5 When Scapy is too high-level

by Eric Davisson

Neighbors, we are hackers. Our power comes from the ability to understand and manipulate things at the lowest level we can get our hands on. Verily, a stack-based buffer overflow makes sense to those who understand machine code and assembly, but it makes no sense to whose who only use high-level languages, for they know not what a program stack is, nor rejoice in the wonders of the ABI.

Likewise with TCP/IP. Those who only use others' applications to talk to a networked host never learn the miracles of the protocols below. Preach to them the good news of Netcat, and of Scapy in Python or Net::Raw in Perl, neighbors—but forget not that these excellent tools may still mask the true glory of the raw bytes below.

This article will take us a step farther down than these tools do. We will create a proper packet in a pcap file with `xxd`. Let us please the ASCII art gods of TCP in the truly proper way, neighbors!

There are books dedicated to TCP/IP, neighbors, such as St. Stevens' *TCP/IP Illustrated Vol. 1*, a very thick and thorough book indeed. But at times when you don't have the Bible a mere tract would suffice; and so here's ours briefest tract on TCP/IP.

Let's begin by compressing the full OSI model to just the four layers that are actually relevant to TCP/IP. From the lowest layer up, we have the Data Link, Network, Transport, and Application layers—but of course it's not what we call these layers that matters, but what bytes they contain.

Each layer has a byte or two that specify which kind of protocol the next layer will be. So the Data Link Layer will specify IPv4 as the Network Layer, which will specify TCP as the Transport Layer, which will specify HTTPS as the Application Layer, and so on. This is really what makes the "stack", and we will tour it from the bottom up.

5.1 The Layers

Data Link Layer This is the first and the simplest layer. For most traffic, it has the destination and source MAC addresses and 2 bytes referring to what the Network Layer should be. The most common next protocol would be IPv4 (`0x0800`). Other possible protocols include IGMP (`0x0641`), ARP (`0x0806`), IPv6 (`0x86DD`), and STP (`0x8181`).

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Destination MAC Address                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Destination MAC Continued |   Source Mac Address            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Source MAC Continued                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Network Layer Protocol     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Network Layer (RFC791) Let's assume we are dealing with IPv4. There are many fields in the IPv4 header; the most interesting ones[7] are: Version, Total length, TTL, Source and Destination IP addresses, Checksum, and—the most important to our next layer—the Protocol byte.

That next layer to the IPv4 network layer protocol can also be many things. The most common are TCP (`0x06`), UDP (`0x11`), and ICMP (`0x01`), but there are well over a hundred other choices such as IGMP (`0x02`), GRE (`0x2F`), L2TP (`0x73`), SKIP (`0x39`), and many others.

[7] *The Pastor notes that* `fragroute` *might beg to differ, and your neighborly IDS might agree. It suffices to say that the IDS evasion party that Rev. Ptacek and Rev. Newsham started in 1998 is still going strong.*

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Transport Layer (RFC793) The intent of this layer is to handle the transportation of data between two hosts. For UDP, this header is just the source and destination ports, length, and a checksum. For "reliable" connections there's TCP, of which we'll talk more later. TCP headers are more complex, since it takes more data to set up a connection with a 3-way handshake and agreed-upon SEQ/ACK numbers. So TCP includes the ports, some flags, a window size, checksum, and some other fields. The destination port is implicitly used to specify what the application layer will be: HTTP (80), HTTPS (443), SSH (22), SMTP (25), and so on.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

And now that the gods as ASCII art have been properly pleased, let's make some packets!

5.2 Crafting a Packet

Link Layer Let's choose a destination MAC address of `12:34:56:78:9A:BC` and a source MAC address of `31:33:37:31:33:37`. We also need to specify the network-layer protocol of IPv4, `0x0800`.

Network Layer (IPv4) The version is 0x4, and that's the first nybble of our header. The header length is going to be twenty bytes, as we will use no IP options.[8]. The second header nybble is the header length in 32-bit words, and so it will be 0x5 to represent our twenty bytes. So the first byte will be 0x45, combining the version and the header length. When you next see this byte at the start of an IP packet's hexdump, give it a smiling node like a good neighbor!

The type of service byte doesn't matter unless your site implements special QoS for things like voice and streaming video, so we'll arbitrarily set that to 0x00. The following field, the total length of this packet, will be 61 bytes (IP+TCP+Payload), 0x003D in hex. We'll just spoof the IP identification field to be 0x1337. Next, let's set the IP flags to not fragment (0b010) and a fragment offset of zero. As these fields share bytes, the hex result of these two bytes will be 0x4000. For the next field, the Time-To-Live, let's be generous and give our packet a TTL of 140 (0x8C), which is higher than Linux or Windows would set by default.[9]

Our higher-layer protocol will be TCP, 0x06. Let's skip over the IP checksum for the moment (although we will have to correct that later). The source IP will be 192.168.1.1 (0xC0A80101) and the destination IP will be 192.168.1.2 (0xC0A80102), an HTTPS server. There will be no options or padding.

To compute the checksum, let's take all our IP header data we filled in so far in two-byte chunks, add it together, then add the overflowing byte back into the result, and subtract from 0xFFFF. So 0x4500 + 0x003D + 0x1337 + 0x4000 + 0x8C06 + 0xC0A8 + 0x0101 + 0xC0A8 + 0x0102 is 0x2A7CD. 0x2 is the overflow, so we add it back in to get 0xA7CD + 0x2 = 0xA7CF. Subtracting this from 0xFFFF, we find 0xFFFF - 0xA7CF is 0x5830, our packet's IPv4 checksum.

It's now time to set up our transport layer, TCP.

Transport Layer (TCP) Let's say our source port will be 0x1337, and the destination port will be 0x01BB, which is decimal 443 for HTTPS. There's no point to any specific SEQ or ACK numbers for this implausible single packet, so we'll just use 0x00000000 and 0x00000000.

The data offset (TCP header length) and flags share some bytes. We will have 32 bytes in our TCP header, including the 12 bytes of TCP options. 32 bytes are eight 32-bit words, so our data offset field is 0x8.

We want this packet to have the flags of PUSH and ACK, so setting these bits gives us 0x18. Combining these two values gives us the 2-byte value of 0x8018, where the middle zero is a reserved nybble.

As we don't care to specify a window size at the moment, we'll default to 0x0000—but keep in mind that putting a zero length in a TCP response is a rather evil trick you should only use on spammers and SEOs (look up the SMTP/TCP "LaBrea Tarpit" technique for more details.) We will do the checksum later, as a TCP checksum applies both to the header and to the payload. Since we won't be using the URG flag to mark this packet as urgent, we'll leave the urgent pointer field as 0x0000.

For the options, we will use two NOPs for padding, to ensure an even number of 32-bit words, 0x0101. Our option will be a timestamp (0x08), with a length of 10 (0x0A). Its TSval will arbitrarily be 0xDEADBEEF, and its TSecr will be 0xFFFFFFFF.

It is now time for the TCP checksum. A TCP checksum is calculated similarly to the IP one, but it also covers some of the IP fields![10] The source IP, the destination IP, and the protocol number must all be included. Also included is the size of the TCP section, including the payload data.

(0xC0A8 + 0x0101 + 0xC0A8 + 0x0102 + 0x0006 + 0x0029) + 0x1337 + 0x01BB + 0x0000 + 0x0000 + 0x0000 + 0x0000 + 0x8018 + 0x0000 + 0x0000 + 0x0101 + 0x080A + 0xDEAD + 0xBEEF + 0xFFFF + 0xFFFF + 0xD796 + 0xC34F + 0x4FC7 + 0xE3C6 + 0xD600 is 0x963A3 with an overflow of 0x9. 0x63A3 + 0x9 is 0x63AC, and 0xFFFF - 0x63AC is 0x9C53, our TCP checksum.

PCAP Metadata So now we have the packet, but to look at it with the standard dissection tools (Tcpdump, Wireshark) or to use it with an injection tool (Tcpreplay), we need to create some metadata first.

[8] *But if you are looking to light up your local IDS like a Christmas tree, by all means add some later! –PML*

[9] *But check out* /proc/sys/net/ipv4/ip_default_ttl*; for Windows, you are on your own—and many happy reboots! –PML*

[10] *Yes, neighbors, it* is *an OSI layering violation—and it has been extracting its cost, in sweat, blood, and 0day. And if you think you are properly scared, you are not scared enough—just think of that SCADA protocol that has kept your neighborhood's lights on, so far. –PML*

We will use the PCAP format, the most common format of packet capture tools.

A PCAP starts with 24 bytes of global file-scope metadata and another 16 bytes of per-packet metadata. The first six of PCAP's 4-byte fields are the magic number (0xA1B2C3D4), the PCAP version (2.4, so 0x00020004), the timezone (GMT, so 0x00000000), the sigfigs field[11] (0x00000000), the snaplen[12] (0x0001000F) and the network's data link type[13] (Ethernet: 0x00000001).

So our global header will be A1B2C3D4000200040000000000000000000000001000F00000001. Fun fact: reversing the order of the magic number to 0xD4C3B2A1 will change the endianness of the PCAP metadata—alerting your packet analyzer that the order of bytes in the capture file from another system should be reversed.

The per-packet data consists of four 4-byte fields: time, microtime, packet length, and captured length. Let's set the time to default day (0x4EBD02CF) and zero out the microtime (0x00000000). Our packet length will be 0x00000004B, and we'll repeat the same value for the capture length.

Saving the pcap. Below you see a massively ugly command. We are echoing all of the above hex data in order, starting with the PCAP file's global metadata and following with the packet data. There isn't a single byte of this that we didn't discuss above; it's all there. We pipe it through xxd and use the -r and -p arguments to convert it from hex to actual binary data (-p tells xxd to expect a continuous hexdump without per-line addresses or offsets, rather than the standard xxd output; any whitespace including line breaks is ignored in this mode). Say hello to lol.pcap:

```
echo A1B2C3D4 00020004 00000000 00000000 0001000F 00000001 \
     4EBD02CF 00000000 0000004B 0000004B \
     \
     12345678 9ABC3133 37313337 0800 \
     \
     45 00 003D 1337 4 000 8C 06 5830 C0A80101 C0A80102 \
     \
     1337 01BB 00000000 00000000 8 0 18 0000 9C53 0000 \
     01 01  08 0A DEADBEEF FFFFFFFF \
     \
     D796C34F4FC7E3C6D6 | xxd -r -p > lol.pcap
```

Now that you have a PCAP (see also Fig. 1), you can open it up in Wireshark and select each field in the Packet Details section to see the corresponding hex data in the Packet Bytes section. If you want to send a hand-crafted packet over your network, just replay it with something like

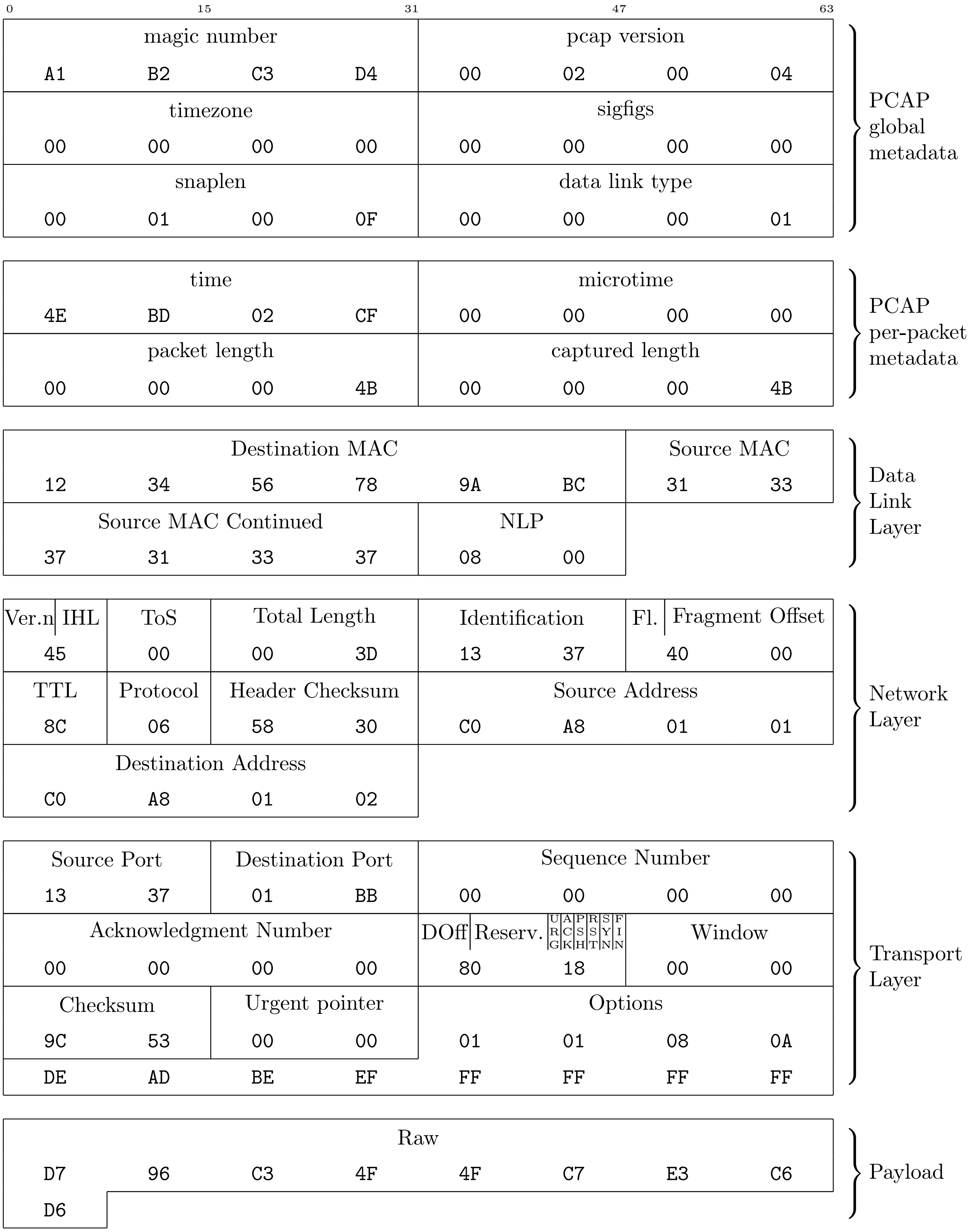
```
sudo tcpreplay -i eth0 lol.pcap
```

Hack around, change some bytes, and see what happens. Do impossible things, like setting the IPv4 layer's first byte to 0x43, which specifies an IPv4 packet with a 12-byte IP header. This means the IP header doesn't have room for its own IP addresses. What will your little Linksys box do when it gets such a packet? What will your newest shiny box with that fruit logo do? And how much do you dare trust that penguin, really? Well, there is—and there has ever been—only one way to find out :)

[11] In theory, this is the accuracy of time stamps in the capture; in practice, typically set to zero.

[12] This is the maximum length of captured packets, in octets, or zero for no limit.

[13] man 7 pcap-linktype (from libpcap0.8-dev or equivalent)



magic number				pcap version			
A1	B2	C3	D4	00	02	00	04
timezone				sigfigs			
00	00	00	00	00	00	00	00
snaplen				data link type			
00	01	00	0F	00	00	00	01

PCAP global metadata

time				microtime			
4E	BD	02	CF	00	00	00	00
packet length				captured length			
00	00	00	4B	00	00	00	4B

PCAP per-packet metadata

Destination MAC						Source MAC	
12	34	56	78	9A	BC	31	33
Source MAC Continued				NLP			
37	31	33	37	08	00		

Data Link Layer

Ver.n IHL	ToS	Total Length		Identification		Fl. Fragment Offset	
45	00	00	3D	13	37	40	00
TTL	Protocol	Header Checksum		Source Address			
8C	06	58	30	C0	A8	01	01
Destination Address							
C0	A8	01	02				

Network Layer

Source Port		Destination Port		Sequence Number			
13	37	01	BB	00	00	00	00
Acknowledgment Number				DOff Reserv.	URG ACK PSH RST SYN FIN	Window	
00	00	00	00	80	18	00	00
Checksum		Urgent pointer		Options			
9C	53	00	00	01	01	08	0A
DE	AD	BE	EF	FF	FF	FF	FF

Transport Layer

Raw							
D7	96	C3	4F	4F	C7	E3	C6
D6							

Payload

Figure 1: Crafted PCAP