Canari Framework Documentation

Release 3.0

Nadeem Douba

Aug 14, 2018

Contents

Contents:

CHAPTER 1

Canari Quick Start

Welcome to the Canari Framework - the world's most advanced rapid transform development framework for Maltego. In this quickstart tutorial we'll go over how you can take advantage of Canari's powerful feature set to create your own Maltego transform package. We'll start by developing a local transform package and then migrate that over to a remote transform package which you can distributed via the Paterva TDS. Enough jibber jabber and let's get this show on the road.

1.1 Installation

Canari requires the following dependencies to get started:

1. Python 2.7 or later (prior to Python 3) - Download
2. setuptools - Download
3. virtualenv - Download

Note: Canari does not support Python version 3.

1.1.1 Installing Dependencies

Linux Debian-based

On Debian-based (Ubuntu, Kali, etc.) systems, all these dependencies can be installed using **apt-get**:

```
$ sudo apt-get install python2.7 python-virtualenv python-setuptools
```

Linux - Fedora-based

On Fedora-based (Fedora, RedHat, CentOS, etc.) systems, all these dependencies can be installed using **yum**:

```
$ sudo yum groupinstall -y 'development tools'
$ sudo yum install zlib-devel bzip2-devel openssl-devel ncurses-devel sqlite-devel \
    readline-devel tk-devel gdbm-devel db4-devel libpcap-devel xz-devel python-devel
$ sudo easy_install virtualenv
```

Mac OS/X

On Mac OS/X, make sure to install Xcode from the App Store, first. Then install the command-line tools like so:

```
$ sudo xcode-select --install
$ wget https://pypi.python.org/packages/source/s/setuptools/setuptools-18.4.tar.gz
$ tar -zxvf setuptools-18.4.tar.gz
$ cd setuptools-18.4 && sudo python setup.py install
$ sudo easy_install virtualenv
```

1.1.2 Installing Canari

Once you have all your dependencies installed, you can now install Canari. We recommend creating a virtual environment to reduce clutter in your default Python site-package directory. Virtual environments can be created easily like so:

```
$ virtualenv canari3
New python executable in canari3/bin/python
Installing setuptools, pip...done.
```

This will create a completely separate Python environment in the `canari3` directory, which you can use to install custom Python libraries to without the risk of corrupting your default Python environment. Another advantage to virtual environments is that they can be easily cleaned up if you no longer need them. To activate your virtual environment, do the following:

```
$ source canari3/bin/activate
$ which python
canari3/bin/python
```

Attention: Virtual environments need to be activated every time you create a new terminal session. Otherwise, you'll be using the default Python installation. You can automate this process by adding the `source` statement above in your `.profile` or `.bashrc` file.

Once you've activated your virtual environment, it is now time to install Canari:

```
$ pip install canari
```

Note: One of the advantages of virtual environments is that you no longer have to use **`sudo`** to install custom Python modules.

Now you're all set to get started developing your first transform package!

1.2 Hello World!

Let's start by creating our first transform package. This will include an example "Hello World!" transform for your convenience. To create a transform package we use the **canari** commander like so:

```
$ canari create-package hello
creating skeleton in hello
--> Project description: My first transform package

--> Author name [ndouba]:

--> Author email: myemail@foo.com

done!
$
```

The `create-package` commandlet creates the skeleton for your transform package. It starts off by asking you some standard information about the package and uses that information to populate authorship information in your transform code.

Note: The **canari** commander has many other commandlets that you can take advantage of. For a full list of commands take a look at the output of **canari list-commands**.

If your transform package was successfully created, you should now see a `hello` folder in your working directory:

```
$ ls
hello ...
```

Let's drop into that directory and run our first transform. As mentioned earlier, each time you create a new transform package, a "Hello World!" transform gets created for your reference. We'll execute this transform using the **canari debug-transform** transform runner:

```
$ cd hello/src
$ canari debug-transform hello.transforms.helloworld.HelloWorld Bob
`- MaltegoTransformResponseMessage:
  `- UIMessages:
  `- Entities:
    `- Entity:  {'Type': 'maltego.Phrase'}
      `- Value: Hello Bob!
      `- Weight: 1
    `- Entity:  {'Type': 'maltego.Phrase'}
      `- Value: This way Mr(s). None!
      `- Weight: 1
    `- Entity:  {'Type': 'maltego.Phrase'}
      `- Value: Hi None!
      `- Weight: 1
```

You'll probably see the output above and you may be wondering why are we seeing `None` in places where we'd expect to see `Bob`. This is because the example transform also demonstrates the use of transform fields. Go ahead and open the transform in your favorite text editor located at `src/hello/transforms/helloworld.py` - you should see the following:

```
class HelloWorld(Transform):
    # The transform input entity type.
    input_type = Person # <------------------------------------------------ 1
```

(continues on next page)

(continued from previous page)

```
    def do_transform(self, request, response, config):
        person = request.entity
        response += Phrase('Hello %s!' % person.value)
        response += Phrase('This way Mr(s). %s!' % person.lastname) # <---- 2
        response += Phrase('Hi %s!' % person.firstnames) # <--------------- 3
        return response
```

In our example, the `HelloWorld` transform expects an input type of `Person` (1). If we look in `HelloWorld.do_transform()` we see that it references the `person.lastname` (2) and `person.firstnames` (3) entity fields. Let's pass these fields to our transform runner:

```
$ canari debug-transform hello.transforms.helloworld.HelloWorld Bob "person.
↪lastname=Doe#person.firstnames=Bob"
`- MaltegoTransformResponseMessage:
  `- UIMessages:
  `- Entities:
    `- Entity:  {'Type': 'maltego.Phrase'}
      `- Value: Hello Bob!
      `- Weight: 1
    `- Entity:  {'Type': 'maltego.Phrase'}
      `- Value: This way Mr(s). Doe!
      `- Weight: 1
    `- Entity:  {'Type': 'maltego.Phrase'}
      `- Value: Hi Bob!
      `- Weight: 1
```

Note: In this case, the entity field names coincidentally matched the names in our code example above. However, this will not always be the case. Take a look at the *canari.maltego.entities* file for a full set of builtin Maltego entity definitions and their fields.

Now that we've run our first transform successfully and understand the use of transform fields, let's create our first custom transform.

1.3 Your First Transform

Using the same package above, in our `hello` directory, let's start off by creating a transform using the **`canari create-transform`** commandlet, like so:

```
$ canari create-transform whatismyip
Creating transform 'whatismyip'...
done!
```

As you may have guessed already, we are going to write a transform that determines our current Internet IP address. Let's use the free JSON API at ipify. First let's make sure you can reach the server by clicking here or typing the following in your terminal:

```
$ curl 'https://api.ipify.org?format=json'
{"ip":"123.123.123.123"}
```

You should see something like the output above, except your IP address would appear in place of "123.123.123.123". Great! Let's write the transform with the following design principles:

1. Our transform will expect a `Location` entity as input.
2. Our transform will return an `IPv4Address` entity as output.

Let's go ahead and open our `src/hello/transforms/whatismyip.py` transform and implement the code:

```
from urllib import urlopen
import json

from canari.maltego.entities import IPv4Address, Location
from canari.maltego.transform import Transform
from canari.framework import EnableDebugWindow

@EnableDebugWindow
class Whatismyip(Transform):
    """Returns my Internet IP Address"""

    input_type = Location

    def do_transform(self, request, response, config):
        ip_json = urlopen('https://api.ipify.org?format=json').read() # <-- 1
        ip_address = json.loads(ip_json)['ip'] # <------------------------ 2
        response += IPv4Address(ip_address) # <---------------------------- 3
        return response # <------------------------------------------------ 4
```

The `input_type` class property tells Canari to expect an input entity of type `Location`. This ensures that the transform will only appear in the context menu of a `Location` entity in Maltego (i.e. under the run transform menu options). Here's what's going on line-by-line inside the `do_transform()`:

1. First we make our request to `ipify` and get our IP address as a JSON string
2. We parse the JSON we got from `ipify` (i.e. `{"ip":"123.123.123.123"}`) and extract our IP address.
3. We then create an `IPv4Address` entity with the default value set to our IP address and append it to our response.
4. Finally we return the response to Maltego.

Let's see if our transform is operating correctly:

```
$ cd src
$ canari debug-transform hello.transforms.whatismyip.Whatismyip Home
`- MaltegoTransformResponseMessage:
  `- UIMessages:
  `- Entities:
    `- Entity:  {'Type': 'maltego.IPv4Address'}
      `- Value: 216.48.160.29
      `- Weight: 1
```

Great! Let's try this out in Maltego. First we need to create a profile that can be imported by Maltego to configure the transforms in the GUI:

```
$ canari create-profile hello
Looking for transforms in hello...
Package loaded.
Creating profile ~/hello/src/hello.mtz...
Installing transform hello.HelloWorld from hello.transforms.helloworld.HelloWorld...
Installing transform hello.Whatismyip from hello.transforms.whatismyip.Whatismyip...
Writing ~/hello/src/hello/resources/etc/hello.conf to /Users/ndouba/tools/canari3/
↪build/hello/src/hello.conf
```

(continues on next page)

(continued from previous page)

```
Updating ~/hello/src/canari.conf...
Writing transform set Hello to ~/hello/src/hello.mtz...
Writing transform set Canari to ~/hello/src/hello.mtz...
Writing server Local to ~/hello/src/hello.mtz

%%%%%%%%%%%%%%%%%%%%%%%%%%% SUCCESS! %%%%%%%%%%%%%%%%%%%%%%%%%%%

 Successfully created /Users/ndouba/tools/canari3/build/hello/src/hello.mtz. You may␣
↪now import this file into
 Maltego.

 INSTRUCTIONS:
 -------------
 1. Open Maltego.
 2. Click on the home button (Maltego icon, top-left corner).
 3. Click on 'Import'.
 4. Click on 'Import Configuration'.
 5. Follow prompts.
 6. Enjoy!

%%%%%%%%%%%%%%%%%%%%%%%%%%% SUCCESS! %%%%%%%%%%%%%%%%%%%%%%%%%%%
```

This should have created a `hello.mtz` file in the directory where you ran the command. Let's import this profile into Maltego:

1. Open Maltego.
2. Click on the Maltego home button (big Maltego icon in the top left corner).
3. Navigate to `Import` then click on `Import Configuration`
4. Select your `hello.mtz` file and accept the defaults in the wizard.

Warning: Canari Maltego profile files are not redistributable. This is because the path of your local transforms and Canari framework files will vary across systems. Instead, developers of local transforms should always include the Canari `create-profile` instructions as part of the transform package's installation steps.

Once you've successfully imported your profile, create a new graph and drag a `Location` entity onto the graph. Then right click on the newly created `Location` entity, look for the `Hello` transform set, and click `Whatismyip`.

If all went well you should now see your IP address magically appear on the graph right below your `Location` entity.

Note: If you're familiar with Canari v1 you may have noticed a few of Canari v3's awesome features at work. One of them is that the transform set and transform name in the Maltego UI are derived from the Canari package and transform names, respectively. If you dig a little deeper, you may also notice that the transform description is derived from the transform class' `__doc__` string property.

Let's say you wanted to change the name of the transform as it appears in Maltego. There are two ways of doing this:

1. You can adjust the transform class' name into camel case (i.e. `Whatismyip` to `WhatIsMyIP`). This will tell Canari to insert a space between each uppercase letter in the transform's name in Maltego.
2. You can set the transform class' `display_name` property to the label of your choice.

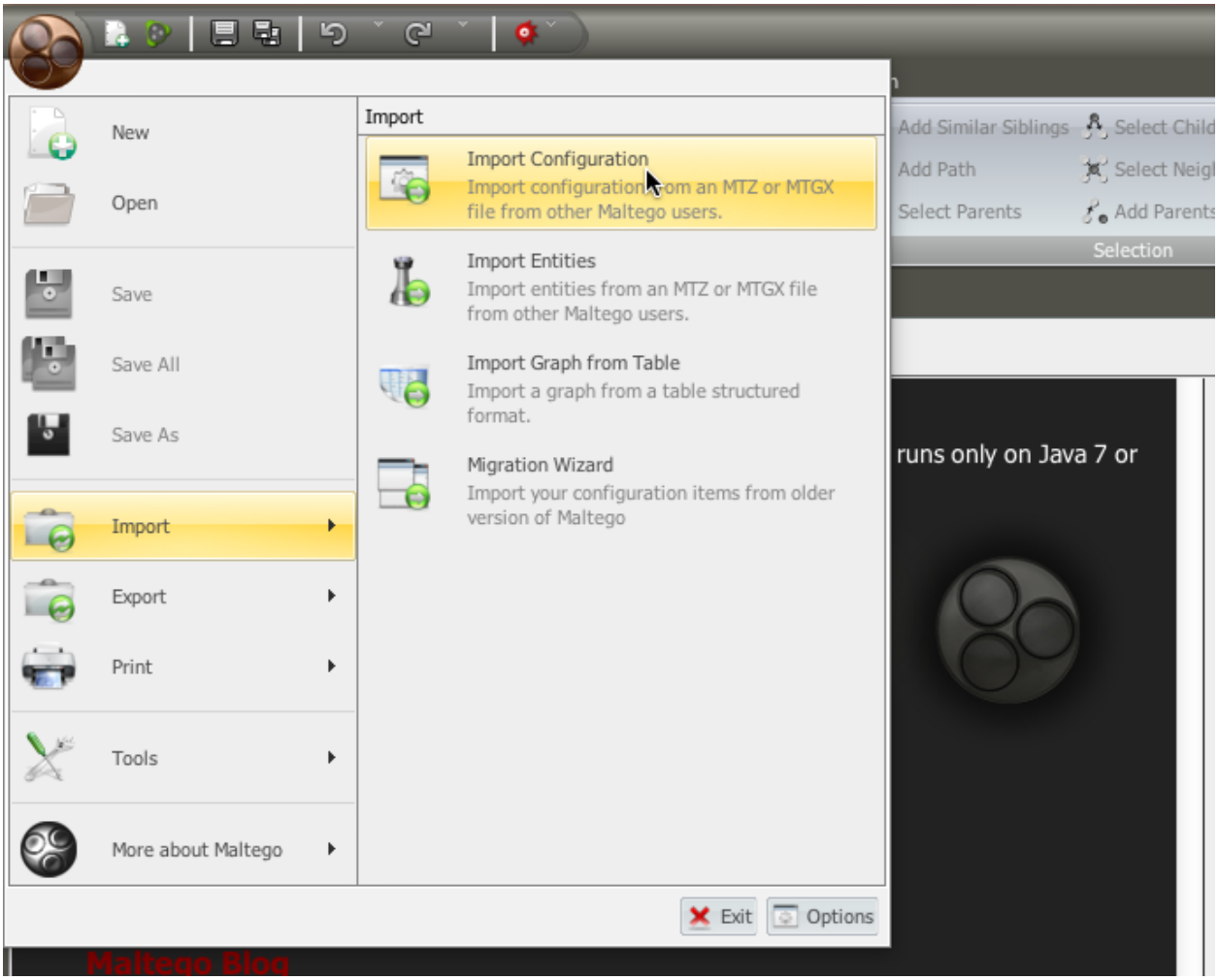

Fig. 1: Maltego `Import Profile` menu option

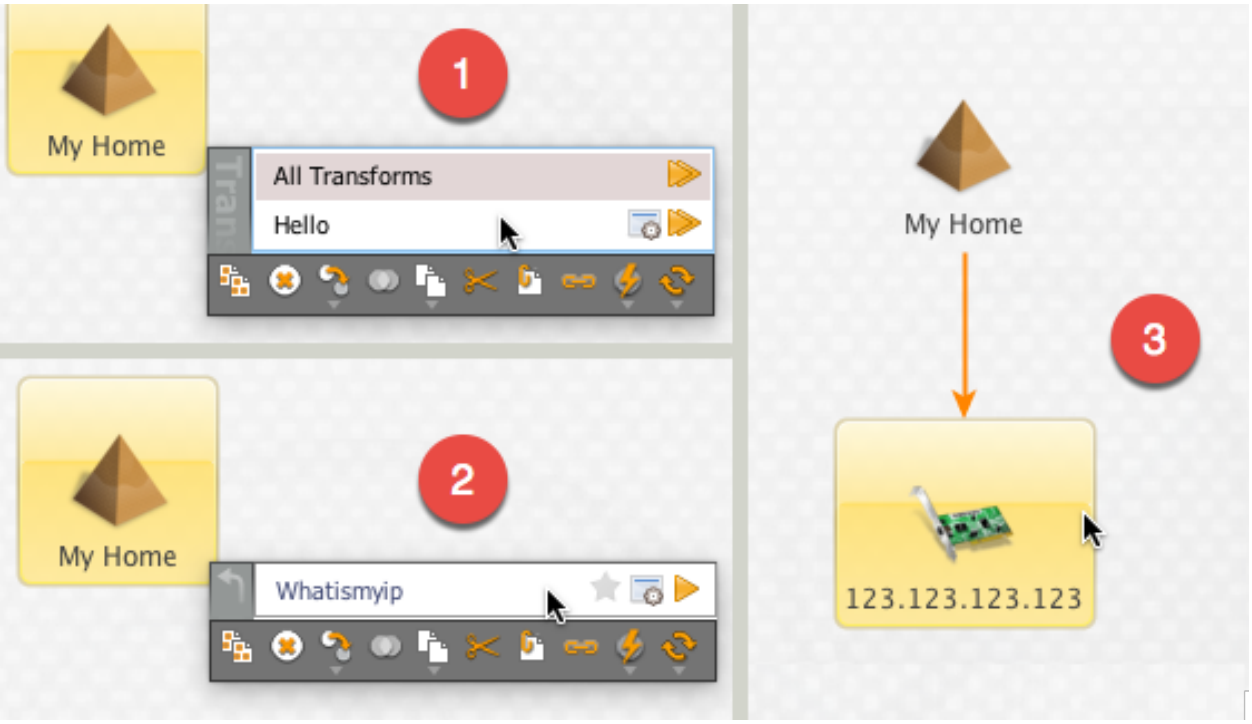

Fig. 2: Maltego run transform steps

Let's try it out by subclassing the `Whatismyip` and adding the following lines to the end of the `src/hello/transforms/whatismyip.py` file:

```
class ToMyIP(Whatismyip):
    pass
```

After you've saved your changes, recreate your Maltego profile using the **`canari create-profile hello`** command, re-import the configuration into Maltego, and run the transform like before. You should now see a `To My IP` transform in the transform context menu.

The previous example demonstrated the use of subclassing to reuse transform code. Subclassing a transform is useful when you want to reuse transform logic that could be applied to other entity types as well. For example, say you have a nifty threat intelligence transform that could be run on either an IP address or a DNS name. Instead of copying and pasting the same code over and over again, you can simply implement it once, subclass the original transform, and adjust the `input_type` property to the desired type in the child class. Let's say we wanted `ToMyIP` in our previous example to only apply to `Phrase` entities then we'd adjust the code, like so:

```
class ToMyIP(Whatismyip):
    # don't forget to import maltego.entities.Phrase
    input_type = Phrase
```

Finally, you may have noticed that we completely ignored the value of the input entity in this example. This is because our transform didn't need to use your location's name in order to get your IP address. Let's create another transform, except this time we'll use the information passed into the transform by the input entity.

Note: You may be wondering if you have to recreate and re-import Maltego profiles each time you make a transform change. The answer is yes and no. If you are only updating the behaviour (i.e. body of the `do_transform()`

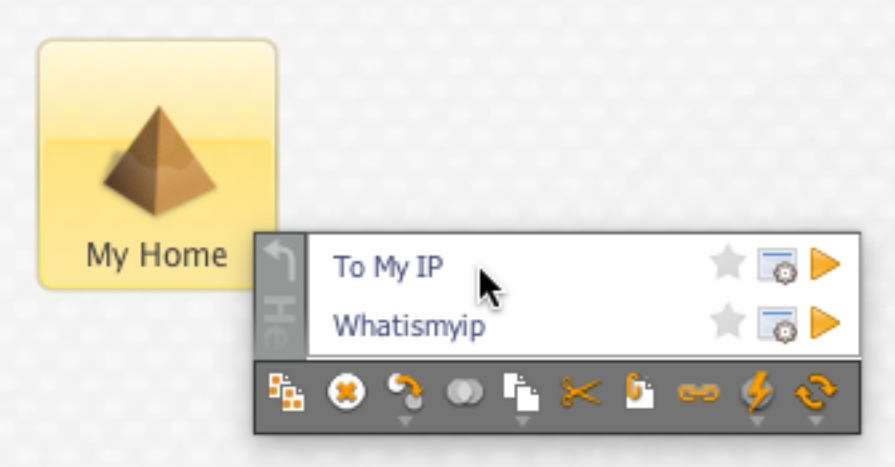

Fig. 3: Transform user-friendly name.

method) of your transform, the answer is no. However, if you want to adjust things such as the display name, the transform class name, transform description, transform set name, then the answer is yes. Often times you will find yourself recreating the profile and reinstalling it whenever you add or rename a transform in your package.

1.3.1 Working With Input Entities

Now that we know how to return entities to Maltego, let's take a look at how to receive input. In this example we'll use the FreeGeoIP JSON API to get the country, city, and region associated with an IP address. The transform will be designed with the following design principles:

1. The transform will accept an `IPv4Address` as input.
2. The transform will return a `Location` entity as output.

First let's create our transform by running **`canari create-transform IPToLocation`** in your terminal:

```
$ canari create-transform IPToLocation
Creating transform 'iptolocation'...
done!
```

Note: This time we've passed the name of the transform in camel case to the `create-transform` command to avoid having to change it later.

Next, let's edit the `src/hello/transforms/iptolocation.py` file and implement our transform logic:

```
import json
from urllib import urlopen

from canari.framework import EnableDebugWindow
from canari.maltego.entities import IPv4Address, Location
from canari.maltego.transform import Transform


@EnableDebugWindow
```

(continues on next page)

(continued from previous page)

```
class IPToLocation(Transform):
    """Get's the city/country associated with a particular IP address."""

    # The transform input entity type.
    input_type = IPv4Address

    def do_transform(self, request, response, config):
        ip_address = request.entity.value # <----------------------- 1

        geoip_str = urlopen('https://freegeoip.net/json/%s' % ip_address).read()
        geoip_json = json.loads(geoip_str)

        l = Location()
        l.country = geoip_json.get('country_name', 'Unknown') # <--- 2
        l.city = geoip_json.get('city')
        l.countrycode = geoip_json.get('country_code')
        l.latitude = geoip_json.get('latitude')
        l.longitude = geoip_json.get('longitude')
        l.area = geoip_json.get('region_name')

        response += l
        return response
```

As you can see, the first line (1) in our `do_transform()` method retrieves the display value of our input entity and stores it in the `ip_address` variable. The display value is the value that is shown below the entity's icon in the Maltego GUI. For example, the display value for an `IPv4Address` entity in Maltego is an IP V4 address (i.e. `192.168.0.1`). The `request` object is where all Maltego request information is stored and has the following properties:

1. The input entity and its fields are stored in the `entity` property; its type is determined by the value of your transform's `input_type`.
2. The `parameters` property returns a list of transform parameters. When Canari is operating in local transform mode, this property contains the unparsed command line arguments. In remote operating mode, the transform parameters passed in by the Maltego client are stored.
3. The `limits` property returns the transforms soft and hard limit. This property is not applicable in local transform mode as Maltego's local transform adapter doesn't pass in this information.

Next we issue our request to FreeGeoIP for the requested IP address and convert the JSON response into a python dictionary. The `Location` entity is then initialized (2) and its respective field values are then set to the values retrieved from our JSON object. Finally, we append the entity to our `response` object and return the output to Maltego.

Note: The default value of a `Location` entity in Maltego's GUI is calculated based on the values of the city and country name entity fields. Therefore, setting a default value for a `Location` entity has no effect and is unnecessary.

In our previous example, we illustrated how to set the values of our output entity's fields using the property setters (i.e. `l.country = 'CA'`). However, we can also set these entity fields by passing them in as keyword arguments. Let's refactor the code in the `IPToLocation.do_transform()` method to demonstrate this feature:

```
def do_transform(self, request, response, config):
    ip_address = request.entity.value

    geoip_str = urlopen('https://freegeoip.net/json/%s' % ip_address).read()
```

(continues on next page)

(continued from previous page)

```
    geoip_json = json.loads(geoip_str)

    response += Location(
        country=geoip_json.get('country_name', 'Unknown'),
        city=geoip_json.get('city'),
        countrycode=geoip_json.get('country_code'),
        latitude=geoip_json.get('latitude'),
        longitude=geoip_json.get('longitude'),
        area=geoip_json.get('region_name')
    )

    return response
```

Let's say we wanted to add a little more information or color to our graphs. Maltego supports both link and entity decorations. Labels, colors, thicknesses and styles can be applied to the links or edges connecting the output entities to their parent input entities. Entities can be bookmarked (or starred) and comments can be attached. Let's add a link label and bookmark the `Location` entity returned in our previous example:

```
def do_transform(self, request, response, config):
    # don't forget to add `from maltego.message import Bookmark`
    ip_address = request.entity.value

    geoip_str = urlopen('https://freegeoip.net/json/%s' % ip_address).read()
    geoip_json = json.loads(geoip_str)

    response += Location(
        country=geoip_json.get('country_name', 'Unknown'),
        city=geoip_json.get('city'),
        countrycode=geoip_json.get('country_code'),
        latitude=geoip_json.get('latitude'),
        longitude=geoip_json.get('longitude'),
        area=geoip_json.get('region_name'),
        link_label='From FreeGeoIP',
        bookmark=Bookmark.Orange
    )

    return response
```

Let's take a look at the before and after difference:

Finally, let's add an icon to our output entity. Since we're working with locations, we'll set the output entity's icon to the flag that corresponds with the country code:

```
def do_transform(self, request, response, config):
    # don't forget to add `from maltego.message import Bookmark`
    ip_address = request.entity.value

    geoip_str = urlopen('https://freegeoip.net/json/%s' % ip_address).read()
    geoip_json = json.loads(geoip_str)

    country_code = geoip_json.get('country_code').lower()

    response += Location(
        country=geoip_json.get('country_name', 'Unknown'),
        city=geoip_json.get('city'),
```

(continues on next page)

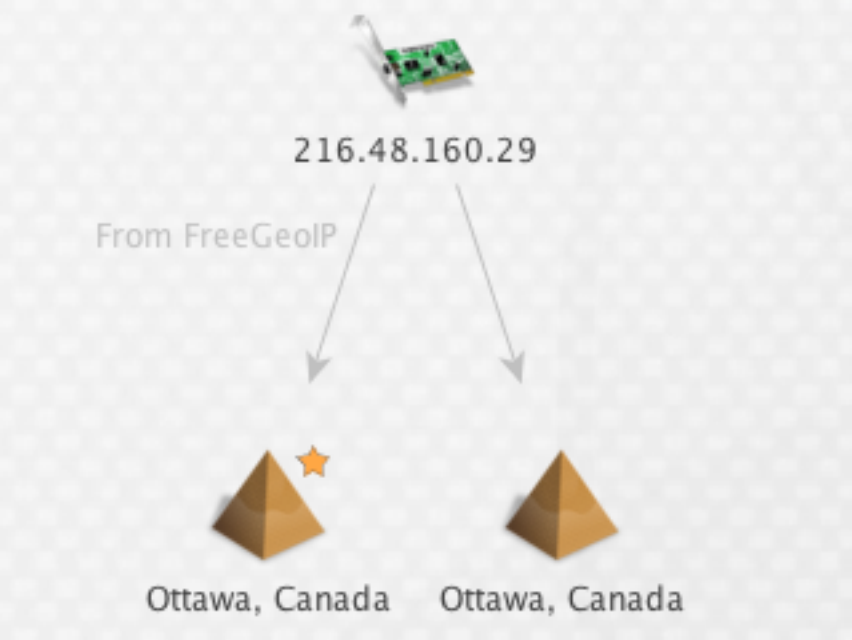

Fig. 4: Entity with link label and bookmark (left) versus undecorated entity (right)

(continued from previous page)

```
        countrycode=country_code,
        latitude=geoip_json.get('latitude'),
        longitude=geoip_json.get('longitude'),
        area=geoip_json.get('region_name'),
        link_label='From FreeGeoIP',
        bookmark=Bookmark.Orange,
        icon_url='http://www.geoips.com/assets/img/flag/256/%s.png' % country_code
    )

    return response
```

Now that we've covered the `request` and `response` parameters, let's take a look at the `config` parameter and how we can use it to make our transforms configurable.

1.3.2 Using Configuration Files

Now that you're familiar with the request and response architecture in Canari, let's make our transforms configurable. Let's assume we want to store the URL to our GeoIP API endpoint for our `IPToLocation` in a configuration file. First, let's open the `src/hello/resources/etc/hello.conf` file in a text editor. You'll notice a bunch of default values in the configuration file:

```
[hello.local]

# TODO: put local transform options here

[hello.remote]

# TODO: put remote transform options here
```

Just like an INI file in Windows, each Canari configuration file is made up of sections whose names ap-

pear within square brackets (`[, ]`), and options that appear as name-value pairs under each section header (`opt_name=opt_value`). Let's add our FreeGeoIP endpoint URL configuration option in the configuration file:

```
[hello.local]

geo_ip_url=https://freegeoip.net/json/{ip}

[hello.remote]

# TODO: put remote transform options here
```

Now let's refactor our `IPToLocation.do_transform()` code to query the configuration file for our API endpoint URL:

```
def do_transform(self, request, response, config):
    ip_address = request.entity.value

    url_template = config['hello.local.geo_ip_url'] # <------------ 1

    geoip_str = urlopen(url_template.format(ip=ip_address)).read()
    geoip_json = json.loads(geoip_str)

    country_code = geoip_json.get('country_code').lower()

    response += Location(
        country=geoip_json.get('country_name', 'Unknown'),
        city=geoip_json.get('city'),
        countrycode=country_code,
        latitude=geoip_json.get('latitude'),
        longitude=geoip_json.get('longitude'),
        area=geoip_json.get('region_name'),
        link_label='From FreeGeoIP',
        bookmark=Bookmark.Orange,
        icon_url='http://www.geoips.com/assets/img/flag/256/%s.png' % country_code
    )

    return response
```

As demonstrated, above, the `config` behaves just like a python dictionary; the keys are derived by appending the option name to the section name using a period (`.`). We've now covered all the basics for local transform development but what if we wanted to make our transforms remotely accessible?

1.3.3 Making Transforms Remote

If you're using Maltego Chlorine or later, you will probably be familiar with the Transform Hub (figure below) that appears as soon as Maltego is opened in the "Home" tab. The transform hub provides access to transforms provided by several providers. These providers operate transform application servers that host remotely accessible transforms or remote transforms.

Take a look at Paterva's documentation on how remote transforms work. As can be seen in the figure below, remote transform requests are proxied via a transform distribution server (or TDS). The TDS hosts a Maltego configuration profile that can be imported into the client via a "seed" URL. The seed URL is unique to each set of remote transforms and can be created via the web-based TDS administration console.

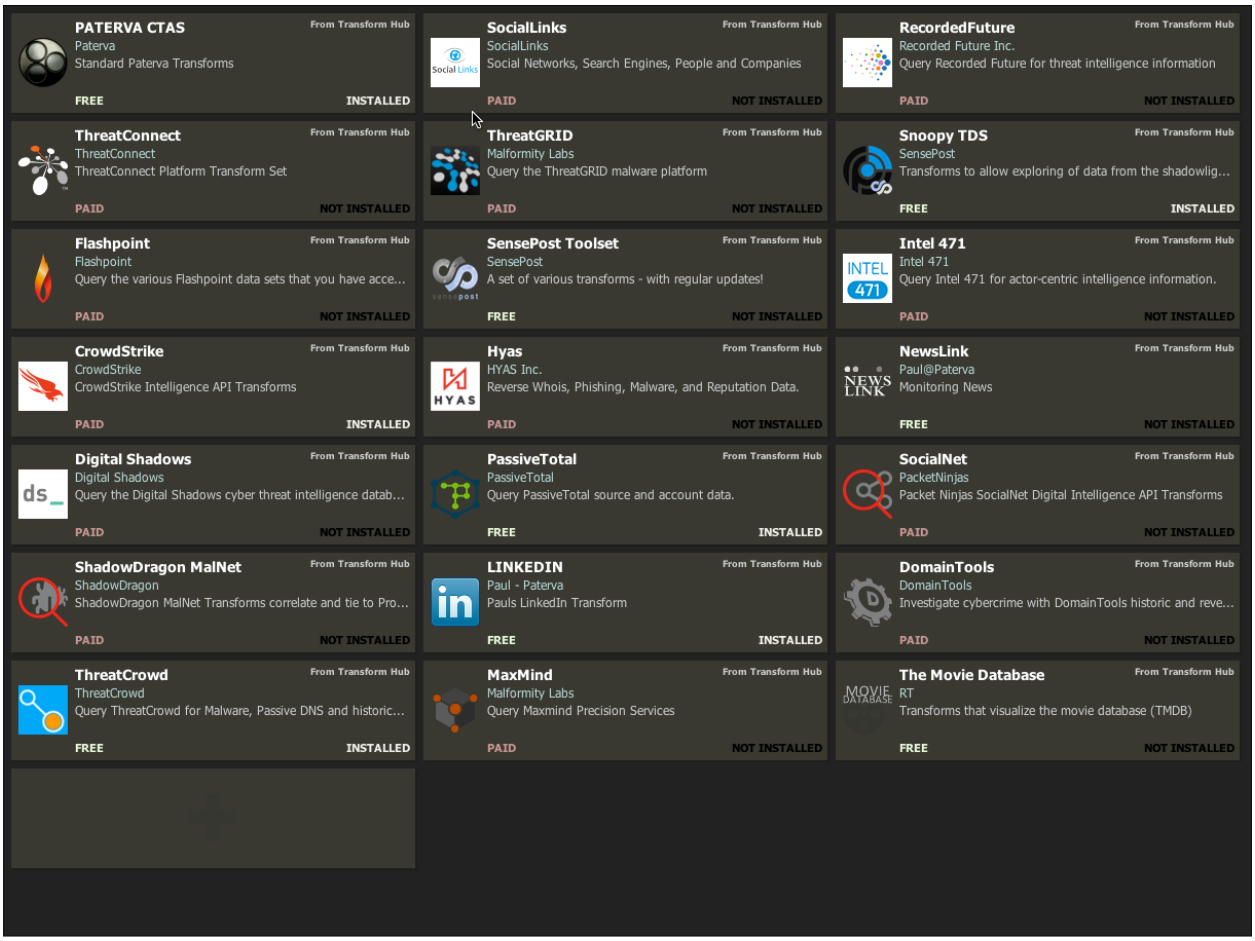

Fig. 5: Maltego Transform Hub

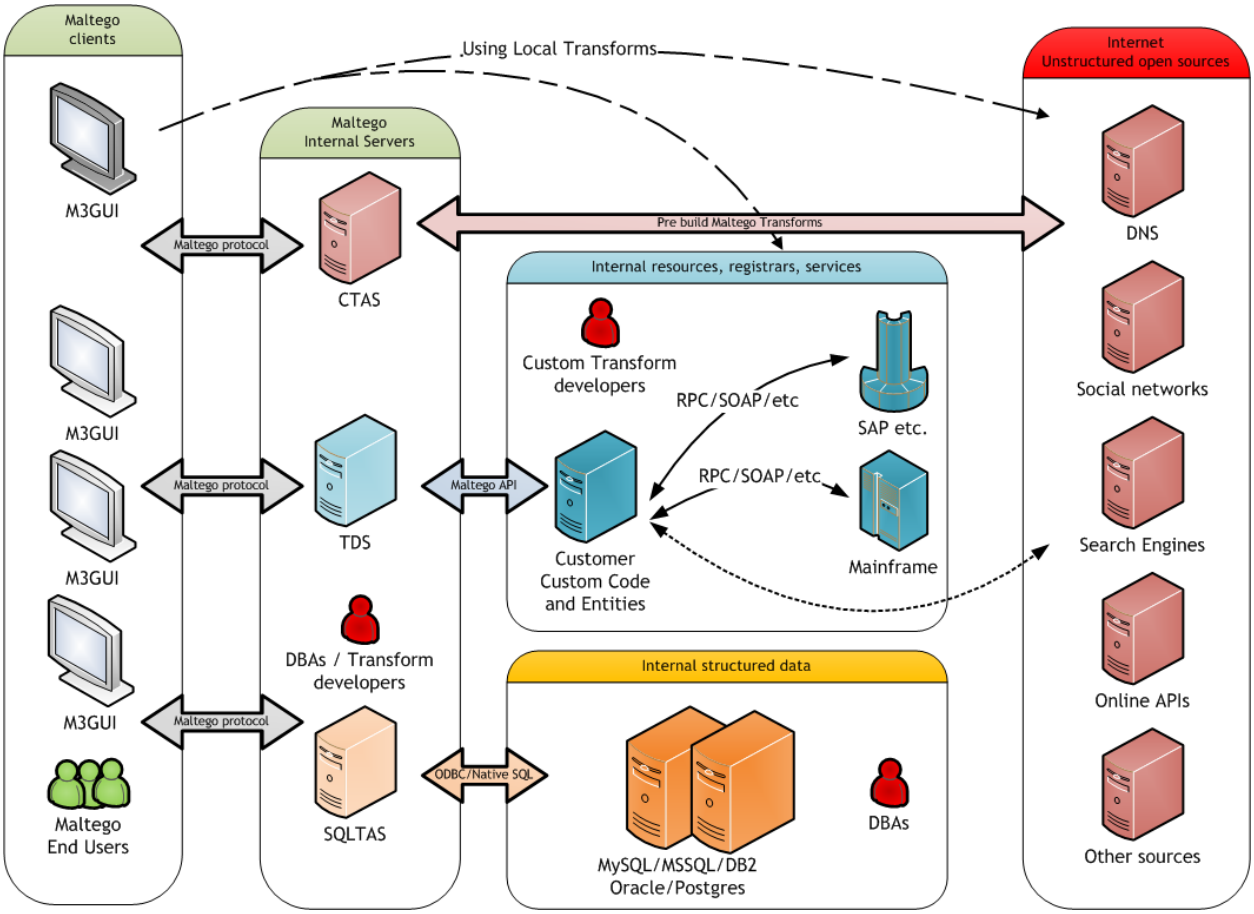

Fig. 6: Maltego TDS infrastructure.

In order to run our transforms remotely, you need to have access to a TDS. You can either buy your own TDS from Paterva if you wish to keep your data private or use their public TDS. Since we're not dealing with sensitive data in our examples, we'll use the public TDS server. Before we start, you'll need to register an account with Paterva's public TDS. Once you've registered for a free account, login to make sure you can access the console.

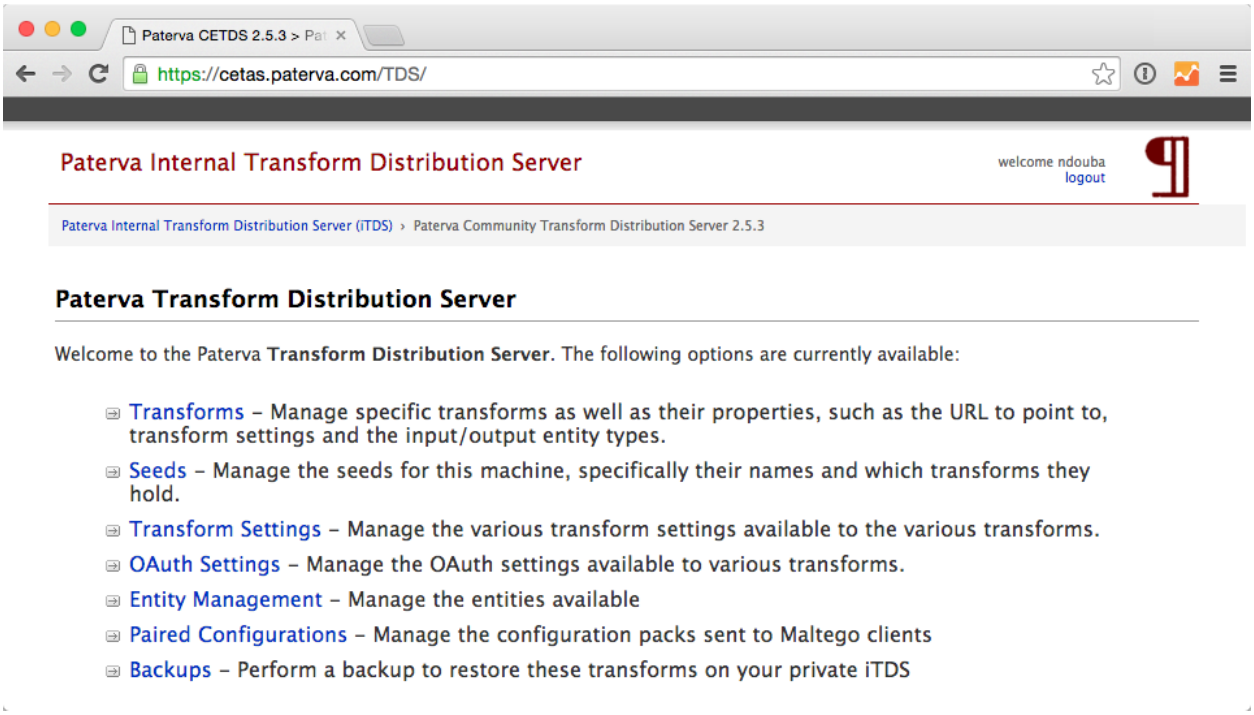

Fig. 7: Paterva TDS console

Great! Now that you're setup with a free TDS account, let's go ahead and create our first seed:

1. Click on Seeds
2. Then Add Seed
3. Leave all fields as-is and click `Add Seed` at the bottom of the form. This will save a new seed called `MySeed` that we'll populate with transforms later. Take note of the `Seed URL` for now as we'll be using it later.

Now that we've created our seed, we can now configure our remote transforms. First, we'll setup our remote transform application server, Plume, on an Internet accessible system. Plume is Canari's remote transform runner and can be used to host and execute the same transforms you wrote earlier with minor modifications to their code. Let's take our IP to location transform and make it a remote transform:

```
import json
from urllib import urlopen

from canari.framework import EnableDebugWindow
from canari.maltego.entities import IPv4Address, Location
from canari.maltego.transform import Transform


@EnableDebugWindow
class IPToLocation(Transform):
    """Get's the city/country associated with a particular IP address."""
```

(continues on next page)

(continued from previous page)

```
    # The transform input entity type.
    input_type = IPv4Address

    # Make our transform remote
    remote = True # <------------ 1

    def do_transform(self, request, response, config):
        ip_address = request.entity.value

        url_template = config['hello.local.geo_ip_url'] # <------------ 1

        geoip_str = urlopen(url_template.format(ip=ip_address)).read()
        geoip_json = json.loads(geoip_str)

        country_code = geoip_json.get('country_code').lower()

        response += Location(
            country=geoip_json.get('country_name', 'Unknown'),
            city=geoip_json.get('city'),
            countrycode=country_code,
            latitude=geoip_json.get('latitude'),
            longitude=geoip_json.get('longitude'),
            area=geoip_json.get('region_name'),
            link_label='From FreeGeoIP',
            bookmark=Bookmark.Orange,
            icon_url='http://www.geoips.com/assets/img/flag/256/%s.png' % country_code
        )

        return response
```

By simply setting the class property `remote` to `True` (1) we have now told Plume that this transform can be run remotely. Now all we have to do is install Canari, Plume, and our transform package on the Internet-accessible server. Follow the same steps to install Canari on your remote transform server as mentioned in the *Installation* section. Now archive and upload your `hello` Canari package to the server and run the **`python setup.py install`** script:

```
$ python setup.py sdist
$ scp dist/hello-1.0.tar.gz root@server:.
```

Note: Plume is only compatible with UNIX-based systems such as Linux, BSD, Darwin, etc. Windows support has not been implemented yet.

Run **`canari install-plume`** and step through the installation wizard on your server. You can simply accept all the defaults (in square brackets) by pressing enter. Here's an example of a successful Plume install:

```
server$ canari install-plume
--> What directory would you like to install the Plume init script in? [/etc/init.d]:

--> What directory would you like to use as the Plume root directory? [/var/plume]:

--> What directory would you like to save Plume logs in? [/var/log]:

--> What directory would you like to save the Plume PID file in? [/var/run]:
```

(continues on next page)

(continued from previous page)

```
--> What user would you like Plume to run as? [nobody]:

--> What group would you like Plume to run as? [nobody]:

--> What port would you like Plume to listen on? [8080]:

--> Would you like Plume to use TLS? [n]:

--> Canari has detected that you're running this install script from within a␣
↪virtualenv.
--> Would you like to run Plume from this virtualenv ('~/venvs/canari') as well? [Y/
↪n]:
Writing canari.conf to '/var/plume'...
done!
```

The Plume root directory (*/var/plume*) is where you will be running the **`canari load-plume-package`** or **`canari unload-plume-package`** commands. It's also where the *canari.conf* file for Plume will be stored as well as any static resources your transform package may rely on. Make note of the path you used for the Plume root directory as we'll be using it later:

```
server$ export PLUME_ROOT=/var/plume
```

Next, decompress your `hello-1.0.tar.gz` archive and run **`python setup.py install`** from within the `hello/` directory. At this point all our dependencies have been installed and all we need to do is configure Plume to load the Canari transform package:

```
server$ cd $PLUME_ROOT
server$ canari load-plume-package hello
Looking for transforms in hello...
Package loaded.
/var/plume/canari.conf already exists. Would you like to overwrite it? [y/N]:
Please restart plume for changes to take effect.
```

At this point, we are ready to go and all we have to do is run our init script (i.e. **`/etc/init.d/plume start`**) from the init script directory:

```
server$ /etc/init.d/plume start
Starting plume: non-SSL server
Looking for transforms in hello...
Package loaded.
Loading transform package 'hello'
Loading transform 'hello.IPToLocation' at /hello.IPToLocation...
done.
```

At this point what need to do is add our transform to our seed on the Paterva community TDS server:

1. Go back to the TDS console in your browser and login, if required.
2. Click on Transforms.
3. Click on Add Transform.
4. Set the following values:
   (a) *Transform Name* to `IPToLocation`.
   (b) *Tranform UI Display* to `IP To Location`.
   (c) *Transform URL* to `http://<server IP or hostname>:<port>/hello.IPToLocation`.

(d) Select the `Paterva Entities` radio button then `maltego.IPv4Address` from the drop-down menu under *Input Entity*.

(e) Select `MySeed` from *Available Seeds* and click the > button.

5. Finally, click *Add Transform* to add your transform to the seed.

Now for the moment of truth, copy the seed URL from the Paterva TDS console and add it to Maltego.

CHAPTER 2

canari.framework - Canari Framework Annotations & Extras

New in version 3.0.

2.1 Annotations

The Canari framework provides a few easy to use transform annotations and extras. Annotations are used to set transform class attributes. Although you can set these attributes without the use of annotations, a good reason for using them is to "pin" the presence of an important setting to the top of the class definition. For example, `@RequireSuperUser` marks the transform as a privileged transform, meaning it requires `root` privileges in order to execute. Whereas `@Deprecated` would signify that the transform has been deprecated, and so on. Annotations are applied to classes in the following manner:

```
@Deprecated
class MyTransform(Transform):
    pass
```

Canari supports the following zero argument annotations:

Annotation	Meaning
`@RequireSuperUser`	the transform requires superuser privileges in order to execute.
`@Deprecated`	the transform is deprecated.
`@EnableRemoteExecution`	the transform can be used as a remote transform.
`@EnableDebugWindow`	the transform should operate in debug mode in Maltego.

Multiple annotations can be combined to achieve the desired transform profile. For example, if we want to enable the debug window and have our user enter their superuser credentials, we would define the transform like so:

```
@RequireSuperUser
@EnableDebugWindow
```

(continues on next page)

(continued from previous page)

```
class MyTransform(Transform):
    pass
```

2.1.1 `@RequireSuperUser` Behavior

As stated above, the `@RequireSuperUser` marks the transform as a privileged transform which requires `root` privileges in order to execute. The behavior of the different Canari transform runners differs when they come across a transform that has this attribute set. When using **`canari run-transform`**, **`canari debug-transform`**, **`dispatcher`**, a graphical password dialog box will appear prompting the user to enter their sudo credentials.

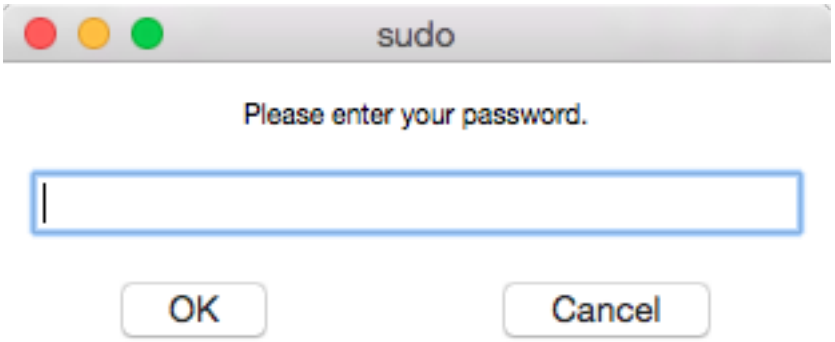

Fig. 1: Canari sudo dialog box

When using the **`canari shell`** or **`plume`** (transform application server), Canari will ask whether or not the user wishes to load superuser transforms. If the user chooses to load superuser transforms, Canari will rerun the user's command using sudo, which will prompt the user for their credentials in the terminal. Here's an example of what the sample output would look like:

```
$ canari shell foo
Looking for transforms in foo...
Package loaded.
A transform requiring 'root' access was detected. Would you like to run this shell as
↪'root'? [y/N]: y
Need to be root to run this transform... sudo'ing...
Password:
Looking for transforms in foo...
Package loaded.
Welcome to Canari 3.0.
>>>
```

2.1.2 `@EnableDebugWindow` Behavior

The `@EnableDebugWindow` annotation instructs the **`canari create-profile`** to create a transform profile that forces Maltego to display the "Debug" output pane each time the transform is executed. This is useful for debugging the response XML or viewing debug or diagnostic messages being written to `stderr`.

Note: Messages written to `stderr` during transform execution will only be seen if transforms are executed locally. Remote transforms do not transmit this information.

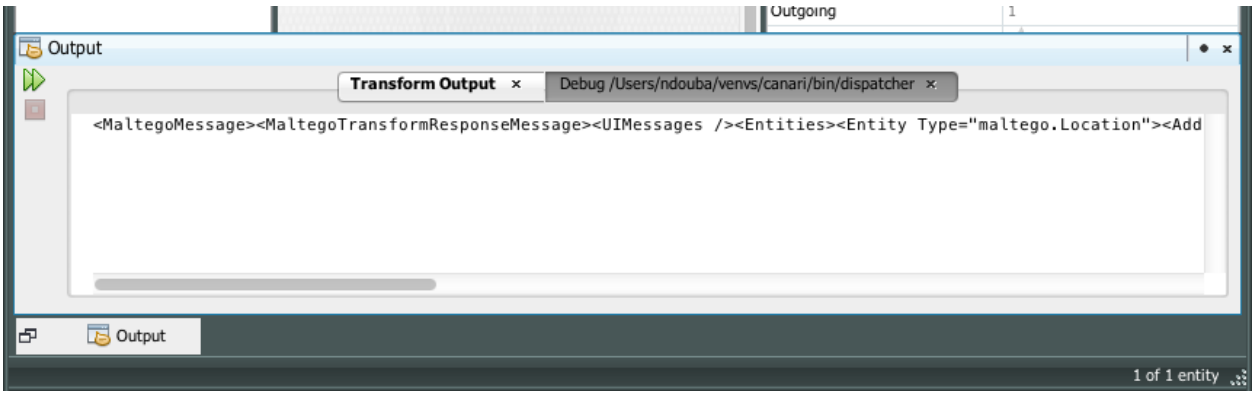

Fig. 2: Maltego "Debug" pane

2.1.3 Request Filtering with `@RequestFilter`

Canari also supports the concept of transform request filtering by applying the `@RequestFilter` annotation to a transform. This is especially useful for use-cases where a license or authorization check needs to be performed prior to transform execution. For example, let's assume that you want to check whether a user is submitting a valid license key prior to executing a transform request:

```
def check_api_key(request, response, config):
    if not request.parameters['my_api.key'] == 'cool':
        raise MaltegoException('Invalid license key!')

@RequestFilter(check_api_key)
class MyTransform(Transform):
    def do_transform(request, response, config):
        # TODO: something cool
        return response
```

The *RequestFilter* annotation accepts the following arguments in its constructor:

class canari.framework.**RequestFilter**(*filter_*[, *remote_only=False*])

Parameters

- **filter** (*callable*) – a callable that accepts three arguments: `request`, `response`, and `config`.
- **remote_only** (*bool*) – `True` if the filter should only be applied when the transform is operating in remote mode, otherwise `False` (default).

Just like the `Transform.do_transform()` method, request filters can also modify the contents of the `request`, `response`, and `config` objects and raise exceptions, if necessary, to interrupt transform execution. Request filters are expected to return either `True`, to cancel transform execution, or `False`, to allow the transform to continue executing. The following example illustrates how you can gracefully interrupt the execution of a transform and communicate the reason via a Maltego UI message:

```
count = 0

def check_access_count(request, response, config):
    global count
    if count == 500:
        response += UIMessage('Access denied: you have reached your limit.')
```

(continues on next page)

(continued from previous page)

```
            return True
        count += 1
```

In the example above, our request filter is keeping track of a global request counter. If that counter reaches its threshold, any subsequent transform requests will be cancelled and the user will be informed of the reason via a Maltego informational UI message. Otherwise, the counter is incremented and transform execution continues.

Attention: When a request filter returns `True` the current transform request will be cancelled and the empty or modified transform `response` object will be returned.

Request filtering can also be enabled for remote transforms only. This makes it easier to develop the transform locally, void of any request filtering checks, by setting the *remote_only* keyword argument to `True` in your `RequestFilter` annotation, like so:

```
def check_api_key(request, response, config):
    if not request.parameters['my_api.key'] == 'cool':
        raise MaltegoException('Invalid license key!')

@RequestFilter(check_api_key, remote_only=True)
class MyTransform(Transform):
    def do_transform(request, response, config):
        # TODO: something cool
        return response
```

In the example above, `check_api_key()` will only be called if `MyTransform` is running in Plume.

2.2 Foreign Language Support

If you've already developed your cool Maltego transforms in another language, such as Perl, Ruby, or Java, you can still take advantage of Canari's powerful transform packaging and distribution features. By setting `Transform.do_transform` to an instance of *ExternalCommand* and placing your transform code in the `<project name>/src/<project name>/resources/external` directory, you have the ability to run transform code written in other languages. For example, let's assume we've written a Perl transform, `do_nothing.pl`, that we'd like to package and distribute using the Canari framework:

```
#!/usr/bin/perl

print "<MaltegoMessage><MaltegoTransformResponseMessage/></MaltegoMessage>\n";
```

First, you'll have to create a transform:

```
class DoNothing(Transform):
    do_transform = ExternalCommand('perl', 'do_nothing.pl')
```

Finally, you'll have to place the *do_nothing.pl* file in your `<project name>/src/<project name>/resources/external` directory (i.e. `foo/src/foo/resources/external`).

See also:

Canari development quick-start guide for information on how to create a transform package and write transform code.

The *ExternalCommand* constructor accepts the following arguments:

class canari.framework.**ExternalCommand**(*interpreter*, *program*[, *args=None*])

Parameters

- **interpreter** (*str*) – the name of the program interpreter (i.e. **perl**, **java**, **ruby**, etc.)
- **program** (*str*) – the name of the transform script or executable file.
- **args** (*iterable*) – an optional list of arguments to pass to the transform executable or script.

In the event that `interpreter` is either *perl*, *ruby*, or *java*, the appropriate flags will be set to include the `<project name>/src/<project name>/resources/external` directory as part of the default module or class search path. This is done to support relative module or library imports without having to modify your pre-existing code.

CHAPTER 3

canari.maltego.message - Maltego Messaging Objects

New in version 3.0.

The *canari.maltego.message* module provides the complete implementation of all the Maltego transform messaging objects. These objects are used to deserialize Maltego transform requests and serialize Canari transform responses for both local and remote transforms. For example, the `request` and `response` objects that gets passed into the `Transform.do_transform()` method are instances of `MaltegoTransformRequest` and `MaltegoTransformResponse`, respectively.

All Maltego messaging objects are subclasses of the `MaltegoElement` super class, which adds support for two arithmetic operations:

Operations	Meaning
`p += c`	Add a child object (`c`) to the parent object (`p`)
`p + c`	Same as += but it can be chained with multiple child objects.

Here's an example demonstrating the use of these two arithmetic operations on the `response` object:

```
from canari.maltego.transform import Transform
from canari.maltego.entities import Phrase, Person

class HelloWorld(Transform):

    input_type = Person

    def do_transform(self, request, response, config):
        person = request.entity
        response += Phrase('Hello %s!' % person.value)
        response = response + Phrase('Hello Mr(s) %s!' % person.lastname) \
                            + Phrase('Hello %s!' + person.firstname)
        return response
```

Finally, each messaging object can be separately serialized and deserialized to and from XML using the `render()` and `parse()` methods:

```
>>> from canari.maltego.entities import Phrase
>>> print (MaltegoTransformResponseMessage() + Phrase('test')).render(pretty=True)
<?xml version="1.0" ?>
<MaltegoTransformResponseMessage>
    <UIMessages/>
    <Entities>
        <Entity Type="maltego.Phrase">
            <Value>test</Value>
            <Weight>1</Weight>
        </Entity>
    </Entities>
</MaltegoTransformResponseMessage>
>>> MaltegoTransformResponseMessage.parse('<MaltegoTransformResponseMessage/>')
<canari.maltego.message.MaltegoTransformResponseMessage object at 0x10e99e150>
```

However, if you're a transform developer you will never really need to use the `render()` or `parse()` methods as they are primarily used by the **`dispatcher`**, **`canari debug-transform`**, and **`plume`** transform runners.

3.1 Maltego Request and Response Objects

The *`MaltegoTransformRequestMessage`* and *`MaltegoTransformResponseMessage`* represent the parent container for Maltego request and response messages, respectively. When a transform is executed, Canari automatically deserializes a request into a *`MaltegoTransformRequestMessage`* object and creates an empty *`MaltegoTransformResponseMessage`*, which it then passes to `Transform.do_transform()`.

Maltego transform request messages can be created using either the factory method `parse()`, which accepts an XML string whose root element is the `<MaltegoTransformRequestMessage>` tag, or by calling the empty constructor.

`class` canari.maltego.message.**`MaltegoTransformRequestMessage`**(***kwarg*)

Return a new Maltego transform request message with no child elements. Each Maltego transform request message comes with the following read-only attributes:

`limits`

A `Limits` object which contains the soft and hard limits for the number of entities Maltego would like returned.

One can access the soft and hard limits of a `request` object by doing the following:

```
>>> print 'Transform hard limit=%s, soft limit=%s' % (request.limits.soft,␣
↪request.limits.hard)
Transform hard limit=500, soft limit=5000
```

Note: *`limits`* do not apply to local transforms since the local transform adapter in Maltego does not transmit this information.

`parameters`

In **local transform execution mode**, *`parameters`* is a list of extraneous command line arguments not handled by the Canari **`dispatcher`**. This is useful in scenarios where you want to use command line arguments to manage the behavior of a transform, for example:

```
# transform executed using 'dispatcher foo.transforms.HelloWorld -u Bob'
def do_transform(self, request, response, config):
```

(continues on next page)

(continued from previous page)

```
    """If '-u' detected in command line arguments make entity value all upper␣
↪case."""
    if '-u' in request.parameters:
        response += Phrase('Hello %s!' + request.entity.value.upper())
    else:
        response += Phrase('Hello %s!' + request.entity.value)
    return response
```

In **remote transform execution mode**, *parameters* is a dictionary of additional transform fields, keyed by their names. Transform fields are typically used to communicate additional transform parameters. For example, many commercial transforms use the transform field to transmit API keys. Alternatively, one can use transform fields to alter transform behaviour - just like in our local mode example. The following is an example of a custom transform that expects an API key:

```
# ...
def do_transform(self, request, response, config):
    fields = request.parameters
    if 'my.license' not in fields or not valid_api_key(fields['my.license'].
↪value):
        raise MaltegoException('Invalid API key! Send cheque!', code=600Å)
    response += Phrase('Hello %s!' + request.entity.value)
    return response
```

Note: If you intend to use a transform package in both local and remote mode, make sure to check Canari's operating mode prior to accessing *parameters*. See *canari.mode* for more information.

entity

The *Entity* object to be processed by the Canari transform. The entity object's type is determined by the value of the `Transform.input_type` attribute. If *Transform.input_type* is not set explicitly, then *entity* will return an entity of type *Unknown*. For example, a *Person* entity will always be returned in the following transform:

```
class HelloWorld(Transform):
    # Ensure request.entity returns a Person object
    input_type = Person

    def do_transform(self, request, response, config):
        person = request.entity
        response += Phrase('Hello %s!' + person.fullname)
        return response
```

MaltegoTransformResponseMessage can be created in the same way as our request objects; either by using `parse()` or by using the constructor explicitly.

class canari.maltego.message.**MaltegoTransformResponseMessage**(***kwarg*)

Return a new Maltego transform response message object with no child elements. The various attributes of the response can also be manipulated using regular list operations via these attributes:

messages

A list of *UIMessage* objects that contain user interface messages to be displayed in Maltego's "Transform Output" pane or in a dialog window. For example, let's say we wanted to display a fatal message:

```
# ...
def do_transform(self, request, response, config):
```

(continues on next page)

(continued from previous page)

```
    response += UIMessage("This transform is not implemented yet!",␣
↪type=UIMessageType.Fatal)
    return response
```

This would result in the following message box appearing in Maltego:

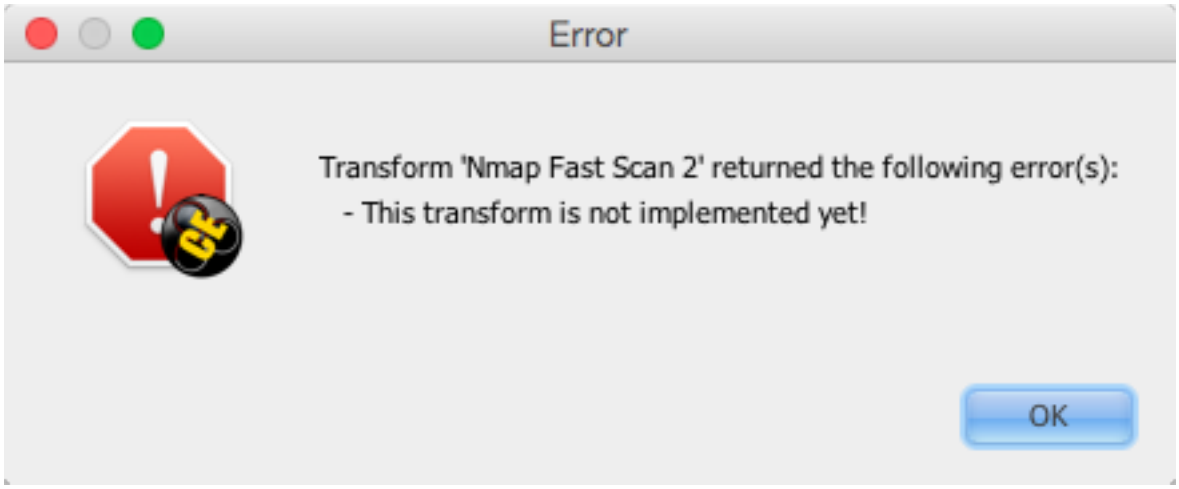

Fig. 1: Fatal UI message appearance

See also:

UIMessage for an overview of the different message types and how they are rendered in Maltego's UI.

entities

The list of *Entity* objects to be returned as transform results to the Maltego UI. Entities can be added to a response message by using the += operator, like so:

```
# ...
def do_transform(self, request, response, config):
    response += Location('Brooklyn')
    return response
```

Or by using the + operator to chain multiple entity results in one line, like so:

```
# ...
def do_transform(self, request, response, config):
    return (response + Location('Brooklyn') + Location('Broadway'))
```

3.2 Communicating Exceptions

Using `MaltegoExceptionResponseMessage` objects, a transform can communicate an error state back to the Maltego user. Canari generates a Maltego exception object if an exception is raised during transform execution. There are two different behaviours when it comes to reporting exceptions. If a transform raises a `MaltegoException` then the exception message is what's communicated to the user. However, other exception types will render a message box with full stack trace details. Here's a visual example:

```
# ...
def do_transform(self, request, response, config):
    raise MaltegoException('Just pooped!')
```

Results in the following dialog box:

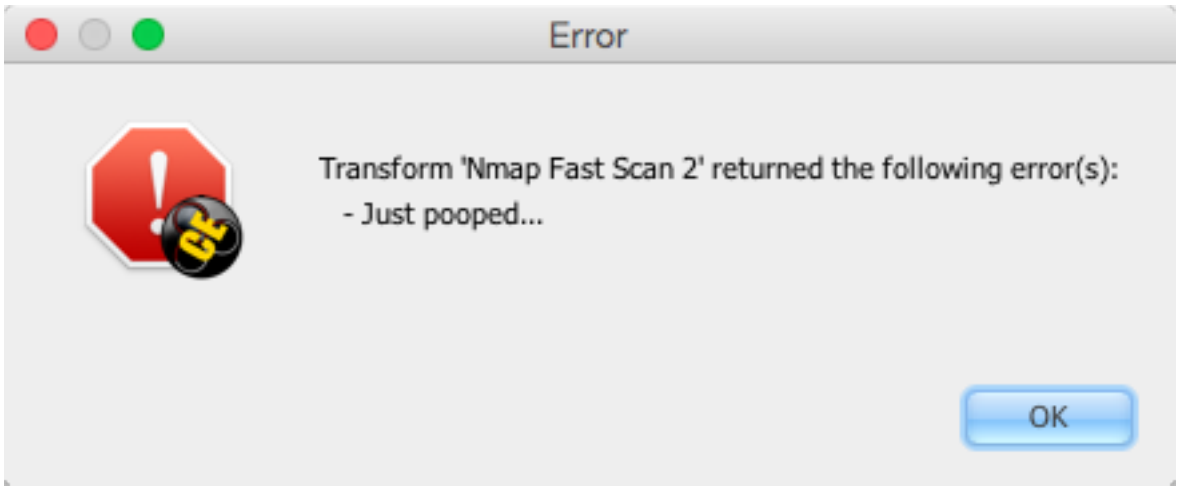

Fig. 2: `MaltegoException` exception appearance

Whereas:

```
# ...
def do_transform(self, request, response, config):
    import foobar # non-existent module
```

Results in the following dialog box:

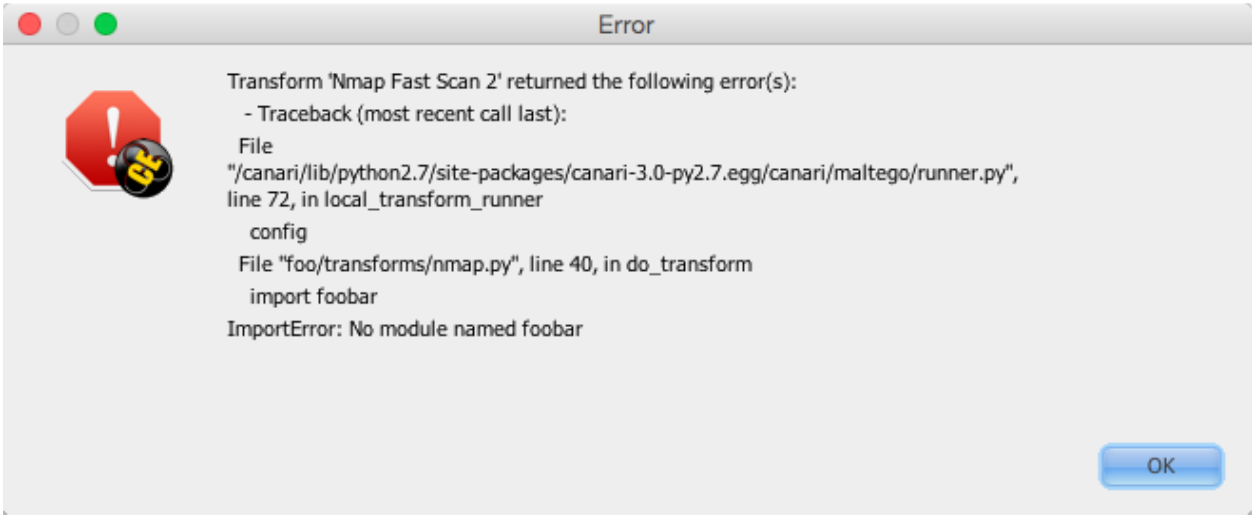

Fig. 3: Non-`MaltegoException` exception appearance

Warning: Users who are security conscious may find this behaviour undesirable since full stack traces often disclose internal information such as file system paths, and module names. Support for cross-referencable logs and generic error messaging will appear in Canari v3.1.

3.3 Communicating Diagnostic Information

A second form of communicating status or diagnostic information is via the use of *UIMessage* objects. UI messages either appear in the "Transform Output" pane (usually at the bottom) or as dialog message boxes depending on the message type assigned to them. For your convenience, Canari has defined all the different UI message types as class attributes in *UIMessageType*:

class canari.maltego.message.**UIMessageType**

Fatal
Fatal errors are communicated to Maltego users using a dialog message box.

Partial
Partial errors are communicated to Maltego users in the "Transform Output" pane and are orange in color.

Inform
Informational errors are communicated to Maltego users in the "Transform Output" pane but are not colored.

Debug
These errors do not appear to be displayed anywhere in the Maltego user interface. Instead they may appear in debug logs.

Communicating diagnostic information to a Maltego user is simple. Simply, use the += or + operators to add a *UIMessage* object to a response object, like so:

```
# ...
def do_transform(self, request, response, config):
    import time
    response += Phrase('Hello sleepy head!')
    time.sleep(3)
    response += UIMessage("This transform took 3 seconds to complete.",␣
↪type=UIMessageType.Inform)
    return response
```

The *UIMessage* accepts two arguments, `msg` and `type`.

class canari.maltego.message.**UIMessage**(*message*[, *type=UIMessageType.Inform*])

Parameters

- **message** (*str*) – The message to communicate to the Maltego user.
- **type** (UIMessageType) – The type of message to communicate to the user (default: *UIMessageType.Inform*).

Values for `message` and `type` can also be set via these attributes:

type
The type of message that will be communicated. Valid values for this attribute are defined in *UIMessageType*.

message
The message to communicate to the user.

Local transforms also support real-time diagnostic messaging. See `debug()` and `progress()` for more information.

3.4 Using and Defining Maltego Entities

An entity in Maltego is comprised of several elements:

1. **A default entity value:** the default property which appears under the Maltego entity on the graph.
2. **Fields:** extra properties belonging to an entity that get passed to transforms as input. These appear in the "Properties View" pane in Maltego. The default entity value is also represented as a property.
3. **Labels:** read-only information that's used to display additional information about an entity. Labels do not get used as transform input. Labels appear in the "Detail View" pane in Maltego.
4. **Notes:** additional notes that are associated with a particular entity. Like labels, notes are not used as transform input. Notes can be viewed in the "Entity Viewer" or on the graph as a call-out.
5. **Link and Entity Decorations:** usually set by a transform on all its output entities to decorate the look and feel of a link (i.e. line thickness, style, etc.) or entity (i.e. bookmarking, icons, etc.). Decorations appear directly on the graph.

Fig. 4: Maltego entity composition

Canari uses the *Entity* type to define Maltego entities programmatically. All Canari entities are subclasses of the *Entity* type. *Entity* objects are used in both request and response messages. Canari comes with a list of

pre-defined entity types that correspond to the built-in types in Maltego. These types can be found in *canari.maltego.entities*. Defining a custom entity in Canari is as simple as this:

```
>>> from canari.maltego.message import Entity, StringEntityField
>>> class Threat(Entity):
...     name = StringEntityField('threat.name', is_value=True)
...     country = StringEntityField('threat.country')
...
>>> t = Threat('Cheese', country='Switzerland')
>>> print 'Detected threat %r from %s' % (t.name, t.country)
Detected threat 'Cheese' from Switzerland.
```

In the example above we are defining a custom entity of type `Threat` with two string entity fields, `name` and `country`. The `is_value` keyword argument in our `name` entity field definition instructs Canari that `name` is the entity's default value. As a result, we can set the value of `name` via the entity's first argument in the constructor. Alternatively, we could have completely omitted the definition of `name` since all entity objects have an entity `value` attribute. All other entity fields can be set using a keyword argument that matches the attribute's name.

Entity objects can be instantiated in the following manner:

class canari.maltego.message.**Entity**(*value=''*, ***kwarg*)

Parameters **value** (*str*) – the default entity field value.

You can also pass the following additional keyword arguments:

Parameters

- **type** (*str*) – The entity's type name (default: `<package name>.<class name>`).
- **value** (*str*) – The entity's default entity field value.
- **weight** (*float*) – The entity's weight value from 0.0 to 1.0. Useful for transforms that return ranked search result entities from search engines.
- **icon_url** (*str*) – The entity's icon URL. Maltego supports the built-in Java URL protocol schemes (`file://`, `http://`, `https://`, etc.).
- **fields** (*list*) – A list of entity fields, of type *Field*, to be added to the entity.
- **labels** (*list*) – A list of entity labels, of type *Label*, to be added to the entity.

The following attributes are also inherited by all the subclasses of the *Entity* type:

value
The default entity value (what appears under the entity's icon in the Maltego graph) as a string.

icon_url
A string containing a valid URL to an image (i.e. `file:///tmp/foo.png`, `http://bar.com/foo.gif`, etc.) to be used as the entity's icon.

Fig. 5: Maltego entity icon

labels
A dictionary of *Label* objects keyed by their names. Labels appear in the "Detail View" pane in the Maltego GUI and are often used to display text fragments, additional information, or hyperlinks that a user can click on for more information.

Note: Labels are not transmitted with input entities on transform requests. If you wish to include information from a label in a transform request, then that information should reside in an entity field.

Adding a label to an entity is as easy using the += operator or passing a list of *Label* objects to the entity constructor, like so:

```
>>> t = Threat('Cheese', country='Switzerland', labels=[Label('Cheese Type',
↪'Swiss')])
>>> t += Label('Cheese Age', '12 years')
```

Which would result in the following content being rendered in the "Detail View" pane in Maltego's UI:

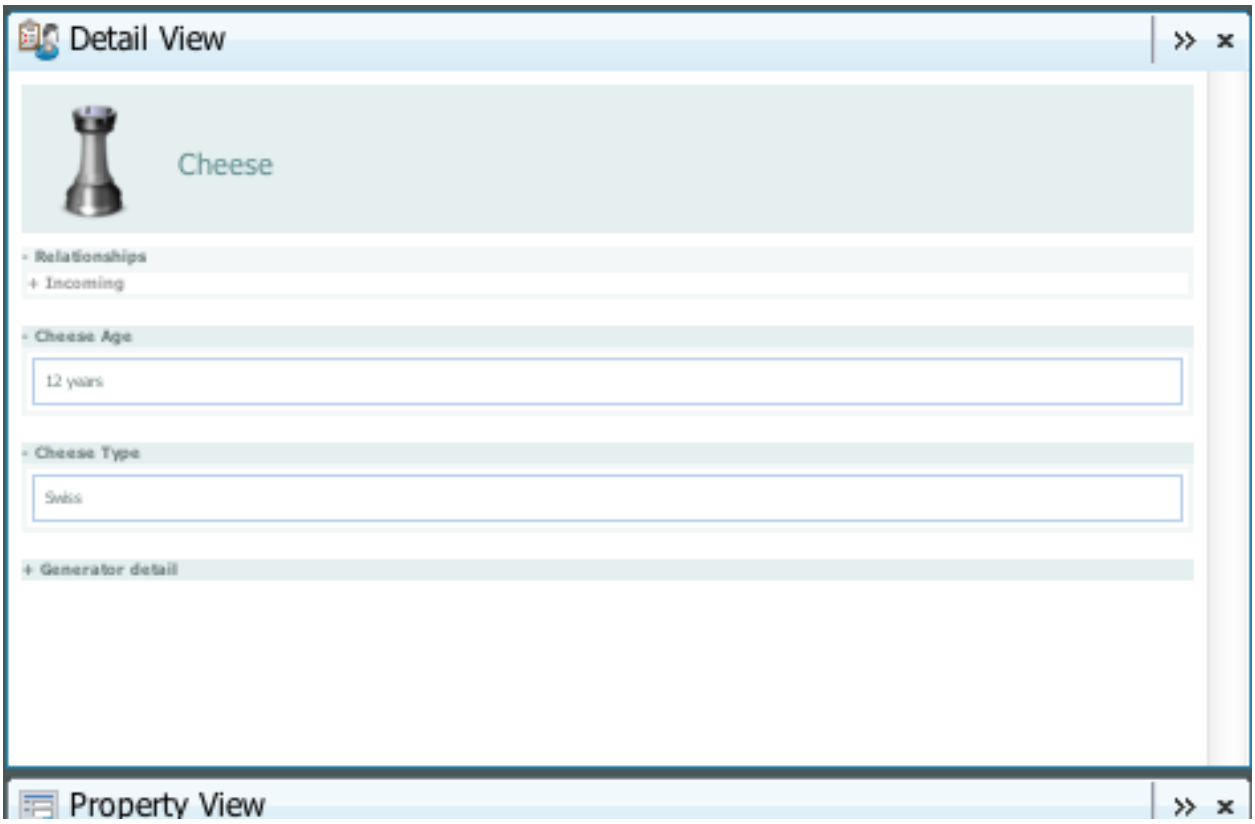

Fig. 6: Entity labels in "Detail View"

fields
A dictionary of *Field* objects keyed by their names. Entity fields are read-write properties that appear in the "Properties View" pane in the Maltego GUI and are used as input for transform requests.

notes
A string containing additional notes that can be attached to a Maltego entity. You can set a note in the following manner:

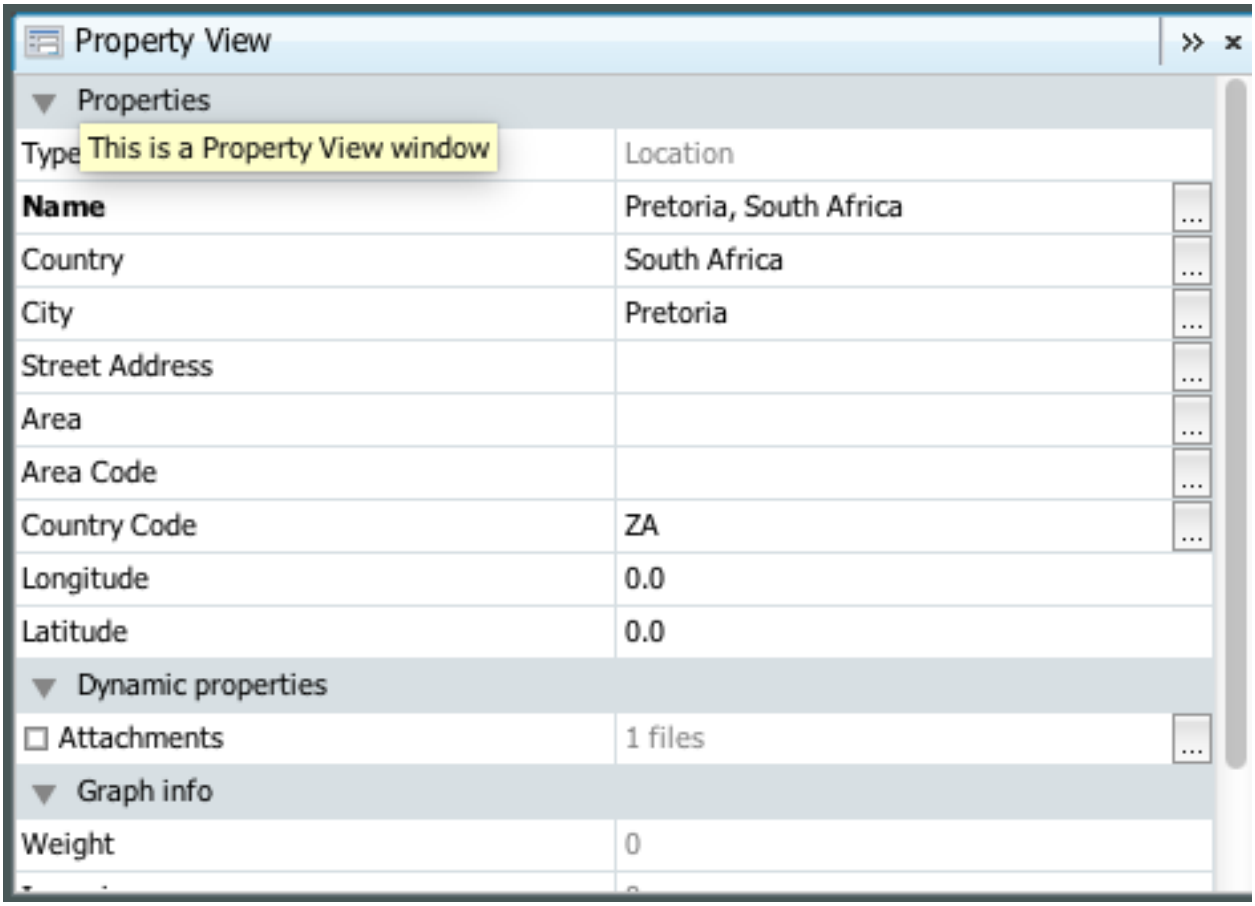

Fig. 7: Entity fields in "Properties View"

```
>>> Threat('Cheese', country='Switzerland', note='This is a note') # or
>>> t = Threat('Wine', country='Italy')
>>> t.note = 'This is another note'
```

The following figure demonstrates the appearance of an entity note in Maltego:

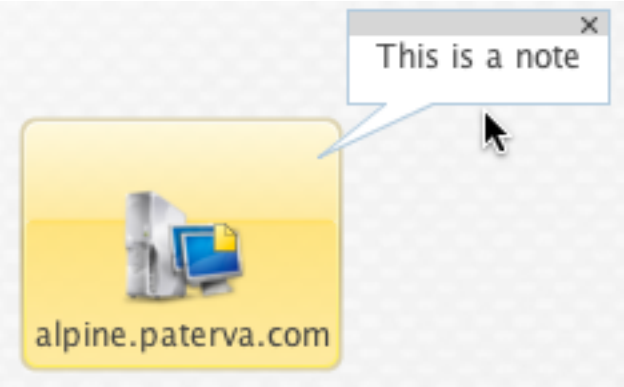

Fig. 8: Maltego Entity Note

Note: Entity notes are not transmitted as transform input. Consider adding an additional entity field that encapsulates the information in your notes if you wish to pass it to your transforms as input.

bookmark
Determines whether an entity should be marked with a colored star. Can be one of the following values:

Value	Appearance
`Bookmark.NoColor`	**(default)**
`Bookmark.Cyan`	
`Bookmark.Green`	
`Bookmark.Yellow`	
`Bookmark.Orange`	
`Bookmark.Red`	

Here's an example of how to set a bookmark:

```
>>> from canari.maltego.message import Bookmark
>>> Threat('Cheese', country='Switzerland', bookmark=Bookmark.Red) # or
>>> t = Threat('Wine', country='Italy')
>>> t.bookmark = Bookmark.Cyan
```

The following figure demonstrates the appearance of an entity bookmark in Maltego:

link_label
A string attribute that adds a label to the link that connects the parent and child entity. Like notes, link labels

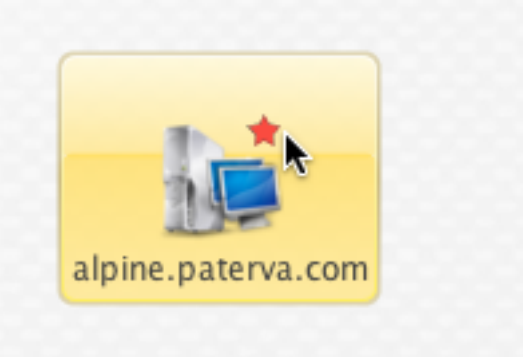

Fig. 9: Maltego entity bookmark

can be set via the `link_label` keyword argument in the constructor or by accessing the `link_label` attribute. Here's an example of the link label in action:

```
# ...
def do_transform(self, request, response, config):
    return (response + IPv4Address('74.207.243.85', link_label='This is a␣
↪link label'))
```

This is what it would look like in Maltego:

Fig. 10: Link label appearance

Link labels can be shown or hidden by setting the *`link_show_label`*.

`link_show_label`
Determines whether or not the link label will be shown based on the following values:

Value	Meaning
`LinkLabel.UseGlobalSetting`	The visibility of the link label will depend on the global setting.
`LinkLabel.Show`	The link label will be visible on the graph.
`LinkLabel.Hide`	The link label value will be set but will not be visible on the graph.

The global setting can be found under the "View" ribbon within the "Links" settings group.

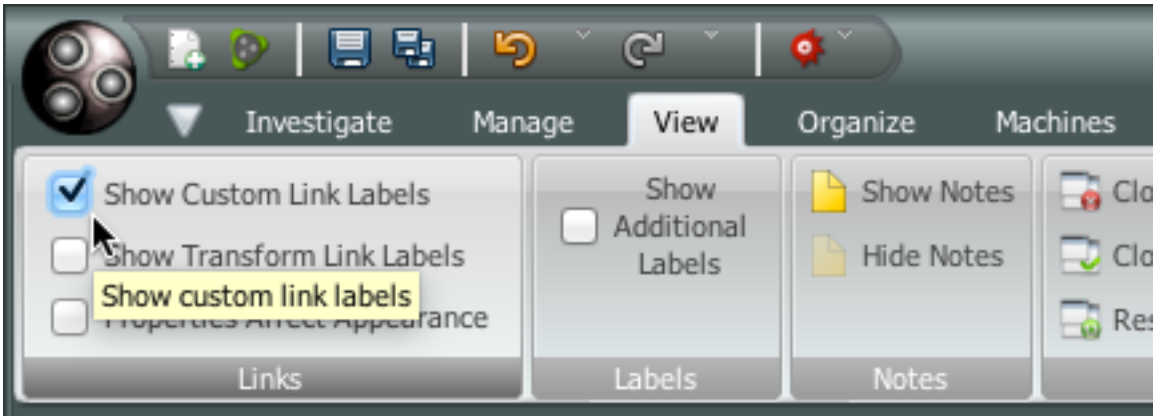

Fig. 11: Maltego global link label visibility setting

Here's an example of the link visibility setting in action:

```
from canari.maltego.message import LinkLabel
# ...
def do_transform(self, request, response, config):
    return (response + IPv4Address('74.207.243.85', link_show_label=LinkLabel.
↪Hide))
```

`link_style`

Dictates the appearance of the link's line, which can be one of the following choices:

Value	Appearance
`LinkStyle.Normal`	**(default)**
`LinkStyle.Dashed`	
`LinkStyle.Dotted`	
`LinkStyle.DashDot`	

Here's an example of the link style in action:

```
from canari.maltego.message import LinkStyle
# ...
def do_transform(self, request, response, config):
    return (response + IPv4Address('74.207.243.85', link_style=LinkStyle.
↪DashDot))
```

This is what it would look like in Maltego:

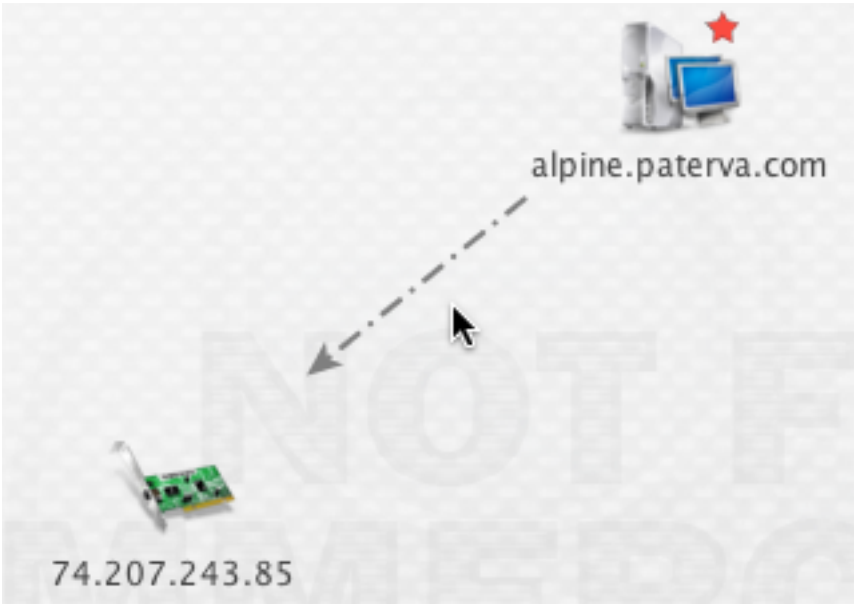

Fig. 12: Link style appearance

link_color

Dictates the color of the link connecting the parent and child entities. The link color is limited to the following values:

Value	Appearance
`LinkColor.Black`	
`LinkColor.DarkGray`	**(default)**
`LinkColor.LightGray`	
`LinkColor.Red`	
`LinkColor.Orange`	
`LinkColor.DarkGreen`	
`LinkColor.NavyBlue`	
`LinkColor.Magenta`	
`LinkColor.Cyan`	
`LinkColor.Lime`	
`LinkColor.Yellow`	
`LinkColor.Pink`	

Here's an example of the link color in action:

```
from canari.maltego.message import LinkColor
# ...
def do_transform(self, request, response, config):
    return (response + IPv4Address('74.207.243.85', link_color=LinkColor.Red))
```

This is what it would look like in Maltego:

Fig. 13: Maltego link color

link_thickness

Dictates the thickness of the link connecting the parent and child entities. Valid values range from `0` to `5`. The greater the number, the thicker the link and vice versa. Here's an example of the link thickness in action:

```
# ...
def do_transform(self, request, response, config):
    return (response + IPv4Address('74.207.243.85', link_thickness=5))
```

This is what it would look like in Maltego:

3.4.1 Defining Entity Fields

Entity fields can be added to an entity in two ways, dynamically and statically. The dynamic approach is recommended in cases where an entity field is not part of the standard entity's definition. For example, if we wanted to attach an additional field, "Tracking Code", to Maltego's built-in `WebSite` entity, we can do so like this:

```
>>> from canari.maltego.entities import WebSite
>>> w = WebSite('www.google.com')
>>> w += Field('tracking.code', '123456789', display_name='Tracking Code')
```

However, if we were looking to use the pre-defined entity fields, defined for a particular entity, we can simplify our code by defining entity field properties in Canari. Entity field properties provide a simple and clean interface to get

Fig. 14: Maltego link thickness

and set values of entity fields in a Pythonic way:

```
>>> from canari.maltego.message import *
>>> class MyEntity(Entity):
...     foo = StringEntityField('foo')
...
>>> e = MyEntity()
>>> e.foo = 1
>>> e.foo
'1'
>>> MyEntity(foo=2).foo
'2'
```

See also:

See *Field* for more information on constructing dynamic fields.

Canari comes with 11 pre-defined entity field types to aid with entity object interaction in your transforms. These entity field types take care of marshalling field data into the appropriate type (i.e. from string to integer, or float, etc.). This is useful for performing rudimentary input validation in your transforms and can ensure that the data is properly formatted for Maltego as well. For example, the *EnumEntityField* can be used to ensure that an entity field's value is restricted to a limited set of acceptable values:

```
>>> class Car(Entity):
...     fuel = EnumEntityField('car.fuel.type', choices=['diesel', 'petroleum'])
...
>>> toyota = Car(fuel='diesel')
>>> volvo = Car(fuel='water')
Traceback (most recent call last):
    ...
ValidationError: Invalid value ('water') set for field 'car.fuel.type'. Expected one␣
↪of these values: ['diesel', 'petroleum'].
```

All entity field types with exception to the *StringEntityField* raise a `ValidationError` if an invalid value is set. Input validation is also performed when retrieving field values from input entities as well.

Note: Input validation is not immediately performed on input entity fields. Instead, input validation checks are performed when a transform attempts to access a strong-typed input entity field.

Validation errors appear in a user friendly manner within the Maltego GUI, like so:

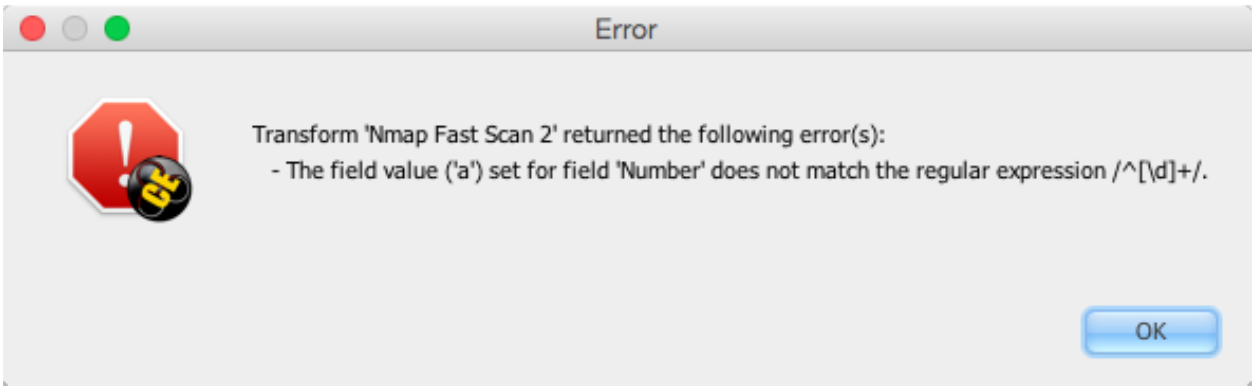

Fig. 15: Input validation error

The following entity field types are supported in Canari:

Entity Field Type	Accepted Types
StringEntityField	All (if not `str`, then result of `object.__str__()` is used).
EnumEntityField	Values defined in list of `choices`.
`IntegerEntityField`	`int`
`BooleanEntityField`	`bool`
`FloatEntityField`	`float`
`LongEntityField`	`long`
`DateTimeEntityField`	`datetime`
`DateEntityField`	`date`
`TimeSpanEntityField`	`timedelta`
RegexEntityField	`str` that contains a regex pattern used in `re.match()`.
`ColorEntityField`	`str` that contains RGB color code (i.e. `'#FF0000'`)

All entity field types are subclasses of *StringEntityField* and can be constructed in the following manner:

class canari.maltego.message.**StringEntityField**(*name*, ***extras*)

Parameters **name** (*str*) – The "Unique property name" of the entity field in Maltego.

The constructor also accepts the following keyword arguments:

Parameters

- **description** (*str*) – The "Description" of the entity field in Maltego.
- **display_name** (*str*) – The "Property display name" of the entity field in Maltego.
- **matching_rule** (*MatchingRule*) – The default matching rule for the entity field (default: `MatchingRule.Strict`).
- **alias** (*str*) – The alias for the "Unique property name". Used for backwards compatible entity fields.

- **error_msg** (*str*) – The custom error message that gets displayed when a `ValidationError` is raised.
- **is_value** (*bool*) – `True` if the property is the main property, else `False` (default).
- **decorator** (*callable*) – A `callable` object (function, method, etc.) that gets called each time the entity field is set. Useful for automating entity decoration, such as applying an entity icon based on the value of the field, or deriving the value of a field based on another field's value.

The following figure illustrates the mapping between the entity field's `name` (2), `description` (3), `display_name` (4) keyword arguments for a *`StringEntityField`* and the form fields in Maltego's entity field wizard:

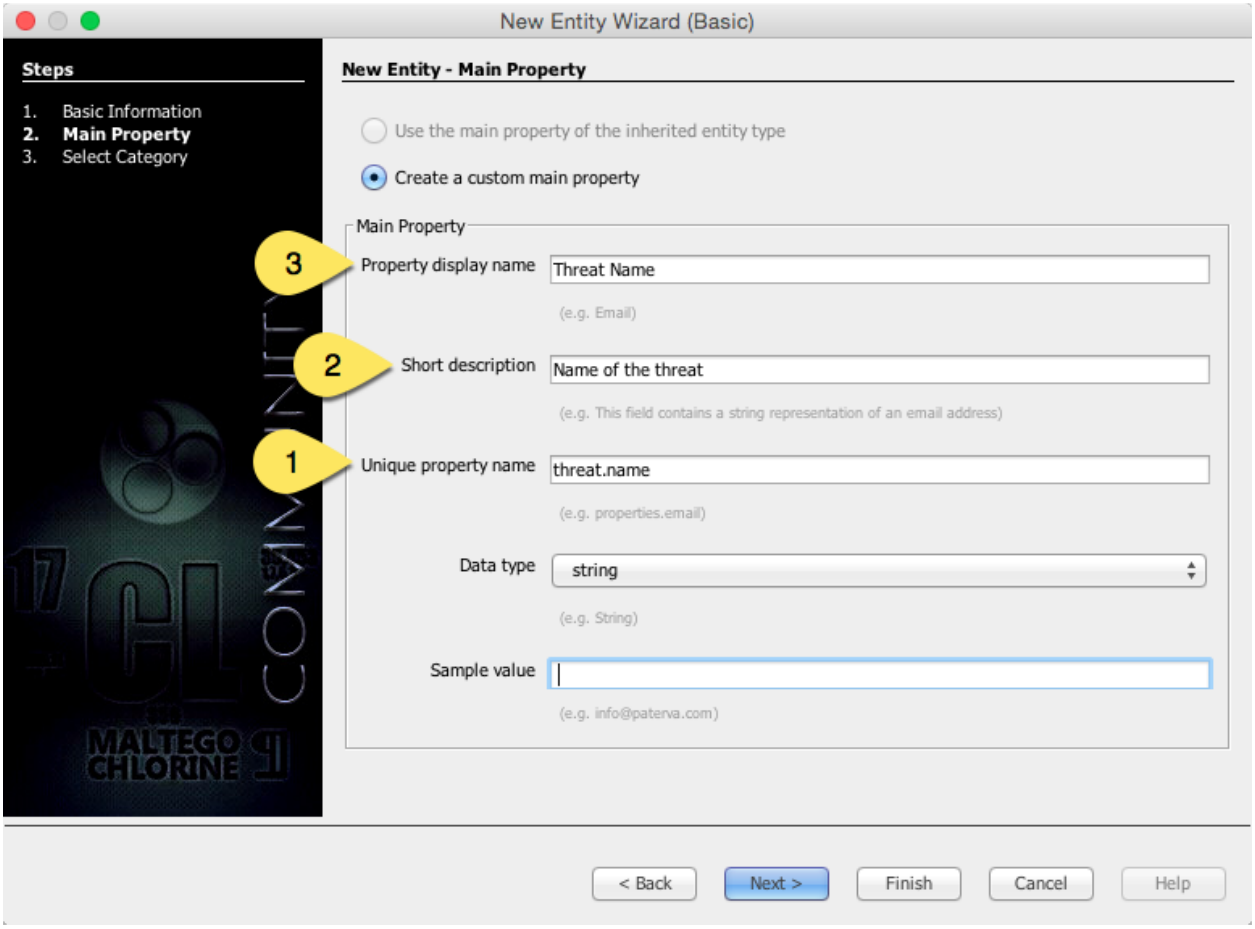

Fig. 16: Maltego entity field code to GUI mappings

When adding an entity field using the entity editor the `name` (1) and `display_name` (3) mappings can be seen below:

The field's `description` (2) can be set after an entity field is added by selecting the field from the left-hand pane and editing the field's description in the right-hand pane of the "Additional Properties" tab in the Maltego entity editor.

Defining the entity fields in the figures above using Canari would result in the following code:

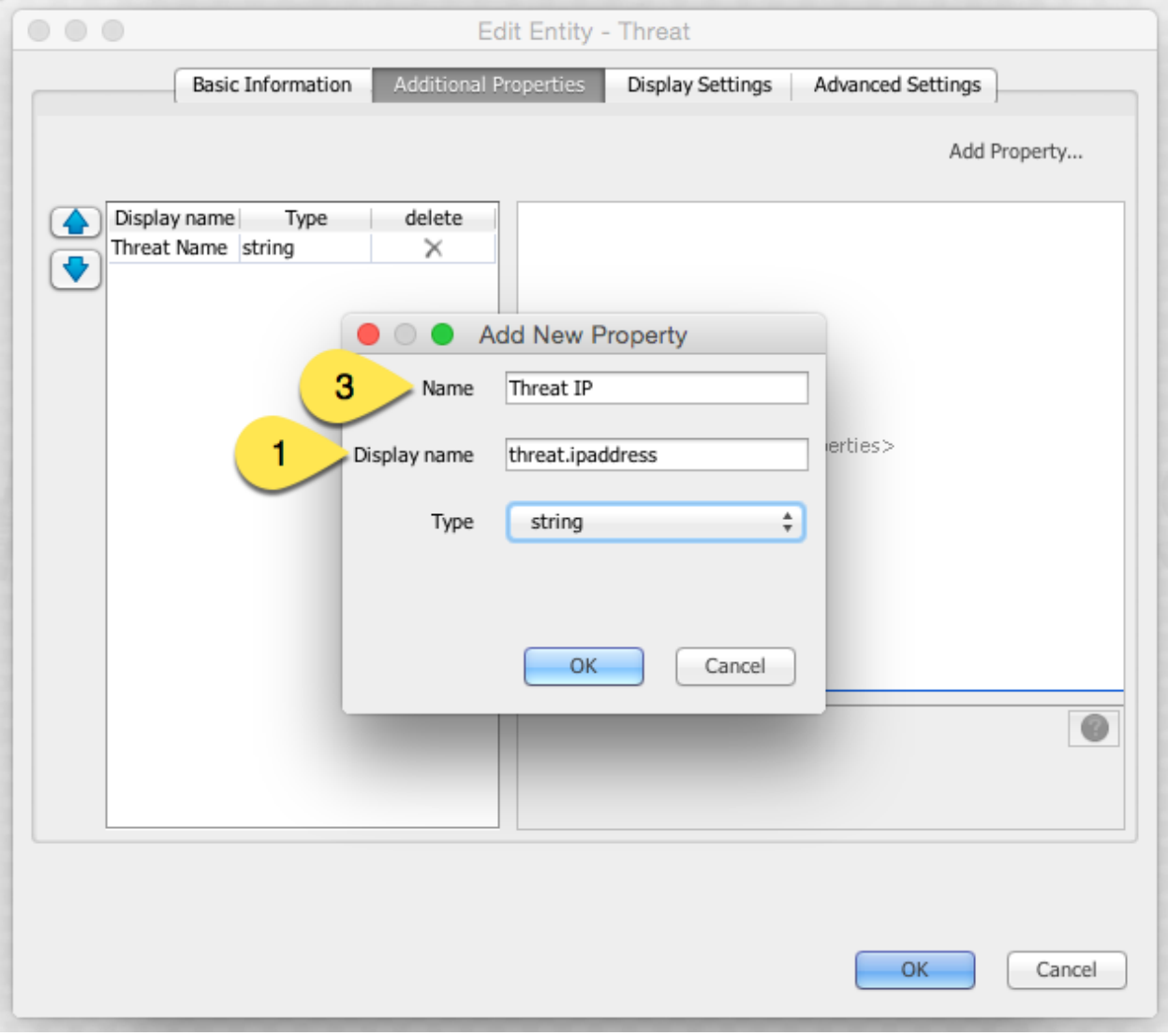

Fig. 17: Adding a field using Maltego entity field editor

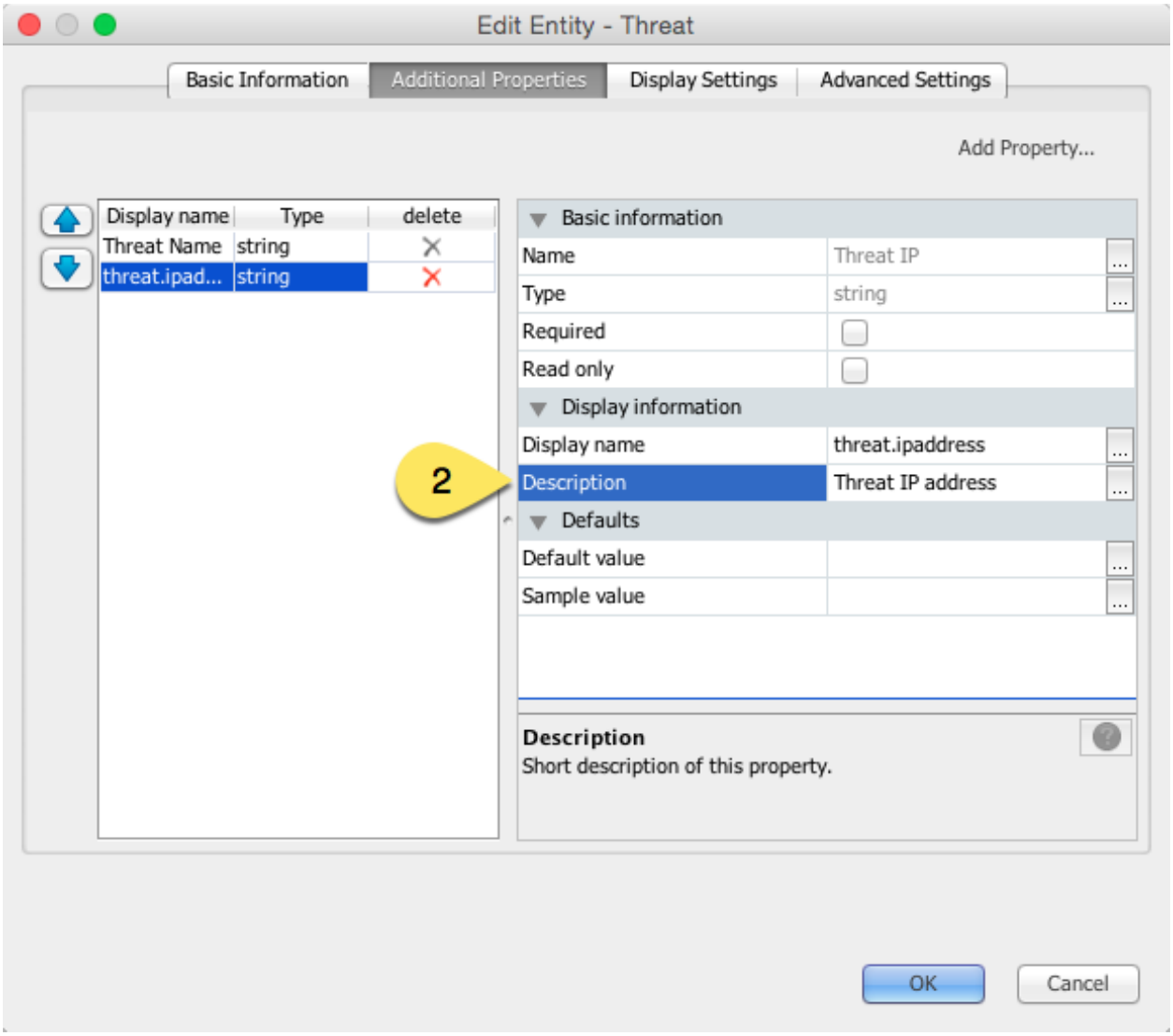

Fig. 18: Setting the description of a new entity field.

```
class Threat(Entity):
    name = StringEntityField('threat.name', display_name='Threat Name',
                             is_value=True, description='Name of the threat')
    ip_address = StringEntityField('threat.ipaddress', display_name='Threat IP',
                                   description='Threat IP address.')
```

Now let's say we wanted to add the geographic location that the IP address originates from. We can do this manually but it would probably be easier to use an entity field decorator. First, we'll add a `location` entity field:

```
class Threat(Entity):
    name = StringEntityField('threat.name', display_name='Threat Name',
                             is_value=True, description='Name of the threat')
    ip_address = StringEntityField('threat.ipaddress', display_name='Threat IP',
                                   description='Threat IP address.')
    location = StringEntityField('threat.location', display_name='Threat Location
↪',
                                 description='Threat location.')
```

Next we need to create a decorator function that accepts two arguments: the entity object, and the new field value. We'll use FreeGeoIP.net's REST-based JSON web API to lookup location information for a particular IP address and apply it to the `ip_address` field:

```
import json
from urllib2 import urlopen

def geo_locate(threat, ip):
    threat.location = json.load(urlopen('https://freegeoip.net/json/%s' % ip))[
↪'country_name']


class Threat(Entity):
    name = StringEntityField('threat.name', display_name='Threat Name',
                             is_value=True, description='Name of the threat')
    ip_address = StringEntityField('threat.ipaddress', display_name='Threat IP',
                                   description='Threat IP address.',␣
↪decorator=geo_locate)
    location = StringEntityField('threat.location', display_name='Threat Location
↪',
                                 description='Threat location.')
```

Let's take a look at the decorator in action (there may be a delay if you're on a slow network):

```
>>> t = Threat('foo', ip_address='4.2.2.1')
>>> print t.location
United States
```

The *EnumEntityField* and *RegexEntityField* types accept additional keyword arguments in their constructors in addition to the arguments accepted by the *StringEntityField* type.

class canari.maltego.message.**EnumEntityField**(*name*, *choices=[]*, ***extras*)

Parameters **choices** (*iterable*) – an iterable of choices for valid values the entity field will accept.

Raises **ValueError** – if `choices` is empty or `None`.

In the event that an entity's field is set to a value that is not specified in the `choices` keyword argument, a `ValidationError` will be raised. Let's add a threat level entity field to our `Threat` entity to demonstrate

the use of *EnumEntityField* types:

```
class Threat(Entity):
        # ...
        threat_level = EnumEntityField('threat.level', display_name='Threat Level
↪',
                                       choices=[1,2,3,4,5], description='Threat␣
↪level.')
```

Let's see the `threat_level` field in action:

```
>>> t = Threat('foo', threat_level=1)
>>> t.threat_level = 6
Traceback (most recent call last):
    ...
ValidationError: Invalid value ('6') set for field 'Threat Level'. Expected one␣
↪of these values: ['1', '2', '3', '4', '5'].
```

Note: *EnumEntityField* entity fields convert all objects in the `choices` iterable to strings using the `str()` function.

class canari.maltego.message.**RegexEntityField**(*name*, *pattern='.*'*, ***extras*)

Parameters **pattern** (*str*) – a regular expression pattern that gets used with `re.match()`.

Each time the field's value is set or retrieved, a call to `re.match()` is made with the specified `pattern`. If the value doesn't match the pattern then a `ValidationError` is raised. Let's add a threat identifier to our previous entity. We'll require users to enter the threat ID using the following syntax `'threat-0000'`:

```
class Threat(Entity):
        # ...
        threat_id = RegexEntityField('threat.id', display_name='Threat ID',
                                     pattern='^threat-\d{4}$', description='Threat␣
↪unique identifier.')
```

Let's see the `threat_id` field in action:

```
>>> t = Threat('foo', threat_id='threat-0123')
>>> t.threat_id = 'threat-12345'
Traceback (most recent call last):
    ...
ValidationError: The field value ('threat-12345') set for field 'Threat ID' does␣
↪not match the regular expression /^threat-\d{4}$/.
>>> t.threat_id = '12345'
Traceback (most recent call last):
    ...
ValidationError: The field value ('12345') set for field 'Threat ID' does not␣
↪match the regular expression /^threat-\d{4}$/.
```

3.4.2 Customizing `ValidationError` Error Messages

You may have noticed that the error messages above are generic in nature. The good news is that you can specify a more user-friendly error message for `ValidationError` exceptions by specifying the `error_msg` keyword argument in your entity field definition. Error messages are formatted using the `str.format()` method and `'{var}'` string notation. Each entity field type accepts the following string formatting arguments:

Type	Error Message Arguments
StringEntityField	Not applicable.
EnumEntityField	field, value, expected
IntegerEntityField	field, value
BooleanEntityField	field, value
FloatEntityField	field, value
LongEntityField	field, value
DateTimeEntityField	field, value
DateEntityField	field, value
TimeSpanEntityField	field, value
RegexEntityField	field, value, pattern
ColorEntityField	field, value

For example, if we wanted to modify the `threat_level` entity field's (of type *EnumEntityField*) default error message in our previous example, we can do this like so:

```
class Threat(Entity):
            # ...
            threat_level = EnumEntityField('threat.level', display_name='Threat Level
↪',
                                           choices=[1,2,3,4,5], description='Threat␣
↪level.',
                                           error_msg='{field!r}: {value!r} not in
↪{expected!r}.')
```

Then our error message would look like this when we encounter a `ValidationError` exception:

```
>>> t = Threat('foo', threat_level=1)
>>> t.threat_level = 6
Traceback (most recent call last):
    ...
ValidationError: 'Threat Level': '6' not in ['1', '2', '3', '4', '5'].
```

See also:

For a comprehensive overview of string formatting syntax, see the Format String Syntax section in the official Python documentation.

3.4.3 Creating Custom Entity Field Types

Entity field types are glorified Python property objects and subclasses of the *StringEntityField* type. You can either subclass *StringEntityField* directly, or leverage one of the many other entity field types available to you and augment their constructors, getters, and setters as required. Let's take a look at how we can create a digest entity field that expects hash values in ascii hex format using the *RegexEntityField* type:

```
class DigestEntityField(RegexEntityField):
    def __init__(self):
        super(DigestEntityField, self).__init__('content.digest', pattern='^[A-Fa-f0-
↪9]$',
                                                description="The message's digest.",
                                                display_name='Message Digest',
                                                error_msg='{field!r}: invalid message␣
↪digest: {value!r}!')
```

(continues on next page)

(continued from previous page)

```
class Document(Entity):
    digest = DigestEntityField()

class DataPacket(Entity):
    digest = DigestEntityField()
```

This can significantly simplify and centralize refactoring efforts on entity fields in cases where the same entity field definition is reused in many other unrelated entity types. Alternatively, you can follow this template if you wish to implement something a bit more complex for field value validation:

```
class MyEntityField(StringEntityField):

    error_msg = 'A default error message with {field} and {value} and other variables␣
↪if you wish.'

    def __init__(self, name, **extras):
        super(MyEntityField, self).__init__(name, **extras)
        # TODO: store any extra attributes that are not handled by StringEntityField.

    def __get__(self, obj, objtype):
        value = super(RegexEntityField, self).__get__(obj, objtype) # get field value
        # TODO: type conversions if necessary
        self.validate(value)
        return value

    def __set__(self, obj, value):
        # TODO: type conversions if necessary
        self.validate(value)
        super(RegexEntityField, self).__set__(obj, value) # set field value

    def validate(self, value):
        is_valid = True # TODO: implement some sort of validation
        if not is_valid:
            raise ValidationError(self.get_error_msg(self.display_name or self.name,␣
↪value))
```

3.4.4 Adding Additional Information to Entities

Sometimes you want to display additional information to users. Either because it doesn't fall into one of the entity's predefined fields or it's just informational data that isn't required for subsequent transform requests. For these use-cases, Canari provides two classes, *Field* and *Label*, that can be used to define dynamic entity fields and read-only information, respectively.

Field is the underlying data container for all the entity field types mentioned in the previous sections, above. In fact, if you took a look at the `fields` attribute in an *Entity* object, you'd notice the presence of this objects in a dictionary. As mentioned earlier, *StringEntityField* and friends are merely proxies to the `fields` dictionary. A field object can be constructed in the following manner:

class canari.maltego.message.**Field**(*name*, *value*, *display_name=''*, *matching_rule=MatchingRule.Strict*)

Parameters

- **name** (*str*) – the unique field identifier, usually in dotted form (i.e. 'threat.name')
- **value** (*str*) – the value of the field or property.

- **display_name** (*str*) – the user-friendly name of the field (i.e. 'Threat Name')
- **matching_rule** (*MatchingRule*) – the matching rule for this field, either `MatchingRule.Strict` (default) or `MatchingRule.Loose`.

See also:

Matching rules for more information on matching rules and how they relate to Maltego graph behavior.

Fields that are pre-defined (or statically defined) for a particular entity in Maltego do not require the specification of the `display_name` argument. The display name defined in Maltego will be used instead. The `display_name` argument is particularly important for dynamic fields (fields that are not part of the entity definition in Maltego). If omitted, and the field is dynamic, Maltego will name the field "Temp" in the "Properties View" pane. Dynamic fields can be attached to entities in Canari in the following manner:

```
>>> from canari.maltego.entities import *
>>> l = Location('Canada')
>>> l += Field('location.symbol', 'Maple Leaf', display_name='National Symbol')
```

In the example above, we've added a previously undefined field, 'location.symbol', and added it to the builtin `Location` entity in Maltego. The figure below illustrates the look and feel of a dynamic property (1) in Maltego:

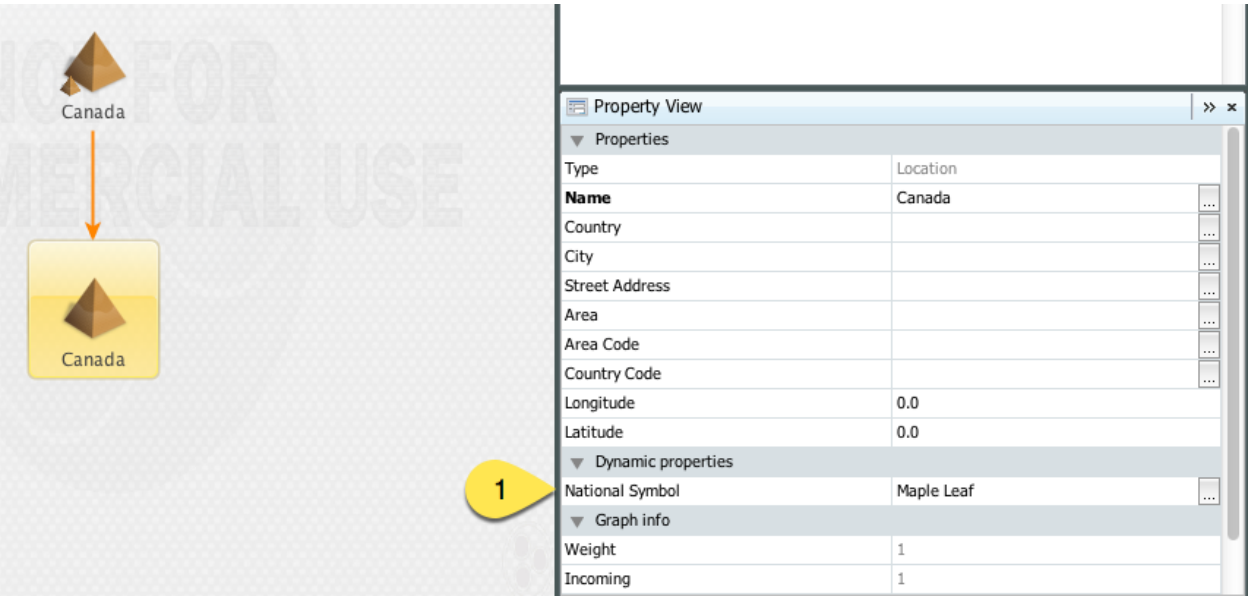

Fig. 19: Dynamic entity field/property

Like static fields, dynamic fields get passed to transforms as input. Retrieving a dynamic field from a transform is as simple as accessing the entity's `fields` attribute. Continuing from our last example:

```
>>> print l['location.symbol'].value
Maple Leaf
```

Note: Dynamic field values are always return as an instance of `str` and need to be manually marshalled into their appropriate types and validated, if necessary.

Now, let's say we wanted to attach additional information to the entity that we do not want to pass as transform input. Labels serve this purpose and allow transform developers to set both text- and HTML-based content in the Maltego "Details View" pane.

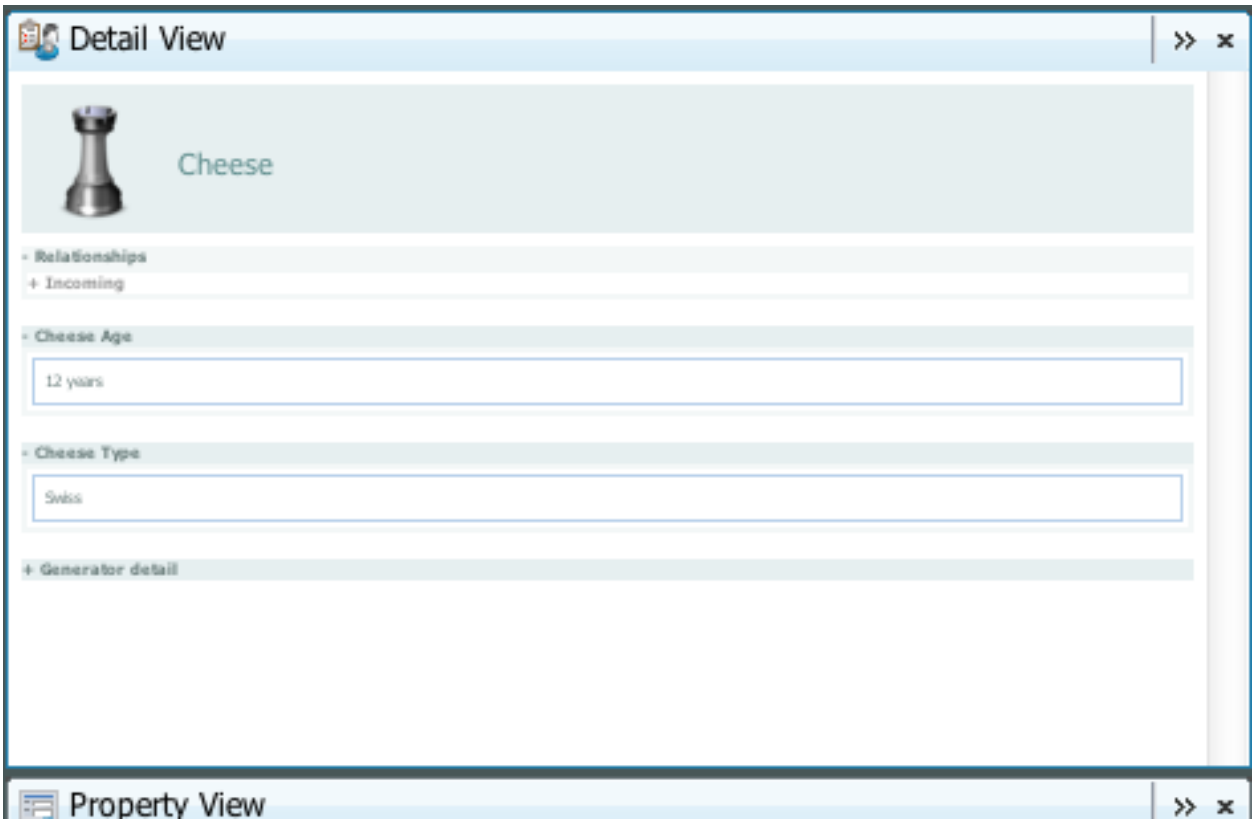

Fig. 20: Entity labels in “Detail View”

Label objects can be constructed in the following manner:

class canari.maltego.message.**Label**(*name*, *value*[, *type='text/text'*])

Parameters

- **name** (*str*) – The title or name of the label.
- **value** (*str*) – The content that will appear under the label's name.
- **type** (*str*) – A MIME type identifying the content's type; either `'text/text'` (default) or `'text/html'`.

Adding labels to an entity is the same as adding dynamic fields:

```
>>> t = Threat('Cheese', country='Switzerland')
>>> t += Label('Age', '12 years')
```

By default, fields are rendered as regular text. If you want to render HTML in the "Details View" you can do so by setting `type` to `'text/html'` and `value` to an HTML fragment, like so:

```
>>> t = Threat('Cheese', country='Switzerland')
>>> t += Label('Table', '<table><tr><th>header</th></tr><tr><td>row</td></tr></table>
↪', 'text/html')
```

3.4.5 Matching Rules and Maltego

Maltego supports the concept of matching rules for entity fields. A matching rule defines how an output entity (returned by a transform) is merged with other pre-existing entities, of the same type, that share the same entity value. Maltego currently supports two matching rules, loose and strict matching, which are represented in Canari with the `MatchingRule.Loose` and `MatchingRule.Strict` attributes, respectively. Take a look at how the behavior of these two matching rules differ when used to compare two entities (`x` and `y`) of the same type:

Value	Meaning
`MatchingRule.Strict`	if `x.value == y.value and x.field == y.field` then allow entities to merge.
`MatchingRule.Loose`	if `x.value == y.value` then `x.field = y.field` and merge entities.

Attention: It is important to note that with loosely matched entity fields, the previous value is overridden with the new value for that field. If you wish to preserve the different values for particular entity field, then you will have to revert to strict matching.

3.4.6 Automatically Generating Canari Entity Definitions

Entity definitions can be automatically generated using the **`canari generate-entities`** command. In order to automatically generate entity definitions, you will have to perform the following steps:

1. Export your custom entities from Maltego using the "Export Entities" wizard and save the profile as `entities.mtz` under the `<project name>/src/<project name>/resources/maltego/` project folder.

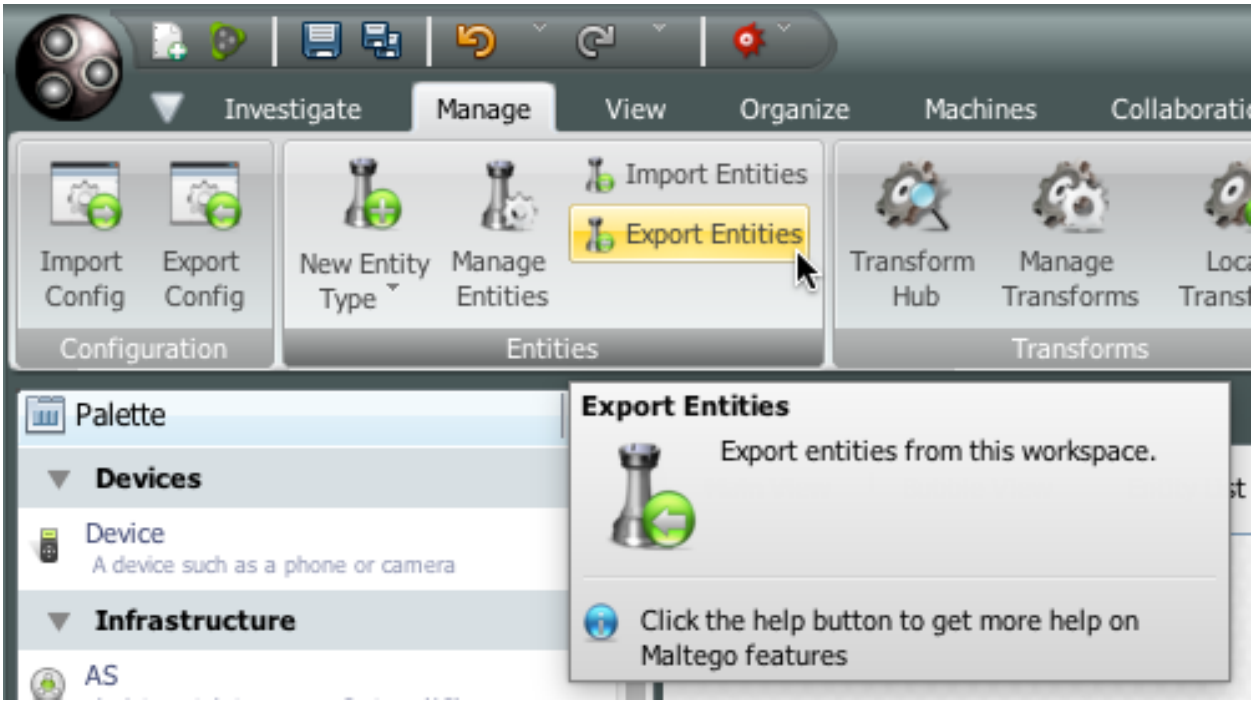

Fig. 21: Export Entities button

2. Run **`canari generate-entities`** in your project folder; this will generate an `entities.py` file in `<project name>/src/<project name>/transforms/common/`. Here's an example of the command in action when running it for a transform package named "foo":

```
$ canari generate-entities
'foo/transforms/common/entities.py' already exists. Are you sure you want to␣
↪overwrite it? [y/N]: y
Generating 'foo/src/foo/transforms/common/entities.py'...
Parsing entity definition Entities/name.Foo.entity...
Generating entity definition for Entities/name.Foo.entity...
done.
```

In the command above we are completely overwriting the entities.py file since we have not defined any entities yet. This will usually be the normal course of action for most transform developers when importing entities into Canari for the first time. However, if you have performed this action before and would like to simply update the pre-existing `entities.py` file then you can pass the `-a` parameter to **`canari generate-entities`**, like so:

```
$ canari generate-entities -a
Discovered 2 existing entities, and 1 namespaces...
Appending to '/Users/ndouba/tools/canari3/foo/src/foo/transforms/common/entities.
↪py'...
Parsing entity definition Entities/name.Foo.entity...
Skipping entity generation for name.Foo as it already exists...
done.
```

The **`canari generate-entities`** command is capable of identifying and skipping over entities that have already been defined in your existing `entities.py` file.

3. Finally, edit the `entities.py` file to your liking (i.e. perhaps change the name of a property to something more memorable).

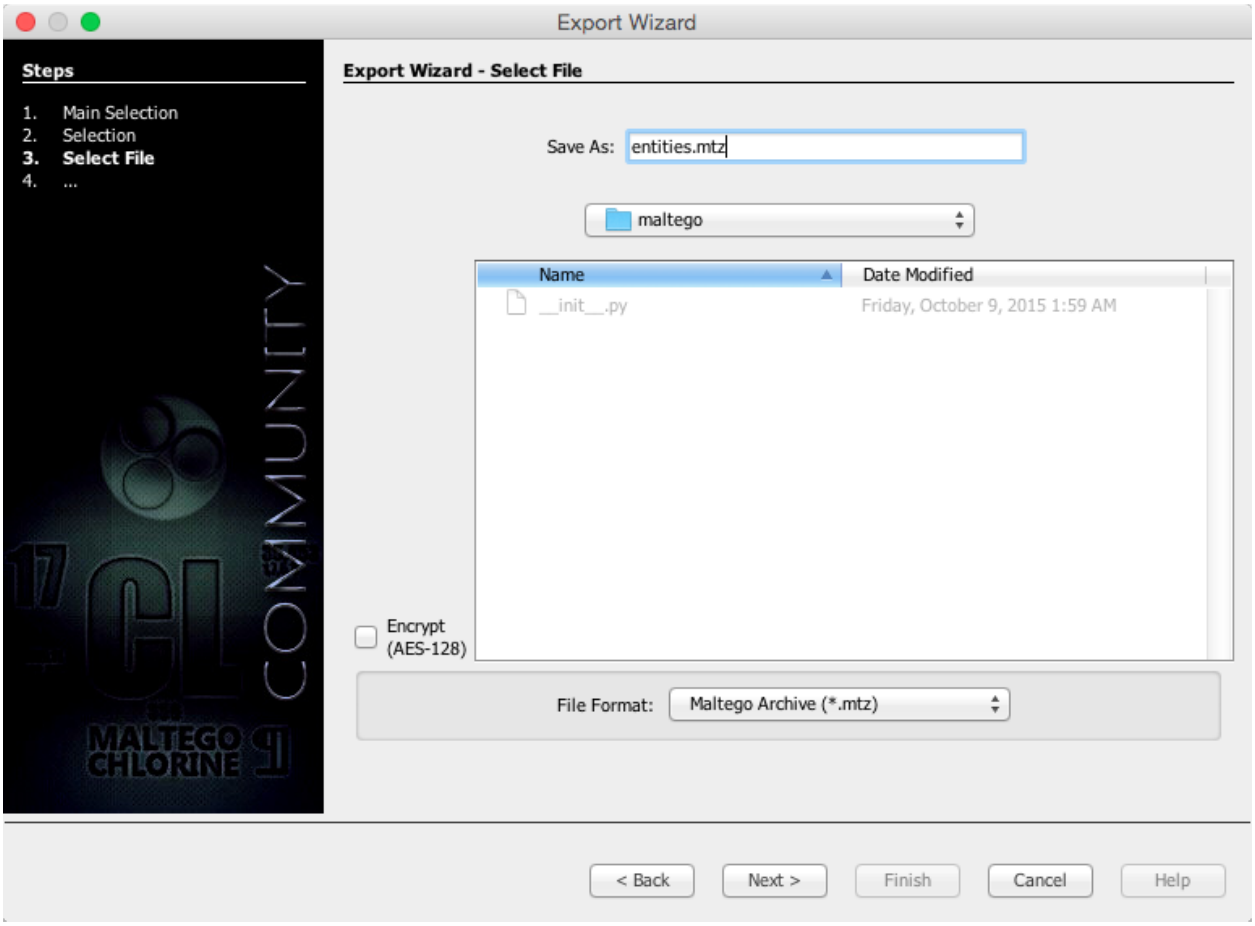

Fig. 22: Saving entities.mtz to `<project name>/src/<project name>/resources/maltego/`

CHAPTER 4

canari.config - Canari Configuration Files

New in version 3.0.

additional child configuration files specified in the canari.local.configs option. These files are usually located in the .canari/ directory in your home directory (i.e ~/.canari/ in Mac/Linux or %HOMEPATH%\.canari\ in Windows). Canari configuration files are loaded in the following manner:

1. Canari checks to see whether or not the transform package being loaded is in the global Python site-package directory. If it is, the canari.conf file in the global .canari directory is loaded. Otherwise, the canari.conf file in the current working directory is used, if present.
2. Once the main configuration file is loaded, Canari will inspect the canari.local.configs option to determine whether there are any additional configuration files to be loaded. Typically this option is populated with a list of configuration files belonging to all the transform packages that have been installed (via **canari create-profile**) using Canari.
3. Canari will then iterate over each configuration filename entry in canari.local.configs and load the configuration files in the same order as they appear in the configuration file. If a configuration option in one configuration file shares the same name and section as one from another, the latest configuration value will be used.

Common use-cases for using the configuration file is to retrieve information such as backend API keys or credentials that you may use to connect to third-party services. Here's an example of how to use the configuration object in your transforms:

```
class MyTransform(Transform):
    def do_transform(request, response, config):
        db = connect_to_db(config['foo.local.username'], config['foo.local.password'])
        results = db.query('SELECT name FROM users WHERE id=?', request.entity.value)
        for r in results:
            response += Phrase(r)
        return response
```

In the example above, the canari.conf file would look like this:

```
[canari.local]
configs = foo.conf

# ...
```

The transform package's configuration file, `foo.conf`, would look like this:

```
[foo.local]
username = bar
password = baz
```

Note: As a best practice for remote transforms, only backend architectural details and license keys should be stored in the configuration file. Client-side API keys can and should be received from the Maltego transform request parameters.

The `config` parameter in our `Transform.do_transform()` method is a `CanariConfigParser` object. By default all transform runners instantiate the configuration object using the *`load_config()`* function with no parameters and pass the result to the transforms. If however, you wish to load a separate configuration file, manually, you can use the *`load_config()`* function in the following manner:

`canari.config.`**`load_config`**(*config_file=None*, *recursive_load=True*)

Parameters

- **`config_file`** (*`str`*) – the absolute path to a custom configuration file.
- **`recursive_load`** (*`bool`*) – `True` if your configuration file has a `canari.local.configs` option and you wish to load the additional configuration files specified in that option. `False` otherwise.

If `recursive_load` is `True` but your configuration file does not have a `canari.local` section or a `configs` option specified under that section, it will be quietly ignored.

Once loaded, configuration objects can be queried in the following manner (where `c` is the configuration object):

Operation	Meaning
`'section.name' in c`	Does the configuration contain the specified section.
`'section.name.option' in c`	Does the configuration contain the specified section and option.
`c['section.name.option']`	Retrieve the value of the specified option and section.

Configuration objects have two additional features over and above regular configuration objects in Python, automatic type marshalling, and advanced string interpolation.

4.1 Automatic Type Marshalling

One of the biggest advantages in using the `CanariConfigParser` over other configuration parsers in Python is its ability to automatically marshal options to the appropriate type. For example, say you had the following configuration file:

```
[foo.local]
username = admin
threshold = 1000
timeout = 0.5
servers = 10.0.0.1, 10.0.0.2
validator = object://foo.validators/simple
```

These options would translate to the following when retrieve from your transform:

```
>>> config['foo.local.username'] # string
'admin'
>>> config['foo.local.threshold'] # integer
1000
>>> config['foo.local.timeout'] # float
0.5
>>> config['foo.local.servers'] # list of strings
['10.0.0.1', '10.0.0.2']
>>> config['foo.local.validator'] # foo.validators.simple object
<function foo.local.validator at 0x1337b33f>
>>>
```

Attention: Options starting with `object://` will return the option as a string in remote transform execution mode.

4.2 Option String Interpolation

In addition to automatic type marshalling, `CanariConfigParser` objects support additional string interpolation features. This allows you to reference other options within your configuration file as well as system environment variables. For example, querying options from the following configuration file:

```
[foo.local]
bar = %(baz)
baz = 1
mypaths = ${PATH}:/custom/path
```

Would result in the following:

```
>>> config['foo.local.bar']
1
>>> config['foo.local.mypaths']
/usr/bin:/bin:/usr/local/bin:/custom/path
>>>
```

CHAPTER 5

canari.mode - Canari Execution Modes

New in version 3.0.

Canari now supports the concept of execution modes. Execution modes allow transform developers to detect what context a transform is operating in (i.e. local or remote, production or debug, etc.) and alter the behaviour of their transforms accordingly. Execution modes also globally enable or disable high-risk functionality or modules that you would normally allow in local transform mode. Here's an example of how a transform can check if it's running as a local or transform:

```
from canari.maltego.entities import Phrase,WebSite
from canari.mode import is_local_exec_mode

class MyTransform(Transform):
    input_type = WebSite

    def do_transform(self, request, response, config):
        website = request.entity
        if is_local_exec_mode():
            # TODO: do something locally
            pass
        else:
            # TODO: do something remotely
            pass
        return response
```

You can also determine which transform runner is invoking the transform or whether it is operating in debug versus production mode, like so:

```
# ...
def do_transform(self, request, response, config):
    if is_local_debug_exec_mode():
        debug("We're running in debug mode.")
```

Canari modes can also be checked in the global scope to prevent a transform, entity, sensitive function or variable from being exposed or defined when operating in a particular mode:

```
if is_local_exec_mode():
    @RequireSuperUser
    class MyTransform(Transform):
        # Does risky stuff in local mode
        pass
else:
    class MyTransform(Transform):
        # Does safer stuff in remote mode
        pass
```

Canari currently supports the following execution modes:

Table 1: Primitive Modes

Value	Meaning
`CanariMode.Local`	Transform is running locally.
`CanariMode.Remote`	Transform is running remotely.
`CanariMode.Debug`	Transform is running in debugging mode.
`CanariMode.Dispatch`	Transform is running in production mode.
`CanariMode.Plume`	Transform is running in Plume container.
`CanariMode.Shell`	Transform is running from **`canari shell`**.

Table 2: Production Modes

Value	Meaning
`CanariMode.LocalDispatch`	Transform running in local production mode.
`CanariMode.RemotePlumeDispatch`	Transform is running in Plume production mode.

Table 3: Debugging Modes

Value	Meaning
`CanariMode.LocalDebug`	Transform running local debugging mode.
`CanariMode.RemotePlumeDebug`	Transform is running in Plume in debugging mode.
`CanariMode.LocalShellDebug`	Transform is running running from **`canari shell`**.

Table 4: Unknown Modes

Value	Meaning
`CanariMode.Unknown`	Canari hasn't been initialized and is operating in an unknown mode.
`CanariMode.RemoteUnknown`	Canari hasn't been initialized but is operating in remote mode.
`CanariMode.LocalUnknown`	Canari hasn't been initialized but is operating in local mode.

The 5 transform runners that come out of the box with Canari operate in the following modes, by default:

Runner	Mode
dispatcher	`CanariMode.LocalDispatch`
canari run-transform	`CanariMode.LocalDispatch`
canari debug-transform	`CanariMode.LocalDebug`
canari shell	`CanariMode.LocalShellDebug`
plume	`CanariMode.RemotePlumeDispatch`

Non-primitive operating modes are or'd bitmasks of the primitive operating modes. For example, `CanariMode.LocalDebug` is equivalent to `CanariMode.Local | CanariMode.Debug`. This makes it possible to perform a broad (i.e. `is_local_exec_mode()`) or narrow (i.e. `is_local_debug_exec_mode()`) check on an operating mode. For example:

```
>>> from canari.mode import *
>>> old_mode = set_canari_mode(CanariMode.LocalDebug)
>>> is_local_exec_mode()
True
>>> is_debug_exec_mode()
True
>>> is_local_debug_exec_mode()
True
```

The *canari.mode* module provides the following functions:

canari.mode.**set_canari_mode**(*mode*)

> **Parameters** **mode** (*CanariMode*) – the desired operating mode.
>
> **Returns** the old operating mode.

Sets the Canari operating mode and returns the old one. The old operating mode can be ignored if you never wish to restore the original operating mode.

canari.mode.**get_canari_mode**()

> **Returns** current Canari operating mode.

Gets the current Canari operating mode. If a prior call to `canari_set_mode()` has not been made, the default operating mode is `CanariMode.Unknown`.

canari.mode.**get_canari_mode_str**()

> **Returns** current Canari operating mode as a user-friendly string.

Gets the current Canari operating mode as a user-friendly string which can be displayed in logs or debugging information. For example:

```
>>> print get_canari_mode_str()
Local (runner=shell, debug=True)
```

As demonstrated earlier, *canari.mode* provides convenience functions that can be used to query the current operating mode. These functions return either `True` or `False` depending on whether the operating mode being queried has the appropriate operating mode bits set:

Function	Returns `True` For Operating Modes
`is_local_exec_mode()`	`CanariMode.Local*`
`is_debug_exec_mode()`	`CanariMode.*Debug*`
`is_dispatch_exec_mode()`	`CanariMode.*Dispatch*`
`is_remote_exec_mode()`	`CanariMode.Remote*`
`is_plume_exec_mode()`	`CanariMode.*Plume*`
`is_shell_exec_mode()`	`CanariMode.*Shell*`
`is_unknown_exec_mode()`	`CanariMode.*Unknown*`
`is_local_debug_exec_mode()`	`CanariMode.Local*Debug*`
`is_local_dispatch_exec_mode()`	`CanariMode.Local*Dispatch*`
`is_local_unknown_exec_mode()`	`CanariMode.LocalUnknown`
`is_remote_plume_debug_exec_mode()`	`CanariMode.RemotePlumeDebug`
`is_remote_plume_dispatch_exec_mode()`	`CanariMode.RemotePlumeDispatch`
`is_remote_unknown_exec_mode()`	`CanariMode.RemoteUnknown`
`is_local_shell_debug_exec_mode()`	`CanariMode.LocalShellDebug`

CHAPTER 6

canari.maltego.entities Maltego Entities

New in version 3.0.

6.1 maltego.TrackingCode (alias: maltego.UniqueIdentifier)

class canari.maltego.entities.**TrackingCode**(***kwargs*)

Parameters

- **unique_identifier** (*str*) – Uniqueidentifier (properties.uniqueidentifier)
- **identifier_type** (*str*) – Identifier Type (identifierType)

6.2 maltego.NSRecord

class canari.maltego.entities.**NSRecord**(***kwargs*)

Parameters **fqdn** (*str*) – DNS Name (fqdn)

6.3 maltego.affiliation.Bebo (alias: AffiliationBebo)

class canari.maltego.entities.**Bebo**(***kwargs*)

Parameters

- **uid** (*str*) – UID (affiliation.uid)
- **profile_url** (*str*) – Profile URL (affiliation.profile-url)
- **person_name** (*str*) – Name (person.name)
- **network** (*str*) – Network (affiliation.network)

6.4 maltego.NominatimLocation

class canari.maltego.entities.**NominatimLocation**(***kwargs*)

Parameters **nominatim** (*str*) – Nominatim Location (properties.nominatimlocation)

6.5 maltego.EmailAddress

class canari.maltego.entities.**EmailAddress**(***kwargs*)

Parameters **email** (*str*) – Email Address (email)

6.6 maltego.affiliation.Spock (alias: AffiliationSpock)

class canari.maltego.entities.**Spock**(***kwargs*)

Parameters

- **websites** (*str*) – Listed Websites (spock.websites)
- **uid** (*str*) – UID (affiliation.uid)
- **profile_url** (*str*) – Profile URL (affiliation.profile-url)
- **person_name** (*str*) – Name (person.name)
- **network** (*str*) – Network (affiliation.network)

6.7 maltego.Unknown

class canari.maltego.entities.**Unknown**(***kwargs*)

6.8 maltego.DNSName

class canari.maltego.entities.**DNSName**(***kwargs*)

Parameters **fqdn** (*str*) – DNS Name (fqdn)

6.9 maltego.Webdir

class canari.maltego.entities.**Webdir**(***kwargs*)

Parameters **name** (*str*) – Name (directory.name)

6.10 maltego.Document

class canari.maltego.entities.**Document**(***kwargs*)

Parameters

- **url** (*str*) – URL (url)
- **title** (*str*) – Title (title)
- **metadata** (*str*) – Meta-Data (document.metadata)

6.11 maltego.affiliation.Zoominfo

class canari.maltego.entities.**Zoominfo**(***kwargs*)

Parameters

- **uid** (*str*) – UID (affiliation.uid)
- **profile_url** (*str*) – Profile URL (affiliation.profile-url)
- **person_name** (*str*) – Name (person.name)
- **network** (*str*) – Network (affiliation.network)

6.12 maltego.BuiltWithRelationship

class canari.maltego.entities.**BuiltWithRelationship**(***kwargs*)

Parameters

- **matches** (*str*) – Matches (matches)
- **builtwith** (*str*) – BuiltWith Technology (properties.builtwithrelationship)

6.13 maltego.Service

class canari.maltego.entities.**Service**(***kwargs*)

Parameters

- **ports** (*str*) – Ports (port.number)
- **name** (*str*) – Description (service.name)
- **banner** (*str*) – Service Banner (banner.text)

6.14 maltego.Organization

class canari.maltego.entities.**Organization**(***kwargs*)

Parameters **name** (*str*) – Name (title)

6.15 maltego.URL

class canari.maltego.entities.**URL**(***kwargs*)

Parameters

- **url** (*str*) – URL (url)
- **title** (*str*) – Title (title)
- **short_title** (*str*) – Short title (short-title)

6.16 maltego.affiliation.Orkut (alias: AffiliationOrkut)

class canari.maltego.entities.**Orkut**(***kwargs*)

Parameters

- **uid** (*str*) – UID (affiliation.uid)
- **profile_url** (*str*) – Profile URL (affiliation.profile-url)
- **person_name** (*str*) – Name (person.name)
- **network** (*str*) – Network (affiliation.network)

6.17 maltego.Device

class canari.maltego.entities.**Device**(***kwargs*)

Parameters **device** (*str*) – Device (properties.device)

6.18 maltego.Location

class canari.maltego.entities.**Location**(***kwargs*)

Parameters

- **streetaddress** (*str*) – Street Address (streetaddress)
- **name** (*str*) – Name (location.name)
- **longitude** (*float*) – Longitude (longitude)
- **latitude** (*float*) – Latitude (latitude)
- **countrycode** (*str*) – Country Code (countrycode)
- **country** (*str*) – Country (country)
- **city** (*str*) – City (city)
- **areacode** (*str*) – Area Code (location.areacode)
- **area** (*str*) – Area (location.area)

6.19 maltego.Banner

class canari.maltego.entities.**Banner**(***kwargs*)

Parameters **text** (*str*) – Banner (banner.text)

6.20 maltego.Hashtag

class canari.maltego.entities.**Hashtag**(***kwargs*)

Parameters **hashtag** (*str*) – Hashtag (twitter.hashtag)

6.21 maltego.AS (alias: ASNumber)

class canari.maltego.entities.**AS**(***kwargs*)

Parameters **number** (*int*) – AS Number (as.number)

6.22 maltego.affiliation.Linkedin (alias: AffiliationLinkedin)

class canari.maltego.entities.**Linkedin**(***kwargs*)

Parameters

- **uid** (*str*) – UID (affiliation.uid)
- **profile_url** (*str*) – Profile URL (affiliation.profile-url)
- **person_name** (*str*) – Name (person.name)
- **network** (*str*) – Network (affiliation.network)

6.23 maltego.File

class canari.maltego.entities.**File**(***kwargs*)

Parameters

- **source** (*str*) – Source (source)
- **description** (*str*) – Description (description)

6.24 maltego.CircularArea

class canari.maltego.entities.**CircularArea**(***kwargs*)

Parameters

- **radius** (*int*) – Radius (m) (radius)
- **longitude** (*float*) – Longitude (longitude)
- **latitude** (*float*) – Latitude (latitude)
- **area_circular** (*str*) – Circular Area (area.circular)

6.25 maltego.IPv4Address (alias: IPAddress)

class canari.maltego.entities.**IPv4Address**(***kwargs*)

Parameters

- **ipv4address** (*str*) – IP Address (ipv4-address)
- **internal** (*bool*) – Internal (ipaddress.internal)

6.26 maltego.affiliation.Facebook (alias: AffiliationFacebook)

class canari.maltego.entities.**Facebook**(***kwargs*)

Parameters

- **uid** (*str*) – UID (affiliation.uid)
- **profile_url** (*str*) – Profile URL (affiliation.profile-url)
- **person_name** (*str*) – Name (person.name)
- **network** (*str*) – Network (affiliation.network)

6.27 maltego.PhoneNumber

class canari.maltego.entities.**PhoneNumber**(***kwargs*)

Parameters

- **phonenumber** (*str*) – Phone Number (phonenumber)
- **lastnumbers** (*str*) – Last Digits (phonenumber.lastnumbers)
- **countrycode** (*str*) – Country Code (phonenumber.countrycode)
- **citycode** (*str*) – City Code (phonenumber.citycode)
- **areacode** (*str*) – Area Code (phonenumber.areacode)

6.28 maltego.Tweet

class canari.maltego.entities.**Tweet**(***kwargs*)

Parameters

- **tweet_id** (*str*) – Tweet ID (id)
- **tweet** (*str*) – Tweet (twit.name)
- **title** (*str*) – Title (title)
- **image_link** (*str*) – Image Link (imglink)
- **date_published** (*str*) – Date published (pubdate)
- **content** (*str*) – Content (content)
- **author_uri** (*str*) – Author URI (author_uri)
- **author** (*str*) – Author (author)

6.29 maltego.affiliation.Flickr (alias: AffiliationFlickr)

class canari.maltego.entities.**Flickr**(***kwargs*)

Parameters

- **uid** (*str*) – UID (affiliation.uid)
- **profile_url** (*str*) – Profile URL (affiliation.profile-url)
- **person_name** (*str*) – Name (person.name)
- **network** (*str*) – Network (affiliation.network)

6.30 maltego.FacebookObject

class canari.maltego.entities.**FacebookObject**(***kwargs*)

Parameters **object** (*str*) – Facebook Object (properties.facebookobject)

6.31 maltego.WebTitle

class canari.maltego.entities.**WebTitle**(***kwargs*)

Parameters **title** (*str*) – Title (title)

6.32 maltego.GPS

class canari.maltego.entities.**GPS**(***kwargs*)

Parameters

- **longitude** (*float*) – Longitude (longitude)
- **latitude** (*float*) – Latitude (latitude)
- **gps** (*str*) – GPS Co-ordinate (properties.gps)

6.33 maltego.MXRecord

class canari.maltego.entities.**MXRecord**(***kwargs*)

Parameters

- **priority** (*int*) – Priority (mxrecord.priority)
- **fqdn** (*str*) – DNS Name (fqdn)

6.34 maltego.affiliation.Affiliation

class canari.maltego.entities.**Affiliation**(***kwargs*)

Parameters

- **uid** (*str*) – UID (affiliation.uid)
- **profile_url** (*str*) – Profile URL (affiliation.profile-url)
- **person_name** (*str*) – Name (person.name)
- **network** (*str*) – Network (affiliation.network)

6.35 maltego.Person

class canari.maltego.entities.**Person**(***kwargs*)

Parameters

- **lastname** (*str*) – Surname (person.lastname)
- **fullname** (*str*) – Full Name (person.fullname)
- **firstnames** (*str*) – First Names (person.firstnames)

6.36 maltego.affiliation.WikiEdit

class canari.maltego.entities.**WikiEdit**(***kwargs*)

Parameters

- **uid** (*str*) – UID (affiliation.uid)
- **profile_url** (*str*) – Profile URL (affiliation.profile-url)
- **person_name** (*str*) – Name (person.name)
- **network** (*str*) – Network (affiliation.network)

6.37 maltego.Domain

class canari.maltego.entities.**Domain**(***kwargs*)

Parameters

- **whois_info** (*str*) – WHOIS Info (whois-info)
- **fqdn** (*str*) – Domain Name (fqdn)

6.38 maltego.Vulnerability (alias: Vuln)

class canari.maltego.entities.**Vulnerability**(***kwargs*)

Parameters **id** (*str*) – ID (vulnerability.id)

6.39 maltego.Alias

class canari.maltego.entities.**Alias**(***kwargs*)

Parameters **alias** (*str*) – Alias (properties.alias)

6.40 maltego.Sentiment

class canari.maltego.entities.**Sentiment**(***kwargs*)

Parameters **sentiment** (*str*) – Sentiment (properties.sentiment)

6.41 maltego.Phrase

class canari.maltego.entities.**Phrase**(***kwargs*)

Parameters **text** (*str*) – Text (text)

6.42 maltego.affiliation.Twitter (alias: AffiliationTwitter)

class canari.maltego.entities.**Twitter**(***kwargs*)

Parameters

- **uid** (*str*) – UID (affiliation.uid)
- **screenname** (*str*) – Screen Name (twitter.screen-name)
- **profile_url** (*str*) – Profile URL (affiliation.profile-url)
- **person_name** (*str*) – Name (person.name)
- **number** (*int*) – Twitter Number (twitter.number)
- **network** (*str*) – Network (affiliation.network)
- **fullname** (*str*) – Real Name (person.fullname)
- **friendcount** (*int*) – Friend Count (twitter.friendcount)

6.43 maltego.BuiltWithTechnology

class canari.maltego.entities.**BuiltWithTechnology**(***kwargs*)

Parameters **builtwith** (*str*) – BuiltWith Technology (properties.builtwithtechnology)

6.44 maltego.Port

class canari.maltego.entities.**Port**(***kwargs*)

Parameters **number** (*str*) – Ports (port.number)

6.45 maltego.TwitterUserList

class canari.maltego.entities.**TwitterUserList**(***kwargs*)

Parameters

- **uri** (*str*) – URI (twitter.list.uri)
- **subscriber_count** (*str*) – Subscriber Count (twitter.list.subscribers)
- **slug** (*str*) – Slug (twitter.list.slug)
- **name** (*str*) – Name (twitter.list.name)
- **member_count** (*str*) – Member Count (twitter.list.members)
- **id** (*str*) – ID (twitter.list.id)
- **full_name** (*str*) – Full Name (twitter.list.fullname)
- **description** (*str*) – Description (twitter.list.description)

6.46 maltego.Company

class canari.maltego.entities.**Company**(***kwargs*)

Parameters **name** (*str*) – Name (title)

6.47 maltego.Website

class canari.maltego.entities.**Website**(***kwargs*)

Parameters

- **ssl_enabled** (*bool*) – SSL Enabled (website.ssl-enabled)
- **ports** (*int*) – Ports (ports)
- **fqdn** (*str*) – Website (fqdn)

6.48 maltego.Twit

class canari.maltego.entities.**Twit**(***kwargs*)

Parameters

- **title** (*str*) – Title (title)
- **pubdate** (*str*) – Date published (pubdate)
- **name** (*str*) – Twit (twit.name)
- **img_link** (*str*) – Image Link (img_link)
- **id** (*str*) – Twit ID (id)
- **content** (*str*) – Content (content)
- **author_uri** (*str*) – Author URI (author_uri)
- **author** (*str*) – Author (author)

6.49 maltego.affiliation.MySpace (alias: AffiliationMySpace)

class canari.maltego.entities.**MySpace**(***kwargs*)

Parameters

- **uid** (*str*) – UID (affiliation.uid)
- **profile_url** (*str*) – Profile URL (affiliation.profile-url)
- **person_name** (*str*) – Name (person.name)
- **network** (*str*) – Network (affiliation.network)

6.50 maltego.Image

class canari.maltego.entities.**Image**(***kwargs*)

Parameters

- **url** (*str*) – URL (fullImage)
- **description** (*str*) – Description (properties.image)

6.51 maltego.Hash

class canari.maltego.entities.**Hash**(***kwargs*)

Parameters

- **owner** (*str*) – Owner (owner)
- **included_media_types** (*str*) – Included Media Types (includeMediaType)
- **hash** (*str*) – Hash (properties.hash)
- **excluded_media_types** (*str*) – Excluded Media Types (excludeMediaType)
- **before** (*date*) – Before (before)
- **after** (*date*) – After (after)

6.52 maltego.Netblock

class canari.maltego.entities.**Netblock**(***kwargs*)

Parameters **ipv4range** (*str*) – IP Range (ipv4-range)

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index