Linux Raw Sockets

And some kernel stuff too...

Jaime Geiger

Sockets for scrubs (a.k.a. normal people)

//I can sock it

import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.bind(('', 6969))

s.listen(123)

s.accept()

read... write... etc... then close

Sockets for programmers (a.k.a. people that are unhealthily obsessed with C)

```
//how to socket
struct sockaddr_in serv, cli;
int fd = socket(AF_INET, SOCK_STREAM, 0);
memset(&serv, 0, sizeof(struct sockaddr_in);
serv.sin_port = htons(6969);
serv.sin_family = AF_INET;
serv.sin_addr.s_addr = INADDR_ANY;
bind(fd, (struct sockaddr*)&serv, sizeof(serv));
listen(fd, 5);
size_t len;
int clifd = accept(fd, (struct sockaddr*)&cli, &len);
read... write... etc etc. then close
```

As if that wasn't annoying enough...

- We do SOCK_RAW
- Why bother?
  - Learning and stuff
  - Sniffing is cool
  - Getting closer to the bits

Socket Buffers: Kernel Magic (50+ members)

```
struct sk_buff {
    struct sk_buff *next, *prev; ...
    struct sock *sk;
    struct net_device *dev; ...
    unsigned int len, data_len;
    __u16 mac_len, hdr_len; …
    union {__wsum csum; struct {__u16 csum_start; __u16
    csum_offset;};}; ...
    __be16 protocol;
    __u16 transport_header, network_header, mac_header; …
    sk_buff_data_t tail, end;
    unsigned char *head, *data;
}
```

How does this all happen?

- Network card gets packet, sends interrupt to the kernel
- Device driver → netif_rx(skb) →
- enqueue_to_backlog(skb, cpu, &qtail) →
  - if queue isn't full and flow rate is not met then add to per-cpu processing queue
  - calls soft irq (NET_RX_SOFTIRQ)
- dev_add_pack → ptype_head(pack_type)
  - Assigns packet type structure to the skb queue item
- net_rx_action(softirq_action_ptr) ...

net_rx_action(softirq_action_ptr)

- Confusing interrupt handler
- struct napi_struct *n;
  - Will contain the queued skb
- n = list_first_entry(&sd->poll_list, struct napi_struct, poll_list);
  - Gets queued packet
- Runs handler functions for packet types
  - ptype_all (ETH_P_ALL)
  - ptype_base[<type hash>] (ETH_P_*)
  - Each type is passed further up their respective protocol stacks

Sending Raw Packets

- You have to manually fill out EVERY FIELD
- There is some header help
  - Ethhdr, iphdr, udphdr, and tcphdr structs, combine them to make entire frames:

```
struct __attribute__((__packed__)) udpframe {
    struct ethhdr ehdr;
    struct iphdr ip;
    struct udphdr udp;
    unsigned char data[ETH_DATA_LEN - sizeof(struct udphdr) - sizeof(struct iphdr)];
};
```

**yeah where is that FCS, don't know, but things packed into this correctly will send :)

Sending Raw Packets

- sendto requires a sockaddr_ll (link layer) to send raw frames
  - Source MAC address, protocol family (PF_PACKET), hwaddr len, and interface index
  - ioctl(sockfd, SIOCGIFINDEX, sifreq) gets us the interface index in sifreq->ifindex
- Checksums suck, UDP doesn't require one, just set it to 0x0000
  - UDP is just easier to send in general, no handshakes to work out
- If you send raw UDP responses to sniffed frames you won't be able to receive them unless you block icmp port unreachable messages

Packet Sniffing

- Send an ioctl to the interface to set flags to allow promiscuous mode:

  *struct ifreq *sifreq;*

  int sockfd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_IP));

  strncpy(sifreq->ifr_name, “eth0”, IFNAMSIZ);

  *ioctl(sockfd, SIOC**GIFFLAGS**, sifreq);*

  sifreq->ifr_flags |= IFF_PROMISC;

  *ioctl(sockfd, SIOC**SIFFLAGS**, sifreq);*

- *Grab packets:*

  recvfrom(sockfd, ...)

PF_PACKET?

- Gets the specified packet type (ETH_P_IP) directly off of the interface driver instead of letting it go farther up the protocol stack
- Why is this good?
  - The sk_buffs get copied, so you can still browse the internet while you sniff! (literally in skb_copy)
  - The packet has not been modified or filtered in any way higher levels of the networking stack

Berkley Packet Filter (BPF)

- So say you don't want all of the IP packets...
  - (Trust me, you don't)
- Sockets can have filters applied to them with setsockopt
  - Filters at the socket level! No need to filter through in *your* code, **that's really slow**
- Use tcpdump to generate a filter

Making and Applying socket_filters

```
$ sudo tcpdump -dd udp and port 31337
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 6, 0x000086dd },
{ 0x30, 0, 0, 0x00000014 },
{ 0x15, 0, 15, 0x00000011 },
{ 0x28, 0, 0, 0x00000036 },
{ 0x15, 12, 0, 0x00007a69 },
{ 0x28, 0, 0, 0x00000038 },
{ 0x15, 10, 11, 0x00007a69 },
{ 0x15, 0, 10, 0x00000800 },
{ 0x30, 0, 0, 0x00000017 },
{ 0x15, 0, 8, 0x00000011 },
{ 0x28, 0, 0, 0x00000014 },
{ 0x45, 6, 0, 0x00001fff },
{ 0xb1, 0, 0, 0x0000000e },
{ 0x48, 0, 0, 0x0000000e },
{ 0x15, 2, 0, 0x00007a69 },
{ 0x48, 0, 0, 0x00000010 },
{ 0x15, 0, 1, 0x00007a69 },
{ 0x6, 0, 0, 0x0000ffff },
{ 0x6, 0, 0, 0x00000000 },

struct sock_filter bpf_code[] = {
    { the, filter, from, above },
};
```

```
struct sock_fprog filter;
filter.len = 20;
filter.filter = bpf_code;
setsockopt(sockfd, SOL_SOCKET,
SO_ATTACH_FILTER, &filter,sizeof(filter));
```

- The above will set the bpf filter on the socket.
- Stored in sock->sk_filter
- Now recvfrom(sockfd, …) will only get IP packets destined for UDP port 31337

Watershell

- A practical C implementation of most of this
- Run commands through iptables!
  - Sniffing pulls packets directly off the interface before iptables has a chance to examine them
  - Packets are passed through iptables after they are picked up by the interface
  - Special port and keyword used to trigger command execution via network (just a system() call for now)
- https://github.com/jgeigerm/watershell.git

You can do all of this with python

- Yeah python makes all of this a lot easier, but C is more hardcore or whatever
- Plus I like compiled binaries I can drop onto systems
  - You can do that with python too
    - ...