspinlock *lock) {
     if (*(u32 *)lock == 0) {
          lock->locked = 1;
          return;
     }

     if (lock->tail != 0 || lock->pending != 0)
          goto queue;

     lock->pending = 1;
     while (lock->locked != 0);
     lock->pending = 0;
     lock->locked = 1;
     return;

queue: ...
}
void spin_unlock(struct qspinlock *lock) {
     lock->locked = 0;
}
```

```
void foo(struct foo *f){
      spin_lock(&f->lock);
      ... critical section ...
      spin_unlock(&f->lock);
}
```

foo.c

CPU 0	CPU 1
spin_lock(&f->lock)	
.. critical section ..	spin_lock(&f->lock)

f->lock: 0x00000001

Implementation Details

```
void spin_lock(struct qspinlock *lock) {
     if (*(u32 *)lock == 0) {
          lock->locked = 1;
          return;
     }

     if (lock->tail != 0 || lock->pending != 0)
          goto queue;

     lock->pending = 1;
     while (lock->locked != 0);
     lock->pending = 0;
     lock->locked = 1;
     return;

queue: ...
}
void spin_unlock(struct qspinlock *lock) {
     lock->locked = 0;
}
```

```
void foo(struct foo *f){
      spin_lock(&f->lock);
      ... critical section ...
      spin_unlock(&f->lock);
}
```

foo.c

CPU 0	CPU 1
spin_lock(&f->lock)	
.. critical section ..	spin_lock(&f->lock)

f->lock: 0x00000101

Implementation Details

```
void spin_lock(struct qspinlock *lock) {
     if (*(u32 *)lock == 0) {
          lock->locked = 1;
          return;
     }

     if (lock->tail != 0 || lock->pending != 0)
          goto queue;

     lock->pending = 1;
     while (lock->locked != 0);
     lock->pending = 0;
     lock->locked = 1;
     return;

queue: ...
}
void spin_unlock(struct qspinlock *lock) {
     lock->locked = 0;
}
```

```
void foo(struct foo *f){
      spin_lock(&f->lock);
      ... critical section ...
      spin_unlock(&f->lock);
}
```

foo.c

CPU 0	CPU 1
spin_lock(&f->lock)	
.. critical section ..	spin_lock(&f->lock)
spin_unlock(&f->lock)	

f->lock: 0x00000100

Implementation Details

```
void spin_lock(struct qspinlock *lock) {
     if (*(u32 *)lock == 0) {
          lock->locked = 1;
          return;
     }

     if (lock->tail != 0 || lock->pending != 0)
          goto queue;

     lock->pending = 1;
     while (lock->locked != 0);
     lock->pending = 0;
     lock->locked = 1;
     return;

queue: ...
}
void spin_unlock(struct qspinlock *lock) {
     lock->locked = 0;
}
```

```
void foo(struct foo *f){
      spin_lock(&f->lock);
      ... critical section ...
      spin_unlock(&f->lock);
}
```

foo.c

CPU 0	CPU 1
spin_lock(&f->lock)	
.. critical section ..	spin_lock(&f->lock)
spin_unlock(&f->lock)	.. critical section ..

f->lock: 0x00000001

Key Observations

```
void spin_lock(struct qspinlock *lock) {
     if (*(u32 *)lock == 0) {
          lock->locked = 1;
          return;
     }

     if (lock->tail != 0 || lock->pending != 0)
          goto queue;

     lock->pending = 1;
     while (lock->locked != 0);
     lock->pending = 0;
     lock->locked = 1;
     return;

queue: ...
}
void spin_unlock(struct qspinlock *lock) {
     lock->locked = 0;
}
```

1. Spinning only on the `locked` byte

Key Observations

```
void spin_lock(struct qspinlock *lock) {
     if (*(u32 *)lock == 0) {
          lock->locked = 1;
          return;
     }

     if (lock->tail != 0 || lock->pending != 0)
          goto queue;

     lock->pending = 1;
     while (lock->locked != 0);
     lock->pending = 0;
     lock->locked = 1;
     return;

queue: ...
}
void spin_unlock(struct qspinlock *lock) {
     lock->locked = 0;
}
```

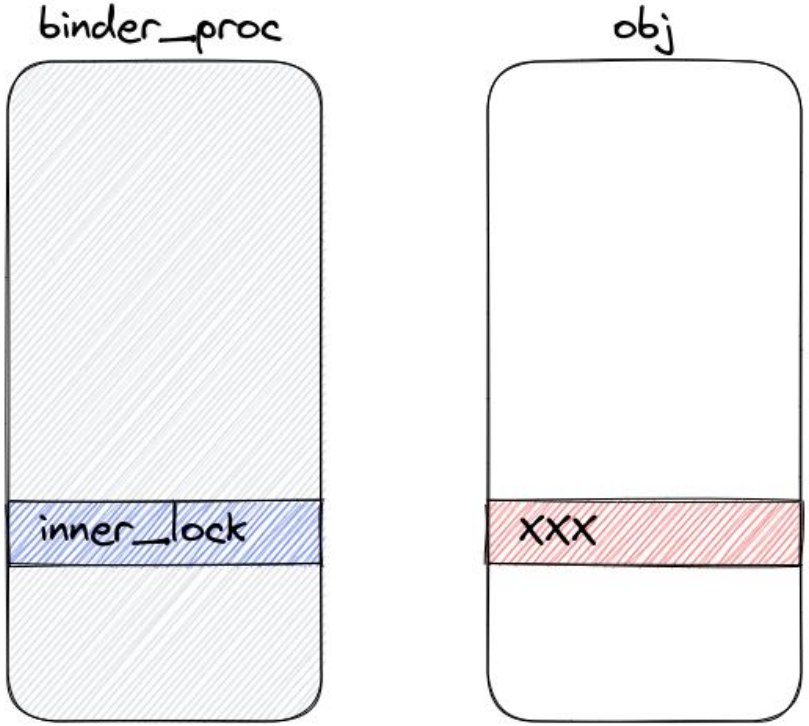
1. Spinning only on the `locked` byte
2. After spinning, only `pending` and `locked` change (`tail` do not)

Key Observations

```
void spin_lock(struct qspinlock *lock) {
      if (*(u32 *)lock == 0) {
            lock->locked = 1;
            return;
      }

      if (lock->tail != 0 || lock->pending != 0)
            goto queue;

      lock->pending = 1;
      while (lock->locked != 0);
      lock->pending = 0;
      lock->locked = 1;
      return;

queue: ...
}
void spin_unlock(struct qspinlock *lock) {
      lock->locked = 0;
}
```

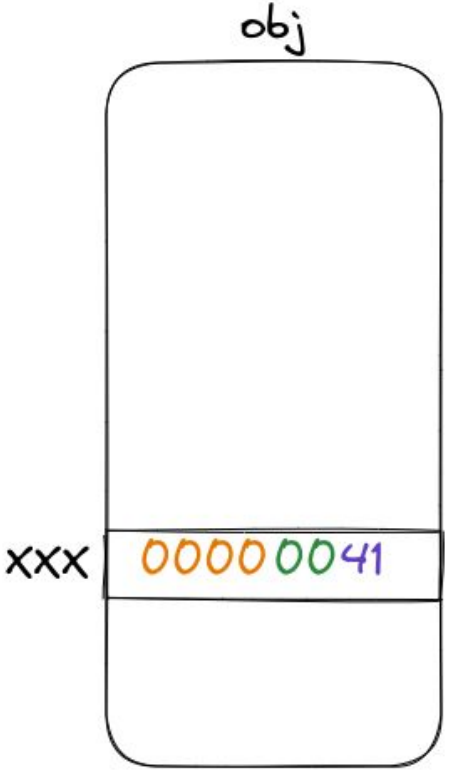
1. Spinning only on the `locked` byte
2. After spinning, only `pending` and `locked` change (`tail` do not)
3. We want to avoid entering `queue`

Extracting Primitives



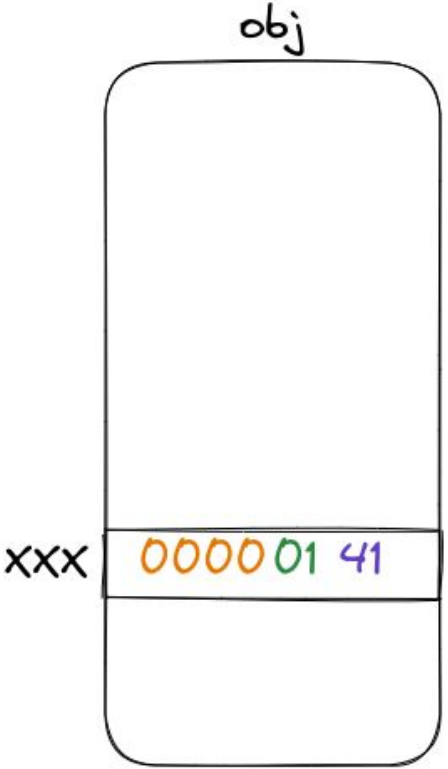
`obj->xxx` changes in some interesting way

Semi-inc primitive



1. **Reallocate** `binder_proc` as `obj`.
2. **Set** `obj->xxx` to a value between `1` and `0xff`.

Semi-inc primitive



1. **Reallocate** `binder_proc` as `obj`.
2. **Set** `obj->xxx` to a value between `1` and `0xff`.
3. **Trigger UAF**. The lock will spin.

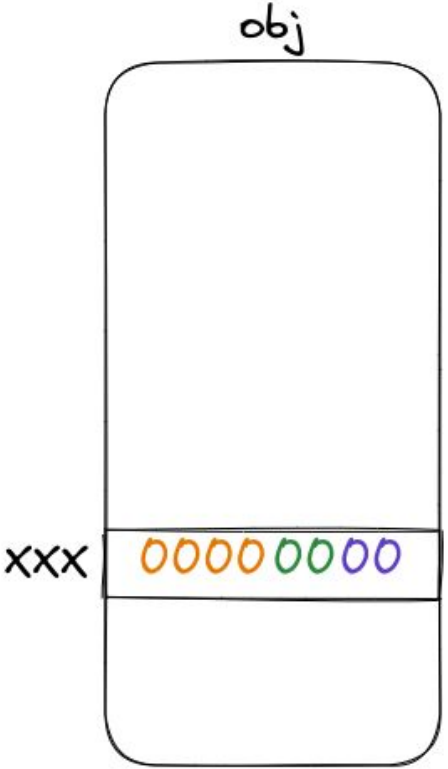
   ⇒ `pending = 1`.

Outcome: `obj->xxx` increased by `0x100`.

Can be useful if `obj->xxx` represents **length** or **flags**.

We didn't find a good instantiation of this primitive.

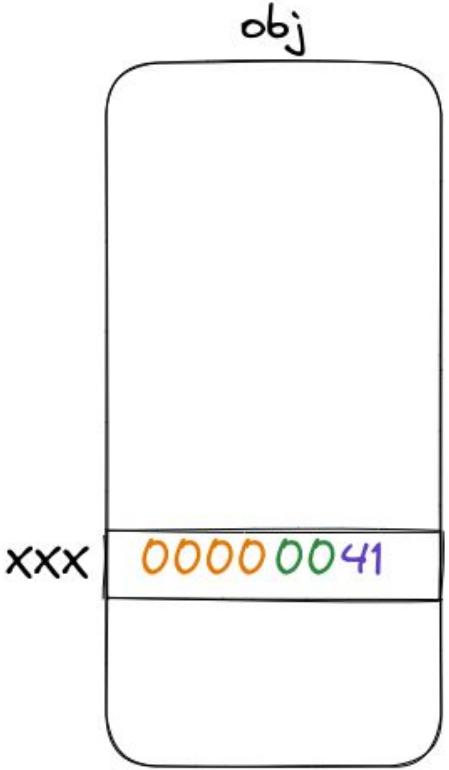
Semi-dec primitive



Assumption: `obj->xxx` represents a refcount.

1. **Reallocate** `binder_proc` as `obj`.

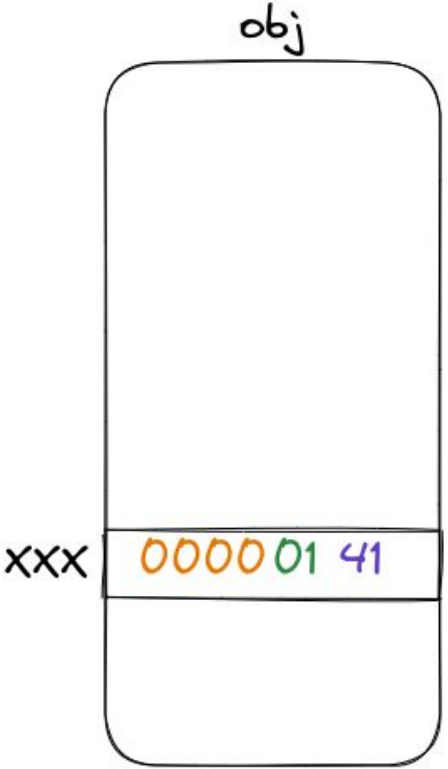
Semi-dec primitive



Assumption: `obj->xxx` represents a refcount.

1. **Reallocate** `binder_proc` as `obj`.
2. **Increment** `obj->xxx` to a value between `1` and `0xff`.

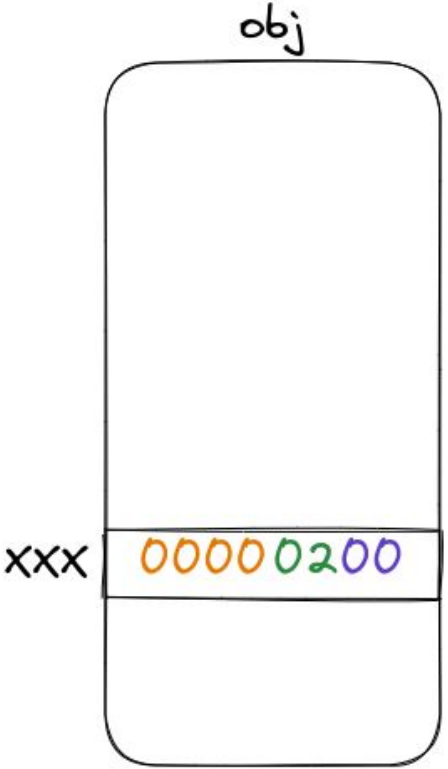
Semi-dec primitive



Assumption: `obj->xxx` represents a refcount.

1. **Reallocate** `binder_proc` as `obj`.
2. **Increment** `obj->xxx` to a value between `1` and `0xff`.
3. **Trigger UAF**. The lock will spin.

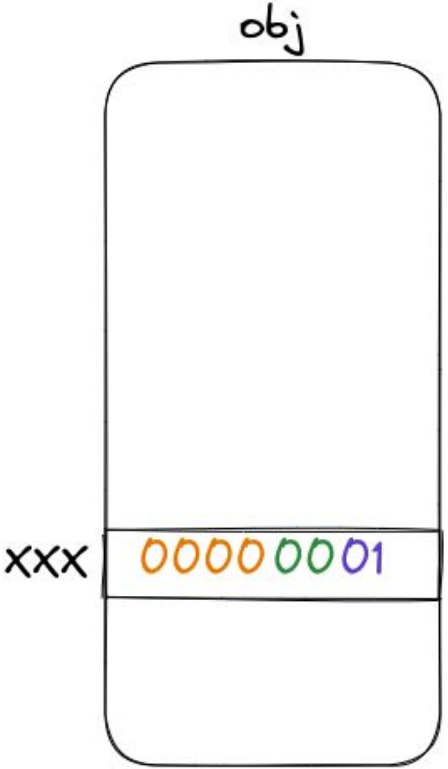
Semi-dec primitive



Assumption: `obj->xxx` represents a refcount.

1. **Reallocate** `binder_proc` as `obj`.
2. **Increment** `obj->xxx` to a value between `1` and `0xff`.
3. **Trigger UAF**. The lock will spin.
4. **Increment** `obj->xxx` to `0x200`.

Semi-dec primitive



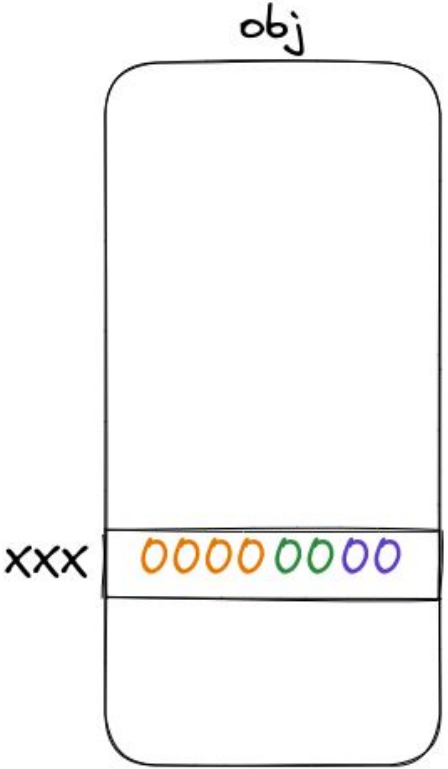
Assumption: `obj->xxx` represents a refcount.

1. **Reallocate** `binder_proc` as `obj`.
2. **Increment** `obj->xxx` to a value between `1` and `0xff`.
3. **Trigger UAF**. The lock will spin.
4. **Increment** `obj->xxx` to `0x200`.

   ⇒ The lock stops spinning.

   ⇒ `pending = 0`, `locked = 1` (for a brief moment).

Semi-dec primitive



Assumption: `obj->xxx` represents a refcount.

1. **Reallocate** `binder_proc` as `obj`.
2. **Increment** `obj->xxx` to a value between `1` and `0xff`.
3. **Trigger UAF**. The lock will spin.
4. **Increment** `obj->xxx` to `0x200`.

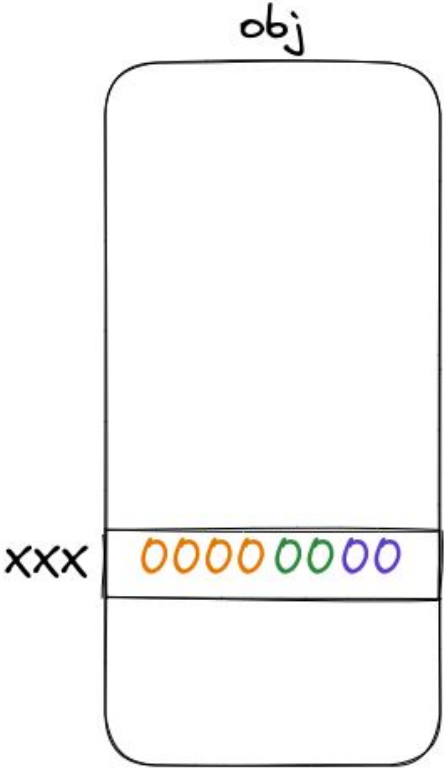
   ⇒ The lock stops spinning.

   ⇒ `pending = 0`, `locked = 1` (for a brief moment).

   ⇒ `spin_unlock()` sets `locked` to 0.

Outcome: `obj` has `0x100` references, but refcount shows 0.

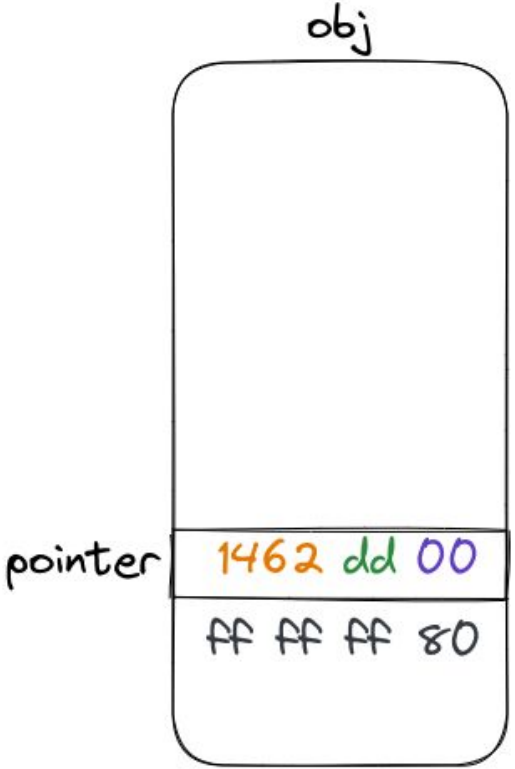
Semi-dec primitive



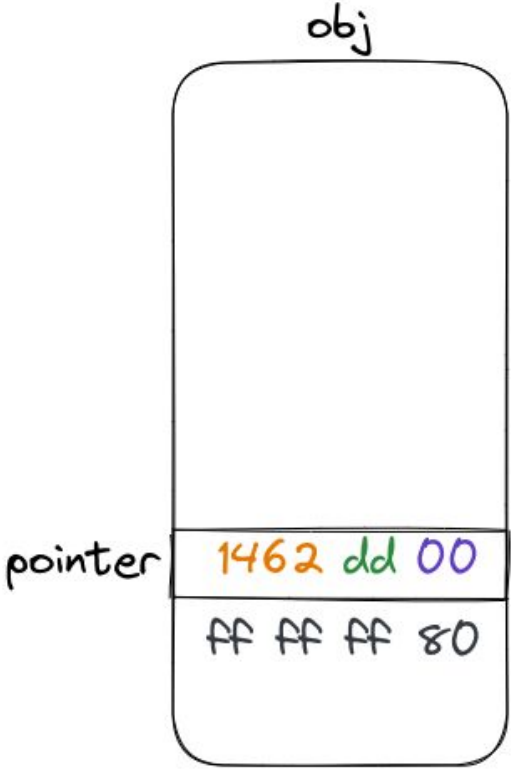
Idea: Do another inc() + dec() to free “obj”.

- ✘ Finding an object with a refcount at a specific offset
  - ✗ Offset changes between devices so not universal exploit
  - ✗ Reduce stability if the object is not in kmalloc-1k (cross-cache)
- ✘ Increment a 0 refcount is considered bad
  - ✗ `CONFIG_REFCOUNT_FULL` converts every `refcount_inc()` to `refcount_inc_not_zero()`

What about pointer corruption?

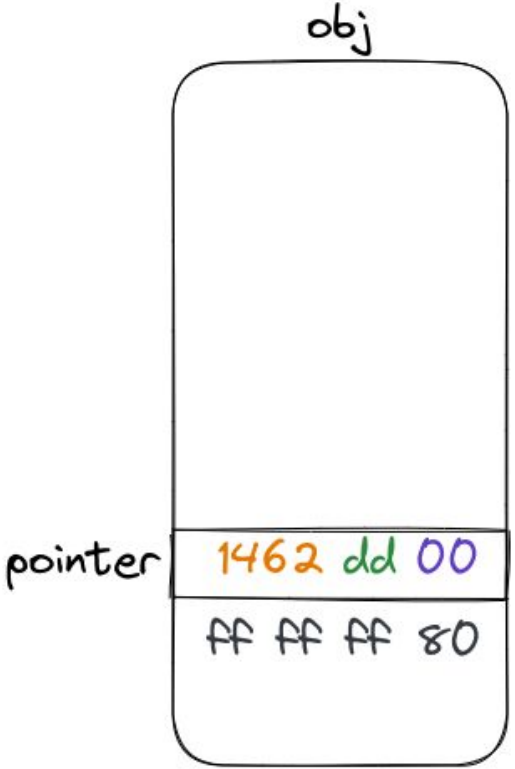


What about pointer corruption?



1. **Reallocate** `binder_proc` as `obj`
2. **Set** `obj->pointer` to some kernel pointer
3. **Trigger UAF**

What about pointer corruption?

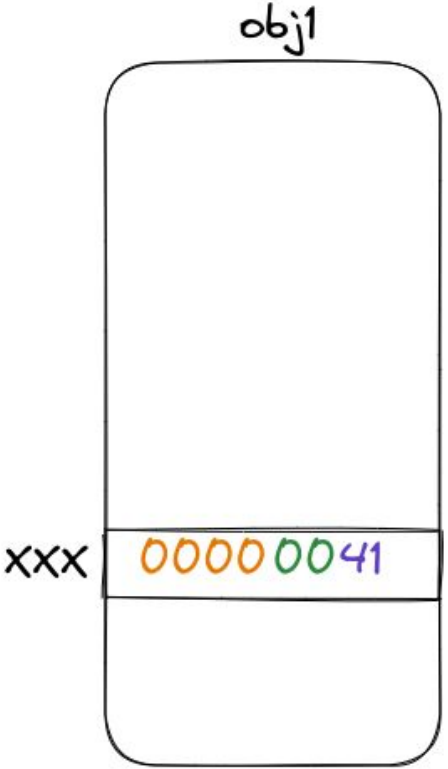


1. **Reallocate** `binder_proc` as `obj`
2. **Set** `obj->pointer` to some kernel pointer
3. **Trigger UAF**

Problem: If (`tail`, `pending`) are non-zero on `spin_lock()` we go to `queue` (... and crash).

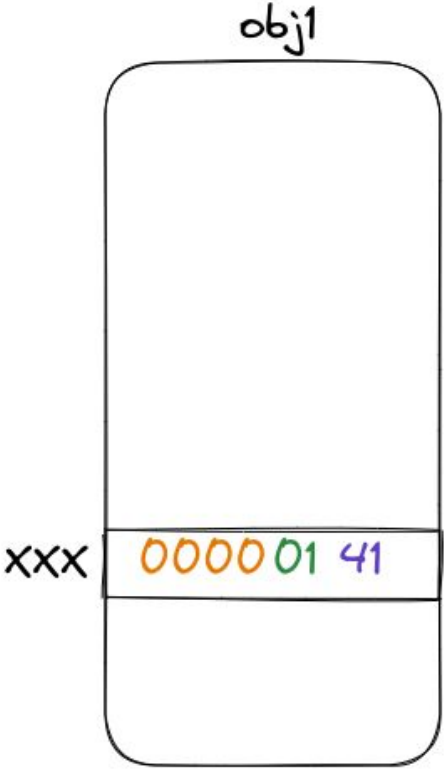
We need `tail == 0`, `pending == 0` and `locked != 0`.

Pointer Corruption: 2nd Attempt



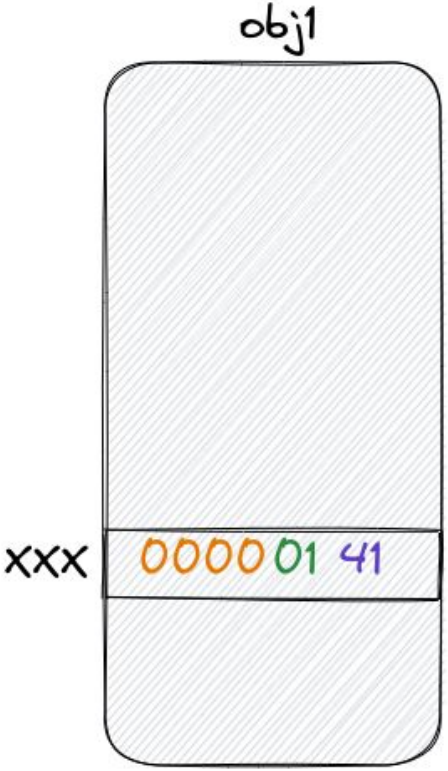
1. **Reallocate** `binder_proc` as `obj1`
2. **Set** `obj1->xxx` to `0x41`

Pointer Corruption: 2nd Attempt



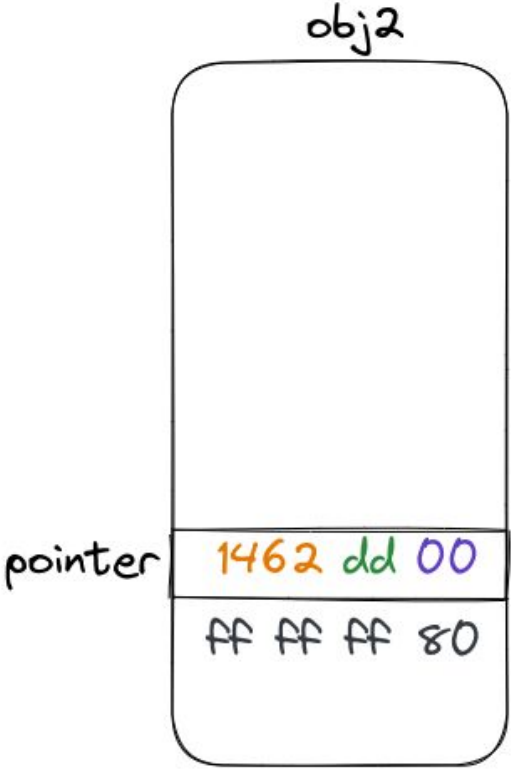
1. **Reallocate** `binder_proc` as `obj1`
2. **Set** `obj1->xxx` to `0x41`
3. **Trigger UAF** from CPU 0

Pointer Corruption: 2nd Attempt



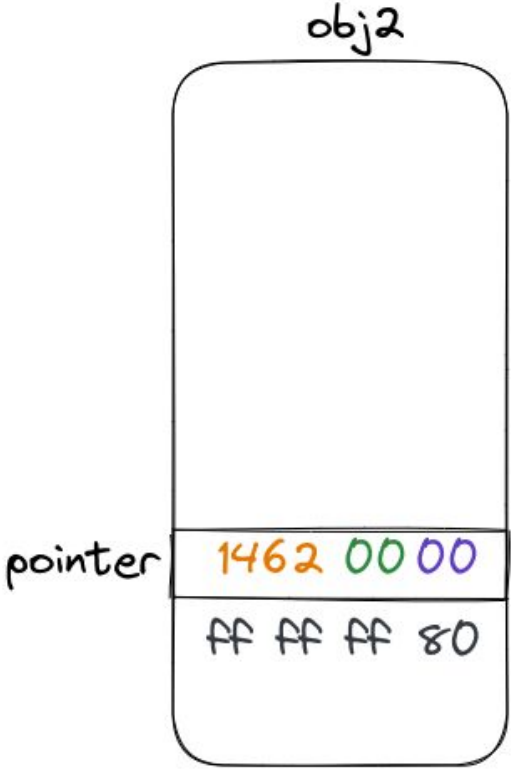
1. **Reallocate** `binder_proc` as `obj1`
2. **Set** `obj1->xxx` to `0x41`
3. **Trigger UAF** from CPU 0
4. **Free `obj1`** from other CPU (lock still spinning)

Pointer Corruption: 2nd Attempt



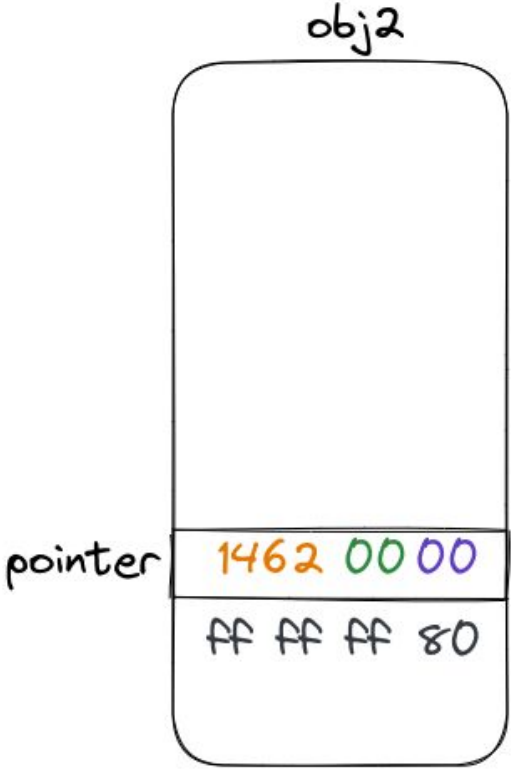
1. **Reallocate** `binder_proc` as `obj1`
2. **Set** `obj1->xxx` to `0x41`
3. **Trigger UAF** from CPU 0
4. **Free `obj1`** from other CPU (lock still spinning)
5. **Reallocate** as `obj2`

Pointer Corruption: 2nd Attempt



1. **Reallocate** `binder_proc` as `obj1`
2. **Set** `obj1->xxx` to `0x41`
3. **Trigger UAF** from CPU 0
4. **Free `obj1`** from other CPU (lock still spinning)
5. **Reallocate** as `obj2`
6. Assuming `obj2->pointer` ends with 0, lock will stop spinning and **zero out the 2nd LSB**

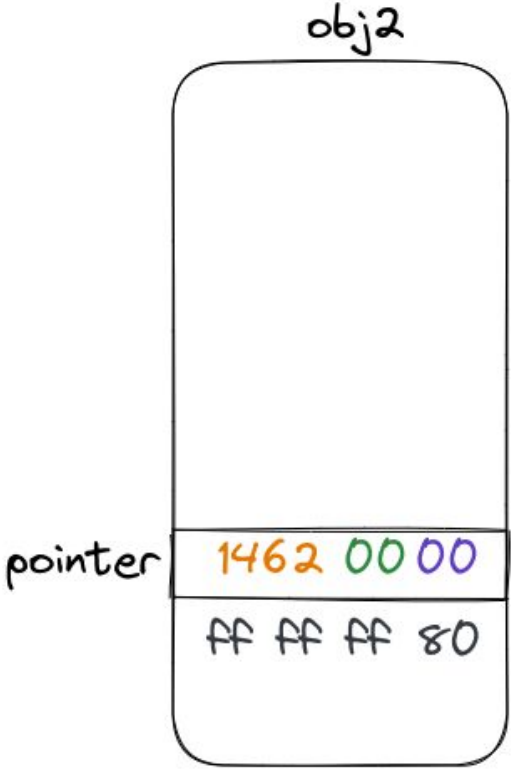
Pointer Corruption: 2nd Attempt



1. **Reallocate** `binder_proc` as `obj1`
2. **Set** `obj1->xxx` to `0x41`
3. **Trigger UAF** from CPU 0
4. **Free `obj1`** from other CPU (lock still spinning)
5. **Reallocate** as `obj2`
6. Assuming `obj2->pointer` ends with 0, lock will stop spinning and **zero out the 2nd LSB**

Allocation is initialized to 0 because of **init_on_alloc**
⇒ We might release the lock too early

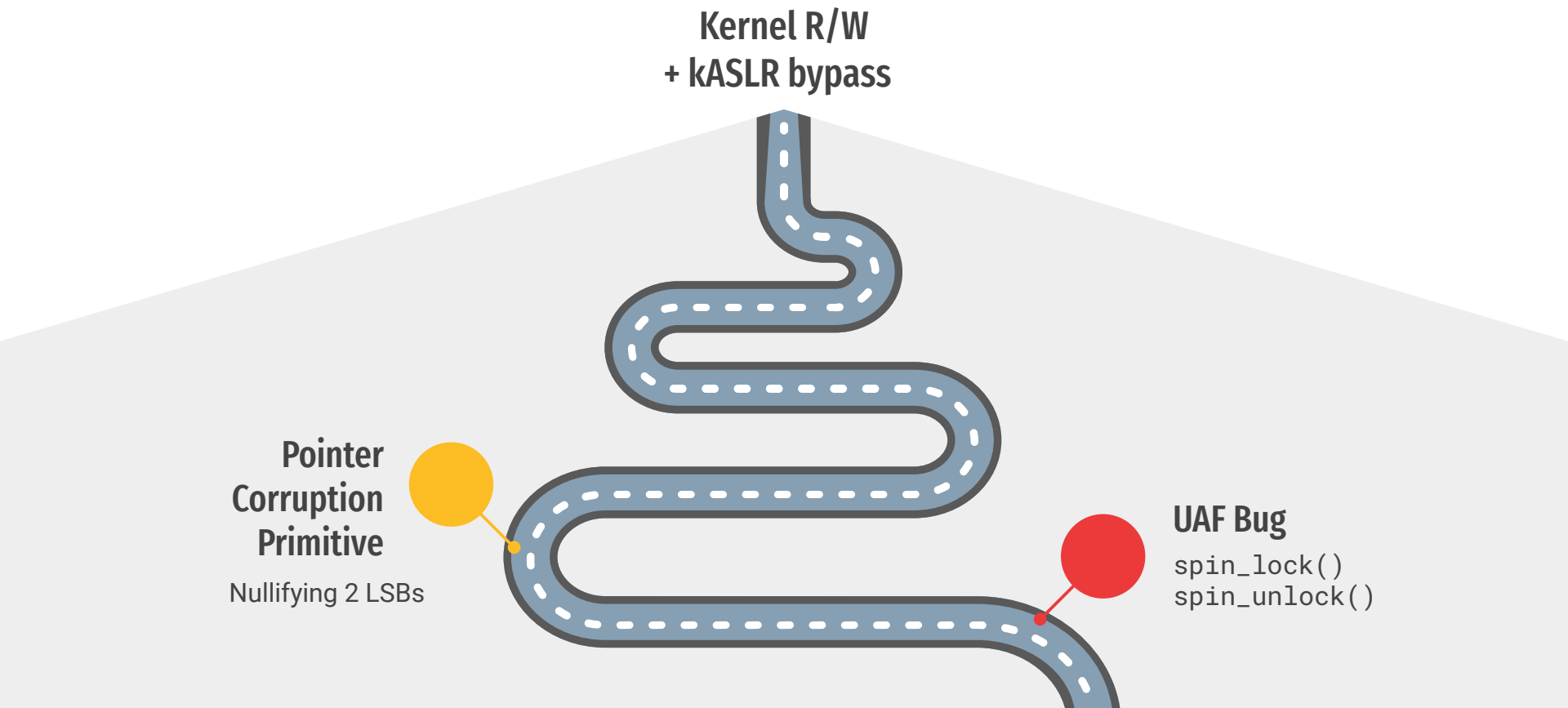
Pointer Corruption: 3rd Attempt



1. **Reallocate** `binder_proc` as `obj1`
2. **Set** `obj1->xxx` to `0x41`
3. **Trigger UAF** from CPU 0
4. **Free `obj1`** from other CPU (lock still spinning)
5. **Reallocate** as `obj2` and **slow down CPU 0 using interrupts*** to win a tiny race
6. Assuming `obj2->pointer` ends with 0, lock will stop spinning and **zero out the 2nd LSB**

* Technique adapted from Jann Horn's "Racing against the clock - hitting a tiny kernel race window"

JSOF


Kernel R/W
+ kASLR bypass
Pointer
Corruption
Primitive
Nullifying 2 LSBs
UAF Bug
spin_lock()
spin_unlock()

Finding good objects

- `obj1[offset]`: 4-byte value with LSB != 0
- `obj2[offset]`: low-half of a kernel pointer with LSB == 0
- “`offset`” might change between devices

`obj1` is a TTY Write Buffer

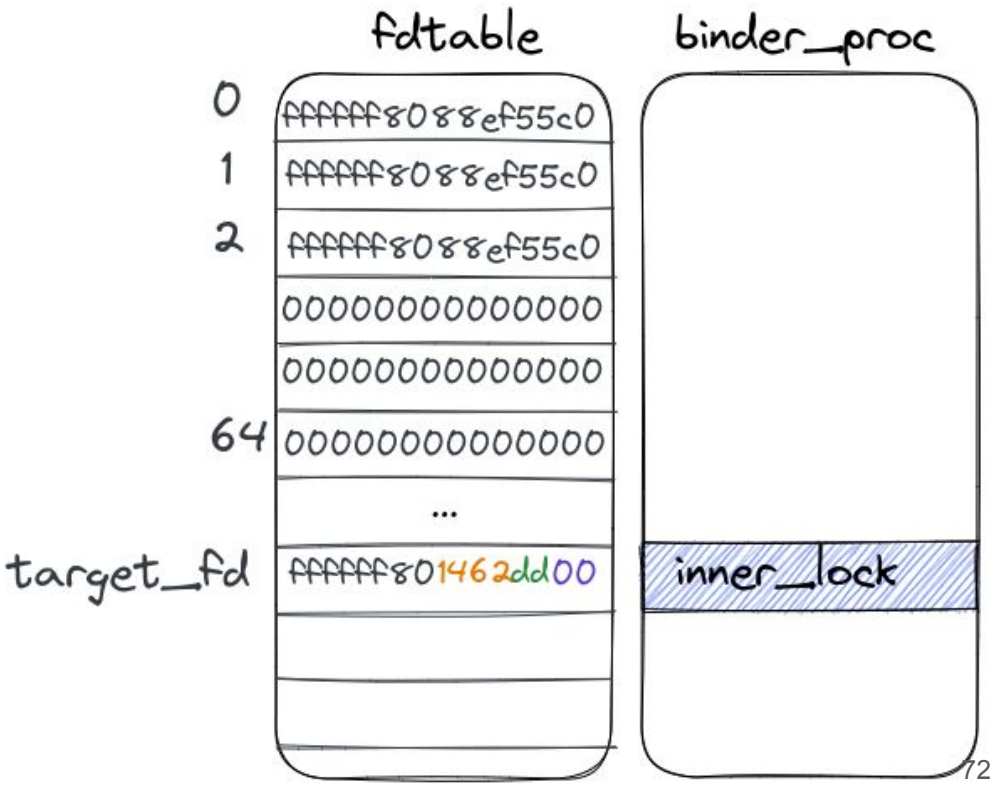
- TTY write buffer can be allocated from **`kmalloc-1k`**

- Contains **arbitrary data** that we control from userspace

```
tty_fd = open(“/dev/ptmx”, O_RDWR);

write(tty_fd, buffer, 1024);
```

obj2 is an fd table

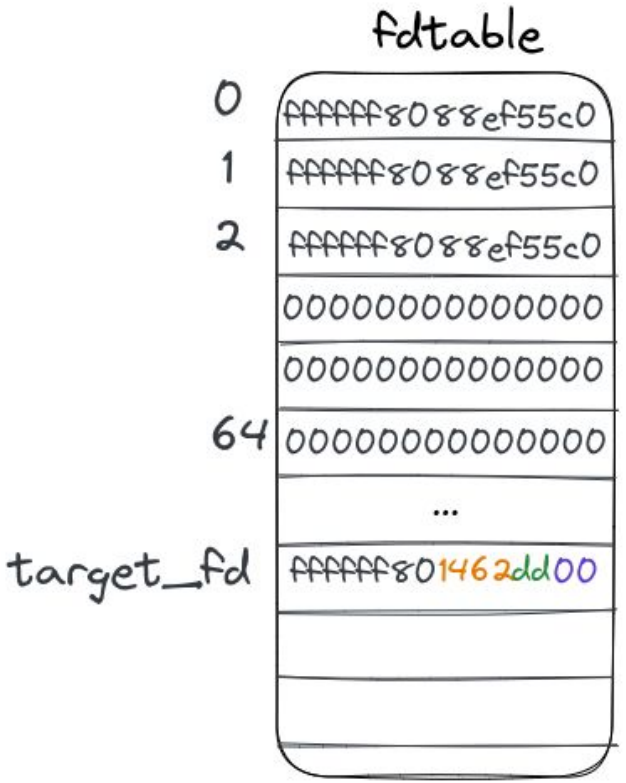
- Array of **`struct file`** pointers
- **Solves** the `inner_lock` offset **fragmentation issue**: will work on any offset that is aligned to 8
- We can access `target_fd` through system calls



`obj2` is an fd table

- Allocated on **kmalloc-1k** if max file descriptor number is in [64, 128)

- Two ways to allocate it:
  1. By **forking** a process with max file descriptor in [64, 128)
  2. By calling **`dup2(fd, new_fd)`** from a process whose max file descriptor < 64 and `new_fd` is in [64, 128)

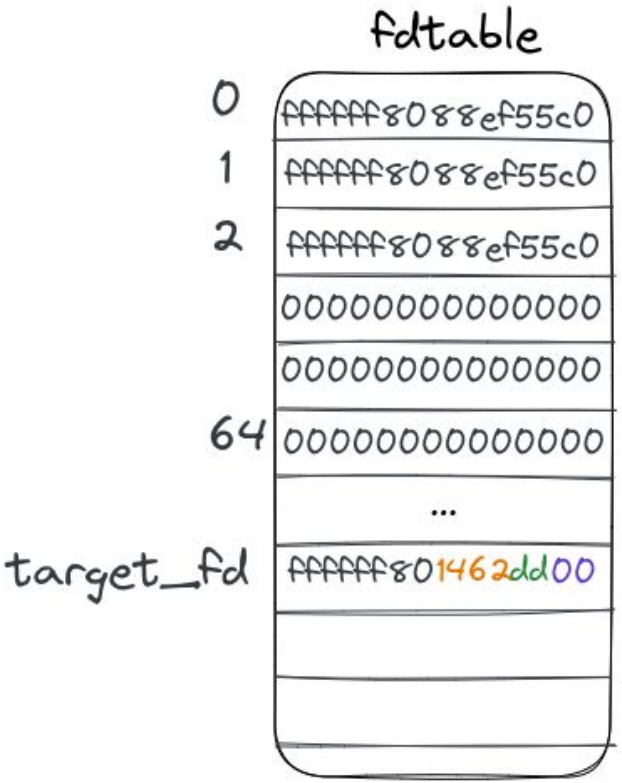


obj2 is an fd table

We used **dup2()** technique:

- ✓ Has less side effects compared to fork()
- ✓ We encountered mostly offsets >= 520

  (meaning `target_fd` >= 65)



Pointer Corruption: Ensuring LSB == 0

- `struct file` pointers do not necessarily end with 0
  - Depends on the size of the struct (our case: 0x140)
- If we catch a pointer with LSB != 0, the lock keep spinning
- Solution: Repeatedly invoke **`dup2(fd, target_fd)`** with random `fd`
  - Probability of LSB == 0 is 7/25 so trying 16 fds succeeds with >99%

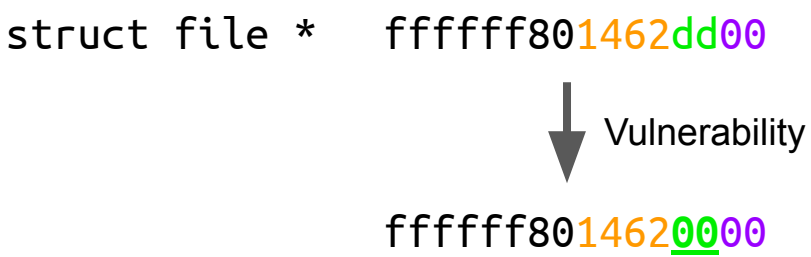
Typical `filp` slab addresses:

```
ffffff80c7cd 8000
ffffff80c7cd 8140
ffffff80c7cd 8280
ffffff80c7cd 83c0
ffffff80c7cd 8500
ffffff80c7cd 8640
ffffff80c7cd 8780
ffffff80c7cd 88c0
ffffff80c7cd 8a00
ffffff80c7cd 8b40
ffffff80c7cd 8c80
ffffff80c7cd 8dc0
…
ffffff80c7cd 9e00
```

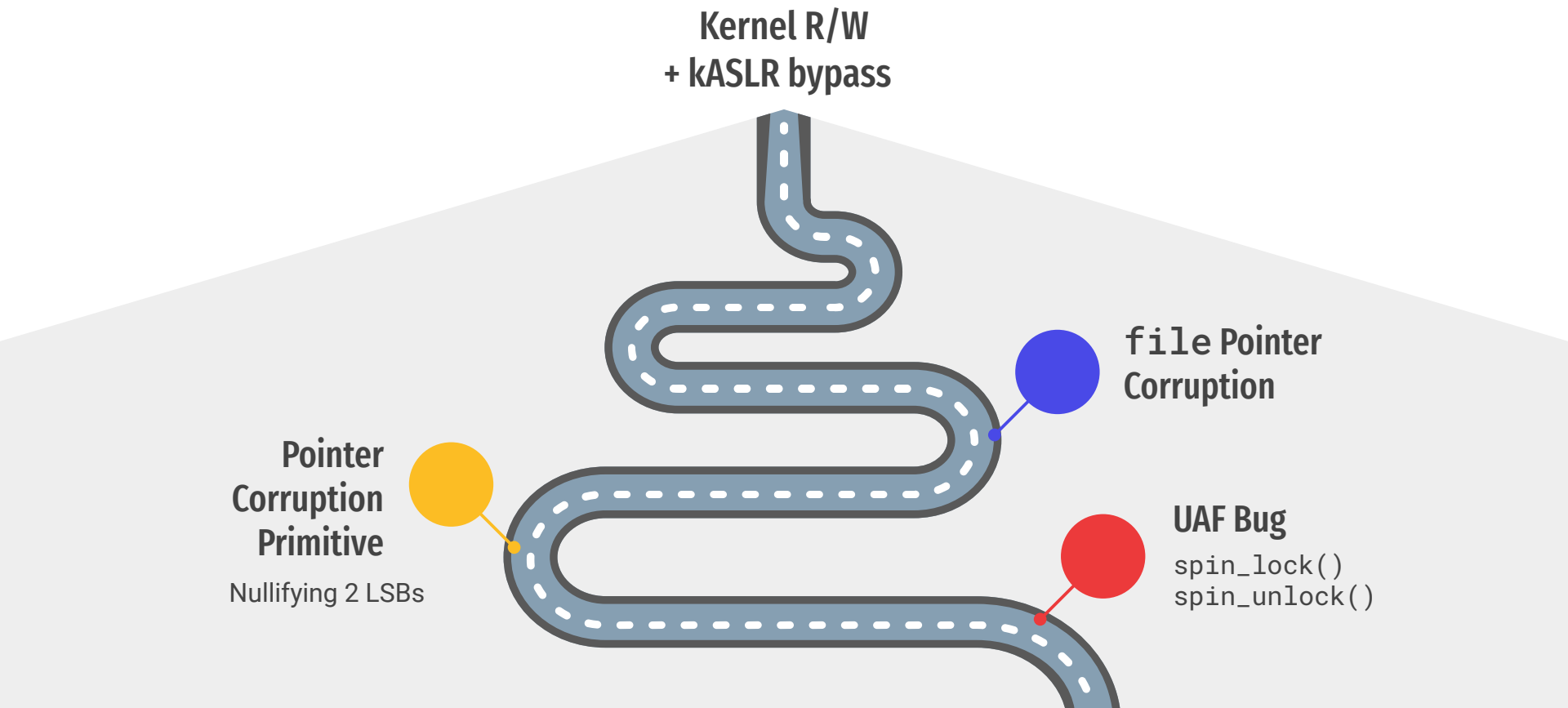
The Primitive

- We can **zero-out the 2 LSBs of a `struct file` pointer**

- The corrupted `struct file` is accessible via specific fd number



JSOF


Kernel R/W
+ kASLR bypass
file Pointer
Corruption
Pointer
Corruption
Primitive
Nullifying 2 LSBs
UAF Bug
spin_lock()
spin_unlock()

The filp cache

- `struct file` is allocated from a **dedicated pool** called “`filp`”

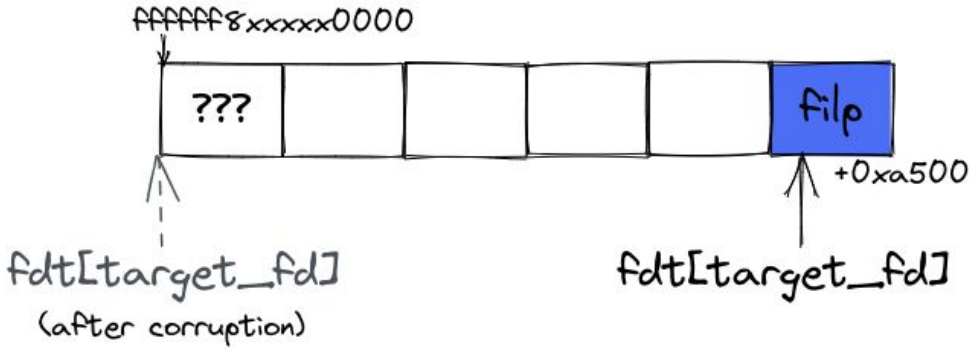
- Each slab consists of **2 pages**
  - ~25 objects per slab

```
f = kmem_cache_zalloc(filp_cachep, GFP_KERNEL);
if (unlikely(!f))
        return ERR_PTR(-ENOMEM);
```

- Slab start address is **aligned** to 2 pages

The filp cache

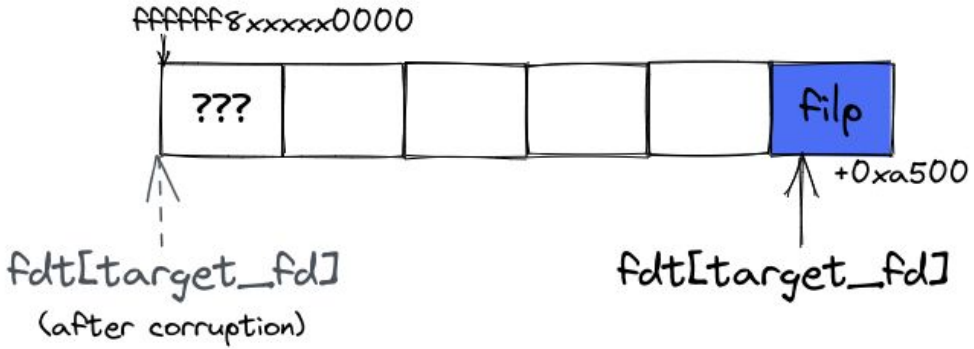
- The corrupted `struct file` **could point outside of its slab**
  - Happens when the slab start address is not aligned to 16 pages



The filp cache

- The corrupted `struct file` **could point outside of its slab**
  - Happens when the slab start address is not aligned to 16 pages

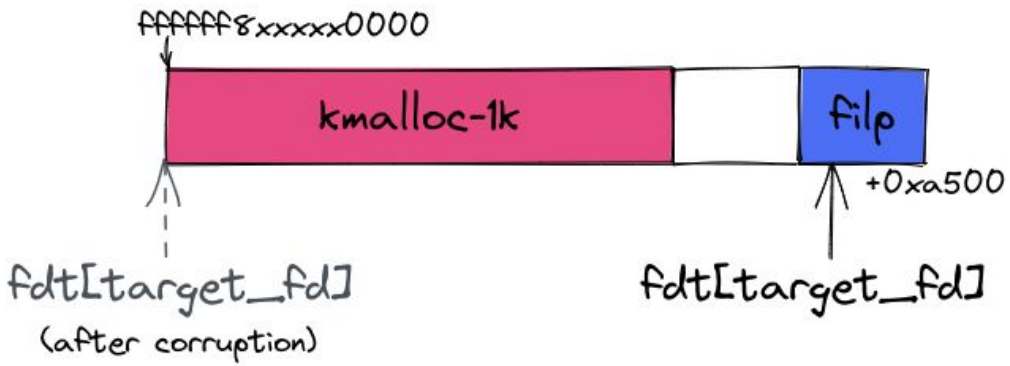
- Our goal: land on an **object under our control**
  - The object will contain a “fake” `struct file`



We want to fake struct file

- Our choice: **TTY write buffer** as our target object

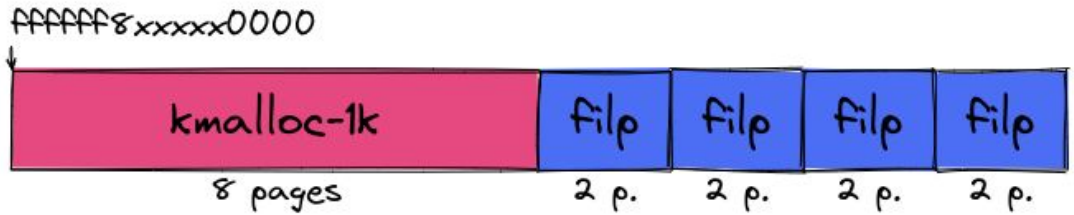
- Allocated from `kmalloc-1k` (8 pages per slab)



Shape physical memory

1. Warm-up: Spray objects in `kmalloc-1k` and `struct files` to **fill-up holes**.
2. **Allocate 32 objects from `kmalloc-1k`.**
3. **Allocate 25*4 `struct files`.**
4. Repeat steps 2 and 3.

Desired situation after shaping:

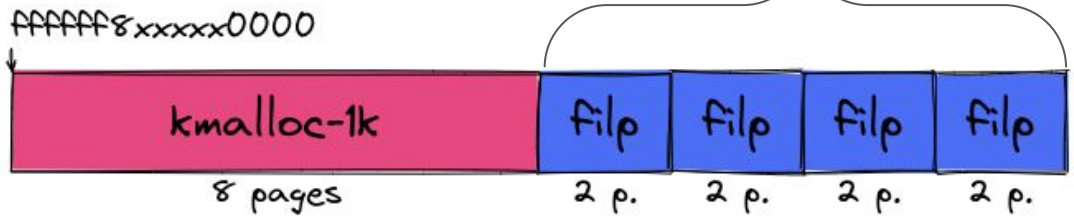


Shape physical memory

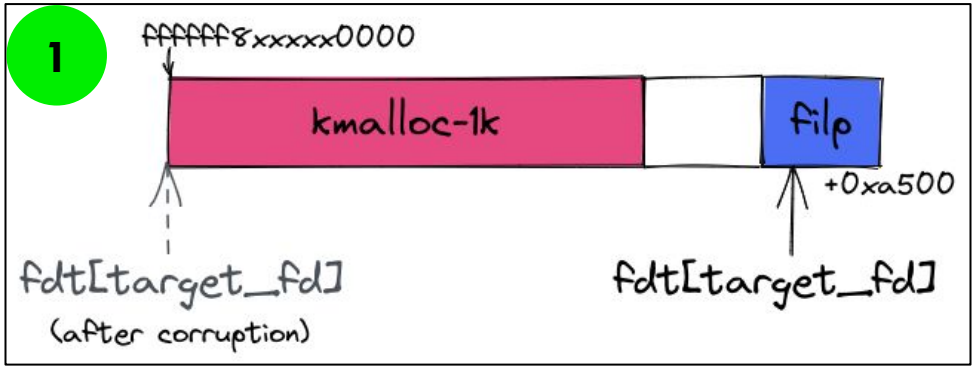
1. Warm-up: Spray objects in `kmalloc-1k` and `struct files` to **fill-up holes**.
2. **Allocate 32 objects from `kmalloc-1k`.**
3. **Allocate 25*4 `struct files`.**
4. Repeat steps 2 and 3.

We use file descriptors from these filp caches for **dup2()**

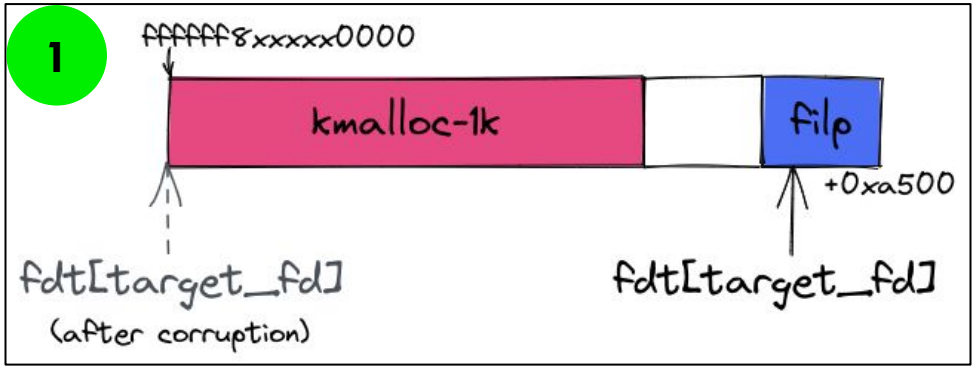
Desired situation after shaping:

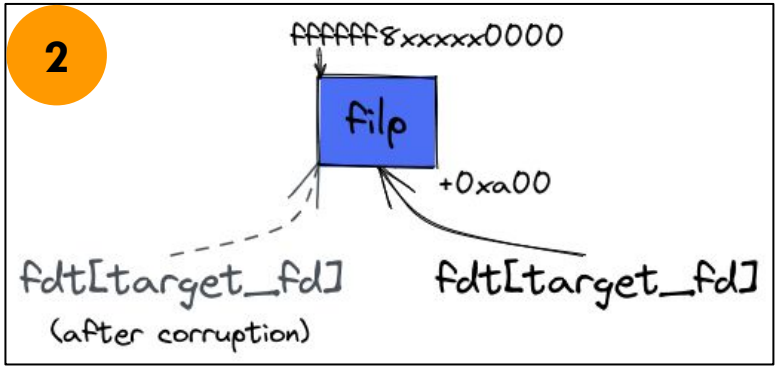


Possible situations after shaping (1/3)

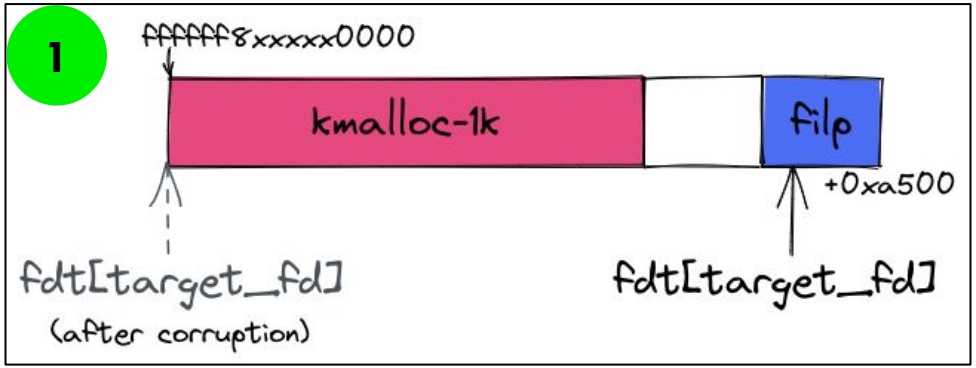


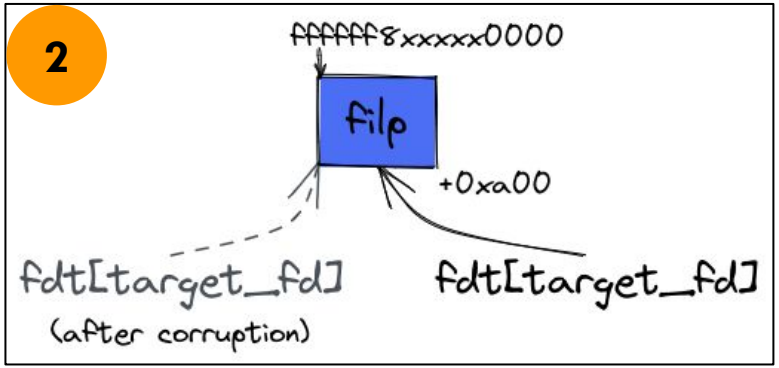
Possible situations after shaping (2/3)

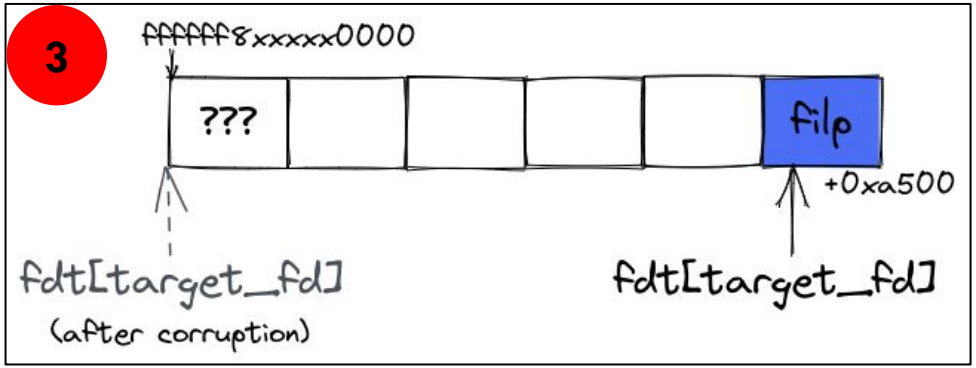




Possible situations after shaping (3/3)

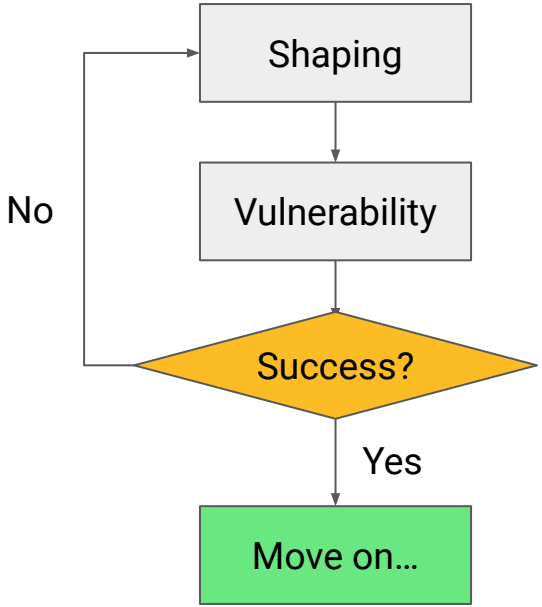




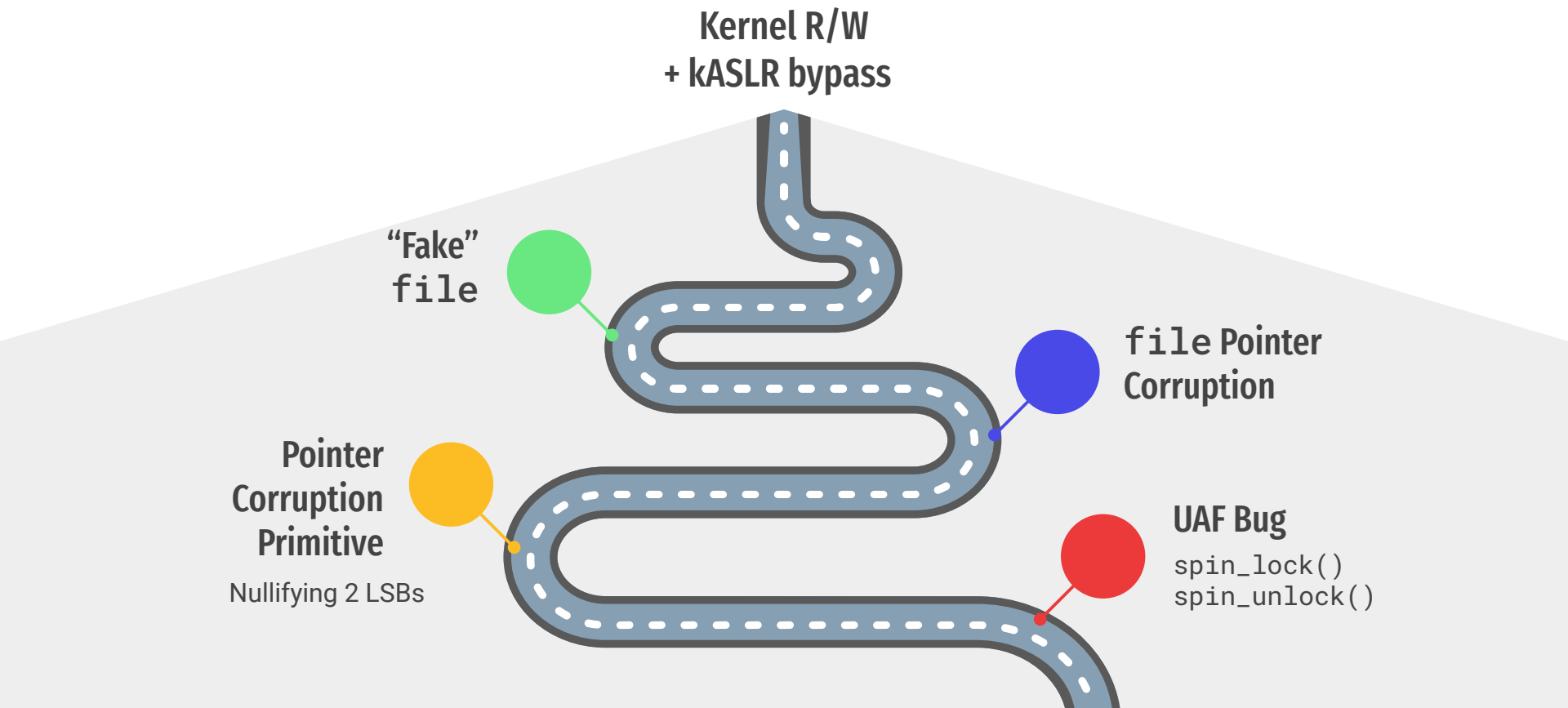


Find out whether we succeeded

- We have access to the corrupted fd
- Idea: Extract bits from the corrupted file
  ⇒ see if match what we expect
- Careful not to dereference any pointer – otherwise we might crash in the "unknown" case



JSOF


Kernel R/W
+ kASLR bypass
“Fake”
file
file Pointer
Corruption
Pointer
Corruption
Primitive
Nullifying 2 LSBs
UAF Bug
spin_lock()
spin_unlock()

What to do with a fake struct file?

- Call **`close()`** on the corrupted fd

- If certain conditions are met, the memory location will be **freed**

- We get **UAF on TTY write buffer** (much stronger)

Closing file descriptors

```
int filp_close(struct file *filp, fl_owner_t id)
{
        int retval = 0;

        if (!file_count(filp)) {
                printk(KERN_ERR "VFS: Close: file count is 0\n");
                return 0;
        }

        if (filp->f_op->flush)
                retval = filp->f_op->flush(filp, id);

        if (likely(!(filp->f_mode & FMODE_PATH))) {
                dnotify_flush(filp, id);
                locks_remove_posix(filp, id);
        }
        fput(filp);
        return retval;
}
```

Ensures `filp->count > 0`

Invoke flush operation if exists

Unless `FMODE_PATH` bit is set, inform any dnotify watchers

Decrement `filp->count` (free it if reaches 0)

Closing file descriptors

```
int filp_close(struct file *filp, fl_owner_t id)
{
        int retval = 0;

        if (!file_count(filp)) {
                printk(KERN_ERR "VFS: Close: file count is 0\n");
                return 0;
        }

        if (filp->f_op->flush)
                retval = filp->f_op->flush(filp, id);

        if (likely(!(filp->f_mode & FMODE_PATH))) {
                dnotify_flush(filp, id);
                locks_remove_posix(filp, id);
        }
        fput(filp);
        return retval;
}
```

Set `filp->count = 1`

Set the `FMODE_PATH` bit

Closing file descriptors

```
int filp_close(struct file *filp, fl_owner_t id)
{
        int retval = 0;

        if (!file_count(filp)) {
                printk(KERN_ERR "VFS: Close: file count is 0\n");
                return 0;
        }

        if (filp->f_op->flush)
                retval = filp->f_op->flush(filp, id);

        if (likely(!(filp->f_mode & FMODE_PATH))) {
                dnotify_flush(filp, id);
                locks_remove_posix(filp, id);
        }
        fput(filp);
        return retval;
}
```

We need to set **`filp->f_op`** to a **valid kernel address** that points to **`NULL`**.

kASLR is not yet bypassed, so we need **fixed** address.

We found such address in the **`vmemmap`** region of the kernel.

Closing file descriptors

- After we all necessary checks are bypassed, **`fput(filp)`** is called

- Internally, it's the function `file_free_rcu()` that frees the file

```
static void file_free_rcu(struct rcu_head *head)
{
        struct file *f = container_of(head, struct file, f_u.fu_rcuhead);

        put_cred(f->f_cred);
        kmem_cache_free(filp_cachep, f);
}
```

Frees the memory location
back to the slab allocator

Closing file descriptors

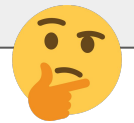
- After we all necessary checks are bypassed, **`fput(filp)`** is called

- Internally, it's the function `file_free_rcu()` that frees the file

```
static void file_free_rcu(struct rcu_head *head)
{
        struct file *f = container_of(head, struct file, f_u.fu_rcuhead);

        put_cred(f->f_cred);
        kmem_cache_free(filp_cachep, f);
}
```

Frees the memory location back to the slab allocator

To which cache the TTY object returns? `filp_cache` or `kmalloc-1k`?



Aside: kmem_cache_free()

```c
void kmem_cache_free(struct kmem_cache *s, void *x)
{
        s = cache_from_obj(s, x);
        if (!s)
                return;
        slab_free(s, virt_to_head_page(x), x, NULL, 1, _RET_IP_);
        trace_kmem_cache_free(_RET_IP_, x);
}
```

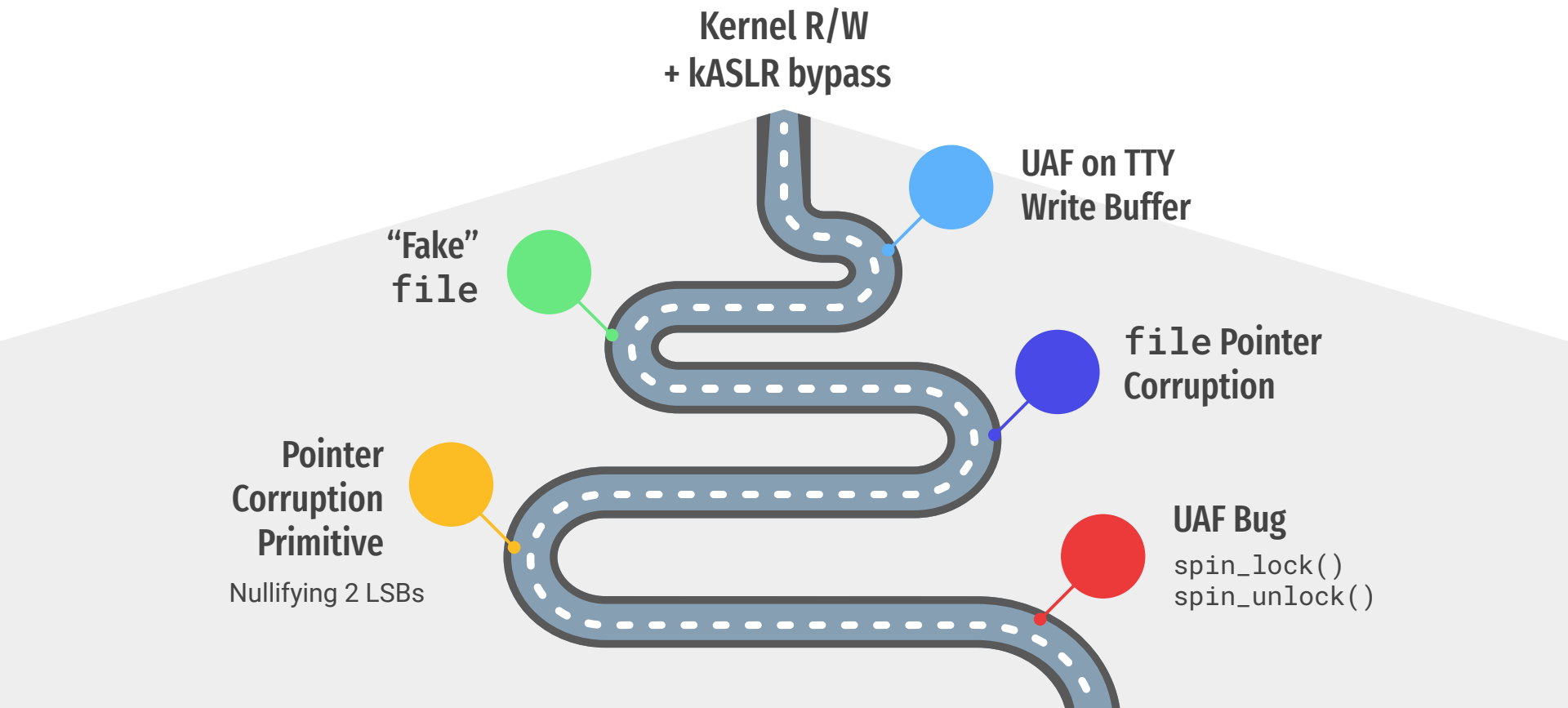
```c
static inline struct kmem_cache *cache_from_obj(struct kmem_cache *s, void *x)
{
        struct kmem_cache *cachep;

        if (!IS_ENABLED(CONFIG_SLAB_FREELIST_HARDENED) &&
            !kmem_cache_debug_flags(s, SLAB_CONSISTENCY_CHECKS))
                return s;

        cachep = virt_to_cache(x);
        if (WARN(cachep && cachep != s,
                 "%s: Wrong slab cache. %s but object is from
                 __func__, s->name, cachep->name))
                print_tracking(cachep, x);
        return cachep;
}
```

```c
static inline struct kmem_cache *virt_to_cache(const void *obj)
{
        struct page *page;

        page = virt_to_head_page(obj);
        if (WARN_ONCE(!PageSlab(page), "%s: Object is not a Slab page!\n",
                                       __func__))
                return NULL;
        return page->slab_cache;
}
```

The cache is determined from the **virtual address**, not argument.

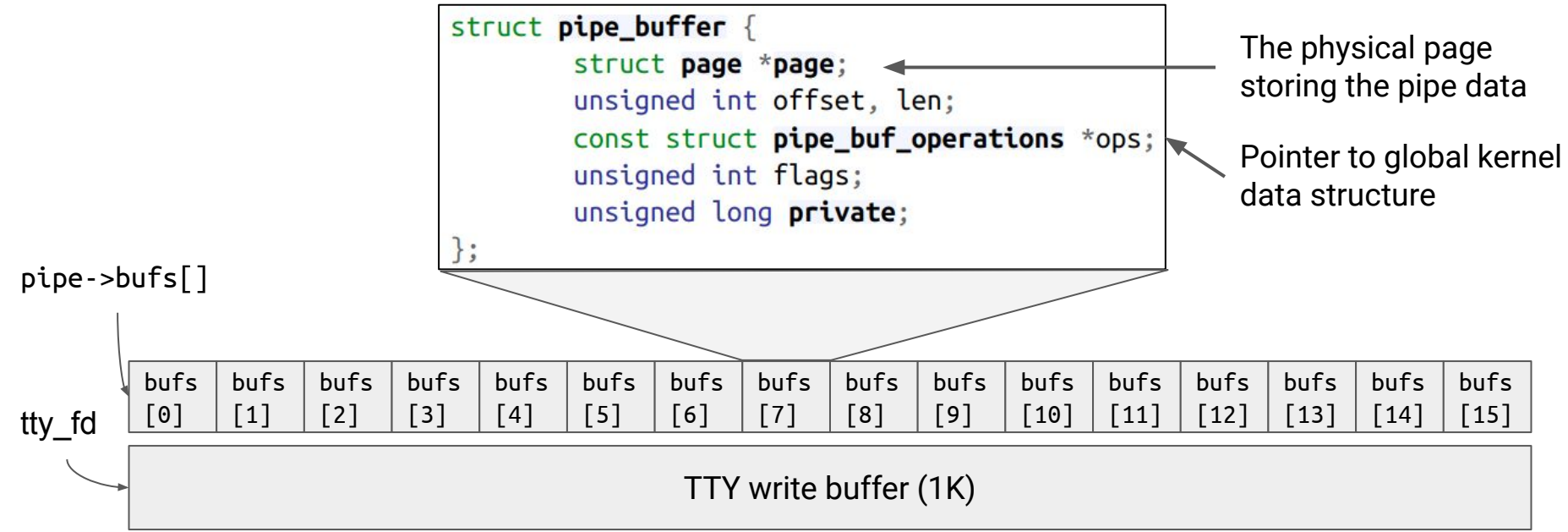
Mismatch leads to a **warning**, not crash.

JSOF


Kernel R/W
+ kASLR bypass
UAF on TTY
Write Buffer
“Fake”
file
file Pointer
Corruption
Pointer
Corruption
Primitive
Nullifying 2 LSBs
UAF Bug
spin_lock()
spin_unlock()

Exploiting the TTY write buffer UAF

- We catch the TTY write buffer with an array of **`pipe_buffer`**s

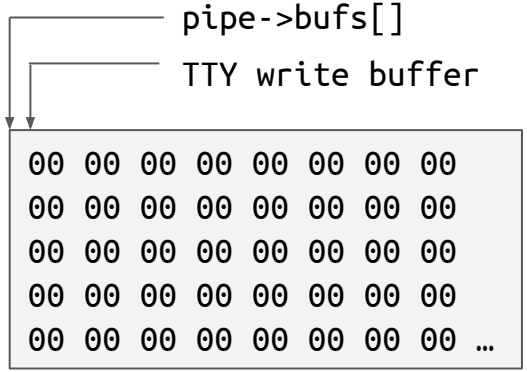


Leaking Pipe Buffer

- Simply reading from TTY file descriptor won't work
- The TTY driver **copies** data from **input buffer** to **output buffer**
- We read from the output buffer

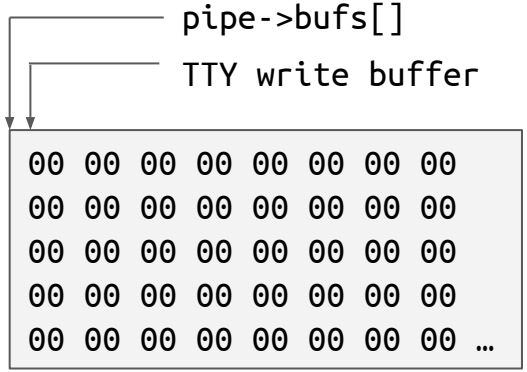
Leaking Pipe Buffer

1. Allocate array of pipe buffers (initialized to 0)



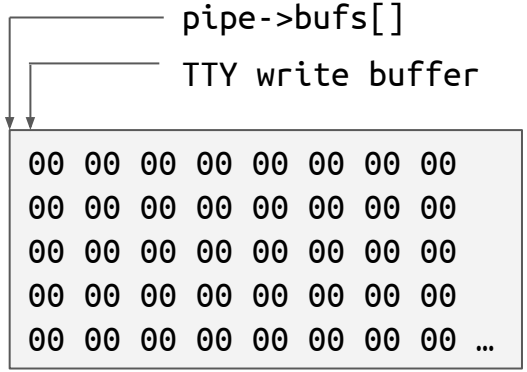
Leaking Pipe Buffer

1. Allocate array of pipe buffers (initialized to 0)
2. Suspend the PTY with **`tcflow(fd, TCOOFF)`**



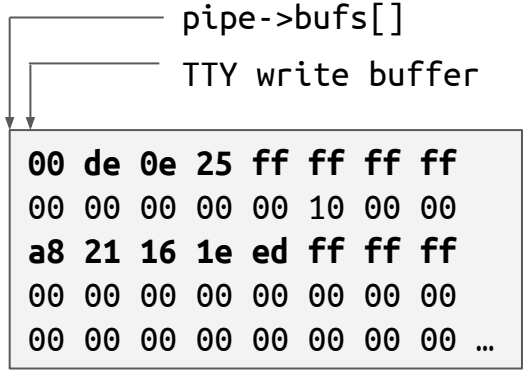
Leaking Pipe Buffer

1. Allocate array of pipe buffers (initialized to 0)
2. Suspend the PTY with **`tcflow(fd, TCOOFF)`**
3. Write 0 on the TTY write buffer
    3.1. Thread waits before the copy to the output buffer



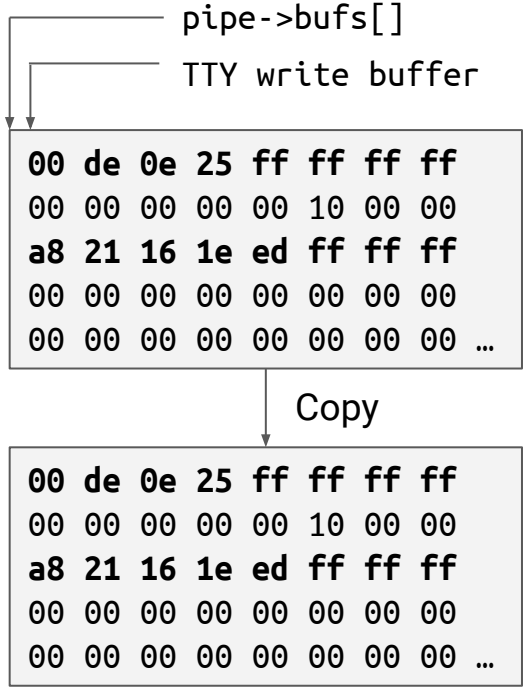
Leaking Pipe Buffer

1. Allocate array of pipe buffers (initialized to 0)
2. Suspend the PTY with **`tcflow(fd, TCOOFF)`**
3. Write 0 on the TTY write buffer
4. Write data to the pipe (populates a pipe buffer)



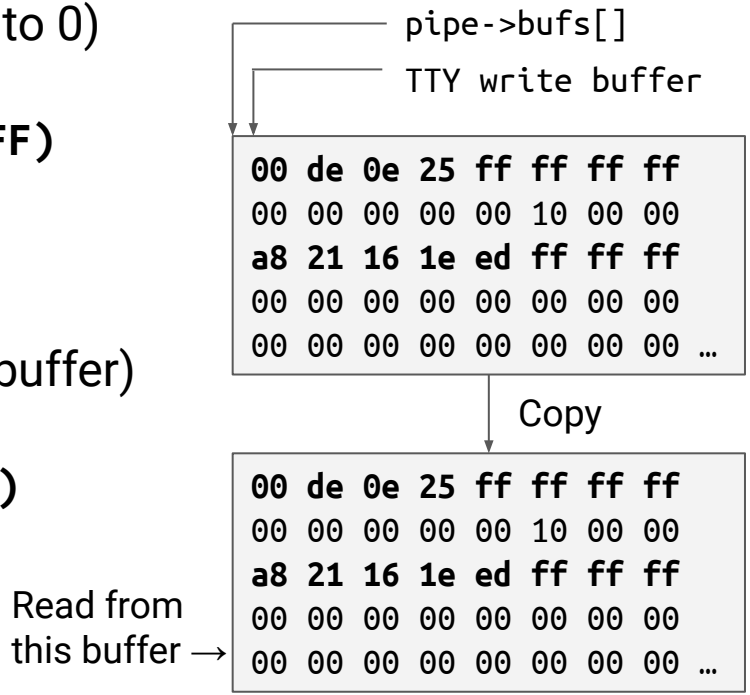
Leaking Pipe Buffer

1. Allocate array of pipe buffers (initialized to 0)
2. Suspend the PTY with **`tcflow(fd, TCOOFF)`**
3. Write 0 on the TTY write buffer
4. Write data to the pipe (populates a pipe buffer)
5. Resume the PTY with **`tcflow(fd, TCOON)`**



Leaking Pipe Buffer

1. Allocate array of pipe buffers (initialized to 0)
2. Suspend the PTY with **`tcflow(fd, TCOOFF)`**
3. Write 0 on the TTY write buffer
4. Write data to the pipe (populates a pipe buffer)
5. Resume the PTY with **`tcflow(fd, TCOON)`**
6. Read from the TTY file descriptor



Leaking Pipe Buffer

- Pipe buffer leaked!
- From leaked **ops** pointer we get kernel image base address
- Defeats kASLR

```
struct pipe_buffer {
        struct page *page;
        unsigned int offset, len;
        const struct pipe_buf_operations *ops;
        unsigned int flags;
        unsigned long private;
};
```

Arbitrary R/W to the Linear Mapping

- By writing to the TTY file descriptor, we can **fake pipe buffer**

- Gives us arbitrary R/W to the linear mapping:
  1. Kernel virtual address in the linear mapping ⇒ struct page address
  2. Fake pipe buffer (esp. the **page** pointer)
  3. R/W from the pipe file descriptors

```
struct pipe_buffer {
        struct page *page;                    Fake
        unsigned int offset, len;
        const struct pipe_buf_operations *ops;
        unsigned int flags;
        unsigned long private;
};
```

Arbitrary R/W

- Use the linear mapping R/W to find our task struct
- Override `addr_limit` to gain full R/W capabilities
- With a UAO (User Access Override) bypass

JSOF

Kernel R/W
+ kASLR bypass

UAF on TTY
Write Buffer

“Fake”
`file`

`file` Pointer
Corruption

Pointer
Corruption
Primitive

Nullifying 2 LSBs

UAF Bug

```
spin_lock()
spin_unlock()
```

Escalate to root

- Disable/bypass SELinux
  - Depending on device: override `enforcing`, write on AVC cache, …

- Run code as root
  - Switch creds to those of init, inject code to a root process (e.g. init), …

Tested devices

- **Samsung Galaxy S22**, Android 12, kernel 5.10.81
- **Google Pixel 6,** Android 12 + **13**, kernel 5.10.[66|107]
- **Samsung Galaxy S21 Ultra**, Android 12, kernel 5.4.129

Our PoC success rate: ~70-80%, varies between devices & background activity

JSOF

Pixel 6 Demo Video

Conclusion

- Wide range of devices are affected by the vulnerability
- A sufficiently motivated attacker can bypass all existing mitigations
  - And run arbitrary code as the root user
- Strong mitigations require stronger vulnerabilities (which are hard to find..)

Conclusion

- Wide range of devices are affected by the vulnerability

- A sufficiently motivated attacker can bypass all existing mitigations
  - And run arbitrary code as the root user

- Strong mitigations require stronger vulnerabilities (which are hard to find..)

Thank You!


github.com/0xkol/badspin


@JSOF18
@0xkol


www.jsof-tech.com