Computer Networking
Lab: Parsing Captured Network Packets
Exercise instructions

PCapture the Flag

Friday, 21 June 2019

DOWNLOAD EXERCISE FILES

In order to solidify our understanding of the layered nature of network protocols, we will be parsing a binary dump of raw network data. By the time you have finished, you will have read and parsed headers for four very important network protocols: Ethernet, IP, TCP and HTTP, and all in raw binary!

Objective

We have recorded a packet capture of an HTTP request and response for an image, performed over an imperfect network. The challenge for you is to parse the capture file, find and parse the packets constituting the image download, and reconstruct the image! It's like a murder mystery, except with a trail of binary data and a hero rather than a villain at the end of it.

At this point, if you have feel that you have a decent understanding of how network protocols are layered, and are confident that you could

look up any protocol's header format as needed, you're ready to proceed! The remainder of this document provides more detailed instructions, for those who need it.

NOTE

If you enjoy "capture the flag" style challenges, you should stop reading here, and start parsing!

Layers of Protocols

As we have learned, the data necessary at one layer of the network protocol stack is wrapped around—and later unwrapped from—the data of the protocol above it.

For this exercise, we have used an Ethernet connection to request and download an image from the web. The request was constructed as an HTTP message, which was wrapped in a TCP segment, followed by an IP datagram, and finally an Ethernet frame to be sent over the wire to the router.[1]

As our message traveled over the network, it was repeatedly unwrapped and re-wrapped to a certain depth: routers parsed the link layer and network layer data, made some changes, and passed the message along. At the server, the entire message was ultimately unwrapped so that the server process could received the request. In order to return a response, the entire process was followed again, with the extra complication that the JPEG file was large enough that it needed to be split into multiple TCP segments, which became multiple IP datagrams, and multiple Ethernet frames.

The program you write will follow this pattern: you will parse the Ethernet frames first, then the IP datagrams from within them, follow by the TCP segments, which when stitched back together will enable you to parse out the HTTP messages.

The pcap File Format

We used the command line utility `tcpdump` to capture the network traffic.[2]

The file is saved as a particular format: a "pcap-savefile". You will need to understand the format to make progress at this point; thankfully it is relatively straightforward, and well documented in `man pcap-savefile` (online version).

In short, the "pcap-savefile" has a global header, followed by a sequence of captured network packets, all with their own per-packet headers. You will need to parse these in order to distinguish the captured packets from one another.

The Global `pcap-savefile` Header

Before you start programming, try to parse the global pcap-savefile header by hand, using a tool like `xxd`. This will give you some practice working with binary, dealing with byte ordering and using `xxd`. `man pcap-savefile` should give you all the information you need about the header format.

Here are some questions you might ask yourself to ensure that you're understanding what's in front of you.

- What's the magic number? What does it tell you about the byte ordering in the pcap-specific aspects of the file?[3]

  3 Your instructor should have already reviewed the concept of byte ordering with you. If they haven't, please remind them!

- What are the major and minor versions? Don't forget about the byte ordering!
- Are the values that ought to be zero in fact zero?
- What is the snapshot length?
- What is the link layer header type?

Most importantly, we will need to know the link layer header type, to ensure that what we've captured are indeed Ethernet frames, as well as the overall pcap-savefile header length, to know where to start looking for actual packets![4]

Per-packet Headers

The bytes 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?
- Was any data truncated?

Now, you should start writing a program with an intermediate goal: to count the total number of packets captured in the file.

To do this, you will need to:

- Read the captured and total length of each packet.
- Verify that the captured and total length are the same.

- Use this length to determine where the *next* packet's header begins.
- Repeat the process on the next packet, until you have reached the end of the file.

NOTE

Counting the packets, you should end up with 99, with no remaining data. If you have arrived at a different number, it's time to debug your program!

Now that we have a meaningful program, we'd like to provide some programming advice: when doing binary parsing, it's incredibly helpful to program defensively by liberally using assertions to catch any bugs or incorrect assumptions as close to the source as possible. Otherwise, you might end up with an answer that is simply *wrong*, caused by an upstream issue that's hard to track down.

Parsing the Ethernet Headers

Once you've read the per-packet header for each packet, you will be able to parse and peel off the next layer. At this point, we encounter our first network protocol: Ethernet, a link layer protocol.

In order to parse the Ethernet headers, you will need to know the format! It's valuable practice to first try to find the exact specification for this header yourself, but if you spend more than 10 minutes trying to track it down and only encounter frustration, scroll ahead to the spoilers for a direct link to the header.

NOTE

We are only able to capture the portion of an Ethernet frame that the network interface controller makes available to the operating system. In our case, this excludes the preamble, start of frame delimiter and frame check sequence.

This is a **common cause of confusion** so please take heed!

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.
- Print the source and destination MAC addresses.[5]

NOTE

While it's not essential to be able to parse the source and destination MAC addresses at this point, it's useful to do so to verify that your parsing logic is correct overall.

Parsing the IP Headers

The next protocol layer is the network layer, responsible for routing between hosts on the network.

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 will need this data later.
- Determine the length of the datagram payload.
- Determine the source and destination IP addresses, and verify that they match your expectations.
- Determine the transport protocol being used, and that all datagrams are using the same one.

NOTE

Remember, only one of the two hosts ever transmits image data. To rebuild the image, you will need a way to filter only the response packets.

Parsing the TCP 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. As a sanity check, ask yourself: How many ports are used? Which ports are they? Does this match your expectations?
- 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.

NOTE

Don't forget that packets can arrive out of order. If you want to build the image, you'll need a way to rectify the packet ordering.

Parsing 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 response. Extend your code to:

- Order the received data by TCP sequence number.
- Combine it into a single binary string.
- Extract the HTTP header, decode it as plain text, print and verify that they all make sense.
- Extract the HTTP body, write it to disk as a file with a `.jpg` extension, and open it!

NOTE

The transmitted image was free of any distortion or discoloration. If your result is skewed, discolored or otherwise distorted in some way, there may be a mistake in your parsing, filtering or ordering logic.

Spoilers

The Ethernet Header Format can be a little tricky to interpret. There are two forms: one at the physical layer and one at the data link layer. The physical layer form requires additional information in order to be able to delimit frames: a 7 octet preamble and 1 octet start of frame delimiter at the beginning, and then a 12 octet interpacket gap at the end.

The 802.1Q tag is optional in Ethernet, and not included in our packet capture. In some cases, the EtherType bytes can be used as a data length, but for this problem it is indeed an EtherType.

The top row of this diagram is what's in use:

802.3 Ethernet packet and frame structure

Layer	Preamble	Start of frame delimiter	MAC destination	MAC source	802.1Q tag (optional)	Ethertype (Ethernet II) or length (IEEE 802.3)	Payload	Frame check sequence (32-bit CRC)	Interpacket gap
	7 octets	1 octet	6 octets	6 octets	(4 octets)	2 octets	46-1500 octets	4 octets	12 octets
Layer 2 Ethernet frame			← 64–1522 octets →						
Layer 1 Ethernet packet & IPG	← 72–1530 octets →								← 12 octets →

The IP header has many fields, but the ones we need should be fairly straightforward to interpret. It is an IPv4 header; in order to determine its length, we must parse the Internet Header Length, which is the lowest order 4 bits of the first byte of the header. You can extract this by performing a logical and against the number 15 (`byte & 0xff` or `byte & 0b1111` in most languages). You must then multiply this by 4, to determine the header length in bytes, as it's specified in 32 bit words!

In our capture, ECN is not used (so should be 0) and the protocol value is always 6, indicating TCP.

Here is an image of the header format for convenience:

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																																

The TCP header format should also be fairly uniform. The source and destination port are the first two pairs of bytes, and the sequence number is the next 4. You will need the sequence number to reorder the segments correctly, and to filter out duplicates. Skipping the header requires parsing the data offset field, which is the high order 4 bits of the 12 byte. You can obtain this by right shifting by 4 bits. Again, it is specified in 32-bit words, so you will need to multiply by 4 to obtain a value in bytes.

TCP Header

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	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.) ...																															

Finally, as a huge spoiler: the image is of one of the main designers of the original versions of two of the protocols you've been parsing!

hello@bradfieldcs.com
576 Natoma St
San Francisco, California