Linux Packet Sniffer

Rationale

During my internship at ST Engineering, I frequently used tcpdump to inspect and analyze network traffic, which generated curiosity in learning how these kinds of tools actually work behind the scenes. Rather than relying solely on high-level tools, I was interested in learning the packet capture mechanism at a low level. This curiosity led me to implement my own packet sniffer from scratch using raw sockets and Berkeley Packet Filters (BPF) to learn the Linux networking stack, and kernel-level filtering.

New Things I Learnt

Understanding PF_PACKET Sockets

Before starting this project, I was mostly familiar with sockets in the context of `AF_INET` and `SOCK_STREAM`. This is the typical setup for TCP and UDP communications. However, I quickly realized that building a packet sniffer required a much lower-level interface. I learned that unlike standard sockets, `PF_PACKET` sockets allow direct access to raw Ethernet frames at Layer 2, bypassing the normal TCP/IP stack.

While digging into PF_PACKET, I gained a clearer understanding of how packets move through the Linux kernel. From the moment a packet hits the NIC (Network Interface Card), it's copied via DMA into kernel memory and wrapped in a structure called `sk_buff`. Normally, this data travels up through the IP and TCP/UDP layers, eventually reaching user-space applications via higher-level socket APIs. But with a PF_PACKET socket, the kernel clones the packet and lets my program access it directly, skipping those protocol layers.

Binding to a Specific Interface

Another thing I learned was how to bind a raw socket to a specific network interface (like `eth0`). Without this step, the sniffer listens to all interfaces, which isn't ideal when debugging or targeting specific traffic. Using `setsockopt()` with the `SO_BINDTODEVICE` option allowed me to precisely target packets received on one interface.

Enabling Promiscuous Mode

Perhaps the most interesting part was learning how to enable promiscuous mode on the NIC. By default, a NIC ignores packets not addressed to it, which defeats the purpose of sniffing all network traffic. Using `ioctl()` system calls and manipulating interface flags (`SIOCGIFFLAGS` and `SIOCSIFFLAGS`), I was able to configure the NIC to accept all packets on the network.

How tcpdump and libpcap Really Work

Lastly, I began to appreciate what tools like `tcpdump` and libraries like `libpcap` are doing under the hood. I had used these tools countless times without thinking twice about what made them tick. Now, I understand that libpcap leverages PF_PACKET sockets, and likely uses similar interface binding and promiscuous mode logic internally.

BPF

I'd always seen `tcpdump` filters in action but never understood how that worked under the hood. This project helped me understand that BPF is not just a helper. It's a virtual machine embedded in the Linux kernel, processing packets *before* they ever reach user space.

I found it fascinating that BPF is essentially a register-based virtual CPU with its own accumulator, index register, and program counter. Its job is to run a tiny program, expressed in assembly-like bytecode, to decide whether a packet should be accepted or dropped. It also showed me how critical it is to filter packets *early*, right when they arrive at the network interface. Otherwise, unnecessary copying from kernel space to user space could quickly become a bottleneck.

Example BPF assembly code:

```
;this BPF program filters and accepts all IP packets
(000) ldh      [12]
(001) jeq      #0x800          jt 2    jf 3
(002) ret      #262144
(003) ret      #0
```

Using `tcpdump -ddd`, I learned how to convert a human-readable filter (like "tcp" or "ip") into actual BPF bytecode, the language the kernel understands. This bytecode is then loaded into the socket via `setsockopt()` using the `SO_ATTACH_FILTER` option.

Seeing the Kernel Internals in Action

Looking through functions like `packet_rcv()` and `run_filter()` in the Linux source helped me visualize exactly when and where filtering happens. I also saw how the kernel ties a filter to a socket and uses `sk_run_filter()` to simulate the execution of BPF instructions.

```
/* Copied from net/packet/af_packet.c */
/* function run_filter is called in packet_rcv*/
static inline unsigned int run_filter(struct sk_buff *skb, struct sock *sk,
                    unsigned int res)
{
    struct sk_filter *filter;

    rcu_read_lock_bh();
    filter = rcu_dereference(sk->sk_filter); // get the filter bound to the socket
    if (filter != NULL)
        res = sk_run_filter(skb, filter->insns, filter->len); // the filtering is inside sk
_run_filter function
    rcu_read_unlock_bh();

    return res;
}
```

```
unsigned int sk_run_filter(struct sk_buff *skb, struct sock_filter *filter, int flen)
{
    struct sock_filter *fentry;   /* We walk down these */
    void *ptr;
    u32 A = 0;          /* Accumulator */
    u32 X = 0;          /* Index Register */
    u32 mem[BPF_MEMWORDS];      /* Scratch Memory Store */
    u32 tmp;
    int k;
    int pc;

    /*
     * Process array of filter instructions.
     */
    for (pc = 0; pc < flen; pc++) {
        fentry = &filter[pc];

        switch (fentry->code) {
        case BPF_ALU|BPF_ADD|BPF_X:
            A += X;
            continue;
        case BPF_ALU|BPF_ADD|BPF_K:
            A += fentry->k;
            continue;
        case BPF_ALU|BPF_SUB|BPF_X:
            A -= X;
            continue;
        case BPF_ALU|BPF_SUB|BPF_K:
            A -= fentry->k;
            continue;
        case BPF_ALU|BPF_MUL|BPF_X:
            A *= X;
            continue;
        /* some code omitted ... */
        case BPF_RET|BPF_K:
            return fentry->k;
        case BPF_RET|BPF_A:
            return A;
        case BPF_ST:
            mem[fentry->k] = A;
            continue;
        case BPF_STX:
            mem[fentry->k] = X;
            continue;
        default:
            WARN_ON(1);
            return 0;
        }
    }

    return 0;
}
```

Understanding Packet Structure More Deeply

To process the filtered packets, I had to dive deeper into the structure of Ethernet and IP headers. I parses the MAC addresses, checkes for IPv4 EtherType, and read source and destination IPs all from raw bytes. I also added transport layer protocol detection (like TCP or UDP) and learned how port numbers are extracted from the header using bitwise operations.

In doing so, I realized how critical it is to understand protocol formats to write meaningful sniffers. You can't just blindly read from a buffer, you need to know exactly where fields lie and what each byte represents.

Finally, wiring everything together, from creating the raw socket, setting promiscuous mode, applying the BPF filter, to reading filtered packets, gave me a new appreciation for how tools like Wireshark and tcpdump actually work under the hood.

Creating the Packet Sniffer

To begin building our packet sniffer, we include a collection of header files required for working with raw sockets, Ethernet interfaces, ioctl system calls, and Berkeley Packet Filter (BPF).

```
c
CopyEdit
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <linux/if_ether.h>
#include <linux/filter.h>
#include <net/if.h>
#include <arpa/inet.h>
```

To help translate numeric protocol values into human-readable names (like `tcp`, `udp`, `icmp`), we define a small utility function. This function maps values from the IP protocol field to strings:

```
char* transport_protocol(unsigned int code) {
    switch(code) {
        case 1: return "icmp";
        case 2: return "igmp";
        case 6: return "tcp";
        case 17: return "udp";
        default: return "unknown";
    }
}
```

Next, we create a raw socket using the `PF_PACKET` protocol family and bind it specifically to Ethernet frames that carry IP packets. We use `SOCK_RAW` to capture raw frames and `htons(ETH_P_IP)` to filter for IP-level protocols:

```
if ((sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_IP))) < 0) {
    perror("socket");
    exit(1);
}
```

To limit our capture to a specific interface (like `eth0`), we use the `SO_BINDTODEVICE` option with `setsockopt`. This ensures we only sniff traffic on the specified device:

```
const char *opt = "eth0";
if (setsockopt(sock, SOL_SOCKET, SO_BINDTODEVICE, opt, strlen(opt) + 1) <
0) {
    perror("setsockopt bind device");
    close(sock);
    exit(1);
}
```

However, by default, a network interface card (NIC) only processes frames destined for itself. To sniff *all* packets traversing the interface (regardless of destination), we need to enable promiscuous mode. This is done via the `ioctl` system call by first fetching the current flags and then adding the `IFF_PROMISC` bit:

```
struct ifreq ethreq;
strncpy(ethreq.ifr_name, "eth0", IF_NAMESIZE);
if (ioctl(sock, SIOCGIFFLAGS, &ethreq) == -1) {
    perror("ioctl get flags");
    close(sock);
    exit(1);
}
ethreq.ifr_flags |= IFF_PROMISC;
if (ioctl(sock, SIOCSIFFLAGS, &ethreq) == -1) {
    perror("ioctl set flags");
    close(sock);
    exit(1);
}
```

With the socket now able to receive all traffic on the interface, we apply a Berkeley Packet Filter (BPF) to reduce overhead by discarding unwanted packets in kernel space. This particular filter allows only TCP traffic. The bytecode was generated using `tcpdump -dd tcp` and specifies exact offsets to inspect in the packet structure:

```
struct sock_filter BPF_code[] = {
    { 0x28, 0, 0, 0x0000000c },
    { 0x15, 0, 5, 0x000086dd },
    { 0x30, 0, 0, 0x00000014 },
    { 0x15, 6, 0, 0x00000006 },
    { 0x15, 0, 6, 0x0000002c },
    { 0x30, 0, 0, 0x00000036 },
    { 0x15, 3, 4, 0x00000006 },
    { 0x15, 0, 3, 0x00000800 },
    { 0x30, 0, 0, 0x00000017 },
    { 0x15, 0, 1, 0x00000006 },
    { 0x6, 0, 0, 0x00040000 },
    { 0x6, 0, 0, 0x00000000 }
};
```

We wrap this filter in a `sock_fprog` structure and attach it to the socket using `setsockopt` and the `SO_ATTACH_FILTER` option:

```
struct sock_fprog Filter;
Filter.len = sizeof(BPF_code) / sizeof(BPF_code[0]);
Filter.filter = BPF_code;

if (setsockopt(sock, SOL_SOCKET, SO_ATTACH_FILTER, &Filter, sizeof(Filter))
< 0) {
    perror("setsockopt attach filter");
    close(sock);
    exit(1);
}
```

With the setup complete, we enter an infinite loop to capture and process packets using `recvfrom`. Each iteration reads a frame into a buffer:

```
while (1) {
    printf("-----------\n");
    n = recvfrom(sock, buffer, sizeof(buffer), 0, NULL, NULL);
    printf("%d bytes read\n", n);
```

We sanity-check the captured packet to ensure it's large enough to contain Ethernet, IP, and TCP/UDP headers (minimum 42 bytes):

```
        if (n < 42) {
        perror("recvfrom()");
        printf("Incomplete packet (errno is %d)\n", errno);
        close(sock);
        exit(0);
    }
```

Next, we parse the Ethernet header. The first 6 bytes are the source MAC address, the next 6 are the destination MAC address. These are displayed in a human-readable format:

```


    ethhead = buffer;
    printf("Source MAC address: %.2x:%.2x:%.2x:%.2x:%.2x:%.2x\n",
           ethhead[0], ethhead[1], ethhead[2],
           ethhead[3], ethhead[4], ethhead[5]);
    printf("Destination MAC address: %.2x:%.2x:%.2x:%.2x:%.2x:%.2x\n",
           ethhead[6], ethhead[7], ethhead[8],
           ethhead[9], ethhead[10], ethhead[11]);
```

Then we move 14 bytes ahead into the Ethernet payload, which should be the IP header. If the first byte of this header is `0x45`, we confirm it's an IPv4 packet with a 20-byte header (no IP options):

```
    iphead = buffer + 14;
    if (*iphead == 0x45) {
```

We extract and print the source and destination IP addresses, followed by source and destination port numbers, which are each 2 bytes long:

```
      printf("Source host %d.%d.%d.%d\n",
          iphead[12], iphead[13], iphead[14], iphead[15]);
      printf("Dest host %d.%d.%d.%d\n",
          iphead[16], iphead[17], iphead[18], iphead[19]);
      printf("Source,Dest ports %d,%d\n",
          (iphead[20] << 8) + iphead[21],
          (iphead[22] << 8) + iphead[23]);
```

Finally, we identify the Layer 4 protocol using the helper function defined earlier, which references the protocol field in the IP header (byte offset 9):

```
      printf("Layer-4 protocol %s\n", transport_protocol(iphead[9]));
    }
  }
```