Parsifal: writing efficient and robust binary parsers, quickly

Olivier Levillain, Hervé Debar and Benjamin Morin

ANSSI / Télécom Sud Paris

October 23rd 2013
Prerequisites for this tutorial

- Linux system (tested on recent Debian or Ubuntu)
- an internet connection
- programming background
- (recommended) functional programming notions (ML/OCaml/Haskell)
Prerequisites for this tutorial

- Linux system (tested on recent Debian or Ubuntu)
- an internet connection
- programming background
- (recommended) functional programming notions (ML/OCaml/Haskell)

```
apt-get install ocaml ocaml-findlib
apt-get install liblwt-ocaml-dev
apt-get install libcryptokit-ocaml-dev
```

```
apt-get install git make
```
Outline

Motivation

Installation

Parsifal’s \( \mathcal{P}\)Types

PNG tools

A CSR validator

Parsifal: past, present and future
Motivation

Outline

Motivation

Installation

Parsifal’s $\mathcal{P}$Types

PNG tools

A CSR validator

Parsifal: past, present and future
Origin of our work on parsers

In 2010, the EFF published datasets of IPv4 scans on port 443:

- 36 GB of raw data (180 GB when considering other sources)
- half of the answers are HTTPS... the rest can be exotic
- some servers do not behave according to the specifications

Our goal: analyse the data to understand what our browsers must deal with

Existing TLS stacks did not meet our needs since they are either:

- limited (rejecting valid parameters)
- laxist (silently accepting invalid parameters)
- fragile (crashing on unexpected data)

For our study (a paper accepted at ACSAC 2012), we wrote a lot of tools using different languages.
SSL/TLS: a brief tour of the messages
Binary protocols/formats studied

Since 2011, we studied several formats:

- X.509 certificates
- SSLv2 and TLS messages
- BGP and MRT messages
- TAR archives (tutorial)
- DNS messages
- PE and PCI expansion ROM (UEFI)
- PNG and JPEG
- Kerberos
Where is the difficulty?

Protocol messages and file formats may be very complex to parse:

- variable length fields (TLS)
- context-dependent fields (DNS so-called compression)
- non-linear parsing (EXIF metadata)
- and a lot of tedious work that seems the same...
Where is the difficulty?

Protocol messages and file formats may be very complex to parse:

- variable length fields (TLS)
- context-dependent fields (DNS so-called compression)
- non-linear parsing (EXIF metadata)
- and a lot of tedious work that seems the same...

Question: how many ways can an integer be encoded?
Motivation

Where is the difficulty?

Protocol messages and file formats may be very complex to parse:

▶ variable length fields (TLS)
▶ context-dependent fields (DNS so-called compression)
▶ non-linear parsing (EXIF metadata)
▶ and a lot of tedious work that seems the same...

Question: how many ways can an integer be encoded?

▶ big-endian / little-endian
Motivation

Where is the difficulty?

Protocol messages and file formats may be very complex to parse:

▶ variable length fields (TLS)
▶ context-dependent fields (DNS so-called compression)
▶ non-linear parsing (EXIF metadata)
▶ and a lot of tedious work that seems the same...

Question: how many ways can an integer be encoded?

▶ big-endian / little-endian
▶ ASN.1 DER proposes three different ways (length, tag, integer value)
Motivation

Where is the difficulty?

Protocol messages and file formats may be very complex to parse:

- variable length fields (TLS)
- context-dependent fields (DNS so-called compression)
- non-linear parsing (EXIF metadata)
- and a lot of tedious work that seems the same...

Question: how many ways can an integer be encoded?

- big-endian / little-endian
- ASN.1 DER proposes three different ways (length, tag, integer value)
- Protobuf has a somewhat efficient bigint
Motivation

Where is the difficulty?

Protocol messages and file formats may be very complex to parse:

- variable length fields (TLS)
- context-dependent fields (DNS so-called compression)
- non-linear parsing (EXIF metadata)
- and a lot of tedious work that seems the same...

Question: how many ways can an integer be encoded?

- big-endian / little-endian
- ASN.1 DER proposes three different ways (length, tag, integer value)
- Protobuf has a somewhat efficient bigint
- TAR uses octal strings (the "101" string means 65)
Motivation

Parsers’ expected properties

Our objective was multiple:

- analyse huge amounts of data to understand protocols:
  - SSL data (using datasets like those of the EFF)
  - BGP routing information (using RIS collectors)
- write robust tools to detect anomalies
- normalise messages or files to remove vulnerabilities

To this purpose, the properties expected are:

- robustness
- efficiency
- ease of development
  - expressive language
  - concise code
  - reusable code
Demonstration

- a tool used for our paper: mapAnswers
- grab and analyse certificates from an HTTPS server
- analyse a PCAP trace using parsifal
Outline

Motivation

Installation

Parsifal’s \( \mathcal{P} \)Types

PNG tools

A CSR validator

Parsifal: past, present and future
Prerequisites (Debian/Ubuntu)

apt-get install ocaml ocaml-findlib
apt-get install liblwt-ocaml-dev
apt-get install libcryptokit-ocaml-dev

apt-get install git make

Remarks:

- Debian Squeeze: patch needed...
- Debian Wheezy: OK
- Ubuntu Raring/Quantal: universe repository is needed
Installation

Downloading and compiling Parsifal

GitHub repository

git clone https://github.com/ANSSI-FR/parsifal.git
cd parsifal
Installation

**Downloading and compiling Parsifal**

**GitHub repository**

```bash
git clone https://github.com/ANSSI-FR/parsifal.git
cd parsifal
```

**Compilation and installation**

```bash
make
LIBDIR=$HOME/.ocamlpath BINDIR=$HOME/bin make install
export OCAMLPATH=$HOME/.ocamlpath
PATH=$HOME/bin:$PATH
```
Installation

Downloading and compiling Parsifal

GitHub repository

git clone https://github.com/ANSSI-FR/parsifal.git
cd parsifal

Compilation and installation

make
LIBDIR=$HOME/.ocamlpath BINDIR=$HOME/bin make install
export OCAMLPATH=$HOME/.ocamlpath
PATH=$HOME/bin:$PATH

Installation check: grab and study some certificates

mkdir tests && cd tests
probe_server -H www.google.com extract-certs
x509show --subject *.pem
x509show --modulus *.pem
x509show -g "**.distributionPoint" *.pem
Outline

Motivation

Installation

Parsifal’s PTypes

PNG tools

A CSR validator

Parsifal: past, present and future
PTypes

In Parsifal, a PType consists of:

- an arbitrary OCaml type \( t \)
- a parsing function \( \text{parse}_t \)
- a dumping function \( \text{dump}_t \)
- a function producing a higher-level value \( \text{value_of}_t \)
In Parsifal, a \( \mathcal{P} \)Type consists of:

- an arbitrary OCaml type \( t \)
- a parsing function \( \text{parse}_t \)
- a dumping function \( \text{dump}_t \)
- a function producing a higher-level value \( \text{value}_t \)

There are three types of \( \mathcal{P} \)Types:

- basic \( \mathcal{P} \)Types (\( \text{uint8} \), \( \text{string} \), \( \text{list} \))
- constructed \( \mathcal{P} \)Types, obtained using keywords and short descriptions
- custom \( \mathcal{P} \)Types
RFC 5246 (TLSv1.2) encodes TLS version field using a 16-bit value.

```python
enum tls_version (16, UnknownVal UnknownVersion) =
    | 0x0002 -> SSLv2
    | 0x0300 -> SSLv3
    | 0x0301 -> TLSv1
    | 0x0302 -> TLSv1_1
    | 0x0303 -> TLSv1_2
```
SSL/TLS: describing alert messages
Structures (1/2)

RFC 5246 (TLSv1.2) extract:

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

enum {
    close_notify(0),
    unexpected_message(10),
    ...
    unsupported_extension(110),
    (255)
} AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```
Parsifal implementation:

```c
enum tls_alert_level (8, Exception) =
    | 1  -> Warning
    | 2  -> Fatal

enum tls_alert_type (8, UnknownVal UnknownAlertType) =
    | 0  -> CloseNotify
    | 10 -> UnexpectedMessage
    ...
    | 110 -> UnsupportedExtension

struct tls_alert =
{
    alert_level : tls_alert_level;
    alert_type : tls_alert_type
}
```
SSL/TLS: handshake messages depend on a type
Unions

Supposing `client_hello` and `server_hello` have been written, we can write:

```plaintext
enum hs_message_type (8, UnknownVal HT_Unknown) =
    | 1 -> HT_ClientHello
    | 2 -> HT_ServerHello
    | ...

union handshake_content [enrich] (Unparsed_HSContent) =
    | HT_ClientHello -> ClientHello of client_hello
    | HT_ServerHello -> ServerHello of server_hello
    | ...

struct handshake_msg = {
    handshake_type : hs_message_type;
    handshake_content : container [uint24] of
        handshake_content(handshake_type)
}
```
Other constructions

- containers (base64, ASN.1 encapsulations)
- asn1_union and asn1_struct
- struct may contain bit fields
Outline

Motivation

Installation

Parsifal’s $P$Types

PNG tools

A CSR validator

Parsifal: past, present and future
PNG 101: a list of chunks

A PNG has the following structure:

▶ a magic string identifying the format ("\x89PNG\r\n\x1a\n")
▶ a list of chunks

Each chunk contains:

▶ the chunk size (on a big endian 32 bits)
▶ the chunk type (a 4 character string)
▶ the chunk data (whose length was given earlier)
▶ a checksum (a CRC32)
Our first project

cd parsifal
./mk_project pngtools
cd pngtools
make
./pngtools
Let’s implement PNG, one step at a time

- Write a program checking the magic string
Let’s implement PNG, one step at a time

- Write a program checking the magic string
- Add support for chunks
Let’s implement PNG, one step at a time

- Write a program checking the magic string
- Add support for chunks
- Simple tool to print the chunk types
Let’s implement PNG, one step at a time

- Write a program checking the magic string
- Add support for chunks
- Simple tool to print the chunk types
- Chunk Filter: only keep critical chunks
Let’s implement PNG, one step at a time

- Write a program checking the magic string
- Add support for chunks
- Simple tool to print the chunk types
- Chunk Filter: only keep critical chunks
- First custom PType: crc_check
Let’s implement PNG, one step at a time

- Write a program checking the magic string
- Add support for chunks
- Simple tool to print the chunk types
- Chunk Filter: only keep critical chunks
- First custom PType: crc_check
- First union to enrich chunk content
Let’s implement PNG, one step at a time

- Write a program checking the magic string
- Add support for chunks
- Simple tool to print the chunk types
- Chunk Filter: only keep critical chunks
- First custom PType: crc_check
- First union to enrich chunk content
- ...

O. Levillain  
https://github.com/ANSSI-FR/parsifal  
October 23rd 2013
Conclusion on PNG

- step by step description of the PNG format
- basic validation of the structure
- possibility to add checks
- such a validation/sanitisation would thwart some known vulnerabilities on libpng:
  - CVE-2011-0408
  - CVE-2008-1382
  - CVE-2007-5266
  - CVE-2007-2445
  - CVE-2004-0597
  - ...
Outline

Motivation

Installation

Parsifal’s $\mathcal{P}Types$

PNG tools

A CSR validator

Parsifal: past, present and future
Null characters in Common Names

In 2009, Moxie Marlinspike showed an attack on Certificate Signing Requests:

- a CSR is submitted for www.mybank.com\x00.attacker.com
- the Certification Authority thinks the signed domain is under attacker.com
- most TLS clients (written in C) stop at the first null character
- CVE-2009-2408

Recently, a similar vulnerability has resurfaced in different languages:

- python (CVE-2013-4238 and)
- ruby (CVE-2013-4073)
CSR specification
Let’s start from the RFC 2314 (PKCS#10):

\[
\text{CertificateRequestInfo} ::= \text{SEQUENCE} \{ \\
\quad \text{version} \quad \text{Version}, \\
\quad \text{subject} \quad \text{Name}, \\
\quad \text{subjectPublicKeyInfo} \quad \text{SubjectPublicKeyInfo}, \\
\quad \text{attributes} \quad [0] \quad \text{IMPLICIT} \quad \text{Attributes} \}
\]

\text{Version} ::= \text{INTEGER}

\text{Attributes} ::= \text{SET OF} \text{ Attribute}

\[
\text{CertificateRequest} ::= \text{SEQUENCE} \{ \\
\quad \text{certificateRequestInfo} \quad \text{CertificateRequestInfo}, \\
\quad \text{signatureAlgorithm} \quad \text{SignatureAlgorithmIdentifier}, \\
\quad \text{signature} \quad \text{Signature} \}
\]

\text{SignatureAlgorithmIdentifier} ::= \text{AlgorithmIdentifier}

\text{Signature} ::= \text{BIT STRING}
Parsifal implementation

Most of the fields have already been implemented for X.509 certificates:

```c
asn1_struct certificationRequestInfo = {
    version : der_smallint;
    name : distinguishedName;
    subjectPublicKeyInfo : subjectPublicKeyInfo;
    attributes : der_object;
};

asn1_struct certificationRequest = {
    certificationRequestInfo : certificationRequestInfo;
    signatureAlgorithm : algorithmIdentifier;
    signatureValue : bitstring_container of signatureType
};
```
Some checks on CSR

Now we have a (partial) description of a CSR, we may want to check

- the RSA signature
- that the subject does not contain any null character
Conclusion on CSRs

- Parsifal can help reuse code very easily
- ASN.1 structures can be described very quickly using
- this kind of robust validator could be used in front of real applications, to drop invalid requests early in the process
Outline

Motivation

Installation

Parsifal’s $\mathcal{P}$Types

PNG tools

A CSR validator

Parsifal: past, present and future
Conclusion

Past:
- since 2011, some protocols and file formats described
- Parsifal tools used to analyse SSL/TLS data

Present:
- work in progress on PNG and JPEG files
- CSR and certificate validation

Future:
- write a complete SSL/TLS stack using Parsifal
- more challenges?
- if you are interested, please tell me, and eventually contribute!

O. Levillain
https://github.com/ANSSI-FR/parsifal
October 23rd 2013
Questions?

Thank you for your attention.

https://github.com/ANSSI-FR/parsifal

olivier.levillain@ssi.gouv.fr