Parsifal: a tutorial

Olivier Levillain

Parsifal

This tutorial is about Parsifal: a generic framework to write binary parsers in OCaml.

After a short introduction, the slides explain how to install and build Parsifal.

The remaining of the presentation are step-by-step implementation of some toy parsers (TAR, DNS).

Outline

Outline

Starting point of our work: SSL data (1/2)

Several campaigns to collect SSL data, between July 2010 and July 2011, using the following methodology:

- enumerating IPv4 hosts with 443/tcp open
- sending `ClientHello` messages
- recording the server answer

Starting point of our work: SSL data (1/2)

Several campaigns to collect SSL data, between July 2010 and July 2011, using the following methodology:

- enumerating IPv4 hosts with 443/tcp open
- sending `ClientHello` messages
- recording the server answer

Using 10 such campaigns, we analysed several parameters:

- TLS parameters
- ceritification chain quality
- server behaviour against different stimuli
- results published at ACSAC 2012

Starting point of our work: SSL data (2/2)

Our goal was to extract from those 140 GB of data

- relevant infomration (the messages and certificates received)
- quickly
- in a robust way

Starting point of our work: SSL data (2/2)

Our goal was to extract from those 140 GB of data

- relevant infomration (the messages and certificates received)
- quickly
- in a robust way

The Electronic Frontier Foundation, which published part of the data we used, also provided some analyses

- they mostly used standard tools (`openssl`)
- they mostly focus on certificates

Some history of our parsers (1/2)

To handle this amount of data, we used custom tools; several implementaions were developped

Some history of our parsers (1/2)

To handle this amount of data, we used custom tools; several implementaions were developped

- a first Python prototype: quick to write, slow to run

Some history of our parsers (1/2)

To handle this amount of data, we used custom tools; several implementaions were developped

- a first Python prototype: quick to write, slow to run
- a second version in C++ (using templates and objects):
  - rather extensible (thanks to a description language)
  - faster than Python, but
  - hard to debug (memory leaks, segfaults)
  - long to write (each new feature required too much code)

Some history of our parsers (1/2)

To handle this amount of data, we used custom tools; several implementaions were developped

- a first Python prototype: quick to write, slow to run
- a second version in C++ (using templates and objects):
  - rather extensible (thanks to a description language)
  - faster than Python, but
  - hard to debug (memory leaks, segfaults)
  - long to write (each new feature required too much code)
- third version in OCaml (using a Domain Specific Language resembling Python):
  - fast and extensible
  - more robust than the previous one
  - but still too much code to write

Some history of our parsers (2/2)

- the last version, still in OCaml, is called Parsifal

Some history of our parsers (2/2)

- the last version, still in OCaml, is called Parsifal
- it relies on a pre-processor to automate most of the tedious steps
  - the resulting code is fast, robust and concise

Code is available on GitHub (`ANSSI-FR/parsifal`)

Parsifal in a nutshell

- Generic framework to write **concise** parsers
- **Speed** of the produced programs
- **Robustness** of the developped tools
- Development methodology adapted to write parsers **incrementally**

Parsifal in a nutshell

- Generic framework to write **concise** parsers
- **Speed** of the produced programs
- **Robustness** of the developped tools
- Development methodology adapted to write parsers **incrementally**

- Parsifal also allows to dump the described objects
- Example: a DNS client in 200 loc

Parsifal in a nutshell

- Generic framework to write **concise** parsers
- **Speed** of the produced programs
- **Robustness** of the developped tools
- Development methodology adapted to write parsers **incrementally**

- Parsifal also allows to dump the described objects
- Example: a DNS client in 200 loc

- Parsifal main goals
  - trusted analysis tools (SSL, X.509, Kerberos, OpenPGP...)
  - basic blocks to sanitize files or protocol messages (PNG, PKCS#10 CSR...)

Outline

OCaml installation

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

apt-get install git
```

(Tested on Debian Wheezy)

Parsal compilation and installation

Cloning git repository

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

Compilation

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

(Without `LIBDIR` or `BINDIR`, system dirs are used)

How to create a project

First project

```
./mk_project.sh helloworld
cd helloworld
make
./helloworld
```

The project created simply display `Hello, world!`, but uses a Makefile using `parsifal_syntax`, the preprocessor.

Let us now discover the constructions provided by Parsifal.

Outline

Principle

The idea is to let the developper write short type descrptions, and to expand them automatically to obtain the following elements:

- an OCaml type `t`
- a parsing function `parse_t`
- a dumping function `dump_t`
- a function to convert `t` to a printable value `value_of_t`

Function prototypes

```
parse_t : string_input -> t
dump_t : POutput.t -> t -> unit
value_of_t : t -> value
```

PTypes

Such an enriched type (an OCaml type accompagnied by the thhree functions.

There are three kinds of PTypes:

- basic PTypes (integers, strings, lists, etc.) are provided by the Parsifal core library
- keyword-assisted PTypes, described in the following slides
- custom PTypes can be written manually

Enumerations

Goal: use a sum type resembling C enums with strong types

TLS versions are encoded by a 2-byte (16-bit) value:

```
enum tls_version (16, UnknownVal V_Unknown) =
  | 0x0002 -> V_SSLv2, "SSLv2"
  | 0x0300 -> V_SSLv3, "SSLv3"
  | 0x0301 -> V_TLSv1, "TLSv1.0"
  | 0x0302 -> V_TLSv1_1, "TLSv1.1"
  | 0x0303 -> V_TLSv1_2, "TLSv1.2"
```

Enums also come with useful extra functions:

- `int_of_tls_version`
- `string_of_tls_version`
- `tls_version_of_int`
- `tls_version_of_string`

Structures (1/2)

Goal: handle a sequence of fields

For example, TLS alerts simply are a structure containing two 1-byte fields, described as follows in RFC 5246 (TLSv1.2):

```
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)

TLS alerts in Parsifal:

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

enum tls_alert_type (8, UnknownVal AT_Unknown) =
  | 0 -> AT_CloseNotify
  | 10 -> AT_UnexpectedMessage
    ...
  | 110 -> AT_UnsupportedExtension

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

Unions

Goal: create a type depending on a discriminating value

```
union autonomous_system [enrich] (UnparsedAS) =
  | 16 -> AS16 of uint16
  | 32 -> AS32 of uint32

struct bgp_as_path_segment [param as_size] =
{
  path_segment_type : uint8;
  path_segment_length : uint8;
  path_segment_value : list(path_segment_length) of
                       autonomous_system(as_size)
}
```

Alias

Goal: aliases allow for renaming PTypes or ASN.1 structures

```
alias ustar_magic = magic["ustar"]
alias tar_file = list of tar_entry

struct atv_content = {
  attributeType : der_oid;
  attributeValue : der_object
}
asn1_alias atv
asn1_alias rdn = set_of atv
asn1_alias distinguishedName = seq_of rdn
```

Other constructions

- PContainers, allowing transparent transformations at parsing and/or dumping time
  - 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, length-checking
- `asn1_union` and `asn1_struct`
- bit fields

Outline

The TAR format (1/3)

A TAR file is composed of entries:

Offset	Len	Description
0	512	TAR header, padded with zero bytes
512	file size aligned at the 512-byte boundary)	file content, padded with zero bytes

The TAR format (2/3)

TAR header:

Offset	Len	Description	
		TAR	`ustar`
0	100	Filename	
100	8	Permissions	
108	8	UID	
116	8	GID	
124	12	File size	
136	12	Timestamp	
148	8	Header checksum	
156	1	Link type	File type
157	100	Linked file	
257	5	-	"`ustar`" marker
263	3	-	`ustar` version
265	32	-	Owner
297	32	-	Group
329	8	-	Device major
337	8	-	Device minor
345	155	-	Prefix

The TAR format (3/3)

Link type/File type values:

Character	**Description**	`ustar`**-specific**
`<NUL>, 0`	regular file	-
1	hard link	-
2	symbolic link	-
3	character device	yes
4	block device	yes
5	directory	yes
6	FIFO	yes
7	contiguous file	yes

TAR v1

`tar-steps/tar1.ml` describe a primitive version of the TAR file format:

- a `file_type` `enum` to represent the different values
- a `struct` type to describe the complete `ustar` header
- in TAR, integer as encoded as a string representing an octal value; to decode the `file_size` field, the file contains a `int_of_tarstring` function
- then, a second `struct` to describe a TAR entry
- finally, the `main` program opens a file and prints the names of the files contained in the archive

TAR v2

`tar-steps/tar2.ml` allows the `ustar` header to be optional:

- the `tar_header` is now terminated by a `string` field (which corresponds to the remaining string) and encapsulated inside a 512-byte container
- the `ustar`-specific part of the header is extracted in a new `struct`
- the header includes the `ustar_header` field as an optional field

TAR v3

As explained earlier, integers are encoded as strings representing their octal value in TAR archive.

To handle integers better, we write a custom `tar_numstring` PType in `tar-steps/tar3.ml`:

- the `tar_numstring` type is `int`, the *intended* value
- to create a working PType, we need to write the `parse_tar_numstring`, `dump_tar_numstring` and the `value_of_numstring` functions
- we replace the old `string` fields by `tar_numstring` ones

As our new PType needs the length argument at parsing and dumping time, the argument is specified using [] instead of ()

TAR v4/5

However tar3.ml is bugged since the device_major and device_minor are filled with zero bytes when the file type is not a device.

Thus, parse_tar_numstring fails and the whole ustar header is ignored.

There are two possible fixes:

- tar4.ml defines a new custom PType optional_tar_numstring
- tar5.ml creates a union using file_type as the discriminating value

Improvements

`tar6.ml`, `tar7.ml` and `tar8.ml` later improve the parser:

- better display of strings, by taking into account the trailing zeroes (`tar6.ml`)
- add a `try..with` in the `main` function to handle the end of file (`tar7.ml`)
- add a list and a `checkpoint` to handle the end of file (alternative solution, `tar8.ml`)

Outline

DNS messages (1/4)

The original specification is RFC 1035 (updated by many RFCs since)

Message layout:

Offset	**Len.**	**Description**
0	2	QId
2	2	*flags*
4	2	Question count
6	2	Answer count
8	2	Authority record count
10	2	Additional record count
12	?	Questions
?	?	Answers
?	?	Authority records
?	?	Additional records

DNS messages (2/4)

Question format:

- a domain,
- a 16-bit `query_type`
- a 16-bit `query_class`

A domain is a sequence of labels, a label being:

- a string if the two first bits are zeroes, the six next bits containing the string length
- an empty label (a zero byte) to signal the end of a domain
- a pointer to compress the domain, two bytes beginning by `0b11` and followed by a 14-bit offset to retrieve the end of the domain in the already parsed message

DNS messages (3/4)

Resource Record (RR) format:

- a domain (cf. previous slide)
- a 16-bit `rr_type`
- a 16-bit `rr_class`
- a 32-bit TTL (time-to-live)
- a 16-bit integer representing the size of the data
- the RR data

Here are examples of RR data:

- `A` RR contain an IPv4 address (32 bits)
- `CNAME` RR is an alias pointing towards a domain
- `MX` RR establishes mail exchanger information (a 16-bit integer and a domain)

DNS messages (4/4)

Here a some possible values of `query_type` / `rr_type`:

Value	Description	Compatible with `rr_type`
1	`A`	yes
2	`NS`	yes
5	`CNAME`	yes
6	`SOA`	yes
12	`PTR`	yes
15	`MX`	yes
255	`*`	-

And some values for `query_class` / `rr_class`:

Value	Description	Compatible with `rr_class`
1	`Internet`	yes
2	`CSNET`	yes
3	`CHAOS`	yes
4	`Hesiod`	yes
255	`*`	-

DNS: first implementation

`dns-steps/dns1.ml` is a first description of DNS messages:

- two enums for `rr_type` and `rr_class`
- a custom `label` PType
- a custom `domain` PType to implement a list of `labels`
- two `structs` describing questions and RRs
- a `dns_message` structure to wrap everything up
- finally, some piece of code to use the generated `parse_dns_message` funcion$\frac{1}{2}$

DNS: enriched RRs

`dns-steps/dns2.ml` enriches the `rdata` field:

- new PType: a `union` called `rdata` where the discriminator is the RR type
- for some RRs, description of the RR data (A, CNAME, MX)
- change of the `rdata` field type from `binstring` to `rdata(rtype)` (the union just defined)

Toward a trivial DNS client

- `dns3.ml` consists of a rewrite of `domain` and `label`
- `dns4.ml` introduces a parameter to the `domain` PType, `context`, to record and expand the label compression at parsing time
- `dns5.ml`, `dns6.ml` and `dns7.ml` add some code in the `main` function to send a request to a real server, and to print the result

Outline

Related work

- Scapy
- Hachoir
- OCaml `bitstring` library
- NetZob
- Bro's binpac language

File formats and network protocols implemented

- X.509
- TLS
- MRT+BGP
- TAR
- PCAP/IP/TCP/UDP (trivial description)
- DNS
- PNG
- PE (work in progress)
- ExpROM (work in progress)
- Kerberos (work in progress)

Questions ?

Thank you for your attention.