Fixing the igb Driver XDP/AF_XDP TX Queue Stall

A Deep Dive into NAPI Polling, AF_XDP Zero-Copy, and the TX Watchdog in the Linux Kernel

Alex Dvoretsky

February 2026

Based on a 3-patch series for the Linux kernel `igb` driver targeting the Intel I210/I211 Gigabit Ethernet controllers.

Kernel version: 6.17

Contents

Chapter 1

Introduction

1.1 The Problem

Consider a typical deployment scenario: an Intel I210 Gigabit Ethernet controller driven by the `igb` kernel module, with an XDP program attached and an AF_XDP zero-copy application processing packets at high speed. Everything works perfectly—until the AF_XDP application is killed.

```
$ kill -9 $AF_XDP_PID
```

Within five seconds, the kernel logs an ominous message:

```
NETDEV WATCHDOG: CPU: 3: enp8s0 (igb): transmit queue 0 timed out 5008 ms
```

The interface becomes completely unresponsive. No packets are transmitted or received. Pings fail, SSH sessions hang, and the only recovery is to reload the `igb` module or reboot the machine.

The root cause is a chain of interacting bugs in the `igb` driver's handling of XDP program removal when AF_XDP zero-copy sockets are active. The chain involves:

1. An infinite NAPI poll loop in the zero-copy RX path (`igb_clean_rx_irq_zc()`).
2. A missing guard in the TX timeout handler (`igb_tx_timeout()`).
3. Race conditions during the XDP program transition in `igb_xdp_setup()`.

1.2 Scope of This Book

This book explains the Linux kernel subsystems that interact to produce this bug, then walks through a 3-patch fix series. Each chapter builds the prerequisite knowledge for understanding the patches:

Chapter 2
: explains the NAPI polling architecture—how interrupts are coalesced into poll cycles, how poll completion works, and how `napi_synchronize()` depends on the poll function signaling "done."

Chapter 3
: covers AF_XDP zero-copy in the `igb` driver: the buffer pool, the ZC receive path, and the `ndo_xsk_wakeup()` callback.

Chapter 4
traces the lifecycle of an XDP program: how `igb_xdp_setup()` installs and removes programs, and how `igb_close()`/`igb_open()` perform the device reset.

Chapter 5
describes the TX watchdog timer, the timeout handler, and how stale timestamps trigger false alarms.

Chapter 6
analyzes each patch: the root cause it addresses, the fix, and why it is correct.

1.3 Affected Hardware

The bug affects all hardware supported by the `igb` driver that has AF_XDP zero-copy support. In practice, this means the Intel I210 and I211 controllers (PCI device IDs `0x1533` and `0x1539`), which are common on embedded and server platforms.

The testing for this patch series was performed on an Intel I210 (PCI address `0000:08:00.0`) using the `igb` driver version shipped with Linux 6.17.

Chapter 2

NAPI Polling Architecture

NAPI (New API) is the Linux kernel's mechanism for efficient network packet processing. Rather than handling every incoming packet via a hardware interrupt, NAPI coalesces interrupts into *poll cycles* where the driver processes packets in batches.

Understanding NAPI is essential for this book because the TX stall bug is, at its core, a NAPI poll loop that never terminates.

2.1 The Interrupt-to-Poll Transition

The lifecycle of a NAPI poll cycle has four phases:

1. **Hardware interrupt fires.** The NIC signals that packets are available (or TX descriptors have been completed).

2. **Driver schedules NAPI.** The interrupt handler calls `napi_schedule()`, which sets the `NAPI_STATE_SCHED` bit and adds the NAPI instance to the per-CPU poll list.

3. **Softirq runs the poll function.** The `NET_RX_SOFTIRQ` handler calls the driver's `poll` callback with a `budget` (typically 64 packets).

4. **Poll completes or re-arms.** If the driver processed fewer packets than the budget (`work_done < budget`), it calls `napi_complete_done()` to signal completion and re-enable interrupts. If it consumed the full budget, it returns `budget` and NAPI will call it again.

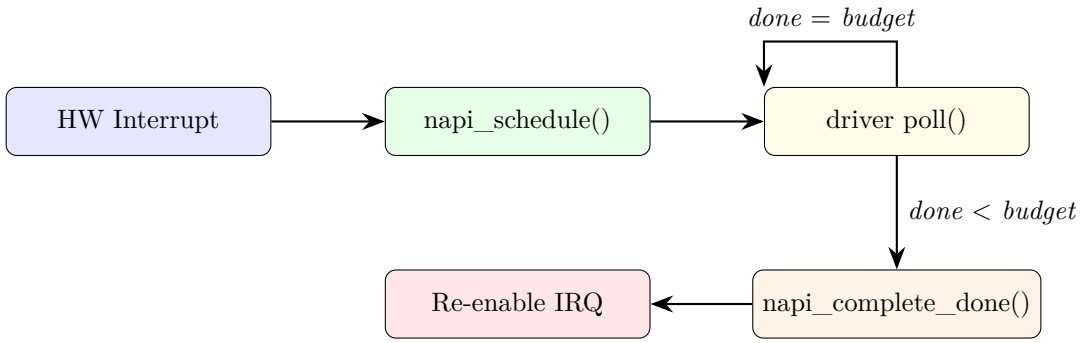

Figure 2.1: NAPI poll lifecycle. The poll function is called repeatedly as long as it returns `budget`; it exits the cycle by returning less than `budget`.

2.2 The `napi_struct`

Each NAPI instance is represented by a `napi_struct`:

```
struct napi_struct {
    unsigned long     state;
    struct list_head  poll_list;
    int               weight;
    int               (*poll)(struct napi_struct *, int);
    struct net_device *dev;
    struct hrtimer    timer;
    /* ... */
};
```

Listing 2.1: Simplified `napi_struct` (include/linux/netdevice.h)

The `state` field is a bitmask of flags. The ones relevant to this book are:

Flag	Meaning
`NAPI_STATE_SCHED`	NAPI is scheduled for polling. Set by `napi_schedule_prep()`, cleared by `napi_complete_done()`.
`NAPI_STATE_MISSED`	New work arrived while NAPI was completing. If `napi_complete_done()` sees this flag, it re-schedules the poll instead of completing.
`NAPI_STATE_DISABLE`	NAPI is being disabled. `napi_schedule_prep()` will refuse to schedule when this is set.

Table 2.1: Key NAPI state flags.

2.3 The Budget Contract

The NAPI budget contract is the fundamental agreement between the kernel and the driver's poll function:

- The kernel calls `poll(napi, budget)` with a positive budget (default weight is 64).
- If the driver returns a value **equal to** `budget`, the kernel assumes there is more work and will call `poll` again.
- If the driver returns a value **less than** `budget`, the driver must have called `napi_complete_done()` to signal that the poll cycle is finished.

This contract is critical: if a poll function *always* returns `budget` without ever returning a smaller value, NAPI will keep polling forever. This is exactly what happens in the bug we are fixing.

2.4 `napi_complete_done()`: The Completion CAS Loop

`napi_complete_done()` is the function that transitions a NAPI instance from the "polling" state back to the "idle" state. Its implementation contains an atomic compare-and-swap (CAS) loop that handles the `MISSED` flag:

```
bool napi_complete_done(struct napi_struct *n, int work_done)
{
    unsigned long flags, val, new, timeout = 0;
    bool ret = true;

    if (unlikely(n→state & (NAPIF_STATE_NPSVC |
                            NAPIF_STATE_IN_BUSY_POLL)))
        return false;

    /* ... GRO flush, poll_list removal ... */

    val = READ_ONCE(n→state);
    do {
        WARN_ON_ONCE(!(val & NAPIF_STATE_SCHED));

        new = val & ~(NAPIF_STATE_MISSED | NAPIF_STATE_SCHED |
                      NAPIF_STATE_SCHED_THREADED |
                      NAPIF_STATE_PREFER_BUSY_POLL);

        /* If MISSED was set, keep SCHED set for one more poll */
        new |= (val & NAPIF_STATE_MISSED) /
               NAPIF_STATE_MISSED * NAPIF_STATE_SCHED;
    } while (!try_cmpxchg(&n→state, &val, new));

    if (unlikely(val & NAPIF_STATE_MISSED)) {
        __napi_schedule(n);
        return false;
    }

    /* ... hrtimer for GRO ... */
    return ret;
}
```

Listing 2.2: `napi_complete_done()` core logic (net/core/dev.c)

The key insight is the `MISSED` flag re-scheduling logic. When another CPU (or the driver itself) sets `MISSED` during the CAS window, `napi_complete_done()` sees it and re-schedules NAPI instead of completing. This is the "missed interrupt" recovery mechanism, and it is what gets exploited by the infinite loop bug.

2.5 `napi_synchronize()`: Waiting for Poll Completion

`napi_synchronize()` is a simple busy-wait that spins until the `NAPI_STATE_SCHED` bit is clear:

```
static inline void napi_synchronize(const struct napi_struct *n)
{
    if (IS_ENABLED(CONFIG_SMP))
        while (test_bit(NAPI_STATE_SCHED, &n→state))
            msleep(1);
    else
        barrier();
}
```

Listing 2.3: `napi_synchronize()` (include/linux/netdevice.h)

This function does not *disable* future activations—it merely waits for the current poll cycle to end. It is used by `igb_down()` to ensure all in-flight NAPI poll callbacks have completed before tearing down hardware resources.

Critical property: if the poll function never returns `done < budget`, then `napi_complete_done()` is never called, `NAPI_STATE_SCHED` is never cleared, and `napi_synchronize()` **hangs forever**.

2.6 `igb_poll()`: The igb Driver's NAPI Callback

The `igb` driver registers `igb_poll()` as the NAPI poll callback. It processes TX completions first, then RX packets:

```
static int igb_poll(struct napi_struct *napi, int budget)
{
    struct igb_q_vector *q_vector = container_of(napi,
                        struct igb_q_vector, napi);
    struct xsk_buff_pool *xsk_pool;
    bool clean_complete = true;
    int work_done = 0;

    if (q_vector→tx.ring)
        clean_complete = igb_clean_tx_irq(q_vector, budget);

    if (q_vector→rx.ring) {
        int cleaned;

        xsk_pool = READ_ONCE(q_vector→rx.ring→xsk_pool);
        cleaned = xsk_pool ?
            igb_clean_rx_irq_zc(q_vector, xsk_pool, budget) :
            igb_clean_rx_irq(q_vector, budget);

        work_done += cleaned;
        if (cleaned >= budget)
            clean_complete = false;
    }

    if (!clean_complete)
        return budget;

    if (likely(napi_complete_done(napi, work_done)))
        igb_ring_irq_enable(q_vector);

    return work_done;
}
```

Listing 2.4: `igb_poll()` (igb_main.c:8280)

The decision tree is:

1. If `igb_clean_tx_irq()` returns `false` (more TX work), `clean_complete` is set to `false`.
2. If the RX cleaner returns `budget` or more, `clean_complete` is set to `false`.
3. If `clean_complete` is `false`, return `budget` without calling `napi_complete_done()`—NAPI will call us again.
4. Otherwise, call `napi_complete_done()` and re-enable interrupts.

2.7 The `__IGB_DOWN` Early-Exit Pattern

`igb_clean_tx_irq()` begins with an early-exit check:

```c
static bool igb_clean_tx_irq(struct igb_q_vector *q_vector,
                             int napi_budget)
{
    struct igb_adapter *adapter = q_vector→adapter;
    /* ... */

    if (test_bit(__IGB_DOWN, &adapter→state))
        return true;

    /* ... process TX completions ... */
}
```

Listing 2.5: `__IGB_DOWN` check in `igb_clean_tx_irq()` (igb_main.c:8344)

When the adapter is going down (`__IGB_DOWN` is set by `igb_down()`), `igb_clean_tx_irq()` returns `true` immediately. In the context of `igb_poll()`, `true` means "TX work is complete" (sets `clean_complete = true`).

This pattern is crucial: it ensures that during shutdown, the TX cleaning path does not hold up NAPI completion. **The bug is that `igb_clean_rx_irq_zc()` lacks this same check.**

Chapter 3

AF_XDP Zero-Copy in igb

AF_XDP is a high-performance packet I/O mechanism that allows user-space applications to send and receive network packets with minimal kernel overhead. When used in *zero-copy mode*, the NIC's DMA engine reads and writes directly to user-space memory, bypassing the kernel's `sk_buff` allocation entirely.

3.1 AF_XDP Architecture

An AF_XDP socket operates on a shared memory region called a UMEM (User Memory for Efficient Mapping). The UMEM is divided into fixed-size frames, and four ring buffers coordinate frame ownership between the kernel and user-space:

Ring	**Direction**	**Purpose**
Fill Queue (FQ)	User → Kernel	User provides empty frames for RX
Completion Queue (CQ)	Kernel → User	Kernel returns frames after TX
RX Ring	Kernel → User	Kernel delivers received packets
TX Ring	User → Kernel	User submits packets for TX

Table 3.1: AF_XDP ring buffers.

In zero-copy mode, the kernel maps the UMEM directly into the NIC's DMA address space. The driver programs the NIC's RX descriptor ring with DMA addresses pointing into the UMEM, so received packets land directly in user-space memory.

3.2 Zero-Copy vs. Copy Mode

Copy mode The kernel receives packets into its own buffers via the normal RX path, then copies the packet data into the AF_XDP socket's UMEM. This adds a `memcpy` per packet but works with any driver.

Zero-copy mode The driver is aware of AF_XDP and programs the NIC to DMA directly into the UMEM. This eliminates the copy but requires explicit driver support. The `igb` driver gained zero-copy support in commit `2c6196013f84` ("igb: Add AF_XDP zero-copy Rx support").

3.3 The XSK Buffer Pool

The kernel-side representation of the UMEM for a particular queue is the `xsk_buff_pool`. It is created when a user-space application binds an AF_XDP socket to a specific queue:

```
struct igb_ring {
    /* ... */
    struct xsk_buff_pool *xsk_pool;
    /* ... */
    union {
        struct igb_rx_buffer *rx_buffer_info;
        struct xdp_buff **rx_buffer_info_zc;
    };
    /* ... */
};
```

Listing 3.1: XSK pool reference in `igb_ring` (igb.h)

When the pool is present, the ring uses `rx_buffer_info_zc` (an array of `xdp_buff` pointers) instead of the normal `rx_buffer_info`. Buffer allocation uses `xsk_buff_alloc()` or `xsk_buff_alloc_batch()` to obtain frames from the Fill Queue.

Key observation: when the AF_XDP application dies, the pool is destroyed. After that point, `xsk_buff_alloc()` returns `NULL`—there are no more frames to allocate.

3.4 `igb_clean_rx_irq_zc()`: The Zero-Copy RX Path

`igb_clean_rx_irq_zc()` is the RX poll function used when a ZC pool is active on a ring. It is selected by `igb_poll()` based on the presence of `xsk_pool`:

```
xsk_pool = READ_ONCE(q_vector→rx.ring→xsk_pool);
cleaned = xsk_pool ?
    igb_clean_rx_irq_zc(q_vector, xsk_pool, budget) :
    igb_clean_rx_irq(q_vector, budget);
```

Listing 3.2: ZC RX path selection in `igb_poll()` (igb_main.c:8300)

The function processes received packets from the descriptor ring, runs them through the XDP program, and allocates replacement buffers from the pool:

```
int igb_clean_rx_irq_zc(struct igb_q_vector *q_vector,
                        struct xsk_buff_pool *xsk_pool,
                        const int budget)
{
    struct igb_adapter *adapter = q_vector→adapter;
    struct igb_ring *rx_ring = q_vector→rx.ring;
    unsigned int total_packets = 0;
    /* ... */

    /* NO __IGB_DOWN CHECK HERE (this is the bug) */

    xdp_prog = READ_ONCE(rx_ring→xdp_prog);

    while (likely(total_packets < budget)) {
        /* Try to read a completed RX descriptor */
        rx_desc = IGB_RX_DESC(rx_ring, ntc);
        size = le16_to_cpu(rx_desc→wb.upper.length);
        if (!size)
            break;    /* no more completed descriptors */
```

```
20
21          /* ... process the packet via XDP ... */
22          total_packets++;
23      }
24
25      /* ... update ring pointers ... */
26
27      /* Try to refill the ring with new buffers */
28      entries_to_alloc = igb_desc_unused(rx_ring);
29      if (entries_to_alloc >= IGB_RX_BUFFER_WRITE)
30          failure |= !igb_alloc_rx_buffers_zc(rx_ring, xsk_pool,
31                                              entries_to_alloc);
32
33      if (xsk_uses_need_wakeup(xsk_pool)) {
34          /* ... wakeup handling ... */
35          return (int)total_packets;
36      }
37      return failure ? budget : (int)total_packets;
38  }
```

Listing 3.3: `igb_clean_rx_irq_zc()` (igb_xsk.c:341), simplified

3.4.1 The Return Value Semantics

The return value of `igb_clean_rx_irq_zc()` determines whether NAPI considers the poll complete:

- If `total_packets < budget`, NAPI calls `napi_complete_done()` and the poll cycle ends.
- If `total_packets >= budget` (or `failure` returns `budget`), NAPI continues polling.

The `failure` flag is set when buffer allocation fails—meaning the Fill Queue is empty. In that case, the function returns `budget` to request another poll cycle, hoping that the user-space application will refill the Fill Queue.

3.4.2 What Happens When the Pool Is Destroyed

When the AF_XDP application dies:

1. The XSK buffer pool is destroyed.
2. `xsk_buff_alloc_batch()` returns 0 buffers—the Fill Queue is gone.
3. `igb_alloc_rx_buffers_zc()` returns `false` (failure), so `failure = true`.
4. There are no completed RX descriptors (the NIC has no buffers to write to), so the `while` loop processes 0 packets.
5. `total_packets = 0`, but `failure = true`, so the function returns `budget`.
6. Back in `igb_poll()`: `cleaned >= budget` ⇒ `clean_complete = false` ⇒ return `budget` without calling `napi_complete_done()`.
7. NAPI re-schedules the poll. Go to step 2.

This creates an **infinite NAPI poll loop**: the function keeps returning `budget` because it cannot allocate buffers, and NAPI keeps calling it because it thinks there is more work to do.

3.5 igb_xsk_wakeup(): The ndo_xsk_wakeup Callback

User-space triggers NAPI polling via the `sendmsg()` system call on the AF_XDP socket, which eventually calls the driver's `ndo_xsk_wakeup` callback. In `igb`, this is `igb_xsk_wakeup()`:

```
int igb_xsk_wakeup(struct net_device *dev, u32 qid, u32 flags)
{
    struct igb_adapter *adapter = netdev_priv(dev);
    struct e1000_hw *hw = &adapter→hw;
    struct igb_ring *ring;
    u32 eics = 0;

    if (test_bit(__IGB_DOWN, &adapter→state))
        return -ENETDOWN;

    if (!igb_xdp_is_enabled(adapter))
        return -EINVAL;

    /* ... validation ... */

    if (!napi_if_scheduled_mark_missed(
            &ring→q_vector→napi)) {
        /* Cause software interrupt */
        if (adapter→flags & IGB_FLAG_HAS_MSIX) {
            eics |= ring→q_vector→eims_value;
            wr32(E1000_EICS, eics);
        } else {
            wr32(E1000_ICS, E1000_ICS_RXDMT0);
        }
    }

    return 0;
}
```

Listing 3.4: `igb_xsk_wakeup()` (igb_xsk.c:527)

Important: this function is called under `rcu_read_lock()` by the AF_XDP core. This means that if we want to ensure no more `igb_xsk_wakeup()` calls are in flight, we need `synchronize_rcu()`. This fact becomes relevant in Patch 3 (Chapter 6).

3.6 Buffer Pool Lifecycle

The full lifecycle of a zero-copy buffer pool:

1. **Creation.** User-space creates an AF_XDP socket and binds it to queue N. The kernel creates an `xsk_buff_pool` and stores it in `adapter->rx_ring[N]->xsk_pool`.
2. **Active use.** The driver's ZC RX/TX paths use the pool to allocate and complete buffers.
3. **Destruction.** The AF_XDP socket is closed (either normally or because the process dies). The pool is destroyed, and `xsk_pool` is set to `NULL`—but not necessarily before the NAPI poll function has already read the old pointer.

The race between pool destruction and NAPI polling is what triggers the bug: `igb_poll()` reads a non-NULL `xsk_pool` pointer and calls `igb_clean_rx_irq_zc()`, but by the time the function tries to allocate buffers, the pool is already gone.

Chapter 4

XDP Program Lifecycle and Device Reset

4.1 `igb_xdp_setup()`: Installing and Removing XDP Programs

The netdev core calls `igb_xdp_setup()` when an XDP program is attached to or detached from the interface. The function handles two cases:

Program replacement (e.g., updating the XDP program without changing modes): the new program is atomically swapped in without a reset.

Mode transition (XDP → non-XDP or non-XDP → XDP): the device must be reset because the ring layout changes.

```
static int igb_xdp_setup(struct net_device *dev,
                         struct netdev_bpf *bpf)
{
    struct igb_adapter *adapter = netdev_priv(dev);
    struct bpf_prog *prog = bpf→prog, *old_prog;
    bool running = netif_running(dev);
    bool need_reset;

    /* ... frame size validation ... */

    old_prog = xchg(&adapter→xdp_prog, prog);
    need_reset = (!!prog != !!old_prog);

    if (need_reset && running) {
        igb_close(dev);
    } else {
        for (i = 0; i < adapter→num_rx_queues; i++)
            (void)xchg(&adapter→rx_ring[i]→xdp_prog,
                adapter→xdp_prog);
    }

    if (old_prog)
        bpf_prog_put(old_prog);

    if (!need_reset)
        return 0;

    /* ... xdp_features update ... */
```

```
29
30      if (running)
31          igb_open(dev);
32
33      return 0;
34  }
```

Listing 4.1: `igb_xdp_setup()` (igb_main.c:2890)

When a mode transition occurs on a running device, the function calls `igb_close()` followed by `igb_open()`. This is a complete device teardown and reinitialization—the most disruptive operation the driver performs.

4.1.1 The `xchg()` Swap

The program pointer is swapped atomically using `xchg()`:

```
1  old_prog = xchg(&adapter→xdp_prog, prog);
```

This is an atomic read-modify-write: it stores `prog` into `adapter->xdp_prog` and returns the previous value. The `need_reset` flag is then computed by comparing whether the "has XDP" state changed:

```
1  need_reset = (!!prog != !!old_prog);
```

The double negation (`!!`) converts the pointer to a boolean. If both are non-NULL (program replacement) or both are NULL (no-op), `need_reset` is false and no close/open is needed.

4.2 `igb_close()`: Tearing Down the Device

`igb_close()` is the network device's close callback (`ndo_stop`). It delegates to `__igb_close()`:

```
1  static int __igb_close(struct net_device *netdev,
2                         bool suspending)
3  {
4      struct igb_adapter *adapter = netdev_priv(netdev);
5
6      igb_down(adapter);
7      igb_free_irq(adapter);
8      igb_free_all_tx_resources(adapter);
9      igb_free_all_rx_resources(adapter);
10     /* ... */
11     return 0;
12 }
```

Listing 4.2: `igb_close()` and `__igb_close()` (igb_main.c:4259)

The real work happens in `igb_down()`.

4.3 `igb_down()`: The Shutdown Sequence

`igb_down()` is the core shutdown function. Understanding its sequence is essential for understanding the bug:

```
1  void igb_down(struct igb_adapter *adapter)
2  {
3      struct net_device *netdev = adapter→netdev;
4      struct e1000_hw *hw = &adapter→hw;
```

```
5
6       /* (1) Signal shutdown */
7       set_bit(__IGB_DOWN, &adapter->state);
8
9       /* (2) Disable HW RX */
10      rctl = rd32(E1000_RCTL);
11      wr32(E1000_RCTL, rctl & ~E1000_RCTL_EN);
12
13      /* (3) Stop TX queues, disable HW TX */
14      netif_carrier_off(netdev);
15      netif_tx_stop_all_queues(netdev);
16      tctl = rd32(E1000_TCTL);
17      tctl &= ~E1000_TCTL_EN;
18      wr32(E1000_TCTL, tctl);
19      wrfl();
20      usleep_range(10000, 11000);
21
22      /* (4) Disable interrupts */
23      igb_irq_disable(adapter);
24
25      /* (5) Wait for NAPI, then disable it */
26      for (i = 0; i < adapter->num_q_vectors; i++) {
27          if (adapter->q_vector[i]) {
28              napi_synchronize(
29                  &adapter->q_vector[i]->napi);
30              napi_disable(
31                  &adapter->q_vector[i]->napi);
32          }
33      }
34
35      /* (6) Cancel timers, update stats, reset HW */
36      timer_delete_sync(&adapter->watchdog_timer);
37      /* ... */
38      igb_reset(adapter);
39
40      /* (7) Clean all rings */
41      igb_clean_all_tx_rings(adapter);
42      igb_clean_all_rx_rings(adapter);
43  }
```

Listing 4.3: `igb_down()` (igb_main.c:2170)

The sequence has a clear dependency chain:

1. **Set `__IGB_DOWN`** (step 1) to signal all paths that the adapter is going down.

2. **Disable hardware** (steps 2–4) to stop new packets and interrupts.

3. **Wait for NAPI** (step 5): `napi_synchronize()` blocks until the current poll cycle completes, then `napi_disable()` prevents future scheduling.

4. **Clean up** (steps 6–7) now that no concurrent access is possible.

The critical point: step 5 can *only* proceed if the poll function eventually returns `done < budget`. If the poll function is stuck in an infinite loop (as with the ZC bug), step 5 hangs forever, and steps 6–7 never execute.

4.4 igb_up() and __igb_open(): Bringing the Device Up

igb_up() (and its more complete sibling __igb_open()) reverses the shutdown:

```
int igb_up(struct igb_adapter *adapter)
{
    /* (1) Configure hardware */
    igb_configure(adapter);

    /* (2) Clear shutdown flag */
    clear_bit(__IGB_DOWN, &adapter→state);

    /* (3) Enable NAPI */
    for (i = 0; i < adapter→num_q_vectors; i++) {
        napi = &adapter→q_vector[i]→napi;
        napi_enable(napi);
    }

    /* (4) Enable interrupts and start TX queues */
    igb_irq_enable(adapter);
    netif_tx_start_all_queues(adapter→netdev);

    /* (5) Start watchdog */
    schedule_work(&adapter→watchdog_task);

    return 0;
}
```

Listing 4.4: igb_up() key steps (igb_main.c:2122)

The __IGB_DOWN bit is cleared *after* configuration but *before* enabling NAPI and starting TX queues.

4.5 igb_reinit_locked(): The Reset Helper

igb_reinit_locked() is a convenience wrapper that performs igb_down() + igb_up() while holding the __IGB_RESETTING flag:

```
void igb_reinit_locked(struct igb_adapter *adapter)
{
    while (test_and_set_bit(__IGB_RESETTING,
                            &adapter→state))
        usleep_range(1000, 2000);
    igb_down(adapter);
    igb_up(adapter);
    clear_bit(__IGB_RESETTING, &adapter→state);
}
```

Listing 4.5: igb_reinit_locked() (igb_main.c:2238)

This is called by igb_reset_task() (the workqueue handler for TX timeouts) and various reconfiguration paths.

4.6 The Close/Open Window

When igb_xdp_setup() calls igb_close() followed by igb_open(), there is a window during which:

1. The adapter is down (`__IGB_DOWN` is set).
2. All TX queues are stopped (`netif_tx_stop_all_queues()`).
3. The TX watchdog timer may still be armed from before the close.
4. The `trans_start` timestamps on the TX queues are stale (they reflect the last transmission *before* the close).

This window is where the TX watchdog can fire a spurious timeout, because it sees stopped queues with stale timestamps.

Chapter 5

TX Watchdog and Timeout Handling

5.1 `dev_watchdog()`: The Network TX Watchdog Timer

The Linux kernel maintains a per-device watchdog timer that monitors TX queue health. The timer callback, `dev_watchdog()`, fires periodically and checks whether any *stopped* TX queue has exceeded its timeout:

```
static void dev_watchdog(struct timer_list *t)
{
    struct net_device *dev = /* ... */;

    if (netif_device_present(dev) &&
        netif_running(dev) &&
        netif_carrier_ok(dev)) {

        for (i = 0; i < dev->num_tx_queues; i++) {
            txq = netdev_get_tx_queue(dev, i);
            if (!netif_xmit_stopped(txq))
                continue;

            trans_start = READ_ONCE(txq->trans_start);

            if (time_after(jiffies,
                    trans_start + dev->watchdog_timeo)) {
                /* TIMEOUT! */
                netdev_crit(dev,
                    "NETDEV WATCHDOG: CPU: %d: "
                    "transmit queue %u timed out "
                    "%u ms\n", ...);
                dev->netdev_ops->ndo_tx_timeout(dev, i);
            }
        }
    }
}
```

Listing 5.1: `dev_watchdog()` (net/sched/sch_generic.c:500), simplified

The default `watchdog_timeo` is 5 seconds (`5*HZ`). The watchdog only checks queues that are *stopped*—if a queue is actively transmitting, its `trans_start` is continuously updated and the watchdog never fires.

5.1.1 The Conditions for a TX Timeout

All of the following must be true for the watchdog to fire:

1. The device is present, running, and has carrier.
2. A TX queue is stopped (via `netif_stop_subqueue()` or `netif_tx_stop_all_queues()`).
3. The queue's `trans_start` timestamp is more than `watchdog_timeo` jiffies in the past.

5.2 `igb_tx_timeout()`: The Driver's Timeout Handler

When the watchdog detects a timeout, it calls the driver's `ndo_tx_timeout` callback. In `igb`, this is `igb_tx_timeout()`:

```
static void igb_tx_timeout(struct net_device *netdev,
                           unsigned int __always_unused txqueue)
{
    struct igb_adapter *adapter = netdev_priv(netdev);
    struct e1000_hw *hw = &adapter->hw;

    /* Do the reset outside of interrupt context */
    adapter->tx_timeout_count++;

    if (hw->mac.type >= e1000_82580)
        hw->dev_spec._82575.global_device_reset = true;

    schedule_work(&adapter->reset_task);
    wr32(E1000_EICS,
         (adapter->eims_enable_mask & ~adapter->eims_other));
}
```

Listing 5.2: `igb_tx_timeout()` before patches (igb_main.c:6649)

The handler does two things:

1. Schedules `igb_reset_task()` on a workqueue, which will call `igb_reinit_locked()` (= `igb_down()` + `igb_up()`).
2. Triggers a software interrupt to attempt to flush any stuck descriptors.

5.3 `igb_reset_task()`: The Reset Workqueue Handler

```
static void igb_reset_task(struct work_struct *work)
{
    struct igb_adapter *adapter;
    adapter = container_of(work, struct igb_adapter,
                           reset_task);

    rtnl_lock();
    /* If we're already down or resetting, just bail */
    if (test_bit(__IGB_DOWN, &adapter->state) ||
        test_bit(__IGB_RESETTING, &adapter->state)) {
        rtnl_unlock();
        return;
    }

    igb_dump(adapter);
    netdev_err(adapter->netdev, "Reset adapter\n");
    igb_reinit_locked(adapter);
    rtnl_unlock();
```

```
19 }
```

Listing 5.3: `igb_reset_task()` (igb_main.c:6665)

Note that `igb_reset_task()` *does* check `__IGB_DOWN` before proceeding. However, this check races with the timeout detection: the watchdog fires, `igb_tx_timeout()` calls `schedule_work()`, and by the time the work item runs, the adapter state may have changed. The work item may already be queued and waiting to run—and it will run *after* `igb_open()` completes in the XDP transition path.

5.4 trans_start and txq_trans_cond_update()

The per-queue timestamp `trans_start` is updated by the transmit path each time a packet is queued:

```
1 static inline void txq_trans_cond_update(
2     struct netdev_queue *txq)
3 {
4     unsigned long now = jiffies;
5     if (READ_ONCE(txq→trans_start) != now)
6         WRITE_ONCE(txq→trans_start, now);
7 }
```

Listing 5.4: `txq_trans_cond_update()` (include/linux/netdevice.h)

The legacy single-queue helper `netif_trans_update()` updates queue 0's `trans_start`:

```
1 static inline void netif_trans_update(struct net_device *dev)
2 {
3     struct netdev_queue *txq = netdev_get_tx_queue(dev, 0);
4     txq_trans_cond_update(txq);
5 }
```

Listing 5.5: `netif_trans_update()` (include/linux/netdevice.h)

5.5 The Stale Timestamp Problem

During the close/open transition in `igb_xdp_setup()`:

1. `igb_close()` calls `igb_down()`, which calls `netif_tx_stop_all_queues()`. The TX queues are now stopped.
2. The `trans_start` values still reflect the last transmission *before* the close. No code updates them.
3. The TX watchdog timer may have been armed before the close and fires during the close/open window.
4. The watchdog sees: stopped queues + stale `trans_start` that is more than 5 seconds old = **timeout**.
5. `igb_tx_timeout()` schedules `igb_reset_task()`.

This is a false positive: the queues are stopped because the device is being reconfigured, not because of a hardware hang. But the timeout handler does not distinguish between the two cases.

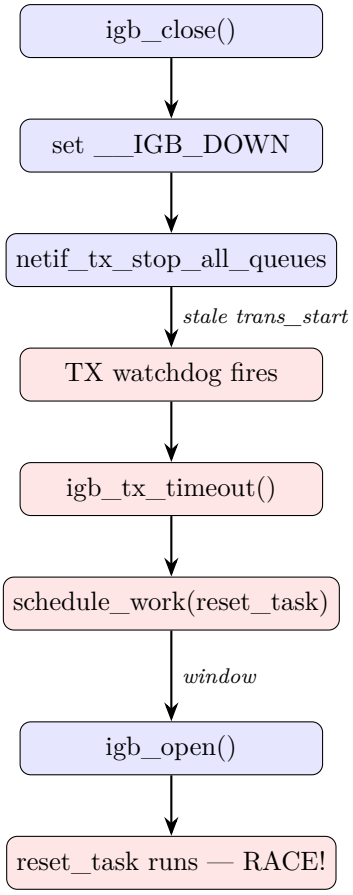

Figure 5.1: The stale timestamp race during XDP close/open transition.

Chapter 6

The Patch Series—Analysis and Correctness

This chapter analyzes each of the three patches in the fix series. For each patch, we describe the root cause, the fix, and why it is correct and sufficient.

6.1 Patch 1: `__IGB_DOWN` Check in `igb_clean_rx_irq_zc()`

6.1.1 Commit Message

```
igb: check __IGB_DOWN in igb_clean_rx_irq_zc()

When an AF_XDP zero-copy application terminates abruptly (e.g.,
kill -9), the XSK buffer pool is destroyed but NAPI polling
continues. igb_clean_rx_irq_zc() keeps returning budget (no
descriptors, no buffers to allocate, xsk_buff_alloc() returns
NULL) which makes napi_complete_done() re-arm the poll
indefinitely.

Meanwhile, igb_down() -> napi_synchronize() waits for a NAPI
poll cycle that signals completion with done < budget -- which
never happens. This blocks igb_down() forever, and the 5-second
TX watchdog fires because no TX completions are processed while
NAPI is stuck.

Fix this by adding an __IGB_DOWN check at the top of
igb_clean_rx_irq_zc(), returning 0 immediately when the adapter
is going down.

Fixes: 2c6196013f84 ("igb: Add AF_XDP zero-copy Rx support")
```

6.1.2 Root Cause Analysis

The root cause is a missing early-exit check in `igb_clean_rx_irq_zc()`. Let's trace the exact sequence of events:

1. An AF_XDP zero-copy application is running on queue 0 with an XDP program attached.
2. The application is killed (`kill -9`).
3. The AF_XDP socket closes, destroying the XSK buffer pool.
4. The XDP program is detached (either by the dying process or by the administrator).

5. `igb_xdp_setup()` is called with `prog = NULL` and `need_reset = true`.
6. `igb_close()` is called, which calls `igb_down()`.
7. `igb_down()` sets `__IGB_DOWN` and eventually calls `napi_synchronize()` (step 5 in Section 4.3).
8. But NAPI is stuck: `igb_clean_rx_irq_zc()` keeps returning `budget` because:
   - No RX descriptors are completed (NIC has no buffers).
   - Buffer refill fails (`failure = true`).
   - The function returns `budget` on failure.
9. `napi_synchronize()` waits for `NAPI_STATE_SCHED` to clear, but it never does because the poll function never returns `done < budget`.
10. `igb_down()` hangs at step 5.
11. Meanwhile, `igb_clean_tx_irq()` is also not processing TX completions (it returns `true` immediately due to its own `__IGB_DOWN` check, but the NAPI poll loop is dominated by the RX side returning `budget`).
12. After 5 seconds, the TX watchdog fires because no TX completions have been processed and `trans_start` is stale.
13. `igb_tx_timeout()` schedules a reset, but the reset cannot proceed because `igb_down()` is already hung.
14. The TX queue is permanently stalled.

6.1.3 The Fix

```
int igb_clean_rx_irq_zc(struct igb_q_vector *q_vector,
                        struct xsk_buff_pool *xsk_pool,
                        const int budget)
{
    struct igb_adapter *adapter = q_vector→adapter;
    /* ... */

   if (test_bit(__IGB_DOWN, &adapter→state))
       return 0;

    /* xdp_prog cannot be NULL in the ZC path */
    xdp_prog = READ_ONCE(rx_ring→xdp_prog);
```

Listing 6.1: Patch 1 diff (igb_xsk.c)

The fix adds a single check: if `__IGB_DOWN` is set, return 0 immediately.

6.1.4 Why It Is Correct

1. **Returns 0, not `budget`.** Returning 0 means "no work done, poll is complete." In `igb_poll()`, this results in `cleaned = 0 < budget`, so `clean_complete` remains `true`, and `napi_complete_done()` is called.
2. **Matches the existing pattern.** `igb_clean_tx_irq()` already has the same check at line 8344, returning `true` ("TX complete") when `__IGB_DOWN` is set. This patch extends the pattern to the ZC RX path.

3. **Breaks the infinite loop.** With this check, once `igb_down()` sets `__IGB_DOWN`, the very next poll cycle will have both TX and RX returning "done," so `napi_complete_done()` is called, `NAPI_STATE_SCHED` is cleared, and `napi_synchronize()` returns.
4. **No spurious packet loss.** When `__IGB_DOWN` is set, the hardware RX and TX are already disabled. No new packets will arrive, and any remaining descriptors will be cleaned by `igb_clean_all_rx_rings()` in step 7 of `igb_down()`. Returning 0 does not lose any packets.
5. **Safe to call at any point.** The `test_bit` is a plain atomic read with no side effects. It does not take any locks or modify any state.

6.1.5 What About the Normal (Non-ZC) RX Path?

The normal RX path (`igb_clean_rx_irq()`) does *not* have this bug because it does not depend on a buffer pool. When there are no packets to process, it returns 0 (no packets cleaned), which is less than the budget. The ZC path is different because it returns `budget` on allocation failure, creating the infinite loop.

6.2 Patch 2: `__IGB_DOWN` Check in `igb_tx_timeout()`

6.2.1 Commit Message

```
igb: skip reset in igb_tx_timeout() during XDP transition

When igb_xdp_setup() transitions between XDP and non-XDP mode on
a running device, it calls igb_close() followed by igb_open().
During this window the adapter is down and trans_start is stale,
so the TX watchdog can fire a spurious timeout.

The resulting schedule_work(&adapter->reset_task) races with the
igb_open() path: the reset task may run while the device is
being brought back up, or immediately after, causing unexpected
ring reinitialisation and register writes.

Fix this by checking __IGB_DOWN at the top of igb_tx_timeout().
```

6.2.2 Root Cause Analysis

During the `igb_xdp_setup()` close/open transition:

1. `igb_close()` calls `igb_down()`, which sets `__IGB_DOWN` and stops all TX queues.
2. The TX watchdog timer was armed before the close and fires during the transition window (or shortly after).
3. `dev_watchdog()` sees stopped queues with stale `trans_start` ⇒ calls `igb_tx_timeout()`.
4. `igb_tx_timeout()` unconditionally schedules `igb_reset_task()` via `schedule_work()`.
5. The reset work item may execute:
   - While `igb_open()` is still in progress (racing with ring setup).
   - After `igb_open()` completes (causing an unnecessary second reset).

Although `igb_reset_task()` does check `__IGB_DOWN` before calling `igb_reinit_locked()`, this check is not sufficient because `reset_task` is a work item that may not execute until after the XDP transition completes (by which time `__IGB_DOWN` has been cleared by `igb_open()`).

6.2.3 The Fix

```
static void igb_tx_timeout(struct net_device *netdev,
             unsigned int __always_unused txqueue)
{
    struct igb_adapter *adapter = netdev_priv(netdev);
    struct e1000_hw *hw = &adapter→hw;

   /* Do not schedule a reset if the adapter is
    * already going down or being reconfigured.
    */
   if (test_bit(__IGB_DOWN, &adapter→state))
       return;

    /* Do the reset outside of interrupt context */
    adapter→tx_timeout_count++;
    /* ... */
    schedule_work(&adapter→reset_task);
```

Listing 6.2: Patch 2 diff (igb_main.c)

6.2.4 Why It Is Correct

1. **Prevents spurious reset scheduling.** When the adapter is down (either shutting down or in the XDP transition window), a TX timeout is expected and not indicative of a hardware problem. Scheduling a reset is pointless: the subsequent `igb_open()` will reinitialize everything.

2. **Does not mask real timeouts.** Real TX hangs occur when the adapter is *up* and running. The `__IGB_DOWN` check only suppresses timeouts during shutdown and reconfiguration, when timeouts are known to be false positives.

3. **Eliminates the race with `igb_open()`.** By not scheduling the reset work item, we avoid the race where `igb_reset_task()` runs concurrently with or after `igb_open()`.

4. **Consistent with `igb_reset_task()`.** `igb_reset_task()` already checks `__IGB_DOWN` and bails out. This patch moves the guard one step earlier—to the point where the work is scheduled—to prevent the work item from being queued at all.

5. **No counter increment on spurious timeout.** By returning before `adapter->tx_timeout_count++`, we avoid inflating the timeout counter with false positives, which could confuse monitoring tools.

6.2.5 Why Not Just Rely on `igb_reset_task`'s Check?

One might ask: `igb_reset_task()` already checks `__IGB_DOWN` and returns early. Why add another check in `igb_tx_timeout()`?

The answer is **timing**. `igb_reset_task()` is a work item that runs asynchronously. Between the time it is scheduled and the time it runs, the XDP transition may complete:

1. `igb_tx_timeout()` schedules `reset_task()` (adapter is down, `__IGB_DOWN` is set).

2. `igb_xdp_setup()` completes: `igb_open()` clears `__IGB_DOWN`.

3. `reset_task()` runs: `__IGB_DOWN` is now clear, so it proceeds with `igb_reinit_locked()`, performing an unnecessary and potentially disruptive reset.

By checking __IGB_DOWN in igb_tx_timeout() itself, we prevent the work item from ever being queued during the transition window.

6.3 Patch 3: XDP Transition Guards in igb_xdp_setup()

6.3.1 Commit Message

```
igb: add XDP transition guards in igb_xdp_setup()

igb_xdp_setup() calls igb_close() + igb_open() when
transitioning between XDP and non-XDP mode on a running device.
This has two issues:

1. When removing an XDP program that has AF_XDP zero-copy
   sockets, ndo_xsk_wakeup() may be executing concurrently under
   rcu_read_lock(). If igb_close() tears down the rings while
   ndo_xsk_wakeup() is still accessing them, it races with the
   teardown. Add synchronize_rcu() before igb_close() when
   removing an XDP program.

2. The igb_close()/igb_open() window leaves trans_start stale.
   Add netif_trans_update() after igb_open() to refresh the
   timestamp, and cancel_work() to drain any reset_task queued
   during the window.
```

6.3.2 The Three Sub-fixes

Patch 3 contains three logically distinct fixes, all applied to igb_xdp_setup():

Fix 3a: synchronize_rcu() Before Close

```
1      if (need_reset && running) {
2         if (!prog)
3             /* Wait until ndo_xsk_wakeup completes. */
4             synchronize_rcu();
5          igb_close(dev);
```

Listing 6.3: Fix 3a: RCU synchronization (igb_main.c)

Root cause: igb_xsk_wakeup() is called under rcu_read_lock() by the AF_XDP core. When removing an XDP program (prog = NULL), any in-flight igb_xsk_wakeup() calls may still be accessing the adapter's rings, interrupt registers, and NAPI state. If igb_close() tears down these resources while igb_xsk_wakeup() is running, we get a use-after-free or register access on disabled hardware.

Why the fix is correct:

- synchronize_rcu() waits for all existing RCU read-side critical sections to complete. Since igb_xsk_wakeup() runs under rcu_read_lock(), this ensures all in-flight wakeup calls have returned before we proceed to igb_close().
- The condition !prog ensures we only synchronize when *removing* an XDP program. When *adding* a program, there are no AF_XDP sockets yet (you cannot bind a ZC socket without an XDP program), so the race does not exist.
- synchronize_rcu() may sleep, but igb_xdp_setup() is called from process context (under RTNL lock), so sleeping is safe.

Fix 3b: `netif_trans_update()` After Open

```
    if (running)
        igb_open(dev);

    if (need_reset && running) {
        netif_trans_update(dev);
```

Listing 6.4: Fix 3b: Timestamp refresh (igb_main.c)

Root cause: After `igb_open()` completes, the TX queues are started but `trans_start` still has the stale value from before `igb_close()`. If the watchdog timer fires before the next actual transmission, it sees the stale timestamp and triggers a false timeout.

Why the fix is correct:

- `netif_trans_update()` sets `trans_start` on TX queue 0 to the current `jiffies`. This prevents the watchdog from seeing a stale timestamp.
- For the I210/I211 (which typically has a single TX queue), updating queue 0 is sufficient.
- The update happens after `igb_open()`, so the TX queue is already started and the timestamp is meaningful.

Fix 3c: `cancel_work(&adapter->reset_task)`

```
        cancel_work(&adapter→reset_task);
    }

    return 0;
}
```

Listing 6.5: Fix 3c: Cancel stale reset work (igb_main.c)

Root cause: Even with Patch 2's guard in `igb_tx_timeout()`, there is a small window where a timeout could fire and schedule a `reset_task()` before the `__IGB_DOWN` bit is checked. Or a timeout could have been scheduled from a previous event. This stale work item would run after `igb_open()` and cause an unnecessary reset.

Why the fix is correct:

- `cancel_work()` (a variant of `cancel_work_sync()`) ensures that any queued `reset_task()` is cancelled and, if it is currently executing, waits for it to complete.
- This is a belt-and-suspenders approach: Patch 2 prevents most spurious scheduling, and this `cancel_work()` catches anything that slipped through the cracks.
- Placing it after `igb_open()` ensures the device is fully up before we drain the work queue, so we don't accidentally cancel a legitimate reset.

6.4 How the Three Patches Work Together

The three patches form a defense-in-depth strategy against the close/open transition bugs:

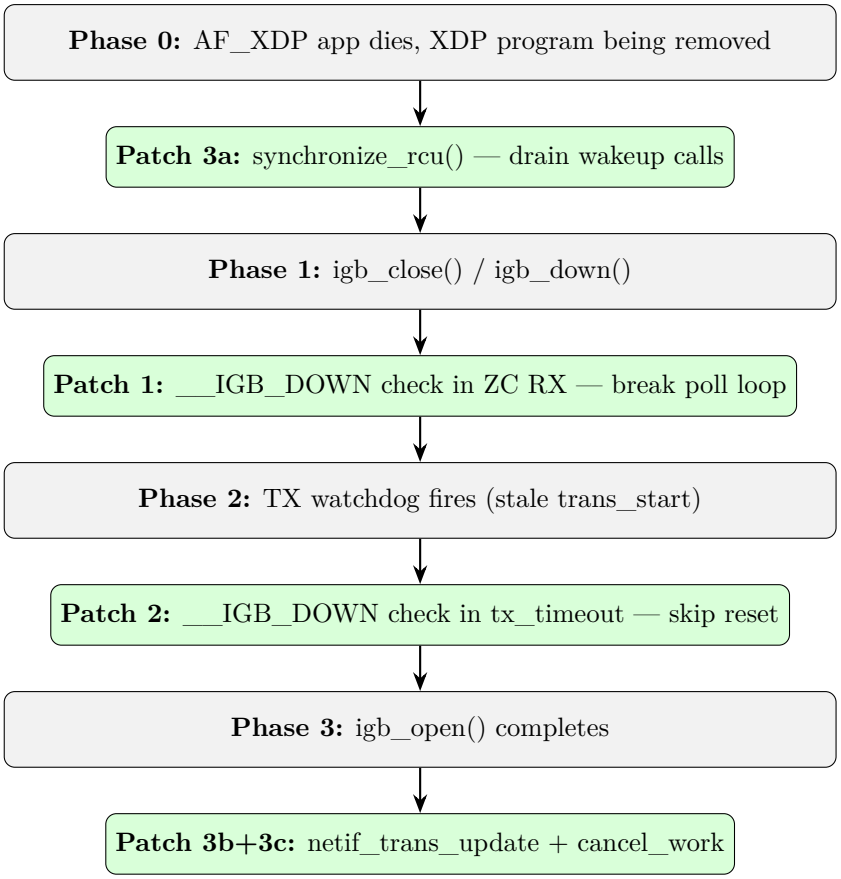

Figure 6.1: The three patches address different phases of the XDP transition.

Patch 1 (the critical fix)
Breaks the infinite NAPI poll loop that causes `igb_down()` to hang. Without this patch, nothing else matters because the driver is stuck forever.

Patch 2 (secondary defense)
Prevents the TX watchdog from scheduling a spurious reset during the transition. This is important even with Patch 1 because the watchdog can still fire due to the stale `trans_start` during the close/open window.

Patch 3 (transition hardening)
Addresses three remaining race conditions:

- RCU synchronization prevents use-after-free in `igb_xsk_wakeup()`.
- Timestamp refresh prevents false timeouts after the transition.
- Work cancellation catches any stale reset work items.

6.5 Testing

The patch series was tested on an Intel I210 controller with the following scenarios:

1. **Attach XDP program, run AF_XDP app, kill -9.** Before patches: permanent TX stall, requires reboot. After patches: clean recovery, interface returns to normal operation.
2. **Detach XDP program while AF_XDP is running.** Before patches: TX stall and/or kernel oops. After patches: clean recovery.

3. **Repeated attach/detach cycles.** Before patches: intermittent stalls and timeouts. After patches: stable operation across hundreds of cycles.

Appendix A

Full Patch Listing

This appendix contains the verbatim patch files as they would be submitted to the Linux kernel mailing list.

A.1 Patch 0/3: Cover Letter

```
From: Alex Dvoretsky <advoretsky@gmail.com>
Date: Sat, 14 Feb 2026 23:11:59 +0100
Subject: [PATCH net 0/3] igb: fix TX queue stall on XDP program
 removal with AF_XDP zero-copy

Killing an AF_XDP zero-copy application while the XDP program is still
attached causes a permanent TX queue 0 stall on igb. The interface
becomes unresponsive (no ping, no SSH) and requires a reboot or
module reload to recover.

Root cause: igb_clean_rx_irq_zc() has no __IGB_DOWN check, so when the
XSK buffer pool is destroyed (process dies) but NAPI keeps polling, the
function returns budget every time. This prevents napi_complete_done()
from signaling completion, which blocks napi_synchronize() in igb_down()
indefinitely. The 5-second TX watchdog then fires because TX completions
are not processed while NAPI is stuck.

The series has three patches:

1. Add __IGB_DOWN check in igb_clean_rx_irq_zc() -- breaks the infinite
   NAPI poll loop, matching the pattern in igb_clean_tx_irq().

2. Add __IGB_DOWN check in igb_tx_timeout() -- prevents spurious
   reset_task scheduling during the igb_close()/igb_open() window in
   XDP transitions.

3. Add RCU synchronization and TX watchdog guards in igb_xdp_setup() --
   ensures ndo_xsk_wakeup() completes before teardown and prevents
   false TX timeouts from stale trans_start.

Tested on Intel I210 (igb) with AF_XDP zero-copy sockets:
- Attach XDP program, run AF_XDP app, kill -9 -> no TX stall
- Detach XDP program while AF_XDP is running -> clean recovery
- Repeated attach/detach cycles -> stable

Alex Dvoretsky (3):
  igb: check __IGB_DOWN in igb_clean_rx_irq_zc()
  igb: skip reset in igb_tx_timeout() during XDP transition
  igb: add XDP transition guards in igb_xdp_setup()

 drivers/net/ethernet/intel/igb/igb_main.c | 22 ++++++++++++++++++++++
 drivers/net/ethernet/intel/igb/igb_xsk.c  |  3 +++
 2 files changed, 25 insertions(+)
```

A.2 Patch 1/3: `igb_clean_rx_irq_zc __IGB_DOWN` Check

```
From: Alex Dvoretsky <advoretsky@gmail.com>
Date: Sat, 14 Feb 2026 23:09:31 +0100
Subject: [PATCH net 1/3] igb: check __IGB_DOWN in igb_clean_rx_irq_zc()

When an AF_XDP zero-copy application terminates abruptly (e.g.,
kill -9), the XSK buffer pool is destroyed but NAPI polling continues.
igb_clean_rx_irq_zc() keeps returning budget (no descriptors, no
buffers to allocate, xsk_buff_alloc() returns NULL) which makes
napi_complete_done() re-arm the poll indefinitely.

Meanwhile, igb_down() -> napi_synchronize() waits for a NAPI poll cycle
that signals completion with done < budget -- which never happens. This
blocks igb_down() forever, and the 5-second TX watchdog fires because
no TX completions are processed while NAPI is stuck. Since igb_down()
never finishes, igb_up() is never called, and the TX queue remains
permanently stalled.

Fix this by adding an __IGB_DOWN check at the top of
igb_clean_rx_irq_zc(), returning 0 immediately when the adapter is
going down. This allows napi_synchronize() in igb_down() to complete,
matching the pattern already used in igb_clean_tx_irq().

Fixes: 2c6196013f84 ("igb: Add AF_XDP zero-copy Rx support")
Cc: stable@vger.kernel.org
Signed-off-by: Alex Dvoretsky <advoretsky@gmail.com>
---
 drivers/net/ethernet/intel/igb/igb_xsk.c | 3 +++
 1 file changed, 3 insertions(+)

diff --git a/drivers/net/ethernet/intel/igb/igb_xsk.c
            b/drivers/net/ethernet/intel/igb/igb_xsk.c
index 30ce5fbb5..ca4aa4d93 100644
--- a/drivers/net/ethernet/intel/igb/igb_xsk.c
+++ b/drivers/net/ethernet/intel/igb/igb_xsk.c
@@ -351,6 +351,9 @@ int igb_clean_rx_irq_zc(struct igb_q_vector *q_vector,
         u16 entries_to_alloc;
         struct sk_buff *skb;

+        if (test_bit(__IGB_DOWN, &adapter->state))
+                return 0;
+
         /* xdp_prog cannot be NULL in the ZC path */
         xdp_prog = READ_ONCE(rx_ring->xdp_prog);
```

A.3 Patch 2/3: `igb_tx_timeout __IGB_DOWN` Check

```
From: Alex Dvoretsky <advoretsky@gmail.com>
Date: Sat, 14 Feb 2026 23:09:47 +0100
Subject: [PATCH net 2/3] igb: skip reset in igb_tx_timeout() during
 XDP transition

When igb_xdp_setup() transitions between XDP and non-XDP mode on a
running device, it calls igb_close() followed by igb_open(). During
this window the adapter is down and trans_start is stale, so the TX
watchdog can fire a spurious timeout.

The resulting schedule_work(&adapter->reset_task) races with the
igb_open() path: the reset task may run while the device is being
brought back up, or immediately after, causing unexpected ring
reinitialisation and register writes.

Fix this by checking __IGB_DOWN at the top of igb_tx_timeout(). If the
adapter is down (either during normal close or during the XDP close/open
transition), there is nothing useful a reset can do -- the subsequent
igb_open() will reinitialise everything.

Fixes: 9cbc948b5a20 ("igb: add XDP support")
```

```
Cc: stable@vger.kernel.org
Signed-off-by: Alex Dvoretsky <advoretsky@gmail.com>
---
 drivers/net/ethernet/intel/igb/igb_main.c | 9 +++++++++
 1 file changed, 9 insertions(+)

diff --git a/drivers/net/ethernet/intel/igb/igb_main.c
            b/drivers/net/ethernet/intel/igb/igb_main.c
index dbea37269..e82f7184f 100644
--- a/drivers/net/ethernet/intel/igb/igb_main.c
+++ b/drivers/net/ethernet/intel/igb/igb_main.c
@@ -6652,6 +6652,15 @@ static void igb_tx_timeout(struct net_device
     *netdev, unsigned int __always_unus
         struct igb_adapter *adapter = netdev_priv(netdev);
         struct e1000_hw *hw = &adapter->hw;

+        /* Do not schedule a reset if the adapter is already going
+         * down or being reconfigured (e.g., XDP program transition
+         * via igb_close/igb_open). The stale trans_start from before
+         * the close will trigger a spurious timeout that resolves
+         * once igb_open() completes.
+         */
+        if (test_bit(__IGB_DOWN, &adapter->state))
+                return;
+
         /* Do the reset outside of interrupt context */
         adapter->tx_timeout_count++;
```

A.4 Patch 3/3: XDP Transition Guards in `igb_xdp_setup`

```
From: Alex Dvoretsky <advoretsky@gmail.com>
Date: Sat, 14 Feb 2026 23:11:25 +0100
Subject: [PATCH net 3/3] igb: add XDP transition guards in
 igb_xdp_setup()

igb_xdp_setup() calls igb_close() + igb_open() when transitioning
between XDP and non-XDP mode on a running device. This has two issues:

1. When removing an XDP program that has AF_XDP zero-copy sockets,
   ndo_xsk_wakeup() may be executing concurrently under rcu_read_lock().
   If igb_close() tears down the rings while ndo_xsk_wakeup() is still
   accessing them, it races with the teardown. Add synchronize_rcu()
   before igb_close() when removing an XDP program to ensure all
   in-flight ndo_xsk_wakeup() calls complete first.

2. The igb_close()/igb_open() window leaves trans_start stale from
   before the close: the TX watchdog can fire a spurious timeout and
   queue a reset_task that races with igb_open(). Add
   netif_trans_update() after igb_open() to refresh the timestamp, and
   cancel_work() to drain any reset_task queued during the window.

Fixes: 9cbc948b5a20 ("igb: add XDP support")
Cc: stable@vger.kernel.org
Signed-off-by: Alex Dvoretsky <advoretsky@gmail.com>
---
 drivers/net/ethernet/intel/igb/igb_main.c | 13 +++++++++++++
 1 file changed, 13 insertions(+)

diff --git a/drivers/net/ethernet/intel/igb/igb_main.c
            b/drivers/net/ethernet/intel/igb/igb_main.c
index e82f7184f..54a47a10d 100644
--- a/drivers/net/ethernet/intel/igb/igb_main.c
+++ b/drivers/net/ethernet/intel/igb/igb_main.c
@@ -2913,6 +2913,9 @@ static int igb_xdp_setup(struct net_device
     *dev, struct netdev_bpf *bpf)

         /* device is up and bpf is added/removed */
         if (need_reset && running) {
+                if (!prog)
+                        /* Wait until ndo_xsk_wakeup completes. */
```

```
+                       synchronize_rcu();
                igb_close(dev);
        } else {
                for (i = 0; i < adapter->num_rx_queues; i++)
@@ -2936,6 +2939,16 @@ static int igb_xdp_setup(struct net_device
     *dev, struct netdev_bpf *bpf)
        if (running)
                igb_open(dev);

+       /* Refresh trans_start to prevent the TX watchdog from
+        * firing on a stale timestamp from before igb_close().
+        * Cancel any reset_task that igb_tx_timeout() may have
+        * queued between igb_close() setting __IGB_DOWN and the
+        * actual napi_synchronize() completion.
+        */
+       if (need_reset && running) {
+               netif_trans_update(dev);
+               cancel_work(&adapter->reset_task);
+       }
+
        return 0;
 }
```

Appendix B

How to Apply and Test the Patches

B.1 Prerequisites

- A Linux kernel source tree (v6.1 or later).
- An Intel I210 or I211 network controller using the `igb` driver.
- An AF_XDP-capable test tool such as `xdpsock` (from the kernel's `samples/bpf/` directory) or `xdp-tools`.
- A simple XDP program (e.g., `xdp_pass`).

B.2 Applying the Patches

Navigate to the kernel source directory and apply:

```
$ cd ~/linux
$ git am ~/patches/0001-igb-check-__IGB_DOWN-in-igb_clean_rx_irq_zc.patch
$ git am ~/patches/0002-igb-skip-reset-in-igb_tx_timeout-during-XDP-transiti.patch
$ git am ~/patches/0003-igb-add-XDP-transition-guards-in-igb_xdp_setup.patch
```

Alternatively, apply all three at once:

```
$ git am ~/patches/000*.patch
```

B.3 Building and Installing

Build only the `igb` module:

```
$ make -C /lib/modules/$(uname -r)/build \
    M=drivers/net/ethernet/intel/igb modules
$ sudo modprobe -r igb
$ sudo insmod drivers/net/ethernet/intel/igb/igb.ko
```

Or rebuild and install the full kernel:

```
$ make -j$(nproc)
$ sudo make modules_install install
$ sudo reboot
```

B.4 Testing Procedure

B.4.1 Test 1: Kill AF_XDP Application

```
# Load XDP program
$ sudo ip link set dev enp8s0 xdp obj xdp_pass.o sec xdp

# Start AF_XDP zero-copy application
$ sudo xdpsock -i enp8s0 -q 0 -z &
$ AF_XDP_PID=$!

# Kill the application
$ sudo kill -9 $AF_XDP_PID

# Verify: no TX stall, interface recovers
$ ping -c 3 <gateway_ip>
$ dmesg | grep -i "watchdog\|timeout\|stall"
```

B.4.2 Test 2: Remove XDP Program While AF_XDP Is Running

```
$ sudo ip link set dev enp8s0 xdp obj xdp_pass.o sec xdp
$ sudo xdpsock -i enp8s0 -q 0 -z &

# Remove XDP program while application is running
$ sudo ip link set dev enp8s0 xdp off

# Verify: clean recovery
$ ping -c 3 <gateway_ip>
```

B.4.3 Test 3: Repeated Transitions

```
for i in $(seq 1 100); do
    sudo ip link set dev enp8s0 xdp obj xdp_pass.o sec xdp
    sudo xdpsock -i enp8s0 -q 0 -z &
    PID=$!
    sleep 0.5
    sudo kill -9 $PID
    wait $PID 2>/dev/null
    sudo ip link set dev enp8s0 xdp off 2>/dev/null
    sleep 0.1
done

# Verify: interface stable after all iterations
$ ping -c 3 <gateway_ip>
$ ethtool -S enp8s0 | grep tx_timeout
```

B.5 Expected Results

Test	**Before Patches**	**After Patches**
Kill AF_XDP app	TX stall, reboot needed	Clean recovery
Remove XDP while ZC	TX stall or oops	Clean recovery
Repeated transitions	Intermittent stalls	Stable

Appendix C

How to Submit Patches to the Linux Kernel Mailing List

C.1 Formatting

Linux kernel patches must follow a strict format. The `git format-patch` command generates properly formatted patches:

```
$ git format-patch -3 --cover-letter -v1 -o outgoing/
```

This creates:

- `v1-0000-cover-letter.patch` — edit the subject and body.
- `v1-0001-igb-check-__IGB_DOWN-in-igb_clean_rx_irq_zc.patch`
- `v1-0002-igb-skip-reset-...patch`
- `v1-0003-igb-add-XDP-transition-guards-...patch`

C.2 Running `checkpatch.pl`

Before submitting, verify that the patches pass the kernel's style checker:

```
$ ./scripts/checkpatch.pl outgoing/*.patch
```

Fix any warnings or errors. The most common issues are:

- Lines exceeding 80 characters (now 100 in some subsystems).
- Missing `Signed-off-by` line.
- Incorrect indentation (tabs, not spaces, for C code).

C.3 Identifying the Maintainers

Use `get_maintainer.pl` to find the correct mailing lists and maintainers:

```
$ ./scripts/get_maintainer.pl outgoing/v1-0001-*.patch
Tony Nguyen <anthony.l.nguyen@intel.com> (maintainer)
intel-wired-lan@lists.osuosl.org (list)
netdev@vger.kernel.org (list)
```

C.4 Sending with git send-email

```
$ git send-email \
    --to intel-wired-lan@lists.osuosl.org \
    --cc netdev@vger.kernel.org \
    --cc anthony.l.nguyen@intel.com \
    outgoing/v1-*.patch
```

Key options:

- `-to`: primary mailing list (subsystem list).
- `-cc`: secondary lists and individual maintainers.
- `-annotate`: allows editing each email before sending.

C.5 The Review Process

After submitting:

1. Wait for automated CI bots (e.g., `patchwork`, kernel test robot) to report build/test results.
2. Maintainers and reviewers will reply on the mailing list with comments, questions, or `Reviewed-by`/`Acked-by` tags.
3. Address feedback by sending a `v2` series with `git format-patch -v2`.
4. Once accepted, the patches flow through the maintainer's tree (e.g., `intel-wired-lan/net`) into `net.git`, and eventually into Linus's tree.

C.6 Backporting to Stable

The `Cc: stable@vger.kernel.org` tag in the commit message signals that the patch should be backported to stable kernel releases. The stable team will pick it up automatically after it lands in Linus's tree, or you can request a manual backport via:

```
$ git send-email --to stable@vger.kernel.org \
    --in-reply-to=<original-message-id> \
    outgoing/v1-*.patch
```