// Send an IPv4 TCP packet via raw socket at the link layer (ethernet frame).

// Need to have destination MAC address.

// sending data, no TCP options data.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h> // close()

#include <string.h> // strcpy, memset(), and memcpy()

#include <netdb.h> // struct addrinfo

#include <sys/types.h> // needed for socket(), uint8\_t, uint16\_t, uint32\_t

#include <sys/socket.h> // needed for socket()

#include <netinet/in.h> // IPPROTO\_TCP, INET\_ADDRSTRLEN

#include <netinet/ip.h> // struct ip and IP\_MAXPACKET (which is 65535)

#define \_\_FAVOR\_BSD // Use BSD format of tcp header

#include <netinet/tcp.h> // struct tcphdr

#include <arpa/inet.h> // inet\_pton() and inet\_ntop()

#include <sys/ioctl.h> // macro ioctl is defined

#include <bits/ioctls.h> // defines values for argument "request" of ioctl.

#include <net/if.h> // struct ifreq

#include <linux/if\_ether.h> // ETH\_P\_IP = 0x0800, ETH\_P\_IPV6 = 0x86DD

#include <linux/if\_packet.h> // struct sockaddr\_ll (see man 7 packet)

#include <net/ethernet.h>

#include <errno.h> // errno, perror()

// Define some constants.

#define ETH\_HDRLEN 14 // Ethernet header length

#define IP4\_HDRLEN 20 // IPv4 header length

#define TCP\_HDRLEN 20 // TCP header length, excludes options data

// Function prototypes

uint16\_t checksum (uint16\_t \*, int);

uint16\_t tcp4\_checksum (struct ip, struct tcphdr);

char \*allocate\_strmem (int);

uint8\_t \*allocate\_ustrmem (int);

int \*allocate\_intmem (int);

char data[]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";

int

main (int argc, char \*\*argv)

{

int i, status, frame\_length, sd, bytes, \*ip\_flags, \*tcp\_flags;

char \*interface, \*target, \*src\_ip, \*dst\_ip;

struct ip iphdr;

struct tcphdr tcphdr;

uint8\_t \*src\_mac, \*dst\_mac, \*ether\_frame;

struct addrinfo hints, \*res;

**struct sockaddr\_in \*ipv4;**

**struct sockaddr\_ll device;**

**struct ifreq ifr;**

void \*tmp;

// Allocate memory for various arrays.

**src\_mac = allocate\_ustrmem (6);**

**dst\_mac = allocate\_ustrmem (6);**

**ether\_frame = allocate\_ustrmem (IP\_MAXPACKET);**

**interface = allocate\_strmem (40);**

**target = allocate\_strmem (40);**

**src\_ip = allocate\_strmem (INET\_ADDRSTRLEN);**

**dst\_ip = allocate\_strmem (INET\_ADDRSTRLEN);**

**ip\_flags = allocate\_intmem (4);**

**tcp\_flags = allocate\_intmem (8);**

// Interface to send packet through.

strcpy (interface, "eth0");

// Submit request for a socket descriptor to look up interface.

if ((**sd = socket (PF\_PACKET, SOCK\_RAW, htons (ETH\_P\_ALL))) < 0)** {

perror ("socket() failed to get socket descriptor for using ioctl() ");

exit (EXIT\_FAILURE);

}

// Use ioctl() to look up interface name and get its MAC address.

**memset (&ifr, 0, sizeof (ifr));**

**snprintf (ifr.ifr\_name, sizeof (ifr.ifr\_name), "%s", interface);**

if (ioctl (sd, SIOCGIFHWADDR, &ifr) < 0) {

perror ("ioctl() failed to get source MAC address ");

return (EXIT\_FAILURE);

}

close (sd);

// Copy source MAC address.

**memcpy (src\_mac, ifr.ifr\_hwaddr.sa\_data, 6 \* sizeof (uint8\_t));**

// Report source MAC address to stdout.

printf ("MAC address for interface %s is ", interface);

for (i=0; i<5; i++) {

printf ("%02x:", src\_mac[i]);

}

printf ("%02x\n", src\_mac[5]);

// Find interface index from interface name and store index in

// struct sockaddr\_ll device, which will be used as an argument of sendto().

**memset (&device, 0, sizeof (device));**

**if ((device.sll\_ifindex = if\_nametoindex (interface)) == 0) {**

perror ("if\_nametoindex() failed to obtain interface index ");

exit (EXIT\_FAILURE);

}

printf ("Index for interface %s is %i\n", interface, device.sll\_ifindex);

**// Set destination MAC address: you need to fill these out**

**dst\_mac[0] = 0xff;**

**dst\_mac[1] = 0xff;**

**dst\_mac[2] = 0xff;**

**dst\_mac[3] = 0xff;**

**dst\_mac[4] = 0xff;**

**dst\_mac[5] = 0xff;**

// Source IPv4 address: you need to fill this out

// strcpy (src\_ip, "192.168.1.132");

**strcpy (src\_ip, "127.0.0.1");**

// Destination URL or IPv4 address: you need to fill this out

//strcpy (target, "www.google.com");

**strcpy (target, "localhost");**

// Fill out hints for getaddrinfo().

**memset (&hints, 0, sizeof (struct addrinfo));**

**hints.ai\_family = AF\_INET;**

**hints.ai\_socktype = SOCK\_STREAM;**

**hints.ai\_flags = hints.ai\_flags | AI\_CANONNAME;**

// Resolve target using getaddrinfo().

**if ((status = getaddrinfo (target, NULL, &hints, &res)) != 0) {**

fprintf (stderr, "getaddrinfo() failed: %s\n", gai\_strerror (status));

exit (EXIT\_FAILURE);

}

**ipv4 = (struct sockaddr\_in \*) res->ai\_addr;**

**tmp = &(ipv4->sin\_addr);**

**if (inet\_ntop (AF\_INET, tmp, dst\_ip, INET\_ADDRSTRLEN) == NULL) {**

status = errno;

fprintf (stderr, "inet\_ntop() failed.\nError message: %s", strerror (status));

exit (EXIT\_FAILURE);

}

freeaddrinfo (res);

**// Fill out sockaddr\_ll.**

**device.sll\_family = AF\_PACKET;**

**memcpy (device.sll\_addr, src\_mac, 6 \* sizeof (uint8\_t));**

**device.sll\_halen = 6;**

// IPv4 header -----------------------------------------------------

// IPv4 header length (4 bits): Number of 32-bit words in header = 5

**iphdr.ip\_hl = IP4\_HDRLEN / sizeof (uint32\_t);**

// Internet Protocol version (4 bits): IPv4

**iphdr.ip\_v = 4;**

// Type of service (8 bits)

**iphdr.ip\_tos = 0;**

// Total length of datagram (16 bits): IP header + TCP header

**iphdr.ip\_len = htons (IP4\_HDRLEN + TCP\_HDRLEN);**

// ID sequence number (16 bits): unused, since single datagram

**iphdr.ip\_id = htons (0);**

// Flags, and Fragmentation offset (3, 13 bits): 0 since single datagram

// Zero (1 bit)

**ip\_flags[0] = 0;**

// Do not fragment flag (1 bit)

**ip\_flags[1] = 0;**

// More fragments following flag (1 bit)

**ip\_flags[2] = 0;**

// Fragmentation offset (13 bits)

**ip\_flags[3] = 0;**

**iphdr.ip\_off = htons ((ip\_flags[0] << 15)**

**+ (ip\_flags[1] << 14)**

**+ (ip\_flags[2] << 13)**

**+ ip\_flags[3]);**

// Time-to-Live (8 bits): default to maximum value

**iphdr.ip\_ttl = 255;**

// Transport layer protocol (8 bits): 6 for TCP

**iphdr.ip\_p = IPPROTO\_TCP;**

// Source IPv4 address (32 bits)

**if ((status = inet\_pton (AF\_INET, src\_ip, &(iphdr.ip\_src))) != 1) {**

fprintf (stderr, "inet\_pton() failed.\nError message: %s", strerror (status));

exit (EXIT\_FAILURE);

}

// Destination IPv4 address (32 bits)

**if ((status = inet\_pton (AF\_INET, dst\_ip, &(iphdr.ip\_dst))) != 1) {**

fprintf (stderr, "inet\_pton() failed.\nError message: %s", strerror (status));

exit (EXIT\_FAILURE);

}

// IPv4 header checksum (16 bits): set to 0 when calculating checksum

**iphdr.ip\_sum = 0;**

**iphdr.ip\_sum = checksum ((uint16\_t \*) &iphdr, IP4\_HDRLEN);**

// TCP header ------------------------------------------------------

// Source port number (16 bits)

**tcphdr.th\_sport = htons (60);**

// Destination port number (16 bits)

**tcphdr.th\_dport = htons (80);**

// Sequence number (32 bits)

**tcphdr.th\_seq = htonl (0);**

// Acknowledgement number (32 bits): 0 in first packet of SYN/ACK process

**tcphdr.th\_ack = htonl (0);**

// Reserved (4 bits): should be 0

**tcphdr.th\_x2 = 0;**

// Data offset (4 bits): size of TCP header in 32-bit words

**tcphdr.th\_off = TCP\_HDRLEN / 4;**

// Flags (8 bits)

**// FIN flag (1 bit)**

**tcp\_flags[0] = 0;**

**// SYN flag (1 bit): set to 1**

**tcp\_flags[1] = 0;**

**// RST flag (1 bit)**

**tcp\_flags[2] = 0;**

**// PSH flag (1 bit)**

**tcp\_flags[3] = 0;**

**// ACK flag (1 bit)**

**tcp\_flags[4] = 0;**

**// URG flag (1 bit)**

**tcp\_flags[5] = 0;**

**// ECE flag (1 bit)**

**tcp\_flags[6] = 0;**

**// CWR flag (1 bit)**

**tcp\_flags[7] = 0;**

**tcphdr.th\_flags = 0;**

for (i=0; i<8; i++) {

tcphdr.th\_flags += (tcp\_flags[i] << i);

}

// Window size (16 bits)

**tcphdr.th\_win = htons (65535);**

// Urgent pointer (16 bits): 0 (only valid if URG flag is set)

**tcphdr.th\_urp = htons (0);**

// TCP checksum (16 bits)

**tcphdr.th\_sum = tcp4\_checksum (iphdr, tcphdr);**

//strcpy(data,"ABCDEFGHIJKLMNOPQRSTUVWXYZ");

// Fill out ethernet frame header. ---------------------------------

// Ethernet frame length = ethernet header (MAC + MAC + ethernet type) + ethernet data (IP header + TCP header)

**frame\_length = 6 + 6 + 2 + IP4\_HDRLEN + TCP\_HDRLEN;**

// Destination and Source MAC addresses

**memcpy (ether\_frame, dst\_mac, 6 \* sizeof (uint8\_t));**

**memcpy (ether\_frame + 6, src\_mac, 6 \* sizeof (uint8\_t));**

// Next is ethernet type code (ETH\_P\_IP for IPv4).

// http://www.iana.org/assignments/ethernet-numbers

**ether\_frame[12] = ETH\_P\_IP / 256;**

**ether\_frame[13] = ETH\_P\_IP % 256;**

// Next is ethernet frame data (IPv4 header + TCP header).

// IPv4 header

**memcpy (ether\_frame + ETH\_HDRLEN, &iphdr, IP4\_HDRLEN \* sizeof (uint8\_t));**

// TCP header

**memcpy (ether\_frame + ETH\_HDRLEN + IP4\_HDRLEN, &tcphdr, TCP\_HDRLEN \* sizeof (uint8\_t));**

// Submit request for a raw socket descriptor.

if ((**sd = socket (PF\_PACKET, SOCK\_RAW, htons (ETH\_P\_ALL))) < 0)** {

perror ("socket() failed ");

exit (EXIT\_FAILURE);

}

// Send ethernet frame to socket.

if ((bytes = sendto (sd, ether\_frame, frame\_length, 0, (struct sockaddr \*) &device, sizeof (device))) <= 0) {

perror ("sendto() failed");

exit (EXIT\_FAILURE);

}

// Close socket descriptor.

close (sd);

// Free allocated memory.

free (src\_mac);

free (dst\_mac);

free (ether\_frame);

free (interface);

free (target);

free (src\_ip);

free (dst\_ip);

free (ip\_flags);

free (tcp\_flags);

return (EXIT\_SUCCESS);

}

// Computing the internet checksum (RFC 1071).

// Note that the internet checksum does not preclude collisions.

uint16\_t

checksum (uint16\_t \*addr, int len)

{

int count = len;

register uint32\_t sum = 0;

uint16\_t answer = 0;

// Sum up 2-byte values until none or only one byte left.

while (count > 1) {

sum += \*(addr++);

count -= 2;

}

// Add left-over byte, if any.

if (count > 0) {

sum += \*(uint8\_t \*) addr;

}

// Fold 32-bit sum into 16 bits; we lose information by doing this,

// increasing the chances of a collision.

// sum = (lower 16 bits) + (upper 16 bits shifted right 16 bits)

while (sum >> 16) {

sum = (sum & 0xffff) + (sum >> 16);

}

// Checksum is one's compliment of sum.

answer = ~sum;

return (answer);

}

// Build IPv4 TCP pseudo-header and call checksum function.

uint16\_t

tcp4\_checksum (struct ip iphdr, struct tcphdr tcphdr)

{

uint16\_t svalue;

char buf[IP\_MAXPACKET], cvalue;

char \*ptr;

int chksumlen = 0;

ptr = &buf[0]; // ptr points to beginning of buffer buf

// Copy source IP address into buf (32 bits)

memcpy (ptr, &iphdr.ip\_src.s\_addr, sizeof (iphdr.ip\_src.s\_addr));

ptr += sizeof (iphdr.ip\_src.s\_addr);

chksumlen += sizeof (iphdr.ip\_src.s\_addr);

// Copy destination IP address into buf (32 bits)

memcpy (ptr, &iphdr.ip\_dst.s\_addr, sizeof (iphdr.ip\_dst.s\_addr));

ptr += sizeof (iphdr.ip\_dst.s\_addr);

chksumlen += sizeof (iphdr.ip\_dst.s\_addr);

// Copy zero field to buf (8 bits)

\*ptr = 0; ptr++;

chksumlen += 1;

// Copy transport layer protocol to buf (8 bits)

memcpy (ptr, &iphdr.ip\_p, sizeof (iphdr.ip\_p));

ptr += sizeof (iphdr.ip\_p);

chksumlen += sizeof (iphdr.ip\_p);

// Copy TCP length to buf (16 bits)

svalue = htons (sizeof (tcphdr));

memcpy (ptr, &svalue, sizeof (svalue));

ptr += sizeof (svalue);

chksumlen += sizeof (svalue);

// Copy TCP source port to buf (16 bits)

memcpy (ptr, &tcphdr.th\_sport, sizeof (tcphdr.th\_sport));

ptr += sizeof (tcphdr.th\_sport);

chksumlen += sizeof (tcphdr.th\_sport);

// Copy TCP destination port to buf (16 bits)

memcpy (ptr, &tcphdr.th\_dport, sizeof (tcphdr.th\_dport));

ptr += sizeof (tcphdr.th\_dport);

chksumlen += sizeof (tcphdr.th\_dport);

// Copy sequence number to buf (32 bits)

memcpy (ptr, &tcphdr.th\_seq, sizeof (tcphdr.th\_seq));

ptr += sizeof (tcphdr.th\_seq);

chksumlen += sizeof (tcphdr.th\_seq);

// Copy acknowledgement number to buf (32 bits)

memcpy (ptr, &tcphdr.th\_ack, sizeof (tcphdr.th\_ack));

ptr += sizeof (tcphdr.th\_ack);

chksumlen += sizeof (tcphdr.th\_ack);

// Copy data offset to buf (4 bits) and

// copy reserved bits to buf (4 bits)

cvalue = (tcphdr.th\_off << 4) + tcphdr.th\_x2;

memcpy (ptr, &cvalue, sizeof (cvalue));

ptr += sizeof (cvalue);

chksumlen += sizeof (cvalue);

// Copy TCP flags to buf (8 bits)

memcpy (ptr, &tcphdr.th\_flags, sizeof (tcphdr.th\_flags));

ptr += sizeof (tcphdr.th\_flags);

chksumlen += sizeof (tcphdr.th\_flags);

// Copy TCP window size to buf (16 bits)

memcpy (ptr, &tcphdr.th\_win, sizeof (tcphdr.th\_win));

ptr += sizeof (tcphdr.th\_win);

chksumlen += sizeof (tcphdr.th\_win);

// Copy TCP checksum to buf (16 bits)

// Zero, since we don't know it yet

\*ptr = 0; ptr++;

\*ptr = 0; ptr++;

chksumlen += 2;

// Copy urgent pointer to buf (16 bits)

memcpy (ptr, &tcphdr.th\_urp, sizeof (tcphdr.th\_urp));

ptr += sizeof (tcphdr.th\_urp);

chksumlen += sizeof (tcphdr.th\_urp);

return checksum ((uint16\_t \*) buf, chksumlen);

}

// Allocate memory for an array of chars.

char \*

allocate\_strmem (int len)

{

void \*tmp;

if (len <= 0) {

fprintf (stderr, "ERROR: Cannot allocate memory because len = %i in allocate\_strmem().\n", len);

exit (EXIT\_FAILURE);

}

tmp = (char \*) malloc (len \* sizeof (char));

if (tmp != NULL) {

memset (tmp, 0, len \* sizeof (char));

return (tmp);

} else {

fprintf (stderr, "ERROR: Cannot allocate memory for array allocate\_strmem().\n");

exit (EXIT\_FAILURE);

}

}

// Allocate memory for an array of unsigned chars.

uint8\_t \*

allocate\_ustrmem (int len)

{

void \*tmp;

if (len <= 0) {

fprintf (stderr, "ERROR: Cannot allocate memory because len = %i in allocate\_ustrmem().\n", len);

exit (EXIT\_FAILURE);

}

tmp = (uint8\_t \*) malloc (len \* sizeof (uint8\_t));

if (tmp != NULL) {

memset (tmp, 0, len \* sizeof (uint8\_t));

return (tmp);

} else {

fprintf (stderr, "ERROR: Cannot allocate memory for array allocate\_ustrmem().\n");

exit (EXIT\_FAILURE);

}

}

// Allocate memory for an array of ints.

int \*

allocate\_intmem (int len)

{

void \*tmp;

if (len <= 0) {

fprintf (stderr, "ERROR: Cannot allocate memory because len = %i in allocate\_intmem().\n", len);

exit (EXIT\_FAILURE);

}

tmp = (int \*) malloc (len \* sizeof (int));

if (tmp != NULL) {

memset (tmp, 0, len \* sizeof (int));

return (tmp);

} else {

fprintf (stderr, "ERROR: Cannot allocate memory for array allocate\_intmem().\n");

exit (EXIT\_FAILURE);

}

}