Parsifal: a pragmatic solution to the binary parsing problem

Olivier Levillain

ANSSI

May 18th 2014

Agenda

Motivation: studying SSL/TLS answers

Parsifal

Results

Lessons learned

Agenda

Motivation: studying SSL/TLS answers

Parsifal

Results

Lessons learned

Analysing SSL/TLS data

How to analyse the 180 GB of data collected on port 443?

- complex message format
- presence of corrupted data
- presence of other protocols (HTTP, SSH...)
- more subtle errors may arise

Analysing SSL/TLS data

How to analyse the 180 GB of data collected on port 443?

- complex message format
- presence of corrupted data
- presence of other protocols (HTTP, SSH...)
- more subtle errors may arise

What should you expect from a server when you only propose the `AES128-SHA` and `DHE-RSA-AES128-SHA` ciphersuites?

Analysing SSL/TLS data

How to analyse the 180 GB of data collected on port 443?

- complex message format
- presence of corrupted data
- presence of other protocols (HTTP, SSH...)
- more subtle errors may arise

What should you expect from a server when you only propose the `AES128-SHA` and `DHE-RSA-AES128-SHA` ciphersuites?

A `AES128-SHA`

Analysing SSL/TLS data

How to analyse the 180 GB of data collected on port 443?

- complex message format
- presence of corrupted data
- presence of other protocols (HTTP, SSH...)
- more subtle errors may arise

What should you expect from a server when you only propose the `AES128-SHA` and `DHE-RSA-AES128-SHA` ciphersuites?

A `AES128-SHA`

B `DHE-RSA-AES128-SHA`

Analysing SSL/TLS data

How to analyse the 180 GB of data collected on port 443?

- complex message format
- presence of corrupted data
- presence of other protocols (HTTP, SSH...)
- more subtle errors may arise

What should you expect from a server when you only propose the `AES128-SHA` and `DHE-RSA-AES128-SHA` ciphersuites?

A `AES128-SHA`

B `DHE-RSA-AES128-SHA`

C an alert

Analysing SSL/TLS data

How to analyse the 180 GB of data collected on port 443?

- complex message format
- presence of corrupted data
- presence of other protocols (HTTP, SSH...)
- more subtle errors may arise

What should you expect from a server when you only propose the `AES128-SHA` and `DHE-RSA-AES128-SHA` ciphersuites?

A `AES128-SHA`

B `DHE-RSA-AES128-SHA`

C an alert

D something else (`RC4_MD5`)

Existing tools

To reliably analyse the data, we needed relatively fast and reliable tools

- they should handle gracefully corrupted (or even malicious) input

Standard TLS stacks did not meet our needs, since they can be

- fragile
- incomplete
- silently laxist

Among the existing tools to write parsers, we found nothing suitable:

- Scapy/Hachoir, Python tools
- existing Haskell/OCaml libraries
- binpac, a C preprocessor from the Bro project

Existing tools

To reliably analyse the data, we needed relatively fast and reliable tools

- they should handle gracefully corrupted (or even malicious) input

Standard TLS stacks did not meet our needs, since they can be

- fragile
- incomplete
- silently laxist

Among the existing tools to write parsers, we found nothing suitable:

- Scapy/Hachoir, Python tools
- existing Haskell/OCaml libraries
- binpac, a C preprocessor from the Bro project
- Nail

Homemade tools

To handle the SSL/TLS data, several *parsers* were developed, using different languages

- Python: quick to write, but too slow at runtime
- C++: flexible, fast at runtime, but verbose and hard to debug
- OCaml: robust, efficient, but still too much code
- OCaml with an integrated preprocessor: everything looks fine

Agenda

Marketting

- Parsifal lets you describe constructions
- The corresponding parsing (and dumping) functions are generated
- For example, a simple DNS client can fit in 200 locs

Marketting

- Parsifal lets you describe constructions
- The corresponding parsing (and dumping) functions are generated
- For example, a simple DNS client can fit in 200 locs

- With Parsifal, parsers can be written with **concise** code
- The resulting programs are **fast**
- They are also **robust**
- Parsers can be developed **incrementally**

Marketting

- Parsifal lets you describe constructions
- The corresponding parsing (and dumping) functions are generated
- For example, a simple DNS client can fit in 200 locs

- With Parsifal, parsers can be written with **concise** code
- The resulting programs are **fast**
- They are also **robust**
- Parsers can be developed **incrementally**

- Possible usages of Parsifal
  - robust analysis tools
  - basic blocks for sanitisation tools
  - secure protocol implementations

First example: a trivial PNG parsr

```
struct png_file = {
  png_magic : magic("\x89\x50\x4e\x47\x0d\x0a\x1a\x0a");
  png_content : binstring;
}
```

First example: a trivial PNG parsr

```
struct png_file = {
  png_magic : magic("\x89\x50\x4e\x47\x0d\x0a\x1a\x0a");
  png_content : binstring;
}

let input = input_of_filename "image.png" in
let png = parse_png_file input in
print_value (value_of_png_file png)
```

First example: a trivial PNG parsr

```
struct png_file = {
  png_magic : magic("\x89\x50\x4e\x47\x0d\x0a\x1a\x0a");
  png_content : binstring;
}

let input = input_of_filename "image.png" in
let png = parse_png_file input in
print_value (value_of_png_file png)
```

Program output:

```
value {
  png_magic: 89504e470d0a1a0a (8 bytes)
  png_content: 0000000d49484... (264 bytes)
}
```

Chunk handling (1/2)

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

Chunk handling (1/2)

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

```
struct chunk = {
  chunk_size : uint32;
  chunk_type : string(4);
  data : binstring(chunk_size);
  crc : uint32;
}
```

Chunk handling (2/2)

Sortie du programme:

```
value {
  png_magic: 89504e470d0a1a0a (8 bytes)
  chunks {
    chunks[0] {
      chunk_size: 13 (0x0000000d)
      chunk_type: "IHDR" (4 bytes)
      data: 0000001400000016040300000 (13 bytes)
      crc: 846176565 (0x326fa135)
    }
  ... 4 other chunks ...
  }
}
```

Chunk enriching: IHDR

```
struct chunk = {
  chunk_size : uint32;
  chunk_type : string(4);
  data : container(chunk_size) of chunk_content;
  crc : uint32;
}
```

Chunk enriching: IHDR

```
struct chunk = {
  chunk_size : uint32;
  chunk_type : string(4);
  data : container(chunk_size) of chunk_content;
  crc : uint32;
}

union chunk_content [enrich] (UnparsedChunkContent) =
| "IHDR" → ImageHeader of image_header
```

Chunk enriching: IHDR

```
struct chunk = {
  chunk_size : uint32;
  chunk_type : string(4);
  data : container(chunk_size) of chunk_content;
  crc : uint32;
}

union chunk_content [enrich] (UnparsedChunkContent) =
| "IHDR" → ImageHeader of image_header

struct image_header = {
  width: uint32; height : uint32;
  bit_depth : uint8;
  color_type : color_type;
  ...
}
```

Chunk enriching: IHDR

```
struct chunk = {
  chunk_size : uint32;
  chunk_type : string(4);
  data : container(chunk_size) of chunk_content;
  crc : uint32;
}

union chunk_content [enrich] (UnparsedChunkContent) =
| "IHDR" → ImageHeader of image_header

struct image_header = {
  width: uint32; height : uint32;
  bit_depth : uint8;
  color_type : color_type;
  ...
}

enum color_type (8, UnknownVal UnknownColorType) =
| 0 → Grayscale
| 2 → Truecolor
...
```

Features

Beyond `enum`, `struct` and `union`, Parsifal also has

- `asn1_*` keywords to descrive ASN.1 structures (DER format)
- bit fields
- a notion of containers to automate:
    - compression (`ztext : zlib_container of string;`)
    - encoding (e.g.base64)
    - cryptographic transformations (e.g. `pkcs1_container`)
    - additional constraints
- a toolbox of predefined PTypes

The produced tools are robust against invalid inputs, by construction

- static typing
- strict interpretation

Features

Beyond `enum`, `struct` and `union`, Parsifal also has

- `asn1_*` keywords to descrive ASN.1 structures (DER format)
- bit fields
- a notion of containers to automate:
  - compression (`ztext : zlib_container of string;`)
  - encoding (e.g.base64)
  - cryptographic transformations (e.g. `pkcs1_container`)
  - additional constraints
- a toolbox of predefined PTypes

The produced tools are robust against invalid inputs, by construction

- static typing
- strict interpretation

Once we had this hammer, every binary format *really* looked like a nail

Features

Beyond `enum`, `struct` and `union`, Parsifal also has

- `asn1_*` keywords to descrive ASN.1 structures (DER format)
- bit fields
- a notion of containers to automate:
  - compression (`ztext : zlib_container of string;`)
  - encoding (e.g.base64)
  - cryptographic transformations (e.g. `pkcs1_container`)
  - additional constraints
- a toolbox of predefined PTypes

The produced tools are robust against invalid inputs, by construction

- static typing
- strict interpretation

Once we had this hammer, every binary format *really* looked like a nail

But Parsifal always allows to mix manually written types

Agenda

Some figures

Three home-made TLS analysers (certificate extraction)

	C++	OCaml	Parsifal
LOC	8,500	4,000	1,000
Processing time	100 s	40 s	8 s

Three tools to analyse BGP messages:

	`libbgpdump`	OCaml	Parsifal
LOC	4,000	1,200	550
Processing time	23 s	180 s	35 s
Robustness	NO	yes	yes

Other formats

Here are a list of formats (at least partially) implemented

- DNS
- NTP
- PNG
- OpenPGP
- Kerberos
- PE
- UEFI Firmware
- DVI

Agenda

On formats

There is something as a bad format:

- PE and EXIF include non-linear structures
- DVI force you to know the whole spec to parse a file
- integers may come in very different flavours

On formats

There is something as a bad format:

- PE and EXIF include non-linear structures
- DVI force you to know the whole spec to parse a file
- integers may come in very different flavours
  - at least 4 in ASN.1 DER
  - do you know the TAR way to represent them?

On formats

There is something as a bad format:

- PE and EXIF include non-linear structures
- DVI force you to know the whole spec to parse a file
- integers may come in very different flavours
  - at least 4 in ASN.1 DER
  - do you know the TAR way to represent them?

On the contrary, we like

- Tag-Length-Value which allows extensibility
- canonical representations
- reusable elements
- simple, linear parsing

On the language

- OCaml proved to be a robust language
- The presence of a GC is often seen as a major advantage
- (unless you *want* to handle some memory cells)
- For me, the real pro is the exhaustive pattern matching
- Also, strong typing keep you on track

On the process

- Implementing parsers gives you real insight in formats and protocols
- Parsifal automates most of the mind-numbing repetetive tasks
- Parsifal allows for incremental development

On the process

- Implementing parsers gives you real insight in formats and protocols
- Parsifal automates most of the mind-numbing repetetive tasks
- Parsifal allows for incremental development

- Yet our methodology aims at *checking the validity* of values with robust tools, not so much at fuzzing

Conclusion

- Three years of writing parsers led us to Parsifal
- Our hammer looks more and more like a Swiss knife

- Until now, we mainly used it to understand formats and analyse data
- Sanitization tools have been prototyped (certificates, PNG)
- Next step: more real-world use cases

- Since June 2013, the code is available on GitHub

Questions?

Thank you for your attention

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

olivier.levillain@ssi.gouv.fr