NAME

wireshark-filter − Wireshark display filter syntax and reference

SYNOPSIS

wireshark [other options] [**−Y** "display filter expression" | **−−display−filter** "display filter expression"]

tshark [other options] [**−Y** "display filter expression" | **−−display−filter** "display filter expression"]

DESCRIPTION

Wireshark and **TShark** share a powerful filter engine that helps remove the noise from a packet trace and lets you see only the packets that interest you. If a packet meets the requirements expressed in your filter, then it is displayed in the list of packets. Display filters let you compare the fields within a protocol against a specific value, compare fields against fields, and check the existence of specified fields or protocols.

Filters are also used by other features such as statistics generation and packet list colorization (the latter is only available to **Wireshark**). This manual page describes their syntax. A comprehensive reference of filter fields can be found within Wireshark and in the display filter reference at https://www.wireshark.org/docs/dfref/.

FILTER SYNTAX

Check whether a field or protocol exists

The simplest filter allows you to check for the existence of a protocol or field. If you want to see all packets which contain the IP protocol, the filter would be "ip" (without the quotation marks). To see all packets that contain a Token−Ring RIF field, use "tr.rif".

Think of a protocol or field in a filter as implicitly having the "exists" operator.

Comparison operators

Fields can also be compared against values. The comparison operators can be expressed either through English−like abbreviations or through C−like symbols:

```
eq, ==    Equal
ne, !=    Not Equal
gt, >     Greater Than
lt, <     Less Than
ge, >=    Greater than or Equal to
le, <=    Less than or Equal to
```

Search and match operators

Additional operators exist expressed only in English, not C−like syntax:

```
contains     Does the protocol, field or slice contain a value
matches, ~   Does the protocol or text string match the given
             case−insensitive Perl−compatible regular expression
```

The "contains" operator allows a filter to search for a sequence of characters, expressed as a string (quoted or unquoted), or bytes, expressed as a byte array, or for a single character, expressed as a C−style character constant. For example, to search for a given HTTP URL in a capture, the following filter can be used:

```
http contains "https://www.wireshark.org"
```

The "contains" operator cannot be used on atomic fields, such as numbers or IP addresses.

The "matches" or "~" operator allows a filter to apply to a specified Perl−compatible regular expression (PCRE). The "matches" operator is only implemented for protocols and for protocol fields with a text string representation. Matches are case−insensitive by default. For example, to search for a given WAP WSP User−Agent, you can write:

```
wsp.header.user_agent matches "cldc"
```

This would match "cldc", "CLDC", "cLdC" or any other combination of upper and lower case letters.

You can force case sensitivity using

```
wsp.header.user_agent matches "(?−i)cldc"
```

This is an example of PCRE’s **(?*option)*** construct. **(?−i)** performs a case−sensitive pattern match but other options can be specified as well. More information can be found in the pcrepattern(3)|https://www.pcre.org/original/doc/html/pcrepattern.html man page.

Functions

The filter language has the following functions:

```
upper(string−field) − converts a string field to uppercase
lower(string−field) − converts a string field to lowercase
len(field)          − returns the byte length of a string or bytes field
count(field)        − returns the number of field occurrences in a frame
string(field)       − converts a non−string field to string
```

upper() and lower() are useful for performing case−insensitive string comparisons. For example:

```
upper(ncp.nds_stream_name) contains "MACRO"
lower(mount.dump.hostname) == "angel"
```

string() converts a field value to a string, suitable for use with operators like "matches" or "contains". Integer fields are converted to their decimal representation. It can be used with IP/Ethernet addresses (as well as others), but not with string or byte fields. For example:

```
string(frame.number) matches "[13579]$"
```

gives you all the odd packets.

Protocol field types

Each protocol field is typed. The types are:

```
ASN.1 object identifier
Boolean
Character string
Compiled Perl−Compatible Regular Expression (GRegex) object
Date and time
Ethernet or other MAC address
EUI64 address
Floating point (double−precision)
Floating point (single−precision)
Frame number
Globally Unique Identifier
IPv4 address
IPv6 address
IPX network number
Label
Protocol
Sequence of bytes
Signed integer, 1, 2, 3, 4, or 8 bytes
Time offset
```

```
Unsigned integer, 1, 2, 3, 4, or 8 bytes
1−byte ASCII character
```

An integer may be expressed in decimal, octal, or hexadecimal notation, or as a C−style character constant. The following six display filters are equivalent:

```
frame.len > 10
frame.len > 012
frame.len > 0xa
frame.len > '\n'
frame.len > '\x0a'
frame.len > '\012'
```

Boolean values are either true or false. In a display filter expression testing the value of a Boolean field, "true" is expressed as 1 or any other non−zero value, and "false" is expressed as zero. For example, a token−ring packet's source route field is Boolean. To find any source−routed packets, a display filter would be:

```
tr.sr == 1
```

Non source−routed packets can be found with:

```
tr.sr == 0
```

Ethernet addresses and byte arrays are represented by hex digits. The hex digits may be separated by colons, periods, or hyphens:

```
eth.dst eq ff:ff:ff:ff:ff:ff
aim.data == 0.1.0.d
fddi.src == aa−aa−aa−aa−aa−aa
echo.data == 7a
```

IPv4 addresses can be represented in either dotted decimal notation or by using the hostname:

```
ip.src == 192.168.1.1
ip.dst eq www.mit.edu
```

IPv4 addresses can be compared with the same logical relations as numbers: eq, ne, gt, ge, lt, and le. The IPv4 address is stored in host order, so you do not have to worry about the endianness of an IPv4 address when using it in a display filter.

Classless Inter−Domain Routing (CIDR) notation can be used to test if an IPv4 address is in a certain subnet. For example, this display filter will find all packets in the 129.111 network:

```
ip.addr == 129.111.0.0/16
```

Remember, the number after the slash represents the number of bits used to represent the network. CIDR notation can also be used with hostnames, as in this example of finding IP addresses on the same network as 'sneezy' (requires that 'sneezy' resolve to an IP address for filter to be valid):

```
ip.addr eq sneezy/24
```

The CIDR notation can only be used on IP addresses or hostnames, not in variable names. So, a display filter like "ip.src/24 == ip.dst/24" is not valid (yet).

Transaction and other IDs are often represented by unsigned 16 or 32 bit integers and formatted as a hexadecimal string with "0x" prefix:

```
(dhcp.id == 0xfe089c15) || (ip.id == 0x0373)
```

Strings are enclosed in double quotes:

```
http.request.method == "POST"
```

Inside double quotes, you may use a backslash to embed a double quote or an arbitrary byte represented in either octal or hexadecimal.

```
browser.comment == "An embedded \" double−quote"
```

Use of hexadecimal to look for "HEAD":

```
http.request.method == "\x48EAD"
```

Use of octal to look for "HEAD":

```
http.request.method == "\110EAD"
```

This means that you must escape backslashes with backslashes inside double quotes.

```
smb.path contains "\\\\SERVER\\SHARE"
```

looks for \\SERVER\SHARE in "smb.path". This may be more conveniently written as

```
smb.path contains r"\\SERVER\SHARE"
```

String literals prefixed with 'r' are called "raw strings". Such strings treat backslash as a literal character. Double quotes may still be escaped with backslash but note that backslashes are always preserved in the result.

The slice operator

You can take a slice of a field if the field is a text string or a byte array. For example, you can filter on the vendor portion of an ethernet address (the first three bytes) like this:

```
eth.src[0:3] == 00:00:83
```

Another example is:

```
http.content_type[0:4] == "text"
```

You can use the slice operator on a protocol name, too. The "frame" protocol can be useful, encompassing all the data captured by **Wireshark** or **TShark**.

```
token[0:5] ne 0.0.0.1.1
llc[0] eq aa
frame[100−199] contains "wireshark"
```

The following syntax governs slices:

```
[i:j]    i = start_offset, j = length
[i−j]    i = start_offset, j = end_offset, inclusive.
```

```
[i]      i = start_offset, length = 1
[:j]     start_offset = 0, length = j
[i:]     start_offset = i, end_offset = end_of_field
```

Offsets can be negative, in which case they indicate the offset from the **end** of the field. The last byte of the field is at offset −1, the last but one byte is at offset −2, and so on. Here's how to check the last four bytes of a frame:

```
frame[−4:4] == 0.1.2.3
```

or

```
frame[−4:] == 0.1.2.3
```

A slice is always compared against either a string or a byte sequence. As a special case, when the slice is only 1 byte wide, you can compare it against a hex integer that is 0xff or less (which means it fits inside one byte). This is not allowed for byte sequences greater than one byte, because then one would need to specify the endianness of the multi−byte integer. Also, this is not allowed for decimal numbers, since they would be confused with hex numbers that are already allowed as byte strings. Nevertheless, single−byte hex integers can be convenient:

```
frame[4] == 0xff
```

Slices can be combined. You can concatenate them using the comma operator:

```
ftp[1,3−5,9:] == 01:03:04:05:09:0a:0b
```

This concatenates offset 1, offsets 3−5, and offset 9 to the end of the ftp data.

The membership operator

A field may be checked for matches against a set of values simply with the membership operator. For instance, you may find traffic on common HTTP/HTTPS ports with the following filter:

```
tcp.port in {80, 443, 8080}
```

as opposed to the more verbose:

```
tcp.port == 80 or tcp.port == 443 or tcp.port == 8080
```

To find HTTP requests using the HEAD or GET methods:

```
http.request.method in {"HEAD", "GET"}
```

The set of values can also contain ranges:

```
tcp.port in {443, 4430..4434}
ip.addr in {10.0.0.5 .. 10.0.0.9, 192.168.1.1..192.168.1.9}
frame.time_delta in {10 .. 10.5}
```

Type conversions

If a field is a text string or a byte array, it can be expressed in whichever way is most convenient.

So, for instance, the following filters are equivalent:

```
http.request.method == "GET"
http.request.method == 47.45.54
```

A range can also be expressed in either way:

```
frame[60:2] gt 50.51
frame[60:2] gt "PQ"
```

Bit field operations

It is also possible to define tests with bit field operations. Currently the following bit field operation is supported:

```
bitwise_and, &        Bitwise AND
```

The bitwise AND operation allows testing to see if one or more bits are set. Bitwise AND operates on integer protocol fields and slices.

When testing for TCP SYN packets, you can write:

```
tcp.flags & 0x02
```

That expression will match all packets that contain a "tcp.flags" field with the 0x02 bit, i.e. the SYN bit, set.

Similarly, filtering for all WSP GET and extended GET methods is achieved with:

```
wsp.pdu_type & 0x40
```

When using slices, the bit mask must be specified as a byte string, and it must have the same number of bytes as the slice itself, as in:

```
ip[42:2] & 40:ff
```

Logical expressions

Tests can be combined using logical expressions. These too are expressible in C−like syntax or with English−like abbreviations:

```
and, &&   Logical AND
or,  ||   Logical OR
not, ! Logical NOT
```

Expressions can be grouped by parentheses as well. The following are all valid display filter expressions:

```
tcp.port == 80 and ip.src == 192.168.2.1
not llc
http and frame[100−199] contains "wireshark"
(ipx.src.net == 0xbad && ipx.src.node == 0.0.0.0.0.1) || ip
```

Remember that whenever a protocol or field name occurs in an expression, the "exists" operator is implicitly called. The "exists" operator has the highest priority. This means that the first filter expression must be read as "show me the packets for which tcp.port exists and equals 80, and ip.src exists and equals 192.168.2.1". The second filter expression means "show me the packets where not exists llc", or in other words "where llc does not exist" and hence will match all packets that do not contain the llc protocol. The third filter expression includes the constraint that offset 199 in the frame exists, in other words the length of the frame is at least 200.

Each comparison has an implicit exists test for any field value. Care must be taken when using the display filter to remove noise from the packet trace. If, for example, you want to filter out all IP multicast packets to address 224.1.2.3, then using:

```
ip.dst ne 224.1.2.3
```

may be too restrictive. This is the same as writing:

```
ip.dst and ip.dst ne 224.1.2.3
```

The filter selects only frames that have the "ip.dst" field. Any other frames, including all non−IP packets, will not be displayed. To display the non−IP packets as well, you can use one of the following two expressions:

```
not ip.dst or ip.dst ne 224.1.2.3
not ip.dst eq 224.1.2.3
```

The first filter uses "not ip.dst" to include all non−IP packets and then lets "ip.dst ne 224.1.2.3" filter out the unwanted IP packets. The second filter also negates the implicit existance test and so is a shorter way to write the first.

FILTER FIELD REFERENCE

The entire list of display filters is too large to list here. You can can find references and examples at the following locations:

- The online Display Filter Reference: https://www.wireshark.org/docs/dfref/
- *View:Internals:Supported Protocols* in Wireshark
- `tshark -G fields`on the command line
- The Wireshark wiki: https://gitlab.com/wireshark/wireshark/−/wikis/DisplayFilters

NOTES

The **wireshark−filter(4)** manpage is part of the **Wireshark** distribution. The latest version of **Wireshark** can be found at https://www.wireshark.org.

Regular expressions in the "matches" operator are provided by GRegex in GLib. See https://developer−old.gnome.org/glib/stable/glib−regex−syntax.html or https://www.pcre.org/ for more information.

This manpage does not describe the capture filter syntax, which is different. See the manual page of pcap−filter(7) or, if that doesn't exist, tcpdump(8), or, if that doesn't exist, https://gitlab.com/wireshark/wireshark/−/wikis/CaptureFilters for a description of capture filters.

Display Filters are also described in the User's Guide: https://www.wireshark.org/docs/wsug_html_chunked/ChWorkBuildDisplayFilterSection.html

SEE ALSO

wireshark(1), tshark(1), editcap(1), pcap(3), pcap−filter(7) or tcpdump(8) if it doesn't exist.

AUTHORS

See the list of authors in the **Wireshark** man page for a list of authors of that code.