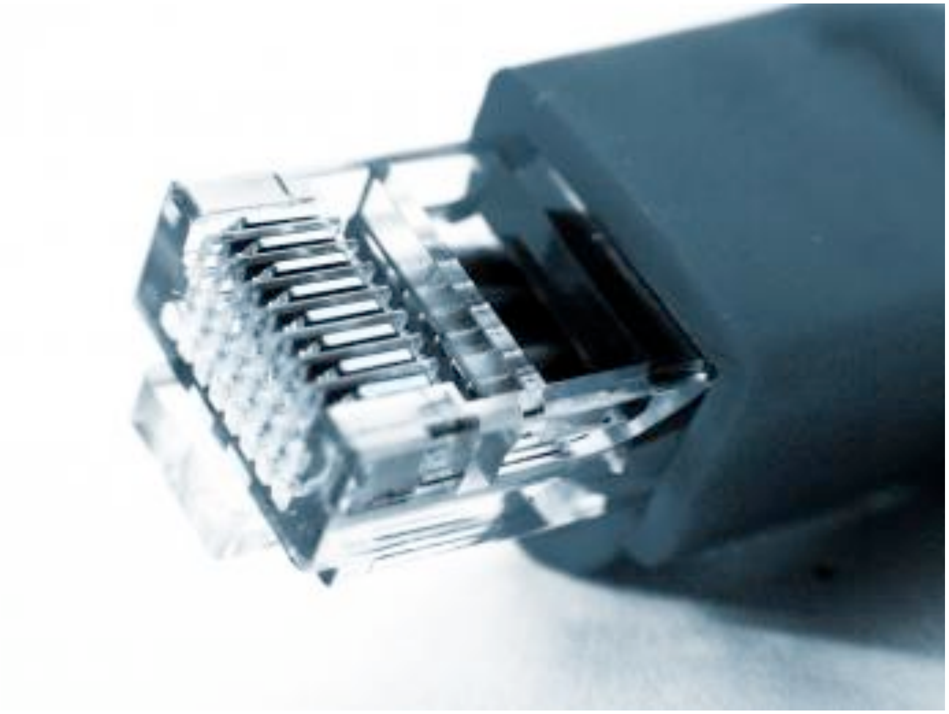
Packet Mastering the Monkey Way

Recon, 2005

jose nazario <jose@monkey.org>

Raw IP vs socket based networking



capture	send
reassemble	drive

pcap	dnet
nids	event

sniff	hijack
disrupt	reroute

Linux	UN*X	Mac OS X	Windows

cross platform

cross language

C/C++/C#	Python	Tcl	Perl

libevent – event wrapper library

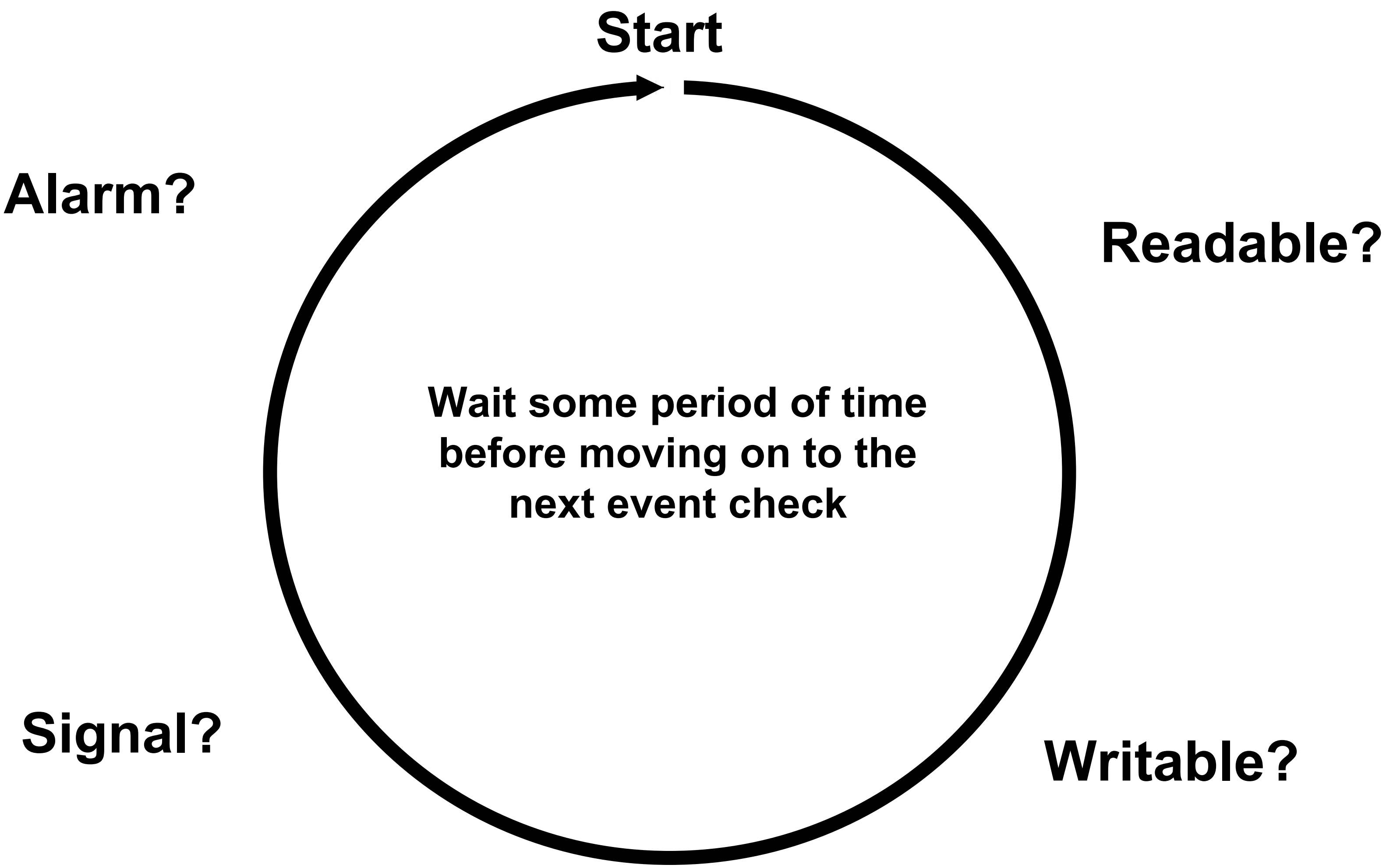
Abstract event framework

- **uses poll(), select(), epoll(), kqueue()**
- **optimized for target platform at libevent compile time**
- **write once, optimized everywhere**
- **handles signals and alarms, too**
- **works on file descriptors**

libevent programming basics:

1. Initialize event framework:	**`event_init()`**
2. Create event:	**`event_set()`**
3. Install event into list to check:	**`event_add()`**
4. Run the events:	**`event_dispatch()`**

The Main Event Loop



Start

Readable?

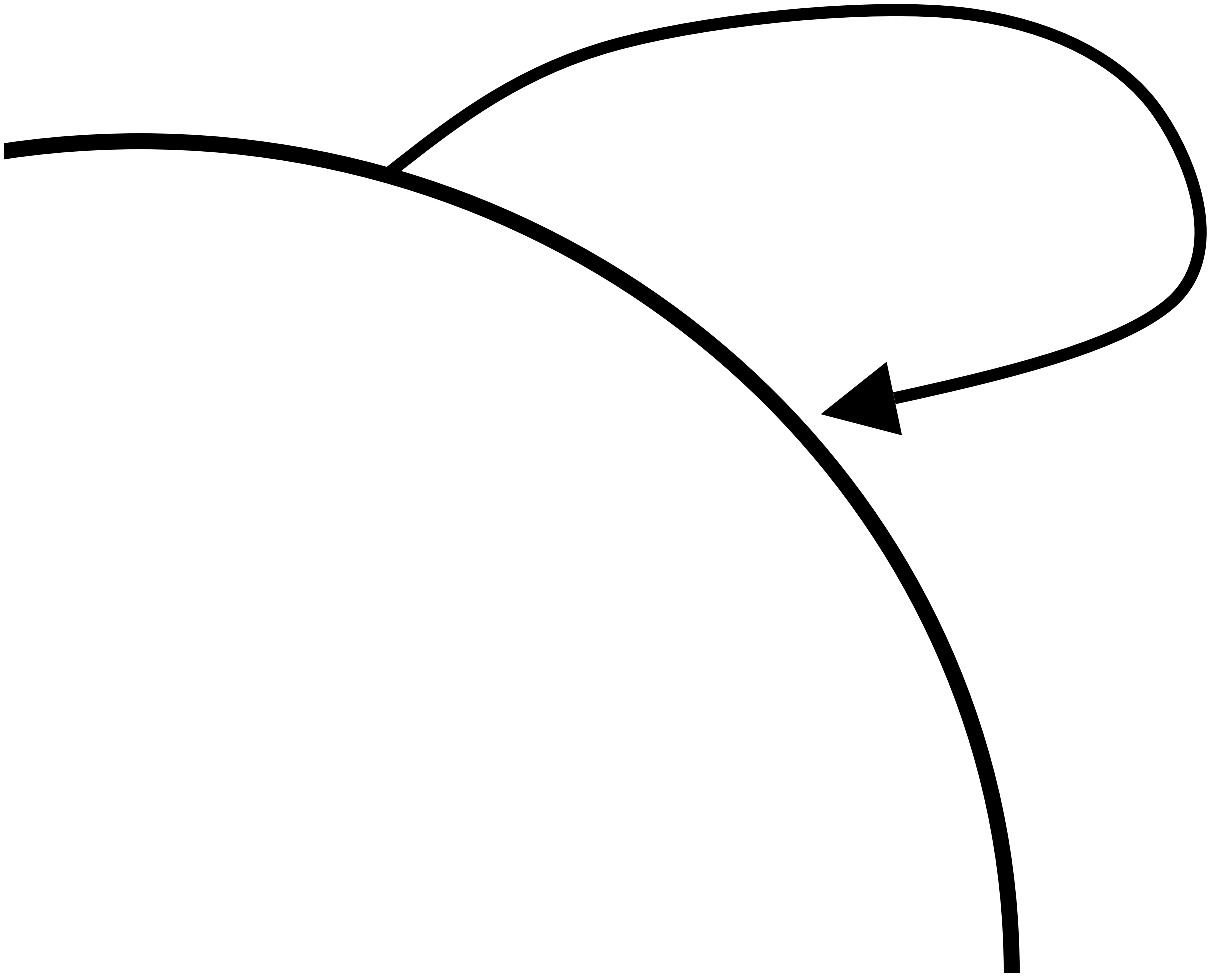
Writable?

Signal?

Alarm?

Wait some period of time before moving on to the next event check

When an Event Is Caught



Execute callback, return to main event loop.

Pass data to the event. Example: data to write to a file descriptor.

Event is removed from queue upon completion, unless EV_PERSIST is set or the event is rescheduled.

Events vs threads

Both used in high performance programming
Both excellent for high performance packet actions

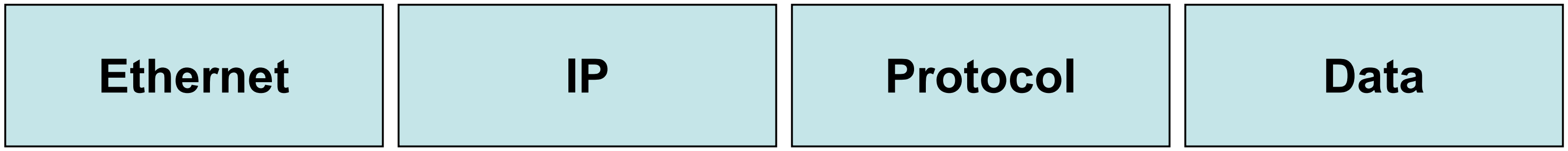
- Spawn threads for tasks (read, write, process)
- Any thread can wait until it has input, overall program still moves
- Threads are difficult to debug
- Threads can deadlock against each other
- Not all functions are thread safe, clobbering data
- Onus is on you to choreograph a careful dance, easy to mess up

- Main thread of execution loops over possible actions
- Actions include: read, write, signal, alarm
- Every possible action has an associated “callback”
- Callbacks process data
- Easy to debug, look at active event handler
- Deadlocks don’t happen, data not clobbered by stray thread
- Program is always doing something, or looking for something to do

Events for Packets

- Packets are read
- Packets are written
- Packet data is periodically processed
  - Rates (pps, bps)
- Timeouts (ie retransmits)
- Usually no need for multiple packets at once

IP Stack Structure



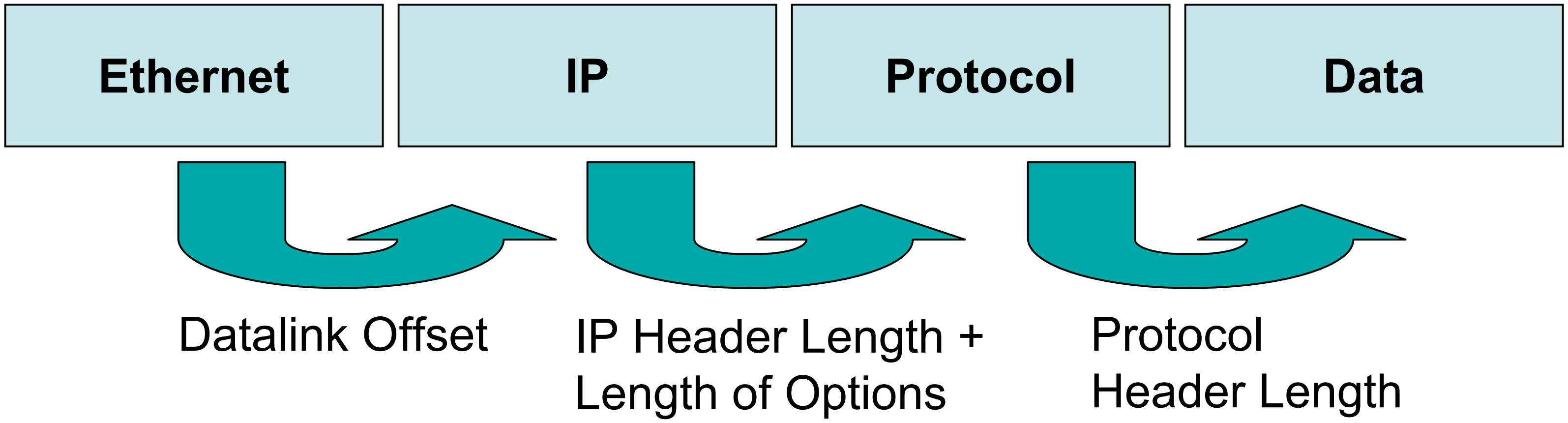
Ethernet

IP

Protocol

Data

Parsing Order



Ethernet

IP

Protocol

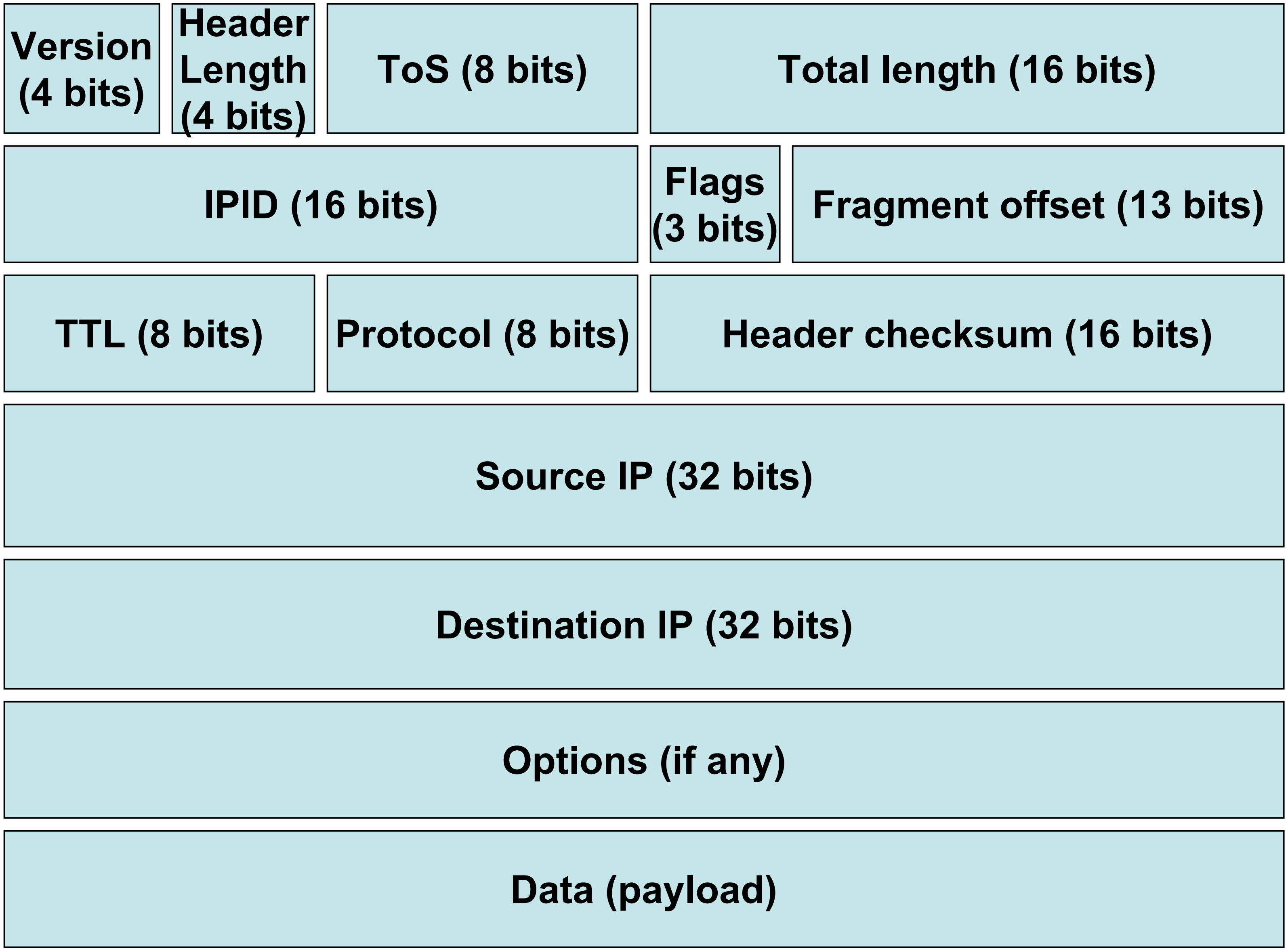
Data

Datalink Offset

IP Header Length +
Length of Options

Protocol
Header Length

IPv4 Header Structure



Version (4 bits)

Header Length (4 bits)

ToS (8 bits)

Total length (16 bits)

IPID (16 bits)

Flags (3 bits)

Fragment offset (13 bits)

TTL (8 bits)

Protocol (8 bits)

Header checksum (16 bits)

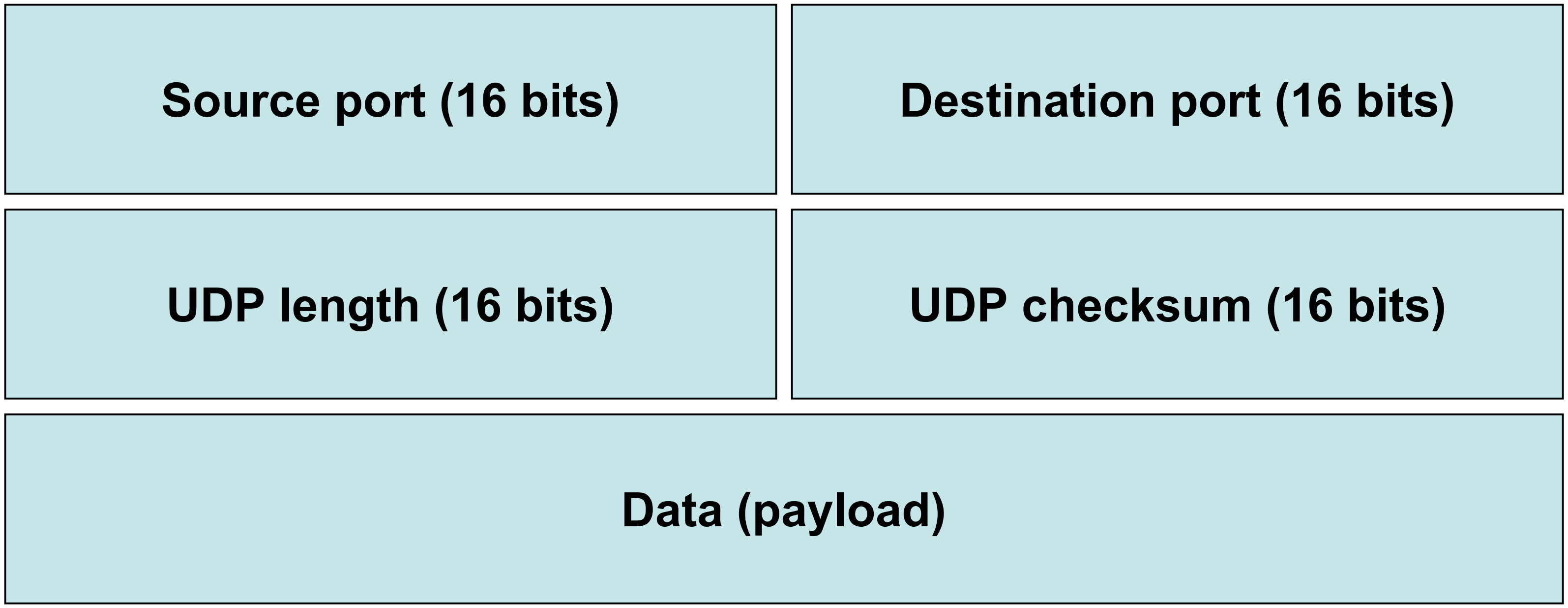
Source IP (32 bits)

Destination IP (32 bits)

Options (if any)

Data (payload)

UDP Header Structure



Source port (16 bits)

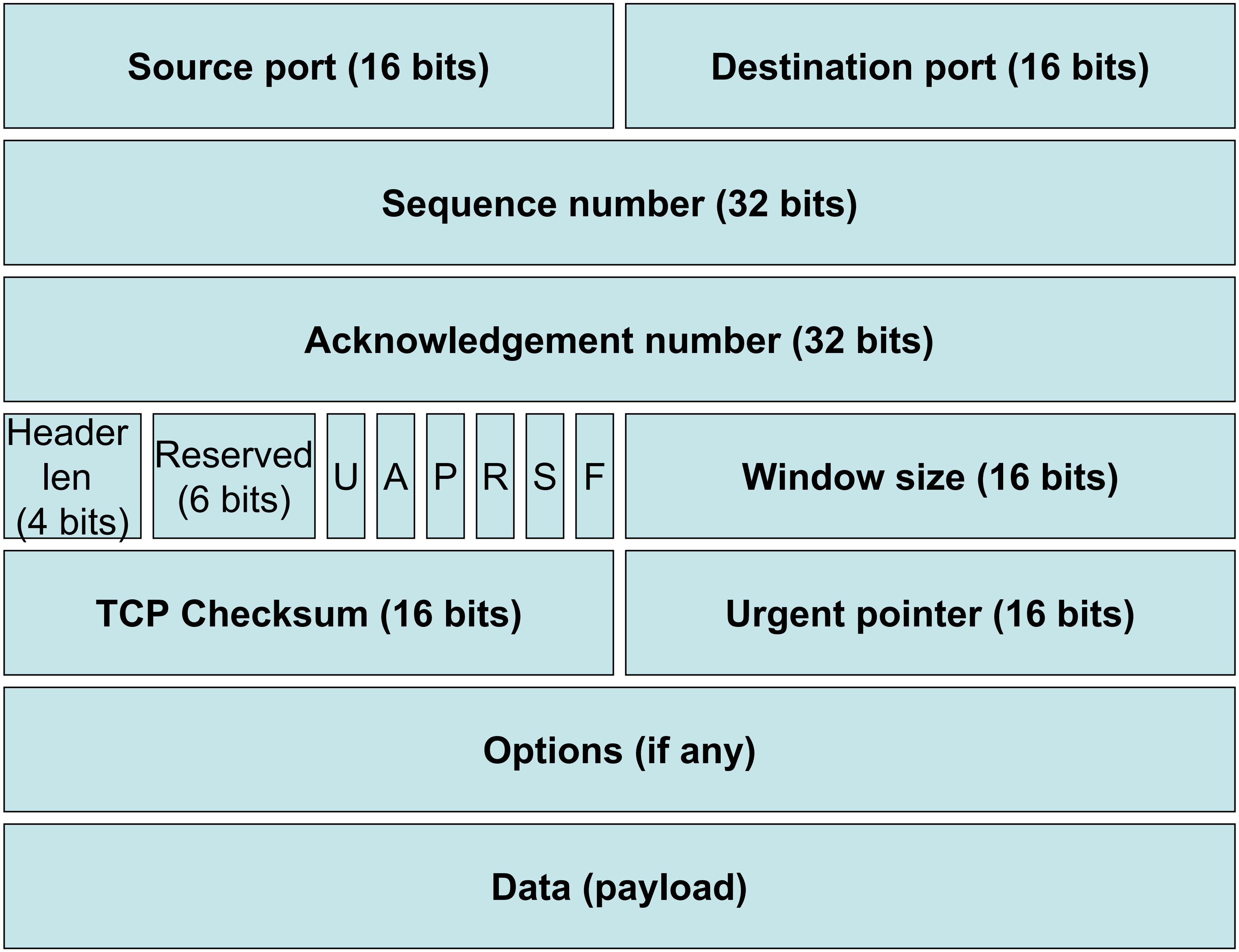
Destination port (16 bits)

UDP length (16 bits)

UDP checksum (16 bits)

Data (payload)

TCP Header Structure



Source port (16 bits) | Destination port (16 bits)

Sequence number (32 bits)

Acknowledgement number (32 bits)

Header len (4 bits) | Reserved (6 bits) | U | A | P | R | S | F | Window size (16 bits)

TCP Checksum (16 bits) | Urgent pointer (16 bits)

Options (if any)

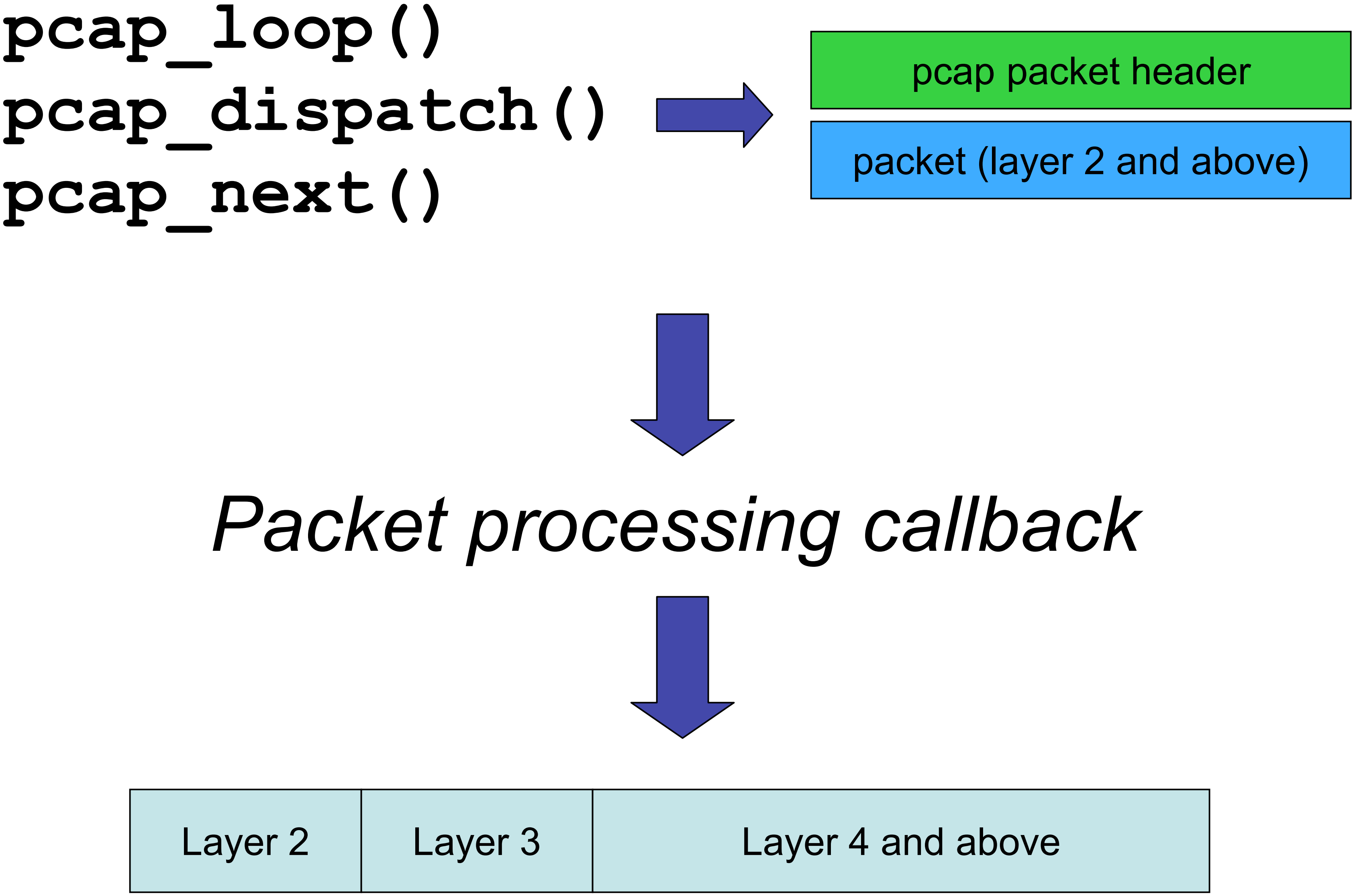
Data (payload)

libpcap – packet capture

Platform independent
Flexible
Relatively decent performance
Very standard

Order of operations

1. Create a pcap object: `pcap_open_live()`
2. Get data from the network, send to callback
3. Close pcap object: `pcap_close()`



```
pcap_loop()
pcap_dispatch()
pcap_next()
```

pcap packet header

packet (layer 2 and above)

Packet processing callback

Layer 2	Layer 3	Layer 4 and above

libdnet – low level networking

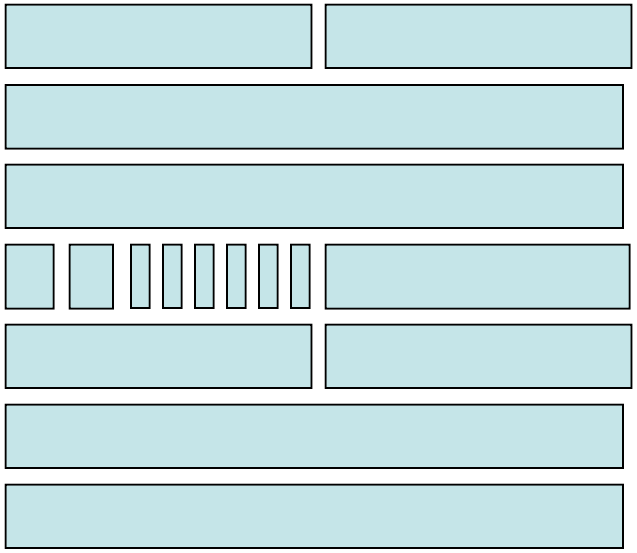
Simple interface to network, kernel material
Cross platform (Win, OS X, UN*X)
Easy to use interface

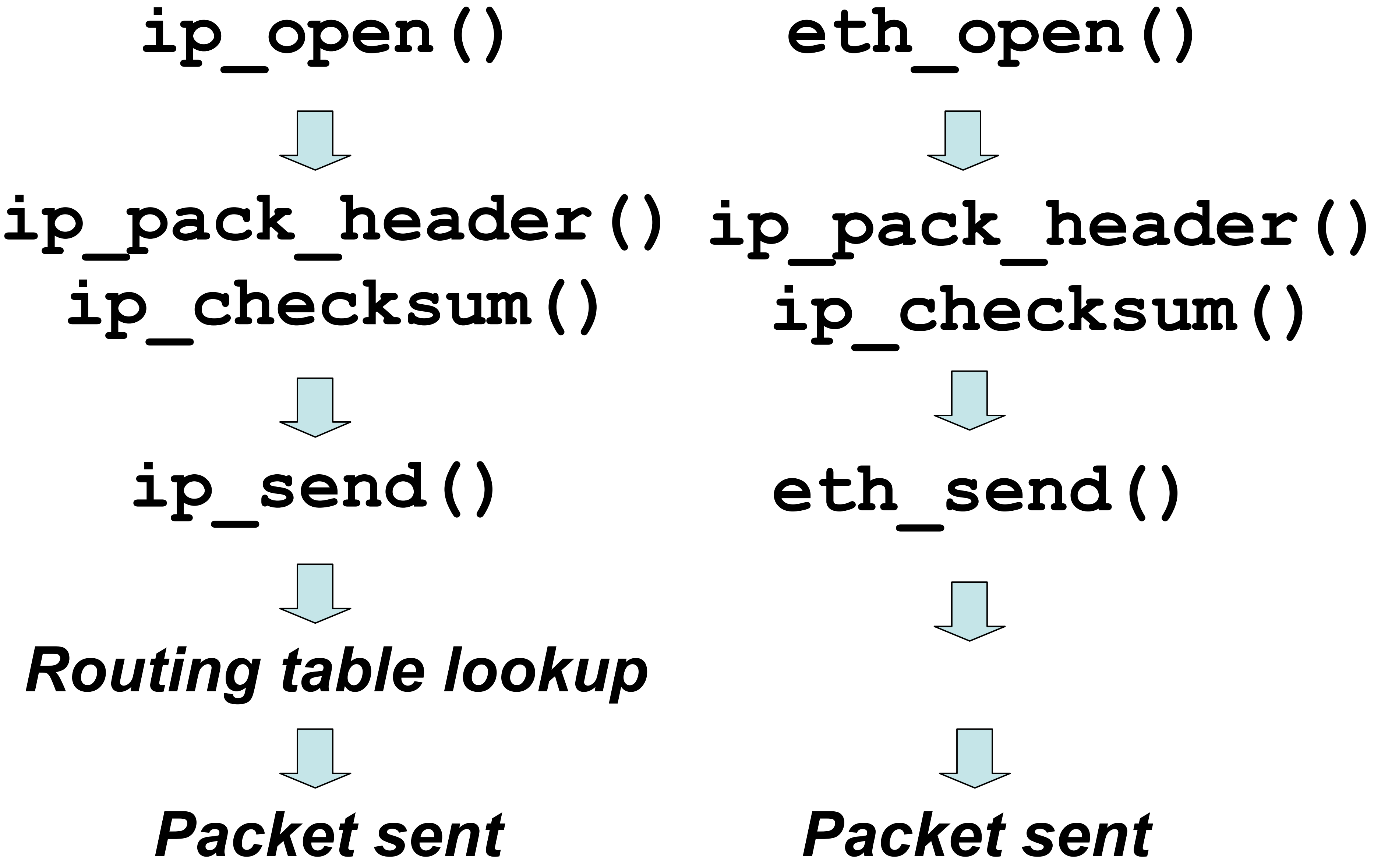
Libdnet basics:

1. Open network object:	`ip_open()`
2. Allocate packet memory	
3. Construct TCP packet:	`tcp_pack_header()`
4. Construct IP packet:	`ip_pack_header()`
5. Checksum:	`ip_checksum()`
6. Write the packet:	`ip_send()`
7. Close the object:	`ip_close()`

IP	IPv6
Addressing	Arp
Routing	Firewalling
Ethernet	Interfaces
Blobs	Random Numbers

```
tcp_pack_hdr(hdr,
             sport,
             dport,
             seq,
             ack,
             flags,
             win,
             urp)
```





`ip_open()`

`ip_pack_header()`
`ip_checksum()`

`ip_send()`

Routing table lookup

Packet sent

`eth_open()`

`ip_pack_header()`
`ip_checksum()`

`eth_send()`

Packet sent

pcap, event, dnet example: **jscan**

- TCP SYN port scanner
- OS fingerprinting
- Passive fingerprinting
- Passive port scanning
- Active port scanning
- Can be decoupled

http://monkey.org/~jose/software/jscan/

jscan Program flow

- Open pcap object (for receiving)
- Open IP object (for sending)
- Create and set send and receive events
- Send callback
- Receive callback
  - Fingerprint OS using the packet
  - Report results
- Loop until all ports scanned or stopped

```
<includes>
<report results>
<send callback>
<receive callback>

int main (int argc, char *argv[])
{
      <getopt setup>
      ctx.rand = rand_open();
      ctx.p = pcap_open_live(intf, 1500,
            (ctx.flags == SCAN_FLAGS_PASSIVE), 500,
             ebuff);
      if (ctx.p == NULL)
            err(1, "pcap_open_live");
      ctx.dl_len = pcap_dloff(ctx.p);
      <event setup>
      printf("scan completed in %d seconds.\n",
            time(NULL) - start);
      return (1);
}
```

event setup:

```
event_init();
ctx.tv.tv_sec = 0;
ctx.tv.tv_usec = 500;
p_fd = pcap_fileno(ctx.p);

event_set(&ctx.recv_ev, p_fd, EV_READ,
         _recv, (void *) &ctx);
event_add(&ctx.recv_ev, &ctx.tv);
if (ctx.flags == SCAN_FLAGS_ACTIVE) {
        ctx.ip = ip_open();
        if (ctx.ip == NULL)
                err(1, "ip_open() failed ..");
        event_set(&ctx.send_ev, p_fd, EV_WRITE,
                 _send, (void *) &ctx);
        event_add(&ctx.send_ev, &ctx.tv);
        ctx.dport = 1;
}
event_dispatch();
```

receive callback:

```
static void _recv(int fd, short event, void *arg)
{
        struct myctx *ctx = (struct myctx *) arg;
        struct pcap_pkthdr ph;
        u_char *pread;

        if ((ctx->flags == SCAN_FLAGS_ACTIVE)
                && (ctx->dport > 65535));
        else
                /* reschedule */
                event_add(&ctx->recv_ev, &ctx->tv);
        if ((pread = (u_char *) pcap_next(ctx->p, &ph)) != NULL)
                report(pread, ctx);
        return;
}
```

send callback:

```
static void _send(int fd, short event, void *arg)
{
        struct myctx *ctx = (struct myctx *) arg;
        struct jscan_pkt *pkt;
        u_char buf[BUFSIZ];
        int len = IP_HDR_LEN + TCP_HDR_LEN, dport;
        if (ctx->dport > 65535)
                return;
        event_add(&ctx->send_ev, &ctx->tv);
        pkt = (struct jscan_pkt *) buf;
        ip_pack_hdr(&pkt->pkt_hdr_i.ip, IP_TOS_LOWDELAY,
                len, rand_uint16(ctx->rand), 0, 128, IP_PROTO_TCP,
                ctx->src.addr_ip, ctx->dst.addr_ip);
        tcp_pack_hdr(&pkt->pkt_hdr_t.tcp, rand_uint16(ctx->rand),
                ctx->dport, rand_uint32(ctx->rand),
                rand_uint32(ctx->rand), TH_SYN,
                rand_uint16(ctx->rand), 0);
        ip_checksum(pkt, len);
        ip_send(ctx->ip, pkt, len);
        ctx->dport += 1;           /* we SYNed that port */
        return;
}
```

report callback (2 pages):

```
static void report(u_char * packet, void *arg)
{
        struct myctx *ctx = (struct myctx *) arg;
        static struct ip_hdr *ip_h;
        u_char *tmp;
        const char *p;
        struct addr ip_src;
        static struct entry *np, *n2;
        tmp = packet + ctx->dl_len;
        ip_h = (struct ip_hdr *) tmp;
        if (ip_h->ip_v != 4)
                return;
        p = inet_ntoa(ip_h->ip_src);
        if ((addr_aton(p, &ip_src)) == -1)
                return;
        /*
         * if it's a passive scan, don't care about the src
         * address. if it's an active scan, make sure it was the
         * dest we specified. make sure it's a TCP packet, too,
         * and has SA set.
         */
```

```
      if (((ctx->flags == SCAN_FLAGS_PASSIVE) ||
          ((addr_cmp(&ip_src, &(ctx->dst))) == 0)) &&
            (ip_h->ip_p == IP_PROTO_TCP)) {
             struct tcp_hdr *tcp_h =
                    (struct tcp_hdr *)(tmp + IP_HDR_LEN);
             if (tcp_h->th_flags == 0x12) {    /* SYN ACK */
                    struct servent *serv;
                    char *s_name = "unknown", *os = NULL;
                    if (ctx->osfile != NULL)
                          os = osprint(ctx, ntohs(tcp_h->th_win),
                                 ip_h->ip_ttl, ip_h->ip_off,
                                 ntohs(ip_h->ip_len));
                    serv = getservbyport(tcp_h->th_sport, "tcp");
                    if (serv != NULL)
                          s_name = strdup(serv->s_name);
                    printf("%-16s %35s %15s %6d/tcp\n",
                          addr_ntoa(&ip_src),       os ?
                          os : “unknown", s_name,
                          htons(tcp_h->th_sport));
             }
      }
      return;
}
```

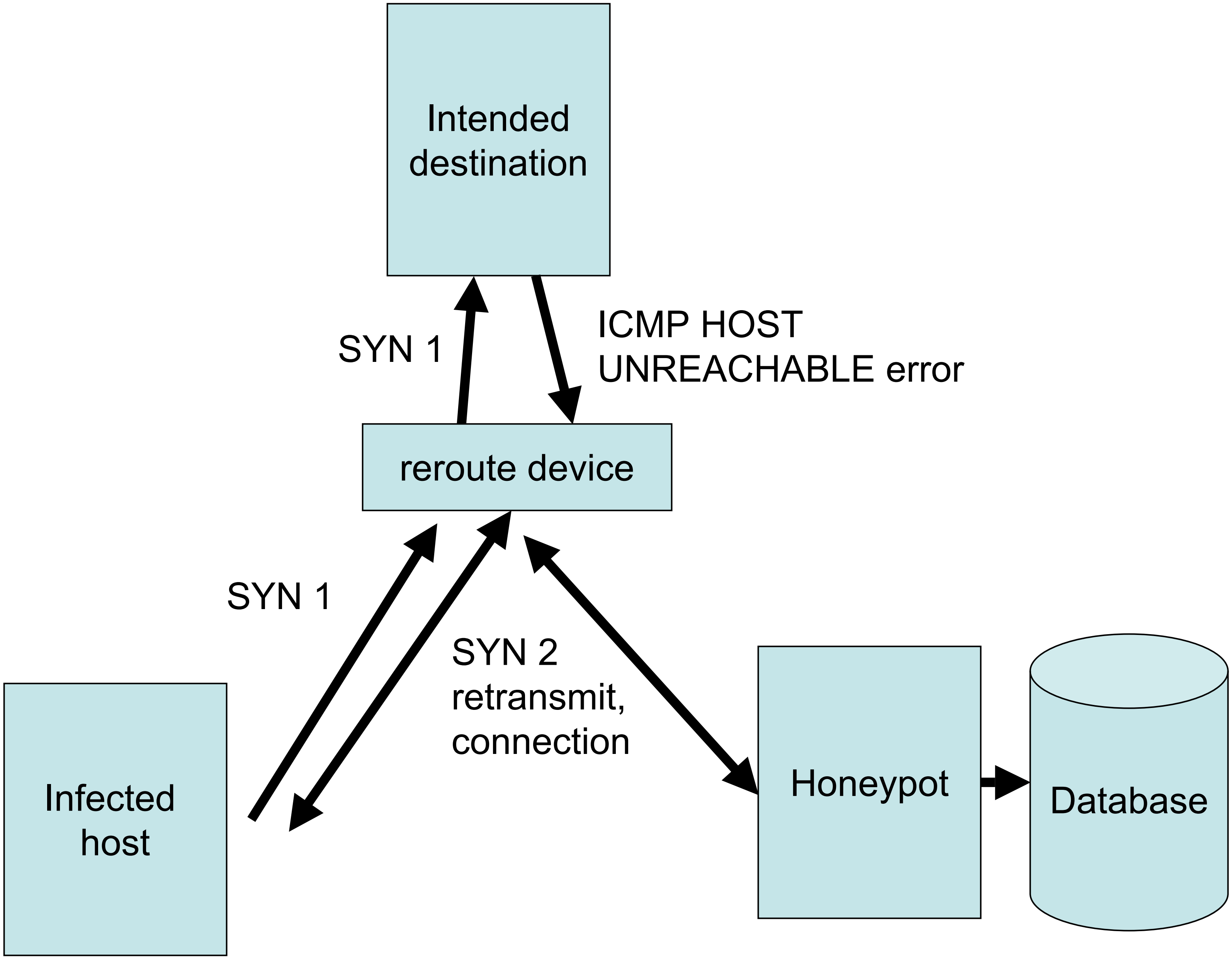
jscan output

```
$ sudo jscan -t passive -i fxp0 -f compat/pf.os
scan started, type is passive, listening on fxp0
192.48.159.40            unknown             www     80/tcp
216.136.204.117  FreeBSD 4.6-4.8             www     80/tcp


$ sudo jscan -t active -s 192.168.3.4 -d 192.168.1.4
   -i fxp0 -f compat/pf.os
scan started, type is active, listening on fxp0
192.168.1.4         Linux 2.0.3x             ssh     22/tcp
192.168.1.4         Linux 2.0.3x           whois     43/tcp
192.168.1.4         Linux 2.0.3x            auth    113/tcp
192.168.1.4         Linux 2.0.3x             bgp    179/tcp
scan completed. total execution time was 70 seconds.
```

dnet, pcap example: **reroute**

- Small daemon for honeypot usage
- Adds a firewall rule to block ICMP errors from getting to source
- Listens for ICMP HOST UNREACHABLE errors
- ICMP error? Inject a local route to redirect destination host to the honeypot
- Lets you honeypot for arbitrary hosts
- Suggested by IBM Billy Goat project

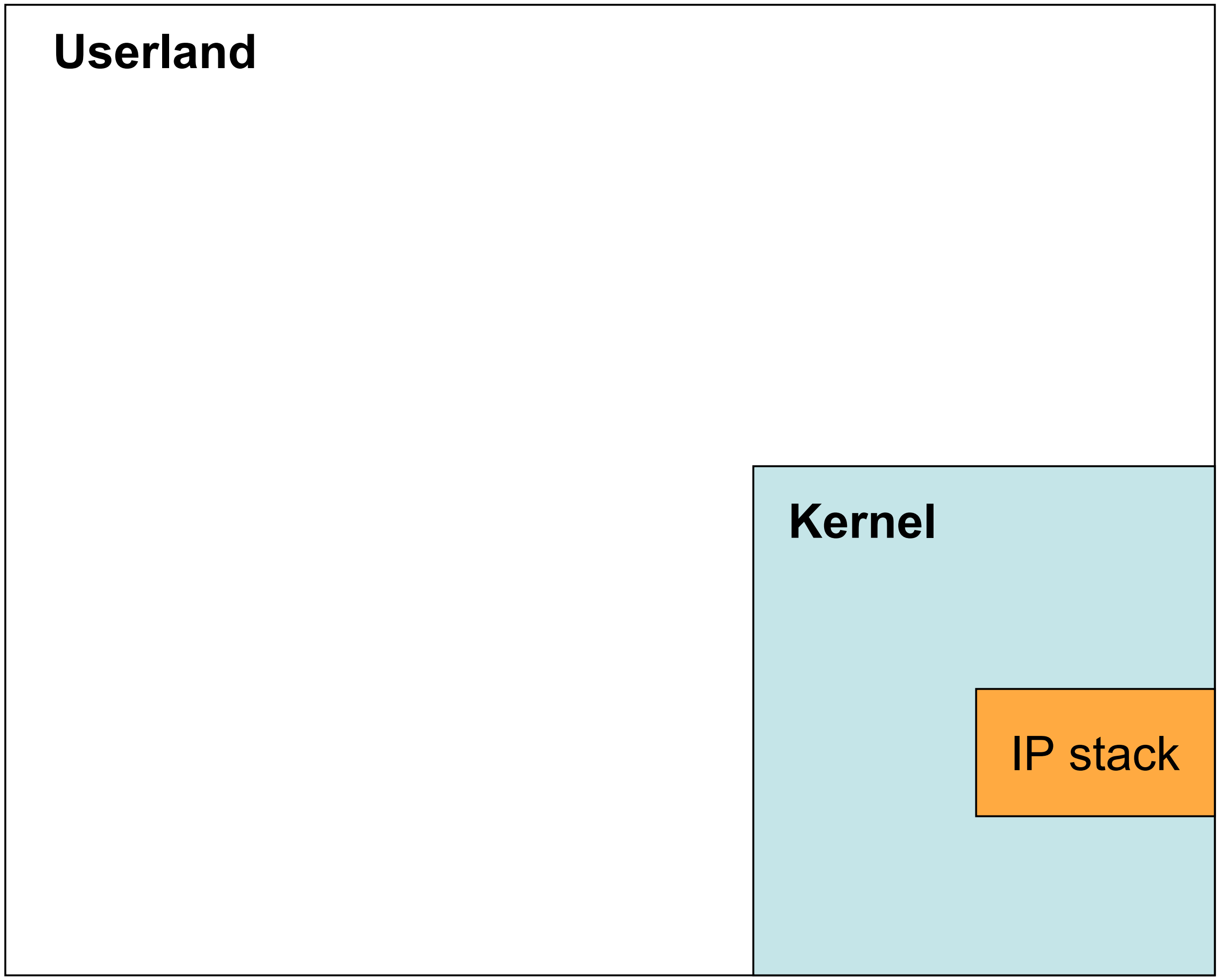

Intended destination
SYN 1
ICMP HOST UNREACHABLE error
reroute device
SYN 1
SYN 2 retransmit, connection
Infected host
Honeypot
Database

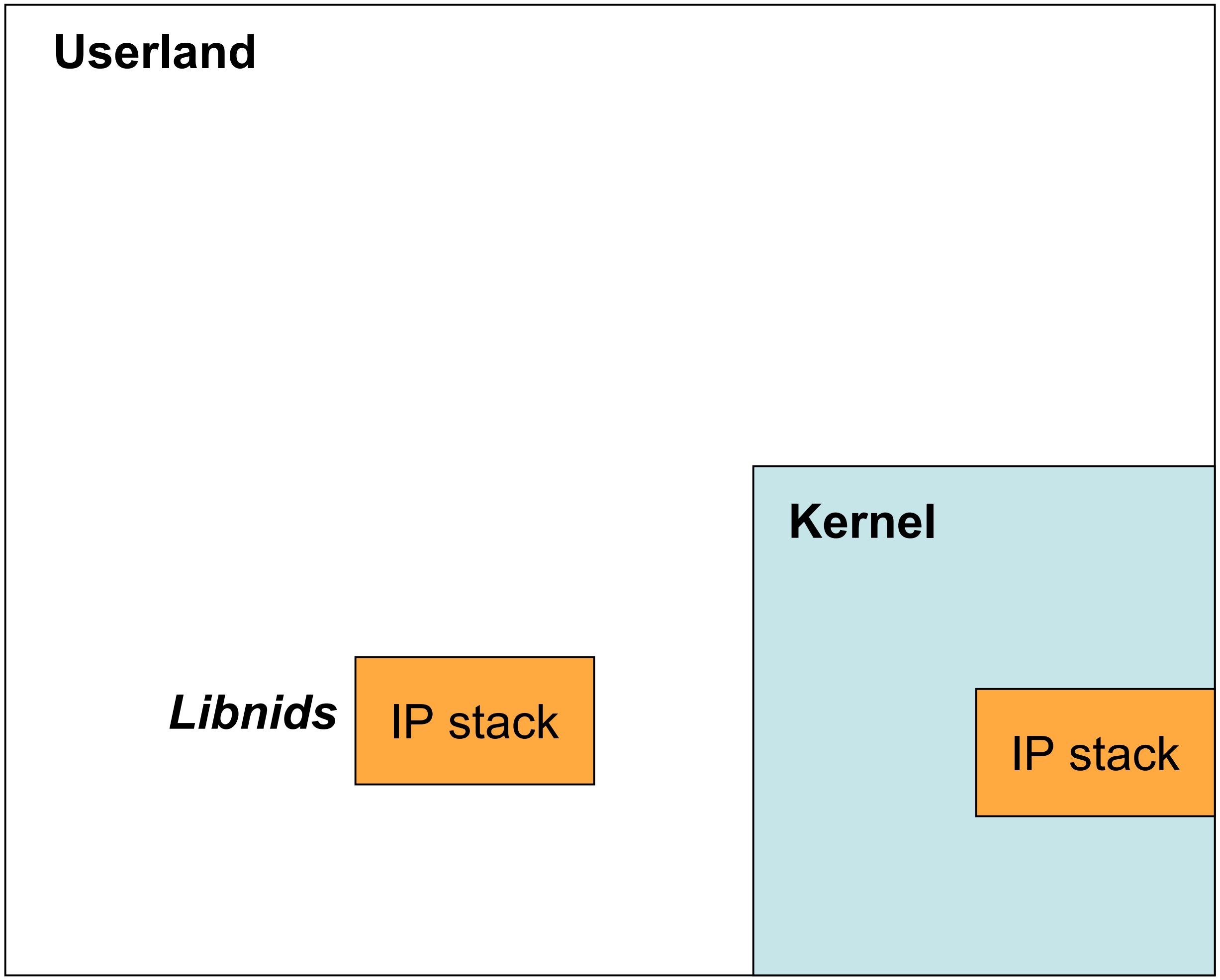
reroute program flow

- Initialize
  - pcap, open route handle, open firewall handle
  - Inject firewall rules (block ICMP host errors)
- Watch for ICMP host errors
  - Record intended source
  - Inject route to honeypot
- At close, withdraw routes, firewall rule

libnids – reassemble IP streams

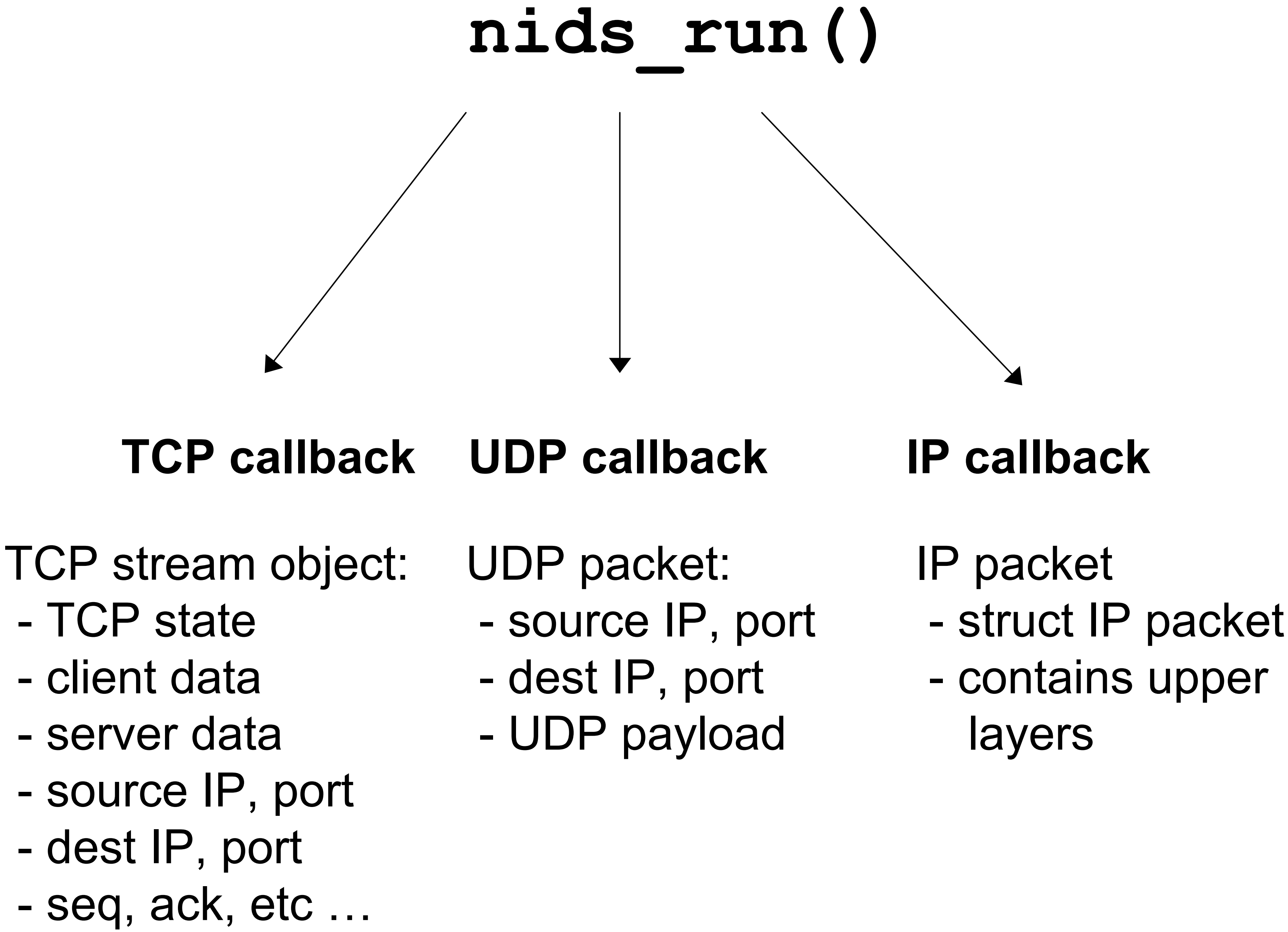
NIDS “E” box (event generation box)
Userland TCP/IP stack
Based on Linux 2.0.36 IP stack
Uses libpcap, libnet internally
IP fragment reassembly
IP stream construction


Userland
Kernel
IP stack


Userland
Kernel
Libnids
IP stack
IP stack

libnids Basics

- Initialize
  - **`nids_init()`**
- Register callbacks
  - **`nids_register_tcp()`**
  - **`nids_register_ip()`**
  - **`nids_register_udp()`**
- Run!
  - **`nids_run()`**
- React
  - **`nids_kill_tcp()`**



`nids_run()`

TCP callback

TCP stream object:
- TCP state
- client data
- server data
- source IP, port
- dest IP, port
- seq, ack, etc …

UDP callback

UDP packet:
- source IP, port
- dest IP, port
- UDP payload

IP callback

IP packet
- struct IP packet
- contains upper layers

libnids TCP states

- **`NIDS_JUST_ESTABLISHED`**
  - New TCP connected state (3WHS)
  - Must set **`stream-> {client,server}.collect=1`** to get stream payload collected
- **`NIDS_DATA`**
  - Data within a known, established TCP connection
- **`NIDS_RESET, NIDS_CLOSE, NIDS_TIMED_OUT`**
  - TCP connection is reset, closed gracefully, or was lost

libnids doesn’t expose SYN_SENT, FIN_WAIT, etc …

Example libnids code: **jflow**

- jflow
  - Pcap to NetFlow v1 summaries
  - Daemonizes
  - Sends to a receiving host over UDP
- Limitations of jflow
  - Not very lightweight
  - Inaccurate for some things

http://monkey.org/~jose/software/jflow/

```
<includes>
<export record>
<ip callback>

int main (int argc, char *argv[])
{
     <getopt handler>
     <UDP socket connect>
     nids_init();
     nids_register_ip(monitor_ip);
     nids_run();
     return(0);
}
```

ip callback (*truncated*):

```
void monitor_ip(struct ip *pkt)
{
        struct ip_record rec;
        int i;

        rec.rec.srcaddr = (u_int)(pkt->ip_src.s_addr);
        rec.rec.dstaddr = (u_int)(pkt->ip_dst.s_addr);
        rec.rec.nexthop = inet_addr("0.0.0.0");
        rec.rec.dOctets = htonl(pkt->ip_len);
        rec.rec.pad = 0x0;
        rec.rec.prot = pkt->ip_p;
        rec.rec.tos = 0x0;
        rec.rec.tcp_flags = 0x0;
        rec.rec.pad_2 = 0x0;
        rec.rec.pad_3 = 0x0;
        for (i = 0; i < 4; i++)
                rec.rec.reserved[i] = 0x0;
        export_ip_record(&rec);
        return;
}
```

export record:

```
void export_ip_record(struct ip_record *rec)
{
        time_t now;
        /* fill out the header */
        now = time(NULL);
        rec->hdr.version = htons(1);
        rec->hdr.count = htons(1);
        rec->hdr.SysUptime = htonl(get_uptime());
        rec->hdr.unix_secs = htonl(now);
        rec->hdr.unix_nsecs = 0;   /* XXX */
        if (write(ctx.u, rec, sizeof(struct ip_record))
            < sizeof(struct ip_record))
                warn("ip_export_record(): short write()");
        else ctx.count += 1;
        return;
}
```

jflow output

```
$ sudo tcpdump -lni fxp0 -s1500 -Tcnfp udp port 5000
11:21:50.256833 NetFlow v1, 611.550 uptime, 1095175310.0,  2 recs
  started 7209.020, last 536870.912
    65.205.8.103:80 > 192.168.1.190:37116 >> 0.0.0.0
    6 tos 0, 623 (623 octets)
  started 1103956.071, last 167772.606
    192.168.1.190:37116 > 65.205.8.103:80 >> 0.0.0.0
    6 tos 0, 4851 (4851 octets)
...
11:21:58.578965 NetFlow v1, 626.438 uptime, 1095175810.0,  1 recs
  started 1893728.316, last 2220884.028
    192.168.1.160:137 > 192.168.1.255:137 >> 0.0.0.0
    17 tos 0, 1 (50 octets) (ttl 64, id 8693)
```

pynids example: flowgrep

- Marries sniffing with regular expressions
- A lot like ngrep, tcpkill, and dsniff
  - Logs the whole connection, not just a packet
- Look for data in streams using regular expressions
- Log or kill selected streams
- Dirt cheap IDS or IPS
  - Under 400 lines of code

```
#! /usr/bin/env python
import getopt, os, pwd, re, string,
struct, sys, time
import nids

def main():
 global flags, crelist, srelist, logdir
 usagestr = """%s: TCP/IP payload 'grep'
             utility””” %sys.argv[0]
 <getopt stuff>
 <RE setup>
 <nids setup>
 while 1:
  try: nids.run() # loop forever
  except KeyboardInterrupt:
   break
   sys.exit(1)
```

nids setup

```
if len(args) > 0:
  nids.param("pcap_filter",\
     string.join(args))
nids.param("scan_num_hosts", 0)
try: nids.init()
except nids.error, e:
  print "initialization error -", e
  sys.exit(1)
<drop privs>
nids.register_tcp(handleTcp)
nids.register_udp(handleUdp)
nids.register_ip(handleIp)
```

handleUdp

```
def handleUdp(addr, payload, pkt):
 # format of addr: ((src, sport), (dst, dport))
 match = 0
 for clientre in crelist:
  if clientre.search(payload): match = 1
 for serverre in srelist:
  if serverre.search(payload): match = 1
 if flags['l']:
  if match and not flags['v']:
   logPkt(addr, payload)
  if not match and flags['v']:
   logPkt(addr, payload)
```

Common Mistakes

- Bytesex: ntohl(), ntohs(), htonl(), htons()
- Hardcoded values (header lengths)
- Don’t reschedule an event
- Pointer problems
- Not setting stream.*.collect = 1

Performance considerations

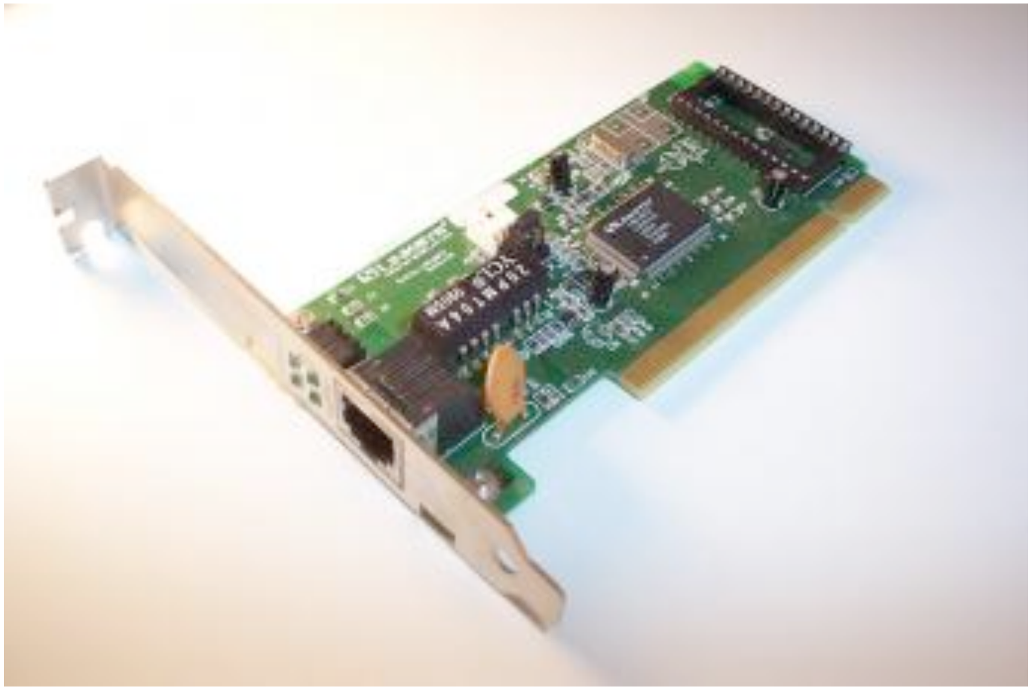
List

1 …
2….
3….
4….
…

O(n)

Loop:

1. **open file**
2. **malloc()**
3. **parse file**



Interrupts

Hardware checksumming

Memory management

Resources

- http://libdnet.sourceforge.net
- http://www.tcpdump.org
- http://www.packetfactory.net/projects/libnids
- http://libevent.sourceforge.net
- http://monkey.org/~jose/software/{jscan,jflow,jtrace,flowgrep,reroute}

Related

- dscan, dsniff, dpkt
- scapy
- libnet
- pads
- passivefist
- ngrep, tcpstream
- p0f, nemesis, CASL

Additional Resources

- Stevens, TCP/IP Illustrated vols 1 and 2
- Schiffman, Building Open Source Network Security Tools
- RFCs from the IETF