https://github.com/je-clark/DFC19-PacketAnalysis

Example 1: Split a packet capture based on time

Take a large packet capture with known timestamps of interest, and break those times out for manual analysis

Required knowledge: PCAP File Format

Data Sourced from https://wiki.wireshark.org/Development/LibpcapFileFormat

```
PCAP
└── GlobalHeader
    ├── Packet0Header
    │   └── PacketContents
    └── Packet1Header
        └── PacketContents
```

Global Header Format

```
typedef struct pcap_hdr_s {
        guint32 magic_number;   /* magic number */
        guint16 version_major;  /* major version number */
        guint16 version_minor;  /* minor version number */
        gint32  thiszone;       /* GMT to local correction */
        guint32 sigfigs;        /* accuracy of timestamps */
        guint32 snaplen;        /* max length of captured packets, in
        guint32 network;        /* data link type */
} pcap_hdr_t;
```

Packet Header Format

```
typedef struct pcaprec_hdr_s {
        guint32 ts_sec;         /* timestamp seconds */
        guint32 ts_usec;        /* timestamp microseconds */
        guint32 incl_len;       /* number of octets of packet saved i
        guint32 orig_len;       /* actual length of packet */
} pcaprec_hdr_t;
```

What Do We Care About?

1. Magic Number: *Indicates the endianness of the system that generated the PCAP*
2. Timestamp (seconds): *Compare this with the given timestamp of interest*
3. Included Length: *Necessary to move the packet contents to a new file*

Let's Do It!

```
[ ]  # Imports
     import sys, os, argparse
     from datetime import datetime
```

```
[ ]  def cli_args():
         ep='''\
     Default output file is <infile>_<anchor_time>_<duration>. To
     overwrite, use the -o flag.

     This will use local system time to compute timestamps. If the
     capture originated in
     a different time zone, convert the anchor_time argument to local
     time when calling this application.
     '''
         desc = 'Automatically slice PCAP files by time.'
         parser = argparse.ArgumentParser(description=desc, epilog=ep,
     formatter_class=argparse.RawTextHelpFomatter)
         parser.add_argument('infile', action='store', help='Input
     file. Must be a PCAP')
         parser.add_argument('anchor_time', nargs=2, action='store',
     help='Start time or center time. Use format mm/dd/yy hh:mm:ss')
         parser.add_argument('duration', action='store',
     help='Duration to end time, or duration from center time when
     using -s. In seconds')
         parser.add_argument('-s', '--symmetric', action='store_true',
     help='Interpret the anchor time as the center point')
         parser.add_argument('-o', '--outfile', action='store',
     help='Override the default output file name')

         return parser.parse_args()
```

```
[ ]  def determine_endianness(global_header): # Expect this to be a
     byte array
         magic_num_test = b'x\a1\xb2\xc3\xd4'
         magic_num = global_header[0:4]
         if magic_num == magic_num_test:
             return 'big'
         else:
             return 'little'
```

```
[ ]  def pcap_split(infile, outfile, anchor_time, duration,
     symmetric=False): # Expecting most of this to be command line
     input, so strings
         # Get start and end time
         duration = int(duration)
         anchor = int(datetime.strptime(' '.join(anchor_time),
     r'%m/%d/%y %H:%M:%S').timestamp())
         if symmetric:
             start_time = anchor - duration
             end_time = anchor + duration
         else:
             start_time = anchor
             end_time = anchor + duration

         print(f'This uses UTC timestamps for start time
     ({start_time}) and end time ({end_time})')

         # Open input and output files
         in_ptr = open(infile, 'rb+')
         out_ptr = open(outfile, 'wb+')

         # Global Header
         global_header = in_ptr.read(24)
         endianness = determine_endianness(global_header)
         print(f'This file was generated on a {endianness}-endian
     computer')
         out_ptr.write(global_header)

         # Packets
         while True:
             # Packet Header
             packet_header = in_ptr.read(16)
             if not packet_header:
                 break # Python returns '' for EOF, which evaluates
     FALSE

             timestamp = int.from_bytes(packet_header[0:4],
     byteorder=endianness)
             packet_length = int.from_bytes(packet_header[8:12],
     byteorder=endianness)
```

```
            # Determine if we care about this packet
            if timestamp > end_time: break

            if start_time <= timestamp:
                packet = in_ptr.read(packet_length)
                out_ptr.write(packet_header)
                out_ptr.write(packet)
            else:
                # Move the file pointer to the beginning of the
                # next packet header
                in_ptr.seek(packet_length,1)

    # Be nice to your pointers. Put them away when you're done
with them
    in_ptr.close()
    out_ptr.close()

    return
```

Let's start messing with it

My sample capture is './sample.pcap'

Starting date/time is Apr 18, 2019 9:33:03

Ending date/time is Apr 26, 2019 11:25:07

```
[ ]  # Just get everything from April 19?
     pcap_split('./sample.pcap', 'apr_19.pcap',
     ['04/19/19','00:00:00'], '86400')
```

```
[ ]  # Let's get April 20 symmetrically
     pcap_split('./sample.pcap', 'apr_20_symmetric.pcap',
     ['04/20/19','12:00:00'], '43200', symmetric=True)
```

Example 2: TLS Version Audit

Audit the TLS version used by incoming connections to the server so we can upgrade disable insecure protocols without relying on a scream test

Required Knowledge: Packet Layer Headers

Ethernet Header

Modern computers use Type II Ethernet

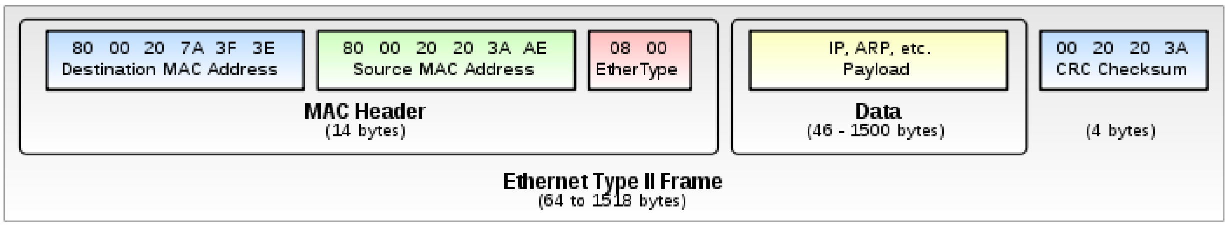
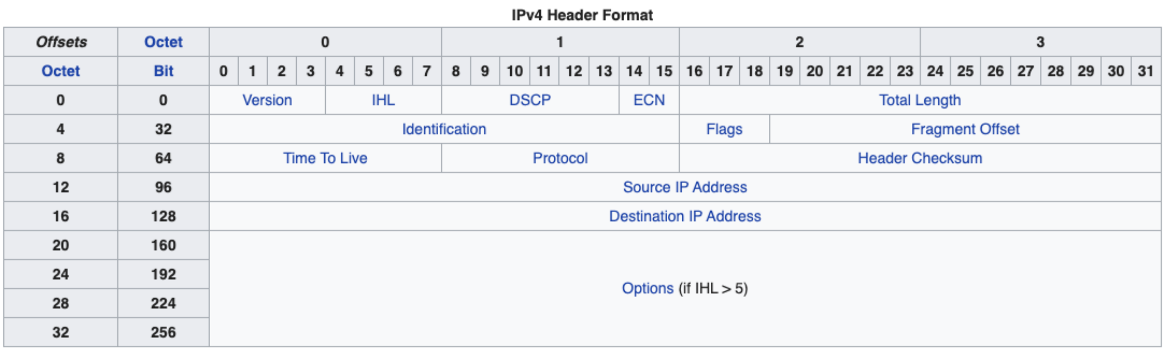


Image from https://en.wikipedia.org/wiki/Ethernet_frame

IP Header

We're only going to work with IPv4 for this example. It's complicated enough already.



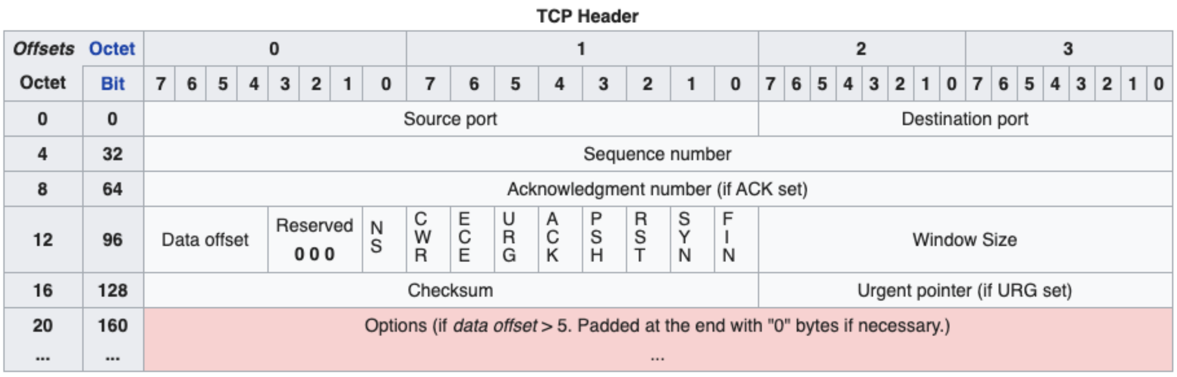
IPv4 Header Format

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags			Fragment Offset												
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															
24	192																																
28	224																																
32	256																																

Image screenshotted from https://en.wikipedia.org/wiki/IPv4

TCP Header

There's still more.



TCP Header

Offsets	Octet	0								1								2								3							
Octet	Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0			NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
16	128	Checksum																Urgent pointer (if URG set)															
20	160	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.)																															
...	...	...																															

Image screenshotted from https://en.wikipedia.org/wiki/Transmission_Control_Protocol

TLS Record Layer

There are many components to TLS, so the record layer keeps track of them.

Byte	0	1	2	3	4
0	Content Type	Version		Length	
1	Payload				

TLS Handshake Layer

These are just the parts we care about for this application.

Byte	0	1	2	3	4	5
0	Handshake Type	Length			Version	

General Approach

- The final, negotiated TLS version is reported in the Server Hello. We need to find those and extract the client's information
- Emphasize speed of execution. Interpret values out of order to agressively skip packets

Let's Go!

```
# Set up classes as a useful structure to store information and
retrieve it later

class TLSSummary:
    def __init__(self, src_ip, dst_ip, src_port, dst_port,
tls_version):
        self.src_ip = src_ip
        self.dst_ip = dst_ip
        self.src_port = src_port
        self.dst_port = dst_port
        self.tls_version = tls_version

    def get_src_ip(self):
        return self.src_ip

    def get_dst_ip(self):
        return self.dst_ip

    def get_src_port(self):
        return self.src_port

    def get_dst_port(self):
            return self.dst_port

    def get_tls_version(self):
            return self.tls_version

    def print_summary(self):
            sum = '''
            Source IP:          {}
            Destination IP:     {}
            Source Port:        {}
            Destination Port:   {}
            TLS Version:        {}
            '''.format(self.src_ip, self.dst_ip, self.src_port,
self.dst_port, self.tls_version)
            print(sum)
```

```
[ ]  # A few helper functions

def bytes_to_ip_str(ip_in_bytes):

    first_octet = str(int.from_bytes(ip_in_bytes[0:1], byteorder
= 'big'))
    second_octet = str(int.from_bytes(ip_in_bytes[1:2], byteorder
= 'big'))
    third_octet = str(int.from_bytes(ip_in_bytes[2:3], byteorder
= 'big'))
    fourth_octet = str(int.from_bytes(ip_in_bytes[3:4], byteorder
= 'big'))

    return '.'.join([first_octet, second_octet, third_octet,
fourth_octet])

def is_server_hello(packet):

    # We know we have some things between us and the TLS layer
    # Let's deal with them one at a time

    # Ethernet Header - Always 14 bytes, unless we have VLANs.
That's beyond the scope of this script.
    eth_hdr_len = 14

    # IP Header - Length defined in the IHL field in words.
    WORDS_TO_BYTES = 4 # Conversion factor
    IHL_MASK = int('00001111',2) # IHL field isn't a full byte,
so we need to use a bit mask to get the value
    ver_ihl_field = packet[eth_hdr_len:eth_hdr_len + 1] #
Version/IHL field is the first one in the IP Header
    ihl = int.from_bytes(ver_ihl_field, byteorder = 'big') &
IHL_MASK # Bitwise AND
    ip_hdr_len = ihl * WORDS_TO_BYTES
    # TCP Header - Length defined in data offset field
    DATA_OFFSET_MAST = int('11110000',2)
    data_offset_byte = packet[eth_hdr_len + ip_hdr_len + 12:
eth_hdr_len + ip_hdr_len + 12 + 1]
    # Since the data offset is upper 4 bits of the byte, we need
to shift it down to get the right value
    tcp_hdr_len = ((int.from_bytes(data_offset_byte, byteorder =
'big') & DATA_OFFSET_MAST) >> 4) * WORDS_TO_BYTES

    # Look at TLS Layer
    tls_layer = eth_hdr_len + ip_hdr_len + tcp_hdr_len
    TLS_HANDSHAKE = b'\x16' # Evaluates to an integer value of 22
    SERVER_HELLO = b'\x02'
    tls_content_type = packet[tls_layer:tls_layer+1]
    if tls_content_type == TLS_HANDSHAKE:
        handshake_type = packet[tls_layer + 5: tls_layer + 5 + 1]
        if handshake_type == SERVER_HELLO:
            tls_version = packet[tls_layer + 9: tls_layer + 9 +
2]
            return (True, tls_version, tcp_hdr_len, ip_hdr_len)
            # So yes, this violates "do one thing well", but I'm
already here, already have the info,
            # and I've already told you I'm bad at coding
    return (False, '',0,0) # It's not what I wanted, so I'm mad
at the world and don't wanna give info

def get_ip_addrs(ip_hdr):
    src_ip_bytes = ip_hdr[12:16]
    dst_ip_bytes = ip_hdr[16:20]

    src_ip = bytes_to_ip_str(src_ip_bytes)
    dst_ip = bytes_to_ip_str(dst_ip_bytes)

    return src_ip, dst_ip

def get_tcp_ports(tcp_hdr):
    src_port_bytes = tcp_hdr[0:2]
    dst_port_bytes = tcp_hdr[2:4]

    src_port = int.from_bytes(src_port_bytes, byteorder='big')
    dst_port = int.from_bytes(dst_port_bytes, byteorder='big')

    return src_port, dst_port

def readable_tls_version(tls_version_bytes):

    if tls_version_bytes == b'\x03\x01':
        return "TLS 1.0"
    elif tls_version_bytes == b'\x03\x02':
        return "TLS 1.1"
    elif tls_version_bytes == b'\x03\x03':
        return "TLS 1.2"
    elif tls_version_bytes == b'\x03\x00':
        return "SSL 3.0"
    elif tls_version_bytes == b'\x03\x04':
        return "TLS 1.3"
    else:
        print(f'Unknown version displayed as
{tls_version_bytes.hex()}')
        return "Unknown"
```

```
[ ]  def tls_version_audit(pcap_file):

         # Open File
         pcap_ptr = open(pcap_file, 'rb+')

         # Get Endianness
         endianness = determine_endianness(pcap_ptr.read(24))

         # Deal with the packets
         tls_connection_list = []
         while True:

             # Get a packet
             pkt_hdr = pcap_ptr.read(16)
             if not pkt_hdr:
                 break
             pkt_len = int.from_bytes(pkt_hdr[8:12],
     byteorder=endianness)
             pkt = pcap_ptr.read(pkt_len)

             # Is it a Server Hello?
             resp = is_server_hello(pkt)
             if not resp[0]:
                 continue # Go to the next packet
             # I would wrap this in an else:, but I don't need to
             tls_version_bytes = resp[1]
             tcp_hdr_len = resp[2]
             ip_hdr_len = resp[3]

             # Gather necessary values
             # Really only need server port and client IP, but
     collecting both source and destination because I can
             ip_hdr = pkt[14: 14 + ip_hdr_len]
             tcp_hdr = pkt[14 + ip_hdr_len: 14 + ip_hdr_len +
     tcp_hdr_len]

             src_ip, dst_ip = get_ip_addrs(ip_hdr)
             src_port, dst_port = get_tcp_ports(tcp_hdr)
             tls_version = readable_tls_version(tls_version_bytes)
```

```
        tls_connection_list.append(TLSSummary(src_ip, dst_ip,
src_port, dst_port, tls_version))

    # Let's see how we did!
    print(f'There were {len(tls_connection_list)} connections
observed')
    for conn in tls_connection_list:
        if conn.get_tls_version() in ("SSL 3.0", "TLS 1.0", "TLS
1.1", "Unknown"):
            conn.print_summary()
```

```
[ ]    tls_version_audit('./sample.pcap')
```