stixmarx Documentation

Release 1.0.7

The MITRE Corporation

Apr 28, 2020

Contents

See the *Examples* page for sample usage.

Contents:

CHAPTER 1

Overview

`stixmarx` is a Python library that allows the application of STIX data markings to python-stix, python-cybox and python-maec API objects. It is intended for use by Python software developers who want to incorporate field-level data marking capabilities into their STIX-processing software. For more about STIX data markings, see the documentation on data markings from the STIX project.

The library serves several broad functions:

- Creating and managing markings on a STIX package
- Parsing markings that are present in input XML

For a full set of examples with explanations, see the *Examples* page. For the full `MarkingContainer` API, see the *API Reference*.

Be sure also to read the Usage Notes, below, for information on a few important non-obvious behaviors and usage requirements.

1.1 Applying Markings

The STIX specification allows data markings to be applied to any combination of attributes and elements that can be described by XPath. Now, the `stixmarx` API can ease the process of apply markings by automating the process of generating XPath expressions during serialization. It provides support for global markings that is, markings the cover the entire STIX Package. Also, major STIX components (like Indicators, Incidents, etc.) and to CybOX Observables embedded in a STIX document. Finally, to specific attributes or text within the document.

Using the API to mark an element with the descendants option set to `True` also causes the marking to apply to all descendant elements and attributes under that marked element. For example, consider python-stix code to create an Incident with a related Indicator embedded inside of it:

```
container = stixmarx.new()
package = container.package

incident = Incident(title="An Incident")
package.add_incident(incident)
```

```
indicator = Indicator(idref="example:indicator-4273c681-487f-4e3b-0ef2-f36ebd0a6295")
incident.related_indicators.append(indicator)
```

This corresponds to the XML result:

```
<stix:Incident>
    <incident:Title>An Incident</incident:Title>
    <incident:Related_Indicators>
        <incident:Related_Indicator>
            <stixCommon:Indicator idref="example:indicator-4273c681-487f-4e3b-0ef2-
↪f36ebd0a6295"/>
        </incident:Related_Indicator>
    </incident:Related_Indicators>
</stix:Incident>
```

When using the `stixmarx` API to mark the outer Incident, the marking will also apply to all content inside the Incident, including the embedded `<Title>`, `<Related_Indicators>`, and `<Indicator>` elements.

```
marking_struct = TLPMarkingStructure(color='GREEN')
marking_spec = MarkingSpecification()
marking_spec.marking_structures.append(marking_struct)

container.add_marking(incident, marking_spec, descendants=True)
```

This corresponds to the XML Result:

```
<stix:Incident>
    <incident:Title>An Incident</incident:Title>
    <incident:Related_Indicators>
        <incident:Related_Indicator>
            <stixCommon:Indicator idref="example:indicator-4273c681-487f-4e3b-0ef2-
↪f36ebd0a6295"/>
        </incident:Related_Indicator>
    </incident:Related_Indicators>
    <incident:Handling>
        <marking:Marking>
            <marking:Controlled_Structure>../../../descendant-or-self::node() | ../../
↪../descendant-or-self::node()/@*</marking:Controlled_Structure>
            <marking:Marking_Structure xsi:type='tlpMarking:TLPMarkingStructureType'␣
↪color="GREEN"/>
        </marking:Marking>
    </incident:Handling>
</stix:Incident>
```

The central class for adding markings to a document is `MarkingContainer`, which contains a `python-stix` STIXPackage object as its `package` property. The `MarkingContainer` class provides the `add_marking`, `get_markings`, and `remove_marking` functions for adding, retrieving, and removing markings on STIX, CyboX or MAEC components. It also supports the `add_global` and `remove_global` functions for adding and removing markings that apply to an entire document.

1.2 Parsing Markings

When parsing XML into a python-stix data structure, stixmarx will attempt to capture any markings expressed in the XML and include them in the `MarkingContainer`.

```
container = stixmarx.parse("stix_input.xml")
package = container.package
incident = package.incidents[0]
print(container.get_markings(incident))
>>> [<stix.data_marking.MarkingSpecification object at 0x...>, ...]
```

Now that the library can operate on major components and field-level markings, stixmarx's parsing capabilities are also extended to consume component markings, top-level collections (e.g., Indicators, TTPs, Observables) and field-level markings. Alternatively, package-level markings may apply to every element and attribute in the document (e.g., `//node() | //@*`), to create a global marking.

Note that previous iterations of STIX Language documentation used `//node()` to select entire documents. This was found to be misaligned with the XPath 1.0 Specification and as such, the STIX Language documentation has been updated to reflect the proper selectors. This tool produces and consumes these selectors.

When encountering an unsupported or invalid XPath in a `<Controlled_Structure>`, the parser will fail to apply the marking.

For more parsing examples with explanations, see the *Examples* page.

1.3 Usage Notes

Below are several important notes about the design and usage of `stixmarx`. Many of these points are addressed in examples on the *Examples* page.

- When API functions say they accept a "marking" object, it means they accept a `MarkingSpecification` object, not a `MarkingStructure`. If you have a `MarkingStructure` you wish to use to mark something, you must first place it inside a `MarkingSpecification` object and supply that object to the API.
- You can now use a single `MarkingSpecification` object multiple times. The API will create copies of the marking objects when it uses them during serialization (or flush). So it is now safe to apply markings using the same `MarkingSpecification` object for each call to `add_marking` and `add_global`.
- `MarkingSpecification` objects supplied to `add` functions (`add_global`, `add_marking`) should have an empty XPath `controlled_structure` value. The XPath will be populated by the library. The API user only needs to specify what object should be marked, and the API produces the appropriate XPath for that logical marking operation.
- Some python built-in types may be coerced into markable *stixmarx.api.types* when applying markings or parsing a document with markings that resolve or involve python built-in structures (e.g. str, datetime).
- `add_marking(...)` with descendants set to `True` marks an element and all descendant elements under it. It is the equivalent of a component marking in older versions of stixmarx. Correspondingly, `get_markings(...)` will return a list of all markings that apply directly to the given element *and* all inherited markings that have been applied to that element's ancestors.
- `remove_marking(element, marking)` can only remove markings that have been applied directly to the given element. Markings inherited from ancestor elements cannot be directly removed from a descendant element.

CHAPTER 2

Getting Started

`stixmarx` is a library for applying data markings to STIX documents. It is written in Python and is intended for use by developers.

The `stixmarx` package requires the python-stix library. The `stixmarx` package provides marking functionality for `python-stix`, `python-cybox` and `python-maec` objects.

The library was made and tested with the use of the following libraries:

Library	Release	Link
python-stix	1.2.0.1	STIX
python-cybox	2.1.0.13	CybOX
python-maec	4.1.0.13	MAEC

For a conceptual overview of how data markings work in STIX, see the documentation on data markings from the STIX project.

For a description of how the library works, see *Overview*. For sample usage, see the *Examples* page.

CHAPTER 3

API Reference

class stixmarx.container.**MarkingContainer**(*package*)

Enables the operation of data markings on STIX, CybOX and MAEC objects.

A MarkingContainer provides interfaces for applying, accessing, clearing and removing data marking information from its wrapped STIX Package. A MarkingContainer has methods for processing marking information and serialization which translate the marked object model into XPath controlled structures.

Note: A MarkingContainer should not be created directly. Instead, use stixmarx.parse() or stixmarx.new() to create MarkingContainer instances.

---

package

stix.core.STIXPackage – The package object (from python-stix) wrapped by this container.

global_markings

list of MarkingSpecification – List of markings that apply to the container *package* as a whole (every descendant).

field_markings

dict, maps markable objects to MarkingSpecification – Dictionary where keys are *markable* entities and the values are a *list* that contain tuples of MarkingSpecification and descendants (True/False) option.

null_markings

(list of MarkingSpecification): List of markings that apply to this container but, will NOT mark anything inside. This means, no controlled structure will be resolved for this objects.

add_global(*marking*)

Add the *marking* MarkingSpecification object to the set of globally applicable markings (markings that apply to this container's package and all of its descendants).

Markings added here will be included in the set returned from get_markings() for any valid field.

Parameters **marking** – A MarkingSpecification object.

Raises

- TypeError – If *marking* is not a MarkingSpecification object.

- `MarkingPathNotEmpty` – If *marking* controlled_structure is set.
- `DuplicateMarkingError` – If *marking* is already present in *global_markings* collection.

add_marking(*markable*, *marking*, *descendants=False*)

Add the *marking* to the *markable* field/object. If *markable* is a built-in immutable Python type, it will be coerced into a stixmarx.api.types datatype.

Note: The add_marking() function may not always be able to apply the markings in-place. Users should set the input field to the return object after calling add_marking().

Note: Use this method to apply null markings. This is, markings that are present within the document but, do not apply to any field. The *markable* parameter MUST be None.

Example

```
>>> print type(indicator.title)
<type 'str'>
>>> marked_title = add_marking(indicator.title, marking)
>>> print type(marked_title)
<class 'stixmarx.api.types.MarkableBytes'>
>>> indicator.title = marked_title  # set the title to the return value
```

Example

```
>>> print type(indicator.timestamp)
<type 'datetime.datetime'>
>>> marked_timestamp = add_marking(indicator.timestamp, marking)
>>> print type(marked_timestamp)
<class 'stixmarx.api.types.MarkableDateTime'>
>>> indicator.timestamp = marked_timestamp # set timestamp to the return value
```

Example

```
>>> print type(indicator)
<class 'stix.indicator.indicator.Indicator'>
>>> marked_indicator = add_marking(indicator, marking, descendants=True) #␣
↪The equivalent of a component marking
>>> print type(marked_indicator)
<class 'stix.indicator.indicator.Indicator'>
>>> indicator = marked_indicator
```

Parameters

- **markable** – An object to mark (e.g., an Indicator.title string).
- **marking** – A python-stix MarkingSpecification object.
- **descendants** – If true, add the marking to all descendants *markable*.

Returns The *markable* object with data marking information attached. If *markable* is a built-in immutable Python type (e.g., str), it will be changed to a stixmarx.api.types datatype.

Raises

- `UnmarkableError` – If *markable* is a STIXPackage object.
- `DuplicateMarkingError` – If *markable* is already marked by *marking*.
- `MarkingPathNotEmpty` – If *marking* controlled_structure is set.

clear_markings(*markable*, *descendants=False*)
Remove all markings from the *markable* marked object.

Parameters

- **markable** – A marked object (e.g., indicator.title)
- **descendants** – If True, clear markings from *markable* and its descendants.

Raises `UnmarkableError` – If *markable* is not an markable entity.

field_markings
Return the field markings that have been set via add_marking().

Note: This property DOES NOT return markings that were applied by MarkingParser.

Returns Dictionary where keys are *markable* entities and values are a list of tuples with MarkingSpecification objects and their corresponding (True/False) descendants option.

Return type dict

flush()
Flush markings onto package object.

Markings are buffered in the MarkingContainer until explicitly flushed out to the MarkingContainer's package through this method.

Note: The global and fields collection will reset after this call.

Returns

A STIX Package with all makings explicitly applied from the container.

Return type stix.core.STIXPackage

get_markings(*markable*, *descendants=False*, *null_markings=False*)
Return the markings associated with the input *markable* object.

Note: This will include any global markings that have not been explicitly applied to this field.

Parameters

- **markable** – A markable object (e.g., indicator.title).
- **descendants** – If True, return markings which apply to the input field and all of its descendants.

- **null_markings** – If True, return internal markings that do NOT apply to any markable. This null markings have not been explicitly set to the wrapped document. Use utils.get_null_markings(...) to find null markings that have been explicitly set in the document.

Returns A list of MarkingSpecification objects.

Return type list

global_markings
Return the global markings that have been set via add_global().

Note: This property DOES NOT return markings that were applied by MarkingParser (even markings that were applied to all nodes in the parsed document).

Returns Tuple containing MarkingSpecification objects.

Return type tuple

is_marked(*markable*, *marking=None*, *descendants=False*)
Return True if *markable* contains marking information.

Parameters

- **markable** – An markable object.
- **marking** – A MarkingSpecification object.
- **descendants** – If set, inspect descendant fields for marking information.

Raises

- `UnmarkableError` – If *markable* is not an markable entity.
- `UnknownMarkingError` – If *marking* is not a MarkingSpecification object.

Returns

True under the following conditions: if *markable* contains marking information, if *markable* is marked by *marking*, if *markable* descendants contain markings or if global markings have been added through add_global(). Otherwise False.

Return type bool

null_markings
Return the null markings that have been set via add_markings(). Where *markable* is None.

Note: This property DOES NOT return markings that were applied by MarkingParser.

Returns Tuple containing MarkingSpecification objects.

Return type tuple

package
Package wrapped by this MarkingContainer

remove_global(*marking*)
Remove a globally-applied marking from the internal collection or from a parsed document that contain globally-applied markings.

Parameters **marking** (*MarkingSpecification*) – marking to un-apply from global

Raises MarkingNotFoundError – If *marking* is not found in the global markings registry.

remove_marking(*markable*, *marking*, *descendants=False*)
Remove the *marking* MarkingSpecification from *markable*.

Note: Use remove_global to remove globally applied markings.

Parameters

- **markable** – An object which contains data markings.
- **marking** – A MarkingSpecification object.
- **descendants** – If True, remove *marking* from any descendants.

Raises

- UnmarkableError – If *markable* is not an markable entity.
- MarkingNotFoundError – If *markable* (or descendant of *markable* if *descendants* is True) is marked by *marking*. If marking was not found in the internal marking collection.
- MarkingRemovalError – If *marking* is inherited from an ancestor OR if *markable* is STIXPackage object.
- UnknownMarkingError – If *marking* is not a MarkingSpecification object.

to_dict(**args*, ***kwargs*)
Return a dictionary of the STIX Package represented by the Package object, with markings applied through the MarkingContainer.

Uses the same arguments as stix.Entity.to_dict().

to_xml(**args*, ***kwargs*)
Return an XML string of the STIX package represented by the Package object, with markings applied through the MarkingContainer.

Uses the same arguments as stix.Entity.to_xml().

CHAPTER 4

Examples

The following examples will provide developers initial background on how to use the stixmarx API.

4.1 Managing Markings

4.1.1 Creating a Container

To apply any markings, we first need to create a `MarkingContainer`. The `stixmarx.new()` function creates a new container, with an empty STIX Package inside.

```
import stixmarx

container = stixmarx.new()
stix_package = container.package
```

You can also create a container with a non-empty package by supplying an XML filepath to `stixmarx.parse(...)`:

```
import stixmarx

container = stixmarx.parse("stix_input.xml")
stix_package = container.package
```

Or supply a `read`-able IO object:

```
import stixmarx
from mixbox.vendor.six import StringIO

xml_string = "<stix:STIX_Package>..."
xml_readable = StringIO(xml_string)

container = stixmarx.parse(xml_readable)
stix_package = container.package
```

When parsing the XML input, the library will attempt to apply any markings found in `<Handling>` sections. This behavior is limited to markings defined by XPaths that the library can handle; see *Parsing Markings* for more information.

4.1.2 Applying Markings

STIX Components can be marked by `MarkingContainer::add_marking`. The `add_marking` function accepts a STIX, CybOX and MAEC entity, a `MarkingSpecification` object and a optional `True` or `False` value to signal the container the marking applies only to the entity or to the entity and all descendants.

The following example resembles the regular component marking.

```
import stixmarx
from stix.indicator import Indicator
from stix.data_marking import MarkingSpecification
from stix.extensions.marking.tlp import TLPMarkingStructure

container = stixmarx.new()
stix_package = container.package

indicator = Indicator(title="Test")
stix_package.add_indicator(indicator)

marking_struct = TLPMarkingStructure(color='RED')
marking_spec = MarkingSpecification()
marking_spec.marking_structures.append(marking_struct)

container.add_marking(indicator, marking_spec, descendants=True)
```

Note that marking functions accept a `MarkingSpecification` object, not a `MarkingStructure` object. A `MarkingSpecification` object contains `MarkingStructure` objects (per the `marking_spec.marking_structures.append(marking_struct)` line). The marking object's `controlled_structure` property should not be set. The API will set it for you, based on how you use the API to apply the marking.

Markings can be applied to an entire document with `add_global`:

```
import stixmarx
from stix.indicator import Indicator
from stix.data_marking import MarkingSpecification
from stix.extensions.marking.tlp import TLPMarkingStructure

container = stixmarx.new()
stix_package = container.package

indicator = Indicator(title="Test")
stix_package.add_indicator(indicator)

marking_struct = TLPMarkingStructure(color='RED')
marking_spec = MarkingSpecification()
marking_spec.marking_structures.append(marking_struct)

container.add_global(marking_spec)
```

Note that you cannot mark a STIX Package object with `add_marking`. Instead, you must use `add_global`.

User's may also apply markings to specific attributes or text from the document.

```
import stixmarx
from stix.indicator import Indicator
from stix.data_marking import MarkingSpecification
from stix.extensions.marking.tlp import TLPMarkingStructure

container = stixmarx.new()
stix_package = container.package

indicator = Indicator(title="Test")
indicator.description = "A test description"
stix_package.add_indicator(indicator)

marking_struct = TLPMarkingStructure(color='RED')
marking_spec = MarkingSpecification()
marking_spec.marking_structures.append(marking_struct)

indicator.description.value = container.add_marking(indicator.description.value,␣
↪marking_spec)
```

Note: When markings to attribute or text are applied, the descendants option is ignored. Also, if the values to mark are Python built-in types, they will be coerced into `stixmarx.api.types` objects according to the following table:

Python built-in type	stixmarx.api.types
bool	MarkableBool
int	MarkableInt
long (only in Python 2)	MarkableLong
float	MarkableFloat
six.binary_type	MarkableBytes
six.text_type	MarkableText
datetime.date	MarkableDate
datetime.datetime	MarkableDateTime

Attribute example,

```
import stixmarx
from stix.indicator import Indicator
from stix.data_marking import MarkingSpecification
from stix.extensions.marking.tlp import TLPMarkingStructure

container = stixmarx.new()
stix_package = container.package

indicator = Indicator(title="Test")
stix_package.add_indicator(indicator)

marking_struct = TLPMarkingStructure(color='RED')
marking_spec = MarkingSpecification()
marking_spec.marking_structures.append(marking_struct)

indicator.timestamp = container.add_marking(indicator.timestamp, marking_spec)
```

Marking only the node example,

```
import stixmarx
from stix.indicator import Indicator
from stix.data_marking import MarkingSpecification
from stix.extensions.marking.tlp import TLPMarkingStructure
```

```
container = stixmarx.new()
stix_package = container.package

indicator = Indicator(title="Test")
stix_package.add_indicator(indicator)

marking_struct = TLPMarkingStructure(color='RED')
marking_spec = MarkingSpecification()
marking_spec.marking_structures.append(marking_struct)

container.add_marking(indicator.title, marking_spec)
```

4.1.3 Reading Markings

The `MarkingContainer::get_markings` function returns a list of markings that apply to an element.

```
import stixmarx
from stix.indicator import Indicator
from stix.data_marking import MarkingSpecification
from stix.extensions.marking.tlp import TLPMarkingStructure

container = stixmarx.new()
stix_package = container.package
indicator = Indicator(title="Test")
stix_package.add_indicator(indicator)

marking_spec = MarkingSpecification(marking_structures=TLPMarkingStructure(color='RED
↪'))
container.add_marking(indicator, marking_spec)

print container.get_markings(indicator)
```

Since markings are applied recursively to descendants, any descendant elements nested inside of a marked element also report the ancestor's markings.

```
import stixmarx
from stix.indicator import Indicator
from stix.incident import Incident
from stix.data_marking import MarkingSpecification
from stix.extensions.marking.tlp import TLPMarkingStructure

container = stixmarx.new()
package = container.package

incident = Incident(title="My incident")
package.add_incident(incident)
indicator = Indicator(title="Sample indicator")
incident.related_indicators.append(indicator)

green_marking_spec = MarkingSpecification(marking_
↪structures=TLPMarkingStructure(color='GREEN'))
container.add_global(green_marking_spec)

amber_marking_spec = MarkingSpecification(marking_
↪structures=TLPMarkingStructure(color='AMBER'))
container.add_marking(incident, amber_marking_spec, descendants=True)
```

```
red_marking_spec = MarkingSpecification(marking_structures=TLPMarkingStructure(color=
↪'RED'))
container.add_marking(indicator, red_marking_spec, descendants=True)

print container.get_markings(incident)
print container.get_markings(indicator)
```

This will show a list of two `MarkingSpecification` objects applied to the Incident: global GREEN and local AMBER. It will show three markings on the nested Indicator: global GREEN, parent AMBER, and local RED.

Global markings for a container are stored in the `MarkingContainer::global_markings` list. While component and field markings are stored in `MarkingContainer::field_markings`.

4.1.4 Removing Markings

To remove a marking, use `MarkingContainer::remove_marking`.

```
import stixmarx
from stix.indicator import Indicator
from stix.data_marking import MarkingSpecification
from stix.extensions.marking.tlp import TLPMarkingStructure

container = stixmarx.new()
package = container.package

indicator = Indicator(title="Test")
package.add_indicator(indicator)

marking_spec = MarkingSpecification(marking_structures=TLPMarkingStructure(color='RED
↪'))

container.add_marking(indicator, marking_spec, descendants=True)
print container.get_markings(indicator)

container.remove_marking(indicator, marking_spec, descendants=True)
print container.get_markings(indicator)
```

A marking can only be removed from an element if that marking was originally applied directly to that element. That means, you cannot remove a marking inherited from an ancestor. The rule applies for both situations: generating or parsing existing content.

```
import stixmarx
from stix.indicator import Indicator
from stix.incident import Incident
from stix.data_marking import MarkingSpecification
from stix.extensions.marking.tlp import TLPMarkingStructure

container = stixmarx.new()
package = container.package

marking_spec = MarkingSpecification(marking_structures=TLPMarkingStructure(color='RED
↪'))

incident = Incident(title="Test Incident")
package.add_incident(incident)
```

```
indicator = Indicator()
incident.related_indicators.append(indicator)

container.add_marking(incident, marking_spec, descendants=True)

# show marking, inherited from incident
print container.get_markings(indicator)

# ERROR: indicator was never marked; it inherits a marking from incident
container.remove_marking(indicator, marking_spec)
```

Also, when generating content, the same marking and descendants option MUST be supplied to `MarkingContainer::remove_marking` in order to properly remove the marking.

Global markings can be removed via the `MarkingContainer::remove_global` method.

```
import stixmarx
from stix.indicator import Indicator
from stix.incident import Incident
from stix.data_marking import MarkingSpecification
from stix.extensions.marking.tlp import TLPMarkingStructure

container = stixmarx.new()
package = container.package

marking_spec = MarkingSpecification(marking_structures=TLPMarkingStructure(color='RED
↪'))

incident = Incident(title="Test Incident")
package.add_incident(incident)

indicator = Indicator()
incident.related_indicators.append(indicator)

container.add_global(marking_spec)

# show marking, inherited from global
print container.get_markings(indicator)

container.remove_global(marking_spec)
```

4.1.5 Observable Markings

Observables can be marked in the same way that STIX components are marked.

```
import stixmarx
from stix.data_marking import MarkingSpecification
from stix.extensions.marking.tlp import TLPMarkingStructure
from cybox.core import Observable
from cybox.objects.address_object import Address

container = stixmarx.new()
package = container.package

red_marking = MarkingSpecification(marking_structures=TLPMarkingStructure(color='RED
↪'))
```

```
observable = Observable(Address(address_value='10.0.0.1'))
package.add_observable(observable)

container.add_marking(observable, red_marking, descendants=True)

print container.get_markings(observable)
```

However, because observables cannot contain their own `Marking` element, and must be marked externally, stixmarx will store the resulting `Marking` under the STIX_Header of the document when there is no available ancestor with a `Handling` element. Note: `id` attributes are no longer used for XPath generation since they are optional.

4.1.6 Top Collections Markings

The stixmarx API is now capable of marking Top Collections, for example: (TTPs, Indicators, Observables). Top Collections can be marked the same way as any other STIX component.

```
import stixmarx
from stix.data_marking import MarkingSpecification
from stix.extensions.marking.tlp import TLPMarkingStructure
from cybox.core import Observable
from cybox.objects.address_object import Address

container = stixmarx.new()
package = container.package

red_marking = MarkingSpecification(marking_structures=TLPMarkingStructure(color='RED
↪'))

observable = Observable(Address(address_value='10.0.0.1'))
package.add_observable(observable)

container.add_marking(package.observables, red_marking, descendants=True)

print container.get_markings(observable)
```

4.1.7 Output Markings

To output an XML document string with markings included, use `MarkingContainer::to_xml`:

```
import stixmarx
from stix.data_marking import MarkingSpecification
from stix.extensions.marking.tlp import TLPMarkingStructure

container = stixmarx.parse("input_package.xml")

marking_spec = MarkingSpecification(marking_structures=TLPMarkingStructure(color='RED
↪'))

container.add_global(marking_spec)
print container.to_xml()
```

`MarkingContainer::to_xml` takes the exact same arguments as python-stix's `Entity::to_xml`.

4.2 Parsing Markings

These examples demonstrate how to write XML that can be parsed into marking data structures by stixmarx. See the *Parsing Markings* section from the *Overview* page for more information.

4.2.1 Parsing STIX Components

To mark an element from its <Handling> section, the XPath `../../../descendant-or-self::node() | ../../../descendant-or-self::node()/@*` is appropriate. This XPath should go in a <Controlled_Structure>, inside a <Marking>, inside the element being marked, like an <Indicator>.

Suppose we have a this content in `doc.xml`, where a RED TLP marking has been applied to an Indicator:

```
<stix:STIX_Package
    xmlns:example="http://example.com"
    xmlns:indicator="http://stix.mitre.org/Indicator-2"
    xmlns:marking="http://data-marking.mitre.org/Marking-1"
    xmlns:stix="http://stix.mitre.org/stix-1"
    xmlns:tlpMarking="http://data-marking.mitre.org/extensions/MarkingStructure#TLP-1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="example:Package-
↪5c5ecffe-9025-42ac-83ea-a3b7158bea1a" version="1.2">
    <stix:Indicators>
        <stix:Indicator id="example:indicator-64d3d246-487e-4003-9366-90aa2b49570c"␣
↪timestamp="2016-03-30T15:58:30.662000Z" xsi:type='indicator:IndicatorType'>
            <indicator:Title>Test Title</indicator:Title>
            <indicator:Handling>
                <marking:Marking>
                    <marking:Controlled_Structure>../../../descendant-or-self::node()␣
↪| ../../../descendant-or-self::node()/@*</marking:Controlled_Structure>
                    <marking:Marking_Structure xsi:type=
↪'tlpMarking:TLPMarkingStructureType' color='RED'/>
                </marking:Marking>
            </indicator:Handling>
        </stix:Indicator>
    </stix:Indicators>
</stix:STIX_Package>
```

Then we can parse the `doc.xml` document and inspect the marking like so:

```
import stixmarx

container = stixmarx.parse("doc.xml")
stix_package = container.package

indicator = stix_package.indicators[0]
print container.get_markings(indicator)
```

4.2.2 Parsing Global Markings

Markings in the the <Handling> section of the <STIX_Header> may apply to every element and attribute in the document (e.g., `//node | //@*`). This corresponds to a global marking in the API. Now, MarkingContainer does not return global markings from it's global_markings. User's can check the STIX Package for markings.

```
<stix:STIX_Package
    xmlns:example="http://example.com"
    xmlns:indicator="http://stix.mitre.org/Indicator-2"
    xmlns:marking="http://data-marking.mitre.org/Marking-1"
    xmlns:stix="http://stix.mitre.org/stix-1"
    xmlns:tlpMarking="http://data-marking.mitre.org/extensions/MarkingStructure#TLP-1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="example:Package-
↪3972115d-3e88-4688-8dff-6208bcf1db55" version="1.2">
    <stix:STIX_Header>
        <stix:Handling>
            <marking:Marking>
                <marking:Controlled_Structure>//node() | //@*</marking:Controlled_
↪Structure>
                <marking:Marking_Structure xsi:type=
↪'tlpMarking:TLPMarkingStructureType' color='RED'/>
            </marking:Marking>
        </stix:Handling>
    </stix:STIX_Header>
    <stix:Indicators>
        <stix:Indicator id="example:indicator-08174fbc-115f-46d6-890b-4214fbd8c3e4"␣
↪timestamp="2016-03-30T15:58:30.662000Z" xsi:type='indicator:IndicatorType'>
            <indicator:Title>Test</indicator:Title>
        </stix:Indicator>
    </stix:Indicators>
</stix:STIX_Package>
```

We can parse the document (named, e.g., `doc.xml`) and read the global marking like so:

```
import stixmarx

container = stixmarx.parse("doc.xml")
package = container.package

print container.get_markings(package)
```

Note: The example above will only return globally applied markings. Now, if a user desires to capture all markings present in the document regardless of the scope (global, component or field) see example below.

```
<stix:STIX_Package
    xmlns:example="http://example.com"
    xmlns:indicator="http://stix.mitre.org/Indicator-2"
    xmlns:marking="http://data-marking.mitre.org/Marking-1"
    xmlns:stix="http://stix.mitre.org/stix-1"
    xmlns:tlpMarking="http://data-marking.mitre.org/extensions/MarkingStructure#TLP-1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="example:Package-
↪3972115d-3e88-4688-8dff-6208bcf1db55" version="1.2">
    <stix:STIX_Header>
        <stix:Handling>
            <marking:Marking>
                <marking:Controlled_Structure>//node() | //@*</marking:Controlled_
↪Structure>
                <marking:Marking_Structure xsi:type=
↪'tlpMarking:TLPMarkingStructureType' color='RED'/>
            </marking:Marking>
        </stix:Handling>
    </stix:STIX_Header>
    <stix:Indicators>
        <stix:Indicator id="example:indicator-08174fbc-115f-46d6-890b-4214fbd8c3e4"␣
↪timestamp="2016-03-30T15:58:30.662000Z" xsi:type='indicator:IndicatorType'>
```

```
            <indicator:Title>Test</indicator:Title>
            <stix:Handling>
                <marking:Marking>
                    <marking:Controlled_Structure>../../../descendant-or-self::node()␣
↪| ../../../descendant-or-self::node()/@*</marking:Controlled_Structure>
                    <marking:Marking_Structure xsi:type=
↪'tlpMarking:TLPMarkingStructureType' color='AMBER'/>
                </marking:Marking>
            </stix:Handling>
        </stix:Indicator>
    </stix:Indicators>
</stix:STIX_Package>
```

Parse the document and return all markings present in the document.

```
import stixmarx

container = stixmarx.parse("doc.xml")
package = container.package

print container.get_markings(package, descendants=True)
```

4.2.3 Parsing Observable Markings

Since CybOX observables do not have their own <Handling> element, they must be marked by a STIX element that contains them. In stixmarx, this is done via a top <STIX_Header>-level XPath that applies to the observable. It is not required for an Observable to have an id to be marked.

Here is an example with an observable:

```
<stix:STIX_Package
    xmlns:example="http://example.com"
    xmlns:AddressObj="http://cybox.mitre.org/objects#AddressObject-2"
    xmlns:cybox="http://cybox.mitre.org/cybox-2"
    xmlns:indicator="http://stix.mitre.org/Indicator-2"
    xmlns:marking="http://data-marking.mitre.org/Marking-1"
    xmlns:stix="http://stix.mitre.org/stix-1"
    xmlns:tlpMarking="http://data-marking.mitre.org/extensions/MarkingStructure#TLP-1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="example:Package-
↪9cec4a05-3e12-45aa-acda-0c514aeec88e" version="1.2">
    <stix:STIX_Header>
        <stix:Handling>
            <marking:Marking>
                <marking:Controlled_Structure>../../../../stix:Observables[1]/
↪descendant-or-self::node() | ../../../../stix:Observables[1]/descendant-or-
↪self::node()/@*</marking:Controlled_Structure>
                <marking:Marking_Structure xsi:type=
↪'tlpMarking:TLPMarkingStructureType' color='RED'/>
            </marking:Marking>
        </stix:Handling>
    </stix:STIX_Header>
    <stix:Observables cybox_major_version="2" cybox_minor_version="1" cybox_update_
↪version="0">
        <cybox:Observable id="example:Observable-70b700de-bf18-42c6-9ff1-b6b75ecc9873
↪">
            <cybox:Object id="example:Address-d828f9b1-0069-4448-b3ad-81bf57075ff5">
```

```
                <cybox:Properties xsi:type="AddressObj:AddressObjectType">
                    <AddressObj:Address_Value>10.0.0.1</AddressObj:Address_Value>
                </cybox:Properties>
            </cybox:Object>
        </cybox:Observable>
    </stix:Observables>
</stix:STIX_Package>
```

When this document (named `doc.xml` below) is parsed, the observable marking can be read:

```
import stixmarx

container = stixmarx.parse("doc.xml")
stix_package = container.package
observable = stix_package.observables[0]

print container.get_markings(observable)
```

4.2.4 Parsing Field-Level Markings

To access field level markings from a parsed document, use their corresponding attribute.

```
<stix:STIX_Package
    xmlns:example="http://example.com"
    xmlns:indicator="http://stix.mitre.org/Indicator-2"
    xmlns:marking="http://data-marking.mitre.org/Marking-1"
    xmlns:stix="http://stix.mitre.org/stix-1"
    xmlns:tlpMarking="http://data-marking.mitre.org/extensions/MarkingStructure#TLP-1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="example:Package-
↪3972115d-3e88-4688-8dff-6208bcf1db55" version="1.2">
    <stix:Indicators>
        <stix:Indicator id="example:indicator-08174fbc-115f-46d6-890b-4214fbd8c3e4"␣
↪timestamp="2016-03-30T15:58:30.662000Z" xsi:type='indicator:IndicatorType'>
            <indicator:Title>Test</indicator:Title>
            <indicator:Handling>
                <marking:Marking>
                    <marking:Controlled_Structure>../../../indicator:Title[1]/
↪self::node()</marking:Controlled_Structure>
                    <marking:Marking_Structure xsi:type=
↪'tlpMarking:TLPMarkingStructureType' color='RED'/>
                </marking:Marking>
            </indicator:Handling>
        </stix:Indicator>
    </stix:Indicators>
</stix:STIX_Package>
```

When this document is parsed, the field level markings can be read:

```
import stixmarx

container = stixmarx.parse("doc.xml")
package = container.package
indicator = package.indicators[0]

print container.get_markings(indicator.title)
# Note: Only the Title node is marked! Not its text.
```

4.2.5 Parsing Text or Attribute Markings

To access text or attribute markings from a parsed document, use their corresponding property.

```
<stix:STIX_Package
    xmlns:example="http://example.com"
    xmlns:indicator="http://stix.mitre.org/Indicator-2"
    xmlns:marking="http://data-marking.mitre.org/Marking-1"
    xmlns:stix="http://stix.mitre.org/stix-1"
    xmlns:tlpMarking="http://data-marking.mitre.org/extensions/MarkingStructure#TLP-1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="example:Package-
↪3972115d-3e88-4688-8dff-6208bcf1db55" version="1.2">
    <stix:Indicators>
        <stix:Indicator id="example:indicator-08174fbc-115f-46d6-890b-4214fbd8c3e4"␣
↪timestamp="2016-03-30T15:58:30.662000Z" xsi:type='indicator:IndicatorType'>
            <indicator:Title>Test Title</indicator:Title>
            <indicator:Description>Test Description</indicator:Description>
            <indicator:Handling>
                <marking:Marking>
                    <marking:Controlled_Structure>../../../indicator:Description[1]/
↪text()</marking:Controlled_Structure>
                    <marking:Marking_Structure xsi:type=
↪'tlpMarking:TLPMarkingStructureType' color='RED'/>
                </marking:Marking>
                <marking:Marking>
                    <marking:Controlled_Structure>../../../@timestamp</
↪marking:Controlled_Structure>
                    <marking:Marking_Structure xsi:type=
↪'tlpMarking:TLPMarkingStructureType' color='AMBER'/>
                </marking:Marking>
            </indicator:Handling>
        </stix:Indicator>
    </stix:Indicators>
</stix:STIX_Package>
```

When this document is parsed, the field level markings can be read:

```
import stixmarx

container = stixmarx.parse("doc.xml")
package = container.package
indicator = package.indicators[0]

print container.get_markings(indicator.description.value)
print container.get_markings(indicator.timestamp)
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index