Dispatch No. 008 · Field Notes

The Packet Doesn't Lie

Reading tcpdump filters byte by byte, and the 20-byte assumption that breaks the classic cheat-sheet recipes.

M.J. Martin
Outlan.net
October 2026 · Version 1.2.2

ABSTRACT

It's not the capture tool that finds the problem, rather it's knowing where the bytes are.

Because everything depends on the network, it is always the first suspect. "The network is so slow today" is the complaint. The reality is that the network is not "slow today," it's running as fast as it was the day before and the day before that. The fastest way I know to end that argument is a packet capture. A capture is evidence. It shows what was actually on the wire, at what time, and in what order.

That however only holds if the filter caught what you think it caught. A filter that silently misses half the traffic gives you a clean, convincing and wrong answer. In this dispatch we look at how tcpdump filters address individual bytes, walk a real packet in hex, and then take apart the assumption that breaks most of the byte-offset recipes still circulating on the internet.

1. A FOLDER FROM 2010

Recently I was cleaning out some old project files and came across a folder from around 2010, when I was building a network performance test harness. It's called "TCP Dump Filters." It holds the usual collection: a well-traveled tcpdump cheat sheet from 2008, the Wireshark wiki page on capture filters, a few papers on Ethernet overhead and switch latency, and my own hex breakdown of a captured packet.

Rereading that breakdown sixteen years later, I found that I had labeled the last 17 bytes of the packet as data. They are not all data. Twelve of them are TCP options (so much for reading the packet instead of the diagram). As it turns out, I was in good company. The cheat sheet in the same folder makes the same mistake, and so do a lot of the recipes copied from it since. That mistake is the whole lesson of this piece, so we will come back to it.

2. WHERE THE BYTES ARE

Most people use tcpdump with the named primitives: `host`, `net`, `port`, `tcp`, `udp`, `icmp`. They cover the common cases and they are the right place to start.

```
tcpdump -i eth0 host 192.0.2.9
tcpdump -i eth0 'tcp and dst port 80 and not src net 198.51.100.0/24'
```

Below the primitives is a lower-level syntax that reads raw header bytes. The form is `proto[offset:size]`, where `proto` is the header to read (ip, tcp, udp, icmp, ip6, ether), `offset` is the byte position counted from zero, and `size` is 1, 2 or 4 bytes. If size is left off, it is 1. So `ip[8]` is the IP Time to Live (TTL) byte, and `ip[2:2]` is the two-byte Total Length field.

Many fields are smaller than a byte, so the second tool is the bit mask. The expression `byte & mask` keeps the bits where the mask is 1 and zeroes the rest. Three tests cover nearly everything:

Table 1. Mask tests

Expression	True when
`x & m != 0`	at least one of the masked bits is set
`x & m = m`	all of the masked bits are set
`x & m = v`	the masked bits equal exactly the pattern v (the others are ignored)

Offsets are relative to the start of the named header, not the start of the frame. That one fact matters more than anything else in this dispatch.

3. A PACKET, BYTE BY BYTE

Here is the packet from my 2010 notes, with the addresses replaced by documentation addresses (192.0.2.0/24) and the checksums recalculated. The rest is unchanged. tcpdump prints the IP datagram in two-byte groups when run with `-x`:

```
0x0000:  4500 0039 f0b6 4000 3f06 c6cf c000 0209
0x0010:  c000 022f ebba 2742 c022 10d6 9b08 a6c1
0x0020:  8018 ffff 461a 0000 0101 080a 174f 794a
0x0030:  0c97 0a8e 6f70 656e 0a
```

The IP header comes first. Byte 0 is `45`. The high nibble (half byte) is the version, 4. The low nibble is the Internet Header Length (IHL), counted in 32-bit words, so 5 × 4 = 20 bytes. Bytes 2 and 3 are the Total Length, `0039`, which is 57 bytes for the whole datagram. Byte 6 is `40`, the Don't Fragment (DF) bit. Byte 8 is `3f`, a TTL of 63. Byte 9 is `06`, the protocol number for TCP. Bytes 12 through 19 are the source and destination addresses.

The TCP header starts at byte 20. Its first four bytes are the ports, `ebba` (60346) and `2742` (10050). Then come the sequence and acknowledgment numbers. Byte 12 of the TCP header, highlighted above, is `80`. Its high nibble is the Data Offset, the TCP header length in 32-bit words: 8 × 4 = 32 bytes. Byte 13 is `18`, the flags PSH and ACK.

A plain TCP header is 20 bytes. This one is 32, so 12 bytes of options follow the fixed header: `01 01 08 0a` and eight more. `01` is a No-Operation (NOP) pad byte. `08 0a` is option kind 8, length 10, the TCP Timestamps option (RFC 7323), carrying two 4-byte timestamp values. Only the last 5 bytes, `6f 70 65 6e 0a`, are payload. In ASCII they read "open" followed by a newline.

The arithmetic closes: 20 bytes of IP header, plus 32 bytes of TCP header, plus 5 bytes of payload, equals the 57 bytes in the Total Length field. My 2010 note counted 20 + 20 + 17. It read the fixed header size from a diagram instead of reading the Data Offset from the packet.

> **NOTE** The two length fields work the same way. IP stores its header length in the low nibble of byte 0. TCP stores its header length in the high nibble of byte 12. Both count 32-bit words, so multiply by 4 to get bytes. Neither header is a fixed size, and every byte-offset filter that reaches past the fixed part has to account for that.

4. THE 20-BYTE ASSUMPTION

Now the recipe. The 2008 cheat sheet in my folder, along with a good number of blog posts since, matches HTTP GET requests by reading the first four bytes after the TCP header:

```
tcpdump -i eth0 'tcp[20:4] = 0x47455420'     # "GET " in hex
```

The value is right. `47 45 54 20` is "GET " in ASCII. The offset is the problem. `tcp[20]` is the first payload byte only when the TCP header is exactly 20 bytes long. The cheat sheet warns that the filter "would not match packets with IP options set." That warning is backwards. Because `tcp[]` offsets start at the TCP header, IP options do not move them. TCP options do.

You can see this in the compiled filter. `tcpdump -d` prints the Berkeley Packet Filter (BPF) program that libpcap builds, and every `tcp[]` expression includes this instruction:

```
(006) ldxb     4*([14]&0xf)
```

Byte 14 of the frame is the first byte of the IP header (after the 14-byte Ethernet header). The instruction reads the IHL nibble from it, multiplies by 4 and stores the result as the base for every TCP offset. libpcap does the IP header arithmetic for you. Nothing does the TCP header arithmetic unless you write it.

The fix is a one-liner. Read the Data Offset nibble and compute where the payload starts:

```
tcpdump -i eth0 'tcp[((tcp[12:1] & 0xf0) >> 2):4] = 0x47455420'
```

The mask `0xf0` keeps the high nibble of byte 12. Shifting right by 2 is the same as shifting right by 4 (to get the word count) and multiplying by 4 (to get bytes). For the packet above, `0x80 >> 2` is 32, which is exactly where the payload starts.

Does it matter in practice? Linux enables TCP timestamps by default (`net.ipv4.tcp_timestamps = 1`), and a Linux sender that negotiates them puts the 12-byte option in every data segment on that connection. So the fixed-offset recipe misses requests from many Linux clients and from anything talking to a Linux server that agreed to timestamps. I built three GET requests and ran both filters against them:

Table 2. Matching "GET " three ways

Test packet	`tcp[20:4]`	Computed offset
TCP timestamps (32-byte TCP header)	missed	matched
No TCP options (20-byte TCP header)	matched	matched
IP options, no TCP options	matched	matched

The same cheat sheet gets it right a few pages later, in its SSH example: `tcp[(tcp[12]>>2):4] = 0x5353482D` matches "SSH-". That shorter form skips the mask and still works on real traffic, because the two lowest bits of byte 12 shift out and the bits above them are reserved and normally zero. The masked form costs nothing extra and does not depend on those reserved bits, so it is the one I keep in my recipe file.

One more caution before leaving payload matching. These filters read the first bytes of each packet, not a reassembled stream. HTTP/1.1 in clear text is a shrinking share of traffic (and inside TLS there is no "GET " to find), so the technique is the point here, not the HTTP example.

That however does not make encrypted traffic invisible to a byte filter. Every TLS connection starts with a handshake, and the client's first data segment begins with a TLS record header: one content type byte (`0x16`

for handshake), two version bytes and two length bytes. The byte after that is the handshake type, and `0x01` is the ClientHello (RFC 8446). Using the same computed offset:

```
tcpdump -i eth0 'tcp[((tcp[12:1] & 0xf0) >> 2)] = 0x16 and tcp[((tcp[12:1] & 0xf0) >>
2) + 5] = 0x01'
```

The ClientHello is worth catching because it carries the server name (SNI) in clear text, unless the client uses Encrypted Client Hello. Against my test packets it skipped the ServerHello and application data records. And yes, the fixed-offset shortcut fails here too:

Table 3. Finding the TLS ClientHello

Test packet	`tcp[20] = 0x16 and tcp[25] = 0x01`	Computed offset
ClientHello, TCP timestamps	missed	matched
ClientHello, no TCP options	matched	matched

5. BOTH LENGTH FIELDS AT ONCE

Here is the filter that puts the whole lesson in one line. It matches only the TCP segments that carry data, by working out the payload length from both header length fields:

```
tcpdump -i eth0 'tcp and (((ip[2:2] - ((ip[0]&0xf)<<2)) - ((tcp[12]&0xf0)>>2)) != 0)'
```

Read it from the inside out. `ip[2:2]` is the Total Length. `(ip[0]&0xf)<<2` is the IP header length in bytes (the IHL nibble times 4). `(tcp[12]&0xf0)>>2` is the TCP header length in bytes. What is left is payload. For the packet from my 2010 notes that is 57 − 20 − 32 = 5, the five bytes of "open." The tcpdump manual page carries a version of this filter for port 80, and it is the right answer to "show me the traffic, not the ACKs." Against my test packets it skipped pure ACKs, with and without IP options, and caught data segments with TCP timestamps, with IP options, and the TLS ClientHello. Like everything with `ip[]` in it, it is IPv4 only.

6. THE OPTION ORDER ASSUMPTION

TCP options have a second trap. The Maximum Segment Size (MSS) option rides in the SYN, and a small MSS is a good hint that a tunnel or VPN sits somewhere in the path. The usual shortcut reads it from the first option slot:

```
tcpdump -i eth0 'tcp[13] & 0x17 = 0x02 and tcp[20] = 2 and tcp[22:2] < 1460'
```

`tcp[20] = 2` checks that the first option is kind 2 (MSS), and `tcp[22:2]` reads its value. Most stacks put MSS first in the SYN, so the filter works most of the time. It caught a SYN advertising 1360. It missed the same SYN with two NOP pad bytes in front of the MSS option, because nothing in the TCP standard requires that order. BPF cannot loop through an option list, so this is as good as a byte filter gets. Treat a miss here as "not proven," not as "no small MSS."

7. TCP FLAGS

The flags sit in TCP byte 13, one bit each, from high to low: CWR, ECE, URG, ACK, PSH, RST, SYN, FIN.

Table 4. TCP byte 13

Bit value	Hex	Flag
128	0x80	CWR (Congestion Window Reduced)
64	0x40	ECE (ECN Echo)
32	0x20	URG
16	0x10	ACK
8	0x08	PSH
4	0x04	RST
2	0x02	SYN
1	0x01	FIN

The usual recipe for connection attempts is `tcp[13] = 2`, an exact match on SYN alone. It has the same flaw as the GET recipe: it assumes the rest of the byte is zero. A host that requests Explicit Congestion Notification (ECN, RFC 3168) sends its SYN with ECE and CWR also set, which makes byte 13 equal to `0xc2`, and the exact match misses it.

The better question is "SYN set, and ACK, RST and FIN clear." That is a mask of `0x17` (ACK + RST + SYN + FIN) with an expected value of `0x02`. libpcap also has names for the byte and the bits, which read better in a recipe file:

```
tcpdump -i eth0 'tcp[13] & 0x17 = 0x02'
tcpdump -i eth0 'tcp[tcpflags] & (tcp-syn|tcp-ack|tcp-rst|tcp-fin) == tcp-syn'
```

Table 5. Flag filters, tested

Filter	Matches
`tcp[13] = 2`	plain SYN only. Misses the ECN-setup SYN.
`tcp[13] & 0x17 = 0x02`	plain SYN and ECN-setup SYN. Not SYN-ACK.
`tcp[13] & 2 = 2`	anything with SYN set, including SYN-ACK
`tcp[13] = 18`	SYN-ACK (0x12)
`tcp[13] & 4 = 4`	any segment with RST set
`tcp[13] & 3 = 3`	SYN and FIN together

The last row is worth keeping. SYN and FIN together is not a valid combination in a normal connection. When you see it, someone is probing your stack or fingerprinting your firewall.

While you are in the TCP header, bytes 14 and 15 are the receive window. A window of zero means the receiver's buffer is full and the sender has to stop and wait. That is a host or application problem showing up on the wire, which is exactly the evidence you want when the network is being blamed:

```
tcpdump -i eth0 'tcp[14:2] = 0 and tcp[13] & 0x04 = 0'
```

The second test drops RST segments, which often carry a window of zero and mean something else entirely.

8. IP HEADER FIELDS

The IP header has fewer surprises, but the same rule applies: mask the bits you care about instead of matching the whole byte.

Table 6. IP header filters, tested

Question	Filter
IP options present?	`ip[0] & 0x0f > 5`
DF bit set?	`ip[6] & 0x40 != 0`
Any fragment (first, middle or last)?	`ip[6:2] & 0x3fff != 0`
More Fragments set (all but the last)?	`ip[6] & 0x20 != 0`
Low TTL (traceroute, loops)?	`ip[8] < 5`
Datagram longer than 600 bytes?	`ip[2:2] > 600`
Marked DSCP EF (voice)?	`ip[1] & 0xfc = 0xb8`
ECN Congestion Experienced?	`ip[1] & 0x03 = 0x03`

The fragment filter is the one to understand. Bytes 6 and 7 hold three flag bits and a 13-bit Fragment Offset. The mask `0x3fff` drops the reserved bit and the DF bit and keeps More Fragments plus the offset. Any non-zero result is a fragment: the first fragment has More Fragments set, the middle ones have both, and the last one has only an offset.

Byte 1 is the same trap as the SYN. It holds the 6-bit Differentiated Services Code Point (DSCP, RFC 2474) in the high bits and the 2-bit ECN field in the low bits. Expedited Forwarding (EF, DSCP 46, the usual voice marking) is `0xb8` when the ECN bits are zero. The exact match `ip[1] = 0xb8` misses EF traffic from a host that also sets an ECN codepoint (`0xb9`), and the masked form catches both. When the QoS policy is the suspect, that is the difference between "the marking is missing" and "the filter is wrong." The low two bits on their own tell you whether a router marked the packet Congestion Experienced (`11`).

The 2008 cheat sheet also warns that `ip[0] > 69` (a byte value above 0x45) would match IPv6 traffic. Under libpcap it does not, because `ip[]` only applies to frames with the IPv4 EtherType. The mask is still the right way to write it, because it says what you mean.

9. ICMP, INCLUDING WHAT IS INSIDE IT

ICMP type and code are bytes 0 and 1, and libpcap names them: `icmp[icmptype] = icmp-echo` matches echo requests. The more useful trick is reading the packet that an ICMP error carries. A Destination Unreachable message includes at least the IP header and the first 8 bytes of the datagram that caused it, starting at ICMP byte 8.

So to find port-unreachable messages caused by connections to TCP port 80:

```
tcpdump -i eth0 'icmp[0] = 3 and icmp[1] = 3 and icmp[17] = 6 and icmp[30:2] = 80'
```

`icmp[17]` is the protocol byte of the inner IP header (8 + 9). `icmp[30:2]` is the inner TCP destination port (8 + 20 + 2). The second offset assumes a 20-byte inner IP header, which is true for almost all traffic (there is that 20 again), and now you know exactly when it is not.

One ICMP message deserves a recipe of its own: Fragmentation Needed (type 3, code 4). A router sends it when a packet with DF set is too big for the next link, and it puts the next-hop MTU in ICMP bytes 6 and 7

(RFC 1191). When something in the path drops these messages, Path MTU Discovery fails quietly. The handshake works, small requests work, and large transfers hang. This shows who is sending them and what MTU they report:

```
tcpdump -i eth0 'icmp[0] = 3 and icmp[1] = 4 and icmp[6:2] < 1500'
```

One recipe from the old folder to retire: `icmp[0] = 4`, ICMP Source Quench, sent by routers to signal congestion. Hosts and routers stopped using it long ago, and RFC 6633 formally deprecated it in 2012. If one shows up in a capture today, it was not sent for congestion control.

10. UDP IS THE EASY ONE

After all of that, UDP is a relief. Its header is a fixed 8 bytes, so a `udp[]` offset past byte 8 lands where you expect (over IPv4). DNS is the obvious use. The DNS header starts at `udp[8]`, its flags sit in DNS bytes 2 and 3, and RFC 1035 puts the response code in the low nibble of byte 3 and the Truncation (TC) bit in byte 2:

```
tcpdump -i eth0 'udp src port 53 and udp[11] & 0x0f = 3'     # NXDOMAIN
tcpdump -i eth0 'udp src port 53 and udp[10] & 0x02 != 0'    # truncated (TC)
```

A run of NXDOMAIN responses can point at a bad search domain, a typo in a config file, or malware walking a list of generated names. A truncated response means the answer did not fit in UDP and the client is about to retry over TCP port 53, which a firewall that only allows UDP 53 will break. Each filter matched only its own case in testing. DNS over TLS or HTTPS hides all of this, the same way TLS hides the GET.

11. IPV6 CHANGES THE RULES

Every `tcp[]`, `udp[]` and `icmp[]` offset in this dispatch is an IPv4 filter. In libpcap 1.10.4, `ip6 and tcp[13] & 2 = 2` fails with "expression rejects all packets." For IPv6, you read from the start of the fixed 40-byte IPv6 header instead:

```
tcpdump -i eth0 'ip6 and ip6[6] = 6 and ip6[53] & 0x17 = 0x02'
```

`ip6[6]` is the Next Header byte, and `ip6[53]` is TCP byte 13 (40 + 13). It works only when TCP follows the base header directly. An extension header (hop-by-hop, routing, fragment) moves every offset after it, and BPF has no simple way to walk the chain. That is the IPv6 version of the 20-byte assumption (yes, it keeps showing up), and it is a good reason to prefer the named primitives (`tcp port 443`, `ip6 proto 6`) whenever they can answer the question.

ICMPv6 has the same problem, and here it matters most, because IPv6 routers never fragment. Packet Too Big (ICMPv6 type 2, RFC 4443) is how a sender normally learns the path MTU. With nothing between the base header and ICMPv6, `ip6[6] = 58 and ip6[40] = 2` finds it. Put a hop-by-hop header in front and it misses. So does libpcap's own named form, `icmp6[icmp6type] = icmp6-packettoobig`, which I expected to do better. It did not.

12. VLAN TAGS MOVE EVERYTHING

One more offset mover. An 802.1Q tag adds 4 bytes between the Ethernet header and the IP header, so on a trunk every `ip[]` and `tcp[]` offset is off by four. libpcap handles this with the `vlan` keyword, which shifts every offset that comes after it in the expression:

```
tcpdump -i eth0 'tcp[13] & 0x17 = 0x02'            # misses tagged SYNs
tcpdump -i eth0 'vlan and tcp[13] & 0x17 = 0x02'   # matches them
```

Order matters, because the keyword only moves what follows it, and the second filter stops matching untagged frames, so it is one or the other. Test it on the capture host itself. I tested from saved files. On Linux, the kernel can strip the tag before libpcap sees the frame (VLAN offload) and hand it over as metadata, so a live capture can behave differently than a file.

13. WHY FILTER AT THE BYTE LEVEL AT ALL

So why bother with any of this? Why not just capture everything and sort it out later in Wireshark? Well, on a quiet link you can. On a busy one you can't, and the reason is packets per second, not bits per second. It's the transaction rate, not the bandwidth, that buries a capture host.

A minimum Ethernet frame is 64 bytes, but it occupies 84 bytes of wire time once you add the 8-byte preamble and the 12-byte interframe gap. At 100Mbps that is 148,809 frames per second. At 1Gbps it is 1,488,095. At 10Gbps it is 14,880,952. Each frame is a transaction the capture host has to copy, timestamp and write. A capture filter runs in the kernel before any of that work happens, so the packets you do not want never cost you a copy. The filter is what keeps the capture host from becoming the next performance problem you are asked to investigate.

A filter does have limits. BPF looks at one packet at a time and keeps no state. It cannot tell you that a segment is a retransmission, that one arrived out of order, or that a connection never finished its handshake. That takes the whole conversation, and that is where Wireshark earns its keep. The filter decides what you keep. The analyzer tells you what it means.

14. KEEP THE RECIPE AND THE TEST

Every flawed filter in this dispatch has the same root cause. It treats a variable-length header as fixed, or a multi-bit field as a single value. Each one worked on the packets its author tested, and each one failed on packets its author never built.

There is an old line, "Trust, but verify." With byte-offset filters, I would skip the trust part. So my rule of thumb is simple. Do not trust a byte-offset filter until you have run it against a packet you made yourself. Scapy (a Python packet-crafting library) can build a test case for each variation in a few lines, and `tcpdump -r test.pcap 'filter'` tells you in a second whether it matches. Every filter in this dispatch was checked that way against 50 crafted packets, including the ones that did not match.

Then write it down. Keep the filter, what it matches, what it misses and the packet that proves it in the same file. I kept that folder for sixteen years, and the one entry I never tested was the one that was wrong. Document, document, document.

The packet does not lie. The filter can.

REFERENCES

[1] J. Postel, "Internet Protocol," RFC 791, September 1981. rfc-editor.org

[2] W. Eddy, Ed., "Transmission Control Protocol (TCP)," RFC 9293, August 2022. rfc-editor.org

[3] D. Borman et al., "TCP Extensions for High Performance," RFC 7323, September 2014. rfc-editor.org

[4] K. Ramakrishnan, S. Floyd and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP," RFC 3168, September 2001. rfc-editor.org

[5] F. Gont, "Deprecation of ICMP Source Quench Messages," RFC 6633, May 2012. rfc-editor.org

[6] P. Mockapetris, "Domain Names: Implementation and Specification," RFC 1035, November 1987. rfc-editor.org

[7] J. Mogul and S. Deering, "Path MTU Discovery," RFC 1191, November 1990. rfc-editor.org

[8] K. Nichols, S. Blake, F. Baker and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers," RFC 2474, December 1998. rfc-editor.org

[9] A. Conta, S. Deering and M. Gupta, Ed., "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification," RFC 4443, March 2006. rfc-editor.org

[10] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, August 2018. rfc-editor.org

[11] IEEE, "IEEE Standard for Local and Metropolitan Area Networks: Bridges and Bridged Networks," IEEE Std 802.1Q.

[12] The Tcpdump Group, tcpdump(1) manual page. tcpdump.org

[13] The Tcpdump Group, pcap-filter(7), packet filter syntax. tcpdump.org

[14] Red Hat, "Benefits of TCP timestamps," RHEL 9 Monitoring and Managing System Status and Performance, section 33.13. docs.redhat.com