Filtros Wireshark[1]

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 matches 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 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$"

[1] Disponível em: **www.wireshark.org.**

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
```

[Feel free to contribute more]

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.

View All HTTP trafic

```
http
```

View all flash video stuff

```
http.request.uri contains "flv" or http.request.uri contains "swf" or http.content_type contains "flash" or http.content_type contains "video"
```

Show non-google cache-control

```
http.cache_control != "private, x-gzip-ok="""
```

or

```
(((((http.cache_control != "private, x-gzip-ok=""") && !(http.cache_control == "no-cache, no-store, must-revalidate, max-age=0, proxy-revalidate, no-transform, private")) && !(http.cache_control == "max-age=0, no-store")) && !(http.cache_control == "private")) && !(http.cache_control == "no-cache")) && !(http.cache_control == "no-transform")
```

Show only certain responses

```
#404: page not found

http.response.code == 404



#200: OK

http.response.code == 200
```

Show only certain HTTP methods ^

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

Show only filetypes that begin with "text"

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

Show only javascript

```
http.content_type contains "javascript"
```

Show all http with content-type="image/(gif|jpeg|png|etc)"

```
http.content_type[0:5] == "image"
```

Show all http with content-type="image/gif"

```
http.content_type == "image/gif"
```

Do not show content http, only headers

```
http.response !=0 || http.request.method != "TRACE"
```

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

An integer may be expressed in decimal, octal, or hexadecimal notation. The following three display filters are equivalent:

```
frame.pkt_len > 10
frame.pkt_len > 012
frame.pkt_len > 0xa
```

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

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 InterDomain 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 Class-B 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 Class C network as 'sneezy':

```
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).

IPX networks are represented by unsigned 32-bit integers. Most likely you will be using hexadecimal when testing IPX network values:

```
ipx.src.net == 0xc0a82c00
```

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".

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
```

You can concatenate slices 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.

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 (llc exists)", 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.

A special caveat must be given regarding fields that occur more than once per packet. "ip.addr" occurs twice per IP packet, once for the source address, and once for the destination address. Likewise, "tr.rif.ring" fields can occur more than once per packet. The following two expressions are not equivalent:

```
    ip.addr ne 192.168.4.1
not ip.addr eq 192.168.4.1
```

The first filter says "show me packets where an ip.addr exists that does not equal 192.168.4.1". That is, as long as one ip.addr in the packet does not equal 192.168.4.1, the packet passes the display filter. The other ip.addr could equal 192.168.4.1 and the packet would still be displayed. The second filter says "don't show me any packets that have an ip.addr field equal to 192.168.4.1". If one ip.addr is 192.168.4.1, the packet does not pass. If neither ip.addr field is 192.168.4.1, then the packet is displayed.

It is easy to think of the 'ne' and 'eq' operators as having an implicit "exists" modifier when dealing with multiply-recurring fields. "ip.addr ne 192.168.4.1" can be thought of as "there exists an ip.addr that does not equal 192.168.4.1". "not ip.addr eq 192.168.4.1" can be thought of as "there does not exist an ip.addr equal to 192.168.4.1".

Be careful with multiply-recurring fields; they can be confusing.

Care must also 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. Filtering with "ip.dst" selects only those IP packets that satisfy the rule. Any other packets, 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 or ip.dst ne 224.1.2.3
not ip.addr eq 224.1.2.3
```

The first filter uses "not ip" 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 has already been explained above where filtering with multiply occurring fields was discussed.