

An application supporter's guide to TCP/IP packet capture and viewing cap files

By Magnus Karlsson

Contents

- [Summary](#)
 - [Objectives](#)
 - [Prerequisites](#)
- [Capturing a pcap file](#)
 - [Intro to Wireshark](#)
 - [Tapping a network interface with Wireshark](#)
 - [Capture filters](#)
 - [Snapshot length](#)
 - [File rotation](#)
- [What do you see in a packet capture](#)
 - [Filtering](#)
- [What do we find in TCP/IP packets](#)
 - [Internet protocol \(IP\)](#)
 - [Transmission Control Protocol \(TCP\)](#)
 - [TCP control messages that are good to know about](#)
 - [TCP Keep-Alive](#)
 - [Two useful things you can see from the TCP handshake in a pcap file](#)
 - [A final note](#)
- [Further resources](#)

Summary

This chapter covers how to use *pcap* (packet **capture**) files, why they are useful and a primer on how to read them. It describes how to capture traffic, how to interpret it, what to expect to see, what you can not see and what is not much of a point to look at for our intents and purposes in CSS.

Objectives

When finished, you will be able to understand when to use packet capture and the basic anatomy of the contents of a pcap file. This is not going to give you a complete understanding of what the traffic looks like but should rather be seen as a primer. This is a huge topic and some people build their whole careers around this so we are not going into too much detail here, but the aim is rather to give you enough of a base to stand on so that you can comfortably go into further depths of your own if you need to. (Or if you just love looking at packets!)

The main thing about pcap files when you're new to them is that they are *daunting*! The first question to ask oneself is "What *don't* I want to look at?" and filter that out. But to be able to answer that question you first need to have at least a rudimentary understanding of the protocols you are looking at. This chapter aims to bridge that gap and to do so from a CSS point of view, focussing on the parts that can be useful for application support troubleshooting.

Prerequisites

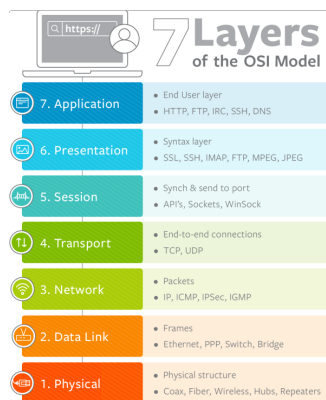


- Understand basic TCP/IP networking and routing
- Understand the OSI model
- Download and install Wireshark here: [Wireshark · Download](#)

Introduction

So you have this problem where users are complaining about things being slow, you've checked your bit about the product you are supporting, the server metrics are fine and the network admins tell you that everything is just fine with the network too. Now both sides are just pointing to each other. Then it could be a good idea to do a packet capture and have a look at the actual packets travelling between the users and the application to see if there's are clues there that can get the ball rolling again with the appropriate team.

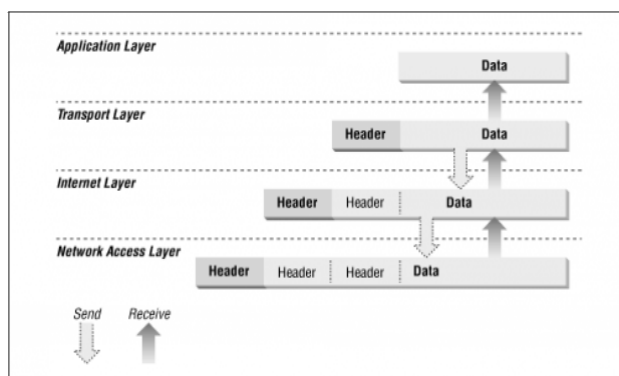
Why? Let's remind ourselves about the OSI model.



Layers 1 – 3 are the clear domain of the network people and layers 5 – 7 are the clear domain of the application/server people. Layer 4? Well, that's often a bit of a gray area.

Packet captures are also useful for examining security problems or just to see what some software actually does over the network when there is inadequate documentation or unexpected behaviour.

In support we are troubleshooting *applications* and its data will be delivered over the network in datagrams wrapped in a TCP/IP header pair which make up the packets that wizz around the networks to deliver data to and from the applications and clients. Schematically this transmitted data looks like this:




When looking at captured traffic, it's good to keep in mind that each layer is unaware of the data structures used by the layers above and below it. The TCP and IP headers of TCP/IP will be read and interpreted by different parts of the network transmission chain. Until a packet has reached its destination network only the IP header will be read and the rest will just be passed on (generally speaking). Once the packet has reached its destination, the mechanism that then reads the packet to extract the data from it will not care about what is in the IP layer since it is not relevant anymore.

As part of your job you've checked application/server part (application, presentation, session layers) and the network admins have determined that their part is good too (network, data link, physical

layers). What remains is the *transport* layer where TCP and IP lives and to investigate that we need to capture the packets travelling across the network into a *pcap* (Packet CAPture) file that we can examine.

There are different ways of doing this, like with for example the ubiquitous `'tcpdump'` command in a shell, but in this chapter we will be looking at using **Wireshark** as it is a commonly available, powerful and widely used GUI tool for capturing and analyzing network packets.

 For practical purposes the phrase "on the wire" will be used in this chapter when talking about the actual physical transmission going over the Physical/Data Link layers, including when transmitted over wireless links.

Capturing a pcap file

There are many ways to capture packets from the network. It can be done from a client computer, a server, a piece of networking equipment like a router or a switch or from a dedicated piece of hardware (a "tap").

Why?


It's the most complete look at network traffic you can get and you can save it on disk for further analysis. It's also hardware agnostic; a packet capture will look the same if captured on a laptop or a network appliance.

Why not?

The drawback is that since you are capturing *all* traffic the files can become very large quickly and the amount of information can be overwhelming. For application troubleshooting it is a very blunt instrument and should only be used in special cases when application layer debug tools can not be applied for some reason. Trawling through a pcap file to find out if it's the application that's slow or the network is overkill and very inefficient compared to saving a HAR file in a browser.

Note that on most local networks only traffic where your network interface is the intended recipient will be seen by it since the upstream router will only send you those. It will not be possible to eavesdrop on your neighbours. (To do that you need to be higher upstream in the network chain.)

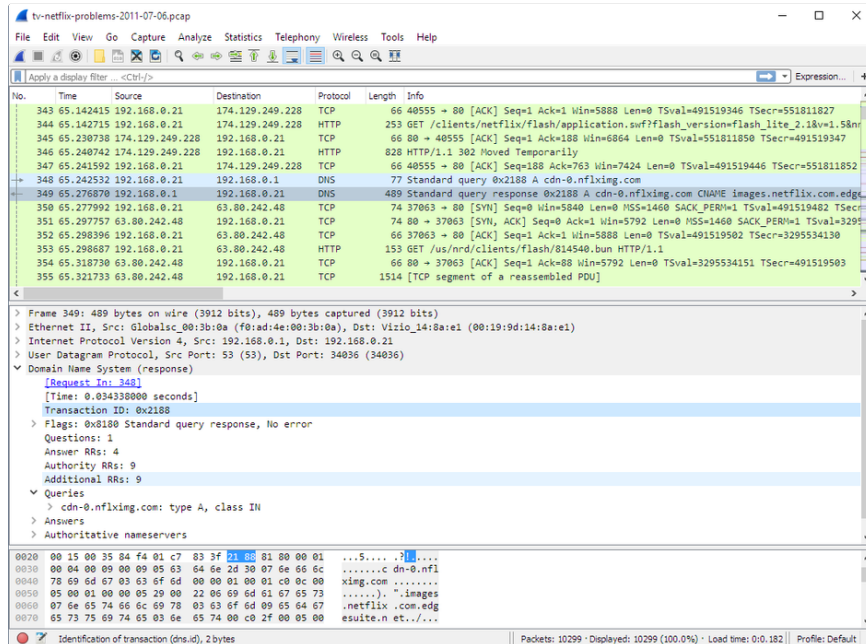
It is also important to keep in mind that encrypted traffic, like TLS, will not get decrypted. We are tapping the network interface for this data and the interface does not know how to decode it nor does it have the keys to do so. If you want to view HTTPS traffic payload, you should use a browsers developer tools and capture the HTTPS traffic in a HAR file instead.

 If you get a pcap file from a customer, it is important to remember to ask them *where* exactly on the network it was captured as well as if they used any filters when capturing it.

Intro to Wireshark

A very common application to capture and view pcap files is Wireshark. It allows you to see the packet stream as well as the structure and content of the packets in one configurable screen. It also provides powerful filtering and packet inspection features as well as being highly customizable. It is open source and you can download your own copy of it here: [Wireshark · Download](#) (There are others, but for this chapter we will be using Wireshark to capture and view pcap files.)

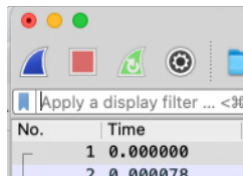
A Wireshark view of captured packets waiting to be examined:



Tapping a network interface with Wireshark

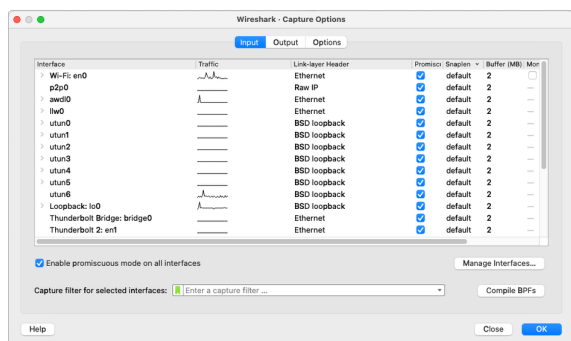
Since all network traffic seen by that interface will be written to file, pcap files can become very large. How large simply depends on how much traffic the capturing interface sees and for how long the capture is going on. If your pcap files are hundreds of MBs or even GBs, then working with them is going to be slow and cumbersome. Since most of the time we are not interested in examining *all* network traffic but only a portion of it, we can use capture filters to keep the file size down. Wireshark also allows you to set a file size limit and number of files to save before overwriting the first one, so that you can capture traffic continuously for some time and still have manageable file sizes. This is particularly useful when troubleshooting a problem that happens intermittently and unpredictably.

There are four buttons at the top left related to capture



From left to right they are **start capture**, **stop capture**, **restart current capture** and **capture options**.

First we need to select which interface we want to capture on by clicking *capture options*. Under *Input* you will see a list of available network interfaces. These are the ones your computer has, but not necessarily uses. You can locate your main interface by looking at the traffic graph next to the name. Here "Wi-Fi:en0" is an Ethernet interface with traffic on it, so that's probably the one we want to capture on. (You can remove interfaces you never use from this list with the button **Manage Interfaces...**)



Then we press **OK** and click the blue fin button to start the capture. You will see packets appearing on the screen more or less immediately. If you stop the capture for a while and then start it again, it will continue to add to the already existing packets captured. To start afresh you need to click the green **restart current capture** fin button instead.

Capture filters

The **Capture Options** dialog in the section above also has **Capture filter for selected interfaces**. If we are only interested in a specific type of traffic, we can specify a capture filter here to keep the pcap file size down. For example, if we are only interested in TCP packets we can specify that here and only TCP packets will be captured while discarding everything else (such as UDP, network control packets, etc.). The filter syntax is the same as for when we filter an already captured pcap file, so we will get to more details and some examples in a later section.

The drawback with capture filters is that you might miss something that you did not anticipate.

If in doubt, just grab everything and filter the file later.

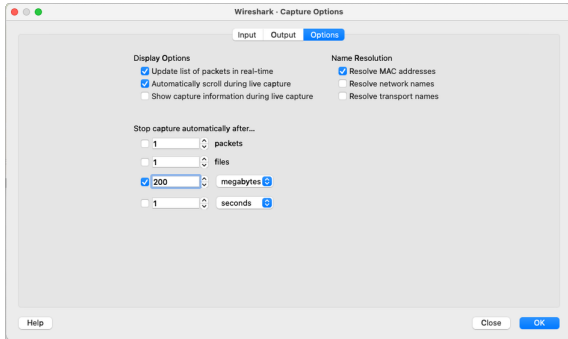
Snapshot length

One of the columns in the Input interface list is **Snaplen**. In the dialog above the value is "default", but if you click on it you can set a **length in bytes per packet**, i.e. once the set number of bytes have been captured, no more bytes will be saved for that particular packet. This can be very useful

to keep the pcap file size down. For example, if we are only interested in the TCP headers but not the HTTP payload the packet contains, why save all that payload? (which is often going to be encrypted anyway) How much to set it to depends on which protocol you are aiming for, but for example for a TCP packet over IPv4 you'll need the maximum possible IPv4 + TCP bytes, i.e. 60 + 60 = 120 bytes.

File rotation

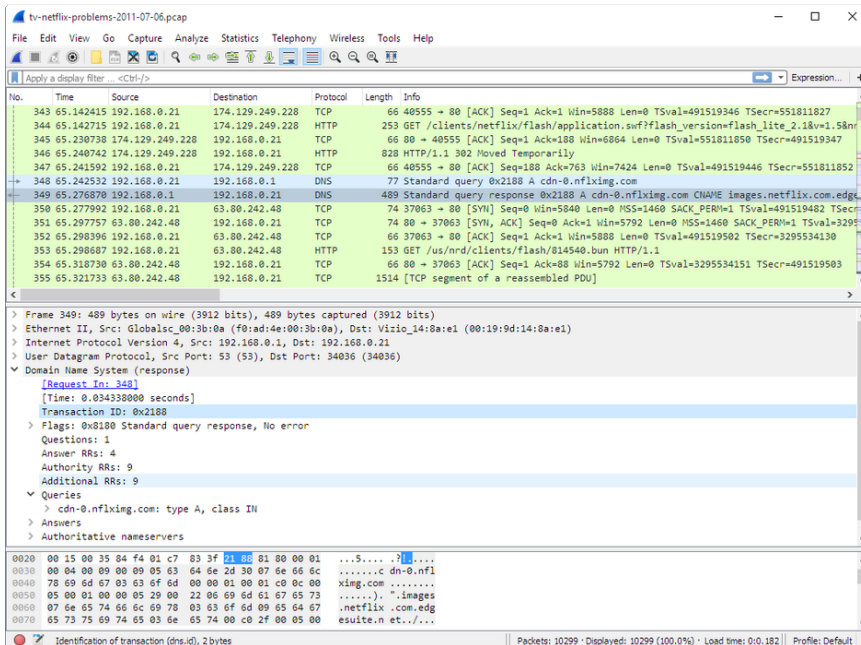
Sometimes it is useful to capture traffic for a longer period of time, like when troubleshooting an intermittent and unpredictable problem. To avoid that those captures become absolutely huge, you can set a file ring buffer under the **Output** tab in the **Capture Options** dialog. Then Wireshark will capture to a file until it's reached its specified maximum and then it will create a new file and write to that until it too has reached its maximum, and so on. The default number of files is 2, but you can set it to anything you want. In the screenshot below we have set it to start a new file after 100 MB and keep 10 files. When it has written 10 files, it will go ahead and overwrite the first one and we can be sure that our total capture never exceeds 1 GB.



What do you see in a packet capture?

As the application support role is about troubleshooting software issues, we will only be focusing on TCP packets here. But a pcap file usually contains *all* traffic on that network interface which will include a lot of other kinds of packets used for network control, asset discovery and other purposes (like UDP, ARP, ICMP, etc...). For this reason it's often a good idea to filter all the other traffic out when looking at the pcap file. More on this further down.

This is the main window of Wireshark.



It's made up by a packet list on top, expandable details about the selected packet in the middle and the packet bytes at the bottom.

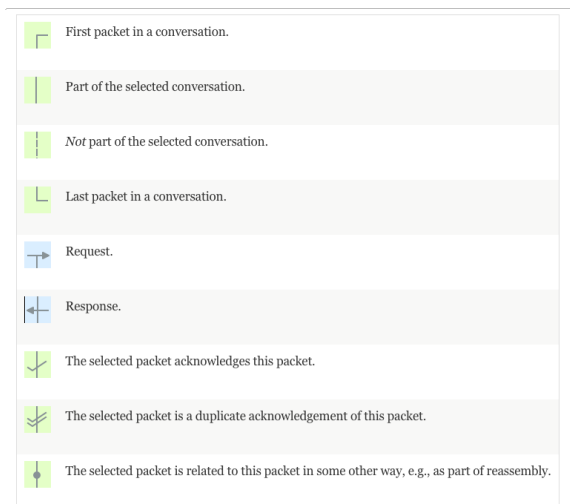
✓ Depending on your preferences and your screen you may want to change this layout to one of liking. For example, I like to have the two lower panes next to each other horizontally instead. But it's a matter of personal preference and can be set by going to Preferences > Appearance > Layout. (Also note the "packet diagram" option selectable for one of the frames showing you a reference of the anatomy of the selected packet and details for reference, which can be handy sometimes.) You can also change the colour markup of different packets and a lot of other things, which might be useful for your particular purposes when you're comfortable with the basics. It's all described in the Wireshark User Manual, so we are going going in to that more here. [3.3. The Main window](#)

✓ One very useful column that is not in the default set is *"Delta time displayed"*. This will give you the time difference compared to the previous packet, which makes packets taking unusually long time much more easier to spot than if you just look at the Time column.

The **packet list** will show you the packets captured with a list of basic details about them, such as relative time, source, destination and so on. The list can be customized by right-clicking on the column description bar and selecting "*Column preferences...*" to add and remove columns. By right-clicking you also get the useful "*Resize to Contents*" which automatically sorts out your column width for you (also available in the button bar).

Apart from a list these details, the packet list pane also conveys information about the packets with colors and symbols. What the colors mean can be found in the menu under **View > Coloring Rules** where you can also add coloring rules of your own, if you wish.

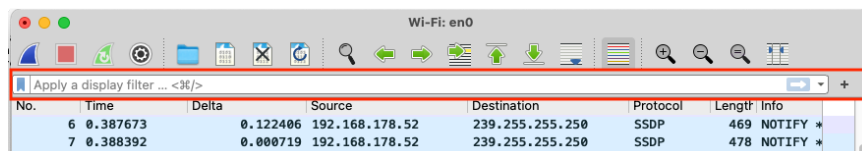
To the left, just before the packet number, you can in the screen above also see a dotted line and two arrows. This is to help you to keep track of a packet sequence. When a packet is selected, all packets in the same sequence will have a bracket around them here. This is what those symbols mean.



We will return to the packet details and packet bytes panes further down when getting into the TCP/IP packets.

Filtering

Just between the packet list pane and the button bar, you will find one of the absolutely most powerful features of Wireshark: filters!



As a pcap file will contain *all* traffic visible to that network interface (unless a capture filter was specified at the time of the capture), it will be a lot of packets in there that we just don't care about as application troubleshooter folks. Who cares about those asset discovery protocol packets and other irrelevant stuff that are zooming around on that wire, right? We want to go straight to only the packets we are interested in — which will generally be TCP packets carrying HTTP protocol payloads with our application stuff in them.

The filter bar will be red when the syntax is not right and will turn green when it's good! Also note the **plus sign** at the right of the filter bar. By pressing that button you can save the current filter as a new button on the filter bar so you don't have to type it again each and every time!


Filters consist of a keyword which may or may not contain one or more sub-keywords (for example, there is the "http" filter which will show you all packets with HTTP protocol, but there are also sub-keywords under http, such as "http.request" or "http.response.code") and keywords can be combined or compared with operator such as "or" and "eq" and "contains", etc.

We are not going to get deep into the filters here since Wireshark filtering capabilities are massive. There are literally many thousands of filter options that you can combine in a myriad of ways. But below you find some examples of common ones that can be useful when doing basic support analysis, which should be good ones to get you started and get you used to how the filter syntax works.

ip.addr	tcp	tcp.port
ip.src	http	http.request
ip.dst	arp	http.response.code
	dns	

Examples:

ip.addr == 10.0.0.123	show only packets involving this IP address
tcp.port == 443	show only TCP packets involving port 443
http.response.code == 200	show only HTTP response packets that had the response code 200 (OK)
tcp contains puppies	show only TCP packets that contain the word "puppies"
dns or http	show only packets that contain either the DNS or HTTP protocol

 **Pitfall:** Note that *and* is an inclusive keyword. If you for example want to see only DNS and HTTP packets, then you have to write it as "dns or http" and not use "and". If you use "and" the filter will only match if a packet contains *both* the protocols DNS and HTTP in it. (which is impossible and will never match anything)

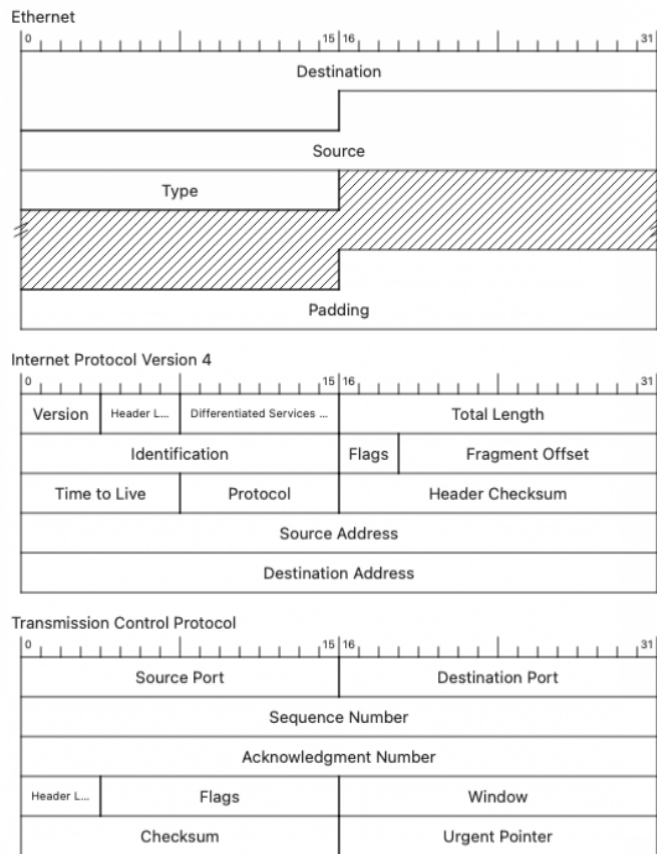
You can also filter *out* traffic by negating the filter with the "!" operator, for example "!(arp or dns or icmp)" to filter out those three protocols and keep the rest.

For a more extensive list of the possible operators and more examples, see the Wireshark "Building Display Filter Expressions" documentation page [6.4. Building Display Filter Expressions](#)

See [Wireshark · Display Filter Reference: Index](#) for a list of available filters and sub-keywords

What do we find in TCP/IP packets

It's been covered in earlier chapters, but since understanding what packets look like and what to expect is key to analyzing pcap files, let's remind ourselves of the structure of the TCP/IP protocol headers and what they are used for. Any TCP/IP packet you will be looking at will be in the form of this structure:



(The scale on top are the number of bytes of each field)

In the Wireshark *Packet Details* pane (lower left as default) you can see this an expandable list like this example:

```
> Frame 359: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface
en0, id 0
> Ethernet II, Src: XXXXXX:XX:XX (78:4f:43:XX:XX:XX), Dst: XXXXX:XX:XX
(e8:df:70:XX:XX:XX)
> Internet Protocol Version 4, Src: 192.168.178.17, Dst: 192.168.176.12
> Transmission Control Protocol, Src Port: 60047, Dst Port: 443, Seq: 518, Ack:
5502, Len: 0
```

The last bytes of the MAC address are replaced with X's to protect the innocent. 🙄

In application support we do not really care about the **Frame** and **Ethernet** layers since our job is to troubleshoot our application. Anything in these will only be meaningful if you know that particular local network. It will only be meaningful to the local network admins and examining them will not help you in your quest.

- **Frame** is a data unit transmitted over the actual physical link. For our purposes we can just ignore this part.
- The **Ethernet** layer, or Network Access Layer, describes how to deliver the data between the actual physical network devices. It will mainly contain the MAC (Media Access Control) addresses of the source and destination addresses. These are the unique 48-bit addresses that

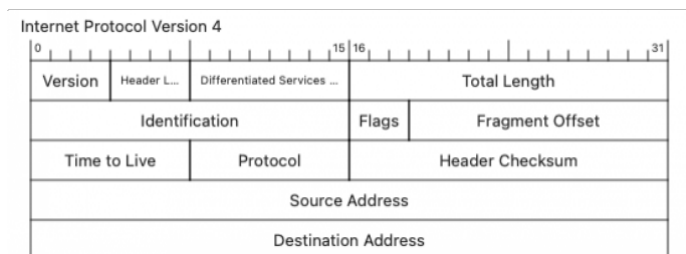
every physical network device have and is used for local routing. (For example, each computer on a local network will advertise its MAC address on the network (see ARP/RARP https://en.wikipedia.org/wiki/Address_Resolution_Protocol) and router on that network will then know from this over which "wire" it should route the packet with the IP address that belongs that particular physical network interface (MAC address). Great fun for network engineers, a waste of time for application support.

Internet protocol (IP)

The Internet Protocol describes where a packet is from, where it is going and what kind of packet it is. It is used to route a packet correctly to its destination.

In CSS we will not have a reason to look at IP much as most of the action we are interested in happens in the Transport layer (TCP) and higher, but it's good to know what it does and a few key headers, since you will come across IP headers in all the packets you have captured and at least the source/destination addresses and Time to Live may be useful for our purposes.

Below is the structure of an IP header.



- **Time to live (TTL):** This defines the maximum number of *hops* this packet can make before it should be just dropped. These are the same kind of hops you would see when running the `'traceroute'` command and we don't want misrouted packets to continue to roam the internet for the rest of time. With every hop, that router decreases this number by 1. When a router gets a packet with TTL=0, it will just drop it. This number can be anything and is determined by the driver in the OS, but 128 is a pretty common starting point. This can be for example useful to determine if the packet we are looking at was captured on the client or server end, if we don't know. If you, for example, see a packet sent from a client that sets TTL to 128 and the TTL in the packet you are looking it is 128, then you can know that this was captured on the client, to mention one case where this can be useful.
- **Protocol:** Defines which protocol to expect, TCP, UDP, etc.
- **Source address**
- **Destination address**

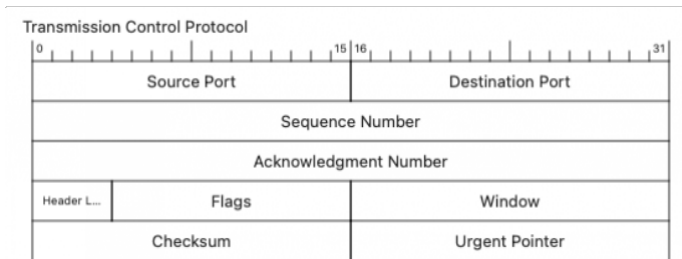
Note: Keep in mind that the source and destination addresses are in relation to the point on the network the packets were captured.

We are leaving out IPv6 of this for now to not make this primer more complicated than it has to be. If you want to check out the IPv6 datagram headers – there are more than one – you can find them in the IPv6 RFC <https://datatracker.ietf.org/doc/html/rfc2460>

Transmission Control Protocol (TCP)

TCP is the Transmission Control Protocol that envelops the data transmitted over the link. It ensures that the receiver of the data get it all exactly as it was sent and in the correct sequence.

The TCP packet looks like this:



In Wireshark, all calculated values are inside brackets. These are values that are not in the packet data itself but are often very handy to have readily available, like the TCP segment length and next sequence number in this example. Thanks, Wireshark!

We are just going to focus on a few key fields here. If you want to know what all of them are, they are described in their respective RFCs. (See *Further resources* below)

- **Source port, Destination port** are the ports agreed on during the TCP handshake process.
- **Sequence Number** This field is a fundamental part of TCP and is used to keep track of the amount of data transmitted and received, to ensure they are assembled in the right order at the recipient end and to detect if a packet is a duplicate (in case it has been retransmitted). Every octet (byte) of data sent over a TCP sequence has a 32-bit sequence number. For example, the sequence number for a packet is X and the length for this packet is Y octets. If this packet is transmitted to another side successfully, then the sequence number for the next packet is X+Y. Since every octet is sequenced, each of them can also be acknowledged. The acknowledgment is cumulative so that an acknowledgment of sequence number X indicates that all octets up to but not including X have been received. The starting point for any sequence is effectively a random number, i.e. not 0 or 1 as you might expect. The important thing is that both sides have agreed on the starting point. But this makes the raw sequence numbers hard to follow for humans, but fortunately Wireshark and other tools calculate a more friendly *relative* number for us which starts at 0 for each sequence, like in this example from the Wireshark packet details pane:

◦

```
1 [TCP Segment Len: 1384]
2 Sequence Number: 1 (relative sequence number)
3 Sequence Number (raw): 593855529
4 [Next Sequence Number: 1385 (relative sequence number)]
```

In this example the actual 32-bit sequence number is 593855529, but the calculated relative sequence number is 1. The next relative sequence number is going to be that plus the TCP segment length for this packet of 1384, i.e. $1 + 1384 = 1385$.

- **Acknowledgement Number** contains the value of the next sequence number the sender of the segment is expecting to receive.
- **Window** is the number of bytes the sender of this segment is willing to accept back. Sneakily, for historical reasons this number is often actually a lot less than the *actual* window size the sender is willing to accept, see the *Window Scaling* option below.

- Options is where different modifiers for TCP are defined. There are a number of them, but the most interesting ones for us are *Window Scaling* and *SACK* Below is an example from the Wireshark packet details pane.

```

1 Options: (12 bytes), Maximum segment size, No-Operation (NOP), Window scal
2   TCP Option - Maximum segment size: 1460 bytes
3   TCP Option - No-Operation (NOP)
4   TCP Option - Window scale: 8 (multiply by 256)
5   TCP Option - No-Operation (NOP)
6   TCP Option - No-Operation (NOP)
7   TCP Option - SACK permitted

```

- Maximum Segment Size (MSS)** tells about the packet size which the client can accept from the server. (Note: that's the full segment, not just the window size which is part of the segment)
- Window scaling** In TCP the Window field is only 16 bits long and can therefore only represent a maximum value of $2^{16}=65536$ bytes. This was probably enough with room to spare back in 1981 when TCP was defined and memory was hugely more expensive than today. These days memory is cheap and that maximum 64 KB window is rather small these days, so it is common for the two sides to agree on a *multiplier* for the window size, which is called the window scaling factor. It can be 0-14, which is the number of bits to add to the window field of 16 bits. So the maximum you can get is $16 + 14 = 30$, i.e. 2^{30} bits = 1 GB.* In the above example the scaling factor is 8, which Wireshark has inside parentheses kindly clarified means that we should multiply the window size by 256 to get the actual permitted size.
- SACK** is a commonly used extension to the TCP protocol and means Selective ACKnowledgement. It is a mechanism to reduce traffic by telling the sender about lost packets so that the sender only has to re-transmit those. Without SACK *all* packages of that sequence will have to be resent if not all the packages got received.
- No-operation (NOP)** These can simply be ignored. By the TCP protocol definition these options have to be multiples of 8 bits in length and if the length of the option is less than that it will have to be padded out with bytes that are just "01", which is defined to just mean NOP.
- Payload** is the actual data that we wanted to send across the network that required us to use TCP/IP in the first place. For example if you are requesting/sending HTTP then the HTTP protocol headers and data will go here.

* So why do we even bother defining this and not just go with 1 GB max window for all requests? Well, it has a simple explanation: real world limitations. Every active TCP connection on a machine uses a TCP *socket* on it, i.e. a certain amount of resources get reserved for it and the number of available sockets are limited since machine resources are not unlimited. If every TCP connection reserves 1 GB of memory each, that machine will not be able to handle many simultaneously active TCP connections.

TCP control messages that are good to know about

TCP control messages are by default marked as red with black background in Wireshark, so you can end up seeing something like this which can look a bit scary.

100	5.222811	0.000000	192.168.1.78.1	192.168.1.78.47	HTTP	60	4020 -> 80 [SYN, SEQ=54250, WINDOW=0, LEN=0, MSS=1460, TSV=1548122884, TSRC=288113750, SRE=5]
108	5.281840	0.000000	192.168.1.78.27	192.168.1.78.23	TCP	70	57812 -> 54250 [PSH, ACK, Seq=1, Ack=268, Len=0, TSV=1548122884, TSRC=288113750]
109	6.391849	0.000000	192.168.1.78.27	192.168.1.78.23	TCP	70	[TCP Retransmission] 57812 -> 54250 [PSH, ACK, Seq=1, Ack=268, Len=0, TSV=1548122884, TSRC=288113750]
110	5.500020	0.000000	192.168.1.78.27	192.168.1.78.23	TCP	70	[TCP Retransmission] 57812 -> 54250 [PSH, ACK, Seq=1, Ack=268, Len=0, TSV=1548122884, TSRC=288113750]
111	6.841891	0.000000	192.168.1.78.23	192.168.1.78.27	TCP	60	54250 -> 57812 [ACK, Seq=5, Win=4096, Len=0, TSV=1548122884, TSRC=288128195]
113	6.827734	0.000000	192.168.1.78.23	192.168.1.78.27	TCP	70	[TCP Dup ACK 111#1] 54250 -> 57812 [ACK, Seq=5, Win=4096, Len=0, TSV=1548122884, TSRC=288128195, SRE=5]
115	6.807737	0.000000	192.168.1.78.23	192.168.1.78.27	TCP	70	[TCP Dup ACK 111#1] 54250 -> 57812 [ACK, Seq=5, Win=4096, Len=0, TSV=1548122884, TSRC=288128195, SRE=5]
116	6.113675	0.000000	192.168.1.78.1	192.168.1.78.27	HTTP	60	[TCP Retransmission] [TCP Port numbers rounded] 42820 -> 80 [SYN, Seq=0, Win=26208, Len=0, MSS=1460, SACK_PERM=1, TSV=15111]

These are sometimes misinterpreted as something being broken. This is not (necessarily) the case. It's just the TCP control mechanism doing its job to ensure that missed packages are resent and that the integrity of TCP connection is maintained. Fewer is better but especially wireless networks can be a bit noisy. If there are lots of control messages like that then it would be a good idea to ask the local network admins for that particular network to look into it as that indicates some network trouble. But just one here and there is by itself no reason for alarm.

TCP Keep-Alive

Another one of those red on black packets are TCP **Keep-Alives**, which requires some explanation as it may be something we (CSS) have to investigate.

If you see them a lot you *might* have an application performance issue. Every active TCP connection on a machine uses a TCP *socket* on it, i.e. a certain amount of resources are reserved for it and the number of sockets are limited (since machine resources are not unlimited). For example, every connection has a memory receive buffer equal to the TCP window size described in the previous section. Therefore it does not want to keep a connection open for longer than it has to. But sometimes the application just takes some time to process the request. Then the client machine may send a TCP Keep-Alive packet to ask the server "Hey, are you still there or can I just drop this connection?". If all is well, the server machine responds with an ACK saying "I'm here! Just have a lot of work to do for you. Hang in there and don't drop the connection." Typically TCP Keep-Alives are sent every 45 or 60 seconds on an idle TCP connection, and the connection is dropped after 3 sequential ACKs are missed. But this is defined on operating system level and can be pretty much anything. These are normal to see sometimes but if you see them consistently for a specific action for example, then it would be a good idea to check out why the application is taking so long to respond. Then other tools like HAR files and application performance logs will be of more help than a TCP dump.

Two useful things you can see from the TCP handshake in a pcap file

From previous chapters we know that the [TCP handshake](#) consists of the three first packets exchanged, SYN, SYN/ACK and ACK. Since the first SYN packet does not contain any payload data, it is only going to be handled in almost no time by the receiving network interface. That means that the delta time between the SYN and SYN/ACK packets is practically the network roundtrip time between the two.

As described above, the TTL field in the IP header is the max number of network hops the packet can go before it should just be dropped. So if we know the sender's starting point (commonly 128, but it is not safe to assume this) we can deduct the packet's TTL as we received it from the starting point and we then know how many network hops away from each other the two interfaces are.

A final note

It is important to know *where* a pcap file has been captured to be able to make sense of it. Is it on the client or the server side? Is it on a network device somewhere?

We can also ever only see the packets as they were seen by that particular network interface. Sometimes a network device on the way may have tampered with the TCP/IP headers, like for example changing the window size, TTL or something else. We can never be 100% sure from just one pcap file that the packet we are looking at will be received by the other end in the same shape. If there is a suspicion that something on the way is tampering with the packets, the only way to know is to capture packets on both the client and server end at the same time and compare the packets.

Further resources

RFCs:

Internet Protocol v4: [RFC 791: Internet Protocol](#)

Internet Protocol v6: [RFC 2460: Internet Protocol, Version 6 \(IPv6\) Specification](#)

Transmission Control Protocol: [RFC 793: Transmission Control Protocol](#)

Author: Magnus Karlsson (2022)