Netfilter vs struct Packettype

1 Introduction

Netfilter is a framework inside the Linux kernel that provides packet filtering, network address translation (NAT), and other packet mangling capabilities. It allows developers and administrators to intercept, examine, and manipulate network packets as they traverse the networking stack. Netfilter is commonly used to implement firewalls, packet filters, and traffic shaping modules.

At its core, Netfilter operates by registering **hook functions** at specific points in the packet processing path of the Linux kernel. These hooks enable kernel modules to process packets at different stages, allowing fine-grained control over network traffic.

2 Netfilter Hook Mechanism

Netfilter organizes packet processing using **hook points**, which correspond to well-defined stages in the network stack. Each hook point allows registered functions to inspect and optionally modify packets.

2.1 Hook Points

Netfilter defines the following primary hook points for IPv4 packets:

- **NF_INET_PRE_ROUTING**: Triggered immediately after a packet is received, before routing decisions. Typically used for packet filtering before routing.
- **NF_INET_LOCAL_IN**: Invoked for packets destined for the local host. Commonly used to filter incoming packets before they reach the socket layer.
- **NF_INET_FORWARD**: Called for packets that are routed through the host (i.e., not destined locally). Useful for firewall rules on a router or gateway.
- **NF_INET_LOCAL_OUT**: Triggered for packets generated locally before they are sent out. Useful for modifying or filtering outgoing traffic.
- **NF_INET_POST_ROUTING**: Invoked after routing and just before the packet leaves the network interface. Commonly used for final packet modifications.

2.2 Registration of Hooks

Kernel modules register hooks using the `struct nf_hook_ops` structure:

```
static struct nf_hook_ops my_hook_ops = {
    .hook     = my_hook_function,
    .pf       = PF_INET,
    .hooknum  = NF_INET_PRE_ROUTING,
    .priority = NF_IP_PRI_FIRST,
};

nf_register_net_hook(&init_net, &my_hook_ops);
nf_unregister_net_hook(&init_net, &my_hook_ops);
```

3 Packet Buffering in Netfilter

Hooks receive packets in the form of **sk_buff (socket buffer)** structures, which represent network packets in the kernel. The `sk_buff` contains:

- Header pointers: Ethernet, IP, and transport headers.
- Data buffer: Actual packet payload.
- Metadata: Routing information, protocol type, checksum, etc.

4 Packet Verdict Macros

Hook functions return a **verdict** to tell Netfilter what to do with the packet:

- **NF_ACCEPT**: Allow the packet to continue.
- **NF_DROP**: Discard the packet.
- **NF_STOLEN**: Hook has taken ownership; Netfilter will not process it further.
- **NF_QUEUE**: Queue the packet to user space.
- **NF_REPEAT**: Retry the hook.

Example:

```
unsigned int my_hook_function(void *priv,
                              struct sk_buff *skb,
                              const struct nf_hook_state *state)
{
    if (!skb) return NF_ACCEPT;

    struct iphdr *iph = ip_hdr(skb);
    if (iph && iph->protocol == IPPROTO_ICMP) {
        return NF_DROP;
    }
    return NF_ACCEPT;
}
```

5 Return Values in Netfilter Hook Functions

Netfilter hook functions have the following prototype:

```
unsigned int (*hook)(void *priv,
                     struct sk_buff *skb,
                     const struct nf_hook_state *state);
```

The return value (verdict) determines how the packet will be processed by the Netfilter framework and the networking stack.

Verdict Macro	Meaning	Kernel Effects and Changes
`NF_ACCEPT`	Accept the packet and continue normal processing.	The packet is passed to the next hook or the next stage of the networking stack. No automatic counter changes unless explicitly updated in the handler.
`NF_DROP`	Silently discard the packet.	The kernel frees the `skb` and halts further processing. Device or protocol drop counters may be incremented depending on configuration.
`NF_STOLEN`	Take ownership of the packet buffer.	The handler assumes responsibility for freeing the `skb`. Netfilter stops processing the packet. This is commonly used for asynchronous processing or packet redirection.
`NF_QUEUE`	Queue the packet to userspace via NFQUEUE.	The packet is handed to the userspace process for inspection or modification. Packet remains in kernel buffers until verdict is returned from userspace. May cause additional memory usage and latency.
`NF_REPEAT`	Reinvoke the same hook for the packet.	The hook function is executed again for the same packet. Useful for deferred decisions but may increase CPU load if overused.

Kernel Structures Affected

- `sk_buff` life cycle: Depending on the verdict, the `skb` may be freed, queued, or retained.
- Netfilter queue buffers (when using `NF_QUEUE`).
- Drop counters in `struct net_device->stats` when packets are discarded.

6 The nf_hook_state Structure

The nf_hook_state struct provides context for the hook function, including the network namespace, input/output devices, and the hook number:

```
struct nf_hook_state {
    struct net        *net;         // Network namespace
    struct sock       *sk;          // Socket associated with
       ↪ packet
    const struct nf_hook_ops *hook; // Hook operations
    struct net_device *in;          // Input device
    struct net_device *out;         // Output device
    u_int8_t          pf;           // Protocol family (PF_INET /
       ↪ PF_INET6)
    u_int8_t          hook;         // Hook number
    u_int8_t          okfn;         // Next function to call
};
```

This structure allows hook functions to make decisions based on the packet's routing, the device it arrived on, and the socket context if relevant.

7 Summary

Netfilter is a modular kernel framework for packet filtering. Packets traverse defined hook points where kernel modules can inspect, modify, or drop them. Hooks receive sk_buff packets and return a verdict. The nf_hook_state struct provides context for each packet, enabling flexible and powerful network control within the kernel.

8 The packet_type Structure

While Netfilter hooks operate at higher layers in the networking stack, the `packet_type` structure is part of the lower-level packet reception mechanism in the Linux kernel. It defines handlers for specific types of packets (based on Ethernet protocol types) and is used by subsystems such as ARP, IPv4, and IPv6 to register interest in certain protocol frames.

8.1 Structure Definition

The definition of `packet_type` (simplified) is as follows:

```
struct packet_type {
    __be16            type;         // Ethernet protocol in big-
       ↪ endian
    struct net_device *dev;          // Bound network device (NULL
       ↪  for all)
    int (*func)(struct sk_buff *skb,
                struct net_device *dev,
                struct packet_type *pt,
                struct net_device *orig_dev); // Packet handler
    struct list_head  list;          // List node for registration
    struct net_device *af_packet_priv;
    bool              ignore_outgoing; // Ignore locally
       ↪ generated packets
};
```

8.2 Field Descriptions

- **type**: The protocol identifier, typically one of the `ETH_P_*` constants defined in `linux/if_ether.h` (e.g., `ETH_P_IP` for IPv4, `ETH_P_ARP` for ARP).
- **dev**: A pointer to a specific `net_device` if the handler should only receive packets from that device; set to `NULL` to receive from all devices.
- **func**: The callback function invoked when a matching packet is received. The function processes the `sk_buff` and may consume, modify, or drop it.
- **list**: Used internally to store this `packet_type` in the kernel's protocol handler table.
- **ignore_outgoing**: If true, the handler ignores packets generated locally and only processes incoming packets.

8.3 Registration and Usage

Modules register their interest in certain packet types using `dev_add_pack()` and remove them with `dev_remove_pack()`.

Example: Registering a handler for IPv4 packets.

```
static int my_packet_rcv(struct sk_buff *skb,
                         struct net_device *dev,
                         struct packet_type *pt,
                         struct net_device *orig_dev)
{
    pr_info("Got␣packet␣of␣length␣%u␣from␣%s\n",
            skb->len, dev->name);
    return NET_RX_SUCCESS; // Indicate successful reception
}

static struct packet_type my_packet_type __read_mostly = {
    .type = cpu_to_be16(ETH_P_IP),
    .func = my_packet_rcv,
};

static int __init my_module_init(void)
{
    dev_add_pack(&my_packet_type);
    return 0;
}

static void __exit my_module_exit(void)
{
    dev_remove_pack(&my_packet_type);
}
```

9 Return Values in `struct packet_type` Callback

The `func` callback registered in `struct packet_type` has the following prototype:

```
int (*func)(struct sk_buff *skb,
            struct net_device *dev,
            struct packet_type *pt,
            struct net_device *orig_dev);
```

Its return value determines how the kernel handles the received packet and whether it continues processing it through other protocol handlers.

Return Value	**Meaning**	**Kernel Effects and Changes**
`NET_RX_SUCCESS (0)`	Packet successfully processed; ownership of the `skb` is transferred to the handler.	The handler must free the `skb` (e.g., with `kfree_skb()`) to avoid memory leaks. Packet processing chain may continue for other matching handlers (e.g., `ptype_all`). No automatic counter increments unless explicitly updated by the handler.
`NET_RX_DROP (1)`	Packet is intentionally dropped or could not be processed.	Kernel frees the `skb`. Device statistics counters are updated: `dev->stats.rx_dropped++`. Packet processing stops for this packet.
`NET_RX_BAD (2)`	Packet is considered malformed or invalid.	Kernel frees the `skb`. Device error counters are updated: `dev->stats.rx_errors++` and potentially more specific fields (e.g., `rx_crc_errors`, `rx_frame_errors`). Packet processing stops.

Kernel Structures Affected

- `struct net_device->stats` counters such as `rx_packets`, `rx_dropped`, and `rx_errors`.
- `skb->users` reference counter, which determines when the buffer is released.
- Packet processing chain flow (continuation or termination).

10 Advantages and Disadvantages of Netfilter

10.1 Advantages (Pros)

- **Kernel-level packet filtering:** Operates entirely in kernel space, avoiding user-space context switches and providing high performance.
- **Multi-layer support:** Can operate at both L3/L4 (IP, TCP/UDP) and L2 in bridging mode (`PF_BRIDGE`).
- **Modular architecture:** Hook points allow selective packet interception at desired stages of the networking stack.
- **User-space communication:** Packets can be sent to user-space via `NF_QUEUE` for deep inspection or advanced processing.
- **Protocol agnostic:** Supports IPv4 (`PF_INET`), IPv6 (`PF_INET6`), and bridging (`PF_BRIDGE`).
- **Direct `sk_buff` access:** Full control over both packet headers and payload.

10.2 Disadvantages (Cons)

- **Limited early L2 access:** Default usage focuses on L3/L4; true early L2 processing may require `struct packet_type` or driver-level hooks.
- **Performance impact:** Poorly optimized hook functions can degrade network throughput.
- **Development complexity:** Kernel-level code is harder to debug; mistakes can crash the system.
- **Hook ordering issues:** Multiple hooks at the same stage require careful `priority` management to avoid conflicts.
- **Protocol-specific hooks:** Family-specific design means IPv4 and IPv6 often require separate code paths.
- **Security risks:** Incorrect hook logic may drop or misroute packets, causing connectivity issues.
- **Requires deep API knowledge:** Understanding of `sk_buff`, `nf_hook_ops`, and `nf_hook_state` is essential.

11 Advantages and Disadvantages of `struct packet_type`

11.1 Advantages (Pros)

- **Early L2 packet interception:** Operates before the IP layer, allowing direct access to Ethernet headers and raw frame data.
- **Protocol-specific registration:** Can be bound to specific Ethernet protocol types (e.g., `ETH_P_IP`, `ETH_P_ARP`).
- **Works outside Netfilter:** Does not rely on Netfilter hooks, making it suitable for custom L2 packet processing pipelines.
- **High performance for specialized filtering:** Minimal processing overhead when only specific protocol frames are needed.
- **Direct `sk_buff` manipulation:** Full access to both packet payload and metadata at the earliest possible stage in the receive path.
- **Flexible protocol handling:** Can process non-IP protocols like ARP, custom Ethernet types, or experimental network stacks.

11.2 Disadvantages (Cons)

- **No built-in NAT or connection tracking:** Lacks higher-level features available in Netfilter.
- **Lower abstraction level:** Requires manual parsing and handling of protocols; no automatic header helpers like in IP stack processing.
- **Limited integration with firewall tools:** Not directly usable with `iptables` or `nftables`.
- **Potential redundancy:** May duplicate work already performed by higher layers if combined with Netfilter hooks.
- **Security and stability risks:** Incorrect parsing or buffer handling can corrupt packets or crash the kernel.
- **Less documentation and community examples:** Compared to Netfilter, fewer real-world use cases and learning resources exist.

12 Comparison: Netfilter vs. `struct packet_type`

Feature	**Netfilter**	`struct packet_type`
Layer of Operation	Primarily L3/L4 (IP/TCP/UDP), L2 in bridging mode (`PF_BRIDGE`)	L2 (Ethernet frame level) before IP stack processing
Access to Headers	IP and transport headers by default; Ethernet headers available in certain modes	Full Ethernet header + payload access from the start
Built-in Features	NAT, connection tracking, `NF_QUEUE` to user space	None; purely a packet reception mechanism
Hook Mechanism	Defined hook points (`NF_INET_PRE_ROUTING`, etc.)	Protocol-specific registration via `ptype_all` or specific `ETH_P_*` types
Performance	Slightly more overhead due to higher-layer processing	Lower latency for raw L2 packet handling
Protocol Support	IPv4, IPv6, bridging	Any L2 protocol (ARP, custom EtherTypes, IP, etc.)
Tool Integration	Fully compatible with `iptables`, `nftables`	Not directly compatible with firewall tools
Use Cases	Firewalls, NAT, traffic shaping, packet mangling	Custom protocol analyzers, L2 filtering, raw frame processing
Ease of Development	Higher-level API with helper functions; more documentation	Lower-level API, manual parsing required
Risks	Misconfigured hooks can cause packet drops or routing issues	Buffer mismanagement can corrupt frames or crash kernel

13 Effect of NF_DROP on Device Statistics: L2 vs L3

Aspect	**L2 (struct packet_type / PF_BRIDGE)**	**L3 (Netfilter / PF_INET)**
Packet Level	Ethernet frame level	IP packet level
Drop Point	Driver callback or early L2 hook	Netfilter hook in IP stack
rx_packets / rx_success	Incremented by driver until callback; may need manual decrement if packet dropped early	Incremented by driver; NF_DROP does not affect `rx_packets`
rx_dropped	Automatically updated if driver drop occurs; can be manually incremented in callback	Not updated automatically; must manually increment if you want to reflect L3 drop in stats
sk_buff ownership	Callback owns skb; must free if consumed	NF_DROP frees skb within Netfilter path
CPU / Processing Impact	Minimal if filtering few EtherTypes; early drop reduces upper layer processing	Slightly higher; packet traverses IP stack until hook point

14 Process Power and Buffer Usage Comparison

Feature	**Netfilter**	`struct packet_type`
Process Power	Operates at L3/L4; higher CPU usage due to NAT, connection tracking, and multiple hook processing. Stateful operations increase processing load.	Operates at L2; generally lower CPU cost since packets are intercepted before IP stack. Processing cost rises if registered with `ptype_all` for all frames.
Buffer Usage	Uses `sk_buff` optimized for IP layer; may require buffer cloning for NAT/queue. Ethernet headers may be stripped in some cases; connection tracking adds extra metadata.	Accesses raw Ethernet frame in `sk_buff` without additional abstraction; no automatic metadata overhead, but manual parsing required. Care needed to avoid buffer overflows.