eTPL: An Enhanced Version of the TLS Presentation Language Suitable for Automated Parser Generation

Andreas Walz, Axel Sikora
Institute of Reliable Embedded Systems and Communication Electronics
Offenburg University of Applied Sciences
Badstrasse 24, 77652 Offenburg, Germany
Email: {andreas.walz, axel.sikora}@hs-offenburg.de

***Abstract*—The specification of the *Transport Layer Security* (TLS) protocol defines its own presentation language used for the purpose of semi-formally describing the structure and *on-the-wire* format of TLS protocol messages. This *TLS Presentation Language* (TPL) is more expressive and concise than natural language or tabular descriptions, but as a result of its limited objective has a number of deficiencies. We present *eTPL*, an enhanced version of TPL that improves its expressiveness, flexibility, and applicability to non-TLS scenarios. We first define a generic model that describes the parsing of binary data. Based on this, we propose language constructs for TPL that capture important information which would otherwise have to be picked manually from informal protocol descriptions. Finally, we briefly introduce our software tool `etpl-tool` which reads eTPL definitions and automatically generates corresponding message parsers in `C++`. We see our work as a contribution supporting sniffing, debugging, and rapid-prototyping of wired and wireless communication systems.**

I. Introduction

Typically, the specification of an implementable network protocol requires a paper-friendly description of its messages. In many cases this includes a description of the messages' encoding, i.e. their binary *on-the-wire* format. To this end, the specifications of quite a number of both long-standing as well as recent protocols resort to informal, typically table-like illustrations of the binary layout of messages; e.g. IP [1], TCP [2], and 6LoWPAN [3]. This approach still appears to be the most popular one, despite the long time that has passed since network protocols are subject to written specification.

Having said that, there are other options which can serve a similar purpose, though at a higher layer of abstraction. So-called *Data Description Languages* (DDL) can be used to describe messages in an abstract way. The *Abstract Syntax Notation One* (ASN.1) [4] probably is the most popular example, but many more DDLs exist; see e.g. [5], [6], [7]. Most DDLs effectively disentangle the description of the message content from the matter of message encoding. Typically, the latter is handled in a platform-independent way by well-established encoding schemes. For ASN.1, a popular example is given by the *Distinguished Encoding Rules* (DER). Unfortunately, using DDL-based message exchange comes with a certain overhead, both in terms of implementation and communication.

The authors of the *Transport Layer Security* (TLS) protocol specification have taken yet another approach. Defined solely for the purpose of semi-formally describing the structure and binary format of TLS and DTLS messages as sent over the network, a new description ("presentation") language has been introduced [8], [9], [10], [11], [12]. In the following, we refer to this presentation language as *TLS Presentation Language* (TPL).

Despite its originally quite limited scope, TPL has also been used and extended in the context of other protocol specifications. For example, the *European Telecommunications Standards Institute* is using its own version of TPL for describing the message and certificate formats for *Intelligent Transport Systems* [13]. The family of IEEE Standards for *Wireless Access in Vehicular Environments* (IEEE 1609) used a similar, TPL-based presentation language in an earlier version of the standard [14] but has switched to ASN.1 in a later version [15].

It can be seen as an indication of the general appeal of TPL that it has been made use of well beyond the scope of TLS. However, there are significant deficiencies of TPL that prevent its wider and more formal use. First, despite the fact that TPL is used more or less consistenly throughout the TLS specification, the definition of TPL itself is somewhat casual and lacks a formal basis. Furthermore, there remain some important aspects of the format of messages that – though affecting the message parsing – cannot be expressed in TPL [16]. For example, TPL provides no mechanism to describe the interplay between different protocol layers. As a result, developers still have to consult ancillary text that is written in natural language and distributed over various places in the specification. This fact not only complicates the manual implementation of corresponding message parsers, but also renders an automated generation of message parsers from TPL virtually impossible.

We strive to overcome some of the shortcomings of TPL that limit its usability or let alternative options appear preferable. To this end, we propose a generic model for parsing binary data. Based on this, we present eTPL, an enhanced version of TPL, along with `etpl-tool`, a software tool for automated parser generation from eTPL descriptions. eTPL essentially extends the syntax and semantics of TPL to improve its expressiveness and flexibility. Our work highlights a key advantage of building on TPL: the possibility to automatically generate message parsers from the TPL definitions given in the TLS specification requiring only minor adaptions. To the best of our knowledge, we are the first to propose and develop a more formally usable version of TPL.

Our paper is structured as follows: Section II presents our parsing model. Section III gives a brief overview of the original version of TPL before introducing eTPL, our enhanced version of TPL. Section IV presents `etpl-tool`, our software tool for processing eTPL definitions. Finally, Section V conlcudes and outlines future work.

II. A Generic Parsing Model

As a first step towards a more formally usable version of TPL, we define a generic model that describes how complex and nested messages with a flat and stream-like binary encoding shall be parsed. Our model has been inspired by the parsing paradigm that is inherent in the TLS protocol. Therefore, the model is particularly suitable to describe the parsing of TPL-specified data[1]. However, it is likewise applicable to a wide range of other protocols as well as data and message types.

A. Streams, Parsers, and Trees

In our model, *parsing* (or *dissecting*) denotes the process of transforming a flat stream of binary data (a message in its on-the-wire encoding) into a structured, tree-like representation. *Serializing* refers to the reverse process, turning the tree-like representation into a flat stream of binary data. Parsing is always performed linearly in forward direction without looking ahead. That is, during parsing there must not be any jumping back and forth inside the input stream to be parsed. Correspondingly, serialization is done by depth-first tree traversal.

We use the following terminology: a *type* is a recipe for parsing data according to its format definition. A type is either *atomic* without parsing-relevant substructure (e.g. an integer value or an opaque blob of raw data) or *composite* with several subelements (e.g. a network packet comprising multiple distinct fields). A *field* is an instance of a certain type, typically within the definition of a composite type. Here, the type of a field may be composite itself, inducing a nested structure and giving rise to the tree-like data representation.

A *parser* (or *dissector*) is the implementation of a type's parsing recipe. While linearly consuming the flat input stream of binary data, a parser grows the corresponding branch of the tree-like message representation. The parser of a composite type hands off the input stream to its subparsers (which correspond to the composite type's fields) one after the other[2]. Where fields depend on the parsing result of preceding fields, the corresponding subparsers get instantiated on-the-fly. The input stream to be parsed may be supplied in arbitrarily fragmented pieces, i.e. intermediate discontinuity in the stream does not affect the parsing result.

[1]Note that this model is *not* about how the (e)TPL description itself is parsed. A software tool that parses (e)TPL is presented in Section IV.

[2]This can be regarded as some kind of *datafall*, a waterfall of data.

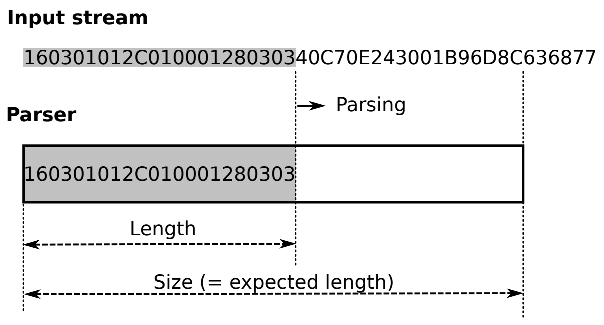

Fig. 1. Illustration of a parser consuming a flat stream of binary data. The amount of data consumed by a parser up to a certain point in time is referred to as its *length*. Optionally, a target value for the parser's length may be defined, referred to as the parser's *size*. A parser stops parsing when it is saturated or when its length has reached its size.

The amount of data consumed by a parser up to a certain point in time is referred to as its *length*. A parser consumes data from the input stream until it is saturated (e.g. all subparsers of a composite type are saturated) or until its length has reached an optionally defined target value, referred to as the parser's – or the type's – *size* (see Fig. 1). Unless undefined, a parser's size may either be determined by its type or by the parsing context.

While parsing, a possible source of inconsistency is a mismatch between a parser's size and its state of saturation. Such situations are most likely for parsers of composite types. For example, consider the parser of a composite type whose size is determined by the value of some previously parsed length field. An *overflow situation* is given if all subparsers of the composite type's parser have completed parsing while there is remaining input data to fill the gap between the parser's length and its size. An *underflow situation* is given if the parent parser's length has reached its size while not all subparsers have completed parsing. Overflow and underflow situations are illustrated in Fig. 2.

B. Generic Message Trees

In this section, we introduce *Generic Message Trees* (GMTs) as a conjoint concept for the implementation of parsers (as described previously) and the tree-like message representation generated by such parsers.

A GMT is an ordered rooted tree with nodes that each represent a particular component of a parsed message. A leaf node corresponds to an atomic message component without parsing-relevant substructure. Hence, leaf nodes hold their corresponding unparsable portion of the original input stream as raw data. Contrary to leaf nodes does an internal node correspond to a composite message component whose substructure is given recursively by its child nodes. Fig. 3 shows an excerpt of a GMT that represents a TLS message.

GMTs are closely related to the concept of parse trees. In contrast to the latter, however, GMTs constitute an active,

```
16 03 01 01 2C 01 00 01 28 03 03 40 C7 0E 24 30 01 B9 6D 8C 63 68 77 38 69 64 32 D3 E6 F9 49 10 7A AB AD 84 50 CD FF D6 A2 66 E4 00 00 92 C0 30 ...
--------------------------------------------------------------------------------------------------------------------------------------------------------
{0:TLSRecord}                    TLSRecord
|--[0:type]                      RecordType             | 16                       | handshake(22)
|--[1:version]                   ProtocolVersion        | 03 01                    | TLSv1(769)
|--[2:length]                    Integer                | 01 2C                    | 300
 --{3:msg}                       HandshakeMessage
    |--[0:type]                  HandshakeType          | 01                       | client_hello(1)
    |--[1:length]                Integer                | 00 01 28                 | 296
     --{2:msg}                   ClientHello
        |--[0:version]           ProtocolVersion        | 03 03                    | TLSv1_2(771)
        |--[1:random]            OpaqueBlob             | 40 C7 0E 24 30 01 B9 6D |
        |                                               | 8C 63 68 77 38 69 64 32 |
        |                                               | D3 E6 F9 49 10 7A AB AD |
        |                                               | 84 50 CD FF D6 A2 66 E4 |
        |--[2:session_id_length] Integer                | 00                       | 0
        |--[3:session_id]        OpaqueBlob             |                          |
        |--{4:cipher_suites}     ClientHello_cipher_suites
        |  |--[0:_N]             Integer                | 00 92                    | 146
        |   --{1:_V}             DynamicVector
        |     |--[0:CipherSuite] CipherSuite            | C0 30                    | TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384(49200)
       ...   ...                 ...                       ...
```

Fig. 3. Illustration of the raw data representation (octet string in hexadecimal notation above the horizontal line) as well as the GMT representation (tree structure below the horizontal line) of the first part of a TLS Record containing a TLS ClientHello message. GMT nodes are printed in the same order as they are traversed for serialization. The four columns of the GMT's visualization reflect (from left to right) the tree structure, the nodes' types, the leaf nodes' raw data representations (blank for internal nodes), and a human-readable representation (where applicable), respectively.

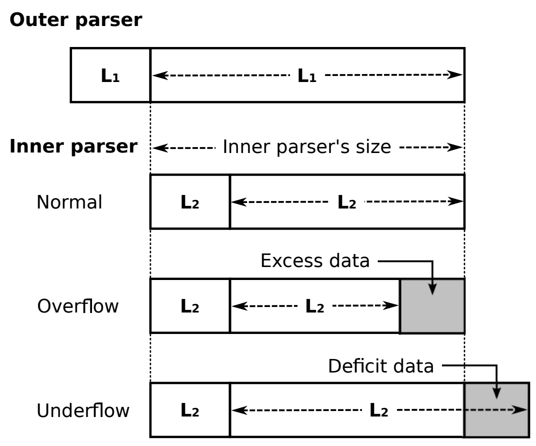

Fig. 2. Illustration of overflow and underflow situations while parsing nested message components with variable length. Inconsistencies in length information can result in an *overflow* situation with real *excess data* (data provided by the outer parser which the inner parser cannot attribute to any field) or an *underflow* situation with *deficit data* (data the inner parser expects but the outer parser does not provide).

dynamic, and self-manipulating data structure. Every GMT node implements the parser that corresponds to its type. In this way, GMT nodes corresponding to composite types self-inflate by creating child nodes while parsing the input stream[3]. Furthermore, a GMT may be navigated relative to any of its nodes and at any state of parsing. That is, the (partially grown) GMT that a parser (i.e. a parsing GMT node) is embedded in can serve as a dynamic source of context information which may affect the parsing process. For navigating a GMT (relative) paths similar to those of the folder structure of a file system can be used.

[3]That is, a top-level message is parsed by using a GMT node that corresponds to the message's top-level type as a seeding root node.

In order to facilitate efficient operations, GMTs have several further features: a GMT node may have a dynamic type component in addition to its immutable and static one. While its static component is determined by the type it implements, the dynamic type may determined by the concrete data the GMT has parsed at run-time. Furthermore, any GMT node may hold supplemental meta information to be passed to and from entities operating on the GMT.

III. TPL and Enhanced TPL

The syntax of TPL resembles the one of the "C" programming language and comprises basic types such as integer and enumeration types (`enum`), as well as lists (also referred to as vectors) of types, composite types (`struct`), and variants (`select`/`case`, i.e. dynamic choices within composite types). For a detailed introduction to TPL, please refer to any version of the TLS specification [8], [9], [10].

Fig. 4 shows the definition of a TLS ClientHello message using TPL, and at the same time illustrates one of the deficiencies of TPL: an ill-defined construct that is barely suitable for more than an informal guidance for human readers.

Generally, TPL allows to capture and concisely express *most* of the information that is required to parse messages. However, it fails to do so in way that would allow to automatically generate message parsers from TPL definitions.

In the following, we summarize the TPL enhancements that we propose as a delta between TPL and eTPL. Some of these enhancements have been inspired by *Parsifal* [17]. Our enhancements are disjunct and compatible to those introduced in [14] and [13]. Note that in the following we also list enhancements which are not strictly necessary to enable automated parser generation. Instead, we also propose enhancements that generally improve flexibility and applicability.

- **Explicit size of types and fields**: We introduce a language construct that allows to explicitly define the parser

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-2>;
    CompressionMethod
            compression_methods<1..2^8-1>;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ClientHello;
```

Fig. 4. Definition of a TLS ClientHello message with several fields using TPL (taken directly from the specification [10]). The definition of referenced types is not shown here. The syntax of TPL resembles the one of the "C" programming language, but is less formal: for example, `select (extensions_present)` is not well-defined and is barely suitable for more than an informal guidance for human readers.

size of types or fields. The size definition may either use fixed values or variable context information. For instance, this mechanism may be used to bind the size of a payload field to the decoded value of a preceeding length field.

- **Type parametrization**: We allow types to have parameters and define a language construct to set these parameters when instantiating a type. The acceptable numeric limits (minimum/maximum) of integer types are one example of useful type parameters.
- **Cross-layer fragmentation**: We introduce a mechanism in the language to express that a field within a composite type conveys a fragment of an independent binary stream. After parsing, fragments are handed off to a stream manager which tries to reassemble the original stream. Finally, the reassembled stream can be handed off to some suitable parser. This mechanism is necessary in order to be able to handle the fragmentation of handshake messages in DTLS.
- **Indefinite-length vectors**: We allow vectors to have an indefinite length. The parser of an indefinite-length vector consumes as much input data as it is supplied with. That is, whenever parsing of one of its subelements is complete, a new element is generated and parsing continues. Note that the parser of an indefinite-length vector still stops parsing if the parser's length reaches its size (if specified). Indefinite-length vectors can be used to parse a stream of consecutive protocol messages of the same type.
- **Element-based vector length**: We introduce a language construct that allows to specify the length of a vector in terms of the *number of its elements*. In the original version of TPL the length of a vector may only be given in terms of the *number of bytes* its on-the-wire encoding requires.
- **Fallback enumeration item:** In the definition of a enumeration type we allow to set a fallback (default) enumeration item. If defined, the fallback item matches the input data whenever no other item matches. This fallback mechanism is useful if not every single enumeration option a protocol supports is relevant for parsing or further processing.
- **Padding field**: We introduce a new padding type to be used within composite types. A padding field parses a variable amount of data to make the total length of the composite type's parser an integer multiple of a certain block length. The desired block length is a parameter of the padding type. A padding field may be used at any place within a composite type.
- **Optional fields**: We allow trailing fields within a composite type to be marked as *optional*. Optional fields only appear in the resulting GMT if all preceeding fields have been parsed completely while there is still more data available to be parsed. Note that the latter condition is not fulfilled if the parser's length has reached its size. For example, making the "`extensions`" field in Fig. 4 optional allows to avoid the ill-defined construction mentioned previously.
- **Transient fields**: We allow fields within a composite type to be marked as *transient*. A transient field is parsed normally but does not appear in the resulting GMT. This is useful in cases where a field is of no interest for further processing at all (as might be the case e.g. for padding fields).
- **Distinctive fields**: We allow fields within a composite type to be marked as *distinctive*. In essence, the dynamic type of a GMT node corresponding to a distinctive field contributes to the dynamic type of the parent GMT node, i.e. the GMT node corresponding to the composite type the distinctive field belongs to. This mechanism allows to propagate relevant dynamic information towards lower nesting levels, i.e. towards the GMT's root.
- **Extensible composite types**: We introduce *extensible* composite types. The parser of an extensible composite type does not run into saturation if all its fields have been parsed completely and does not blindly discard excess data in an overflow situation. In contrast, an additional opaque field is created dynamically as needed. The additional field absorbs all input data which a non-extensible parser would discard.
- **Bit granularity**: We use the bit as the smallest information quantum. The original version of TPL is focused on byte-oriented data where the length not only of every message has to be an integer multiple of a byte but also the length of every single field. This is in contrast to some protocols which use fields of fractional byte length, e.g. the four-bit data offset in the TCP header [2]. Bit granularity is applicable in any place where a length or a size is specified.

- **Dynamic parsing context**: We introduce language constructs that allow to make use of the dynamic parsing context provided by GMTs. The parsing context is given by the (partially grown) GMT a parser is embedded in. Concrete elements (i.e. GMT nodes or meta information associated to GMT nodes) are referenced using paths relative to the parser's position within the GMT.

Having listed our TPL enhancements, we briefly present a software tool for processing eTPL-based message definitions in the following section.

IV. A Versatile eTPL Software Tool

Implementing message parsers can be a daunting task. Typically, parsing is highly protocol-specific while requiring low-level source code. In order to support developers, a number of parsing tools exist, including the popular tool Wireshark [18] and various Python-based tools like Scapy [19], Construct [20], Hachoir [21], and Kaitai Struct [22]. However, we are not aware of any tool that uses TPL as the description language for deriving parsing routines[4].

Therefore, we developed `etpl-tool` [24], a software tool which comprises a parser for eTPL as well as a code generator for `C++`-based message parsers. Message parsing is based on `gmt-cpp` [25], a `C++` implementation of the parsing model and the GMT concept as it is described in Section II.

V. Conclusion and Outlook

In this paper, we presented a set of enhancements to the TLS Presentation Language (TPL) that are aimed at compensating for some of its deficencies.

As a first step, we presented a generic model for parsing complex binary-encoded messages. *Generic Message Trees* (GMTs), a powerful concept for a tree-like representation of messages, form an integral part of this model. Based on this, we proposed eTPL as a version of TPL that is usable in a more formal way than TPL is. eTPL provides language constructs that can capture important information which would otherwise have to be given in informal descriptions. These new language features pave the way for automatically generating message parsers from (e)TPL definitions. e.g. for TLS messages using their TPL definitions from the TLS specification.

Moreover, eTPL introduces features which are likely to extend its scope of applicability. For example, eTPL allows to use the bit as the smalles information quantum; for both whole messages as well as individual message fields.

Finally, we briefly presented `etpl-tool`, a software tool that automatically generates `C++` message parsers from eTPL definitions. While still being work in progress, our software tool may serve as a valuable tool whenever complex messages need to be translated between their flat binary on-the-wire and their structured tree-like respresentation. For instance, it allows rapid prototyping of network protocols and their implementations. We plan to extend eTPL with further useful features and improve our software tool accordingly.

[4]There is one tool which implements a rudimentary parser for TPL, though it does not seem to go any further than that [23].

Acknowledgment

We would like to thank Prof. Dr. Dirk Timmermann from Rostock University for his scientific support.

References

[1] J. Postel, "Internet Protocol," RFC 791, Internet Engineering Task Force, Sep. 1981.

[2] ——, "Transmission Control Protocol," RFC 793, Internet Engineering Task Force, Sep. 1981.

[3] J. Hui and P. Thubert, "Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks," RFC 6282, Internet Engineering Task Force, Sep. 2011.

[4] Olivier Dubuisson, *ASN.1 – Communication Between Heterogeneous Systems*, June 2000.

[5] Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan, "Interpreting the Data: Parallel Analysis with Sawzall," *Scientific Programming Journal: Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure*, vol. 13, no. 4, pp. 227–298.

[6] "Google Protocol Buffers," https://developers.google.com/protocol-buffers.

[7] Mark Slee, Aditya Agarwal and Marc Kwiatkowski, "Thrift: Scalable Cross-Language Services Implementation," http://thrift.apache.org/static/files/thrift-20070401.pdf, Tech. Rep.

[8] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," RFC 2246, Internet Engineering Task Force, Jan. 1999.

[9] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," RFC 4346, Internet Engineering Task Force, Apr. 2006.

[10] ——, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, Aug. 2008.

[11] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security," RFC 4347, Internet Engineering Task Force, Apr. 2006.

[12] ——, "Datagram Transport Layer Security Version 1.2," RFC 6347, Internet Engineering Task Force, Jan. 2012.

[13] "Intelligent Transport Systems (ITS); Security; Security header and certificate formats (TS 103 097)," European Telecommunications Standards Institute, Jun. 2015.

[14] "IEEE Standard for Wireless Access in Vehicular Environments – Security Services for Applications and Management Messages," *IEEE Std 1609.2-2013 (Revision of IEEE Std 1609.2-2006)*, pp. 1–289, April 2013.

[15] "IEEE Standard for Wireless Access in Vehicular Environments – Security Services for Applications and Management Messages," *IEEE Std 1609.2-2016 (Revision of IEEE Std 1609.2-2013)*, pp. 1–240, March 2016.

[16] Affeldt, Reynald and Marti, Nicolas, "Towards Formal Verification of TLS Network Packet Processing Written in C," in *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*, ser. PLPV '13, 2013, pp. 35–46. [Online]. Available: http://doi.acm.org/10.1145/2428116.2428124

[17] O. Levillain, "Parsifal: A Pragmatic Solution to the Binary Parsing Problem," in *2014 IEEE Security and Privacy Workshops*, May 2014, pp. 191–197.

[18] "Wireshark: A network protocol analyzer," https://www.wireshark.org/.

[19] "Scapy: the python-based interactive packet manipulation program & library," http://www.secdev.org/projects/scapy/.

[20] "Construct: A powerful declarative parser (and builder) for binary data," http://construct.readthedocs.io/en/latest/.

[21] "Hachoir: A Python library to view and edit a binary stream field by field," http://hachoir3.readthedocs.io/.

[22] "Kaitai Struct: A new way to develop parsers for binary structures," http://kaitai.io/.

[23] Rich Salz, "A parser for the TLS data description language used in the IETF RFC's," https://github.com/richsalz/tlsparser.

[24] "`etpl-tool`: A tool for processing (e)TPL message definitions and generating corresponding `C++` message parsers," https://github.com/phantax/etpl-tool.

[25] "`gmt-cpp`: An implementation of Generic Message Trees (GMTs) in C++ ," https://github.com/phantax/gmt-cpp.