[PoC] Parsifal: a pragmatic solution to the binary parsing problem

Olivier Levillain
French Network and Information Security Agency (ANSSI)
https://github.com/ANSSI-FR
firstname.lastname@ssi.gouv.fr

***Abstract*—Parsers are pervasive software basic blocks: as soon as a program needs to communicate with another program or to read a file, a parser is involved. However, writing robust parsers can be difficult, as is revealed by the amount of bugs and vulnerabilities related to programming errors in parsers. It is especially true for network analysis tools, which led the network and protocol laboratory of the French Network and Information Security Agency (ANSSI) to write custom tools. One of them, Parsifal, is a generic framework to describe parsers in OCaml, and gave us some insight into binary formats and parsers. After describing our tool, this article presents lessons we learned about format complexity, parser robustness and the role the language used played.**

In 2010, the Electronic Frontier Foundation scanned the Internet to find out how servers answered on the 443/TCP port worldwide [EFF12], [EB10a], [EB10b]. We studied this significant amount of data with custom tools, to gain thorough insight of the data collected [LEDM12]. However, the data contained legitimate SSL [DR08] messages, as well as invalid messages or even messages related to other protocols (HTTP, SSH). To face this challenge and extract relevant information, we needed robust tools on which we can depend. Yet, existing TLS stacks did not fit our needs: they can be limited (some valid options are rejected), laxist (they silently accept wrong parameters) or fragile (they crash on unexpected values, even licit ones). Thus, we decided to write our own parsers.

Our first attempt to write an SSL parser was with the Python language; it was quickly written and allowed us to extract some information. However, this implementation was unacceptedly slow. The second parser was in C++, using templates and object-oriented programming; its goal was to be flexible and fast. Yet, the code was hard to debug (memory leaks, segmentation faults on flawed inputs), and lacked extensibility since every evolution of the parsers required many lines of code.

So a new version was written, in OCaml: it used a DSL (Domain Specific Language) close to Python to describe the structures to be studied. This third parser was as fast as the previous one, less error-prone, but still needed a lot of lines to code simple features. That is why we finally decided to use a preprocessor to do most of the tedious work, letting the programmer deal only with what's important, structure description, while avoiding usual mistakes in low-level memory management. This last implementation, Parsifal, has all the properties we expected: efficient and robust parsers, written using few lines of code.

Our work originally covered X.509 certificates and SSL/TLS messages, but we soon tried Parsifal on other network protocols (BGP/MRT, DNS, TCP/IP stack, Kerberos) and on some file formats (TAR, PE, PCAP, PNG). Some of these parsers are still at an early stage, but one of the strength of Parsifal is that it is easy to describe part of a protocol, and focus only on what really needs to be dissected.

I. Parsifal: a quick tour

Parsifal relies on the idea that tedious code should not be written by humans since it can be generated. The basic blocks of a Parsifal parser are PTypes, that is OCaml types augmented by the presence of some manipulation functions: a PType `t` is composed of:

- the corresponding OCaml type `t`;
- a `parse_t` function, to transform a binary representation of an object into the type `t`;
- a `dump_t` function, that does the reverse operation, that is dumping a binary representation out of a constructed type `t`;
- a `value_of_t` function, to translate a constructed type `t` into an abstract representation, which can then be printed, exported as JSON, or analysed using generic functions.

There are essentially three kinds of PTypes:

- basic PTypes, provided by the standard library, like integers, strings, lists or ASN.1 DER basic objects;
- keyword-assisted PTypes, like structures, are descriptions using a pseudo-DSL integrated to the language thanks to a preprocessor (some of the offered constructions are presented in the remaining of the section);
- custom PTypes: when the corresponding structure is too complex to simply describe, you can always go back to manually writing the type and the functions.

In the remaining of this section we present some examples of keyword-assisted PTypes, the heart of Parsifal.

A. Examples of construction

Among the TLS messages, TLS alerts are used to signal a problem during the session. Such messages are simply composed of an alert level (one byte with two possible values) and an alert type (another byte). Here is the corresponding extract of the specification:

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
    close_notify(0),
     ...
    unsupported_extension(110),
    (255)
} AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

It is possible to describe such messages in Parsifal with the following code:

```
enum alert_level (8, UnknownVal AL_Unknown) =
  | 1 -> AL_Warning
  | 2 -> AL_Fatal

enum alert_type (8, UnknownVal AT_Unknown) =
  | 0 -> AT_CloseNotify
  ...
  | 110 -> AT_UnknownExtension

struct alert = {
  alert_level : alert_level;
  alert_type : alert_type
}
```

As a result, the preprocessor will generate three OCaml types, and some functions:

```
(* alert_level *)

type alert_level =
    AL_Warning
  | AL_Fatal
  | AL_Unknown of int

(* parse/dump/value_of functions *)
val parse_alert_level : input -> alert_level
val dump_alert_level : output -> alert_level -> unit
val value_of_alert_level : alert_level -> value


(* alert_type *)

type alert_type =
    AT_CloseNotify
  ...
  | AT_Unknown of int

(* ... 3 functions, similar to those relative *)
(*            to alert_level ...              *)

(* alert *)

type alert = {
  alert_level : alert_level;
  alert_type : alert_type;
}
val parse_alert : input -> alert
val dump_alert : output -> alert -> unit
val value_of_alert : alert -> value
```

The constructions available in Parsifal are enumerations (`enum` keyword), records (`struct`), choices allowing for types depending on a parameter (`union`), ASN.1 DER structures and choices (`asn1_struct` and `asn1_union`) and aliases (`alias` and `asn1_alias`).

B. A basic PNG parser

As an example of the conciseness of Parsifal code, this section briefly describes how to write a simple PNG parser. A PNG image is composed of a magic number (`"\x89PNG\r\n\x1a\n"`) followed by a list of chunks, which are described as follows:

Offset	Field	Size	Type
0	Chunk size `sz`	4	Big-endian integer
4	Chunk type	4	String
8	Chunk data	`sz`	Chunk-dependent
8 + `sz`	CRC	4	CRC 32

Using this first definition of the PNG format, it is easy to write some code in Parsifal:

```
struct png_chunk = {
  chunk_size : uint32;
  chunk_type : string(4);
  chunk_data : binstring(chunk_size);
  chunk_crc : uint32;
}

struct png_file = {
  png_magic : magic("\x89\x50\x4e\x47\x0d\x0a\x1a\x0a");
  chunks : list of png_chunk;
}
```

The first `struct` definition describes what a chunk is, and the `png_file` one what a PNG file is. Since Parsifal automatically generates the associated `parse`, `dump` and `value_of` functions, a complete tool extracting PNG chunks can be written by adding some lines:

```
let input = string_input_of_filename Sys.argv.(1) in
let png_file = parse_png_file input in
print_endline (print_value (value_of_png_file png_file));
```

The result of the compiled programs on a

```
value {
  png_magic: 89504e470d0a1a0a (8 bytes)
  chunks {
    chunks[0] {
      chunk_size: 13 (0x0000000d)
      chunk_type: "IHDR" (4 bytes)
      chunk_data: 000000140000001604030000000 (13 bytes)
      chunk_crc: 846176565 (0x326fa135)
    }
    chunks[1] {
      chunk_size: 15 (0x0000000f)
      chunk_type: "PLTE" (4 bytes)
      chunk_data: ccfffffffcc99996633333333000000 (15 bytes)
      chunk_crc: 1128124197 (0x433dcf25)
    }
    chunks[2] {
      chunk_size: 1 (0x00000001)
      chunk_type: "bKGD" (4 bytes)
      chunk_data: 04 (1 bytes)
      chunk_crc: 2406013265 (0x8f68d951)
    }
    chunks[3] {
      chunk_size: 77 (0x0000004d)
      chunk_type: "IDAT" (4 bytes)
      chunk_data: 78da63602005b8000184c5220804... (77 bytes)
      chunk_crc: 466798482 (0x1bd2c792)
    }
    chunks[4] {
      chunk_size: 86 (0x00000056)
      chunk_type: "tEXt" (4 bytes)
      chunk_data: 436f6d6d656e7400546869732061... (86 bytes)
      chunk_crc: 1290335428 (0x4ce8f4c4)
    }
    chunks[5] {
      chunk_size: 0 (0x00000000)
      chunk_type: "IEND" (4 bytes)
      chunk_data: "" (0 byte)
      chunk_crc: 2923585666 (0xae426082)
    }
  }
}
```

Of course, this is only the beginning, and one would likely want to improve the description by enriching the chunk content. This is actually a strength of our methodology: it is generally easy to describe the big picture and then to refine the parser to take into account more details.

C. Union and progressive enrichment

To illustrate how to enrich the chunk data, we begin with the image header, corresponding to the `"IHDR"` type. It is supposed to contain the following structure:

```
struct image_header = {
  width : uint32;
  height : uint32;
  bit_depth : uint8;
  color_type : uint8;
  compression_method : uint8;
  filter_method : uint8;
  interlace_method : uint8;
}
```

To use this new structure when dealing with a `"IHDR"` chunk, we have to create a union PType, depending on the chunk type:

```
union chunk_content [enrich;param ctx] (
    UnparsedChunkContent) =
| "IHDR" -> ImageHeader of image_header(ctx)
```

Finally, we have to rewrite the `chunk_data` field in the `png_chunk` structure:

```
  chunk_data : container(chunk_size) of chunk_content(
    chunk_type);
```

It should be clear how to enrich other chunk types from now: write the PType corresponding to the chunk content, and then add a line in the `chunk_content union`. When facing an unknown chunk type, `parse_chunk_content` will produce an `UnparsedChunkContent` value containing the unparsed string.

D. PContainers: a useful concept

As we began using Parsifal to describe various file formats and network protocols, it dawned on us that it might be useful to create another concept that could be reused: PContainers. Initially, the only existing containers were lists, but the notion of containers can be broader: the abstraction of a container containing a PType make it possible to automate some processing that has to be done at parsing and/or dumping time. Here are some examples:

- encoding: hexadecimal, base64;
- compression: DEFLATE, zLib or gzip containers;
- safe parsing: some containers provide a fall-back strategy when the contained PType can not be parsed;
- miscellaneous checks: CRC containers are useful to check a CRC at parsing time and to generate the CRC value at dumping time.

There again, the idea is to reuse code to reduce the time spent writing the same code time and again. Here is an example of a PNG chunk caled `iCCP` (Embedded Profile), which contains a string that should not exceed 80 characters, a byte field and a compressed string. Using containers, the chunk can then be described as follows:

```
struct embedded_profile = {
  name : length_constrained_container(AtMost 80) of
    cstring;
  compression_method : uint8;
  compress : ZLib.zlib_container of string;
}
```

E. A custom PType

Finally, when it is not possible to describe a PType using keywords like `struct` or `union`, it is always possible to write the Ptype from scratch.

Assuming it does not exist already in the standard library, here is how you could describe a null-terminated string as a custom PType. The *intended* type is a simple string, but the corresponding `parse` and `dump` functions have to be written manually:

```
type cstring = string

let rec parse_cstring input =
  let next_char = parse_char input in
  if next_char <> '\x00'
  then (String.make 1 next_char) ^ (parse_cstring input)
  else ""

let dump_cstring buf s =
  POutput.add_string buf s;
  POutput.add_char buf '\x00'
```

II. Results so far

For the moment, our main goal has been to write robust parsers to analysing data but, most importantly, to understand how some protocols or file formats really work. The initial parsers covered SSL/TLS messages and X.509 certificates, but we have been describing more and more formats, sometimes to study new areas, sometimes as a challenge to test Parsifal's expressivity. At the beginning, these challenges required changes in Parsifal preprocessor or standard library, but such modifications have become rare lately, which means we have reached a balance between language expressivity and implementation complexity. Here are some example of Parsifal developments and the issues encountered.

a) DNS: At first analysing DNS messages was a challenge to better understand the notion of *parsing context*. DNS resource records can indeed be *compressed* by referencing to previously read domain names. This form of compression requires a context retaining the domain names encountered during parsing. The same effort must be done when dumping a message: record the offsets of the names produced and reuse them to compress the result.

b) PE: To better understand UEFI, some co-workers had to study Portable Executable programs[1]. To this aim, they tried Parsifal, and had to deal with what strikes us as a bad idea: non-linear parsing. To analyse a PE, you have to walk through the file using offsets, forwards and sometimes backwards. This is now possible in Parsifal, but it is very hard to check properties on such file formats[2]

[1] Indeed, the format of Windows executables is used in UEFI.

[2] Actually, we discovered that the specification was so complex that it theoretically allowed baroque structure interleaving, whereas common tools assume a straightforward layout. This leads to documented inconsistencies when checking a signature.

c) Other formats: MRT/BGP, PNG, JPG, Kerberos messages...

The contribution of Parsifal to security is twofold. First it can help provide sound tools to analyse complex file formats or network protocols. Secondly we can implement robust detection/sanitization systems.

III. Lessons learned

Parsifal has been developped at ANSSI for more than 3 years and its interface is becoming rather stable. In this section, we discuss several lessons we learned while writing binary parsers.

On formats

There exists such thing as a good or a bad format. Formats relying on simple TLV (Tag/Length/Value) structures are very easy to parse. Moreover, they allow for partial parsing (for example when considering unknown extensions). A concrete example of such a good format, according to our experience, is PNG: chunks respect the TLV logic and the corresponding structures are rather simple.

On the contrary, several properties leads to error-prone parsers and should be avoided. For example, non-linear parsing makes it unnatural to check whether the data we parse are in a licit location; this is the case in PE and JFIF formats, the latter actually being similar to a filesystem with directories and entries. Another undesirable property is when parsers are required to know every option to correctly interpret the file, such as the DVI format; DVI files are sequences of variable-length commands, but you have to know how to interpret a command to know its length, to avoid command misalignment which would completely change the meaning of the rest of the file.

On the language

Parsifal was written in OCaml, a strongly-typed functional language. Here are the advantages of this language, regarding our goal to write parsers:

- the language is naturally expressive, which helps to write concise code;
- higher order functions allows to write containers easily (e.g. `parse_base64_container` naturally takes as an argument a parse function);
- as the memory is handled automatically by the garbage collector, several classes of attacks (double frees, buffer overflows) are impossible[3];
- pattern matching exhaustiveness check is a very helpful feature when dealing with complex structures or algorithms, since the compiler will remind you of unhandled cases.

On the process

In the end, our choice to automatically generate the tedious parts of the code paid: Parsifal allows to quickly write binary parsers, even for complex formats. What's more, the description process turned out to be accessible, even for persons with no initial OCaml (or functional programming) background.

However, you should not try to add everything in your DSL or in your constructions. Some parts of a parser are so complicated that they should remain manually written. That is the reason why we kept the possibility to fall back to manual PTypes when needed.

IV. Conclusion

Parsifal is a generic framework to describe parsers in OCaml which has been used to describe several file formats and network protocols. From our point of view, this tool has all the expected properties: expressive language, code conciseness, efficient and robust programs. Moreover, Parsifal allows for an incremental description, which is useful to progressively learn the internals of a new format.

Parsifal is publicly available under an open source license since June 2013 (https://github.com/ANSSI-FR/parsifal) and has been the subject of a tutorial during a conference last year [LDM13]. The git repository contains examples of step-by-step code description concerning TAR archives, the DNS protocol, PNG images and PKCS#10 CSR (Certificate Signing Request).

References

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[EB10a] P. Eckersley and J. Burns. An Observatory for the SSLiverse, Talk at Defcon 18, 2010.

[EB10b] P. Eckersley and J. Burns. Is the SSLiverse a safe place?, Talk at 27C3, 2010.

[EFF12] Electronic Frontier Foundation. The EFF SSL Observatory. https://www.eff.org/observatory, 2010-2012.

[LDM13] Olivier Levillain, Hervé Debar, and Benjamin Morin. Parsifal: writing efficient and robust binary parsers, quickly. In *8th International Conference on Risks and Security of Internet and Systems (CRISIS)*, 2013.

[LEDM12] Olivier Levillain, Arnaud Ebalard, Hervé Debar, and Benjamin Morin. One Year of SSL Measurement. In *ACSAC*, 2012.

[3]However, this alone is far from perfect, since we may avoid arbitrary execution but not a fatal exception.