Introduction to **CTF 2**

Common **T**race **F**ormat

Philippe Proulx
pproulx@efficios.com
eepp

EfficiOS

Contents

What is CTF?

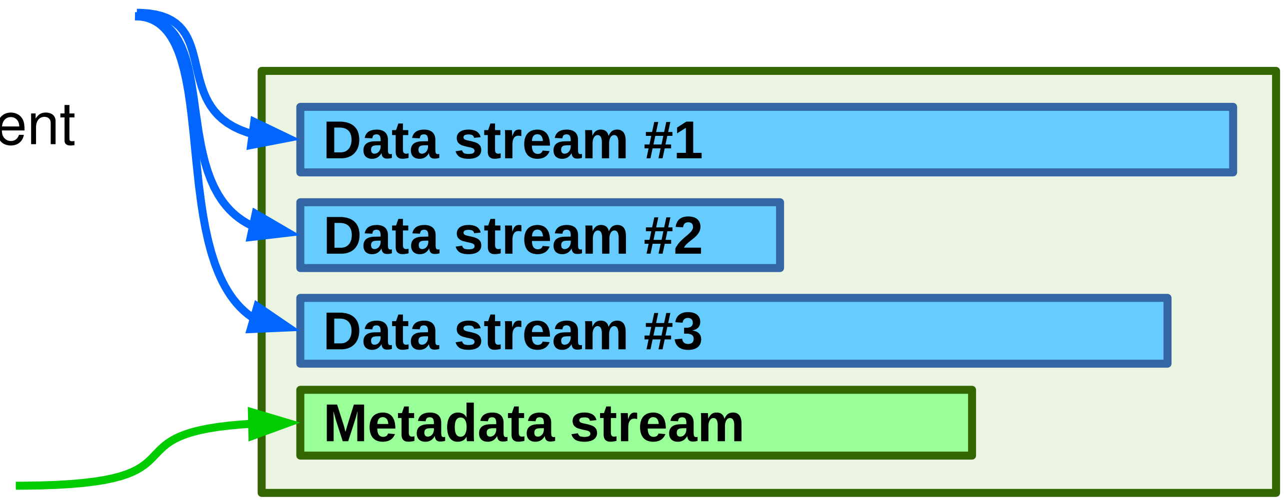
- "**C**ommon **T**race **F**ormat"
- Self-described binary trace format
- CTF 1 specified in 2010-2011
- Focused on producer's performance
  - Supports big-endian and little-endian fields
  - Supports bit fields
  - Supports custom field alignments
  - Supports multiple data streams
    - Data streams of packets of event records

What is CTF?

Anatomy of a CTF trace:

One or more **data streams**:

- Binary data from tracer
- Contains packets of event records



One **metadata stream**:

- TSDL (CTF 1) or JSON (CTF 2)
- Describes the data streams

What is CTF?

Example:

CTF 1 metadata stream

```
// ...
event.header := struct {
  uint64le timestamp;
  uint16be id;
};
// ...
event {
  name = new_msg;
  id = 23;
  fields := struct {
    uint32le msg_id;
    string msg;
  } align(32);
};
// ...
```

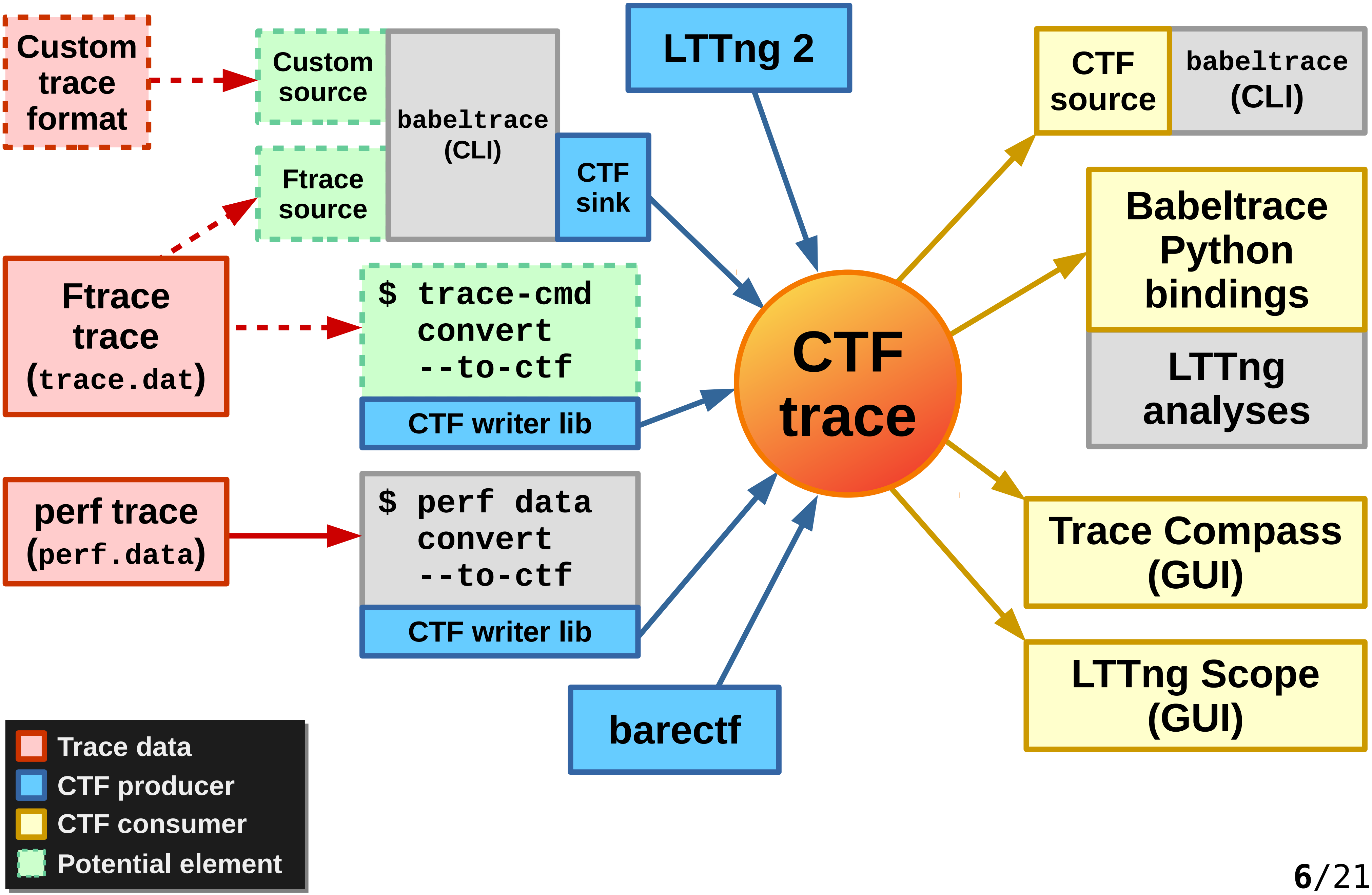
CTF data stream

```
...7d ee 9c b8 8b 99 d1
89 dd ed 84 c3 02 00 00
00 17 00 00 2d ff 00 00
48 65 6c 6c 6f 2c 20 57
6f 72 6c 64 21 00 2d ff
40 52 d9 8d ff 90 ff...
```

Encoded event record:

- *Name*: "new_msg"
- *timestamp*: **15h47:11.2839912**
- *msg_id*: **65325** (**0xff2d**)
- *msg*: "**Hello, World!**"

CTF ecosystem:



Limitations of CTF 1

Metadata language (TSDL) is hard to consume

- Complex grammar (subset of C w/ additions)
- Many implicit parsing rules, e.g.:
  - “Magic” field names, e.g. `uuid`, `id`, `timestamp`
- Lexically scoped type aliases
- Useful when you write the metadata stream manually, but who does that?

Limitations of CTF 1

Metadata language (TSDL) is hard to consume

Parsing this valid TSDL is left as an exercise to the reader:

```
struct {
  typealias integer {size = 33;} :=
    some_int;
  enum : integer {
    size = 17;
    align = 0b100;
    byte_order = be; // big endian
    base = x;        /* "hex" */
    signed = true;
  } {
    INIT = 0x23d,
    "/* best */ state" = -50 ... 21,
  } state[17]
        [stream.packet.context.a.b.c];
  variant var <previous.selection> {
    some_int CHOICE0;
    struct {string z;}
      align(32) SOME_ENTRY[2];
  };
} align(64);
```

EfficiOS

Limitations of CTF 1

Metadata language (TSDL) is hard to consume

Solution:

- Use JSON
- Require explicit references and descriptions so as to simplify the consumers
- Have only one level of type aliases
- Keep semantic compatibility with TSDL

Limitations of CTF 1

Metadata language (TSDL) is hard to consume

```
event {
  id = 23;
  name = "my_event";
  loglevel = 4;
  fields := struct {
    my_int intField;
    string stringField;
  } align(64);
};
```

```
{
  "fragment": "event-record-class",
  "user-attrs": {
    "diamon.org/ctf/ns/basic": {
      "name": "my_event",
      "log-level": 4
    }
  },
  "id": 23,
  "payload-field-type": {
    "field-type": "struct",
    "alignment": 64,
    "fields": [
      {
        "name": "intField",
        "field-type": "my_int"
      },
      {
        "name": "stringField",
        "field-type": {
          "field-type": "string"
        }
      }
    ]
  }
}
```

Limitations of CTF 1

Metadata language (TSDL) is hard to extend

- Strict grammar
- No extension points specified
- For example, metadata cannot express:
  - Format strings for types
  - Tag a specific field as an instruction pointer
  - Tag a specific field as a stack trace
- We have to rely on field names: this is precarious

Metadata language (TSDL) is hard to extend

```
event.context := struct {
    uint32 ip; /* instruction pointer */
    ...;
};
```

"Magic" ***ip*** field in event record's context represents an instruction pointer in LTTng CTF traces.

It could mean "IPv4 address" for another tracer, for example.

Limitations of CTF 1

Metadata language (TSDL) is hard to extend

Solution:

- Have a `user-attrs` property in selected metadata objects
  - Field types, event classes, stream classes, trace, etc.
- User attributes are part of a specific namespace (tracer, vendor, specification, etc.) to avoid conflicts

Metadata language (TSDL) is hard to extend

```
{
    ...,
    "event-record-context-field-type": {
        "field-type": "struct",
        "fields": [
            {
                "name": "func_addr",
                "field-type": "uint64",
                "user-attrs": {
                    "lttng.org/ns/ctf": { "is-ip": true }
                }
            },
            ...
        ]
    }
}
```

Not named *ip*

Namespace

Limitations of CTF 1

CTF 1 is missing useful field types

CTF 1 integer field types are always encoded on a fixed number of bits, but in some scenarios, the values are often small.

Solution:

- Add **variable-length integer** field type
- Add **variable-length enumeration** field type
- Variable-length field types use the popular LEB128 encoding (DWARF, protobuf, Android’s DEX)

Limitations of CTF 1

CTF 1 is missing useful field types

CTF 1 has no way to express boolean fields; we currently use integer fields for this.

Boolean and integer programming language types have different semantics.

Solution:

- Add fixed-size **boolean** field type
- All bits cleared means *false*, anything else means *true*

Limitations of CTF 1

CTF 1 is missing useful field types

CTF 1 has no way to express null fields; we currently use empty structure fields for this.

Solution:

- Add 0-bit **null** field type
- Used to represent *nothing* as a variant field type's choice
- Used to align the consumer without consuming actual payload bits

Limitations of CTF 1

CTF 1 is missing useful field types

CTF 1 has no way to indicate that a given binary payload can be decoded in more than one way.

Solution:

- Add **union** field type
- Decoding position *must* be the same after decoding, whichever field type the consumer chooses to use
- Used to indicate alternative “views” of binary data (like in C)
- Used to introduce new field types in future CTF 2 revisions
- *Examples*:
  - 32-bit bit-endian integer *vs.* 4-byte array for IPv4 address
  - Sequence of bytes (known as of 2.0) *vs.* UTF-16 string (possible future field type, unknown as of 2.0)

CTF 2: planned adoption

- **Babeltrace (consumer and producer)**: v2.1
- **LTTng**: ~v2.11/v2.12 if the discussion is active enough.
  - *Condition*: Babeltrace v2.1 *must* be released/packaged.
  - *Idea*: Implement a temporary hybrid mode where you can choose to generate either a CTF 1 or a CTF 2 trace. No interest so far.
- **barectf**: As soon as Babeltrace v2.1 is released.
- **LTTng Scope**: Synchronized with LTTng producing CTF 2 traces.

Resources

- **CTF website:** http://diamon.org/ctf/
- **CTF 2 proposal**:
  - https://lists.linuxfoundation.org/pipermail/diamon-discuss/2016-October/000099.html
  - **HTML version**:
    - http://diamon.org/ctf/files/CTF2-PROP-1.0.html
- **Other documents**:
  - http://diamon.org/ctf/files/CTF2-BASICATTRS-1.0.html
  - http://diamon.org/ctf/files/CTF2-DOCID-1.0.html
  - http://diamon.org/ctf/files/CTF2-FS-1.0.html
  - http://diamon.org/ctf/files/CTF2-PMETA-1.0.html

Q&A