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

Outline

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

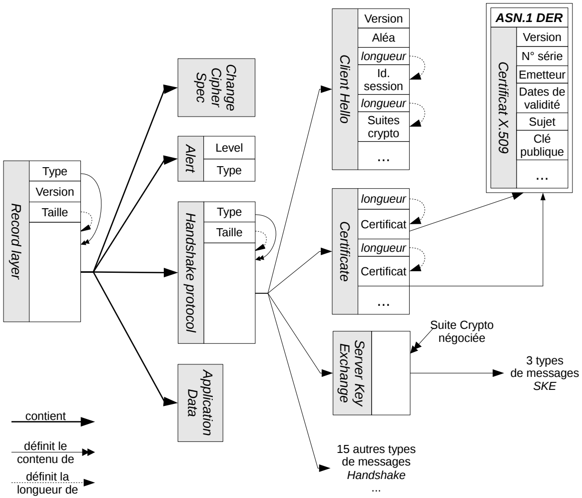
Our goal: analyse the data to understand what our browsers must deal with

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?

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

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)

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

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)

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

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

Downloading and compiling Parsifal

GitHub repository

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

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
```

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

$\mathcal{PT}$ypes

In Parsifal, a $\mathcal{PT}$ype consists of:

- an arbitrary OCaml type `t`
- a parsing function `parse_t`
- a dumping function `dump_t`
- a function producing a higher-level value `value_of_t`

$\mathcal{PT}$ypes

In Parsifal, a $\mathcal{PT}$ype consists of:

- an arbitrary OCaml type `t`
- a parsing function `parse_t`
- a dumping function `dump_t`
- a function producing a higher-level value `value_of_t`

There are three types of $\mathcal{PT}$ypes:

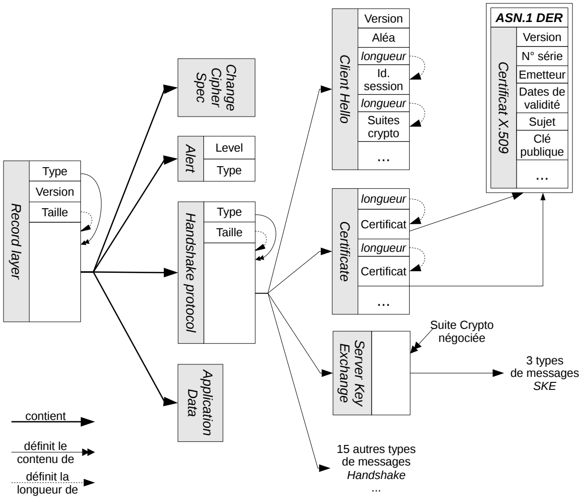
- basic $\mathcal{PT}$ypes (`uint8`, `string`, `list`)
- constructed $\mathcal{PT}$ypes, obtained using keywords and short descriptions
- custom $\mathcal{PT}$ypes

Enumerations

RFC 5246 (TLSv1.2) encodes TLS version field using a 16-bit value.

```
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:

```
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;
```

Structures (2/2)

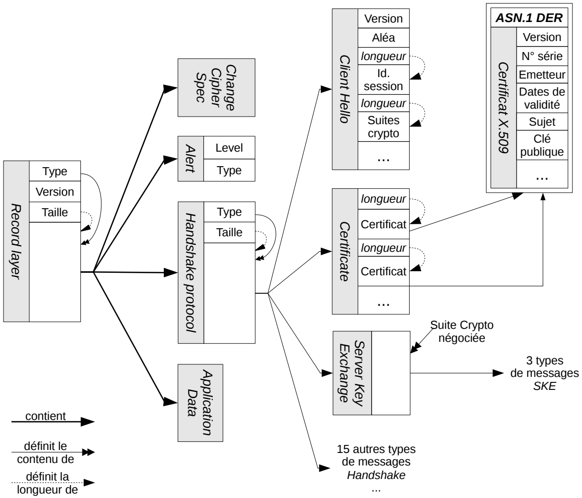
Parsifal implementation:

```
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:

```
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

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 $\mathcal{PT}$ype: `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 $\mathcal{PT}$ype: `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 $\mathcal{PT}$ype: `crc_check`
- First union to enrich chunk content
- ...

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

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):

```
CertificationRequestInfo ::= SEQUENCE {
  version Version,
  subject Name,
  subjectPublicKeyInfo SubjectPublicKeyInfo,
  attributes [0] IMPLICIT Attributes }

Version ::= INTEGER

Attributes ::= SET OF Attribute

CertificationRequest ::= SEQUENCE {
  certificationRequestInfo CertificationRequestInfo,
  signatureAlgorithm SignatureAlgorithmIdentifier,
  signature Signature }

SignatureAlgorithmIdentifier ::= AlgorithmIdentifier

Signature ::= BIT STRING
```

Parsifal implementation

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

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

asn1_struct certificationRequest = {
  certificationRequestInfo : certificationRequestInfo;
  signatureAlgorithm : algorithmIdentifier;
  signatureValue : bitstring_container of signature(signatureTyp
}
```

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

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!

Questions?

Thank you for your attention.

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

olivier.levillain@ssi.gouv.fr