From: **Joep Gommers** joep@intelworks.com
Subject: Intelworks implementation of STIX in JSON
Date: June 16, 2015 at 08:09
To: cti@lists.oasis-open.org

JG

Dear all,

With the excellent work going on from @Bret Jordan on STIX in JSON, we thought it helpful to share Intelworks approach to STIX in JSON and ensure the community learned from our mistakes and investments. Props to list- and team member @Wouter Bolsterlee for his work on this!

In short, lessons learned

- Compound structures are objects
- Attributes and child elements are key/value pairs
- Relations are nested objects (or arrays of objects)
- Flat is better than nested
- And some ID and corner cases see below

Full details further down in this email. Your feedback is much appreciated.

We do have work-in-progress libraries available for (store-less) bi-directional transformation of XML, JSON and YAML notations which might help those implementing STIX in JSON down the road. If you'd like to know more, please contact me off-list.

Best regards,
Joep

Founder & CEO
Intelworks Intelligence Powered Defence
www.intelworks.com

Find me at
+31 615489825
@joepgommers

====

The STIX language uses quite a few advanced XML modelling techniques (multiple namespaces, xsi:type substitutions in instance documents, QName identifiers, and so on), making it quite complex to work with/implement. The JSON format used by Intelworks tries to be much simpler to work with. Structurally it mirrors most of the original XML tree structure, but the resulting tree structures are not identical since the JSON representation favours flat objects over nested structures.

Compound structures are objects

In general, each compound structure is converted into a JSON object (dict in Python). These objects always have a `type` key to indicate the type of the structure:

```
{
  "type": "indicator",
  "...": "..."
}
```

Each of the main STIX constructs (see the STIX architecture) is represented as a JSON object. The `type` keys used are:

Defining schema	XML Schema type	Object `type` field
STIX (Core)	`STIXType`	`package`
STIX (Campaign)	`CampaignType`	`campaign`
STIX (Course of Action)	`CourseOfActionType`	`course-of-action`
STIX (Exploit Target)	`ExploitTargetType`	`exploit-target`
STIX (Incident)	`IncidentType`	`incident`
STIX (Indicator)	`IndicatorType`	`indicator`
STIX (TTP)	`TTPType`	`ttp`
STIX (Threat Actor)	`ThreatActorType`	`threat-actor`
CybOX	`ObservableType`	`observable`

Secondary constructs use these additional types (this list is NON EXHAUSTIVE! And just a representation of potential)

Defining schema	XML Schema type	Object `type` field
STIX (Common)	`IdentityType`	`identity`
STIX (Common)	`InformationSourceType`	`information-source`
STIX (Common)	`StatementType`	`statement`
STIX (Course of Action)	`ObjectiveType`	`objective`
STIX (Indicator)	`ValidTimeType`	`valid-time`
STIX (Markings)	`MarkingSpecificationType`	`marking-specification`
STIX (Markings)	`MarkingStructureType` (and extensions)	`marking-structure`
STIX (TTP)	`InfrastructureType`	`infrastructure`
STIX (TTP)	`MalwareInstanceType`	`malware-instance`
STIX (TTP)	`ResourceType`	`resource`
STIX (TTP)	`ToolInformationType`	`tool-information`
STIX (TTP)	`VictimTargetingType`	`victim-targeting`
CybOX	`MeasureSourceType`	`measure-source`
CybOX	`ObjectType`	`cybox-object`
CybOX	`ToolInformationType`	`tool-information`

Attributes and child elements are key/value pairs

Both the attributes and child elements defined for a compound structure usually map to additional key/value pairs of the JSON objects:

```
{
  "type": "indicator",
  "negate": false,
  "title": "This is the title."
}
```

Relations are nested objects (or arrays of objects)

For one-to-one relations, the value is a nested object, and the key is a singular noun (`observable` in the example):

```
{
  "type": "indicator",
  "observable": {
    "type": "observable",
    "...": "..."
  },
  "...": "..."
}
```

For one-to-many relations, the value is a JSON array containing the child objects, and the key is a plural noun (`indicators` in the example):

```
{
  "type": "package",
  "indicators": [
    {
      "type": "indicator",
      "...": "..."
    },
    {
```

```
      "type": "indicator",
      "...": "..."
    }
  ],
  "...": "..."
}
```

Additionally, the many `RelatedXYZ` constructs (and the surrounding container objects) in STIX are also flattened: the target of the relation is the child object (or a list of those), and any additional relationship information is embedded into the child object(s):

```
{
  "type": "indicator",
  "indicated_ttps": [
    {
      "type": "ttp",
      "relationship": "...",
      "relationship_information_source": "...",
      "...": "..."
    },
    {
      "type": "ttp",
      "relationship": "...",
      "relationship_information_source": "...",
      "...": "..."
    }
  ],
  "...": "..."
}
```

See also the notes about nesting below.

Flat is better than nested

The STIX XML representation is deeply nested, partly due to the way XML is typically used. The JSON representation tries to be a bit more pragmatic and adheres to the "flat is better than nested" adage.

In practice, this means that nested container structures are flattened as much as possible. Unnecessary container structures are simply removed. For example, the `<stix:Indicators>` container structure used in the XML representation does not exist as such in the JSON representation, since using an array is sufficient.

To further reduce the number of nested objects, various XML constructs using container elements with (optional) attributes are flattened into the parent object by using multiple related keys. This is best explained using an example.

For example, the `StructuredTextType` used in both STIX and CybOX is basically a string that can optionally carry a `structuring_format` attribute. A naive conversion would require a nested object to represent this:

```
{
  "type": "...",
  "description": {
    "structuring_format": "html",
    "value": "Description goes here."
  },
  "...": "..."
}
```

Since the `structuring_format` is optional, this approach would often result in a small nested object with only a single key/value pair (the `value`). To avoid this, *objectivistix* takes an alternative approach using two related keys in the containing object:

```
{
  "type": "...",
  "description": "Description goes here.",
  "description_structuring_format": "html",
  "...": "..."
```

```
}
```

In case the `structuring_format` is not specified, the `description_structuring_format` key/value pair would simply not be present:

```
{
  "type": "...",
  "description": "Description goes here.",
  "...": "..."
}
```

ID handling

All `id` and `idref` attributes in STIX XML are not simply string values, but qualified names (QName in XML), meaning that they contain a namespace prefix which resolves to a namespace URI. To avoid any explicit mappings for these prefixes and their associated namespace URI, the JSON representation always expresses `id` and `idref` values in their canonical form using the so-called Clark notation, which looks like this: `{http://example.com/ns/uri}local-name`.

The top level object may optionally contain an `id_namespaces` mapping that maps prefixes to namespace URIs. This mapping will be used to determine the prefixes used for `id` and `idref` attribute values when converting the object to XML, as illustrated by the example below:

```
{
  "type": "package",
  "id": "{http://example.org/}Package-b3ba766b-d3e6-4d92-82b2-5940f0cb763c",
  "id_namespaces": {
    "example": "http://example.org/"
  }
}
```

```
<stix:STIX_Package
  xmlns:stix="http://stix.mitre.org/stix-1"
  xmlns:example="http://example.com/"
  id="example:Package-b3ba766b-d3e6-4d92-82b2-5940f0cb763c">
  Š
</stix:STIX_Package>
```

In case no `id_namespaces` mapping is present, a unique namespace prefix will be used instead. The `id_namespaces` can safely be left out with no semantical loss, since the prefix is arbitrary and only used for serialized XML data, and not for the in-memory model.

Special conversion notes

- STIX package header

  The package header is not treated as a first-class structure. Since the `STIX_Header` construct only applies to`STIX_Package`, it is merged completely into the main `package` object (this avoids having an additional nested object for the header):

  ```
  {
    "type": "package",
    "description": "Description goes here.",
    "...": "..."
  }
  ```

- Structured text

  The `StructuredTextType` construct is not transformed into a child object. Instead, the keys `foo` and (optionally)`foo_structuring_format` are added to the containing object.

- Observable composition

  An Œobservable composition[1] structure does not result in a nested object for the composition itself. Instead, the`composition` key contains the child objects, and the `composition_operator` specifies the operator:

```
{
  "type": "indicator",
  "observable": {
    "composition_operator": "or",
    "composition": [
      {
        "type": "observable",
        "...": "..."
      },
      {
        "type": "observable",
        "...": "..."
      }
    ]
  },
  "...": "..."
}
```