Parsifal: writing efficient and robust binary parsers, quickly

Olivier Levillain[*†], Hervé Debar[†] and Benjamin Morin[*]

[*]ANSSI
51 boulebard Latour Maubourg
Paris, France
first.last@ssi.gouv.fr

[†]Télécom Sud Paris
9 rue Charles Fourier
Evry, France
first.last@telecom-sudparis.eu

September 2, 2013

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 hard, as is revealed by the amount of bugs and vulnerabilities linked to programming errors in parsers. In particular, network analysis tools can be very tricky to implement: for example, the Wireshark project regularly publishes security patches on its various dissectors[1].

As security researchers, we need robust tools on which we can depend. The starting point of Parsifal was a study of large amounts of SSL data [1]. The data collected contained legitimate SSL [2] messages, but it also contained invalid messages and other protocols (HTTP, SSH). To face this challenge and extract relevant information, we wrote several parsers, using different languages, which resulted in Parsifal, an OCaml-based parsing engine. Writing parsers and analysing data helped us better understand SSL/TLS, but also X.509 [3] and BGP/MRT [4, 5]. More recently, we have begun studying Kerberos messages.

The contribution of Parsifal to security can be twofold: provide sound tools to analyse complex file formats or network protocols, and implement robust detection/sanitization systems. The goal of this tutorial is to present Parsifal and to use it to write a network protocol parser (DNS) and a file format parser (PNG). The PNG parser will then be used to begin a PNG sanitizer. Alternatively, an X.509 certificate signing request validator can be implemented.

Olivier Levillain is head of the Network and Protocols Laboratory at the French Network and Information Security Agency (ANSSI). His current research interests are web browser security, network protocols (SSL/TLS in particular) and the (in)adequation of programming languages to security goals. He is currently a PhD student under Pr. Hervé Debar and Dr. Benjamin Morin's advisory.

1 Project history

In 2010, the EFF scanned the Internet to find out how servers worldwide answered on the 443/TCP port [6, 7, 8]. We studied this significant amount of data with custom tools, to gain thorough insight of the data collected [1].

Our first attempt to write an SSL parser was in Python; it was quickly written and allowed us to extract some information. However, this implementation was fundamentally 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, and contained too many lines.

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 decided to use a preprocessor to do most of the work, letting the programmer deal only with what's important: structure description. This last implementation, Parsifal, has all the properties 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,

[1]Since the beginning of the 2013 year, 29 CVE have been published on Wireshark.

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 insist only on what really needs to be dissected.

2 Parsifal principle: PTypes

Basically, Parsifal allows you to use PTypes, which are 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.

PTypes are usually built using new keywords: `enum`, `struct`, `union`, etc. However, when dealing with unsupported cases, it is also possible to add custom PTypes, by writing directly the `t` type and the corresponding functions. A lot of basic PTypes are already present in the core library.

2.1 Examples of construction

Among the TLS messages, 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). An extract of the specification is given in figure 1. It is possible to describe such messages in Parsifal with the code given in figure 2. As a result, the preprocessor will generate three OCaml types, and some functions, as presented in figure 3.

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`).

2.2 Examples of base PTypes

Parsifal already understands some basic types: integers, string, IPv4/IPv6 addresses, lists, arrays, *magic* numbers, and ASN.1 basic objects.

Moreover, Parsifal provides an abstraction, the containers, allowing to wrap a PType using some processing. For example, `base64_container` and `hex_container` allow to work with encoded types transparently; `deflate_container` and `zlib_container` uncompress the output when parsing and compress when dumping. Finally, `pkcs1_container` are an elegant way to decrypt the content of a PKCS#1 value and read the enclosed type when given the corresponding key during parsing type[2].

The idea of the core library is to provide most of the basic types and transformations used in protocols and file formats. One of the advantages of Parsifal is that it is easy to partially implement a protocol to only interpret the cases of interest. It also allows for a progressive description of a format.

As our initial goal was to handle a lot of data, including corrupted messages or data not conforming to the specification, it is possible to implement either strict parsers or flexible ones. This was useful for example when dissecting X.509 corrupted fields: an error deep in the certificate should not necessarily invalidate the whole certificate nor the TLS message containing it.

3 Related work

Parsifal may seem similar to two Python projects: `scapy` [9], a toolbox to parse and forge network packets and `hachoir` [10], a generic framework for binary file manipulation library. However, as an OCaml development, Parsifal allows for better perfomance when compiled to native binaries. In our experience, it is as efficient

[2] This proved to be useful to process smoothly Kerberos PKINIT messages.

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

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

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

Figure 1: Specification of `tls_alert` messages (from RFC 5246 [2]).

```
enum tls_alert_level (8, UnknownVal AL_Unknown) =
  | 1 -> AL_Warning, "Warning"
  | 2 -> AL_Fatal, "Fatal"

enum tls_alert_type (8, UnknownVal AT_Unknown) =
  | 0 -> AT_CloseNotify, "CloseNotify"
  ...
  | 115 -> AT_UnknownPSKIdentity, "UnknownPSKIdentity"

struct tls_alert = {
  alert_level : tls_alert_level;
  alert_type : tls_alert_type
}
```

Figure 2: Parsifal description of `tls_alert` messages.

```
(* tls_alert_level *)

type tls_alert_level =
    AL_Warning
  | AL_Fatal
  | AL_Unknown of int

(* Conversion functions to/from int/string *)
val int_of_tls_alert_level : tls_alert_level -> int
val string_of_tls_alert_level : tls_alert_level -> string
val tls_alert_level_of_int : int -> tls_alert_level
val tls_alert_level_of_string : string -> tls_alert_level

(* parse/dump/value_of functions *)
val parse_tls_alert_level : input -> tls_alert_level
val dump_tls_alert_level : output -> tls_alert_level -> unit
val value_of_tls_alert_level : tls_alert_level -> value


(* tls_alert_type *)

type tls_alert_type =
    AT_CloseNotify
  ...
  | AT_Unknown of int

(* 7 functions, similar to those relative to tls_alert_level *)


(* tls_alert *)

type tls_alert = {
  alert_level : tls_alert_level;
  alert_type : tls_alert_type;
}
val parse_tls_alert : input -> tls_alert
val dump_tls_alert : output -> tls_alert -> unit
val value_of_tls_alert : tls_alert -> value
```

Figure 3: Corresponding OCaml code generated (extracts of the interface).

as corresponding C implementations[3]. What's more, OCaml is a well-defined, sound language which brings some safety guarantees regarding memory management that C or Python do not.

Other preprocessors and libraries exist in the OCaml environment, but they do not offer a comprehensive framework to describe complex structures as Parsifal does. For example, the `bitstring` [11] project adds pattern matching on bitsrings, which is only a part of the types handled by our tool.

4 Tutorial goals and outline

Parsifal has already been presented internally and a version of this tutorial has been used to teach several people to using Parsifal. The overall tutorial was given over a 3-hour session. All the material (this short paper, the slides and the code snippets) will be available on the GitHub repository.

The tutorial is intended for developpers and researchers who need to manipulate complex binary file formats or network protocols. The audience would need a basic background in functional programming (OCaml language preferably).

There are three main goals for this tutorial:

- learn to use basic Parsifal constructions;
- write a simple DNS client;
- code a PNG sanitizer

Depending on the audience's interests, the DNS or PNG implementation could be replaced by an X.509 Certificate Signing Request validator.

4.1 Parsifal presentation

As was done in the first part of this document, the tutorial begins with a brief history of the project, and the motivation for writing robust and efficient parsers.

Then, the principle behind Parsifal are presented: the PTypes and the methods to generate them automatically using a preprocessor. This part can be illustrated by several examples of constructions: how to write them and what kind of code is generated.

4.2 Downloading and installing Parsifal

After this short introduction, the tutorial consists of downloading Parsifal source code from the public repository on GitHub (`https://github.com/ANSSI-FR/parsifal`), installing it and using it to write parsers step-by-step.

Members of the audience interested in manipulating on their computers would have to install the OCaml languages and the libraries on which Parsifal depends.

4.3 DNS step-by-step

The first parser proposed for the tutorial is DNS, which is a rather simple protocol, but contains some subtleties. That is why it is a good candidate to begin using Parsifal.

After presenting DNS message formats, the implementation goes as follows:

- description of DNS enumerations (record types and classes);
- first implementation of labels and domains using structures;
- description of more structures (`question`, `rr` and the overall `dns_message` type);
    - at this point, it is possible to parse and print example requests and responses;
- progressive specification of the ressource records (A, CNAME, MX) using a union;
- custom rewrite of labels and domains to handle the compression;

[3] For example, the time needed to parse certificates is comparable with the `openssl x509` command.

- implementation of the UDP connexion
  - at this point, it is possible to have a minimalistic DNS client.

4.4 PNG step-by-step

Another example can be written to parse PNG files. The different steps of the implementation would be:

- description of the outter structure of the image file (essentially a list of chunks);
  - at this point, it is possible to implement a tool printing the types of the chunks present in a given file;
  - it is also possible to write a basic PNG filter, rewriting the file without the comment chunks (`tEXt` for example);
- rewrite of the chunk container to automatically generate the length and the CRC when dumping the chunk;
- as for DNS resource records, it is possible to use a union to progressively describe the different chunk types, starting with the mandatory ones (IHDR, PLTE, IDAT and IEND);
  - at this point, it is possible to write a simple PNG sanitizer that combines all IDAT chunks into one, and rewrites the image file uncompressed. This way, the compression is handled by `zlib_container`, a pure OCaml robust implementation, and not by the end application, which may embed an out-of-date and flawed `zlib`.

4.5 X.509 CSR validation

Another format worth implementing is X.509 Certificate Signing Request. In 2009, Moxie Marlinspike showed how to get a signed certificate for the wrong domain by sending a subject containing null characters [12]. In all the cases, it might be useful to constrain the CSR before it hits the certification authority: remove useless attributes, check the signature before-hand, clean up the subject to follow a given policy. Here is how it can be done using Parsifal:

- description of the ASN.1 structures corresponding to CSRs;
  - at this point, it is possible to implement a tool printing CSRs;
- filter out attributes;
- add a trivial policy on subjects (only one CN field, no null characters, etc.);
- check the signature and the size of the key;
  - at this point, it is possible to write a simple X.509 CSR validator, that can be used as a gate keeper to validate requests.

5 Conclusion

For our needs, we wrote several parsers to analyse a lot of SSL/TLS data. As the collected messages were sometimes corrupted or invalid, standard tools did not allow for sound and robust dissection. Parsifal, an OCaml-based parsing engine, allowed us to gain insight into several important protocols. Parsifal also proved to be versatile and might be useful to the security community to write efficient and robust binary dissectors.

Acknowledgment

The work in this paper has been partially sponsored by the EC 7th Framework Programme as part of the ICT Vis-Sense project (grant no. 257497). The authors would like to thank the Applied and Fundamental Research Division of the French Network and Information Security Agency (ANSSI) for their comments and suggestions.

References

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

[2] 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.

[3] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008.

[4] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006. Updated by RFC 6286.

[5] L. Blunk, M. Karir, and C. Labovitz. Multi-Threaded Routing Toolkit (MRT) Routing Information Export Format. RFC 6396 (Proposed Standard), October 2011.

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

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

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

[9] P. Biondi and the Scapy community. Scapy. `http://www.secdev.org/projects/scapy/`, 2003-2012.

[10] V. Stinner. Hachoir. `https://bitbucket.org/haypo/hachoir/wiki/Home`, 2009-2012.

[11] R. Jones. bitstring. `http://code.google.com/p/bitstring/`, 2003-2012.

[12] Moxie Marlinspike. More Tricks For Defeating SSL In Practice, 2009.