Wireshark Cheat Sheet

Contents

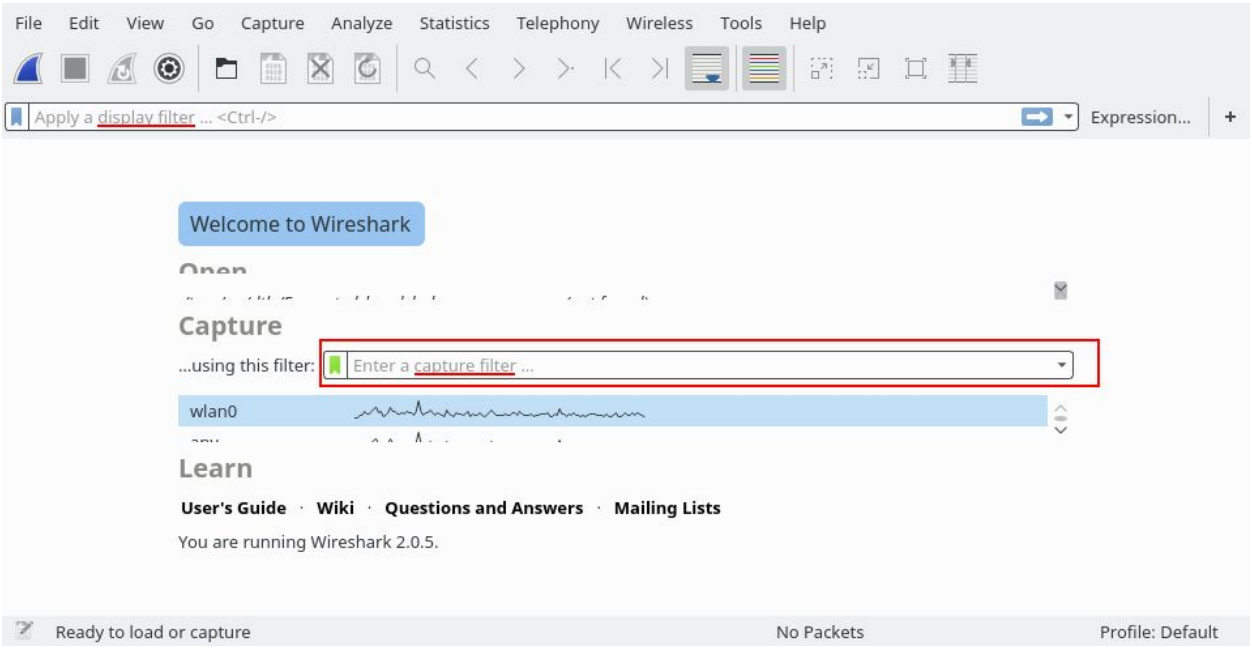
Wireshark Capture Filters Overview

Capture filter is not a display filter

Capture filters (like `tcp port 80`) are not to be confused with display filters (like `tcp.port == 80`). The former are much more limited and are used to reduce the size of a raw packet capture. The latter are used to hide some packets from the packet list.

Capture filters are set before starting a packet capture and cannot be modified during the capture. Display filters on the other hand do not have this limitation and you can change them on the fly.

In the main window, one can find the capture filter just above the interfaces list and in the interfaces dialog. The display filter can be changed above the packet list as can be seen in this picture:



Examples

Capture only traffic to or from IP address 172.18.5.4:

- `host 172.18.5.4`

Capture traffic to or from a range of IP addresses:

- `net 192.168.0.0/24`

or

- `net 192.168.0.0 mask 255.255.255.0`

Capture traffic from a range of IP addresses:

- `src net 192.168.0.0/24`

or

- `src net 192.168.0.0 mask 255.255.255.0`

Capture traffic to a range of IP addresses:

- `dst net 192.168.0.0/24`

or

- `dst net 192.168.0.0 mask 255.255.255.0`

Capture only DNS (port 53) traffic:

- `port 53`

Capture non-HTTP and non-SMTP traffic on your server (both are equivalent):

- `host www.example.com and not (port 80 or port 25)`

```
host www.example.com and not port 80 and not port 25
```

Capture except all ARP and DNS traffic:

- `port not 53 and not arp`

Capture traffic within a range of ports

- `(tcp[0:2] > 1500 and tcp[0:2] < 1550) or (tcp[2:2] > 1500 and tcp[2:2] < 1550)`

or, with newer versions of libpcap (0.9.1 and later):

- `tcp portrange 1501-1549`

Capture only Ethernet type EAPOL:

- `ether proto 0x888e`

Reject ethernet frames towards the Link Layer Discovery Protocol Multicast group:

- `not ether dst 01:80:c2:00:00:0e`

Capture only IPv4 traffic - the shortest filter, but sometimes very useful to get rid of lower layer protocols like ARP and STP:

- `ip`

Capture only unicast traffic - useful to get rid of noise on the network if you only want to see traffic to and from your machine, not, for example, broadcast and multicast announcements:

- `not broadcast and not multicast`

Capture IPv6 "all nodes" (router and neighbor advertisement) traffic. Can be used to find rogue RAs:

- `dst host ff02::1`

Capture HTTP GET requests. This looks for the bytes 'G', 'E', 'T', and ' ' (hex values 47, 45, 54, and 20) just after the TCP header. "tcp[12:1] & 0xf0) >> 2" figures out the TCP header length.

- `port 80 and tcp[((tcp[12:1] & 0xf0) >> 2):4] = 0x47455420`

Useful Filters

Blaster and Welchia are RPC worms. (Does anyone have better links, i.e. ones that describe or show the actual payload?)

Blaster worm:

- `dst port 135 and tcp port 135 and ip[2:2]==48`

Welchia worm:

- `icmp[icmptype]==icmp-echo and ip[2:2]==92 and icmp[8:4]==0xAAAAAAAA`

The filter looks for an icmp echo request that is 92 bytes long and has an icmp payload that begins with 4 bytes of A's (hex). It is the signature of the welchia worm just before it tries to compromise a system.

Many worms try to spread by contacting other hosts on ports 135, 445, or 1433. This filter is independent of the specific worm instead it looks for SYN packets originating from a local network on those specific ports. Please change the network filter to reflect your own network.

```
dst port 135 or dst port 445 or dst port 1433  and tcp[tcpflags] & (tcp-syn) != 0
and tcp[tcpflags] & (tcp-ack) = 0 and src net 192.168.0.0/24
```

Heartbleed Exploit:

```
tcp src port 443 and (tcp[((tcp[12] & 0xF0) >> 4 ) * 4] = 0x18) and
(tcp[((tcp[12] & 0xF0) >> 4 ) * 4 + 1] = 0x03) and (tcp[((tcp[12] & 0xF0) >> 4
) * 4 + 2] < 0x04) and ((ip[2:2] - 4 * (ip[0] & 0x0F)  - 4 * ((tcp[12] & 0xF0)
>> 4) > 69))
```

Default Capture Filters

Wireshark tries to determine if it's running remotely (e.g. via SSH or Remote Desktop), and if so sets a default capture filter that should block out the remote session traffic. It does this by checking environment variables in the following order:

Environment Variable	Resultant Filter
SSH_CONNECTION	not (tcp port *srcport* and *addr_family* host *srchost* and tcp port *dstport* and *addr_family* host *dsthost*)
SSH_CLIENT	not (tcp port *srcport* and *addr_family* host *srchost* and tcp port *dstport*)
REMOTEHOST	not *addr_family* host *host*
DISPLAY	not *addr_family* host *host*
CLIENTNAME	not tcp port 3389

(*addr_family* will either be "ip" or "ip6")

Discussion

BTW, the Symantec page says that Blaster probes 135/tcp, 4444/tcp, and 69/udp. Would

```
(tcp dst port 135 or tcp dst port 4444 or udp dst port 69) and ip[2:2]==48
```

- be a better filter? - *Gerald Combs*

Q: What is a good filter for just capturing SIP and RTP packets?

A: On most systems, for SIP traffic to the standard SIP port 5060,

```
tcp port sip
```

should capture TCP traffic to and from that port,

```
udp port sip
```

should capture UDP traffic to and from that port, and

```
port sip
```

should capture both TCP and UDP traffic to and from that port (if one of those filters gets "parse error", try using `5060` instead of `sip`). For SIP traffic to and from other ports, use that port number rather than `sip`.

In most cases RTP port numbers are dynamically assigned. You can use something like the following which limits the capture to UDP, even source and destination ports, a valid RTP version, and small packets. It will capture any non-RTP traffic that happens to match the filter (such as DNS) but it will capture all RTP packets in many environments.

```
udp[1] & 1 != 1 && udp[3] & 1 != 1 && udp[8] & 0x80 == 0x80 && length < 250
```

Capture WLAN traffic without Beacons:

```
link[0] != 0x80
```

Capture all traffic originating (source) in the IP range 192.168.XXX.XXX:

```
src net 192.168
```

Capture PPPoE traffic:

```
pppoes
pppoes and (host 192.168.0.0 and port 80)
```

Capture VLAN traffic:

```
vlan
vlan and (host 192.168.0.0 and port 80)
```

Wireshark Display Filters Overview

Display filter is not a capture filter

Capture filters (like `tcp port 80`) are not to be confused with display filters (like `tcp.port == 80`).

Examples

Show only SMTP (port 25) and ICMP traffic:

```
tcp.port eq 25 or icmp
```

Show only traffic in the LAN (192.168.x.x), between workstations and servers -- no Internet:

```
ip.src==192.168.0.0/16 and ip.dst==192.168.0.0/16
```

TCP buffer full -- *Source is instructing Destination to stop sending data*

```
tcp.window_size == 0 && tcp.flags.reset != 1
```

Filter on Windows -- *Filter out noise, while watching Windows Client - DC exchanges*

```
smb || nbns || dcerpc || nbss || dns
```

Sasser worm: *--What sasser really did--*

```
ls_ads.opnum==0x09
```

Match packets containing the (arbitrary) 3-byte sequence 0x81, 0x60, 0x03 at the beginning of the UDP payload, skipping the 8-byte UDP header. Note that the values for the byte sequence implicitly are in hexadecimal only. *(Useful for matching homegrown packet protocols.)*

```
udp[8:3]==81:60:03
```

The "slice" feature is also useful to filter on the vendor identifier part (OUI) of the MAC address, see the Ethernet page for details. Thus, you may restrict the display to only packets from a specific device manufacturer. E.g. for DELL machines only:

```
eth.addr[0:3]==00:06:5B
```

It is also possible to search for characters appearing anywhere in a field or protocol by using the contains operator.

Match packets that contains the 3-byte sequence 0x81, 0x60, 0x03 anywhere in the UDP header or payload:

```
udp contains 81:60:03
```

Match packets where SIP To-header contains the string "a1762" anywhere in the header:

```
sip.To contains "a1762"
```

The matches, or ~, operator makes it possible to search for text in string fields and byte sequences using a regular expression, using Perl regular expression syntax. Note: Wireshark needs to be built with libpcre in order to be able to use the matches operator.

Match HTTP requests where the last characters in the uri are the characters "gl=se":

- `http.request.uri matches "gl=se$"`

Note: The $ character is a PCRE punctuation character that matches the end of a string, in this case the end of http.request.uri field.

Filter by a protocol (e.g. SIP) and filter out unwanted IPs:

```
ip.src != xxx.xxx.xxx.xxx && ip.dst != xxx.xxx.xxx.xxx && sip
```

Gotchas

Some *filter fields* match against multiple *protocol fields*. For example, "ip.addr" matches against both the IP source and destination addresses in the IP header. The same is true for "tcp.port", "udp.port", "eth.addr", and others. It's important to note that

- `ip.addr == 10.43.54.65`

is equivalent to

```
ip.src == 10.43.54.65 or ip.dst == 10.43.54.65
```

This can be counterintuitive in some cases. Suppose we want to filter out any traffic to or from 10.43.54.65. We might try the following:

- `ip.addr != 10.43.54.65`

which is equivalent to

```
ip.src != 10.43.54.65 or ip.dst != 10.43.54.65
```

This translates to "pass all traffic except for traffic with a source IPv4 address of 10.43.54.65 **and** a destination IPv4 address of 10.43.54.65", which isn't what we wanted.

Instead we need to negate the expression, like so:

- `! ( ip.addr == 10.43.54.65 )`

which is equivalent to

```
! (ip.src == 10.43.54.65 or ip.dst == 10.43.54.65)
```

This translates to "pass any traffic except with a source IPv4 address of 10.43.54.65 **or** a destination IPv4 address of 10.43.54.65", which is what we wanted.

This can also happen if, for example, you have tunneled protocols, so that you might have two separate IPv4 or IPv6 layers and two separate IPv4 or IPv6 headers, or if you have multiple instances of a field for other reasons, such as multiple IPv6 "next header" fields.

If you have a filter expression of the form *name op value*, where *name* is the name of a field, *op* is a comparison operator such as `==` or `!=` or `<`or..., and *value* is a value against which you're comparing, it should be thought of as meaning "match a packet if there is *at least one* instance of the field named *name* whose value is (equal to, not equal to, less than, ...) *value*".

The negation of that is "match a packet if there are *no*instances of the field named *name* whose value is (equal to, not equal to, less than, ...) *value*"; simply negating *op*, e.g.

replacing == with != or <with >=, give you another "if there is at least one" check, which is not the negation of the original check.