Beyond Custom TLVs

Joe Hildebrand

Brian Trammell

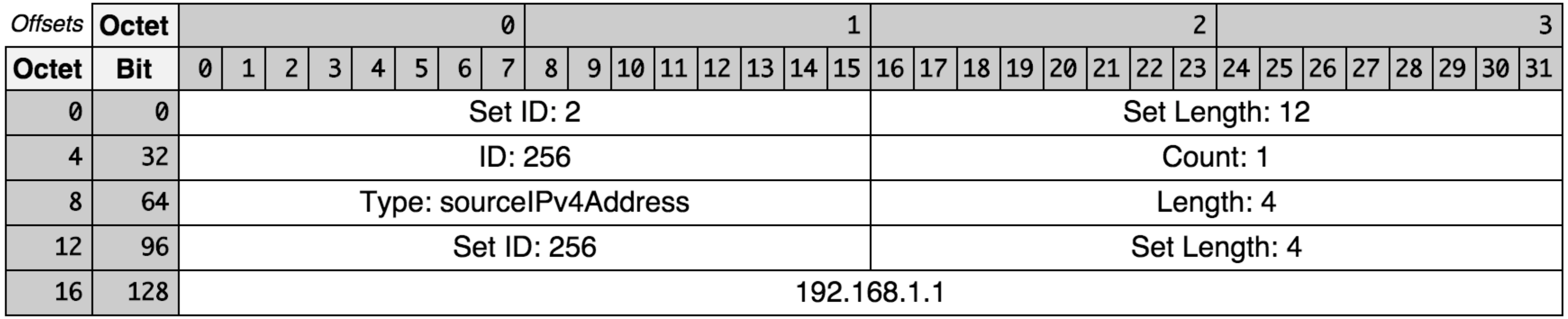
Start: two types of types

- *Storage* types can be parsed generically
  - uint64_t
  - UTF8-encoded string
  - Array of bytes
  - Name/value map

- *Semantic* types drive behavior, **always** protocol-specific
  - Source/destination address: 4/16 byte array? String?
  - Hop count: unsigned integer

Today: Below layer 7

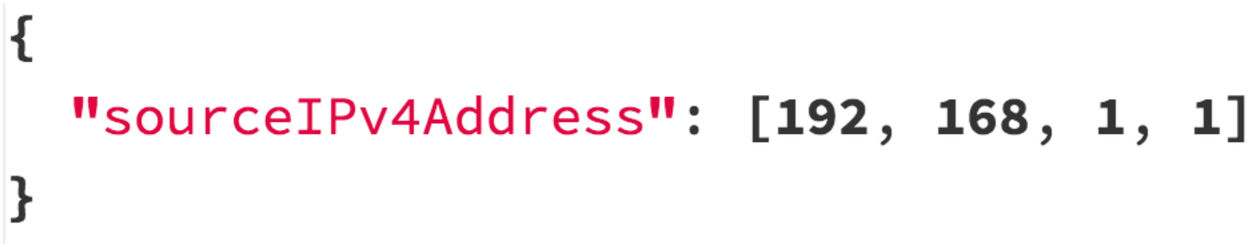
- Many protocols have custom Type, Length, Value formats
- “Type” often means both storage type *and* semantic type
- Custom parsing required
- Custom type system required
- Example: IPFIX sourceIPv4Address
  - Storage type is a 32-bit integer IP address in network byte order
  - Semantic type is “Source IPv4 Address”



<table>
<tr><td><i>Offsets</i></td><td><b>Octet</b></td><td colspan="8">0</td><td colspan="8">1</td><td colspan="8">2</td><td colspan="8">3</td></tr>
<tr><td><b>Octet</b></td><td><b>Bit</b></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
<tr><td>0</td><td>0</td><td colspan="16">Set ID: 2</td><td colspan="16">Set Length: 12</td></tr>
<tr><td>4</td><td>32</td><td colspan="16">ID: 256</td><td colspan="16">Count: 1</td></tr>
<tr><td>8</td><td>64</td><td colspan="16">Type: sourceIPv4Address</td><td colspan="16">Length: 4</td></tr>
<tr><td>12</td><td>96</td><td colspan="16">Set ID: 256</td><td colspan="16">Set Length: 4</td></tr>
<tr><td>16</td><td>128</td><td colspan="32">192.168.1.1</td></tr>
</table>

Today: Applications use JSON

- No schema for parsing
- Storage types: bool, number, string, object, array, etc.
- Semantic types: key names in object, position in array, etc.
- Parse internal field structure in same pass
- Extensions as new keys in a key/value struct
- Ignore what you don’t understand



```
{
  "sourceIPv4Address": [192, 168, 1, 1]
}
```

Why JSON might not be a good fit

- See RFC 7159, search for “interop”
  - More edge cases than you think
  - My favorite: 53-bit integers
- Parser more complicated than you expect
  - Larger code size, more CPU
  - Example: String un-escaping
- Binary data requires encoding (such as Base64)
- Larger wire size

Why CBOR might be a better fit

- RFC 7049: binary encoding of JSON++
- Small wire size: often smaller than TLV
- Small code size (e.g. 880 bytes of ARM code)
- Lower CPU, latency to parse
- Fixes the known issues of JSON (integers, floats, strings, etc.)
- Binary data first-class type
- Defined diagnostic rendering

```
0xa171736f7572636549507634416464726573734 4c0a8
0101

  a1              -- Map with 1 pairs
    71            -- Map[0].key: UTF-8 string length
17:
      736f…        "sourceIPv4Address”
    44            -- Map[0].value: Byte string
length 4
      c0a80101 -- Bytes content: 192, 168, 1, 1
```

How to make CBOR even more suitable

- Profile out the pieces that you don’t need
- Allow parse failures if those features arrive
- Potential removals:
  - Indefinite-length types
  - Tagging (bignums, etc)
  - Floats

Benefits to choosing a single approach

- One set of code
  - Smaller
  - Better code coverage
  - Optimization more likely: e.g. hardware
- Potential security benefits – new syntax sometimes a source of bugs
- Time to market
- Better diagnostic tooling

Suggested topics for discussion

- Could [routing, ops, etc.] protocols use a single approach like this?
- What are the potential downsides?
- Is CBOR a potential encoding?
- Is there a protocol that could be used as an experiment?