Justin Strelka

Wireshark\_report

6/18/20

**Home Network Packet Capture: Wireshark**

**Abstract:**

This paper begins with a brief description of the packet capturing tool called Wireshark followed by some terminology descriptions. The Wireshark description includes the usefulness of the tool and instructions for installation on a Windows machine. Next the paper outlines the essential features of Wireshark along with a walk-through on how to apply Wireshark’s essential features to enable a real-time packet capture session. Lastly, the paper concludes with an analysis of packets captured in a session that identify the sequential procedures used by various protocols to set-up and tear-down connections.

**Motivation:**

This paper is intended for individuals that are new to Wireshark and are seeking a real-life example of the tools use. Upon reading this paper you will be explain what Wireshark is used for and why it’s useful for understanding communications within a network. You will also be able to apply Wireshark to your own network and conduct a real-time packet capturing session. From a captured stream of packets, you will understand how to apply filters in Wireshark to display specific packets that refer to various communication events. You will be able to understand the information provided by Wireshark and formulate an accurate hypothesis of why a communication event occurred.

**Introduction:**

Network packets contain all the information passed over a network and can help us understand how a network functions along with troubleshooting network errors. Wireshark is a network packet capturing utility that subdivides the data contained within a packet into a human readable organized GUI. Packets contain large amounts of data so being able to visualize that data in a GUI will save us time while troubleshooting and securing our networks.

**Method/Measurement:**

Before we begin using the Wireshark utility to capture and analyze real-time packet streams, we must first understand some basic vocabulary. A packet is the carrier that encapsulates the data flowing over a network from the origin to the receiver. Large chunks of data are first broken down into multiple small packets that are easy and quick to transmit over a network. The process of dividing large messages into smaller pieces is called fragmenting. At the receiving end of the transmission packets are reassembled to form the original message prior to fragmenting. [1]

Packets consist of three main sections that serve various purposes. The first segment of a packet is the Header. Headers contain information that is used for transmitting a packet between the origin and the receiver; information such as the source address, destination address, protocol, and sequential number is contained within the header. During transmission not all packets are received in the order they were transmitted. The sequential number of a packet is what the receiver uses during message reassembly and to ensure packets were not dropped during transmission over TCP protocol. The next segment of a packet is called the Payload or data section. The Payload is the fragmented data being transmitted over the network. The data in the Payload is anything you can send across a network such as a python file or possibly a video stream. Live video streams are generally sent over UDP since we are unconcerned with a few dropped packets during a live stream, however dropped packets in a python file can have a catastrophic effect on the integrity of the reassembled data. The final segment of a packet is the Trailer which the contents of can vary greatly by network. However, the standard use of a packet Trailer is to notify the receiver that it has reached the end of the packet. [1]

Most packets are created using the framework outlined by the OSI model. The logical OSI model is composed of 7-layers: Physical, Data Link, Network, Transport, Session, Presentation, and Application layers respectively. For the purpose of this paper we are going to discuss five of the OSI model layers. Beginning with the Physical layer which refers to the actual hardware used to communicate bits across the network.

Above the Physical Layer is the Data Link layer which is responsible for physical addressing or the MAC address of the sender and/or receiver contained within the packet header. The Data Link layer also has some built in mechanisms for error control which can detect damaged or lost packets. Lastly, the Data Link layer can play a role in fragmenting the Payload into frames of a predefined maximum size. [2]

Next is the Network layer which is responsible for routing packets to destinations located in different networks. The Network layer provides the packet header with the logical addressing of the sender and receiver. Then the network layer selects the shortest available path to transmit the packet over. The Network layer is also responsible for fragmenting the payload to be sized within the limits established by the Network Interface Card (NIC). [2]

On top of the Network layer is the Transport layer which provides information used by the Application layer. The port numbers an application runs on for both the sender and receiver are added to the packet header by the Transport Layer from the sending side of the transmission. On the receiving side of the transmission the Transport layer reads the port number and directs incoming packets to the correct application listening on the specified port. The Transport layer is responsible for reassembling packets on the receiving end and acknowledging the successful delivery of the message. [2] If packets are missing over TCP protocol the Transport layer will re-transmit any lost packets.

The final layer we are concerned with for the purposes of this paper is the Application layer. The Application layer sits at the very top of the OSI model and on the sending side of a transmission is responsible for generating the data to be transferred. On the receiving end the Application layer is responsible for displaying the transmitted data to the user through applications such as browsers, skype, teams, zoom, etc.

Now that we understand what network packets are and the role each of the five discussed OSI layers play in generating them, we can begin to analyze packet streams using Wireshark. Wireshark can be downloaded from the following link <https://www.wireshark.org/download.html>. For the purposes of this paper I will be running Wireshark on a Windows 10 machine. During the installation you will want to allow NPCAP to be installed because NPCAP is required for Windows machines to capture packets on your local network. Once the install has finished you will want to run Wireshark in administrator mode to bypass all the permissions issues necessary to capture packets on a Windows machine.

Once Wireshark is running you will be able to see a time lapse of the activity taking place on the various network interfaces. From this page you will be able to see which interfaces are currently transferring packets across the network. If you begin a music streaming session or voice call you should see an increase in activity on the interface where the session is taking place. To begin a packet capturing session in Wireshark we must first highlight the interface in which we want to monitor. Once an interface has been selected, we can apply a filter to remove excess network chatter. After the desired filters have been applied click the blue shark fin icon in the upper left corner of the window to enable a packet capturing session.

Once a session has been enabled Wireshark will begin capturing all packets traveling across the selected interface. For the purposes of this paper I have chosen to use the WiFi interface to scan for communications being established and broken down. The Wireshark GUI is broken into three sections with the top portion of the screen displaying the actual packets being captured across the selected interface. This section will provide us with information such as the absolute time passed since the capturing session was enabled, the source IP address, the destination IP address, transport protocol used, length of the packet, and some descriptive info such as the port number on both sides of the transmission. The next section displays information from a single packet separated into the five discussed layers of the OSI model. In this section you can find the packet sequence number, ports used, protocols, frame size and more. The last section displays the data contained within the packet or the actual Payload.

I began my search using the TCP flag filters in Wireshark to narrow down my results. TCP uses a three-way handshake to establish communication between the client and the host. First the client sends a request to establish a connection along with a synchronized packet number. When the host receives the packet, it will store the client packet synchronize number and send an acknowledgement transmission back along with its own packet synchronize number. When the client receives the response from the host it will store the hosts packet synchronize number and reply with an acknowledgement packet.

TCP packets include flag bits that can define packets as synchronizing requests and/or acknowledgement replies. By using the TCP flag filters for syn and ack I was able to reduce the number of packets to less than 2% of the entire capture. As you can see in **Figure 1** the filter used was “tcp.flags.syn == 1 and tcp.flags.ack == 1”. This allowed me to see all the TCP connections that had initiated during the capture session.

After displaying all the connections, I ordered the TCP packets by their IPv4 source address in descending order. I achieved this by clicking the source tab in the Wireshark GUI. By clicking on any of the tabs in the Wireshark GUI users can order packets in ascending or descending order relative to a desired field.

A screenshot of a social media post

Description automatically generated

**Figure 1:** Initial tcp syn/ack flag filters

After all the packets were in descending order, I was able to deduce that there were 15 unique source addresses and 52 TCP connections initialized during the session. I found it interesting that within a 90 second period my local machine initialized 13 TCP connections with a single IP address. The IP address that my local machine initialized 13 TCP connections was 13.107.42.12. I then used a command line tool called “nslookup” to do a reverse lookup on the aforementioned IP address. As seen in **Figure 2** the reverse lookup returned that the IP address belonged to a Microsoft OneDrive server.

A screenshot of a cell phone

Description automatically generated

**Figure 2:** Reverse IP lookup using nslookup returning 1drv.ms

Next I altered the filter as follows: “ip.dst == 13.107.42.12 || ip.src 13.107.42.12”. By altering the filter, I was able to display only packets that had 13.107.42.12 as the source or destination address. As expected, the very first packet had my local IP address as the source IP, the OneDrive server was the destination IP and the TCP SYN flag was set. This showed that my local machine sent a TCP communication request to the OneDrive server. There were multiple TCP connections being displayed in the packets since my filter was showing me all the packets that referenced the OneDrive server.

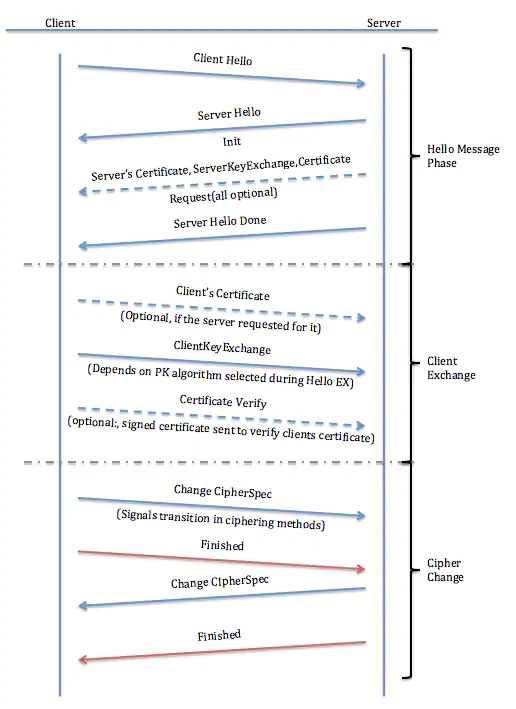
Wireshark assigns each new TCP connection within a capture session a unique identifier.[3] The TCP stream identifier can be found in the Transport layer drop down in the OSI section of the Wireshark GUI. Look for the field named “Stream Index” under the subcategory Transmission Control Protocol (TCP). This unique identifier is set on all the packets that belong to a single TCP stream. To display only the packets that belong to a single TCP connection I adjusted the filter as follows: “tcp.stream == 7”. As you can see in **Figure 3** the entire TCP stream can be viewed, starting at packet number 79 the original SYN request sent by my local machine to the OneDrive server and ending at packet number 129 the last packet sent acknowledging the closing of the stream.

A screenshot of a computer

Description automatically generated

**Figure 3:** Display of the tcp.stream == 7 filter showing a single TCP stream.

As mentioned previously the first three packets in the sequence (79,81,82) show the three-way TCP handshake being executed. Packet number 84 is interesting because it initiates the use of encryption to for transmitting data. My local machine sent the initial Transport Layer Security handshake “Client Hello” packet to the OneDrive server. After the “Client Hello”, packet numbers 87-95 were sent back from the OneDrive server using TCP acknowledging the request to establish a secure connection using TLS. Packet number 96 is the “Server Hello” packet in the TLS handshake and the Payload of the packet contains: the OneDrive server’s certificate, the server’s certificate status, and the server’s public key. Packet 99 is my local machine acknowledging the TLS exchange and the payload of packet 101 contains my local machines public key and the request to change the Cipher specs. The Change Cipher spec protocol is used to create a new set of keys from the shared information between the client and the host that will be used for data encryption and decryption.[4] After the request for the Change of Cipher the OneDrive server replies with a TCP acknowledgement and then sends a “New session ticket” in packet number 110 with the changed cipher specs confirming the initiation of TLS. Next my local machine then sent two packets (112-113) over TLS protocol which most likely contained data from my local OneDrive folder that would be used to ensure the remote OneDrive server is synchronized with my local machine. The OneDrive server then sent an acknowledgement packet back followed by two packets (121-122) over TLS that most likely contained data used to ensure my local OneDrive folder is synchronized with the remote OneDrive server. After both my local OneDrive folder and the remote OneDrive server synchronized with each other the breakdown of the TCP stream began. My local machine sent a packet (124) with the ACK and FIN flags set initiating the end of the stream. Finally, the OneDrive server sent back its own packet (128) with the ACK and FIN flags set to end the stream. **Figure 4** provided by cisco outlines the packet transfer for initializing a TLS handshake.



**Figure 4:** Outline of TLS handshake, by cisco

Next I ran another packet capture on my WiFi interface and began paying attention to the port numbers of the incoming and outgoing packets. I noticed there was an application running on port 4070 creating lots of traffic on my WiFi interface. I applied the following filter to reduce the number of packets: “tcp.port == 4070”. I did a reverse lookup on the remote hosts IP address and the IP was owned by Google. I then changed the filter to show only a single TCP stream and noticed that all the packets referring to the IP address 104.154.127.116 belonged to the same stream index. As shown in **Figure 5,** I noticed the packets in the middle of the connection contained larger Payloads when my local machine was the destination. This means that the remote server was sending large quantities of data to my local machine over TCP and at the time I was streaming music from Spotify. I then looked online to see what the default port that Spotify listens on and it was indeed 4070. This meant that Spotify uses Google’s cloud service to host their application since the IP address was owned by Google.

**A screenshot of a computer

Description automatically generated**

**Figure 5:** Display of large incoming packet sizes over TCP on port 4070

The connection began with a 3-way TCP handshake and my local machine initialized the handshake. After the initial handshake there was some incoming network packets that had the TCP PSH flag set. The default method of packet transmission over TCP is to maximize packet size and minimize the number of packets being transferred over a network. The application layer generates data that is buffered in the Transport Layer until the maximum packet size has been reached and then the Transport layer moves the packet to the Network layer for transmitting to a remote host. This is undesirable for some applications. Setting the PSH flag allows the Application layer to send a signal to the Transport layer to release the packet to the Network early, even if the packet has not reached maximum size.[5] For applications like streaming music this functionality is desirable since the Application layer can send small to mid-sized packets early until the buffer has gotten a desirable amount ahead of the user and then begin sending full-size packets.

This packet stream did not contain any encryption as all packets were sent using TCP without TLS or any other encryption protocols. However, the data in the Payload is stored in an encrypted form on the remote server and sent in its encrypted form over TCP. The data then requires the user to have the Spotify app with a valid license key for decrypting. So, TLS is not necessary for encrypting and preserving copyrights, the data is already encrypted prior to transporting it. Since the entire connection was over TCP the stream was disassembled by one outgoing and one incoming packet with the TCP FIN flags set. Lastly, as shown in **Figure 6** the only packet that did not have the TCP ACK flag set was the initial packet sent from my local machine to request a synchronized connection with the Spotify server.

**A screenshot of a social media post

Description automatically generated**

**Figure 6:** Showing that only the initial packet did not have the ACK flag set

Following I performed another Wireshark packet capturing session that contained packets using SSHv2 packets. Wireshark allowed me to easily investigate what kind of packets were being transferred across the network. Next I applied a filter to only show packets that referenced the remote IPv4 address I found in one of the SSHv2 packets (ip.dst == 192.168.0.9 || ip.src == 192.168.0.9). Since the IP address is in the 192.168.0.# range I knew that the remote host was within my local device’s LAN.

**Figure 7** shows the connection between my local machine and the remote host being set-up. The connection was initiated at packet number 15 with a TCP SYN connection request being sent from my local machine (192.168.0.2) to the remote host (192.168.0.9). The standard TCP three-way handshake took place to establish a synchronize connection between my local machine and the remote host. After the two machines synchronized my local machine sent an OpenSSH request to the host in packet 18. There are multiple places in Wireshark to tell that packet 18 was an OpenSSH request. The first is displayed in the “Info” field where the following message is displayed: Client: Protocol (SSH-2.0-OpenSSH\_7.6p1 Ubuntu-4ubuntu0.3). The message tells us that SSH is the requested protocol and is accompanied by the version supported on my local machine. I am working on a Windows 10 box with the Windows Subsystem Linux (WSL) enabled and I use Ubuntu as the driving OS for my Linux subsystem. Therefore, in the SSH request we can see the version of SSH supported by my local machine is of Ubuntu origin.

Packet 19 is an acknowledgement of the SSH request from the host to the client and it is accompanied by a “TCP Window Update”. A TCP Window Update is sent to confirm how many bytes of data the host can receive before the buffer is full. When transmitting packets, the sender is not supposed to send more bytes than specified in the Window Update packet. This size restriction will be used as a flow control mechanism to prevent the connection buffers from overflowing. As shown in **Figure 7** the set window size value can be found in the OSI section of the Wireshark GUI under the TCP tab. Look for the Window Size Value, Calculated Window Size, and Window Size Scaling Factor fields. We can see that the current Window Size value has been set to 4096. This value can be adjusted during the connection by either the host or the client depending upon how fast the machine is on either end.

Next we see in packet 21 that the host sent a response to our SSH request containing the version of SSH it supports. In packet 22 the client sends its public key to the host and in 24 the host sends its public key back to the client. The client then sends a Diffie-Hellman encryption request and both the client, and the host produce new encryption keys from the shared information. Once the new keys are acknowledged in packet numbers 28 and 30 the devices can begin sending encrypted data. In the middle of the connection we can see “Encrypted Packets” being sent across the communication channel. At the end of the TCP stream in **Figure 8** we can see the client initialized the breakdown of the channel by sending a packet with the ACK and FIN flags set. The host then returns ACK packets for all the remaining “Encrypted Packets” being received and ends the connection with its own packet that has the ACK and FIN flags set.

A screenshot of a social media post

Description automatically generatedA screenshot of a social media post

Description automatically generated

**Figure 8:** Breakdown of SSH communication stream

**Figure 7:** Beginning of SSH communication stream

For the next packet stream, I saw DNS protocol being used in the protocol field within the packet section of the Wireshark GUI. I was able to find the stream number of the connection under the OSI section of the Wireshark GUI. As you can see in **Figure 9** the Stream Index field is located under the UDP subcategory and the value is 2. I then altered my filter to show only the packets that referred to the same stream index (udp.stream == 2). With the applied UDP stream filter only two packets were part of the connection. This makes sense because my local machine sent a DNS query to retrieve the IPv4 address from the supplied human readable domain github.com. The query request went to my local router or the network gateway where the gateway searched its DNS table for the domain name github.com. Once the gateway located the IPv4 address (140.82.114.3) associated with github.com it returned the address within the Payload of the second packet. All the communication within this stream happened within the LAN. Therefore, no authentication nor encryption was necessary to send the DNS query request or receive the DNS query results. Since there was no encryption and the requests were made using DNS over UDP, there was no initial communication set-up or break down necessary.

A screenshot of a social media post

Description automatically generated

**Figure 9:** DNS query and response over UDP

The final packet stream we will be looking at is an example of pushing git commits up to a remote GitHub repository. The connection uses TCP along with TLSv1.3 to encrypt and transport data. The connection was initiated by the standard TCP 3-way handshake and then the client sent the Client Hello TLS request. The host then replied with its own “Server Hello” protocol accompanied by a change of cipher spec. At this point the host began sending encrypted Application data to the client over TLSv1.3. TLSv1.3 differs from the previous version TLSv2.0 we looked at because as there are no Certs or Cert status exchanges between the client and the host. As shown in **Figure 10,** the commits were sent successfully as encrypted data over TLSv1.3 to the GitHub server and the communication was broken down in the standard TCP method of both the client and the host sending packets with the TCP ACK and FIN flags set.

A screenshot of a computer

Description automatically generated

**Figure 7:** Git commits being sent to remote Github repo.

**Conclusion:**

As displayed Wireshark is useful for troubleshooting communication problems over a network. When communication errors happen Wireshark will display the error packets in order. The packets can then be analyzed to hypothesize what could be causing a networking issue. Like how system logs are read to troubleshoot application errors. Since Wireshark categorizes the information it displays into the various stages of the OSI model we can effectively identify what might be causing an issue within our network. By monitoring the packets being sent over our network we can better understand where a network may be vulnerable to attacks. When vulnerabilities are identified over a network, Wireshark allows us to save our packet captures to be easily shared with a cybersecurity team for rectifying.

**Bibliography**

[1] liveaction.com. Retrieved From <https://www.liveaction.com/blog/network-packet/>

[2] geeksforgeeks.org. Retrieved From <https://www.geeksforgeeks.org/layers-of-osi-model/>

[3] wireshark.org. Retrieved From <https://www.wireshark.org/docs/wsug_html_chunked/ChAdvFollowStreamSection.html>

[4] CISCO. 2019. Retrieved From <https://www.cisco.com/c/en/us/support/docs/security-vpn/secure-socket-layer-ssl/116181-technote-product-00.html>

[5] geeksforgeeks.org. Retrieved From <https://www.geeksforgeeks.org/tcp-flags/>

**Reflection:**

Writing this paper helped me solidify my understanding of the OSI model and the purpose each layer serves. Prior to this paper I heard of the OSI model, however I did not know the order or what each layer did. I did not know packets were created in a structured order outlined by the OSI model. I learned how packets are structured and some of the various options that can be set within a packet. I learned there are numerous flags that can be set when transmitting a packet that tell the receiver the purpose of sending the packet across the network. I understand what the headers of a packet contain and how the header data is used to route and read packets within a network. Lastly, I was able to solidify my understanding of how various protocols established and tore down communication sessions. I got to view how 3-way handshakes are performed over TCP along with how the client and the server send packets with the FIN flag set to tear down a TCP stream. I was able to see a TLS connection get initialized and how the public keys were shared between the client and the host to create a cipher for encrypting data and how TLS ran on top of TCP.