wireshark-filter(4) Manual Page

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".

Whenever a protocol or field appears as the argument of a function in a filter, an exists operator for that protocol or field implicitly appears.

Values and operators

Each field has a value, and that value can be used in operations with comparable values (which may be literals, other fields, or function results). The value of a field is not necessarily what appears in the **Wireshark** display or **TShark** output. For example, a protocol is semantically equivalent to the sequence of bytes that it spans, not its displayed text in the protocol tree.

Comparison operators

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
```

The ordering depends on the value type in the usual way (e.g., lexicographic for strings and arithmetic for integers.) A field may appear more than once in a given frame. In that case equality can be strict (all fields must match the condition) or not (any field must match the condition). The inequality is the logical negation of equality. The following table contains all equality operators, their aliases and meaning:

```
eq, any_eq, ==    Any field must be equal
ne, all_ne, !=    All fields must be not equal
    all_eq, ===   All fields must be equal
    any_ne, !==   Any fields must be not equal
```

The operators "any" or "all" can be used with any comparison operator to make the test match any or all fields:

```
all tcp.port > 1024
```

```
any ip.addr != 1.1.1.1
```

The "any" and "all" modifiers take precedence over comparison operators such as "===" and "any_eq".

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 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, or bytes, expressed as a byte array. The type of the left hand side of the "contains" operator must be comparable to that of the right hand side after any implicit or explicit conversions.

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 (PCRE2). The regular expression must be a double quoted string. The left hand side of the "matches" operator must be a string, which can be a non-stringlike field implicitly or explicitly converted to a string. 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 PCRE2'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 pcre2pattern(3)|https://www.pcre.org/current/doc/html/pcre2pattern.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
vals(field)         - converts a field value to its value string
dec(field)          - converts an unsigned integer to a decimal string
hex(field)          - converts an unsigned integer to a hexadecimal string
max(f1,...,fn)      - return the maximum value
min(f1,...,fn)      - return the minimum value
abs(field)          - return the absolute value of numeric fields
```

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. Note that the "matches" operator implicitly converts types of their value string representation; to match against the decimal representation of an integer field use string().

vals() converts an integer or boolean field value to a string using the field's associated value string, if it has one. This produces strings similar to those seen in custom columns. The resultant string can also be used with other operators. E.g.:

```
vals(pfcp.msg_type) contains "Request"
```

would match all packets which have a PFCP request, even if that request is not matched with a response.

dec() and hex() convert unsigned integer fields to decimal or hexadecimal representation. Currently dec() and string() give same result for an unsigned integer, but it is possible that in the future string() will use the native base of the field.

max() and min() take any number of arguments and returns one value, respectively the largest/smallest. The arguments must all have the same type.

There is also a set of functions to test IP addresses:

```
ip_special_name(ip)       - Returns the IP special-purpose block name as a string
ip_special_mask(ip)       - Returns the IP special-purpose block flags as a mask. The bits are:
                                 4 3 2 1 0
                                 ---------
                                 S D F G R
                           S = Source, D = Destination, F = Forwardable, G = Globally-reachable, R =
Reserved-by-protocol


ip_linklocal(ip)          - true if the IPv4 or IPv6 address is link-local
ip_multicast(ip)          - true if the IPv4 or IPv6 address is multicast
ip_rfc1918(ipv4)          - true if the IPv4 address is private-use (from the allocation in RFC 1918)
ip_ula(ipv6)              - true if the IPv6 address is unique-local (ULA) as in RFC 4193
```

Macros

It is possible to define display filter macros. Macro are names that are replaced with the associated expression, possibly performing argument substitution. Macro expansions are purely textual replacements and performed recursively before compilation. They allow replacing long and often used expressions with easy to use names.

Macros are defined using the GUI or directly in the "dmacros" configuration file. For example the definition

```
"addplusone" {$1 + $2 + 1}
```

creates a macro called `addplusone` that takes two arguments and expands to the given expression. Arguments in the replacement expression are given using the dollar sign.

Macros are invoked like function but preceded with a dollar sign (sometimes also called a sigil):

```
$addplusone(udp.src_port,udp.dst_port)
```

results in the expression

```
{udp.src_port + udp.dst_port + 1}
```

after argument substitution. There is an older alternative notation to invoke macros:

```
${addplusone:udp.src_port;udp.dst_port}
```

or

```
${addplusone;udp.src_port;udp.dst_port}
```

Both forms are equivalent and can be used interchangibly as a matter of preference.

Protocol field types

Each protocol field is typed. The types are:

```
ASN.1 object identifier, plain or relative
AX.25 address
Boolean
Byte sequence
Character string
Character, 1 byte
Date and time
Ethernet or other MAC address
EUI64 address
Fibre Channel WWN
Floating point, single or double precision
Frame number
Globally Unique Identifier
IEEE-11073 floating point, 16 or 32 bits
IPv4 address
IPv6 address
IPX network number
Label
OSI System-ID
Protocol
Signed integer, 1, 2, 3, 4, or 8 bytes
Time offset
Unsigned integer, 1, 2, 3, 4, or 8 bytes
VINES address
```

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

```
frame.len > 10
frame.len > 012
frame.len > 0xa
frame.len > 0b1010
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 the word `true` (case-insensitive) or any non-zero number. False is expressed as `false` (case-insensitive) or the number zero. For example, a token-ring packet's source route field is Boolean. To find any source-routed packets, a display filter would be any of the following:

```
tr.sr == 1
tr.sr == true
tr.sr == TRUE
```

Non source-routed packets can be found with:

```
tr.sr == 0
tr.sr == false
tr.sr == FALSE
```

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 following table lists all escape sequences supported with strings and character constants:

```
\'          single quote
\"          double quote
\\          backslash
\a          audible bell
\b          backspace
\f          form feed
\n          line feed
\r          carriage return
\t          horizontal tab
\v          vertical tab
\NNN        arbitrary octal value
\xNN        arbitrary hexadecimal value
\uNNNN      Unicode codepoint U+NNNN
\UNNNNNNNN  Unicode codepoint U+NNNNNNNN
```

Date and time values can be given in ISO 8601 format or using a legacy month-year-time format:

```
"2020-07-04T12:34:56"
"Sep 26, 2004 23:18:04.954975"
```

The 'T' separator in ISO 8601 can be omitted. The timezone can be given as "Z" or an offset from UTC.

When not using ISO 8601 the timezone can be given as the strings "UTC", "GMT" or "UT" for UTC or also given as an offset from UTC, plus some North American and Nautical/Military designations (see the specification for %z in strptime(3) (https://man.netbsd.org/strptime.3)). Note that arbitrary timezone names are not supported however.

If the timezone is omitted then date and time values are interpreted as local time.

The slice operator

You can take a slice of a field if the field base type is a text string or a byte array (the base type of most network address fields is bytes). 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
```

Slice indexes are zero-based. 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
```

As mentioned above the slice operator can be used on string and byte fields and will respectively produce string or byte slices. String slices are indexed on UTF-8 codepoint boundaries (i.e: internationalized characters), so the following comparison is true:

```
"touché"[5] == "é"
```

The example above generates an error because the compiler rejects constant expressions but is otherwise syntactically correct and exemplifies the behaviour of string slices.

To obtain a byte slice of the same string the raw (@) operator can be used:

```
@"touché"[5-6] == c3:a9
```

A slice can always be compared against either a string or a byte sequence.

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 layer operator

A field can be restricted to a certain layer in the protocol stack using the layer operator (#), followed by a decimal number:

```
ip.addr#2 == 192.168.30.40
```

matches only the inner (second) layer in the packet. Layers use simple stacking semantics and protocol layers are counted sequentially starting from 1. For example, in a packet that contains two IPv4 headers, the outer (first) source address can be matched with "ip.src#1" and the inner (second) source address can be matched with "ip.src#2".

For more complicated ranges the same syntax used with slices is valid:

```
tcp.port#[2-4]
```

means layers number 2, 3 or 4 inclusive. The hash symbol is required to distinguish a layer range from a slice.

The at operator

By prefixing the field name with an at sign (@) the comparison is done against the raw packet data for the field.

A character string must be decoded from a source encoding during dissection. If there are decoding errors the resulting string will usually contain replacement characters:

```
browser.comment == "string is ␦␦␦␦"
```

The at operator allows testing the raw undecoded data:

```
@browser.comment == 73:74:72:69:6e:67:20:69:73:20:aa:aa:aa:aa
```

The syntactical rules for a bytes field type apply to the second example.

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}
```

Implicit type conversions

Fields which are sequences of bytes, including protocols, are implicitly converted to strings for comparisons against (double quoted) literal strings and raw strings.

So, for instance, the following filters are equivalent:

```
tcp.payload contains "GET"
tcp.payload contains 47.45.54
```

As noted above, a slice can also be compared in either way:

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

The inverse does not occur; stringlike fields are not implicitly converted to byte arrays. (Some operators allow stringlike fields to be compared with unquoted literals, which are then treated as strings; this is deprecated in general and specifically disallowed by the "matches" operator. Literal strings should be double quoted for clarity.)

A hex integer that is 0xff or less (which means it fits inside one byte) can be implicitly converted to a byte string. This is not allowed for hex integers 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 or octal numbers, since they would be confused with the hex numbers that make up byte string literals. Nevertheless, single-byte hex integers can be convenient:

```
frame[4] == 0xff
frame[1:4] contains 0x02
```

An integer or boolean field that has a value string can be compared to to one of the strings that corresponds with a value. As with stringlike fields and comparisons, it is possible to perform the comparison with an unquoted literal, though this is deprecated and will not work if the literal contains a space (as with "Modify Bearer Response" above). Double quotes are recommended.

If there is a unique reverse mapping from the string literal into a numeric value, the string is converted into that number and the comparison function is applied using arithmetic rules. If the mapping is not unique, then equality and inequality can be tested, but not the ordered comparisons.

This is in contrast to the `string()` and `vals()` function, which convert the field value to a string and applies string (lexicographic) comparisons, as well as work with all operators that take strings. Therefore the following two filters give the same result:

```
gtpv2.message_type <= 35
gtpv2.message_type <= "Modify Bearer Response"
```

whereas

```
vals(gtpv2.message_type) <= "Modify Bearer Response"
```

matches all messages whose value string precedes "Modify Bearer Response" in lexicographical order, and

```
string(gtpv2.message_type) <= "35"
```

matches all messages such that the message type comes before "35" in lexicographical order, i.e. would also match "170" (the message type for "Release Access Bearers Request.")

Bitwise operators

It is also possible to define tests with bitwise operations. Currently the following bitwise operator is supported:

```
bitand, bitwise_and, &  Bitwise AND
```

The bitwise AND operation allows masking bits and 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.

To match locally administered unicast ethernet addresses you can use:

```
eth.addr[0] & 0x0f == 2
```

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
```

Arithmetic operators

Arithmetic expressions are supported with the usual operators:

```
+   Addition
-   Subtraction
*   Multiplication
/   Division
%   Modulo (integer remainder)
```

Arithmetic operations can be performed on numeric types. Numeric types are integers, floating point numbers and date and time values.

Date and time values can only be multiplied by integers or floating point numbers (i.e: scalars) and furthermore the scalar multiplier must appear on the right-hand side of the arithmetic operation.

For example it is possible to filter for UDP destination ports greater or equal by one to the source port with the expression:

```
udp.dstport >= udp.srcport + 1
```

It is possible to group arithmetic expressions using curly brackets (parenthesis will not work for this):

```
tcp.dstport >= 4 * {tcp.srcport + 3}
```

Do not confuse this usage of curly brackets with set membership.

An unfortunate quirk in the filter syntax is that the subtraction operator must be preceded by a space character, so "A-B" must be written as "A -B" or "A - B".

Protocol field references

A variable using a sigil with the form $some.proto.field or ${some.proto.field} is called a field reference. A field reference is a field value read from the currently selected frame in the GUI. This is useful to build dynamic filters such as, frames since the last five minutes to the selected frame:

```
frame.time_relative >= ${frame.time_relative} - 300
```

or more simply

```
frame.time_relative >= $frame.time_relative - 300
```

Field references share a similar notation to macros but are distinct syntactical elements in the filter language.

Logical expressions

Tests can be combined using logical expressions. These too are expressible in C-like syntax or with English-like abbreviations. The following table lists the logical operators from highest to lowest precedence:

```
not, !    Logical NOT   (right-associative)
and, &&   Logical AND   (left-associative)
xor, ^^   Logical XOR   (left-associative)
or,  ||   Logical OR    (left-associative)
```

The evaluation is always performed left to right. Expressions can be grouped by parentheses as well. The expression "A and B or not C or D and not E or F" is read:

```
(A and B) or (not C) or (D and (not E)) or F
```

It's usually better to be explicit about grouping using parenthesis. 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.

Because each comparison has an implicit exists test for field values 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 existence 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://wiki.wireshark.org/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 the PCRE2 library. See 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 (https://www.tcpdump.org/manpages/pcap-filter.7.html)(7) or, if that doesn't exist, tcpdump (https://www.tcpdump.org/manpages/tcpdump.1.html)(8), or, if that doesn't exist, https://wiki.wireshark.org/CaptureFilters for a description of capture filters.

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

SEE ALSO

wireshark(1), tshark(1), editcap(1), pcap (https://www.tcpdump.org/manpages/pcap.3pcap.html)(3), pcap-filter (https://www.tcpdump.org/manpages/pcap-filter.7.html)(7) or tcpdump (https://www.tcpdump.org/manpages/tcpdump.1.html)(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.