wssdl documentation

Release 0.2.0

Franklin "Snaipe" Mathieu

Aug 01, 2017

Contents

CHAPTER 1

Setup

Prerequisites

The library needs a recent version of Wireshark and lua 5.1+.

Although the library is expected to work on older versions of Wireshark, it has only been tested on 2.2.0 and above.

Installing the library

From a release

Grab `wssdl.lua` from the latest release, and put it in one of Wireshark's plugin paths.

Note: Usually, Wireshark loads plugins from `~/.config/wireshark/plugins` and `/usr/lib/wireshark/plugins/<version>`. You can check what directories Wireshark checks by going into Help -> About -> Folders.

From source

Building from source requires as an additional prerequisite luarocks and the luafilesystem module to be installed.

To boostrap the library in one coalesced file, and install it to `~/.config/wireshark/plugins`, run from the project directory:

```
$ make install
```

If you prefer to install it in another location, set the variable `WS_PLUGIN_DIR`. For instance, to install wssdl in the system plugin path for Wireshark 2.2.0:

```
$ sudo make WS_PLUGIN_DIR=/usr/lib/wireshark/plugins/2.2.0 install
```

CHAPTER 2

Getting started

Packet definition

The `packet` function is used to define the structure of your packet.

This function takes a sequence of comma/semicolon-separated fields, with each field using the `<field_id> : <specifier1>(params) ... : specifierN(params)` syntax, where `<field_id>` is an lua identifier for the field that is unique in the current definition scope; and where each `<specifier>` is a wssdl specifier, one of which must be a field type.

See *Specifier reference* for a complete list of specifiers.

```
local wssdl = require 'wssdl'

my_pkt = wssdl.packet {
    foo : u8();
    bar : i32();
    baz : utf8(256);
}
```

Creating a protocol

A `Proto` object can be created by calling the `proto(name, description)` method on the created packet type:

```
my_pkt = wssdl.packet { ... }

proto = my_pkt:proto('proto_id', 'Some protocol')
```

The protocol name and description are passed verbatim to wireshark and as such **must** both be unique.

Registering a dissector

The `dissect` function can be used to register one or more protocols in their relevant dissector tables.

This function takes a sequence of dissector table mappings. Each mapping follows the following syntax: `<key>:<method> { <keyvalues> }`, where `<key>` is the identifier of the desired dissector table, `<method>` is either `set` or `add` (which holds the semantics of `DissectorTable:set` and `DissectorTable:add` respectively), and `<keyvalues>` are key/value entries where the key is the first parameter of `set/add` and the value is the proto object passed as second parameter.

```
wssdl.dissect {
  tcp.proto:add {
    [1234] = my_pkt:proto('proto_id', 'Some protocol')
  }
}
```

CHAPTER 3

Specifier reference

Primitive Field Types

Type	Description
`u8()`	Unsigned 8-bit integer.
`u16()`	Unsigned 16-bit integer.
`u24()`	Unsigned 24-bit integer.
`u32()`	Unsigned 32-bit integer.
`u64()`	Unsigned 64-bit integer.
`i8()`	Signed 8-bit integer.
`i16()`	Signed 16-bit integer.
`i24()`	Signed 24-bit integer.
`i32()`	Signed 32-bit integer.
`i64()`	Signed 64-bit integer.
`int(N)`	Unsigned `N`-bit integer. If `N` isn't specified, the size of the field becomes the remaining payload size. `N` cannot be larger than 64-bits.
`uint(N)`	Unsigned `N`-bit integer. If `N` isn't specified, the size of the field becomes the remaining payload size. `N` cannot be larger than 64-bits.
`f32()`	32-bit floating-point value.
`f64()`	64-bit floating-point value.
`utf8(N)`	UTF8-encoded string w/ a length of `N` code units. If `N` isn't specified, the size of the field becomes the remaining payload size. If used, the field must be aligned on an octet boundary.
`utf8z()`	Null-terminated UTF8-encoded string. If used, the field must be aligned on an octet boundary.
`utf16(N)`	UTF16-encoded string w/ a length of `N` code units. If `N` isn't specified, the size of the field becomes the remaining payload size. If used, the field must be aligned on an octet boundary.
`utf16z()`	Null-terminated UTF16-encoded string. If used, the field must be aligned on an octet boundary.
`bytes(N)`	Byte buffer with a size of `N` octets. If `N` isn't specified, the size of the field becomes the remaining payload size. If used, the field must be aligned on an octet boundary.
`bits(N)`	Bits buffer with a size of `N` bits. `N` cannot be larger than 64-bits.
`bool(N)`	Boolean value with a size of `N` bits. If `N` isn't specified the size of this field is 1 bit. A field value of zero means False, while non-zero means True.
`bit()`	A single bit.
`ipv4()`	IPv4 address. If used, the field must be aligned on an octet boundary.
`ipv6()`	IPv6 address. If used, the field must be aligned on an octet boundary.

Special Field Types

User Types

Any variable declared with `wssdl.packet` can be used as a field type.

Payload Type

`payload(<criterion>, [size])`

The special payload type is used for packets that contains data that needs to be subdissected by another registered dissector.

The `<criterion>` parameter is either a field, or a 2-element table containing a field and a key:

- `payload(<field>, [size])`
- `payload({ <field>, <key> }, [size])`

`<field>` is the field that should be used as the value to lookup the dissector table entry, `<key>` is the dissector table identifier.

If `<key>` is nil or unspecified, then the dissector table identifier becomes `<prototype name>.<field>`.

`<size>` is an optional parameter representing the size of the field in octets.

If `<size>` is nil or unspecified, then the size of the field becomes the remaining packet size.

Other specifiers

Type	Description
`le()`	Parse the field as little-endian. The following types support little-endian: u8, u16, u24, u32, u64, i8, i16, i24, i32, i64, int, uint, f32, f64, utf16, utf16z, ipv4.
`dec()`	Use a decimal format for the integer field (default)
`hex()`	Use a hexadecimal format for the integer field
`oct()`	Use an octal format for the integer field
`name(str)`	Set the display name of the field to `str`.
`description(str)`	Set the description of the field to `str`.

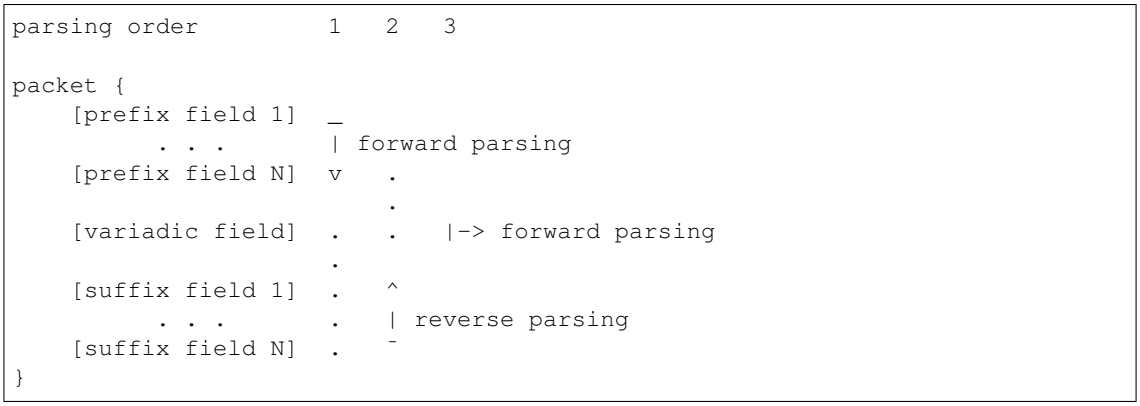
CHAPTER 4

How definitions are parsed

The parsing process

A packet definition always contain zero or more fields, called *prefix* fields, followed optionally by one field with unspecified size, called *variadic* field, followed by zero or more end fields, called *suffix* fields.

1. Prefix fields are parsed first, top-to-bottom, until the end of the packet is reached or a variadic field is reached.
2. If a variadic field is reached, it is skipped and the parser jumps to the last suffix field.
3. All suffix fields are parsed, bottom-to-top, until the variadic field is reached again.
4. The variadic field is parsed, with a size equal to the gap between the last prefix field and the first suffix field.



Example

Given this definition:

```
wssdl.packet {
    prefix  : u8();
    var     : bytes();
    suffix  : u8();
}
```

And this 3-byte raw packet: `ababab` (hexadecimal form)

1. `prefix` is parsed, a value of `0xab` is found.
2. `variadic` is reached, the parser jumps to the last suffix field
3. `suffix` is parsed, a value of `0xab` is found.

Reverse parsing pitfalls

Because suffix fields are parsed bottom-to-top, the resolution rules and the constraints change slightly to make the reverse parsing possible:

- Null-terminated string types (`utf8z`, `utf16z`) are prohibited. This is because the null character would appear first during the reverse parsing, and we would have no way of knowing the size of the field.
- Root packets (i.e. packets used as protocols) are implicitely aligned on an 8-bit boundary – mind the alignment constraint when you have unaligned suffix fields!
- Fields with a size that depends on the value of another field needs to be parsed after the field they depend on is parsed. This means that for suffix fields, dependencies needs to appear *after* the field definition.

  For instance, this is invalid:

```
wssdl.packet {
    prefix    : u8();
    var       : bytes();
    suffix_sz : u8();
    suffix    : bytes(suffix_sz);
}
```

  While this is valid:

```
wssdl.packet {
    prefix    : u8();
    var       : bytes();
    suffix    : bytes(suffix_sz);
    suffix_sz : u8();
}
```