Parsing HTTP/2 packets in Python with dpkt

gendignoux.com/blog/2017/05/30/dpkt-parsing-http2.html

Some time ago I started a project involving analysis of network traffic. It is easy to capture packets and store them in a PCAP file with tcpdump, and to visualize them with Wireshark. But Wireshark is a bit heavy and not really convenient when it comes to custom analysis and scripting. So I looked for Python libraries to parse PCAP files and found dpkt, a lightweight library that supports more than 70 TCP/IP protocols.

For my project, I needed to parse HTTP/2 packets (RFC 7540). This protocol was not supported yet, but it turned out that dpkt is quite easy to extend! So I wrote a new HTTP/2 parser, which is now integrated in the project on GitHub, and I also completed the list of SSL/TLS cipher suites.

In this post, I will show you that it is easy to write your own parsers with dpkt.

Overview of dpkt

To install dpkt, you can simply download the latest release with pip.

```
pip install dpkt
```

Alternatively, if you want the most up-to-date code, you can clone the GitHub repository.

```
git clone https://github.com/kbandla/dpkt
```

You can then read a PCAP file with a few lines of Python.

```
import dpkt

with open('file.pcap', 'rb') as f:
    pcap = dpkt.pcap.Reader(f)
    for timestamp, buf in pcap:
        # Unpack Ethernet frame
        eth = dpkt.ethernet.Ethernet(buf)

        if not isinstance(eth.data, dpkt.ip.IP):
            print("Not an IP packet...")
            continue

        # Extract IP packet
        ip = eth.data

        # TODO: process packet
```

I won’t explain in details how to use dpkt, but you can have a look at full examples on the GitHub repository.

Parsing a simple packet format

Format specification

In this section, we will write a dpkt parser for a toy packet format, that follows the *tag-length-value* (TLV) paradigm. This paradigm is commonly used in network protocols and binary file formats.

The specification of our TLV packets is quite simple. Each TLV packet contains three fields:

- a tag/type (4-byte integer),
- a length N (4-byte integer),
- N bytes of data.

Writing a parser

We now write a TLV parser in Python. We first import dpkt and create a new class derived from `Packet`.

```
import dpkt

class TLVPacket(dpkt.Packet):
```

In this class, we then add a `__hdr__` field containing the definitions of all header fields. For each field, we need to provide a tuple made of the field name, the field format, and a default value. The field format follows the syntax of `struct.unpack`. The default value is useful only if you want to create packets; in this example we only consider parsing so we will leave it to zero.

```
    __hdr__ = (
        ('tag', 'I', 0),
        ('length', 'I', 0),
    )
```

We have now defined the first two fields of our packet, but we still need to extract N bytes according to the length field. For this we re-implement the `unpack` method, which takes as argument a byte buffer. We first call the base implementation to extract the headers.

```
    def unpack(self, buf):
        dpkt.Packet.unpack(self, buf)
```

At this point, fields `self.tag` and `self.length` are populated, and `self.data` contains the remaining of the buffer. We can simply truncate it to the correct length, or raise an exception if not enough data is available.

```
        self.data = self.data[:self.length]
        # raise an exception if we need more data
        if len(self.data) < self.length:
            raise dpkt.NeedData
```

That's it! You can now parse a packet by simply constructing a `TLVPacket` from a byte buffer, e.g. `p = TLVPacket(buf)`. You can then access `p.tag`, `p.length` and `p.data`.

Parsing a stream of packets

A more interesting function is to parse a stream of concatenated packets. For example, you can imagine application packets collected from the transport layer, but not necessarily aligned to TCP/TLS packets.

Let's assume that you have collected a byte buffer from underlying TCP/TLS packets. You can extract application packets with the following loop, checking for the `NeedData` exception.

```
def parse_tlv_stream(buf):
    i = 0
    packets = []

    while True:
        try:
            p = TLVPacket(buf[i:])
            packets.append(p)
            # len(p) = length of (header + data)
            i += len(p)
        except dpkt.NeedData:
            break

    # number of bytes used, list of packets
    return i, packets
```

It turns out that this TLV stream parser is almost a parser for the PNG image format, with the exception that PNG also uses a checksum for each « packet » (called *chunk* for PNG).

Practical pitfalls and workarounds

As we have seen, writing a dpkt parser takes only a few lines of Python code, but here are some tips that you may need in real-world cases.

Endianness

By default, dpkt uses **big-endian** (network) byte order to extract numeric headers with `struct.unpack`. This means that dpkt adds the `>` modifier if you don't specify endianness, as we did in our toy example.

If your format contains little-endian fields, simply add the `<` modifier in corresponding `__hdr__` tuples:

```
    # A little-endian packet format
    __hdr__ = (
        ('tag', '<I', 0),
        ('length', '<I', 0),
    )
```

Non-standard field length

In the case of HTTP/2, the data length is stored in a 24-bit integer field. Unfortunately, `struct.unpack` does not support unpacking of 24-bit integers out of the box. A practical workaround is to define a 3-byte header field and implement the conversion logic in the `unpack` function.

Here is an excerpt from HTTP/2.

```
class Frame(dpkt.Packet):

    # struct.unpack can't handle the 3-byte int,
    # so we parse it as bytes (and store it as
    # bytes so dpkt doesn't get confused), and
    # turn it into an int in a user-facing property
    __hdr__ = (
        ('length_bytes', '3s', 0),
        ('type', 'B', 0),
        # ...
    )

    # property to parse the 24-bit integer length
    @property
    def length(self):
        # add a zero byte and unpack a 32-bit integer
        return struct.unpack('!I', b'\x00' + self.length_bytes)[0]

    def unpack(self, buf):
        dpkt.Packet.unpack(self, buf)
        # self.length is parsed in the @property
        self.data = self.data[:self.length]
        if len(self.data) != self.length:
            raise dpkt.NeedData
```

Conclusion

I hope that this introduction to dpkt was useful! You can now prototype your own parsers for simple binary formats in a few lines of Python. Don’t hesitate to contribute to the project on GitHub or to let me know if you found the HTTP/2 parser useful!

Comments

To react to this blog post please check the Twitter thread.

You may also like

Rust from a C++ and OCaml programmer's perspective (Part 1)
Rust 2020
Tutorial: Profiling Rust applications in Docker with perf
Asynchronous streams in Rust (part 1) - Futures, buffering and mysterious compilation error messages
And 22 more posts on this blog!