CPE 449/549
PROJECT: PORT SCAN DETECTION

OBJECTIVES

All students will complete the following:

- Use Python DPKT library to parse PCAP files from Wireshark. Alternatively, you may use the LIBPCAP library and a C program.
- Detect port scans of various types.

Graduate students will also complete the following:

- Consider how a similar program could be written to detect port knocking.

NMAP SCAN REFERENCE FILES

Use NMAP and Wireshark to create separate PCAP files for each of the following scan types. You will use these files to test your port scan detector program.

- TCP Syn Scan (Half-open Scan)
- TCP Connect Scan
- UDP Scan
- TCP NULL Scan
- TCP Christmas Scan

Steps:

1. Start Wireshark and begin capturing packets.
2. Start an NMAP scan.
3. Stop Wireshark capture and the NMAP scan after a sufficient number of packets have been captured (a few thousand).
4. Save as "Wireshark/tcpdump … (*.pcap, …)"

PYTHON DPKT LIBRARY

- The Python DPKT library can be used to build network packets and to parse PCAP files.
- Reference for using DPKT for parsing PCAP files.
    - https://jon.oberheide.org/blog/2008/10/15/dpkt-tutorial-2-parsing-a-pcap-file/
- DPKT Code examples are at the following link. Navigate to the "Examples" link.
    - http://dpkt.readthedocs.io/en/latest/index.html

LIBPCAP

LIBPCAP is a library used to capture network transactions from a network card. It is used by Wireshark. We will use it to capture network transactions and print information about those transactions. LIBPCAP can open a live connection to your NIC card or can run from files. We will use both. The difference is actually trivial.

Here are some useful functions:

Function	Description

pcap_lookupdev	Returns a pointer to a string giving the name of a network device suitable for use with pcap_open_live.
pcap_open_live	Creates live connection to the network device. Returns pointer to pcap_t which is used like a file handle.
pcap_loop	Calls a call back function (the pcap handler routine) specified by the user every time a Ethernet packet is captured.
pcap_open_offline	Opens a ".pcap" file. Returns same handle as pcap_open_live.
pcap_dispatch	Similar to pcap_loop. Used to process all Ethernet packets found in the open ".pcap" file. Calls the pcap handler routine each time a Ethernet packet is found in the open file.
ntohs	Converts an unsigned short integer from network byte order to host byte order. This handles endianess so you don't have to. If the network endianess is the same as your host's endianess this does nothing. If they are different this converts the number for you.
ether_ntoa	Converts a Ethernet MAC address from network format (hex) to a string which looks like your typical MAC address (eg. 01:23:45:67:89:ab). Use this to print the MAC address.
inet_ntoa	Converts a network address in a struct in_addr to a dots-and-numbers format string. (eg. 192.168.1.1). Use this to print the IP address.

Here are some useful structs:

Struct	Description

struct ether_header	Ethernet header –
	`struct ether_header`
	`{`
	`  u_int8_t  ether_dhost[ETH_ALEN];      /* destination eth addr */`
	`  u_int8_t  ether_shost[ETH_ALEN];      /* source ether addr    */`
	`  u_int16_t ether_type;                 /* packet type ID field */`
	`} __attribute__ ((__packed__));`
	
	`ether_type tells you what is in the packet:`
	
	#define ETHERTYPE_IP 0x0800 /* IP */
	#define ETHERTYPE_ARP 0x0806 /* Address resolution */
	#define ETHERTYPE_IPV6 0x86dd /* IP protocol version 6 */
	
	Available in net/ethernet.h which is included in netinet/if_ether.h.

struct ip	IP header –
	`struct ip {`
	`   unsigned int   ip_hl:4; /* both fields are 4 bits */`
	`   unsigned int   ip_v:4;`
	`   uint8_t        ip_tos;`
	`   uint16_t       ip_len;`
	`   uint16_t       ip_id;`
	`   uint16_t       ip_off;`
	`   uint8_t        ip_ttl;`
	`   uint8_t        ip_p;`
	`   uint16_t       ip_sum;`
	`   struct in_addr ip_src;`
	`   struct in_addr ip_dst;`
	`};`
	
	`ip_src - source IP address`
	`ip_dst - destination IP address`
	
	Available in netinet/ip.h.

Struct	Description
struct tcphdr	TCP header -

```
struct tcphdr {
        unsigned short source;
        unsigned short dest;
        unsigned long seq;
        unsigned long ack_seq;
        #  if __BYTE_ORDER == __LITTLE_ENDIAN
        unsigned short res1:4;
        unsigned short doff:4;
        unsigned short fin:1;
        unsigned short syn:1;
        unsigned short rst:1;
        unsigned short psh:1;
        unsigned short ack:1;
        unsigned short urg:1;
        unsigned short res2:2;
        #  elif __BYTE_ORDER == __BIG_ENDIAN
        unsigned short doff:4;
        unsigned short res1:4;
        unsigned short res2:2;
        unsigned short urg:1;
        unsigned short ack:1;
        unsigned short psh:1;
        unsigned short rst:1;
        unsigned short syn:1;
        unsigned short fin:1;
        #  endif
        unsigned short window;
        unsigned short check;
        unsigned short urg_ptr;
};
```

source – source port
dest – destination port
seq – sequence number

syn, ack, fin, rst, psh, urg – flags

Available in netinet/tcp.h.

Struct	Description

struct udphdr	UDP header –
	`struct udphdr`
	`{`
	`  u_int16_t source;`
	`  u_int16_t dest;`
	`  u_int16_t len;`
	`  u_int16_t check;`
	`};`
	
	source – source port
	dest – destination port
	len – length
	
	Available in netinet/udp.h.

struct icmphdr	ICMP header –
	struct icmphdr
	{
	 u_int8_t type; /* message type */
	 u_int8_t code; /* type sub-code */
	 u_int16_t checksum;
	 union
	 {
	  struct
	  {
	   u_int16_t id;
	   u_int16_t sequence;
	  } echo; /* echo datagram */
	  u_int32_t gateway; /* gateway address */
	  struct
	  {
	   u_int16_t __unused;
	   u_int16_t mtu;
	  } frag; /* path mtu discovery */
	 } un;
	};
	
	type – indicates the ICMP message type. ICMP echo request is type 8.
	
	Available in netinet/ip_icmp.h.

pcap_t	Think of this as a file handle pointing to either an open ".pcap" file or a live connection to the NIC.

The routines *pcap_loop* and *pcap_dispatch* call a function that you define when a Ethernet packet is captured. This function is called the handler. The handler must have the following port list:

void handler(u_char *args, const struct pcap_pkthdr *header, const u_char *packet);

You define the handler and provide a reference to it when you call *pcap_loop* and *pcap_dispatch.* The packet variable in the handler port list is a pointer to the Ethernet header and its data fields.

Remember Ethernet packets encapsulate IP packets. IP packets encapsulate ICMP, TCP or UDP packets.

In memory the packet pointer points to memory which looks like:

Address	What is there?
packet	Ethernet Header
packet + sizeof(Ethernet header)	Ethernet Data = IP Header
packet + sizeof(Ethernet header) + size of(IP header)	IP Data = TCP, UDP, or ICMP header
packet + sizeof(Ethernet header) + size of(IP header) + sizeof(TCP or UDP header)	Data

So you can use pointer arithmetic shown above to create pointers to the various headers. From there you can type cast that pointer and assign it to a pointer to a struct ether_header, struct ip, struct tcphdr, or struct udphdr. From there you can easily print out the required MAC and IP addresses and port numbers.

SOURCE CODE FILE NAME REQUIREMENT

Your script or program must be named one of the following.

C Programs: <chargerID>.c

Python scripts: <chargerID>.py

Replace <ChargerID> with your Charger ID. Your Charger ID is your initials followed by 4 numeric digits and is the prefix (before the @ symbol) of your UAH email address. Instructions to look up your Charger ID are available here: https://www.uah.edu/oit/services/charger-id-and-password.

ADDITIONAL C PROGRAM REQUIREMENT

You code will be compiled by a script with the following shell code. C++ is not allowed. No other library locations are allowed.

```
gcc -I/usr/lib -lpcap -o <chargerID>.exe <chargerID>.c
```

ADDITIONAL PYTHON SCRIPT REQUIREMENT

Your script should function with Python 2.7. Python 3 is not allowed.

COMMAND LINE ARGUMENT REQUIREMENTS

You script should accept two command line arguments; -i <filename>
The script will open and parse <filename>.

OUTPUT REQUIREMENTS

Your code should output only the following:

```
Null: <number of detect Null packets in PCAP input>
XMAS: <number of detect XMAS packets in PCAP input>
UDP: <number of detect UDP packets in PCAP input>
Half-open: <number of detect Half-Open packets in PCAP input>
Connect: <number of detect Connect packets in PCAP input>
```

Replace the <text> with a number for each line above.

For each type of scan your code should print the number of unique ports scanned. For a given IP address the scanner will try many individual ports (count each unique port for that unique IP address). If there are multiple IP addresses scanned, count all the ports scanned for each IP address.

Some port scans have multiple network packets per port. Do not count each packet. For example, an open port for a connect scan will have 3 packets (syn, syn+ack, ack). Only count that group of packets once.

REQUIREMENTS

Null Scan	You program should provide the number of detected Null scans in any PCAP file.	nmap –sN 192.168.0.0/24
Xmas Scan	You program should provide the number of detected XMAS scans in any PCAP file.	nmap –sX 192.168.0.0/24
UDP Scan	You program should provide the number of detected UDP scans in any PCAP file.	nmap –sU 192.168.0.0/24
Half-Open Scan	You program should provide the number of detected Half-Open scans in any PCAP file.	nmap –sS 192.168.0.0/24
Connect Scan	You program should provide the number of detected Connect scans in any PCAP file. Your program should differentiate between Connect and Half-Open scans.	nmap –sT 192.168.0.0/24

GRADING

Item	Points

Input/Output	Correct format: 20 Correct command line argument implementation: 20
Detect NULL	Exact correct #: 10 +/- 5% of correct #: 5
Detect XMAS	Exact correct #: 10 +/- 5% of correct #: 5
Detect UDP	Exact correct #: 20 +/- 5% of correct #: 15
Detect Connect Scans	Exact correct #: 10 +/- 5% of correct #: 10 +/- 25% of correct #: 5
Detect Half-Open Scans	Exact correct #: 10 +/- 5% of correct #: 10 +/- 25% of correct #: 5
Total	80