PROCESSING NETWORK CAPTURE DATA

In order to solidify our understanding of the "enveloping" nature of the layered approach to networking, you are going to parse a binary dump of network data. During this exercise you will:

- Parse binary data according to network protocol specifications.
- Understand exactly how the network layers encapsulate data provided by "higher" layers, and how they handle data from lower layers.
- Play and experiment with real network data.

By the time you are finished you will have written a program that interacts with critical sections of some very common and important network protocols including: Ethernet, TCP, IP, and HTTP.

STAGE ZERO: WHAT DO WE KNOW ABOUT THE PROVIDED DATA?

In order to properly parse any data, we have to know what the format of that data is. The file we've provided to you represents data in the format that it was sent across the internet. The data was collected using a network capture tool, and represents the total data sent during a single HTTP request/response cycle. The HTTP request was for a particular `jpg` image which was delivered by a server. Your ultimate goal is to extract this image data and write it to a file on your computer.

As we have learned, as data travels down the network layer hierarchy, the layer below accepts the data and wraps it with it's own format. In our capture we have 4 of the 5 layers of data represented:

- Application: HTTP
- Transport: TCP
- Network: IP
- Link: Ethernet

In addition to our knowledge of the internet layers, you need know that this data is saved in a specific file format, which is documented here. There are two notable facets to the `.cap` savefile: the **global header__ and the __per packet header**.

Even the most careful reading of the documentation will not tell you anything about which protocols are represented at each network layer. In fact, without reading the data **at each layer** all we can know is that we have data like this:

- One global header, followed by
- Some number of packets, each of which with a header that will tell you long that particular packet is.

Here are some useful facts about **this specific capture** which can help you validate your findings as you go:

- There are 99 packets.
- None of the packets have been truncated at all by the capture process
    - Meaning 100% of the data sent between the two hosts represented here
- All of the IP datagrams use the same version of IP (one of IPv4 or IPv6)
- A lossy connection was simulated, so individual packets may not have arrived exactly once, and may or may not have arrived in the order they were sent.
    - However, we do ensure that all of the data for both the HTTP request and response are represented in the capture.

- There are exactly 2 hosts (2 IP addresses, 2 MAC addresses).

> Some Advice: When doing binary parsing, it is incredibly helpful to program defensively by using lots of assertions to detect if something is wrong with our assumptions.

STAGE ONE: READ THE PCAP HEADERS

The Global Header

Before you start programming, try to manually parse the global Pcap header. This will give you some practice "thinking in binary" and it will force us to encounter and tackle the concept of "endian-ness". Use the xxd command to turn the provided binary dump into a hex dump then, Using the pcap documentation:

- Examine the "Magic Number",
  - What order is it in?
  - What does that tell you about parsing the rest of the pcap provided data?
    - **It plays a crucial role, so don't take this step lightly**
- What are the Major and Minor versions?
- Verify that the values which are always zero are in fact zero.
- What is the snapshot length?
  - Sanity check, the snapshot length is the longest a single packet can be in the capture. Does the number look small enough to be the size of a single packet (in bytes)?
- What is the link layer header type?
  - **We will need the specification details of this header in stage 2**

The Per Packet Header

The bits immediately following the global header will be the first per-packet header data. Parse these values manually as well:

- What is the size of the first packet?
- Verify that the captured length field matches the untruncated length field.

Now, you should start writing a program. You can use any language you want for this, provided the language has mechanisms for reading binary data (but even JavaScript has this so if you're choosing a popular language it should be no problem). Your goal is to read every individual packet header. You should write a program that can:

- Read the captured and total length of each packet
  - Verify that for every packet the captured length and total length are equal.
  - (this is true for the file we provided, but not in general)
    - This will also tell you where the next packet header starts.
- Verify that there are 99 Packets represented in this data.
  - (Again, this is true in the provided data, not generally)

STAGE TWO: READ THE ETHERNET HEADERS

Once you've read the per-packet header for each packet, you want to peel off one more layer. The next layer will be the Link Layer, in our case Ethernet. You should be able to verify from the global header the type of the link layer used in this capture. It's valuable practice to first try and use Google to find the exact specification for this header.

> If you spend more than 10 minutes trying to track it down and only encounter frustration, look ahead at the **spoilers** section for a direct link to the header.

Once you have determined the format of the header, extend your program to:

- Determine the version of the wrapped IP datagram (IPv6 or IPv4) so we can parse that data
    - Verify that all the IP datagrams have the same format.
    - (It's true in this data, but not in an arbitrary capture, that all the IP datagrams use the same IP version)
    - Print the source and destination MAC addresses
    - Verify that the MAC addresses make sense

STAGE THREE: READ THE IP HEADERS

Once again, you should strive to find the specification of the IP header yourself, but it is linked below in the spoilers section. Once you've determined the format of the header you should be able to extend your program to:

- Determine the length of the IP header for each datagram
    - **You'll need this information to later, so save it somehow**
- Determine the length of the datagram payload
- Determine the source and destination IP addresses
    - Verify that the IP addresses make sense
- Determine the relative starting location of the enclosed transport layer segment
- Determine the transport protocol being used
    - Verify that the same transport protocol used for all the IP datagrams

> Keep in mind, only one of the two hosts ever sends image data. If you want to build the image from the data here, you will need to filter the packets somehow…

STAGE FOUR: READ THE TRANSPORT HEADERS

Once you've parsed the IP headers, you'll know where the transport headers start, and which protocol is being used. Once again, you may find the specification yourself, and it is linked in the spoilers section. Once you have this information you should extend your program to:

- Determine the ports used to communicate
    - Sanity check: How many ports are used? Which ports are they? Does that sound right for the protocols being used?
- Determine the length of each transport header
- Determine the sequence number for this packet
- Extract the HTTP data from the packet and store it somewhere

> Keep in mind, packets can arrive out of order. If you want to build the image, you'll need to reconstruct the original order somehow…

STAGE FIVE: PARSE THE HTTP DATA

You should have already stored all the data somewhere, now you need to put it in order, and parse it as an HTTP request. Extend your code to:

- Order the received data by TCP sequence number
- Combine it into a single binary string
- Interpret that binary string as text
- Print that text and verify that the headers all make sense
- Extract just the body, write it to disk as a file and save that file with a `.jpg` extension so your operating system knows what to do with it
- View the file

- Rejoice

SPOILERS:

- Ethernet Header Format
- IP Header Format
- Transport Layer Header Format