**Chapter 2. Using Capture Filters**

**Introduction**

It is important to distinguish between these two types of filters:

* Capture filters are configured before we start to capture data, so only data that is approved with the filters will be captured. All other data will be lost.
* Display filters are filters that filter data after it has been captured. In this case, all data is captured, and you configure what data you wish to display.

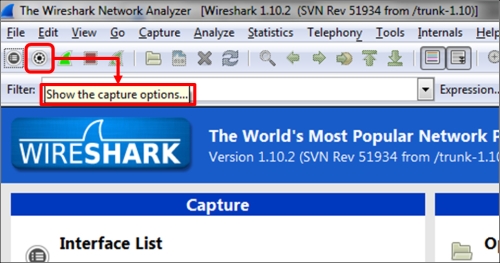
Capture filters are based on the tcpdump syntax presented in the libpcap/WinPcap library, while the display filters syntax was presented some years later. The display and capture filters have different syntaxes!

**Configuring capture filters**

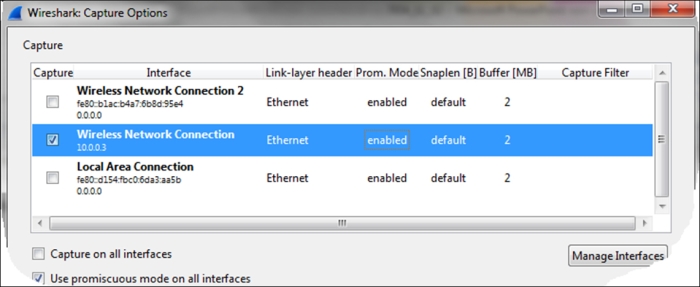
Before configuring a capture filter, you will design what you want to capture, and prepare your filter for this.

For configuring capture filters before starting with the capture, go through the following steps:

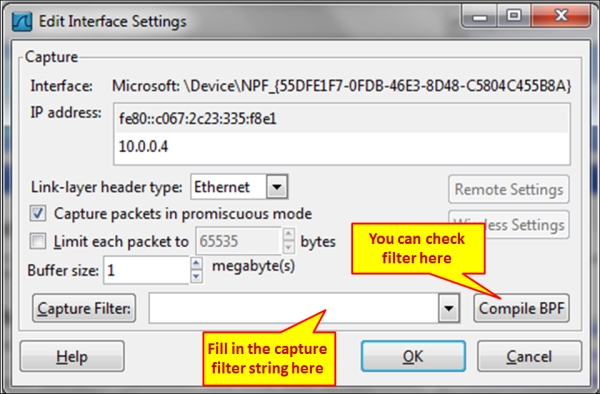
1. For configuring a capture filter, click on the **Show the capture options…** button, second from the left, as shown in the following screenshot:



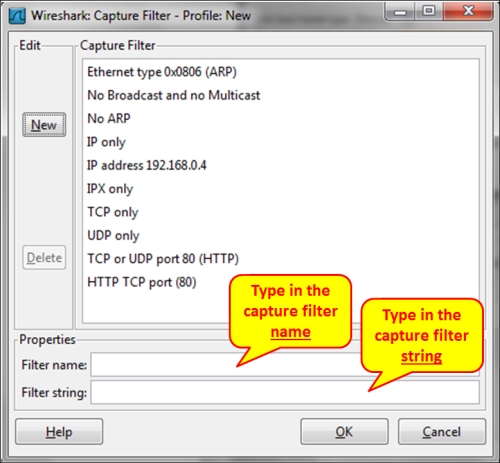
1. The **Wireshark: Capture Options** window will open as you see in the following screenshot:



1. Double-click on the interface on which you want to configure the capture filter.
2. The **Edit Interface Settings** window will open up, as in the following screenshot:



1. Now, we can configure the capture filters by simply writing the filter string in the **Wireshark: Capture Filter** window, or click on the **Capture Filter:** button; the following window will open:



**How it works...**

The **Wireshark: Capture Filter** window enables you to configure filters according to **Berkeley Packet Filter** (**BPF**). After writing a filter string, you can click on the **Compile BPF** button, and the BPF compiler will check your syntax, and if it's wrong you will get an error message. When you type a filter string in the capture filter text box, and the filter string is correct, it will become green, and if not, it will become red.

The BPF filter only checks if the syntax is right. It does not check if the condition is correct. For example, if you type the string host without any parameter, you will get an error and the string will become red, but if you type host 192.168.1.1000, it will pass and the window will become green.

BPF is a syntax coming from the paper *The BSD Packet Filter,* can be seen at: <http://www.tcpdump.org/papers/bpf-usenix93.pdf>.

Capture filters are made out of a string containing a filtering expression. This expression selects the packets which will be captured and which packets will be ignored. Filter expressions consist of one or more primitives. Primitives usually consist of an identifier (name or number) followed by one or more qualifiers. There are three different kinds of qualifiers:

* **Type**: These qualifiers say what kind of thing the identifier name or number refers to. Possible types are host for host name or address, net for network, port for TCP/UDP port, and so on.
* **Dir (direction)**: These qualifiers specify a particular transfer direction to and/or from ID. For example src indicates source, dst indicates destination, and so on.
* **Proto (protocol)**: These are the qualifiers that restrict the match to a particular protocol. For example, ether for Ethernet, ip for Internet Protocol, arp for Address Resolution Protocol, and so on.

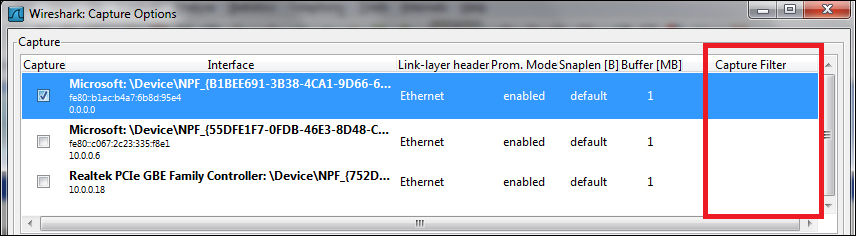
Identifiers are the actual condition that we test. Identifier can be the address 10.0.0.1, port number 53, or network address 192.168.1 (this is an identifier for network 192.168.1.0/24).

For example, in the filter tcp dst port 135, we have:

* dst is the dir qualifier
* port is the type qualifier
* tcp is the Proto qualifier

**There's more...**

You can configure different capture filters on different interfaces:



This can be used when you capture traffic on two interfaces of a device, and want to check for different packets on the two sides.

**Configuring Ethernet filters**

When talking about Ethernet filters, we refer to Layer-2 filters that are MAC address-based filters. In this recipe we will refer to these filters and what we can do with them.

**Getting ready**

The basic Layer 2 filters are:

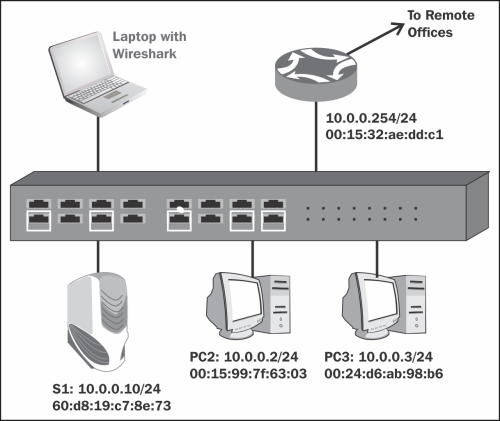
* ether host <Ethernet host>: To get the Ethernet address.
* ether dst <Ethernet host>: To get the Ethernet destination address
* ether src <Ethernet host>: To get the Ethernet source address
* ether broadcast: To capture all Ethernet broadcast packets
* ether multicast: To capture all Ethernet multicast packets
* ether proto <protocol>: To filter only the protocol type indicated in the protocol identifier.
* vlan <vlan\_id>: To pass only packets from a specific VLAN that is indicated in the identifier field.

For negating a filter rule, simply type the word not or ! in front of the primitive. For example:

Not ether host <Ethernet host> or ! Ether host <Ethernet host> will capture packets that are not from/to the Ethernet address specified in the identifier field.

**How to do it...**

In the following diagram, Wireshark is running on the laptop connected to the LAN switch, with port mirror to the entire switch (to VLAN1). The /24 notation refers to a subnet mask of 24 bits, 255.255.255.0 in decimal.



Follow the instructions in the *Configuring capture filters* recipe, and configure filters as follows:

1. To capture packets only from/to a specific MAC address, configure ether host 00:24:d6:ab:98:b6.
2. To capture packets going to a destination MAC address, configure ether dst 00:24:d6:ab:98:b6.
3. To capture packets coming from a source MAC address, configure ether src 00:24:d6:ab:98:b6.
4. To capture broadcast packets, configure ether broadcast or ether dst ff:ff:ff:ff:ff:ff.
5. To capture multicast packets, configure ether multicast.
6. To capture a specific Ether Type (number in Hexadecimal value), configure ether proto 0800.

**How it works…**

The way capture filters work with source host and destination host is simple—the capture engine simply compares the condition with the actual MAC addresses, and passes only what is relevant.

A broadcast address is an address in which the destination address is all 1's, that is, ff:ff:ff:ff:ff:ff:ff, therefore when you configure a broadcast filter, only these addresses will pass the filter. Broadcast addresses can be:

* L3 IPv4 broadcast that is converted to L2 broadcast; for example, IP packet to 10.0.0.255 (class C subnet), which will be converted to L2 broadcast in the destination MAC field.
* A network-related broadcast; for example, IPv4 ARP (Address Resolution Protocol) that sends a broadcast as a part of network operation.

Ethernet protocol refers to the ETHER-TYPE field in the Ethernet packet that indicates what will be the upper Layer protocol. Common values here are 0800 for IPv4, 86dd for IPv6, and 0806 for ARP.

**There's more...**

* To configure filter for a specific VLAN, use vlan <vlan number>

**Configuring host and network filters**

When talking about host and network filters, we refer to Layer 3 filters that are IP address-based filters.

**Getting ready**

The basic Layer 3 filters are:

* ip or ip6: To capture IP or IPv6 packets.
* host <host>: To get host name or address.
* dst host <host>: To get destination host name or address.
* src host <host>: To get source host name or address.
* gateway <Host name or address>: It captures traffic to or from the hardware address but not to the IP of the host. This filter captures traffic going through the specified router. This filter requires a host name that is used and can be found by the local system's name resolution process (for example, DNS).
* net <net>: All packets to or from the specified IPv4/IPv6 network.
* dst net <net>: All packets to the specified IPv4/IPv6 destination network.
* src net <net>: All packets to the specified IPv4/IPv6 destination network.
* net <net> mask <netmask>: All packets to/from the specific network and mask. This syntax is not valid for the IPv6 network.
* dst net <net> mask <netmask>: All packets to/from the specific network and mask. This syntax is not valid for the IPv6 network.
* src net <net> mask <netmask>: All packets to/from the specific network and mask. This syntax is not valid for the IPv6 network.
* net <net>/<len>: All packets to/from the <net> network with <len> length in bits.
* dst net <net>/<len>: All packets to/from the <net> network with <len> length in bits.
* dst net <net>/<len>: All packets to/from the <net> network with <len> length in bits.
* broadcast: All broadcast packets.
* multicast: All multicast packets.
* ip proto <protocol code>: It captures packets while the IP protocol field equals to the <protocol> identifier. There can be various protocols, such as, TCP (6), UDP (17), ICMP (1), and so on.
* icmp[icmptype]==<identifier>: It captures ICMP packets, while the identifier is ICMP codes, such as icmp-echo and icmp-request.

**How to do it...**

Follow the instructions mentioned in the *Configure capture filters* recipe, and configure filters as follows:

1. For capturing packets to/from host 10.10.10.1, configure host 10.10.10.1.
2. For capturing packets to/from host at [www.cnn.com](http://www.cnn.com), configure host www.cnn.com.
3. For capturing packets to host 10.10.10.1, configure dst host 10.10.10.1.
4. For capturing packets from host 10.10.10.1, configure src host 10.10.10.1.
5. For capturing packets to/from network 192.168.1.0/24, configure net 192.168.1 or net 192.168.1.0 mask 255.255.255.0 or net 192.168.1.0/24.
6. For capturing all data without broadcasts or without multicasts, configure not broadcast or not multicast.
7. For capturing only ICMP packets, configure ip proto 1.
8. For filtering only ICMP Echo's pings, you can use ICMP messages or message codes. configure icmp[icmptype]==icmp-echo or icmp[icmptype]==8.

**How it works…**

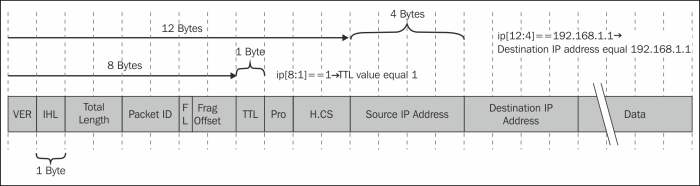
For host filtering, when you type a host name, Wireshark will translate the name to an IP address, and capture packets that refer to this address. For example, if you configure a filter host [www.cnn.com](http://www.cnn.com), it will be translated by a name resolution service (mostly DNS) to an IP address, and will show you all packets going to and from this address. Note that in this case, if CNN website will forward you to other websites on other addresses, only packets to the first address will be captured.

**There's more...**

Some more useful filters:

* ip multicast: IP multicast packets
* ip broadcast: IP broadcast packets
* ip[2:2] == <number>: IP packet size
* ip[8] == <number>: TTL (Time To Live) value
* ip[9] == <number>: Protocol value
* (ip[12:4] = ip[16:4]): IP source equal to IP destination address
* ip[2:2]==<number>: Total length or IP packet
* ip[9] == <number>: Protocol identifier

These filters are further explained in the *Configuring byte offset and payload matching filters* recipe at the end of this chapter. The principle, as illustrated in the following diagram, is that the first number in the brackets defines how many bytes are from the beginning of the protocol header, and the second number indicates how many bytes to watch.



**See also**

For more filters, refer to the tcpdump manual pages at <http://www.tcpdump.org/tcpdump_man.html>.

**Configuring TCP/UDP and port filters**

In this recipe we will present Layer 4 TCP/UDP port filters and how we can use them with capture filters.

**Getting ready**

The basic Layer 4 filters are:

* port <port>: When the packet is a Layer 4 protocol, such as TCP or UDP, this filter will capture packets to/from the port indicated in the identifier field.
* dst port <port>: When the packet is a Layer 4 protocol, such as TCP or UDP, this filter will capture packets to the destination port indicated in the identifier field
* src port <port>: When the packet is a Layer 4 protocol, such as TCP or UDP, this filter will capture packets to the source port indicated in the identifier field

The port-range matching filters are:

* tcp portrange <p1>-<p2> or udp portrange <p1>-<p2>: TCP or UDP packets in the port range of p1 to p2
* tcp src portrange <p1>-<p2> or udp src portrange <p1>-<p2>: TCP or UDP packets in the source port range of p1 to p2
* tcp dst portrange <p1>-<p2> or udp src portrange <p1>-<p2>: TCP or UDP packets in the destination port range of p1 to p2

**How to do it...**

Follow the instructions in the *Configuring capture filters* recipe, and configure filters as follows:

1. To capture packets to port 80 (HTTP), configure dst port 80 or dst port http.
2. To capture packets to or from port 5060 (SIP), configure port 5060.
3. To capture packets to or from port 5060 (SIP), configure port 5060.
4. To capture the start (SYN flag) and end (FIN flag) packets of all TCP connections, configure tcp[tcpflags] & (tcp-syn|tcp-fin) != 0.

In tcp[tcpflags] & (tcp-syn|tcp-fin) != 0, it is important to note that this is a bitwise and operation, not a logical and operation. For example, 010 or 101 equals 111, and not 000.

1. To capture all TCP packets with RST (Reset) flag set to 1, configure tcp[tcpflags] & (tcp-rst) != 0.
2. Length filters are configured in the following way:
   * less <length>: It captures only packets with length less than or equal to length identifier. This is equivalent to len <= <length>.
   * greater <length>: It captures only packets with length greater than or equal to length identifier. This is equivalent to <len >= length>.

For example,

* + tcp portrange 2000-2500
  + udp portrange 5000-6000

Port range filters can be used for protocols that work in a range of ports rather than specific ones.

**How it works…**

Layer 4 protocols, mostly TCP and UDP, are the protocols that connect between end applications. The end node on one side (for example, a web client) sends a message to the other side (for example, a web server), requesting to connect to it. The codes of the processes that send the request and the processes that receive the request are called port numbers.

Both in TCP and UDP, the port numbers indicate the application codes. The difference between them is that TCP is a connection-oriented, reliable protocol, while UDP is a connectionless unreliable protocol. TCP flags are sent in packets in order to establish, maintain, and close connections. A signal is set when a specific bit in the packet is set to 1. The most common flags that are in use are:

* **SYN**: A signal sent in order to open a connection
* **FIN**: A signal sent in order to close a connection
* **ACK**: A signal sent to acknowledge received data
* **RST**: A signal sent for immediate close of a connection
* **PSH**: A signal sent for pushing data for processing by the end process (application)

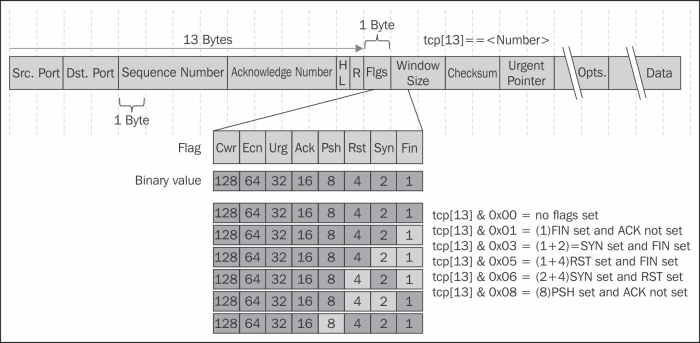
Using capture filters you can filter packets to/from specific applications, along with filtering packets with specific flags turned on.

**There's more...**

Some problematic scenarios (mostly attacks…) are:

* tcp[13] & 0x00 = 0: No flags set (null scan)
* tcp[13] & 0x01 = 1: **FIN** set and **ACK** not set
* tcp[13] & 0x03 = 3: **SYN** set and **FIN** set
* tcp[13] & 0x05 = 5: **RST** set and **FIN** set
* tcp[13] & 0x06 = 6: **SYN** set and **RST** set
* tcp[13] & 0x08 = 8: **PSH** set and **ACK** not set

In the following diagram you can see how it works. tcp[13] is the number of bytes from the beginning of the protocol header, when the values 0,1,3,5,6, and 8 refer to the flag locations.



**Configuring compound filters**

Structure filters are simply made for writing filters out of several conditions. It uses simple conditions, such as not, and, and or for creating structured conditions.

**Getting ready**

Structured filters are written in the following format: [not] primitive [and|or [not] primitive ...].

The following modifiers are commonly used in the Wireshark capture filters:

* ! or not
* && or and
* || or or

**How to do it...**

To configure structured filters, you simply write the conditions according to what we learned in the previous recipes, with conditions to meet your requirements. Some common filters are:

1. For capturing only unicast packets, configure not broadcast and not multicast.
2. For capturing HTTP packets to [www.youtube.com](http://www.youtube.com), configure host www.youtube.com and port 80.
3. A capture filter for telnet that captures traffic to and from a particular host, configures tcp port 23 and host 192.180.1.1.
4. For capturing all telnet traffic not from 192.168.1.1, configure tcp port 23 and not src host 192.168.1.1.

**How it works…**

Some examples for structured filters:

For capturing data to tcp port 23 (Telnet) from source port range of 5000-6000, configure tcp dst port 23 and tcp src portrange 5000-6000.

**There's more...**

Some interesting examples are as follows:

* host www.mywebsite.com and not (port 80 or port 23)
* host 192.168.0.50 and not tcp port 80
* host 10.0.0.1 and not host 10.0.0.2

**Configuring byte offset and payload matching filters**

Byte offset and payload matching filters come to provide us with a flexible tool for configuring self-defined filters (filters for fields that are not defined in the Wireshark dissector and filters for proprietary protocols). By understanding the protocols that we work with and understanding their packet structure, we can configure filters that will watch a specific string in the captured packets, and filter packets according to it. In this recipe we will learn how to configure these types of filters, and we will also see some common and useful examples of the subject.

**Getting ready**

To configure byte offset and payload matching filters, start Wireshark and follow the instructions in the *Configuring capture filters* recipe in the beginning of this chapter.

**How to do it...**

1. String matching filters comes to check a specific string in the packet header. It comes in the following format: proto [Offset: bytes].

With this filter we can create filters for strings over IP, TCP, and UDP.

1. For IP string-matching filters you can create the following filter: ip [Offset:Bytes].
2. For matching application data, that is, to look into the application data that is carried by TCP or UDP, the most common uses of it are: tcp[Offset:Bytes] Or udp[Offset:Bytes].

**How it works…**

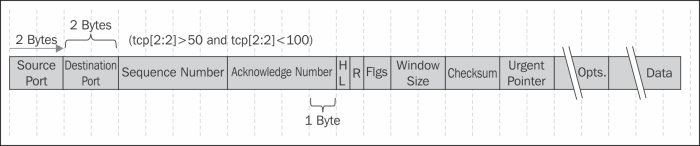
The general structure of offset filter is:

proto [Offset in bytes from the start of the header : Number of bytes to check]

Common examples for string matching filters are:

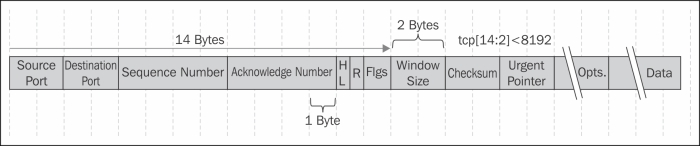
1. For filtering destination TCP ports between 50 and 100, configure (tcp[2:2] > 50 and tcp[2:2] < 100).

Here we count two bytes from the beginning of the TCP header, and check the next two bytes to be lower than 100 and higher than 50.



1. For checking TCP window size smaller then 8192, configure tcp[14:2] < 8192.

Here we count two bytes from the beginning of the TCP header, and check the next two bytes (the window size) to be less than 8192.



**There's more...**

You can also see additional filters in the tcpdump man pages:

1. To print all IPv4 HTTP packets to and from port 80, (that is to print only packets that contain data, not, for example, SYN, FIN or ACK-only packets), configure the following filter: tcp port 80 and (((ip[2:2] - ((ip[0]&0xf)<<2)) - ((tcp[12]&0xf0)>>2)) != 0).
2. To print the start and end packets (the SYN and FIN packets) of each TCP conversation that involves a non-local host, configure tcp[tcpflags] & (tcp-syn|tcp-fin) != 0 and not src and dst net <localnet>.
3. To print IP broadcast or multicast packets that were not sent via Ethernet broadcast or multicast, configure ether[0] & 1 = 0 and ip[16] >= 224.
4. To print all ICMP packets that are not echo requests/replies (that is, not ping packets), configure icmp[icmptype] != icmp-echo and icmp[icmptype] != icmp-echoreply.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Chapter 3. Using Display Filters**

**Introduction**

Display filters are filters that we apply after capturing data (filtered by capture filters or not), and when we wish to display only part of the data. Any data that is sent over the network can be filtered, and when filtered, you can create statistics and graphs according to it.

While using display filters, all the data was already captured and the display filters only decide what to display. Therefore, after filtering data, the capture file still contains the original data that was captured.

**Getting ready**

In general, a display filter string takes the form of a series of primitive expressions connected by conjunctions (and, or, or something else) and optionally preceded by not:

[not] Expression [and|or] [not] Expression...

While Expression can be any filter expression, such as ip.src==192.168.1.1 for the source address, tcp.flags.syn==1 for TCP SYN flag presence, and tcp.analysis.retransmission for TCP retransmissions, and|or are conjunctions that can be used in any combinations of expression, including brackets, multiple brackets, and any lengths of strings. There are several conditions to these. They can be one of the following:

| C-like Syntax | Shortcut | Description | Example |
| --- | --- | --- | --- |
| == | **Eq** | Equal | ip.addr == 192.168.1.1 or ip.addr eq 192.168.1.1 |
| != | **Ne** | Not equal | !ip.addr==192.168.1.1, ip.addr != 192.168.1.1, or ip.addr ne 192.168.1.1 |
| > | **Gt** | Greater than | frame.len > 64 |
| < | **Lt** | Less than | frame.len < 1500 |
| >= | **Ge** | Greater than or equal to | frame.len >= 64 |
| <= | **le** | Less than or equal to | frame.len <= 1500 |
|  | **is present** | A parameter is present | http.response |
|  | **contains** | Contains a string | http.host contains cisco |
|  | **matches** | A string matches the condition | http.host matches www.cisco.com |

You can insert a space character between parameters and operators or leave it without spaces.

Wireshark colorizes the display filter area in yellow whenever you use the != operator for combined expressions such as eth.addr, ip.addr, tcp.port, and udp.port, but this will not work due to the following reason.

When you type a filter expression such as ip.addr != 192.168.1.100, you will see **The packet contains the field ip.addr with a value different from 192.168.1.100**. Because an IP datagram contains both a source and a destination address, the expression will evaluate to true whenever at least one of the two addresses differs from 192.168.1.100. For this reason you should write !(ip.addr == 192.168.1.100); this will display **Show me all the packets for which it is not true that a field ip.addr have the value of 1.2.3.4**.

There are several operators. They can be as follows:

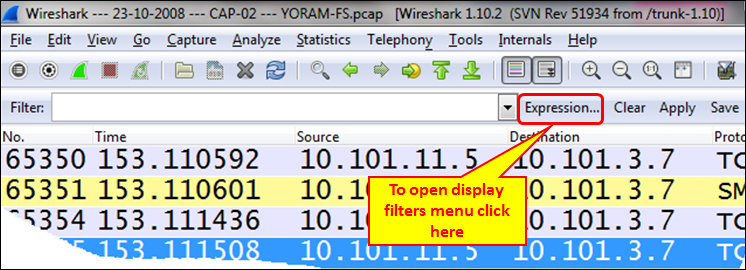
| C-like Syntax | Shortcut | Description | Example |
| --- | --- | --- | --- |
| && | and | Logical AND | ip.src==10.0.0.1 and tcp.flags.syn==1  All SYN flags sent from IP address 10.0.0.1 practically and all connections opened (or tried to be opened) from 10.0.0.1. |
| || | or | Logical OR | ip.addr==10.0.0.1 or ip.addr==10.0.02  All the packets going in or out the two IP addresses. |
| ! | not | Logical NOT | not arp and not icmp  All the packets that are neither ARP nor ICMP. |

**How to do it...**

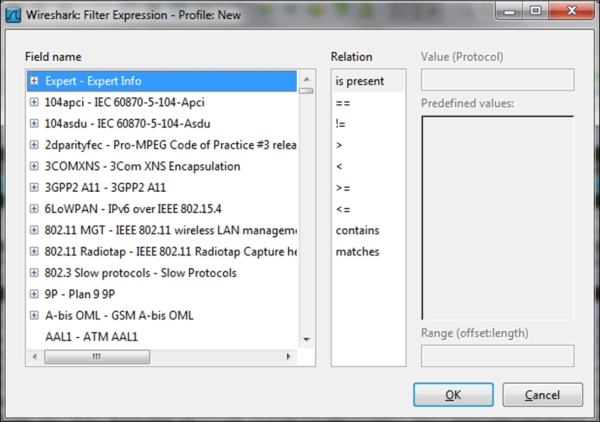
To configure display filters, you can choose any one of the methods mentioned earlier.

**Choosing from the filters menu**

For choosing from the filters menu, navigate to the display filter pane on the upper side of the window and click on the **Expression...** button as you see in the following screenshot:



When you click on the **Expression...** button, the following window will open:



There are five important panes in the filters menu:

* **Field name**: In this pane you configure the filter parameter. You can go to the protocol by typing its name, and get to the protocol parameter by clicking on the **+** sign to the left of the list.

One example for this would be: type ipv4 to get to the **IPv4** protocol, click on the **+** sign to expand the protocol parameters (or press *Enter* twice) and choose **ip.addr** to filter a specific IP address.

Another example would be to type tcp to get to the **TCP** protocol, click on the **+** sign to the left of the protocol parameter and choose **tcp.port** for the source or destination port number.

* **Relation**: This is the pane from where you choose the operator. You can choose == for equal, != for not equal, and so on.

An example for this would be: type sip to get to the **SIP** protocol, choose **sip.Method**, and choose **==** from the **Relation** pane. Type invite in the **Value (Protocol)** pane. This will filter all the SIP INVITE methods.

* **Value**: Here you enter the value of the field that you have chosen before.

An example for this would be: type tcp to get to the **TCP** protocol, click on the **+** sign to go to the protocol parameter, choose **tcp.flags.syn** for the TCP SYN flag, and enter 1 in the **Value** field.

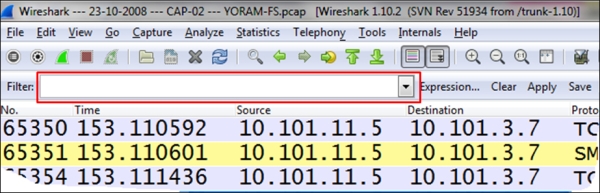
* **Predefined values**: When the value of the field you chose is not Boolean, there might be a list of options in this field.

An example for this would be: under **TCP**, there is an option named **tcp.option\_kind**. This option is related to **TCP** options. You will get a list of values that are possible.

* **Range (offset: length)**: This field provides you the length of the string in the offset:length format.

**Writing the syntax directly into the display filter window**

After getting used to the display filters syntax, you may find it easier to type directly into the filter window:

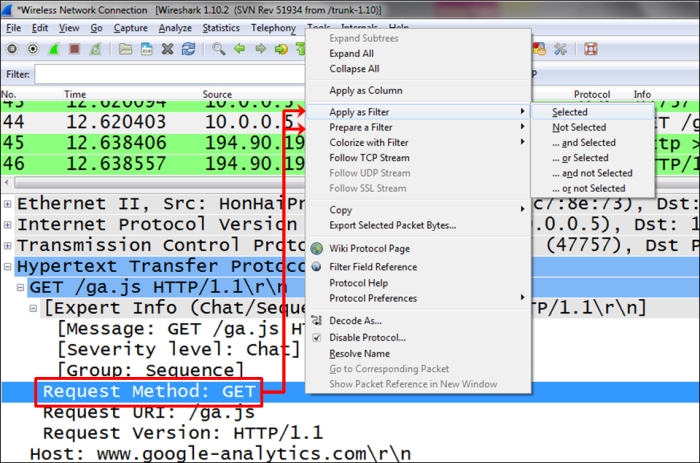


In this case, when you write a filter string into the window, the window will light up in one of the following three colors:

* **Green**: This is when the filter is correct and you can apply it.
* **Red**: This indicates a wrong string. Fix the string before you apply it.
* **Yellow**: Whenever you use the != operator, the display filter area will turn yellow. It doesn't mean your filter will not work, it is just a warning that it *may not* work.

**Choosing a parameter in the packet pane and defining it as a filter**

This is a very convenient option. You can choose any field from the packet detail pane in the captured file; right-click on it and you will get a few options, as illustrated in the following screenshot:



A couple of options are as follows:

* **Apply as Filter**: This will set a filter according to the field you choose and apply it to the captured data.
* **Prepare a Filter**: This will prepare a filter but not apply it. It will be applied when you click on the **Apply** button on the right-hand side of the filter window.

In both the options, you can choose to configure a filter:

* **Selected**: This will choose the selected field and parameter
* **Not Selected**: This will choose the the field and parameter that are not selected

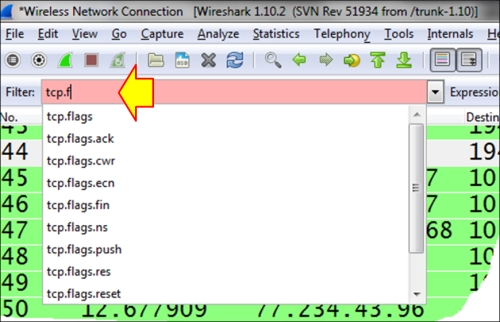
For example, right-clicking on the field http.request.method and choosing **Selected** will come with the filter string http.request.method == GET; while, choosing **Not Selected** will come with the string !(http.request.method == "GET").

You can also choose the options **... and selected**, **... or selected**, **... and not selected**, or **... or not selected** for structured filters.

**How it works...**

The display filter is a proprietary Wireshark language. There are many places where display filters can be used that will be discussed in the later chapters. Additional filters will be introduced in the upcoming recipes of this chapter.

You can always use the autocomplete feature to complete filter strings. For example, if you type in tcp.f, as shown in the following screenshot, the autocomplete feature lists possible display filter values that can be created beginning with tcp.f, that is, TCP flags (SYN, FIN, RST, and so on).

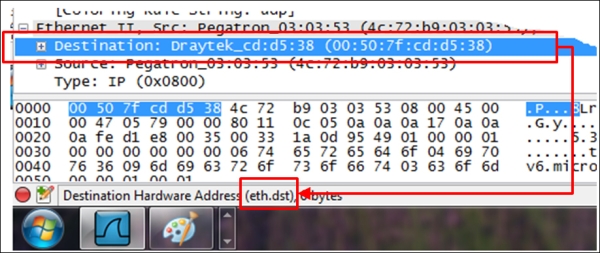


**There's more...**

Now we will cover some additional helpful features.

**What is the parameter we filter?**

Anytime you mark a specific field in the packet details pane, you will see the correlating filter string in the status bar at the bottom-left corner of the Wireshark window.

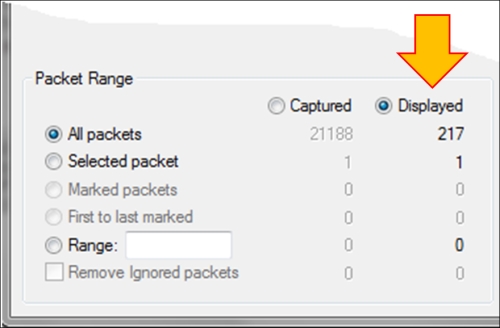


**Adding a parameter column**

You can also right-click on a parameter in the packet pane and choose **Apply as Column**. This will add a column with the specific parameter. For example, you can choose the parameter tcp.window\_size\_value and add it as a column to the packet list pane, so you will be able to watch the TCP window size online. This influences TCP performance, as we will learn in [Chapter 9](http://my.safaribooksonline.com/9781849517645/ch09_html), *UDP/TCP Analysis*.

**Saving the displayed data**

To save the displayed data, you can navigate to **File** | **Export Specified Packets...** and choose which packets to save.



**Configuring Ethernet, ARP, host, and network filters**

In this recipe we will discuss how to configure filters of layers 2 and 3, that is, Ethernet- and IP-based filters respectively. We will also discuss **Address Resolution Protocol** (**ARP**) filters.

**Getting ready**

In layer 2 we will configure Ethernet-based filters, while in layer 3 we will configure IP-based filters. In Ethernet we have filters based on the Ethernet frame and the MAC address, while in IP we have filters based on the IP packet and address.

The common frame delta filters are as follows:

* frame.time\_delta: This is used for the time delta between the current and previously captured frames; this will be used in statistical graphs displayed in [Chapter 5](http://my.safaribooksonline.com/9781849517645/ch05_html), *Using Advanced Statistics Tools*
* frame.time\_delta\_displayed: This is used for the time delta between current and previously displayed frames; this will be used in statistical graphs displayed in [Chapter 5](http://my.safaribooksonline.com/9781849517645/ch05_html), *Using Advanced Statistics Tools*

Since the time between frames can influence TCP performance significantly, we will use the frame.time\_delta parameters in statistical graphs for monitoring TCP performance.

The common layer 2 (Ethernet) filters are as follows:

* eth.addr == <MAC Address>: This is used to display a specific MAC address
* eth.src == <MAC Address>: This is used to get the source MAC address
* eth.dst == <MAC Address>: This is used to get the destination MAC address
* eth.type == <Protocol Type (Hexa)>: This is used to get the Ethernet protocol types

The common ARP filters are as follows:

* arp.opcode == <value>: This is used for ARP requests/responses
* arp.src.hw\_mac == <MAC Address>: This is used to capture the ARP address of the sender

The common layer 3 (IP) filters are as follows:

* ip.addr == <IP Address>: This is used to get the source or destination IP address
* ip.src == <IP Address>: This is used to get the source IP address
* ip.dst == <IP Address>: This is used to get the destination IP address
* ip.ttl == <value>, ip.ttl < value>, or ip.ttl > <value>: This is used to get IP TTL (Time To Live) values
* ip.len = <value>, ip.len > <value>, or ip.len < <value>: This is used to get IP packet length values
* ip.version == <4/6>: This is used to get the IP protocol version (Version 4 or Version 6)

**How to do it...**

Here we will see some common examples for layer 2 and 3 filters.

| Address format | Syntax | Example |
| --- | --- | --- |
| Ethernet (MAC) address | eth.addr == xx:xx:xx:xx:xx:xx  Here, x = 0 to f. | eth.addr == 00:50:7f:cd:d5:38 |
| eth.addr == xx-xx-xx-xx-xx-xx  Here, x = 0 to f. | eth.addr == 00-50-7f-cd-d5-38 |
| eth.addr == xxxx.xxxx.xxxx  Here x = 0 to f. | eth.addr == 0050.7fcd.d538 |
| Broadcast MAC address | Eth.addr == ffff.ffff.ffff |  |
| IPv4 host address | ip.addr == x.x.x.x  Here, x = 0 to 255. | Ip.addr == 192.168.1.1 |
| IPv4 network address | ip.addr == x.x.x.x/y  Here x = 0 to 255, y = 0 to 32. | ip.addr == 192.168.200.0/24  This covers all the addresses in the network 192.168.200.0 mask 255.255.255.0. |
| IPv6 host address | ipv6.addr == x:x:x:x:x:x:x:x  ipv6.addr == x::x:x:x:x  Here in the format of nnnn, n = 0 to f (Hex). | ipv6.addr == fe80::85ab:dc2e:ab12:e6c7 |
| IPv6 network address | ipv6.addr == x::/y  Here x = 0 to f (Hex) and y = 0 to 128. | ipv6.addr == fe80::/16  This covers all the addresses that start with the 16 bits fe80. |

The table refers to ip.addr and ipv6.addr filter strings. The value for any field that has an IP address value can be written the same way.

**Ethernet filters**

These are classified into two categories:

* To display only packets sent from or to specific MAC addresses, use something like these: eth.src == 10:0b:a9:33:64:18 and eth.dst == 10:0b:a9:33:64:18
* To display only broadcasts, use Eth.dst == ffff.ffff.ffff

**ARP filters**

These are classified into two categories:

* To display only ARP requests, use arp.opcode == 1
* To display only ARP responses, use arp.opcode == 2

**IP and ICMP filters**

* To display only packets from a specific IP address, use something like this: ip.src == 10.1.1.254
* To display only packets that are not from a specific address, use something like this: !ip.src == 64.23.1.1
* To display only packets between two hosts, use something like these: ip.addr == 192.168.1.1 and ip.addr == 200.1.1.1
* To display only packets that are sent to multicast IP addresses, use something like this: ip.dst == 224.0.0.0/4
* To display only packets coming from the network 192.168.1.0/24 (mask 255.255.255.0), use ip.src==192.168.1.0/24
* To display only IPv6 packets to/from specific addresses, use something like the following:
  + ipv6.addr == ::1
  + ipv6.addr == 2008:0:130F:0:0:09d0:666A:13ab
  + ipv6.addr == 2006:0:130f::9c2:876a:130b
  + ipv6.addr == ::

**Complex filters**

* To check for packets sent from the network 10.0.0.0/24 to a website that contains the word packt, use ip.src == 10.0.0.0/24 and http.host contains "packt"
* To check for packets sent from the network 10.0.0.0/24 to websites that end with .com, use ip.addr == 10.0.0.0/24 and http.host matches "\.com$"
* To check for all the broadcasts from the source IP address 10.0.0.0, use ip.src == 10.0.0.0/24 and eth.dst == ffff.ffff.ffff
* To check for all the broadcasts that are not ARP requests, use not arp and eth.dst == ffff.ffff.ffff
* To check for all the packets that are not ICMP, use !arp || !icmp, and to check for all the packets that are not ARP, use not arp or not icmp

**How it works...**

Here are some explanations to the filters we saw in the *How to do it...* section of this recipe.

**Ethernet broadcasts**

In Ethernet, broadcasts are packets that are sent to addresses with all 1s in the destination field and this is why, to find all broadcasts in the network, we insert the filter eth.dst == ffff.ffff.ffff.

**IPv4 multicasts**

IPv4 multicasts are packets that are sent to an address in the address range 224.0.0.0 to 239.255.255.255 that is in binary of the address range 11100000.00000000.00000000.00000000 to 11101111.11111111.11111111.11111111.

If you look at the binary representation, a destination multicast address is an address that starts with three 1s and a 0, and therefore, a filter to IPv4 multicast destinations will be ip.dst == 224.0.0.0/4. That is, an address that starts with four 1s (224), and a subnet mask of 4 bits (/4) will indicate a network address ranger from 224 to 239 that will filter multicast addresses.

**IPv6 multicasts**

IPv6 multicasts are packets that are sent to an address that starts with ff (first two hex digits = ff), then one-digit flags, and scope. Therefore when we write the filter ipv6.dst == ff00::/8, it means to display all the packets in IPv6 that are sent to an address that starts with the string ff, that is, IPv6 multicasts.

**See also**

* For more information on Ethernet, refer to [Chapter 7](http://my.safaribooksonline.com/9781849517645/ch07_html), *Ethernet, LAN Switching, and Wireless LAN*

**Configuring TCP/UDP filters**

TCP and UDP are the main protocols in layer 4 that provide connectivity between end applications. Whenever you start an application from one side to another, you start the session from a source port, usually a random number equal or higher than 1,024, and connect to a destination port, which is a well-known or registered port that waits for the session on the other side. These are the port numbers that identify the application that works over the session.

Other types of filters refer to other fields in the UDP and TCP headers. In UDP we have a very simple header with very basic data, while in TCP we have a more complex header that we can get much more information from.

In this recipe we will concentrate on the possibilities while configuring TCP and UDP display filters.

**Getting ready**

As done earlier, we should plan precisely what we want to display and prepare the filters accordingly.

For TCP or UDP port numbers use the following display filters:

* tcp.port == <value> or udp.port == <value>: This is used for specific TCP or UDP ports (source or destination)
* tcp.dstport == <value> or udp.dstport == <value>: This is used for specific TCP or UDP destination ports
* tcp.srcport == <value> or udp.srcport == <value>: This is used for specific TCP or UDP destination ports

In UDP, the header structure is very simple: source and destination ports, packet length, and checksum. Therefore, the only significant information here is the port number.

TCP on the other hand is more complex and uses connectivity and reliability mechanisms that can be monitored by Wireshark. Using tcp.flags, tcp.analysis, and other smart filters will help you resolve performance problems (retransmissions, duplicate ACKs, zero windows, and so on), protocol operation issues such as resets, half-opens, and so on.

Common display filters in this category are as follows:

* tcp.analysis: This is used for TCP analysis criteria such as retransmission, duplicate ACKs, or window issues. Examples for this filter are as follows (you can use the autocomplete feature to get the full list of available filters):
  + tcp.analysis.retransmission: This is used to display packets that were retransmitted.
  + tcp.analysis.duplicate\_ack: This is used to display packets that were acknowledged several times.
  + tcp.analysis.zero\_window: This is used to display packets when a device on the connection end sends a zero-window message (that tells the sender to stop sending data on this connection, until window size increases again).

**Tip**

The tcp.analysis filters do not analyze the TCP header; they provide a protocol analysis through the Wireshark expert system.

* tcp.flags: These are used to find out if a flag(s) is set or not. Examples of this filter are as follows:
  + tcp.flags.syn == 1: This is used to check if the SYN flag is set.
  + tcp.flags.reset == 1: This is used to check if the RST flag is set.
  + tcp.flags.fin == 1: This is used to check if the FIN flag is set.

**Tip**

For TCP flags, the tcp.flags filter will be used to find out whether a specific flag is set or not.

* tcp.window\_size\_value < <value>: This is used to look for small TCP window sizes that are in some cases indications for slow devices.

**How to do it...**

Some examples for filters in TCP/UDP filters:

* To filter all the packets to the HTTP server, use tcp.dstport == 80
* To filter all the packets from the network 10.0.0.0/24 to the HTTP server, use ip.src==10.0.0.0/24 and tcp.dstport == 80
* For all the retransmissions in a specific TCP connection, use tcp.stream eq 16 && tcp.analysis.retransmission

To isolate a specific connection, place the mouse on a packet in the connection you want to watch, right-click on it, and choose **Follow TCP Stream**. A TCP stream is the data that is transferred between the two ends of the connection from the connection establishment to the connection tear down. The string tcp.stream eq <value> will appear in the display filter window. This is the stream you can work on now. In the preceding example, it came out as stream 16, but it can be any stream number (starting the count from stream 1 in the capture file).

Retransmissions are TCP packets that are sent again. It can be due to several reasons, as explained in [Chapter 9](http://my.safaribooksonline.com/9781849517645/ch09_html), *UDP/TCP Analysis*.

**Tip**

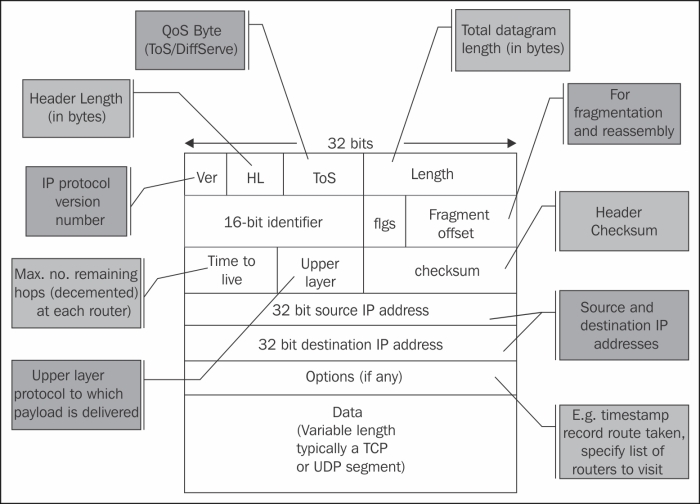
While monitoring phenomena such as retransmissions, duplicate ACKs, and others that influence performance, it is important to remember that these phenomena refer to a specific TCP connection.

Other examples of the types of TCP filters are as follows:

* To transfer all the window problems in a specific connection:
  + tcp.stream eq 0 && (tcp.analysis.window\_full || tcp.analysis.zero\_window)
  + tcp.stream eq 0 and (tcp.analysis.window\_full or tcp.analysis.zero\_window)
* To transfer all the packets from 10.0.0.5 to the DNS server: ip.src == 10.0.0.5 && udp.port == 53
* To transfer all the packets or protocols in TCP (for example HTTP) that contains the string cacti (case sensitive): tcp contains "cacti"
* To check all the packets that are retransmitted from 10.0.0.3: ip.src == 10.0.0.3 and tcp.analysis.retransmission
* To transfer all the packets to any HTTP server: tcp.dstport == 80
* To check all the connections opened from a specific host (if in a form of scan, can be a worm!): ip.src==10.0.0.5 && tcp.flags.syn==1 && tcp.flags.ack==0
* To check all the cookies sent from and to a client: ip.src==10.0.0.3 && (http.cookie || http.set\_cookie)

**How it works...**

The following are illustrations of the IP and TCP header structures respectively. UDP is quite simple; it has only source and destination port numbers, length, and checksum. In the following diagram we see the IP header structure:



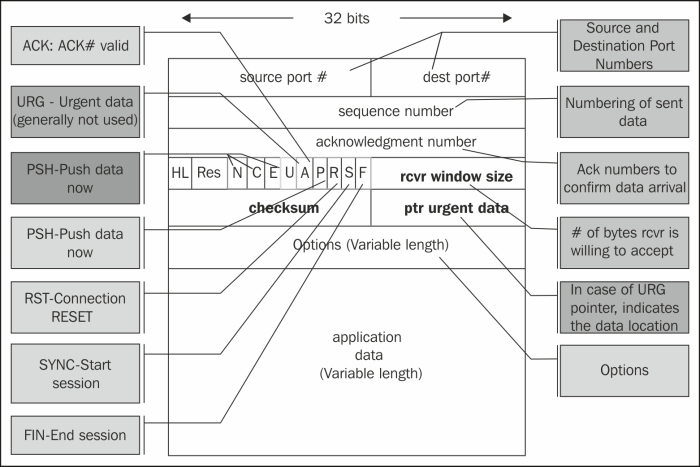
Some important factors in the IP packet are as follows:

* **Ver**: The version is either 4 or 6.
* **Header length (HL)**: The header length is 20 to 24 bytes, with options.
* **Type of Service (ToS)**: This is usually implemented with **Differentiated Services** (**DiffServ**) and provides priority to preferred services.

**Tip**

IP standard (RFC 791 from September 1981) has named this field Type of Service (ToS) and defined its structure. The standards for Differentiated Services were published later (RFCs 2474, 2475 from December 1998 and others) and are used for the implementation of the ToS byte in majority of the applications.

* **Length**: This field indicates the total datagram length in bytes.
* **16-bit identifier, flgs, and Fragment offset**: Every packet has it's own packet ID. When fragmented along with the flags and offset, these will enable the receiver to reassemble it.
* **Time to live (TTL)**: This starts with 64, 128, or 256 (depending on the operation system that sends the packet), when every router on the way decrements the value by one. This comes to prevent packets from traveling endlessly through the network. The router that sees 1 in the packet decrements it to 0 and drops the packet.
* **Upper layer**: This field consists of upper-layer protocols such as TCP, UDP and ICMP.
* **Checksum**: This field represents the packet checksum. The idea here is that the sender uses an error-checking mechanism to calculate a value over the packet. This value is set in the checksum field while the receiver of the packet calculates it again. If the sent value is not equal to the received value, it will be considered as a checksum error.
* **32-bit source and destination IP addresses**: As the names suggest, these are source and destination IP addresses.
* **Options**: This field is usually not in use in IPv4. In the following diagram you see the TCP header:



Some important factors in the TCP packet are as follows:

* **Source and destination ports**: These are the applications codes on either end.
* **Sequence number**: This field counts the bytes that the sender sends to the receiver.
* **Acknowledgement number**: This field indicates the ACK's received bytes. We will discuss this in detail in [Chapter 9](http://my.safaribooksonline.com/9781849517645/ch09_html), *UDP/TCP Analysis*.
* **HL**: This is the header length field and it indicates whether we use the **Options** field or not.
* **Res**: This field is reserved for future flags.
* **Flags**: This field indicates flags to start a connection (SYN), close a connection (FIN), reset a connection (RST), and push data for fast processing (PSH). We will discuss this in detail in [Chapter 9](http://my.safaribooksonline.com/9781849517645/ch09_html), *UDP/TCP Analysis*.
* **Rcvr window size**: This field indicates the buffer that the receiver has allocated to the process.
* **Checksum**: This field indicates the packet checksum.
* **Options**: Timestamps and receiver window enhancement (RFC 1323), and MSS extension. **Maximum Segment Size** (**MSS**) is the maximum side of the TCP payload. It is indicated in this field. Further discussion on this will be done in [Chapter 9](http://my.safaribooksonline.com/9781849517645/ch09_html), *UDP/TCP Analysis*.

**There's more...**

The TTL field in IP is quite a helpful field. While checking a TTL value, it explicitly indicates how many routers the packet has passed. Since operating system defaults are 64, 128, or 256, and the maximum number of hops that a packet will cross through the Internet are 30 (in private networks it is much less). For example, if we see a value of 120, the packet has passed 8 routers, and a value of 52 indicates that the packet has passed 12 routers.

**See also**

* For further information on the TCP/IP protocol stack, refer to [Chapter 9](http://my.safaribooksonline.com/9781849517645/ch09_html), *UDP/TCP Analysis*

**Configuring specific protocol filters**

In this recipe we will have a look at the instructions and examples to configure display filters for common protocols such as DNS, HTTP, and FTP.

The purpose of this recipe is to learn how to configure filters that will help us in network troubleshooting. We will learn about network troubleshooting in the upcoming chapters.

**Getting ready**

To perform this recipe, you will need a running Wireshark software capture; there are no other prerequisites.

**How to do it...**

In this recipe we will see the display filters for some common protocols.

**HTTP display filters**

The following are some common HTTP display filters:

* To display all the HTTP packets going to <"host name">, use http.request.method == <"Request methods">
* To display packets with the HTTP GET method, use http.request.method == "GET"
* To display the URI requested by client, use http.request.method == <"Full request URI">; for example, http.request.uri == "/v2/rating/mail.google.com"
* To display the URI requested by the client that contains a specific string (all requests to Google in the preceding example), use http.request.uri contains "URI String"; for example, http.request.uri contains "mail.google.com"
* To check all the cookie requests sent over the network (note that cookies are always sent from the client to the server), use http.cookie
* To check all the cookie set commands sent from the server to the client, use http.set\_cookie
* To check all the cookies sent by Google servers to your PC, use (http.set\_cookie) && (http contains "google")
* To check all the HTTP packets that contain a ZIP file, use http matches "\.zip" && http.request.method == "GET"

**DNS display filters**

Here, we will look at some common DNS display filters.

To display DNS queries and responses, use:

* dns.flags.response == 0 for DNS queries
* dns.flags.response == 1 for DNS response

To display only DNS responses with four answers or more, use dns.count.answers >= 4.

**FTP display filters**

Some common FTP display filters are as follows:

* To fetch FTP request commands, use ftp.request.command == <"requested command"> - ftp.request.command == "USER"
* To fetch FTP commands from port 2, use ftp, and to fetch FTP data from port 20 or any other configured port, use ftp-data

**How it works...**

The Wireshark regular expression syntax for display filters uses the same syntax as regular expressions in Perl.

Some common modifiers are as follows:

* ^: This is used to match the beginning of the line
* $: This is used to match the end of the line
* |: This is used for alternation purposes
* (): This is used for grouping purposes
* \*: This is used to match either 0 or more times
* +: This is used to match 1 or more times
* ?: This is used to match 1 or 0 times
* {n}: This is used to match exactly n times
* {n,}: This is used to match at least n times
* {n,m}: This is used to match at least n but not more than m times

You can use these modifiers to configure more complex filters. Have a look at the following examples:

* To look for HTTP GET commands that contain ZIP files, use http.request.method == "GET" && http matches "\.zip" && !(http.accept\_encoding == "gzip, deflate")
* To look for HTTP GET commands that contain ZIP files, use http.request.method == "GET" && http matches "\.zip" && !(http.accept\_encoding == "gzip, deflate")
* To look for HTTP messages that contain websites that end with .com, use http.host matches "\.com$"

**See also**

* The Perl regular expression syntax list can be found at <http://www.pcre.org/>, and the manual pages can be found at <http://perldoc.perl.org/perlre.html>

**Configuring substring operator filters**

Offset filters are filters in which you actually say, "Go to field x in the protocol header and check if the next y bytes equal to….".

These filters can be used in many cases in which a known string byte appears somewhere in the packet and you want to display packets that contain it.

**Getting ready**

To step through this recipe, you will need a running Wireshark software and a running capture; there are no other prerequisites. The general representation for offset filters is:

Protocols[x:y] == <value>

Here, x refers to the bytes from the beginning of the header and y refers to the number of bytes to check.

**How to do it...**

Examples for filters that use substring operators are as follows:

* **Packets to IPv4 multicast addresses**: eth.dst[0:3] == 01:00:5e (RFC 1112, section 6.4 allocates the MAC address space of 01-00-5E-00-00-00 to 01-00-5E-FF-FF-FF for multicast addressing)
* **Packets to IPv6 multicast addresses**: eth.dst[0:3] == 33:33:00 (RFC 2464, section 7 allocates the MAC address space that starts with 33-33 for multicast addressing)

**How it works...**

Wireshark enables you to look into protocols and search for specific bytes in it. This is specifically practical for well-known strings in protocols, such as Ethernet in the given example.

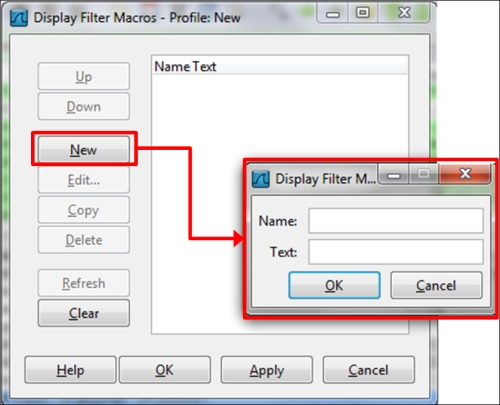
**Configuring macros**

Display filter macros are used to create shortcuts for complex display filters, which you can configure once and use later.

**Getting ready**

To configure display filter macros, navigate to **Analyze** | **Display Filter Macros** | **New**.

You will get the following window:



**How to do it...**

1. In order to configure a macro, you give it a name and fill the textbox with the filter string.
2. In order to activate the macro, you simply write $(macro\_name:parameter1;paramater2;parameter3 …).
3. Let's configure a simple filter name, test01, which takes the following parameters as values:
   * ip.src == <value>
   * tcp.dstport == <value>

This will be a filter that looks for packets from a specific source network that go out to the HTTP port.

A macro that takes these two parameters will be: ip.src==$1 && tcp.dstport==$2.

1. Now, in order to get the filter results for the parameters ip.src == 10.0.0.4 and tcp.dstport == 80, we should write the string ${test01:10.0.0.4;80} in the display window bar.

**How it works...**

Macros work in a simple way; you write a filter string with the sign $ ahead of every positional parameter. While running the macros, it will accept the parameters in order.

