Beagle

Dec 03, 2019

Contents

CHAPTER 1

Subpackages

1.1 beagle.backends package

1.1.1 Submodules

1.1.2 beagle.backends.base_backend module

class beagle.backends.base_backend.**Backend**(*nodes: List[beagle.nodes.node.Node]*)

Bases: object

Abstract Backend Class. All Backends must implement the *graph()* method in order to properly function.

When creating a new backend, you should really subclass the NetworkX class instead, and work on translating the NetworkX object to the other datasource.

See *beagle.backends.networkx.NetworkX*

Parameters **nodes** (*List[Node]*) – Nodes produced by the transformer.

Example

```
>>> nodes = FireEyeHXTransformer(datasource=HXTriage('test.mans'))
>>> backend = BackEndClass(nodes=nodes)
>>> backend.graph()
```

add_nodes(*nodes: List[beagle.nodes.node.Node]*)

This function should allow (or raise an error if not possible to) a user to add additional nodes to an already existing graph.

Parameters **nodes** (*List[Node]*) – The new nodes to add to the graph.

classmethod **from_datasources**(*datasources: Union[DataSource, List[DataSource]], *args, **kwargs*) → Backend

Create a backend instance from a set of datasources

Parameters **datasources** (*Union[DataSource, List[DataSource]]*) – A set of datasources to use when creating the backend.

Returns Returns the configured instance

Return type *Backend*

graph() → Union[str, Any]
When this method is called, the backend should take in the passed in *Node* array and produce a graph.

is_empty() → bool
Returns true if there wasn't a graph created.

to_json() → dict

1.1.3 beagle.backends.dgraph module

class beagle.backends.dgraph.**DGraph**(*host: str = ''*, *batch_size: int = 1000*, *wipe_db: bool = False*, **args*, ***kwargs*)
Bases: *beagle.backends.networkx.NetworkX*

DGraph backend (https://dgraph.io). This backend builds a schema using the *_setup_schema* function. It then pushes each node and retrieves it's assigned UID. Once all nodes are pushed, edges are pushed to the graph by mapping the node IDs to the assigned UIDs

Parameters

- **host** (*str, optional*) – The hostname of the DGraph instance (the default is Config.get("dgraph", "host"), which pulls from the configuration file)
- **batch_size** (*int, optional*) – The number of edges and nodes to push in to the database at a time. (the default is int(Config.get("dgraph", "batch_size")), which pulls from the configuration file)
- **wipe_db** (*bool, optional*) – Wipe the Database before inserting new data. (the default is False)

graph()
Pushes the nodes and edges into DGraph.

setup_schema() → None
Sets up the DGraph schema based on the nodes. This inspect all attributes of all nodes, and generates a schema for them. Each schema entry has the format *{node_type}.{field}*. If a field is a string field, it has the *@index(exact)* predicate added to it.

An example output schema:

```
process.process_image string @index(exact)
process.process_id int
```

1.1.4 beagle.backends.graphistry module

class beagle.backends.graphistry.**Graphistry**(*anonymize: bool = False*, *render: bool = False*, **args*, ***kwargs*)
Bases: *beagle.backends.networkx.NetworkX*

Visualizes the graph using the graphistry platform (https://www.graphistry.com/).

Examples

```
>>> SysmonEVTX('sysmon_evtx_file.evtx').to_graph(Graphistry, render=True)
```

Parameters

- **anonymize** (*bool, optional*) – Should the data be anonymized before sending to graphistry? (the default is False, which does not.)
- **render** (*bool, optional*) – Should the result of *graph()* be a IPython widget? (default value is False, which returns the URL).

anonymize_graph() → networkx.classes.multidigraph.MultiDiGraph
Anonymizes the underlying graph before sending to Graphistry.

Returns The same graph structure, but without attributes.

Return type nx.MultiDiGraph

graph()
Return the Graphistry URL for the graph, or an IPython Widget

Parameters **render** (*bool, optional*) – Should the result be a IPython widget? (default value is False, which returns the URL).

Returns str with URL to graphistry object when render if False, otherwise HTML widget for IPython.

Return type Union[str, IPython.core.display.HTML]

1.1.5 beagle.backends.neo4j module

class beagle.backends.neo4j.**Neo4J**(*uri: str = ''*, *username: str = ''*, *password: str = ''*, *clear_database: bool = False*, **args*, ***kwargs*)
Bases: *beagle.backends.networkx.NetworkX*

Neo4J backend. Converts each node and edge to a Cypher and uses BATCH UNWIND queries to push nodes at once.

Parameters

- **uri** (*str, optional*) – Neo4J Hostname (the default is Config.get("neo4j", "host"), which pulls from the configuration file)
- **username** (*str, optional*) – Neo4J Username (the default is Config.get("neo4j", "username"), which pulls from the configuration file)
- **password** (*str, optional*) – Neo4J Password (the default is Config.get("neo4j", "password"), which pulls from the configuration file)
- **clear_database** (*bool, optional*) – Should the database be cleared before populating? (the default is False)

graph() → str
Generates the MultiDiGraph.

Places the nodes in the Graph.

Returns The generated NetworkX object.

Return type nx.MultiDiGraph

1.1.6 beagle.backends.networkx module

class beagle.backends.networkx.**NetworkX**(*metadata: dict = {}*, *consolidate_edges: bool = False*, **args*, ***kwargs*)

Bases: *beagle.backends.base_backend.Backend*

NetworkX based backend. Other backends can subclass this backend in order to have access to the underlying NetworkX object.

While inserting the Nodes into the graph, the NetworkX object does the following:

1. If the ID of this node (calculated via *Node.__hash__*) is already in the graph, the node is updated with any properties which are in the new node but not the existing node.

2. If we are inserting the an edge type that already exists between two nodes *u* and *v*, the edge data is combined.

Notes

In *networkx*, adding the same node twice keeps the latest version of the node. Since a node that represents the same thing may appear twice in a log (for example, the same process might appear in a process creation event and a file write event). It's easier to simply update the nodes as you iterate over the *nodes* attribute.

Parameters

- **metadata** (*dict, optional*) – The metadata from the datasource.
- **consolidate_edges** (*boolean, optional*) – Controls if edges are consolidated. That is, if the edge of type q from u to v happens N times, should there be one edge from u to v with type q, or should there be N edges.

Notes

Putting

add_nodes(*nodes: List[beagle.nodes.node.Node]*) → networkx.classes.multidigraph.MultiDiGraph
This function should allow (or raise an error if not possible to) a user to add additional nodes to an already existing graph.

Parameters **nodes** (*List[Node]*) – The new nodes to add to the graph.

static from_json(*path_or_obj: Union[str, dict]*) → networkx.classes.multidigraph.MultiDiGraph

graph() → networkx.classes.multidigraph.MultiDiGraph
Generates the MultiDiGraph.

Places the nodes in the Graph.

Returns The generated NetworkX object.

Return type nx.MultiDiGraph

classmethod graph_to_json(*graph: networkx.classes.multidigraph.MultiDiGraph*) → dict

insert_edges(*u: beagle.nodes.node.Node, v: beagle.nodes.node.Node, edge_name: str, instances: List[dict]*) → None
Inserts instances of an edge of type *edge_name* from node *u* to *v*

Parameters

- **u** (Node) – Source Node object
- **v** (Node) – Destination Node object

- **edge_name** (*str*) – Edge Name
- **instances** (*List[dict]*) – The data entries for the node between *u* and *v*.

insert_node(*node: beagle.nodes.node.Node*, *node_id: int*) → None
Inserts a node into the graph, as well as all edges outbound from it.

Parameters

- **node** (Node) – Node object to insert
- **node_id** (*int*) – The ID of the node (*hash(node)*)

is_empty() → bool
Returns true if there wasn't a graph created.

to_json() → dict
Convert the graph to JSON, which can later be used be read in using networkx:

```
>>> backend = NetworkX(nodes=nodes)
>>> G = backend.graph()
>>> data = G.to_json()
>>> parsed = networkx.readwrite.json_graph.node_link_graph(data)
```

Returns node_link compatible version of the graph.

Return type dict

update_node(*node: beagle.nodes.node.Node*, *node_id: int*) → None
Update the attributes of a node. Since we may see the same Node in multiple events, we want to have the largest coverage of its attributes. * See *beagle.nodes.node.Node* for how we determine two nodes are the same.

This method updates the node already in the graph with the newest attributes from the passed in parameter *Node*

Parameters

- **node** (Node) – The Node object to use to update the node already in the graph
- **node_id** (*int*) – The hash of the Node. see `beagle.nodes.node.__hash__()`

Notes

Since nodes are de-duplicated before being inserted into the graph, this should only be used to manually add in new data.

1.1.7 Module contents

class beagle.backends.**Backend**(*nodes: List[beagle.nodes.node.Node]*)
Bases: `object`

Abstract Backend Class. All Backends must implement the *graph()* method in order to properly function.

When creating a new backend, you should really subclass the NetworkX class instead, and work on translating the NetworkX object to the other datasource.

See *beagle.backends.networkx.NetworkX*

Parameters **nodes** (*List[Node]*) – Nodes produced by the transformer.

Example

```
>>> nodes = FireEyeHXTransformer(datasource=HXTriage('test.mans'))
>>> backend = BackEndClass(nodes=nodes)
>>> backend.graph()
```

add_nodes(*nodes: List[beagle.nodes.node.Node]*)
This function should allow (or raise an error if not possible to) a user to add additional nodes to an already existing graph.

Parameters **nodes** (*List[Node]*) – The new nodes to add to the graph.

classmethod from_datasources(*datasources: Union[DataSource, List[DataSource]], *args, **kwargs*) → Backend
Create a backend instance from a set of datasources

Parameters **datasources** (*Union[DataSource, List[DataSource]]*) – A set of datasources to use when creating the backend.

Returns Returns the configured instance

Return type *Backend*

graph() → Union[str, Any]
When this method is called, the backend should take in the passed in *Node* array and produce a graph.

is_empty() → bool
Returns true if there wasn't a graph created.

to_json() → dict

class beagle.backends.**DGraph**(*host: str = ''*, *batch_size: int = 1000*, *wipe_db: bool = False*, **args*, ***kwargs*)
Bases: *beagle.backends.networkx.NetworkX*

DGraph backend (https://dgraph.io). This backend builds a schema using the *_setup_schema* function. It then pushes each node and retrieves it's assigned UID. Once all nodes are pushed, edges are pushed to the graph by mapping the node IDs to the assigned UIDs

Parameters

- **host** (*str, optional*) – The hostname of the DGraph instance (the default is Config.get("dgraph", "host"), which pulls from the configuration file)
- **batch_size** (*int, optional*) – The number of edges and nodes to push in to the database at a time. (the default is int(Config.get("dgraph", "batch_size")), which pulls from the configuration file)
- **wipe_db** (*bool, optional*) – Wipe the Database before inserting new data. (the default is False)

graph()
Pushes the nodes and edges into DGraph.

setup_schema() → None
Sets up the DGraph schema based on the nodes. This inspect all attributes of all nodes, and generates a schema for them. Each schema entry has the format *{node_type}.{field}*. If a field is a string field, it has the *@index(exact)* predicate added to it.

An example output schema:

```
process.process_image string @index(exact)
process.process_id int
```

class beagle.backends.**Graphistry**(*anonymize: bool = False*, *render: bool = False*, **args*, ***kwargs*)

Bases: *beagle.backends.networkx.NetworkX*

Visualizes the graph using the graphistry platform (https://www.graphistry.com/).

Examples

```
>>> SysmonEVTX('sysmon_evtx_file.evtx').to_graph(Graphistry, render=True)
```

Parameters

- **anonymize** (*bool, optional*) – Should the data be anonymized before sending to graphistry? (the default is False, which does not.)
- **render** (*bool, optional*) – Should the result of *graph()* be a IPython widget? (default value is False, which returns the URL).

anonymize_graph() → networkx.classes.multidigraph.MultiDiGraph

Anonymizes the underlying graph before sending to Graphistry.

Returns The same graph structure, but without attributes.

Return type nx.MultiDiGraph

graph()

Return the Graphistry URL for the graph, or an IPython Widget

Parameters **render** (*bool, optional*) – Should the result be a IPython widget? (default value is False, which returns the URL).

Returns str with URL to graphistry object when render if False, otherwise HTML widget for IPython.

Return type Union[str, IPython.core.display.HTML]

class beagle.backends.**Neo4J**(*uri: str = ''*, *username: str = ''*, *password: str = ''*, *clear_database: bool = False*, **args*, ***kwargs*)

Bases: *beagle.backends.networkx.NetworkX*

Neo4J backend. Converts each node and edge to a Cypher and uses BATCH UNWIND queries to push nodes at once.

Parameters

- **uri** (*str, optional*) – Neo4J Hostname (the default is Config.get("neo4j", "host"), which pulls from the configuration file)
- **username** (*str, optional*) – Neo4J Username (the default is Config.get("neo4j", "username"), which pulls from the configuration file)
- **password** (*str, optional*) – Neo4J Password (the default is Config.get("neo4j", "password"), which pulls from the configuration file)
- **clear_database** (*bool, optional*) – Should the database be cleared before populating? (the default is False)

graph() → str

Generates the MultiDiGraph.

Places the nodes in the Graph.

Returns The generated NetworkX object.

Return type nx.MultiDiGraph

class beagle.backends.**NetworkX**(*metadata: dict = {}*, *consolidate_edges: bool = False*, **args*, ***kwargs*)

Bases: *beagle.backends.base_backend.Backend*

NetworkX based backend. Other backends can subclass this backend in order to have access to the underlying NetworkX object.

While inserting the Nodes into the graph, the NetworkX object does the following:

1. If the ID of this node (calculated via *Node.__hash__*) is already in the graph, the node is updated with any properties which are in the new node but not the existing node.

2. If we are inserting the an edge type that already exists between two nodes *u* and *v*, the edge data is combined.

Notes

In *networkx*, adding the same node twice keeps the latest version of the node. Since a node that represents the same thing may appear twice in a log (for example, the same process might appear in a process creation event and a file write event). It's easier to simply update the nodes as you iterate over the *nodes* attribute.

Parameters

- **metadata** (*dict, optional*) – The metadata from the datasource.
- **consolidate_edges** (*boolean, optional*) – Controls if edges are consolidated. That is, if the edge of type q from u to v happens N times, should there be one edge from u to v with type q, or should there be N edges.

Notes

Putting

add_nodes(*nodes: List[beagle.nodes.node.Node]*) → networkx.classes.multidigraph.MultiDiGraph
This function should allow (or raise an error if not possible to) a user to add additional nodes to an already existing graph.

Parameters **nodes** (*List[Node]*) – The new nodes to add to the graph.

static from_json(*path_or_obj: Union[str, dict]*) → networkx.classes.multidigraph.MultiDiGraph

graph() → networkx.classes.multidigraph.MultiDiGraph
Generates the MultiDiGraph.

Places the nodes in the Graph.

Returns The generated NetworkX object.

Return type nx.MultiDiGraph

classmethod graph_to_json(*graph: networkx.classes.multidigraph.MultiDiGraph*) → dict

insert_edges(*u: beagle.nodes.node.Node, v: beagle.nodes.node.Node, edge_name: str, instances: List[dict]*) → None
Inserts instances of an edge of type *edge_name* from node *u* to *v*

Parameters

- **u** (Node) – Source Node object

- **v** (Node) – Destination Node object
- **edge_name** (*str*) – Edge Name
- **instances** (*List[dict]*) – The data entries for the node between *u* and *v*.

insert_node(*node: beagle.nodes.node.Node*, *node_id: int*) → None
Inserts a node into the graph, as well as all edges outbound from it.

Parameters

- **node** (Node) – Node object to insert
- **node_id** (*int*) – The ID of the node (*hash(node)*)

is_empty() → bool
Returns true if there wasn't a graph created.

to_json() → dict
Convert the graph to JSON, which can later be used be read in using networkx:

```
>>> backend = NetworkX(nodes=nodes)
>>> G = backend.graph()
>>> data = G.to_json()
>>> parsed = networkx.readwrite.json_graph.node_link_graph(data)
```

Returns node_link compatible version of the graph.

Return type dict

update_node(*node: beagle.nodes.node.Node*, *node_id: int*) → None
Update the attributes of a node. Since we may see the same Node in multiple events, we want to have the largest coverage of its attributes. * See *beagle.nodes.node.Node* for how we determine two nodes are the same.

This method updates the node already in the graph with the newest attributes from the passed in parameter *Node*

Parameters

- **node** (Node) – The Node object to use to update the node already in the graph
- **node_id** (*int*) – The hash of the Node. see beagle.nodes.node.__hash__()

Notes

Since nodes are de-duplicated before being inserted into the graph, this should only be used to manually add in new data.

1.2 beagle.common package

1.2.1 Submodules

1.2.2 beagle.common.logging module

1.2.3 Module contents

beagle.common.**dedup_nodes**(*nodes: List[beagle.nodes.node.Node]*) → List[beagle.nodes.node.Node]

Deduplicates a list of nodes.

Parameters **nodes** (*List[Node]*) – [description]

Returns [description]

Return type List[*Node*]

beagle.common.**split_path**(*path: str*) → Tuple[str, str]

Parse a full file path into a file name + extension, and directory at once. For example:

```
>>> split_path('c:\ProgramData\app.exe')
(app.exe', 'c:\ProgramData')
```

By default, if it can't split, it'll return as the directory, and None as the image.

Parameters **path** (*str*) – The path to parse

Returns A tuple of file name + extension, and directory at once.

Return type Tuple[str, str]

beagle.common.**split_reg_path**(*reg_path: str*) → Tuple[str, str, str]

Splits a full registry path into hive, key, and path.

Examples

```
>>> split_reg_path(\REGISTRY\MACHINE\SYSTEM\ControlSet001\Control\ComputerName)
(REGISTRY, ComputerName, MACHINE\SYSTEM\ControlSet001\Control)
```

Parameters **regpath** (*str*) – The full registry key

Returns Hive, registry key, and registry key path

Return type Tuple[str, str, str]

1.3 beagle.datasources package

1.3.1 Subpackages

beagle.datasources.memory package

Submodules

beagle.datasources.memory.windows_rekall module

class beagle.datasources.memory.windows_rekall.**WindowsMemory**(*memory_image: str*)

Bases: *beagle.datasources.base_datasource.DataSource*

Yields events from a raw memory file by leveraging Rekall plugins.

This DataSource converts the outputs of the plugins to the schema provided by GenericTransformer.

Parameters **memory_image** (*str*) – File path to the memory image.

category = 'Windows Memory'

connscan() → Generator[[dict, None], None]

events() → Generator[[dict, None], None]
Generator which must yield each event as a dictionary from the datasource one by one, once the generator is exhausted, this signals the datasource is exhausted.

Returns Generator over all events from this datasource.

Return type Generator[dict, None, None]

handles() → Generator[[dict, None], None]
Converts the output of the rekall *handles* plugin to a series of events which represent accessing registry keys or file.

Yields *Generator[dict, None, None]* – One file or registry key access event a time.

metadata() → dict
Returns the metadata object for this data source.

Returns A metadata dictionary to store with the graph.

Return type dict

name = 'Windows Memory'

pslist() → Generator[[dict, None], None]
Converts the output of rekall's *pslist* plugin to a series of dictionaries that represent a process getting launched.

Returns Yields one process launch event

Return type Generator[dict, None, None]

transformers = [<class 'beagle.transformers.generic_transformer.GenericTransformer'>]

Module contents

beagle.datasources.virustotal package

Submodules

beagle.datasources.virustotal.generic_vt_sandbox module

class beagle.datasources.virustotal.generic_vt_sandbox.**GenericVTSandbox**(*behaviour_report_file: str*, *hash_metadata_file: str = None*)

Bases: *beagle.datasources.base_datasource.DataSource*

Converts a Virustotal V3 API behavior report to a Beagle graph.

This DataSource outputs data in the schema accepted by *GenericTransformer*.

Providing the hash's metadata JSON allows for proper creation of a metadata object. * This can be fetched from https://www.virustotal.com/api/v3/files/{id}

Behavior reports come from https://www.virustotal.com/api/v3/files/{id}/behaviours * Beagle generates one graph **per** report in the *attributes* array.

Where {id} is the sha256 of the file.

Parameters

- **behaviour_report** (*str*) – File containing A **single** behaviour report from one of the virustotal linked sandboxes.
- **hash_metadata** (*str*) – File containing the hashes metadata, containing its detections.

KNOWN_ATTRIBUTES = ['files_deleted', 'processes_tree', 'files_opened', 'files_written'

category = 'VT Sandbox'

events() → Generator[[dict, None], None]

Generator which must yield each event as a dictionary from the datasource one by one, once the generator is exhausted, this signals the datasource is exhausted.

Returns Generator over all events from this datasource.

Return type Generator[dict, None, None]

metadata() → dict

Generates the metadata based on the provided hash_metadata file.

Returns Name, number of malicious detections, AV results, and common_name from VT.

Return type dict

name = 'VirusTotal v3 API Sandbox Report Files'

transformers = [<class 'beagle.transformers.generic_transformer.GenericTransformer'>]

beagle.datasources.virustotal.generic_vt_sandbox_api module

class beagle.datasources.virustotal.generic_vt_sandbox_api.**GenericVTSandboxAPI**(*file_hash: str*, *sandbox_name: str = None*)

Bases: *beagle.datasources.base_datasource.ExternalDataSource*, *beagle.*

datasources.virustotal.generic_vt_sandbox.GenericVTSandbox

A class which provides an easy way to fetch VT v3 API sandbox data. This can be used to directly pull sandbox data from VT.

Parameters

- **file_hash** (*str*) – The hash of the file you want to graph.
- **sandbox_name** (*str, optional*) – The name of the sandbox you want to pull from VT (there may be multiple available). (the default is None, which picks the first one)

Raises RuntimeError – If there is not virustotal API key defined.

Examples

```
>>> datasource = GenericVTSandboxAPI(
    file_hash="ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa',
    sandbox_name="Dr.Web vxCube"
)
```

category = 'VT Sandbox'

name = 'VirusTotal v3 API Sandbox Report'

transformers = [<class 'beagle.transformers.generic_transformer.GenericTransformer'>]

Module contents

1.3.2 Submodules

1.3.3 beagle.datasources.base_datasource module

class beagle.datasources.base_datasource.**DataSource**

Bases: object

Base DataSource class. This class should be used to create DataSources which are file based.

For non-file based data sources (i.e performing a HTTP request to an API to get some data). The ExternalDataSource class should be subclassed.

Each datasource requires the following annotations be made:

1. name *string*: The name of the datasource, this should be human readable.
2. transformer *List[Transformer]*: The list of transformers which you can send events from this datasource to.
3. category *string*: The category this datasource outputs data to, this should be human readable.

Not supplying these three will not allow the class to get created, and will prevent beagle from loading.

Examples

```
>>> class MyDataSource(DataSource):
        name = "My Data Source"
        transformers = [GenericTransformer]
        category = "My Category"
```

events() → Generator[[dict, None], None]
Generator which must yield each event as a dictionary from the datasource one by one, once the generator is exhausted, this signals the datasource is exhausted.

Returns Generator over all events from this datasource.

Return type Generator[dict, None, None]

metadata() → dict
Returns the metadata object for this data source.

Returns A metadata dictionary to store with the graph.

Return type dict

to_graph(**args*, ***kwargs*) → Any
Allows to hop immediatly from a datasource to a graph.

Supports parameters for the to_graph() function of the transformer.

see **:py:method:'beagle.transformers.base_transformer.Transformer.to_graph'**

Examples

```
>>> SysmonEVTX('data/sysmon/autoruns-sysmon.evtx').to_graph(Graphistry,␣
↪render=True)
```

Returns Returns the outuput of the Backends *.graph()* function.

Return type Any

to_transformer(*transformer: Transformer = None*) → Transformer
Allows the data source to be used as a functional API. By default, uses the first transformer in the *transformers* attribute.

```
>>> graph = DataSource().to_transformer().to_graph()
```

Returns A instance of the transformer class yielded to.

Return type *Transformer*

class beagle.datasources.base_datasource.**ExternalDataSource**
Bases: *beagle.datasources.base_datasource.DataSource*

This class should be used when fetching data from exteranl sources before processing.

Using a different class allows the web interface to render a different upload page when a data source requiring text input in favor of a file input is used.

Examples

See *beagle.datasources.virustotal.generic_vt_sandbox_api.GenericVTSandboxAPI*

1.3.4 beagle.datasources.fireeye_ax_report module

class beagle.datasources.fireeye_ax_report.**FireEyeAXReport**(*ax_report: str*)
Bases: *beagle.datasources.base_datasource.DataSource*

Yields events one by one from a FireEyeAX Report and sends them to the generic transformer.

The JSON report should look something like this:

```
{
    "alert": [
        {
            "explanation": {
                "malwareDetected": {
                    ...
                },
                "cncServices": {
                    "cncService": [
                        ...
                },
                "osChanges": [
                    {
                        "process": [...],
                        "registry": [...],
                        ...
                }
            }
        }
    ]
}
```

Beagle looks at the *first alert* in the *alerts* array.

Parameters **ax_report** (*str*) – File path to the JSON AX Report, see class description for expected format.

category = 'FireEye AX'

events() → Generator[[dict, None], None]
Generator which must yield each event as a dictionary from the datasource one by one, once the generator is exhausted, this signals the datasource is exhausted.

Returns Generator over all events from this datasource.

Return type Generator[dict, None, None]

metadata() → dict
Returns the metadata object for this data source.

Returns A metadata dictionary to store with the graph.

Return type dict

name = 'FireEye AX Report'

transformers = [<class 'beagle.transformers.fireeye_ax_transformer.FireEyeAXTransforme

1.3.5 beagle.datasources.hx_triage module

class beagle.datasources.hx_triage.**HXTriage**(*triage: str*)
Bases: *beagle.datasources.base_datasource.DataSource*

A FireEye HX Triage DataSource.

Allows generation of graphs from the redline .mans files generated by FireEye HX.

Examples

```
>>> triage = HXTriage(file_path="/path/to/triage.mans")
```

category = 'FireEye HX'

events() → Generator[[dict, None], None]
Yields each event in the triage from the supported files.

metadata() → dict
Returns basic information about the triage.

1. Agent ID
2. Hostname
3. Platform (win, osx, linux)
4. Triggering Alert name (if exists)
5. Link to the controller the triage is from

Returns Metadata for the submitted HX Triage.

Return type dict

name = 'FireEye HX Triage'

parse_agent_events(*agent_events_file: str*) → Generator[[dict, None], None]
Generator over the agent events file. Converts each XML into a dictionary. Timestamps are converted to epoch time.

The below XML entry:

```
<eventItem uid="39265403">
    <timestamp>2018-06-27T21:15:32.678Z</timestamp>
    <eventType>dnsLookupEvent</eventType>
    <details>
    <detail>
        <name>hostname</name>
        <value>github.com</value>
    </detail>
    <detail>
        <name>pid</name>
        <value>12345</value>
    </detail>
    <detail>
        <name>process</name>
        <value>git.exe</value>
    </detail>
    <detail>
        <name>processPath</name>
        <value>c:\windows\</value>
    </detail>
    <detail>
        <name>username</name>
```

(continues on next page)

(continued from previous page)

```
        <value>Bob/Schmob</value>
    </detail>
    </details>
</eventItem>
```

becomes:

```
{
    "timestamp": 1530134132,
    "eventType": "dnsLookupEvent",
    "hostname": "github.com",
    "pid": "12345",
    "process": "git.exe",
    "processPath": "c:\windows\",
    "username": "Bob/Schmob",
}
```

Parameters **agent_events_file** (*str*) – The path to the file containing the agent events.

Returns Generator over agent events.

Return type Generator[dict, None, None]

parse_alert_files(*temp_dir: str*) → Generator[[dict, None], None]

Parses out the alert files from the hits.json and threats.json files

Parameters **temp_dir** (*str*) – Folder which contains the expanded triage.

Yields *Generator[dict, None, None]* – The next event found in the Triage.

transformers = [<class 'beagle.transformers.fireeye_hx_transformer.FireEyeHXTransforme

1.3.6 beagle.datasources.procmon_csv module

class beagle.datasources.procmon_csv.**ProcmonCSV**(*procmon_csv: str*)

Bases: *beagle.datasources.base_datasource.DataSource*

Reads events in one by one from a ProcMon CSV, and parses them into the GenericTransformer

category = 'Procmon'

events() → Generator[[dict, None], None]

Generator which must yield each event as a dictionary from the datasource one by one, once the generator is exhausted, this signals the datasource is exhausted.

Returns Generator over all events from this datasource.

Return type Generator[dict, None, None]

metadata() → dict

Returns the metadata object for this data source.

Returns A metadata dictionary to store with the graph.

Return type dict

name = 'Procmon CSV'

transformers = [<class 'beagle.transformers.procmon_transformer.ProcmonTransformer'>]

1.3.7 beagle.datasources.sysmon_evtx module

class beagle.datasources.sysmon_evtx.**SysmonEVTX**(*sysmon_evtx_log_file: str*)
Bases: *beagle.datasources.win_evtx.WinEVTX*

Parses SysmonEVTX files, see *beagle.datasources.win_evtx.WinEVTX*

category = 'SysMon'

metadata() → dict
Returns the Hostname by inspecting the *Computer* entry of the first record.

Returns

```
>>> {"hostname": str}
```

Return type dict

name = 'Sysmon EVTX File'

parse_record(*record: lxml.etree.ElementTree*, *name=''*) → dict
Parse a single record recursivly into a JSON file with a single level.

Parameters

- **record** (*etree.ElementTree*) – The current record.
- **name** (*str, optional*) – Last records name. (the default is "", which [default_description])

Returns dict representation of record.

Return type dict

transformers = [<class 'beagle.transformers.sysmon_transformer.SysmonTransformer'>]

1.3.8 beagle.datasources.win_evtx module

class beagle.datasources.win_evtx.**WinEVTX**(*evtx_log_file: str*)
Bases: *beagle.datasources.base_datasource.DataSource*

Parses Windows .evtx files. Yields events one by one using the *python-evtx* library.

Parameters **evtx_log_file** (*str*) – The path to the windows evtx file to parse.

category = 'Windows Event Logs'

events() → Generator[[dict, None], None]
Generator which must yield each event as a dictionary from the datasource one by one, once the generator is exhausted, this signals the datasource is exhausted.

Returns Generator over all events from this datasource.

Return type Generator[dict, None, None]

metadata() → dict
Get the hostname by inspecting the first record.

Returns

```
>>> {"hostname": str}
```

Return type dict

name = 'Windows EVTX File'

parse_record(*record: lxml.etree.ElementTree*, *name=''*) → dict
Recursivly converts a etree.ElementTree record to a JSON dictionary with one level.

Parameters

- **record** (*etree.ElementTree*) – Current record to parse
- **name** (*str, optional*) – Name of the current key we are at.

Returns JSON represntation of the event

Return type dict

transformers = [<class 'beagle.transformers.evtx_transformer.WinEVTXTransformer'>]

1.3.9 Module contents

class beagle.datasources.**DataSource**
Bases: object

Base DataSource class. This class should be used to create DataSources which are file based.

For non-file based data sources (i.e performing a HTTP request to an API to get some data). The ExternalDataSource class should be subclassed.

Each datasource requires the following annotations be made:

1. name *string*: The name of the datasource, this should be human readable.
2. transformer *List[Transformer]*: The list of transformers which you can send events from this datasource to.
3. category *string*: The category this datasource outputs data to, this should be human readable.

Not supplying these three will not allow the class to get created, and will prevent beagle from loading.

Examples

```
>>> class MyDataSource(DataSource):
        name = "My Data Source"
        transformers = [GenericTransformer]
        category = "My Category"
```

events() → Generator[[dict, None], None]
Generator which must yield each event as a dictionary from the datasource one by one, once the generator is exhausted, this signals the datasource is exhausted.

Returns Generator over all events from this datasource.

Return type Generator[dict, None, None]

metadata() → dict
Returns the metadata object for this data source.

Returns A metadata dictionary to store with the graph.

Return type dict

to_graph (**args*, ***kwargs*) → Any
Allows to hop immediatly from a datasource to a graph.

Supports parameters for the to_graph() function of the transformer.

see **:py:method:'beagle.transformers.base_transformer.Transformer.to_graph'**

Examples

```
>>> SysmonEVTX('data/sysmon/autoruns-sysmon.evtx').to_graph(Graphistry,␣
↪render=True)
```

Returns Returns the outuput of the Backends *.graph()* function.

Return type Any

to_transformer (*transformer: Transformer = None*) → Transformer
Allows the data source to be used as a functional API. By default, uses the first transformer in the *transformers* attribute.

```
>>> graph = DataSource().to_transformer().to_graph()
```

Returns A instance of the transformer class yielded to.

Return type *Transformer*

class beagle.datasources.**SplunkSPLSearch**(*spl: str*, *earliest: str = '-24h@h'*, *latest: str = 'now'*)
Bases: *beagle.datasources.base_datasource.ExternalDataSource*

Datasource which allows transforming the results of a Splunk search into a graph.

Parameters **spl** (*str*) – The splunk search to transform

Raises RuntimeError – If there are no Splunk credentials configured.

category = 'Splunk'

create_search (*query: str*, *query_kwargs: dict*)
Creates a splunk search with *query* and *query_kwargs* using *splunk_client*

Returns A splunk Job object.

Return type Job

events () → Generator[[dict, None], None]
Generator which must yield each event as a dictionary from the datasource one by one, once the generator is exhausted, this signals the datasource is exhausted.

Returns Generator over all events from this datasource.

Return type Generator[dict, None, None]

get_results (*job*, *count: int*) → list
Return events from a finished Job as an array of dictionaries.

Parameters **job** (*Job*) – Job object to pull results from.

Returns The results of the search.

Return type list

metadata() → dict
Returns the metadata object for this data source.

Returns A metadata dictionary to store with the graph.

Return type dict

name = 'Splunk SPL Search'

patch_spl(*spl: str*) → str
Ensures ‘search ‘ is the first command in the SPL.

setup_session()

transformers = [<class 'beagle.transformers.generic_transformer.GenericTransformer'>]

class beagle.datasources.**CuckooReport**(*cuckoo_report: str*)
Bases: *beagle.datasources.base_datasource.DataSource*

Yields events from a cuckoo sandbox report.

Cuckoo now provides a nice summary for each process under the “generic” summary tab:

```
{
    "behavior": {
        "generic": [
            {
                'process_path':
↪'C:\Users\Administrator\AppData\Local\Temp\It6QworVAgY.exe',
                'process_name': 'It6QworVAgY.exe',
                'pid': 2548,
                'ppid': 2460,
                'summary': {
                    "directory_created" : [...],
                    "dll_loaded" : [...],
                    "file_opened" : [...],
                    "regkey_opened" : [...],
                    "file_moved" : [...],
                    "file_deleted" : [...],
                    "file_exists" : [...],
                    "mutex" : [...],
                    "file_failed" : [...],
                    "guid" : [...],
                    "file_read" : [...],
                    "regkey_re" : [...]
                    ...
                },

            }
        ]
    }
}
```

Using this, we can crawl and extract out all activity for a specific process.

Notes

This is based on the output of the following reporting module: https://github.com/cuckoosandbox/cuckoo/blob/master/cuckoo/processing/platform/windows.py

Parameters **cuckoo_report** (*str*) – The file path to the cuckoo sandbox report.

category = 'Cuckoo Sandbox'

events() → Generator[[dict, None], None]
Generator which must yield each event as a dictionary from the datasource one by one, once the generator is exhausted, this signals the datasource is exhausted.

Returns Generator over all events from this datasource.

Return type Generator[dict, None, None]

global_network_events() → Generator[[dict, None], None]

identify_processes() → Dict[int, dict]
The *generic* tab contains an array of processes. We can iterate over it to quickly generate *Process* entries for later. After grabbing all processes, we can walk the "processtree" entry to update them with the command lines.

Returns

Return type None

metadata() → dict
Returns the metadata object for this data source.

Returns A metadata dictionary to store with the graph.

Return type dict

name = 'Cuckoo Sandbox Report'

process_tree() → Generator[[dict, None], None]

transformers = [<class 'beagle.transformers.generic_transformer.GenericTransformer'>]

class beagle.datasources.**FireEyeAXReport**(*ax_report: str*)
Bases: *beagle.datasources.base_datasource.DataSource*

Yields events one by one from a FireEyeAX Report and sends them to the generic transformer.

The JSON report should look something like this:

```
{
    "alert": [
        {
            "explanation": {
                "malwareDetected": {
                    ...
                },
                "cncServices": {
                    "cncService": [
                        ...
                },
                "osChanges": [
                    {
                        "process": [...],
                        "registry": [...],
                        ...
                }
            }
        }
    ]
}
```

Beagle looks at the *first alert* in the *alerts* array.

Parameters **ax_report** (*str*) – File path to the JSON AX Report, see class description for expected format.

category = 'FireEye AX'

events() → Generator[[dict, None], None]
Generator which must yield each event as a dictionary from the datasource one by one, once the generator is exhausted, this signals the datasource is exhausted.

Returns Generator over all events from this datasource.

Return type Generator[dict, None, None]

metadata() → dict
Returns the metadata object for this data source.

Returns A metadata dictionary to store with the graph.

Return type dict

name = 'FireEye AX Report'

transformers = [<class 'beagle.transformers.fireeye_ax_transformer.FireEyeAXTransforme

class beagle.datasources.**HXTriage**(*triage: str*)
Bases: *beagle.datasources.base_datasource.DataSource*

A FireEye HX Triage DataSource.

Allows generation of graphs from the redline .mans files generated by FireEye HX.

Examples

```
>>> triage = HXTriage(file_path="/path/to/triage.mans")
```

category = 'FireEye HX'

events() → Generator[[dict, None], None]
Yields each event in the triage from the supported files.

metadata() → dict
Returns basic information about the triage.

1. Agent ID
2. Hostname
3. Platform (win, osx, linux)
4. Triggering Alert name (if exists)
5. Link to the controller the triage is from

Returns Metadata for the submitted HX Triage.

Return type dict

name = 'FireEye HX Triage'

parse_agent_events(*agent_events_file: str*) → Generator[[dict, None], None]
Generator over the agent events file. Converts each XML into a dictionary. Timestamps are converted to epoch time.

The below XML entry:

```
<eventItem uid="39265403">
    <timestamp>2018-06-27T21:15:32.678Z</timestamp>
    <eventType>dnsLookupEvent</eventType>
    <details>
    <detail>
        <name>hostname</name>
        <value>github.com</value>
    </detail>
    <detail>
        <name>pid</name>
        <value>12345</value>
    </detail>
    <detail>
        <name>process</name>
        <value>git.exe</value>
    </detail>
    <detail>
        <name>processPath</name>
        <value>c:\windows\</value>
    </detail>
    <detail>
        <name>username</name>
        <value>Bob/Schmob</value>
    </detail>
    </details>
</eventItem>
```

becomes:

```
{
    "timestamp": 1530134132,
    "eventType": "dnsLookupEvent",
    "hostname": "github.com",
    "pid": "12345",
    "process": "git.exe",
    "processPath": "c:\windows\",
    "username": "Bob/Schmob",
}
```

Parameters **agent_events_file** (*str*) – The path to the file containing the agent events.

Returns Generator over agent events.

Return type Generator[dict, None, None]

parse_alert_files(*temp_dir: str*) → Generator[[dict, None], None]

Parses out the alert files from the hits.json and threats.json files

Parameters **temp_dir** (*str*) – Folder which contains the expanded triage.

Yields *Generator[dict, None, None]* – The next event found in the Triage.

transformers = [<class 'beagle.transformers.fireeye_hx_transformer.FireEyeHXTransforme

class beagle.datasources.**WindowsMemory**(*memory_image: str*)

Bases: *beagle.datasources.base_datasource.DataSource*

Yields events from a raw memory file by leveraging Rekall plugins.

This DataSource converts the outputs of the plugins to the schema provided by GenericTransformer.

Parameters **memory_image** (*str*) – File path to the memory image.

category = 'Windows Memory'

connscan() → Generator[[dict, None], None]

events() → Generator[[dict, None], None]
Generator which must yield each event as a dictionary from the datasource one by one, once the generator is exhausted, this signals the datasource is exhausted.

Returns Generator over all events from this datasource.

Return type Generator[dict, None, None]

handles() → Generator[[dict, None], None]
Converts the output of the rekall *handles* plugin to a series of events which represent accessing registry keys or file.

Yields *Generator[dict, None, None]* – One file or registry key access event a time.

metadata() → dict
Returns the metadata object for this data source.

Returns A metadata dictionary to store with the graph.

Return type dict

name = 'Windows Memory'

pslist() → Generator[[dict, None], None]
Converts the output of rekall's *pslist* plugin to a series of dictionaries that represent a process getting launched.

Returns Yields one process launch event

Return type Generator[dict, None, None]

transformers = [<class 'beagle.transformers.generic_transformer.GenericTransformer'>]

class beagle.datasources.**ProcmonCSV**(*procmon_csv: str*)
Bases: *beagle.datasources.base_datasource.DataSource*

Reads events in one by one from a ProcMon CSV, and parses them into the GenericTransformer

category = 'Procmon'

events() → Generator[[dict, None], None]
Generator which must yield each event as a dictionary from the datasource one by one, once the generator is exhausted, this signals the datasource is exhausted.

Returns Generator over all events from this datasource.

Return type Generator[dict, None, None]

metadata() → dict
Returns the metadata object for this data source.

Returns A metadata dictionary to store with the graph.

Return type dict

name = 'Procmon CSV'

transformers = [<class 'beagle.transformers.procmon_transformer.ProcmonTransformer'>]

class beagle.datasources.**SysmonEVTX**(*sysmon_evtx_log_file: str*)
Bases: *beagle.datasources.win_evtx.WinEVTX*

Parses SysmonEVTX files, see *beagle.datasources.win_evtx.WinEVTX*

category = 'SysMon'

metadata() → dict
Returns the Hostname by inspecting the *Computer* entry of the first record.

Returns

```
>>> {"hostname": str}
```

Return type dict

name = 'Sysmon EVTX File'

parse_record(*record: lxml.etree.ElementTree*, *name=''*) → dict
Parse a single record recursivly into a JSON file with a single level.

Parameters

- **record** (*etree.ElementTree*) – The current record.
- **name** (*str, optional*) – Last records name. (the default is "", which [default_description])

Returns dict representation of record.

Return type dict

transformers = [<class 'beagle.transformers.sysmon_transformer.SysmonTransformer'>]

class beagle.datasources.**PCAP**(*pcap_file: str*)
Bases: *beagle.datasources.base_datasource.DataSource*

Yields events from a PCAP file.

Parameters **pcap_file** (*str*) – path to a PCAP file.

category = 'PCAP'

events() → Generator[[dict, None], None]
Generator which must yield each event as a dictionary from the datasource one by one, once the generator is exhausted, this signals the datasource is exhausted.

Returns Generator over all events from this datasource.

Return type Generator[dict, None, None]

metadata() → dict
Returns the metadata object for this data source.

Returns A metadata dictionary to store with the graph.

Return type dict

name = 'PCAP File'

transformers = [<class 'beagle.transformers.pcap_transformer.PCAPTransformer'>]

class beagle.datasources.**GenericVTSandbox**(*behaviour_report_file: str*, *hash_metadata_file: str = None*)
Bases: *beagle.datasources.base_datasource.DataSource*

Converts a Virustotal V3 API behavior report to a Beagle graph.

This DataSource outputs data in the schema accepted by *GenericTransformer*.

Providing the hash's metadata JSON allows for proper creation of a metadata object. * This can be fetched from https://www.virustotal.com/api/v3/files/{id}

Behavior reports come from https://www.virustotal.com/api/v3/files/{id}/behaviours * Beagle generates one graph **per** report in the *attributes* array.

Where {id} is the sha256 of the file.

Parameters

- **behaviour_report** (*str*) – File containing A **single** behaviour report from one of the virustotal linked sandboxes.
- **hash_metadata** (*str*) – File containing the hashes metadata, containing its detections.

KNOWN_ATTRIBUTES = ['files_deleted', 'processes_tree', 'files_opened', 'files_written'

category = 'VT Sandbox'

events() → Generator[[dict, None], None]

Generator which must yield each event as a dictionary from the datasource one by one, once the generator is exhausted, this signals the datasource is exhausted.

Returns Generator over all events from this datasource.

Return type Generator[dict, None, None]

metadata() → dict

Generates the metadata based on the provided hash_metadata file.

Returns Name, number of malicious detections, AV results, and common_name from VT.

Return type dict

name = 'VirusTotal v3 API Sandbox Report Files'

transformers = [<class 'beagle.transformers.generic_transformer.GenericTransformer'>]

class beagle.datasources.**GenericVTSandboxAPI**(*file_hash: str*, *sandbox_name: str = None*)

Bases: *beagle.datasources.base_datasource.ExternalDataSource*, *beagle.datasources.virustotal.generic_vt_sandbox.GenericVTSandbox*

A class which provides an easy way to fetch VT v3 API sandbox data. This can be used to directly pull sandbox data from VT.

Parameters

- **file_hash** (*str*) – The hash of the file you want to graph.
- **sandbox_name** (*str, optional*) – The name of the sandbox you want to pull from VT (there may be multiple available). (the default is None, which picks the first one)

Raises RuntimeError – If there is not virustotal API key defined.

Examples

```
>>> datasource = GenericVTSandboxAPI(
    file_hash="ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa',
    sandbox_name="Dr.Web vxCube"
)
```

category = 'VT Sandbox'

name = 'VirusTotal v3 API Sandbox Report'

transformers = [<class 'beagle.transformers.generic_transformer.GenericTransformer'>]

class beagle.datasources.**WinEVTX**(*evtx_log_file: str*)
Bases: *beagle.datasources.base_datasource.DataSource*

Parses Windows .evtx files. Yields events one by one using the *python-evtx* library.

Parameters **evtx_log_file** (*str*) – The path to the windows evtx file to parse.

category = 'Windows Event Logs'

events() → Generator[[dict, None], None]
Generator which must yield each event as a dictionary from the datasource one by one, once the generator is exhausted, this signals the datasource is exhausted.

Returns Generator over all events from this datasource.

Return type Generator[dict, None, None]

metadata() → dict
Get the hostname by inspecting the first record.

Returns

```
>>> {"hostname": str}
```

Return type dict

name = 'Windows EVTX File'

parse_record(*record: lxml.etree.ElementTree*, *name=''*) → dict
Recursivly converts a etree.ElementTree record to a JSON dictionary with one level.

Parameters

- **record** (*etree.ElementTree*) – Current record to parse
- **name** (*str, optional*) – Name of the current key we are at.

Returns JSON represntation of the event

Return type dict

transformers = [<class 'beagle.transformers.evtx_transformer.WinEVTXTransformer'>]

class beagle.datasources.**DARPATCJson**(*file_path: str*)
Bases: beagle.datasources.json_data.JSONFile

category = 'Darpa TC3'

events() → Generator[[dict, None], None]
Events are in the format:

"datum": {

"com.bbn.tc.schema.avro.cdm18.Subject": { ...

}

This pops out the relevant info under the first key.

name = 'Darpa TC3 JSON'

transformers = [<class 'beagle.transformers.darpa_tc_transformer.DRAPATCTransformer'>]

class beagle.datasources.**ElasticSearchQSSerach**(*index: str = 'logs-*', query: str = '*', earliest: str = '-7d', latest: str = 'now'*)

Bases: *beagle.datasources.base_datasource.ExternalDataSource*

Datasource which allows transforming the results of a Elasticsearch Query String search into a graph.

Parameters

- **index** (*str*) – Elasticsearch index, by default "logs-*"
- **query** (*str*) – Elasticsearch query string, by default "*"
- **earilest** (*str, optional*) – The earliest time modifier, by default "-7d"
- **latest** (*str, optional*) – The latest time modifier, by default "now"

Raises RuntimeError – If there are no Elasticsearch credentials configured.

category = 'Elasticsearch'

events() → Generator[[dict, None], None]

Generator which must yield each event as a dictionary from the datasource one by one, once the generator is exhausted, this signals the datasource is exhausted.

Returns Generator over all events from this datasource.

Return type Generator[dict, None, None]

metadata() → dict

Returns the metadata object for this data source.

Returns A metadata dictionary to store with the graph.

Return type dict

name = 'Elasticsearch Query String'

transformers = [<class 'beagle.transformers.generic_transformer.GenericTransformer'>]

1.4 beagle.nodes package

1.4.1 Submodules

1.4.2 beagle.nodes.alert module

class beagle.nodes.alert.**Alert**(*alert_name: str = None, alert_data: str = None*)

Bases: *beagle.nodes.node.Node*

edges

Returns an empty list, so that all nodes can have their edges iterated on, even if they have no outgoing edges.

Returns []

Return type List

key_fields = ['alert_name', 'alert_data']

1.4.3 beagle.nodes.domain module

class beagle.nodes.domain.**Domain**(*domain: str = None*)
Bases: *beagle.nodes.node.Node*

edges
Returns an empty list, so that all nodes can have their edges iterated on, even if they have no outgoing edges.

Returns []

Return type List

key_fields = ['domain']

class beagle.nodes.domain.**URI**(*uri: str = None*)
Bases: *beagle.nodes.node.Node*

edges
Returns an empty list, so that all nodes can have their edges iterated on, even if they have no outgoing edges.

Returns []

Return type List

key_fields = ['uri']

uri_of = {}

1.4.4 beagle.nodes.edge module

1.4.5 beagle.nodes.file module

class beagle.nodes.file.**File**(*host: str = None*, *file_path: str = None*, *file_name: str = None*, *full_path: str = None*, *extension: str = None*, *hashes: Optional[Dict[str, str]] = {}*)
Bases: *beagle.nodes.node.Node*

edges
Returns an empty list, so that all nodes can have their edges iterated on, even if they have no outgoing edges.

Returns []

Return type List

hashes = {}

key_fields = ['host', 'full_path']

set_extension() → None

1.4.6 beagle.nodes.ip_address module

class beagle.nodes.ip_address.**IPAddress**(*ip_address: str = None*, *mac: str = None*)
Bases: *beagle.nodes.node.Node*

key_fields = ['ip_address']

1.4.7 beagle.nodes.node module

class beagle.nodes.node.**Node**
Bases: object

Base Node class. Provides an interface which each Node must implement

edges
Returns an empty list, so that all nodes can have their edges iterated on, even if they have no outgoing edges.

Returns []

Return type List

key_fields = []

merge_with(*node: beagle.nodes.node.Node*) → None
Merge the current node with the destination node. After a call to *merge_with* the calling node will be updated with the information from the passed in node. This is similar to a dict *update* call.

Parameters **node** (Node) – The node to use to update the current node.

Raises TypeError – Passed in node does not represent the same entity represented by the current node.

to_dict() → Dict[str, Any]
Converts a Node object to a dictionary without its edge objects.

Returns A dict representation of a node.

Return type dict

Examples

Sample node:

```
class AnnotatedNode(Node):
    x: str
    y: int
    key_fields: List[str] = ["x", "y"]
    foo = defaultdict(str)

    def __init__(self, x: str, y: int):
        self.x = x
        self.y = y

    @property
    def _display(self) -> str:
        return self.x
```

```
>>> AnnotatedNode("1", 1).to_dict()
{"x": "1", "y": 1}
```

1.4.8 beagle.nodes.process module

class beagle.nodes.process.**Process**(*host: str = None*, *process_id: int = None*, *user: str = None*, *process_image: str = None*, *process_image_path: str = None*, *process_path: str = None*, *command_line: str = None*, *hashes: Dict[str, str] = {}*)

Bases: *beagle.nodes.node.Node*

edges

Returns an empty list, so that all nodes can have their edges iterated on, even if they have no outgoing edges.

Returns []

Return type List

get_file_node() → beagle.nodes.file.File

hashes = {}

key_fields = ['host', 'process_id', 'process_image']

class beagle.nodes.process.**SysMonProc**(*process_guid: str = None*, **args*, ***kwargs*)

Bases: *beagle.nodes.process.Process*

A custom Process class which extends the regular one. Adds the unique Sysmon process_guid identifier.

key_fields = ['process_guid']

1.4.9 beagle.nodes.registry module

class beagle.nodes.registry.**RegistryKey**(*host: str = None*, *hive: str = None*, *key_path: str = None*, *key: str = None*, *value: str = None*, *value_type: str = None*)

Bases: *beagle.nodes.node.Node*

key_fields = ['hive', 'key_path', 'key']

1.4.10 Module contents

class beagle.nodes.**Node**

Bases: object

Base Node class. Provides an interface which each Node must implement

edges

Returns an empty list, so that all nodes can have their edges iterated on, even if they have no outgoing edges.

Returns []

Return type List

key_fields = []

merge_with(*node: beagle.nodes.node.Node*) → None

Merge the current node with the destination node. After a call to *merge_with* the calling node will be updated with the information from the passed in node. This is similar to a dict *update* call.

Parameters **node** (Node) – The node to use to update the current node.

Raises `TypeError` – Passed in node does not represent the same entity represented by the current node.

to_dict() → Dict[str, Any]
Converts a Node object to a dictionary without its edge objects.

Returns A dict representation of a node.

Return type dict

Examples

Sample node:

```
class AnnotatedNode(Node):
    x: str
    y: int
    key_fields: List[str] = ["x", "y"]
    foo = defaultdict(str)

    def __init__(self, x: str, y: int):
        self.x = x
        self.y = y

    @property
    def _display(self) -> str:
        return self.x
```

```
>>> AnnotatedNode("1", 1).to_dict()
{"x": "1", "y": 1}
```

class beagle.nodes.**URI**(*uri: str = None*)
Bases: *beagle.nodes.node.Node*

edges
Returns an empty list, so that all nodes can have their edges iterated on, even if they have no outgoing edges.

Returns []

Return type List

key_fields = ['uri']

uri_of = {}

class beagle.nodes.**Domain**(*domain: str = None*)
Bases: *beagle.nodes.node.Node*

edges
Returns an empty list, so that all nodes can have their edges iterated on, even if they have no outgoing edges.

Returns []

Return type List

key_fields = ['domain']

class beagle.nodes.**File**(*host: str = None*, *file_path: str = None*, *file_name: str = None*, *full_path: str = None*, *extension: str = None*, *hashes: Optional[Dict[str, str]] = {}*)
Bases: *beagle.nodes.node.Node*

edges
Returns an empty list, so that all nodes can have their edges iterated on, even if they have no outgoing edges.

Returns []

Return type List

hashes = {}

key_fields = ['host', 'full_path']

set_extension() → None

class beagle.nodes.**FileOf**
Bases: beagle.edges.edge.Edge

class beagle.nodes.**IPAddress**(*ip_address: str = None*, *mac: str = None*)
Bases: *beagle.nodes.node.Node*

key_fields = ['ip_address']

class beagle.nodes.**SysMonProc**(*process_guid: str = None*, **args*, ***kwargs*)
Bases: *beagle.nodes.process.Process*

A custom Process class which extends the regular one. Adds the unique Sysmon process_guid identifier.

key_fields = ['process_guid']

class beagle.nodes.**Process**(*host: str = None*, *process_id: int = None*, *user: str = None*, *process_image: str = None*, *process_image_path: str = None*, *process_path: str = None*, *command_line: str = None*, *hashes: Dict[str, str] = {}*)
Bases: *beagle.nodes.node.Node*

edges
Returns an empty list, so that all nodes can have their edges iterated on, even if they have no outgoing edges.

Returns []

Return type List

get_file_node() → beagle.nodes.file.File

hashes = {}

key_fields = ['host', 'process_id', 'process_image']

class beagle.nodes.**RegistryKey**(*host: str = None*, *hive: str = None*, *key_path: str = None*, *key: str = None*, *value: str = None*, *value_type: str = None*)
Bases: *beagle.nodes.node.Node*

key_fields = ['hive', 'key_path', 'key']

class beagle.nodes.**Alert**(*alert_name: str = None*, *alert_data: str = None*)
Bases: *beagle.nodes.node.Node*

edges
Returns an empty list, so that all nodes can have their edges iterated on, even if they have no outgoing edges.

Returns []

Return type List

key_fields = ['alert_name', 'alert_data']

1.5 beagle.transformers package

1.5.1 Submodules

1.5.2 beagle.transformers.base_transformer module

class beagle.transformers.base_transformer.**Transformer**(*datasource: beagle.datasources.base_datasource.DataSource*)

Bases: object

Base Transformer class. This class implements a producer/consumer queue from the datasource to the *transform()* method. Producing the list of nodes is done via *run()*

Parameters **datasource** (DataSource) – The *DataSource* to get events from.

run() → List[beagle.nodes.node.Node]

Generates the list of nodes from the datasource.

This methods kicks off a producer/consumer queue. The producer grabs events one by one from the datasource by iterating over the events from the *events* generator. Each event is then sent to the transformer() function to be transformer into one or more *Node* objects.

Returns All Nodes created from the data source.

Return type List[*Node*]

to_graph(*backend: Backend = <class 'beagle.backends.networkx.NetworkX'>*, **args*, ***kwargs*) → Any

Graphs the nodes created by *run()*. If no backend is specific, the default used is NetworkX.

Parameters **backend** (*[type], optional*) – [description] (the default is NetworkX, which [default_description])

Returns [description]

Return type [type]

transform(*event: dict*) → Optional[Iterable[beagle.nodes.node.Node]]

1.5.3 beagle.transformers.evtx_transformer module

class beagle.transformers.evtx_transformer.**WinEVTXTransformer**(**args*, ***kwargs*)

Bases: *beagle.transformers.base_transformer.Transformer*

name = 'Win EVTX'

process_creation(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.process.Process]

Transformers a process creation (event ID 4688) into a set of nodes.

https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventID=4688

Parameters **event** (*dict*) – [description]

Returns [description]

Return type Optional[Tuple[*Process*, *File*, *Process*, *File*]]

transform()

1.5.4 beagle.transformers.fireeye_ax_transformer module

class beagle.transformers.fireeye_ax_transformer.**FireEyeAXTransformer**(*datasource: beagle.datasources.base_datasou*

Bases: *beagle.transformers.base_transformer.Transformer*

conn_events(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.ip_address.IPAddress]

Transforms a single connection event

Example event:

```
{
    "mode": "connect",
    "protocol_type": "tcp",
    "ipaddress": "199.168.199.123",
    "destination_port": 3333,
    "processinfo": {
        "imagepath": "C:\ProgramData\bloop\some_proc.exe",
        "tainted": true,
        "md5sum": "....",
        "pid": 3020
    },
    "timestamp": 27648
}
```

Parameters **event** (*dict*) – source dns_query event

Returns Process and its image, and the destination address

Return type Tuple[*Process*, *File*, *IPAddress*]

dns_events(*event: dict*) → Union[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.domain.Domain], Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.domain.Domain, beagle.nodes.ip_address.IPAddress]]

Transforms a single DNS event

Example event:

```
{
    "mode": "dns_query",
    "protocol_type": "udp",
    "hostname": "foobar",
    "qtype": "Host Address",
    "processinfo": {
        "imagepath": "C:\ProgramData\bloop\some_proc.exe",
        "tainted": true,
        "md5sum": "....",
        "pid": 3020
    },
    "timestamp": 27648
}
```

Optionally, if the event is "dns_query_answer", we can also extract the response.

Parameters **event** (*dict*) – source dns_query event

Returns Process and its image, and the domain looked up

Return type Tuple[*Process*, *File*, *Domain*]

file_events(*event: dict*) → Union[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.file.File], Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.file.File, beagle.nodes.file.File]]

Transforms a file event

Example file event:

```
{
    "mode": "created",
    "fid": { "ads": "", "content": 2533274790555891 },
    "processinfo": {
        "imagepath": "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.
↪exe",
        "md5sum": "eb32c070e658937aa9fa9f3ae629b2b8",
        "pid": 2956
    },
    "ntstatus": "0x0",
    "value": "C:\Users\admin\AppData\Local\Temp\sy24ttkc.k25.ps1",
    "CreateOptions": "0x400064",
    "timestamp": 9494
}
```

In 8.2.0 the *value* field became a dictionary when the mode is *failed*:

```
"values": {
    "value": "C:\Users\admin\AppData\Local\Temp\sy24ttkc.k25.ps1""
}
```

Parameters **event** (*dict*) – The source event

Returns The process, the process' image, and the file written.

Return type Tuple[*Process*, *File*, *File*]

http_requests(*event: dict*) → Union[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.ip_address.IPAddress, beagle.nodes.domain.URI, beagle.nodes.domain.Domain], Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.ip_address.IPAddress, beagle.nodes.domain.URI], Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.ip_address.IPAddress]]

Transforms a single *http_request* network event. A typical event looks like:

```
{
    "mode": "http_request",
    "protocol_type": "tcp",
    "ipaddress": "199.168.199.1",
    "destination_port": 80,
    "processinfo": {
        "imagepath": "c:\Windows\System32\svchost.exe",
        "tainted": false,
        "md5sum": "1234",
        "pid": 1292
    },
    "http_request": "GET /some_route.crl HTTP/1.1~~Cache-Control: max-age =␣
↪900~~User-Agent: Microsoft-CryptoAPI/10.0~~Host: crl.microsoft.com~~~~",
    "timestamp": 433750
}
```

Parameters **event** (*dict*) – The source *network* event with mode *http_request*

Returns [description]

Return type Tuple[*Node*]

name = 'FireEye AX'

process_events(*event: dict*) → Optional[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.process.Process, beagle.nodes.file.File]]

Transformers events from the *process* entry.

A single process entry looks like:

```
{
    "mode": string,
    "fid": dict,
    "parentname": string,
    "cmdline": string,
    "sha1sum": "string,
    "md5sum": string,
    "sha256sum": string,
    "pid": int,
    "filesize": int,
    "value": string,
    "timestamp": int,
    "ppid": int
},
```

Parameters **event** (*dict*) – The input event.

Returns Parent and child processes, and the file nodes that represent their binaries.

Return type Optional[Tuple[*Process*, *File*, *Process*, *File*]]

regkey_events(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.registry.RegistryKey]

Transforms a single registry key event

Example event:

```
{
    "mode": "queryvalue",
    "processinfo": {
        "imagepath": "C:\Users\admin\AppData\Local\Temp\bar.exe",
        "tainted": True,
        "md5sum": "....",
        "pid": 1700,
    },
    "value":
↪"\REGISTRY\USER\Software\Microsoft\Windows\CurrentVersion\Internet Settings\
↪"ProxyOverride"",
    "timestamp": 6203
},
```

Parameters **event** (*dict*) – source regkey event

Returns Process and its image, and the registry key.

Return type Tuple[*Process*, *File*, RegistrKey]

transform()

Transformers the various events from the AX Report class.

The only edge case is the network type, AX has multiple Nodes under one type when it comes to the network type. For example the following is a DNS event:

```
{
    "mode": "dns_query",
    "protocol_type": "udp",
    "hostname": "foobar",
    "qtype": "Host Address",
    "processinfo": {
        "imagepath": "C:\ProgramData\bloop\some_proc.exe",
        "tainted": true,
        "md5sum": "....",
        "pid": 3020
    },
    "timestamp": 27648
}
```

While the following is a TCP connection:

```
{
    "mode": "connect",
    "protocol_type": "tcp",
    "ipaddress": "192.168.199.123",
    "destination_port": 3333,
    "processinfo": {
        "imagepath": "C:\ProgramData\bloop\some_proc.exe",
        "tainted": true,
        "md5sum": "...",
        "pid": 3020
    },
    "timestamp": 28029
}
```

Both have the "network" event_type when coming from `FireEyeAXReport`

Parameters **event** (*dict*) – The current event to transform.

Returns Tuple of nodes extracted from the event.

Return type Optional[Tuple]

1.5.5 beagle.transformers.fireeye_hx_transformer module

class beagle.transformers.fireeye_hx_transformer.**FireEyeHXTransformer**(**args*, ***kwargs*)

Bases: *beagle.transformers.base_transformer.Transformer*

make_alert(*event: dict*) → Optional[Tuple[beagle.nodes.alert.Alert, ...]]

make_dnslookup(*event: dict*) → Optional[Tuple[beagle.nodes.domain.Domain, beagle.nodes.process.Process, beagle.nodes.file.File]]

Converts a dnsLookupEvent into a Domain, Process, and Process's File node.

Nodes: 1. Domain looked up.

2. Process performing the lookup.
3. File the Process was launched from.

Edges:

1. Process - (DNS Lookup For) -> Domain.
2. File - (FileOf) -> Process.

Parameters **event** (*dict*) – A dnsLookupEvent

Returns The Domain, Process, and File nodes.

Return type Optional[Tuple[*Domain*, *Process*, *File*]]

make_file(*event: dict*) → Optional[Tuple[beagle.nodes.file.File, beagle.nodes.process.Process, beagle.nodes.file.File]]

Converts a fileWriteEvent to two nodes, a file and the process manipulated the file. Generates a process - (Wrote) -> File edge.

Parameters **event** (*dict*) – The fileWriteEvent event.

Returns Returns a tuple contaning the File that this event is focused on, and the process which manipulated the file. The process has a Wrote edge to the file. Also contains the file that the process belongs to.

Return type Optional[Tuple[*File*, *Process*, *File*]]

make_imageload(*event: dict*) → Optional[Tuple[beagle.nodes.file.File, beagle.nodes.process.Process, beagle.nodes.file.File]]

make_network(*event: dict*) → Optional[Tuple[beagle.nodes.ip_address.IPAddress, beagle.nodes.process.Process, beagle.nodes.file.File]]

Converts a network connection event into a Process, File and IP Address node.

Nodes:

1. IP Address communicated to.
2. Process contacting IP.
3. File process launched from.

Edges:

1. Process - (Connected To) -> IP Address
2. File - (File Of) -> Process

Parameters **event** (*dict*) – The ipv4NetworkEvent

Returns The IP Address, Process, and Process's File object.

Return type Optional[Tuple[*IPAddress*, *Process*, *File*]]

make_process(*event: dict*) → Union[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File], Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.process.Process, beagle.nodes.file.File], None]

Converts a processEvent into either one Process node, or two Process nodes with a parent - (Launched) -> child relationship. Additionally, creats File nodes for the images of both of the Processe's identified.

Parameters **event** (*dict*) – The processEvent event

Returns Returns either a single process node, or a (parent, child) tuple where the parent has a launched edge to the child.

Return type Optional[Union[Tuple[*Process*, *File*], Tuple[*Process*, *File*, *Process*, *File*]]]

make_registry(*event: dict*) → Optional[Tuple[beagle.nodes.registry.RegistryKey, beagle.nodes.process.Process, beagle.nodes.file.File]]

make_url(*event: dict*) → Optional[Tuple[beagle.nodes.domain.URI, beagle.nodes.domain.Domain, beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.ip_address.IPAddress]]

Converts a URL access event and returns 5 nodes with 4 different relationships.

Nodes created:

1. URI Accessed (e.g /foobar)
2. Domain Accessed (e.g omer.com)
3. Process performing URL request.
4. File object for the Process image.
5. IP Address the domain resolves to.

Relationships created:

1. URI - (URI Of) -> Domain
2. Domain - (Resolves To) -> IP Address
3. Process - (*http method of event*) -> URI
4. Process - (Connected To) -> IP Address
5. File - (File Of) -> Process

Parameters **event** (*dict*) – The urlMonitorEvent events

Returns 5 tuple of the nodes pulled out of the event (see function description).

Return type Optional[Tuple[*URI*, *Domain*, *Process*, *File*, *IPAddress*]]

name = 'FireEye HX'

transform(*event: dict*) → Optional[Tuple[beagle.nodes.node.Node, ...]]

Sends each event from the FireEye HX Triage to the appropriate node creation function.

Parameters **event** (*dict*) – The source event from the HX Triage

Returns The results of the transforming function

Return type Optional[Tuple[*Node*, ..]]

1.5.6 beagle.transformers.generic_transformer module

class beagle.transformers.generic_transformer.**GenericTransformer**(**args*, ***kwargs*)

Bases: *beagle.transformers.base_transformer.Transformer*

This transformer will properly create graphs for any datasource that outputs data in the pre-defined schema.

make_alert(*event: dict*) → Tuple[beagle.nodes.alert.Alert, ...]

make_basic_file(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.file.File]

Transforms a file based event.

Support events:

1. EventTypes.FILE_DELETED

2. EventTypes.FILE_OPENED
3. EventTypes.FILE_WRITTEN
4. EventTypes.LOADED_MODULE

Parameters **event** (*dict*) – [description]

Returns [description]

Return type Tuple[*Process*, *File*, *File*]

make_basic_regkey(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.registry.RegistryKey]

make_connection(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.ip_address.IPAddress]

make_dnslookup(*event: dict*) → Union[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.domain.Domain, beagle.nodes.ip_address.IPAddress], Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.domain.Domain]]

make_file_copy(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.file.File, beagle.nodes.file.File]

make_http_req(*event: dict*) → Union[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.domain.URI, beagle.nodes.domain.Domain], Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.domain.URI, beagle.nodes.domain.Domain, beagle.nodes.ip_address.IPAddress]]

make_process(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.process.Process, beagle.nodes.file.File]

Accepts a process with the *EventTypes.PROCESS_LAUNCHED* event_type.

For example:

```
{
    FieldNames.PARENT_PROCESS_IMAGE: "cmd.exe",
    FieldNames.PARENT_PROCESS_IMAGE_PATH: "\",
    FieldNames.PARENT_PROCESS_ID: "2568",
    FieldNames.PARENT_COMMAND_LINE: '/K name.exe"',
    FieldNames.PROCESS_IMAGE: "find.exe",
    FieldNames.PROCESS_IMAGE_PATH: "\",
    FieldNames.COMMAND_LINE: 'find /i "svhost.exe"',
    FieldNames.PROCESS_ID: "3144",
    FieldNames.EVENT_TYPE: EventTypes.PROCESS_LAUNCHED,
}
```

Parameters **event** (*dict*) – [description]

Returns [description]

Return type Tuple[*Process*, *File*, *Process*, *File*]

make_regkey_set_value(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.registry.RegistryKey]

name = 'Generic'

transform()

1.5.7 beagle.transformers.procmon_transformer module

class beagle.transformers.procmon_transformer.**ProcmonTransformer**(*datasource: beagle.datasources.base_datasource.Data*

Bases: *beagle.transformers.base_transformer.Transformer*

access_file(*event*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File]

access_reg_key(*event*) → Tuple[beagle.nodes.process.Process, beagle.nodes.registry.RegistryKey]

connection(*event*) → Tuple[beagle.nodes.process.Process, beagle.nodes.ip_address.IPAddress]

name = 'Procmon'

process_create(*event*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.process.Process]

transform()

write_file(*event*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File]

1.5.8 beagle.transformers.sysmon_transformer module

class beagle.transformers.sysmon_transformer.**SysmonTransformer**(**args*, ***kwargs*)

Bases: *beagle.transformers.base_transformer.Transformer*

dns_lookup(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.domain.Domain]

file_created(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.file.File]

name = 'Sysmon'

network_connection(*event: dict*) → Union[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.ip_address.IPAddress], Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.ip_address.IPAddress, beagle.nodes.domain.Domain]]

process_creation(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.process.Process, beagle.nodes.file.File]

registry_creation(*event: dict*) → Optional[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.registry.RegistryKey]]

transform()

1.5.9 Module contents

class beagle.transformers.**Transformer**(*datasource: beagle.datasources.base_datasource.DataSource*)

Bases: object

Base Transformer class. This class implements a producer/consumer queue from the datasource to the *transform()* method. Producing the list of nodes is done via *run()*

Parameters **datasource** (DataSource) – The *DataSource* to get events from.

run() → List[beagle.nodes.node.Node]

Generates the list of nodes from the datasource.

This methods kicks off a producer/consumer queue. The producer grabs events one by one from the datasource by iterating over the events from the *events* generator. Each event is then sent to the `transformer()` function to be transformer into one or more *Node* objects.

Returns All Nodes created from the data source.

Return type List[*Node*]

to_graph(*backend: Backend = <class 'beagle.backends.networkx.NetworkX'>*, **args*, ***kwargs*) → Any

Graphs the nodes created by *run()*. If no backend is specific, the default used is NetworkX.

Parameters **backend** (*[type], optional*) – [description] (the default is NetworkX, which [default_description])

Returns [description]

Return type [type]

transform(*event: dict*) → Optional[Iterable[beagle.nodes.node.Node]]

class beagle.transformers.**WinEVTXTransformer**(**args*, ***kwargs*)

Bases: *beagle.transformers.base_transformer.Transformer*

name = 'Win EVTX'

process_creation(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.process.Process]

Transformers a process creation (event ID 4688) into a set of nodes.

https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventID=4688

Parameters **event** (*dict*) – [description]

Returns [description]

Return type Optional[Tuple[*Process*, *File*, *Process*, *File*]]

transform()

class beagle.transformers.**FireEyeAXTransformer**(*datasource: beagle.datasources.base_datasource.DataSource*)

Bases: *beagle.transformers.base_transformer.Transformer*

conn_events(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.ip_address.IPAddress]

Transforms a single connection event

Example event:

```
{
    "mode": "connect",
    "protocol_type": "tcp",
    "ipaddress": "199.168.199.123",
    "destination_port": 3333,
    "processinfo": {
        "imagepath": "C:\ProgramData\bloop\some_proc.exe",
        "tainted": true,
        "md5sum": "....",
        "pid": 3020
    },
    "timestamp": 27648
}
```

Parameters **event** (*dict*) – source dns_query event

Returns Process and its image, and the destination address

Return type Tuple[*Process*, *File*, *IPAddress*]

dns_events(*event: dict*) → Union[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.domain.Domain], Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.domain.Domain, beagle.nodes.ip_address.IPAddress]]

Transforms a single DNS event

Example event:

```
{
    "mode": "dns_query",
    "protocol_type": "udp",
    "hostname": "foobar",
    "qtype": "Host Address",
    "processinfo": {
        "imagepath": "C:\ProgramData\bloop\some_proc.exe",
        "tainted": true,
        "md5sum": "....",
        "pid": 3020
    },
    "timestamp": 27648
}
```

Optionally, if the event is "dns_query_answer", we can also extract the response.

Parameters **event** (*dict*) – source dns_query event

Returns Process and its image, and the domain looked up

Return type Tuple[*Process*, *File*, *Domain*]

file_events(*event: dict*) → Union[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.file.File], Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.file.File, beagle.nodes.file.File]]

Transforms a file event

Example file event:

```
{
    "mode": "created",
    "fid": { "ads": "", "content": 2533274790555891 },
    "processinfo": {
        "imagepath": "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.
↪exe",
        "md5sum": "eb32c070e658937aa9fa9f3ae629b2b8",
        "pid": 2956
    },
    "ntstatus": "0x0",
    "value": "C:\Users\admin\AppData\Local\Temp\sy24ttkc.k25.ps1",
    "CreateOptions": "0x400064",
    "timestamp": 9494
}
```

In 8.2.0 the *value* field became a dictionary when the mode is *failed*:

```
"values": {
    "value": "C:\Users\admin\AppData\Local\Temp\sy24ttkc.k25.ps1""
}
```

Parameters **event** (*dict*) – The source event

Returns The process, the process' image, and the file written.

Return type Tuple[*Process*, *File*, *File*]

http_requests(*event: dict*) → Union[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.ip_address.IPAddress, beagle.nodes.domain.URI, beagle.nodes.domain.Domain], Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.ip_address.IPAddress, beagle.nodes.domain.URI], Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.ip_address.IPAddress]]

Transforms a single *http_request* network event. A typical event looks like:

```
{
    "mode": "http_request",
    "protocol_type": "tcp",
    "ipaddress": "199.168.199.1",
    "destination_port": 80,
    "processinfo": {
        "imagepath": "c:\Windows\System32\svchost.exe",
        "tainted": false,
        "md5sum": "1234",
        "pid": 1292
    },
    "http_request": "GET /some_route.crl HTTP/1.1~~Cache-Control: max-age =
↪900~~User-Agent: Microsoft-CryptoAPI/10.0~~Host: crl.microsoft.com~~~~",
    "timestamp": 433750
}
```

Parameters **event** (*dict*) – The source *network* event with mode *http_request*

Returns [description]

Return type Tuple[*Node*]

name = 'FireEye AX'

process_events(*event: dict*) → Optional[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.process.Process, beagle.nodes.file.File]]

Transformers events from the *process* entry.

A single process entry looks like:

```
{
    "mode": string,
    "fid": dict,
    "parentname": string,
    "cmdline": string,
    "sha1sum": "string,
    "md5sum": string,
    "sha256sum": string,
    "pid": int,
    "filesize": int,
```

(continues on next page)

(continued from previous page)

```
    "value": string,
    "timestamp": int,
    "ppid": int
},
```

Parameters **event** (*dict*) – The input event.

Returns Parent and child processes, and the file nodes that represent their binaries.

Return type Optional[Tuple[*Process*, *File*, *Process*, *File*]]

regkey_events(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.registry.RegistryKey]

Transforms a single registry key event

Example event:

```
{
    "mode": "queryvalue",
    "processinfo": {
        "imagepath": "C:\Users\admin\AppData\Local\Temp\bar.exe",
        "tainted": True,
        "md5sum": "....",
        "pid": 1700,
    },
    "value":
↪"\REGISTRY\USER\Software\Microsoft\Windows\CurrentVersion\Internet Settings\
↪"ProxyOverride"",
    "timestamp": 6203
},
```

Parameters **event** (*dict*) – source regkey event

Returns Process and its image, and the registry key.

Return type Tuple[*Process*, *File*, RegistrKey]

transform()

Transformers the various events from the AX Report class.

The only edge case is the network type, AX has multiple Nodes under one type when it comes to the network type. For example the following is a DNS event:

```
{
    "mode": "dns_query",
    "protocol_type": "udp",
    "hostname": "foobar",
    "qtype": "Host Address",
    "processinfo": {
        "imagepath": "C:\ProgramData\bloop\some_proc.exe",
        "tainted": true,
        "md5sum": "....",
        "pid": 3020
    },
    "timestamp": 27648
}
```

While the following is a TCP connection:

```
{
    "mode": "connect",
    "protocol_type": "tcp",
    "ipaddress": "192.168.199.123",
    "destination_port": 3333,
    "processinfo": {
        "imagepath": "C:\ProgramData\bloop\some_proc.exe",
        "tainted": true,
        "md5sum": "...",
        "pid": 3020
    },
    "timestamp": 28029
}
```

Both have the "network" event_type when coming from `FireEyeAXReport`

Parameters **event** (*dict*) – The current event to transform.

Returns Tuple of nodes extracted from the event.

Return type Optional[Tuple]

class beagle.transformers.**FireEyeHXTransformer**(**args*, ***kwargs*)

Bases: *beagle.transformers.base_transformer.Transformer*

make_alert(*event: dict*) → Optional[Tuple[beagle.nodes.alert.Alert, ...]]

make_dnslookup(*event: dict*) → Optional[Tuple[beagle.nodes.domain.Domain, beagle.nodes.process.Process, beagle.nodes.file.File]]

Converts a dnsLookupEvent into a Domain, Process, and Process's File node.

Nodes: 1. Domain looked up.

2. Process performing the lookup.
3. File the Process was launched from.

Edges:

1. Process - (DNS Lookup For) -> Domain.
2. File - (FileOf) -> Process.

Parameters **event** (*dict*) – A dnsLookupEvent

Returns The Domain, Process, and File nodes.

Return type Optional[Tuple[*Domain*, *Process*, *File*]]

make_file(*event: dict*) → Optional[Tuple[beagle.nodes.file.File, beagle.nodes.process.Process, beagle.nodes.file.File]]

Converts a fileWriteEvent to two nodes, a file and the process manipulated the file. Generates a process - (Wrote) -> File edge.

Parameters **event** (*dict*) – The fileWriteEvent event.

Returns Returns a tuple contaning the File that this event is focused on, and the process which manipulated the file. The process has a Wrote edge to the file. Also contains the file that the process belongs to.

Return type Optional[Tuple[*File*, *Process*, *File*]]

make_imageload(*event: dict*) → Optional[Tuple[beagle.nodes.file.File, beagle.nodes.process.Process, beagle.nodes.file.File]]

make_network(*event: dict*) → Optional[Tuple[beagle.nodes.ip_address.IPAddress, beagle.nodes.process.Process, beagle.nodes.file.File]]

Converts a network connection event into a Process, File and IP Address node.

Nodes:

1. IP Address communicated to.
2. Process contacting IP.
3. File process launched from.

Edges:

1. Process - (Connected To) -> IP Address
2. File - (File Of) -> Process

Parameters **event** (*dict*) – The ipv4NetworkEvent

Returns The IP Address, Process, and Process's File object.

Return type Optional[Tuple[*IPAddress*, *Process*, *File*]]

make_process(*event: dict*) → Union[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File], Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.process.Process, beagle.nodes.file.File], None]

Converts a processEvent into either one Process node, or two Process nodes with a parent - (Launched) -> child relationship. Additionally, creats File nodes for the images of both of the Processe's identified.

Parameters **event** (*dict*) – The processEvent event

Returns Returns either a single process node, or a (parent, child) tuple where the parent has a launched edge to the child.

Return type Optional[Union[Tuple[*Process*, *File*], Tuple[*Process*, *File*, *Process*, *File*]]]

make_registry(*event: dict*) → Optional[Tuple[beagle.nodes.registry.RegistryKey, beagle.nodes.process.Process, beagle.nodes.file.File]]

make_url(*event: dict*) → Optional[Tuple[beagle.nodes.domain.URI, beagle.nodes.domain.Domain, beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.ip_address.IPAddress]]

Converts a URL access event and returns 5 nodes with 4 different relationships.

Nodes created:

1. URI Accessed (e.g /foobar)
2. Domain Accessed (e.g omer.com)
3. Process performing URL request.
4. File object for the Process image.
5. IP Address the domain resolves to.

Relationships created:

1. URI - (URI Of) -> Domain
2. Domain - (Resolves To) -> IP Address
3. Process - (*http method of event*) -> URI
4. Process - (Connected To) -> IP Address
5. File - (File Of) -> Process

Parameters **event** (*dict*) – The urlMonitorEvent events

Returns 5 tuple of the nodes pulled out of the event (see function description).

Return type Optional[Tuple[*URI*, *Domain*, *Process*, *File*, *IPAddress*]]

name = 'FireEye HX'

transform(*event: dict*) → Optional[Tuple[beagle.nodes.node.Node, ...]]
Sends each event from the FireEye HX Triage to the appropriate node creation function.

Parameters **event** (*dict*) – The source event from the HX Triage

Returns The results of the transforming function

Return type Optional[Tuple[*Node*, ..]]

class beagle.transformers.**GenericTransformer**(**args*, ***kwargs*)
Bases: *beagle.transformers.base_transformer.Transformer*

This transformer will properly create graphs for any datasource that outputs data in the pre-defined schema.

make_alert(*event: dict*) → Tuple[beagle.nodes.alert.Alert, ...]

make_basic_file(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.file.File]
Transforms a file based event.

Support events:

1. EventTypes.FILE_DELETED
2. EventTypes.FILE_OPENED
3. EventTypes.FILE_WRITTEN
4. EventTypes.LOADED_MODULE

Parameters **event** (*dict*) – [description]

Returns [description]

Return type Tuple[*Process*, *File*, *File*]

make_basic_regkey(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.registry.RegistryKey]

make_connection(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.ip_address.IPAddress]

make_dnslookup(*event: dict*) → Union[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.domain.Domain, beagle.nodes.ip_address.IPAddress], Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.domain.Domain]]

make_file_copy(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.file.File, beagle.nodes.file.File]

make_http_req(*event: dict*) → Union[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.domain.URI, beagle.nodes.domain.Domain], Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.domain.URI, beagle.nodes.domain.Domain, beagle.nodes.ip_address.IPAddress]]

make_process(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.process.Process, beagle.nodes.file.File]

Accepts a process with the *EventTypes.PROCESS_LAUNCHED* event_type.

For example:

```
{
    FieldNames.PARENT_PROCESS_IMAGE: "cmd.exe",
    FieldNames.PARENT_PROCESS_IMAGE_PATH: "\",
    FieldNames.PARENT_PROCESS_ID: "2568",
    FieldNames.PARENT_COMMAND_LINE: '/K name.exe"',
    FieldNames.PROCESS_IMAGE: "find.exe",
    FieldNames.PROCESS_IMAGE_PATH: "\",
    FieldNames.COMMAND_LINE: 'find /i "svhost.exe"',
    FieldNames.PROCESS_ID: "3144",
    FieldNames.EVENT_TYPE: EventTypes.PROCESS_LAUNCHED,
}
```

Parameters **event** (*dict*) – [description]

Returns [description]

Return type Tuple[*Process*, *File*, *Process*, *File*]

make_regkey_set_value(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.registry.RegistryKey]

name = 'Generic'

transform()

class beagle.transformers.**ProcmonTransformer**(*datasource: beagle.datasources.base_datasource.DataSource*)

Bases: *beagle.transformers.base_transformer.Transformer*

access_file(*event*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File]

access_reg_key(*event*) → Tuple[beagle.nodes.process.Process, beagle.nodes.registry.RegistryKey]

connection(*event*) → Tuple[beagle.nodes.process.Process, beagle.nodes.ip_address.IPAddress]

name = 'Procmon'

process_create(*event*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.process.Process]

transform()

write_file(*event*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File]

class beagle.transformers.**PCAPTransformer**(**args*, ***kwargs*)

Bases: *beagle.transformers.base_transformer.Transformer*

name = 'PCAP'

transform(*event: Dict*) → Optional[Tuple[beagle.nodes.node.Node, ...]]

class beagle.transformers.**SysmonTransformer**(**args*, ***kwargs*)

Bases: *beagle.transformers.base_transformer.Transformer*

dns_lookup(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.domain.Domain]

file_created(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.file.File]

name = 'Sysmon'

network_connection(*event: dict*) → Union[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.ip_address.IPAddress], Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.ip_address.IPAddress, beagle.nodes.domain.Domain]]

process_creation(*event: dict*) → Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.process.Process, beagle.nodes.file.File]

registry_creation(*event: dict*) → Optional[Tuple[beagle.nodes.process.Process, beagle.nodes.file.File, beagle.nodes.registry.RegistryKey]]

transform()

class beagle.transformers.**DRAPATCTransformer**(**args*, ***kwargs*)
Bases: *beagle.transformers.base_transformer.Transformer*

conn_events(*event: dict*) → Tuple[beagle.transformers.darpa_tc_transformer.TCProcess, beagle.transformers.darpa_tc_transformer.TCIPAddress]

execute_events(*event: dict*) → Tuple[beagle.transformers.darpa_tc_transformer.TCProcess, beagle.transformers.darpa_tc_transformer.TCProcess]

file_events(*event: dict*) → Tuple[beagle.transformers.darpa_tc_transformer.TCProcess, beagle.transformers.darpa_tc_transformer.TCFile]

make_addr(*event: dict*) → Tuple[beagle.transformers.darpa_tc_transformer.TCIPAddress]

make_file(*event: dict*) → Tuple[beagle.transformers.darpa_tc_transformer.TCFile]

make_process(*event: dict*) → Union[Tuple[beagle.transformers.darpa_tc_transformer.TCProcess], Tuple[beagle.transformers.darpa_tc_transformer.TCProcess, beagle.transformers.darpa_tc_transformer.TCProcess]]

make_registrykey(*event: dict*) → Tuple[beagle.transformers.darpa_tc_transformer.TCRegistryKey]

name = 'DARPA TC'

transform()

1.6 beagle.web package

1.6.1 Subpackages

beagle.web.api package

Submodules

beagle.web.api.models module

class beagle.web.api.models.**Graph**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Model

category

comment

file_path

id

meta

sha256

to_json()

class beagle.web.api.models.**JSONEncodedDict**(**args*, ***kwargs*)
Bases: sqlalchemy.sql.type_api.TypeDecorator

impl
alias of sqlalchemy.sql.sqltypes.VARCHAR

process_bind_param(*value*, *dialect*)
Receive a bound parameter value to be converted.

Subclasses override this method to return the value that should be passed along to the underlying TypeEngine object, and from there to the DBAPI execute() method.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

This operation should be designed with the reverse operation in mind, which would be the process_result_value method of this class.

Parameters

- **value** – Data to operate upon, of any type expected by this method in the subclass. Can be None.
- **dialect** – the Dialect in use.

process_result_value(*value*, *dialect*)
Receive a result-row column value to be converted.

Subclasses should implement this method to operate on data fetched from the database.

Subclasses override this method to return the value that should be passed back to the application, given a value that is already processed by the underlying TypeEngine object, originally from the DBAPI cursor method fetchone() or similar.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

Parameters

- **value** – Data to operate upon, of any type expected by this method in the subclass. Can be None.
- **dialect** – the Dialect in use.

This operation should be designed to be reversible by the "process_bind_param" method of this class.

beagle.web.api.views module

beagle.web.api.views.**add**(*graph_id: int*)
Add data to an existing NetworkX based graph.

Parameters **graph_id** (*int*) – The graph ID to add to.

beagle.web.api.views.**adhoc**()
Allows for ad-hoc transformation of generic JSON Data based on one of two CIM models:

1. The Beagle CIM Model (defined in *constants.py*)

2. The OSSEM Model (defined in https://github.com/Cyb3rWard0g/OSSEM)

beagle.web.api.views.**get_backends**()
Returns all possible backends, their names, and their IDs.

The array contains elements with the following structure.

```
>>> {
    id: string, # class name
    name: string # Human-readable name
}
```

These map back to the __name__ attributes of Backend subclasses.

Returns Array of {id: string, name: string} entries.

Return type List[dict]

beagle.web.api.views.**get_categories**()
Returns a list of categories as id, name pairs.

This list is made up of all categories specified in the category field for each datasource.

```
>>> {
    "id": "vt_sandbox",
    "name": "VT Sandbox"
}
```

Returns

Return type List[dict]

beagle.web.api.views.**get_category_items**(*category: str*)
Returns the set of items that exist in this category, the path to their JSON files, the comment made on them, as well as their metadata.

```
>>> {
    comment: str,
    file_path: str,
    id: int,
    metadata: Dict[str, Any]
}
```

Returns 404 if the category is invalid.

Parameters **category** (*str*) – The category to fetch data for.

Returns

Return type List[dict]

beagle.web.api.views.**get_graph**(*graph_id: int*)
Returns the JSON object for this graph. This is a networkx node_data JSON dump:

```
>>> {
    directed: boolean,
    links: [
        {...}
    ],
    multigraph: boolean,
    nodes: [
```

(continues on next page)

(continued from previous page)

```
        {...}
    ]
}
```

Returns 404 if the graph is not found.

Parameters **graph_id** (*int*) – The graph ID to fetch data for

Returns See https://networkx.github.io/documentation/stable/reference/readwrite/generated/networkx.readwrite.json_graph.node_link_graph.html

Return type Dict

beagle.web.api.views.**get_graph_metadata**(*graph_id: int*)

Returns the metadata for a single graph. This is automatically generated by the datasource classes.

Parameters

- **graph_id** (*int*) – Graph ID.
- **404 if the graph ID is not found** (*Returns*) –

Returns A dictionary representing the metadata of the current graph.

Return type Dict

beagle.web.api.views.**get_transformers**()

Returns all possible transformers, their names, and their IDs.

The array contains elements with the following structure.

```
>>> {
    id: string, # class name
    name: string # Human-readable name
}
```

These map back to the __name__ and .name attributes of Transformer subclasses.

Returns Array of {id: string, name: string} entries.

Return type List[dict]

beagle.web.api.views.**new**()

Generate a new graph using the supplied DataSource, Transformer, and the parameters passed to the DataSource.

At minimum, the user must supply the following form parameters:

1. datasource
2. transformer
3. comment
4. backend

Outside of that, the user must supply at **minimum** the parameters marked by the datasource as required.

- Use the /api/datasources endpoint to see which ones these are.
- Programmatically, these are any parameters without a default value.

Failure to supply either the minimum three or the required parameters for that datasource returns a 400 status code with the missing parameters in the 'message' field.

If any part of the graph creation yields an error, a 500 HTTP code is returend with the python exception as a string in the 'message' field.

If the graph is succesfully created, the user is returned a dictionary with the ID of the graph and the URI path to viewing it in the *beagle web interface*.

For example:

```
>>> {
    id: 1,
    self: /fireeye_hx/1
}
```

Returns {id: integer, self: string}

Return type dict

beagle.web.api.views.**pipelines**()

Returns a list of all available datasources, their parameters, names, ids, and supported transformers.

A single entry in the array is formatted as follows:

```
>>> {
    "id": str,
    "name": str,
    "params": [
        {
            "name": str,
            "required": bool,
        }
        ...
    ],
    "transformers": [
        {
            "id": str,
            "name": str
        }
    ]
    "type": "files" OR "external
}
```

If the 'type' field is set to 'files', it means that the parameters represent required files, if it is set to 'external' this means that the parameters represent string inputs.

The main purpose of this endpoint is to allow users to query beagle in order to easily identify what datasource and transformer combinations are possible, as well as what parameters are required.

Returns An array of datasource specifications.

Return type List[dict]

Module contents

1.6.2 Submodules

1.6.3 beagle.web.server module

beagle.web.server.**create_app**(**args*)

beagle.web.server.**root_view**()

1.6.4 beagle.web.wsgi module

1.6.5 Module contents

CHAPTER 2

Submodules

CHAPTER 3

beagle.config module

class beagle.config.**BeagleConfig**(*defaults=None*, *dict_type=<class 'collections.OrderedDict'>*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section='DEFAULT'*, *interpolation=<object object>*, *converters=<object object>*)

Bases: configparser.ConfigParser

get(*section: str*, *key: str*, ***kwargs*)

Get an option value for a given section.

If ‘vars’ is provided, it must be a dictionary. The option is looked up in ‘vars’ (if provided), ‘section’, and in ‘DEFAULTSECT’ in that order. If the key is not found and ‘fallback’ is provided, it is used as a fallback value. ‘None’ can be provided as a ‘fallback’ value.

If interpolation is enabled and the optional argument ‘raw’ is False, all interpolations are expanded in the return values.

Arguments ‘raw’, ‘vars’, and ‘fallback’ are keyword only.

The section DEFAULT is special.

beagle.config.**expand_env_var**(*env_var: str*)

Expands (potentially nested) env vars by repeatedly applying *expandvars* and *expanduser* until interpolation stops having any effect.

CHAPTER 4

beagle.constants module

class beagle.constants.**EventTypes**

Bases: object

CONNECTION = 'connection'

DNS_LOOKUP = 'dns_lookup'

FILE_COPIED = 'file_copied'

FILE_DELETED = 'file_deleted'

FILE_OPENED = 'file_opened'

FILE_WRITTEN = 'file_written'

HTTP_REQUEST = 'http_request'

LOADED_MODULE = 'loaded_module'

PROCESS_LAUNCHED = 'process_launched'

REG_KEY_DELETED = 'reg_key_deleted'

REG_KEY_OPENED = 'reg_key_opened'

REG_KEY_SET = 'reg_key_set'

class beagle.constants.**FieldNames**

Bases: object

ALERTED_ON = 'alerted_on'

ALERT_DATA = 'alert_data'

ALERT_NAME = 'alert_name'

COMMAND_LINE = 'command_line'

DEST_FILE = 'dst_file'

EVENT_TYPE = 'event_type'

FILE_NAME = 'file_name'

FILE_PATH = 'file_path'

HASHES = 'hashes'

HIVE = 'hive'

HTTP_HOST = 'http_host'

HTTP_METHOD = 'http_method'

IP_ADDRESS = 'ip_address'

PARENT_COMMAND_LINE = 'parent_command_line'

PARENT_PROCESS_ID = 'parent_process_id'

PARENT_PROCESS_IMAGE = 'parent_process_image'

PARENT_PROCESS_IMAGE_PATH = 'parent_process_image_path'

PORT = 'port'

PROCESS_ID = 'process_id'

PROCESS_IMAGE = 'process_image'

PROCESS_IMAGE_PATH = 'process_image_path'

PROTOCOL = 'protocol'

REG_KEY = 'reg_key'

REG_KEY_PATH = 'reg_path'

REG_KEY_VALUE = 'reg_key_value'

SRC_FILE = 'src_file'

TIMESTAMP = 'timestamp'

URI = 'uri'

class beagle.constants.**HTTPMethods**

Bases: object

CONNECT = 'CONNECT'

DELETE = 'DELETE'

GET = 'GET'

HEAD = 'HEAD'

OPTIONS = 'OPTIONS'

POST = 'POST'

PUT = 'PUT'

TRACE = 'TRACE'

class beagle.constants.**HashAlgos**

Bases: object

MD5 = 'md5'

SHA1 = 'sha1'

SHA256 = 'sha256'

class beagle.constants.**Protocols**

Bases: object

HTTP = 'HTTP'

ICMP = 'ICMP'

TCP = 'TCP'

UDP = 'UDP'

CHAPTER 5

Module contents

Python Module Index

b

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

W