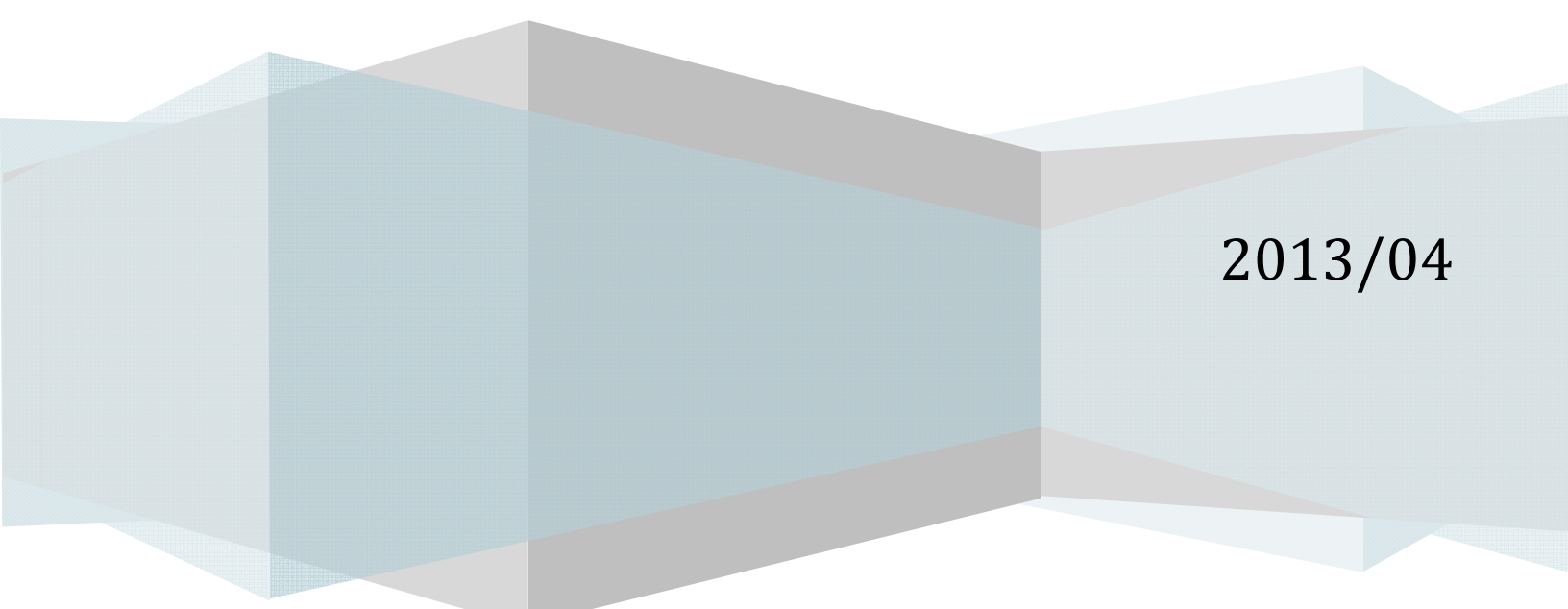


Maltego

TRX

Writing Python transforms (for use with the TDS)

RT



2013/04

Table of Contents

Introduction – why use the TDS?	4
What is TRX?	5
Architecture of the TDS.....	5
Developing transforms for the TDS	6
Publically available TDS.....	6
Internal/private TDS.....	6
The future of the TDS.....	6
Functionality of the NTDS	6
Pricing model of the NTDS.....	7
Development environments	7
PHP	7
Python	7
Prepping a server for use with the TDS (Python).....	7
Install Apache2	7
Install mod_wsgi.....	7
Install bottle	7
Copy & extract TRX files.....	8
Edit Apache configuration.....	8
Install TRX files	8
Restart Apache2	8
Setting up the TDS to use the example transform (DNS2IP).....	9
Setting up a seed	9
Setting up the transform.....	10
Setting up Maltego to use the sample transform.....	11
Under the hood.....	15
Debugging transforms.....	16
Changing to the bottle server.....	16
Deploying transforms with Apache.....	17
Multiple seeds	17
Components of TRX.....	17
debugTRX_Server.py	17
Transform library (e.g. DNSTRANSFORMS.py).....	18

Maltego library (maltego.py)	19
Using TRX.....	19
Basic use	19
Using Display Info.....	22
Link properties, bookmarks and notes.....	25
Entity properties	27
Static vs. dynamic properties.....	27
A quick note on entity design.....	27
Calculated properties	28
Property types.....	28
Reading properties.....	28
Adding dynamic properties	29
Using transform settings	32
Introduction	32
Transform setting types.....	35
Using transform settings in TRX.....	35
Conclusion	39
Entity reference.....	40
V2/V3 Confusion.....	40
Calculated properties.....	40
Inheritance	41
Why are we stuck with V2 properties?.....	41
Entity reference guide	42
The TRX Maltego.py API.....	47
MaltegoMsg class	47
MaltegoTransform class.....	47
MaltegoEntity class	48

Introduction – why use the TDS?

There are several ways to add custom transforms to Maltego. One option is to use local transforms. Local transforms are an easy way to get going but has several drawbacks in the long term:

1. Code base needs to be installed on local machine.
 - a. In many cases this means installing the development environment on local machine (think Python, PHP, PERL).
 - b. This makes it difficult to install on the client PC and means more moving parts.
 - c. Every transform needs to be manually registered on the Maltego GUI or Maltego configuration files needs to be patched. While this is feasible it might not be backward compatible with future releases of Maltego.
 - d. The code needs to be maintained for every installation of Maltego. A change in the transform code means it has to be re-installed on each Maltego client..
2. Limited functionality / access to Maltego features. The following features are not exposed to local transforms:
 - a. Slider value – the user cannot limit how many results are returned.
 - b. Transform settings – cannot access any transform settings. Transform settings are the pop-ups asking the user to configure a transform prior to running it.
 - c. Notes / link labels / bookmarking. Access to these fields will likely be removed from local transforms in future.
 - d. Meta data such as disclaimers etc.
3. When transforms depend on 3rd party software these need to be installed and maintained across all clients. It is not always feasible to install this type of software on each type of platform (e.g. if their Operating Systems are different).
4. No access control or auditing. Future releases of the TDS will include access control and auditing.
5. Transforms such as SocialNet (from Packetninja) are provided as a commercial product – as pay per transform. With local transforms it's more difficult for transform writers to offer their transforms as a service.
6. Hard to control intellectual property. As transforms run locally on the client it becomes difficult to protect the code against reverse engineering.
7. Paterva currently has no plans to provide future enhancements.

Most of these limitations can be somehow bypassed – but these shortcuts are clunky, hardly maintainable and not elegant at all.

Having said that - local transforms have the advantage that:

1. They are simple to write in any language.
2. They can interact with software that requires the use of a GUI. Think here of Skype that's difficult to use head-less.
3. Is infrastructure-less – it can run straight off your workstation.
4. Data does not travel over the wire – everything is contained in a single workstation.

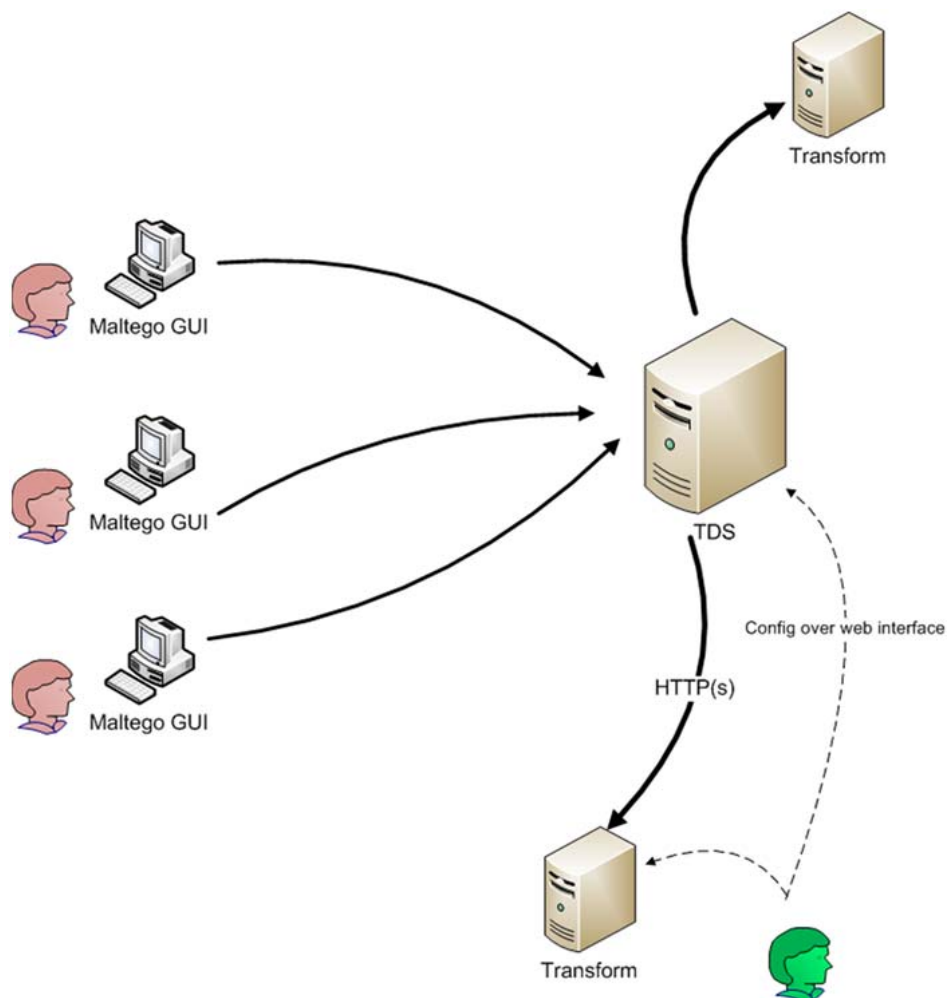
Local transforms are a good start for proof of concept code, rapid development or where interaction with GUI based software is required but in enterprise models or more serious deployments it does not scale well or provide the necessary control mechanisms. These limitations of local transforms are well known to Paterva. The more elegant solution to the problem would be to centralize the transforms on a server and provide access to run them remotely, similar to running the built-in transforms. Such a solution has been implemented using the Transform Distribution Server (TDS).

What is TRX?

TRX is a framework (although this perhaps a bit presumptuous) that facilitates coding Python transforms for Maltego using the TDS. It assumes the transform writer has access to of a web facing Ubuntu server. It uses Apache with mod_WSGI, Bottle and Python.

Architecture of the TDS

The following diagram shows how the Maltego clients, the transforms themselves and the TDS fit together:



The TDS acts like a proxy for transforms. It hides several of the complexities of seed management, entity meta-data, transform settings, transform discovery etc. from the transform writer and allows him/her to concentrate on developing the actual transform. The transform code is hosted on the end user's infrastructure and is served from a web server.

Configuration of transforms, seeds, transform settings and meta-data is performed via a web interface on the TDS.

Developing transforms for the TDS

There are currently two ways to develop transforms for the TDS.

Publically available TDS

This is a TDS that is located on the Internet and is free for all to use. It's a convenient way to immediately start writing transforms. Since this server is located on Paterva's infrastructure data will be flowing from the Maltego GUI to this server and finally to your transform code hosted on a web server of your choice. Note that your transforms need to be hosted on an Internet facing server. This is not the preferred way to deploy transforms in a production environment (especially if the transform content is sensitive) but it allows developers to get a feel for how the system works.

The public TDS requires registration and transforms and seeds are defined per user. The server interface lives at <https://cetas.paterva.com/TDS/>

Internal/private TDS

The internal TDS functions in the same way as the public TDS but can be hosted on the end-user's private infrastructure. It is delivered as Virtual Machine (VMWare image) and is commercial. Several high profile companies use the TDS in its current format.

Clients with a valid license for the TDS will automatically be upgraded to the NTDS (see next section).

The future of the TDS

The TDS was first released in 2010 and has seen very little changes since then. Initially Paterva was reluctant to sell the TDS to clients but since mid 2012 have lifted this restriction. Several high profile clients now use the TDS to centralize and distribute their custom enterprise wide transforms. This renewed attention to the TDS caused us to rethink the future of this server.

Functionality of the NTDS

The following key areas will be developed for the NTDS (New TDS):

1. Distribution of elements including:
 - a. Transforms (currently the only functionality of the TDS)
 - b. Entity definitions
 - c. Transform sets

- d. Machine scripts
- e. Viewlets
- 2. Access control and auditing
- 3. Transform stats, system health
- 4. Future proof for implementation of protocol 3.

Pricing model of the NTDS

Currently the TDS (when used as a private server) is sold at a fixed price but with unlimited amount of transforms and users. This “all or nothing” model has proved problematic for small enterprises that wish to share a handful of transforms amongst a small number of users. In future the NTDS pricing model will be based on users and transforms and should be priced more attractively for smaller enterprises.

The public TDS will also be upgraded to the new NTDS and **will remain free for use**.

Development environments

PHP

An extensive PHP library is provided for the TDS. The library and documentation can be found here: <https://www.paterva.com/web6/documentation/developer-tds.php#7>

Python

The rest of this document focuses around the Python development environment.

In the past Paterva only provided a CGI based Python library. This is easy to work with but has obvious limitations – for every transform that runs an instance of Python was started. This can easily drain resources on the host and is not very efficient. The more efficient method is using WSGI.

Prepping a server for use with the TDS (Python)

We recommend using Ubuntu Server with Apache2, Python 2.6, mod_wsgi and bottle. Assuming a stock standard Ubuntu server with Python installed the following is a recipe for quickly building your environment:

Install Apache2

```
sudo apt-get update
sudo apt-get install apache2
```

Install mod_wsgi

```
sudo apt-get install libapache2-mod-wsgi
```

Install bottle

```
sudo apt-get install python-setuptools
```

```
sudo easy_install bottle
```

Copy & extract TRX files

```
cd /tmp
wget http://www.paterva.com/TRX_Ubuntu.tgz
tar -xvzf TRX_Ubuntu.tgz
```

Edit Apache configuration

Edit the file `/etc/apache2/ports.conf` and add the line:

```
Listen 9001
```

Add it just below the line that reads `Listen 80` so the file looks like so:

```
...
NameVirtualHost *:80
Listen 80
Listen 9001
...
```

Copy the TRX Apache configuration file:

```
sudo cp /tmp/etc/apache2/sites-available/TRX /etc/apache2/sites-
available/
```

Now enable the site:

```
sudo a2ensite TRX
```

Install TRX files

```
sudo mkdir /var/www/TRX
cd /var/www/TRX
sudo cp /tmp/var.www.TRX.tgz .
sudo tar -xvzf var.www.TRX.tgz
```

An example transform `DNS2IP` is provided (it's a function defined in `DNSTRANSFORMS.py`, but more about that later). In the next section we will see how to configure the TDS to use this transform.

Restart Apache2

```
sudo /etc/init.d/apache2 restart
```

Feel free to inspect the configuration of the 'TRX' site defined in `/etc/apache2/sites-available` and customize this to your liking. The configuration will route all traffic on port 9001 to the WSGI script `TRX.wsgi` located in `/var/www/TRX`.

Setting up the TDS to use the example transform (DNS2IP)

Next we are going to configure the TDS to use the sample DNS2IP transform. Start by browsing to the TDS. If you are using an internal/private TDS it will ask you for your SSL certificate provided by Paterva. On the server portal you'll find instructions on how to install the supplied client certificate and use it.

If you are using the public TDS server you need to first log in with the user/password used at registration (and complete that horrible reCAPTCHA).

Once authenticated to the TDS you'll see three sections in the interface:

Paterva Transform Distribution Server

Welcome to the Paterva **Transform Distribution Server**. The following options are currently available:

- [Transforms](#) - Manage specific transforms as well as their properties, such as the URL to point to, transform settings and the input/output entity types.
- [Seeds](#) - Manage the seeds for this machine, specifically their names and which transforms they hold.
- [Transform Settings](#) - Manage the various transform settings available to the various transforms.

Setting up a seed

A seed is really just a URL that points to a group of transforms. It's this URL that Maltego will use to load all of the transforms contained in the seed into its internal configuration. Thus, the first step we want to do is create a new seed (container). Once we've done that we'll create a new transform and put the transform inside that seed.

Click on 'Seeds'. You'll see a list of seeds (if you have any). Click on the 'Add Seed' button'. You need to give a seed name, a seed URL and select which transforms you want to add into the seed. You can complete the form as follows:

Add Seed

Fill in the following form to add a new seed.

Seed Name	<input type="text" value="MyTDSSeed"/>
Seed URL	<input type="text" value="nobodywilleverguessthisone"/>
Transforms	<div><div></div><div>Add Another Transform</div></div>
<input type="button" value="Add Seed"/>	

When you're done, click on the 'Add Seed' button.

Now our seed is configured. It's empty – there are no transforms in the seed, but that's OK – we'll populate it in a bit.

Setting up the transform

You can easily navigate around with the 'bread > crumb' at the top of the screen:

Paterva Internal Transform Distribution Server

Paterva Internal Transform Distribution Server 0.1 > Seeds > Add

Click on the very first item – it will navigate to the root of the TDS.

Click on 'Transforms' – on the internal/private TDS you see a screen with a couple of PHP sample transforms. On the public TDS there won't be any transforms configured (unless you've added them there of course!)

Transforms

The following transforms have already been configured for this application, you can easily edit/delete these here as well as add new transforms.

Name	URL
toWikiEdits	http://localhost/TDSTransforms/wikipedia/searchWikipediaFromIP.php
DomainToSOAInformation	http://localhost/TDSTransforms/Infrastructure/soaLookup.php
DomainToSPFInformation	http://localhost/TDSTransforms/Infrastructure/spfLookup.php
NetblockToNetblocks	http://localhost/TDSTransforms/Infrastructure/netblockTonetblocks.php
AliasToTwitterUser	http://localhost/TDSTransforms/twitter/AliasToTwitterAccount.php
Add Transform	

We want to add a transform, so click on 'Add Transform'. You are now going to configure the transform. Here is a section of that webpage:

Add Transform

Fill in the following form to add a new transform.

Transform Details	
Transform Name	<input type="text" value="MyTransform"/>
Transform URL	<input type="text" value="http://"/>
Input Entity	<input type="text" value="Custom"/> <input type="text" value="iTDS.EntityName"/>
Owner	iTDS
Disclaimer	<input type="text"/>
Description (optional)	<input type="text"/>

Change the fields as follows:

Transform name:	DNS2IP
Transform URL:	http://<Your_DevServer_Here>:9001/DNS2IP
Input Entity:	maltego.DNSName (from the dropdown)
Disclaimer:	Leave empty unless you feel like putting something there
Description:	My first sample transform (or get creative)
Version:	1
Debug:	Checked
Transform settings:	Leave as is – we'll get to this later
Seeds:	Click on 'MyTDSSeed' to highlight.

If everything worked you should see 'Successfully Inserted 'DNS2IP' at the top of the screen:

Add Transform

Fill in the following form to add a new transform.

Successfully Inserted 'DNS2IP'

We now successfully added a transform to the seed and can use this seed in Maltego.

Setting up Maltego to use the sample transform

We now want to tell Maltego to discover transforms from our seed. Don't worry – most of this is a onetime configuration and once it's all set up we never have to touch it again.

We're lazy and so we are going to copy and paste the seed from the TDS. Go to Seeds and right click on the URL of the seed. Copy it to your clipboard:

Seeds

The Seed 'ExampleSeed' has been deleted!

The following seeds have already been configured for this application, you can easily edit/delete these here

Name	URI-Name	URL
MyTDSSeed	nobodywilleverguessthisone	https://10.77.0.106:8081/TDSRunner/runn

Add Seed

Open link in new tab

Open link in new window

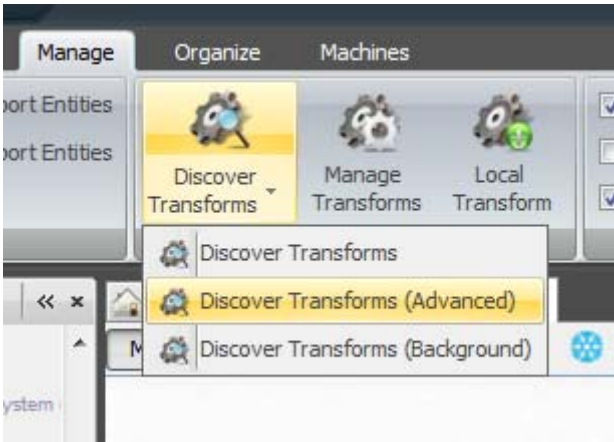
Open link in incognito window

Save link as...

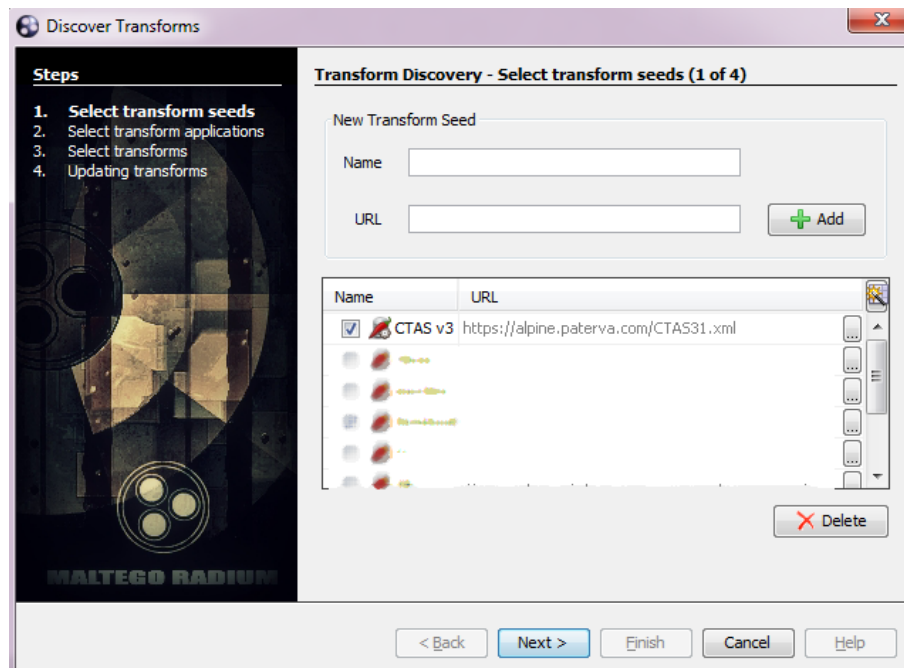
Copy link address

Inspect element

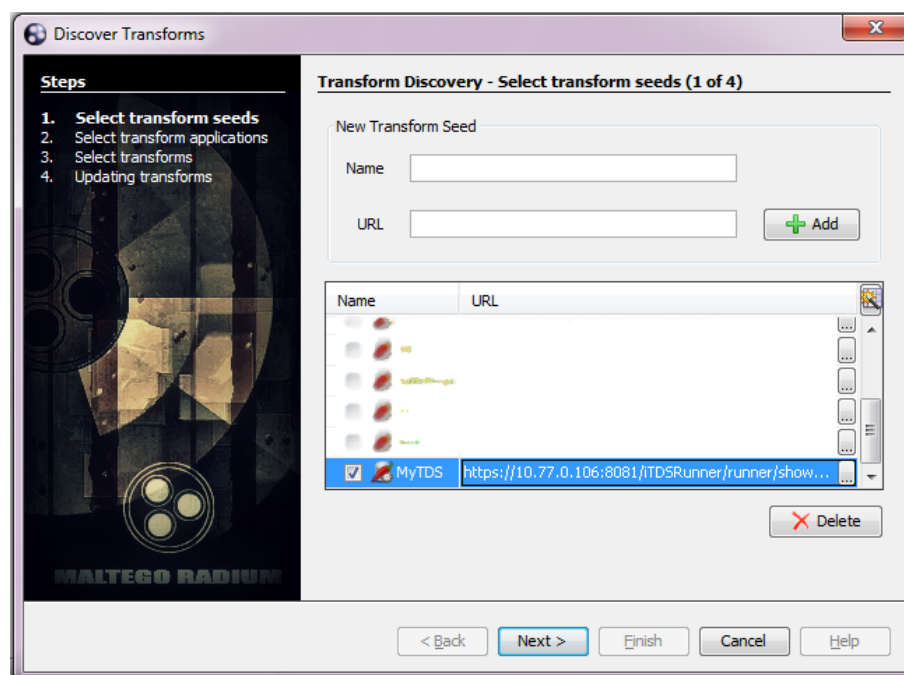
Open Maltego. Inside of Maltego go to the 'Manage' tab and click on the bottom of 'Discover Transforms'. You'll see a dropdown. Select 'Advanced':



The following screen should appear:

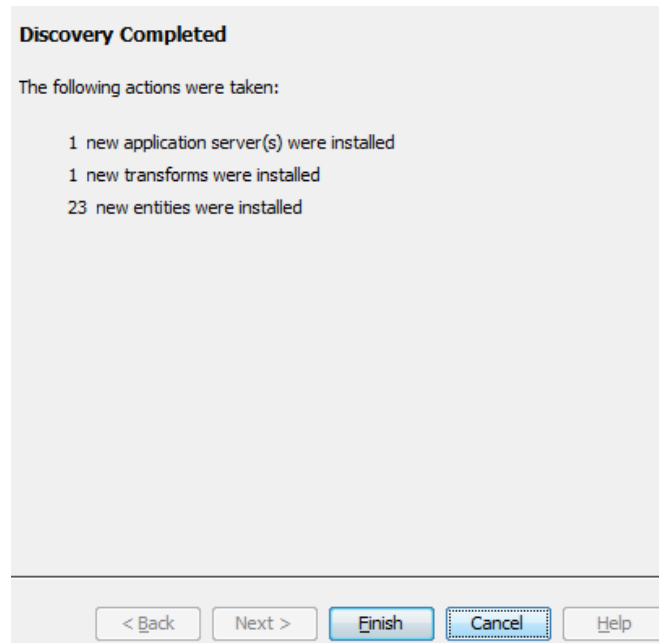


Hopefully yours would not be that blurry 😊. In the Name section type 'MyTDS' (it could be anything you want) and in the URL section paste the URL from the clipboard. Click on the 'Add' button and you'll see it appears in the list of Seeds below:



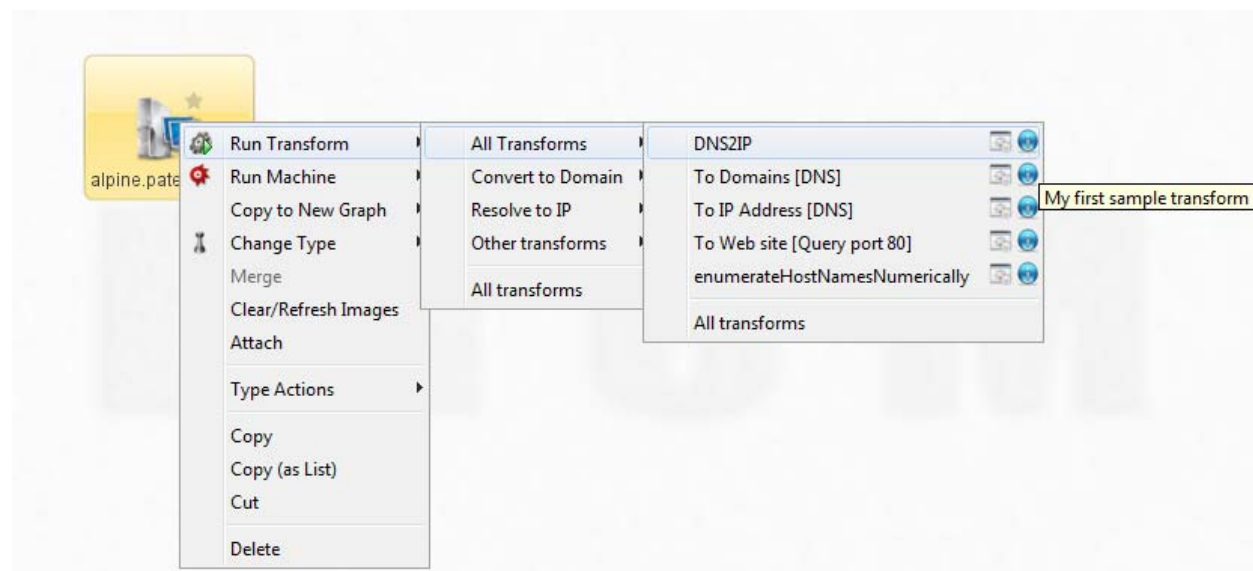
Now all that's left is to discover the seed. You may choose to uncheck your other seeds – it won't do you any harm, but it will ensure that you just discover from the TDS seed. Follow the wizard (next->next).

Finally, you'll see:



Don't worry about the '23 new entities were installed'. That's a legacy bug (actually it's not really, but that's an entire other chapter). Now you are ready to use the transform.

Open a new graph, drag a DNSName from the palette to the graph and right click. Voila – your transform shows up (complete with description):



The transform is fully functional and will resolve the DNS name to an IP address:



If you want to share this transform with anyone you can simply give them the URL and guide them through the discovery process in the same way as which you did. And – if you change the code on the server the change will transparently happen on all clients (as the transform runs on the server). You can now add more transforms to the seed. Maltego periodically checks for new transforms in the seeds – which mean you never need to touch the client side anymore. Joy!

Under the hood

Let's see what really happened here. These are the steps:

- 1) After you discovered from the TDS server Maltego knows that there's an additional transform called `DNS2IP` available on the entity type `Maltego.DNSName`.
- 2) When you right click on a DNS Name it makes this transform visible
- 3) It sends the transform request to the TDS server
- 4) The TDS server knows that the transform really lives on `http://<Your_DevServer_Here>:9001/DNS2IP`
- 5) The TDS makes a connection to port 9001 on the dev server and sends some XML in a POST
- 6) The Apache WSGI configuration on the dev server send all requests on port 9001 to a file located in `/var/www/TRX/TRX.wsgi`
- 7) `TRX.wsgi` sees that it's destined for `/DNS2IP` and routes it to the right piece of code
- 8) The Maltego Python library interprets the XML to something that's easy to work with in Python and passes it along
- 9) A DNS lookup is done
- 10) The reply is wrapped back into XML by the library and the data flows back the way it came in.

Seems like a mouthful. The actual transform code is quite simple and looks like this:

```
def trx_DNS2IP(m):
    TRX = MaltegoTransform()
    DNSName=None
    try:
        DNSName = socket.gethostbyname(m.Value)
        TRX.addEntity("maltego.IPv4Address",DNSName)
    except socket.error as msg:
        TRX.addUIMessage("Problem:"+str(msg),UIM_PARTIAL)

    return TRX.returnOutput()
```

Hardly rocket science.

Debugging transforms

Changing to the bottle server

One of the painful realities of working with WSGI is that you need to restart the Apache server every time you make a change to the code. There are some scripts on the Internet available to make this easier but this method of coding is altogether unintuitive for most coders. Luckily there's a way around it.

Bottle has its own web server built in and this server can be set to reload whenever a change in code happens – this goes for changes to the stub as well as any modules/libraries that it depends on. Let's look a little closer.

Go to `/var/www/TRX`. In here you'll see a file called `debugTRX_Server.py`. This file is a near exact copy of `TRX.wsgi` – but it can run as a standalone server. You'll see that we've uncommented the bottle HTTP server and commented the WSGI server (at the end of the script). In this case it's listening on the IP 10.77.0.106 and port 9001. Before you can run this script you need to disable the Apache server (because it is grabbing port 9001):

```
/etc/init.d/apache2 stop
```

Now you can start the (sub optimal, but great for debugging) server:

```
sudo python debugTRX_Server.py
```

It starts up as follows:

```
Bottle v0.11.6 server starting up (using WSGIRefServer())...
Listening on http://0.0.0.0:9001/
Hit Ctrl-C to quit.
```

What's great about this server is that it automatically reloads every time you make a change to the code. Better still, it verbosely complains about any mistake you might have made in your code. Here's a screenshot of that happening:


```
root@ubuntu: /var/www/TRX
I A DnsTransforms. Row 15 Col 52 6:29 Ctrl-K H for help
import socket;
from Maltego import *

def trx_DNS2IP(m):
    TRX = MaltegoTransform()

    DNSName=None
    try:
        DNSName = socket.gethostbyname(m.Value)
        TRX.addEntity("maltego.IPv4Address",DNSName)
    except socket.error as msg:
        TRX.addErrorMessage("Error:"+str(msg),UIM_PARTIAL)

    #Write the slider value as a UI message - just for fun
    TRX.addUIMessage("Slider value is at: "+str(m.Slider))

    return TRX.returnOutput()

def trx_EnumAS(m):
    TRX = MaltegoTransform()

    if (not m.Value.isdigit()):
        TRX.addUIMessage('Sorry but [' +m.Value+'] is not a whole n
    return TRX.returnOutput()

    #here we know we're good to go.
    howmany = int(m.Value);

File DnsTransforms.py saved

Hit Ctrl-C to quit.

Traceback (most recent call last):
  File "/usr/local/lib/python2.6/dist-packages/bottle-0.11.6-py2.6.egg/bottle
.py", line 764, in _handle
    return route.call(**args)
  File "/usr/local/lib/python2.6/dist-packages/bottle-0.11.6-py2.6.egg/bottle
.py", line 1575, in wrapper
    rv = callback(*a, **ka)
  File "debugTRX.py", line 14, in DNS2IP
    return(trx_DNS2IP(MaltegoMsg(request.body.getvalue())))
  File "/var/www/TRX/DnsTransforms.py", line 15, in trx_DNS2IP
    TRX.addUIMessage("Slider value is at: "+str(m.Slider))
AttributeError: MaltegoMsg instance has no attribute 'Slider'

10.77.0.106 - - [30/Mar/2013 06:27:48] "POST /DNS2IP?value=YwxwaW5LnBhdGVydm
EuY29t&fields=ZnFkbj1hbHBpbmUucGF0ZXJ2YS5jb20j HTTP/1.1" 500 1675
Bottle v0.11.6 server starting up (using WSGIRefServer())...
Listening on http://10.77.0.106:9001/
Hit Ctrl-C to quit.

10.77.0.106 - - [30/Mar/2013 06:28:18] "POST /DNS2IP?value=YwxwaW5LnBhdGVydm
EuY29t&fields=ZnFkbj1hbHBpbmUucGF0ZXJ2YS5jb20j HTTP/1.1" 200 306

4) We retry
Request is now 100% !
```

In the example above we made a silly mistake in a transform by using `m.slider` and not `m.Slider` (note the difference in case). The bottle server complained bitterly about it. We fixed the code and as soon as we saved the file the server reloaded with the changes. (If you wondering – we had our server listen on 10.77.0.106 and not on 0.0.0.0.)

Deploying transforms with Apache

The main reason why we don't want to run the bottle server in production is because it's not optimized for heavy load and we don't want to have to struggle with start-up scripts.

Once you are happy with your transforms you should stop the bottle server, simply uncomment the first couple of lines, change the server to run as a WSGI and start Apache again:

```
/etc/init.d/apache2 start
```

Multiple seeds

Another way to do this is to run the bottle server on a different port. You might want to add another seed ('DEV?') and register the transforms on this port. Once you want to move it into production you can simple change the port and insert it into the production seed.

Components of TRX

There are basically three files that are of interest. All of them are located in `/var/www/TRX/`.

debugTRX_Server.py

This is the dispatcher (`TRX.wsgi` and `debugTRXServer.py` is essentially the same script – the former used with Apache, the latter when debugging). Let's look at the file in a little bit more detail:

```

import os,sys

os.chdir(os.path.dirname(__file__))
sys.path.append(os.path.dirname(__file__))

from bottle import *
from Maltego import *
from DNSTRANSFORMS import *

##### TRANSFORM DISPATCHER
@route('/DNS2IP', method='ANY')
def DNS2IP():
    if request.body.len>0:
        return(trx_DNS2IP(MaltegoMsg(request.body.getvalue())))

@route('/Silly', method='ANY')
def silly():
    if request.body.len>0:
        return(trx_Silly(MaltegoMsg(request.body.getvalue())))

```

Maltego library

Transform code library

The transform URL (goes into TDS)

Router for Transform 1

The transform code lives here

Router for Transform 2

Each transform has a small router stub in here that defines the URL of the transform and to which function it routes to. The 'request.body.getvalue()' reads the entire POST from the message. This XML is sent to the function 'MaltegoMsg' which is defined in the Maltego library. The function returns an object which can be used to easily extract all the information from the XML. This object is passed along to the actual transform. The transform does its work and returns XML - this is returned back to the TDS. We'll look into the transform code in more detail shortly.

In the example we've included the library 'DNSTRANSFORMS'. This is the transform code library - e.g. where the transform itself is defined. In the router for transform 1 (DNS2IP) case it's called 'trx_DNS2IP'. Of course, as you go along you may choose to add more libraries.

Transform library (e.g. DNSTRANSFORMS.py)

You can include as many libraries as you want. For our example we used DNSTRANSFORMS.py. Let's take a look at what's really happening:

```

import socket;
from Maltego import *

def trx_DNS2IP(m):
    TRX = MaltegoTransform()
    DNSName=None
    try:
        DNSName = socket.gethostbyname(m.Value)
        TRX.addEntity("maltego.IPv4Address",DNSName)
    except socket.error as msg:
        TRX.addUIMessage("Problem:"+str(msg),'PartialError')

    #Write the slider value as a UI message - just for fun
    TRX.addUIMessage("Slider value is at: "+str(m.Slider))

    return TRX.returnOutput()

```

Transform request object from router

Maltego return object

Value of node

Slider's value

Returns object in XML

This is the code that actually performs the work of the transform. As input it gets an object that contains all the details about the request – this is called ‘m’ in this case. Next, it creates a ‘vessel’ that will eventually contain the response – here it’s called ‘TRX’. It then does the DNS lookup and adds an entity of type ‘maltego.IPv4Address’ with the value of the lookup to the vessel. It also adds some UIMessages. Finally the transform returns the vessel’s XML to the router which in turn sends it back to the TDS.

Maltego library (maltego.py)

This is the library that shuffles XML to object and back. You are more than welcome to optimize, tinker with this library should you feel inclined to do so. But keep in mind that should we change the library you’ll need to redo any changes you’ve made to it.

Using TRX

Basic use

Let’s look at a couple of examples. You might want to refer to the TRX reference guide at the end of this document every now and again to follow we are using the library.

Let’s say we want to create a transform that runs on a Phrase entity. It will assume that the phrase is a number and it will return as many entities – and just as a test – we’ll return these as AS number entities.

Let’s begin really simple:

Our stub looks like this (in TRX.wsgi or debugTRX_Server.py):

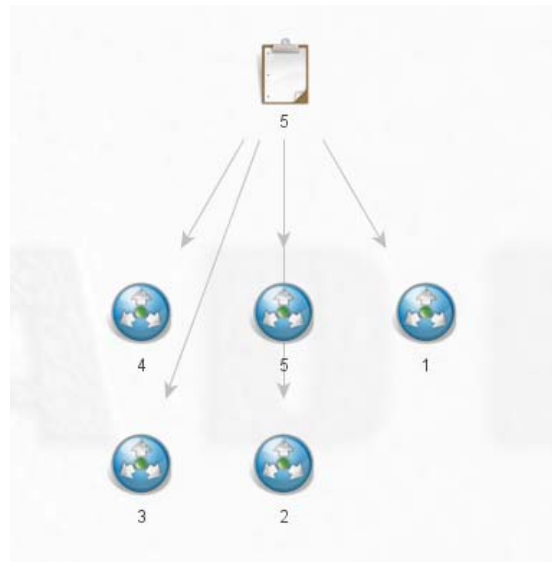
```
@route('/EnumAS', method='ANY')
def EnumAS():
    if request.body.len>0:
        return(trx_EnumAS(MaltegoMsg(request.body.getvalue())))
```

Our transform looks like this (in DNSTRANSFORMS.py – too lazy to put in another library):

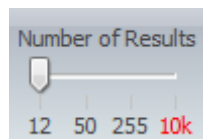
```
def trx_EnumAS(m):
    TRX = MaltegoTransform()
    howmany = int(m.Value)
    for i in range(1,howmany+1):
        TRX.addEntity('maltego.AS', str(i))
    return TRX.returnOutput()
```

We call it ‘EnumAS’ on the TDS and point it to a URL <http://server:9001/EnumAS> . Remember that if you used Apache as a server you will need to restart the server to register the changes (/etc/init.d/apache2 reload). If you used the bottle server and it is running you shouldn’t have to do a thing.

Here is the output:



This code is really nasty. What happens if we put the value '250000' in the graph? What if it's not really a number but a phrase like 'test'? So, as a start, let's make sure we don't go over the limit set by the user using the result slider:



We change the code to read the slider:

```
def trx_EnumAS(m):  
    TRX = MaltegoTransform()  
  
    howmany = int(m.Value)  
  
    if (howmany > m.Slider):  
        howmany=m.Slider  
  
    for i in range(1,howmany+1):  
        TRX.addEntity('maltego.AS', str(i))  
    return TRX.returnOutput()
```

Now we at least honor the user's wishes on how many entities to return. Next, we want to make sure it's a number and if it's not, return a nice message to the user saying he/she is not understanding the transform. The code now changes to this:

```
def trx_EnumAS(m):  
    TRX = MaltegoTransform()
```

```

if (not m.Value.isdigit()):
    TRX.addUIMessage('Sorry but [' + m.Value + '] is not a whole
number', UIM_PARTIAL)
    return TRX.returnOutput()

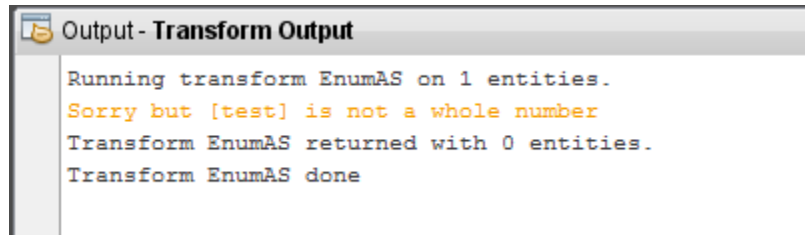
#here we know we're good to go.
howmany = int(m.Value)

if (howmany > m.Slider):
    howmany=m.Slider

for i in range(1,howmany+1):
    TRX.addEntity('maltego.AS', str(i))
return TRX.returnOutput()

```

Right! When we now run our transform on an entity that's not a number we get the following:



It seems that everything is mostly sorted out, but there's one more thing. You'll notice that the entities seem to be laid out on the graph in a random fashion. Maltego actually lays out entities according to their weight (top left to bottom right). Since we didn't set the weight all the entities have the same weight – the default of 100. Let's see if we can fix this:

```

def trx_EnumAS(m):
    TRX = MaltegoTransform()

    if (not m.Value.isdigit()):
        TRX.addUIMessage('Sorry but [' + m.Value + '] is not a whole
number', UIM_PARTIAL)
        return TRX.returnOutput()

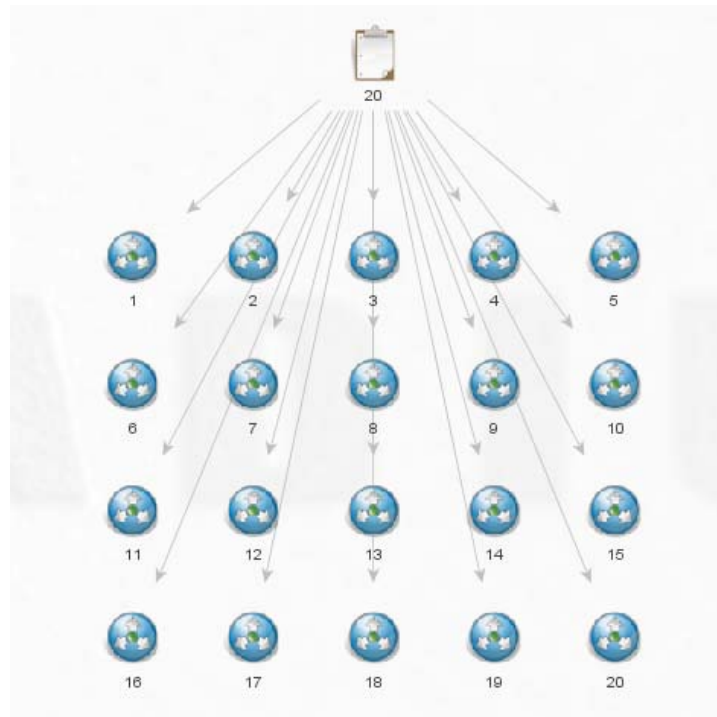
    #here we know we're good to go.
    howmany = int(m.Value)

    if (howmany > m.Slider):
        howmany=m.Slider

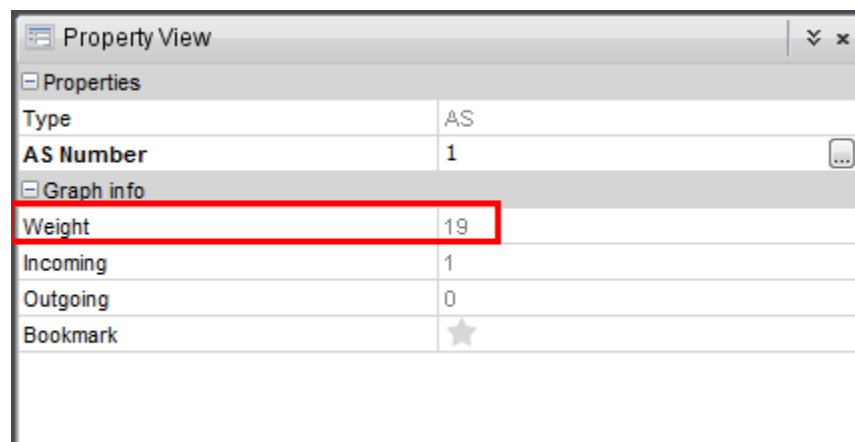
    for i in range(1,howmany+1):
        Ent = TRX.addEntity('maltego.AS', str(i))
        Ent.setWeight(howmany-i)
    return TRX.returnOutput()

```

Here you see that the **addEntity** method actually returns a **MaltegoEntity** object which we can modify. In this case we use it to set the weight and now the graph looks as follows:



The first node (with value '1') has a weight of 19 and the last node ('20') has a weight of 0. You can see this in the Property View. The following screenshot shows the property view when the AS node with value '1' was selected:



Using Display Info

Let's next assume we want to create some fancy HTML in the Detail View to go with our AS entities. The **'addDisplayInformation'** method is used to do this. The content could be plain text or HTML. Let's start with something really simple. We want to display the words "This is number X" on each entity returned. We want this to be rendered in a label called 'AS Number'. Here is how the script looks now:

```

def trx_EnumAS(m):
    TRX = MaltegoTransform()

    if (not m.Value.isdigit()):
        TRX.addUIMessage('Sorry but ['+m.Value+'] is not a whole
number',UIM_PARTIAL)
        return TRX.returnOutput()

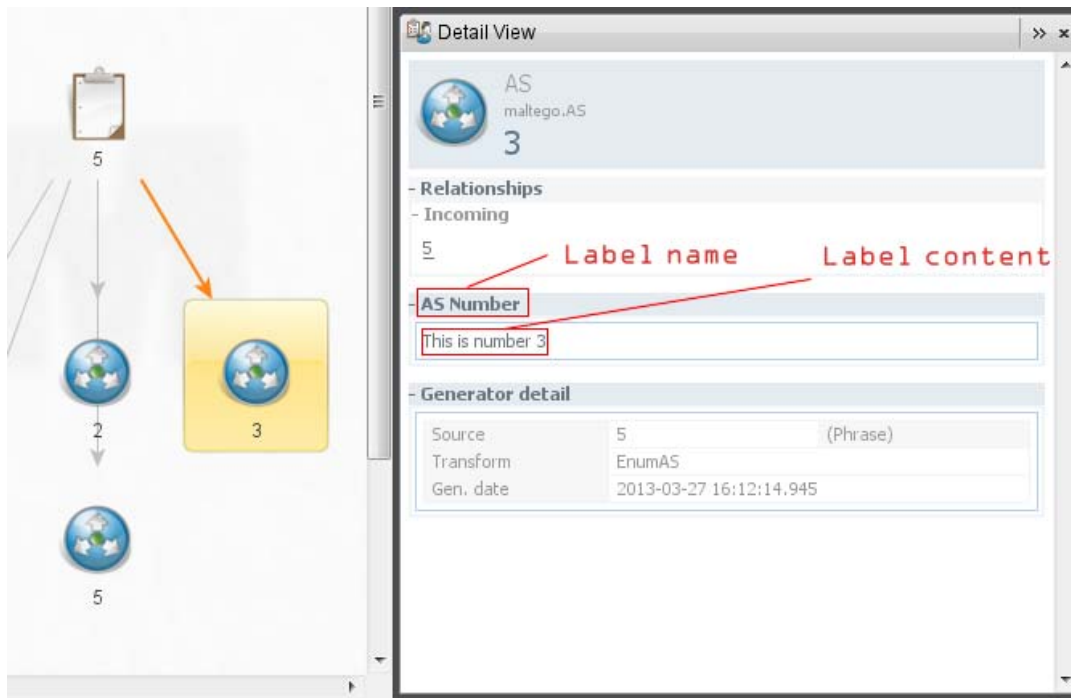
    #here we know we're good to go.
    howmany = int(m.Value)

    if (howmany > m.Slider):
        howmany=m.Slider

    for i in range(1,howmany+1):
        Ent = TRX.addEntity('maltego.AS', str(i))
        Ent.setWeight(howmany-i)
        Ent.addDisplayInformation("This is number "+str(i),"AS
Number")
    return TRX.returnOutput()

```

When you run the transform you'll see that each node rendered the label and content:



You can easily add more labels. Display information is merged when other transforms creates the same entity instance. Maltego will expand the label by default, but Detail View labels can be collapsed and expanded by the user and the client will remember the choice for future operations.

Let's see how it works with HTML. Let's assume we want to make a click-able link in HTML. We will create the link so that it is dynamic and based on the node number. Consider the following code:

```
def trx_EnumAS(m):
    TRX = MaltegoTransform()

    if (not m.Value.isdigit()):
        TRX.addUIMessage('Sorry but ['+m.Value+'] is not a whole
number', UIM_PARTIAL)
        return TRX.returnOutput()

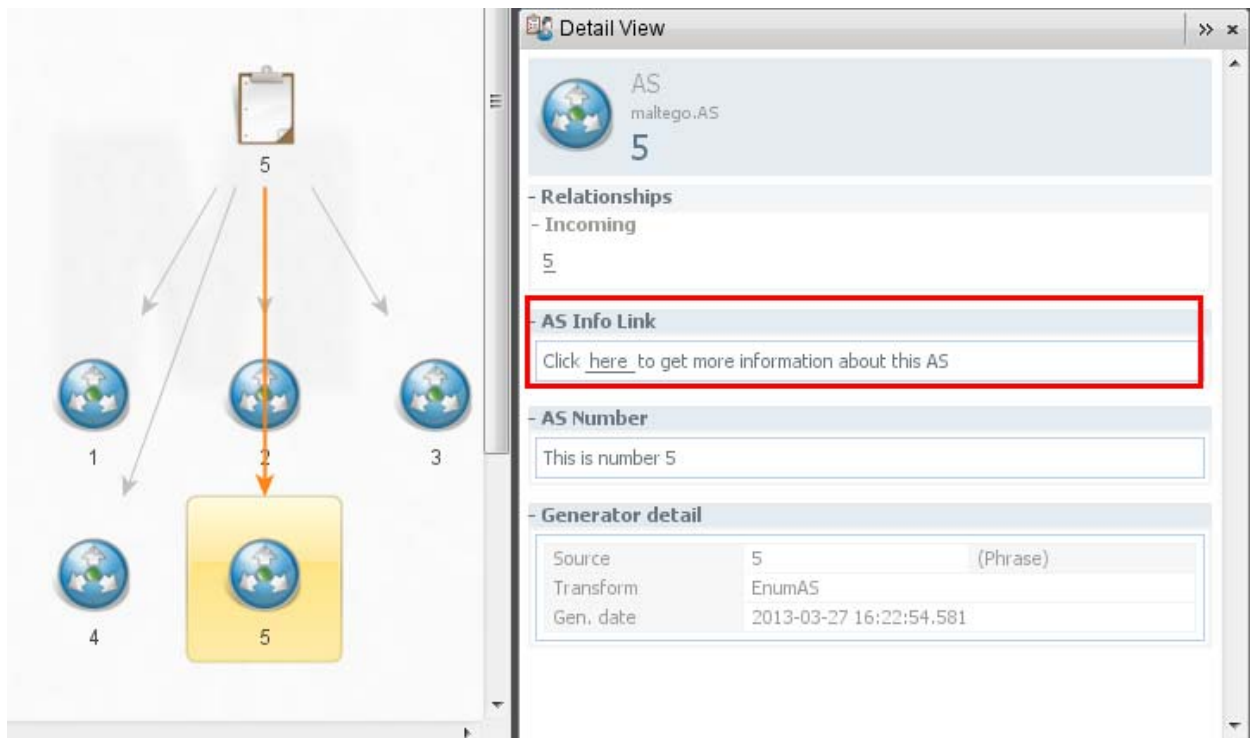
    #here we know we're good to go.
    howmany = int(m.Value)

    if (howmany > m.Slider):
        howmany=m.Slider

    for i in range(1,howmany+1):
        Ent = TRX.addEntity('maltego.AS', str(i))
        Ent.setWeight(howmany-i)
        Ent.addDisplayInformation("This is number "+str(i),"AS
Number")
        Ent.addDisplayInformation('Click <a
href="https://www.ultratools.com/tools/asnInfoResult?domainName=AS'+st
r(i)+'"> here </a> to get more information about this AS','AS Info
Link')

    return TRX.returnOutput()
```

Now each node has a link that will open the browser to a specific webpage:



Clicking on the links gives us:

UltraTools™
Powered by Neustar® UltraDNS®

REGISTERED USERS: 16,479

Email Address Password Login
Remember me Forget your password?

Home Domain Health Report **NEW WHOIS+** Monitoring UltraTools Statistics Community UltraTools Mobile Create Free Account

IP Tools

- Decimal IP Calculator
- ASN Information**
- CIDR/Netmask
- What's your IP
- IP Geo-location Lookup
- IPWHOIS Lookup

ASN Information

The ASN Information tool provides complete autonomous system (AS) information.

Enter an AS number, IP address, or a Company name.

AS5 Go »

Related Tools: [CIDR/Netmask](#) [What's your IP](#) [Decimal IP Calculator](#)

AS5

Country:	US
Registration Date:	1984-02-02
Registrar:	arin
Owner:	SYMBOLICS - Symbolics, Inc.

It's important to note that display information is never sent to the server (when you run subsequent transforms on the node) – it's read only info and is kept in the client.

Link properties, bookmarks and notes

You can set the link color, label, thickness and style really easily. Consider this code snippet (used in the same transform). We are not going to discuss each method in detail because it's rather obvious from the code and the resultant graph:

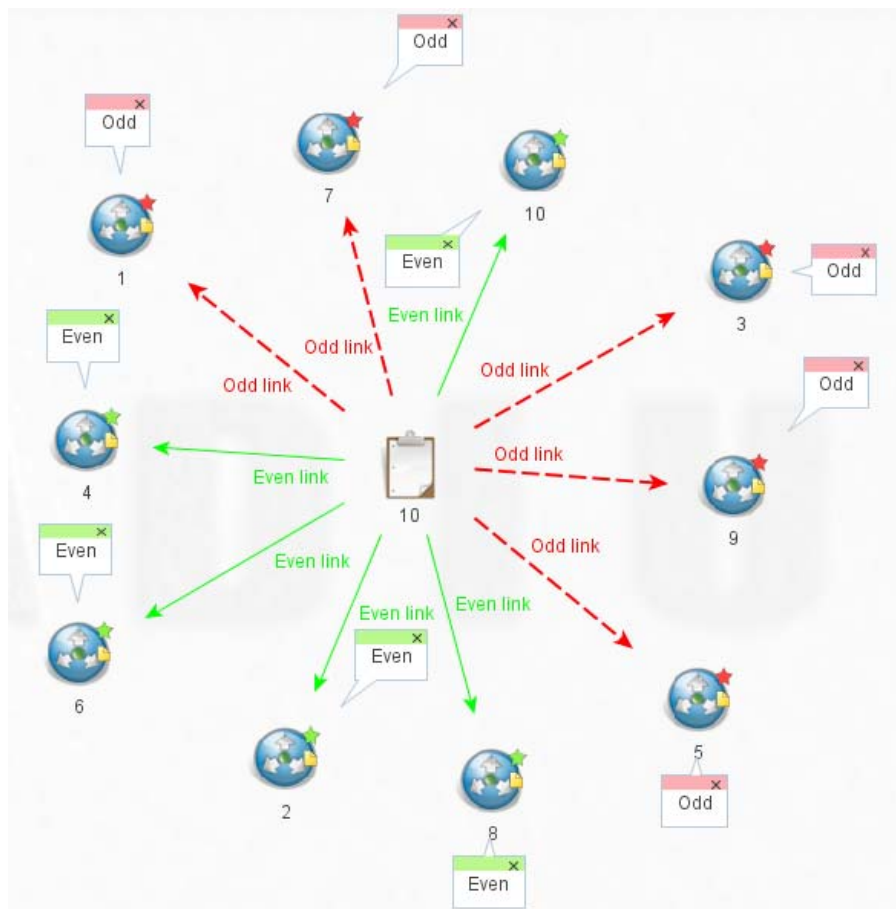
```
if (i%2==0):
    Ent.setLinkColor('0x00FF00')
```

```

Ent.setNote('Even')
Ent.setLinkLabel('Even link')
Ent.setLinkStyle(LINK_STYLE_NORMAL)
Ent.setLinkThickness(1)
Ent.setBookmark(BOOKMARK_COLOR_GREEN)
else:
Ent.setLinkColor('0xFF0000')
Ent.setNote('Odd')
Ent.setLinkLabel('Odd link')
Ent.setLinkStyle(LINK_STYLE_DASHED)
Ent.setLinkThickness(2)
Ent.setBookmark(BOOKMARK_COLOR_RED)

```

Although we recommend you keep it a little simpler this code illustrates the concept. The resultant graph (ugly) looks like this (layout in organic mode - else it's REALLY messy):



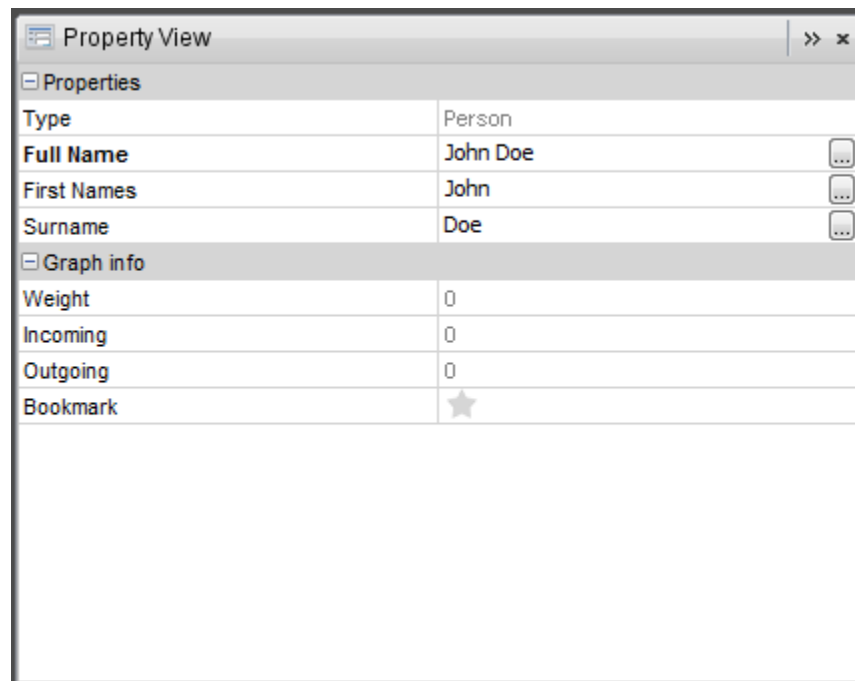
The only thing to keep in mind is that entity notes stack on top of each other – that means that when a subsequent transform writes to a note it will append to the note rather than deleting the previous note.

Entity properties

Entities can have additional properties. For example – a person might have a property called ‘SSN’.

Static vs. dynamic properties

When you design an entity you’ll see that you can add additional properties to the entity. These are called static properties. It means that should a user drag an entity from the palette to the graph these properties will be available for editing. As an example, here is the property view for Person:



The screenshot shows a 'Property View' window with a title bar containing a maximize icon, a close icon, and a '>>' icon. The window is divided into two main sections. The top section is titled 'Properties' and contains a table with the following rows: 'Type' with the value 'Person', 'Full Name' with the value 'John Doe', 'First Names' with the value 'John', and 'Surname' with the value 'Doe'. Each of the last three rows has a small three-dot menu icon to its right. The bottom section is titled 'Graph info' and contains a table with the following rows: 'Weight' with the value '0', 'Incoming' with the value '0', 'Outgoing' with the value '0', and 'Bookmark' with a star icon.

Properties	
Type	Person
Full Name	John Doe
First Names	John
Surname	Doe

Graph info	
Weight	0
Incoming	0
Outgoing	0
Bookmark	★

Transforms can create new properties for entities they return. The transform simply adds a property to the entity when it’s returned. These are called dynamic properties. If a transform returns a property that had been defined as static in the entity definition it will store the value as a static property, if it’s not defined in the entity definition it then becomes dynamic.

The only real difference between static and dynamic properties is that users cannot edit dynamic properties on a fresh node from the palette...simply because the property does not exist yet - a transform needed to create it.

A quick note on entity design

One of the most important questions to ask yourself when doing entity design is if data should be stored in a property of an entity or if it should become a new entity type. Similarly you should consider if some information should only be display information or stored as a property within an entity.

Usually you can answer these questions as follows. If you can think of a transform that will regularly use a property of an entity it would be better to make it a separate entity type. For instance – it would be better to have SSN as an entity type and have a Person to SSN transform than to only have SSN as a property in the Person entity (you might still store the SSN as a property

within the Person entity, but it makes sense to have it as a separate entity too). Similarly if you cannot decide if something should be a property or simply display info consider if another transform would need to read that property to work properly or if it's only shown to the user for information (because display info is never passed back to the server). If it's just useful as display info don't bloat the entity by adding it as a property.

Calculated properties

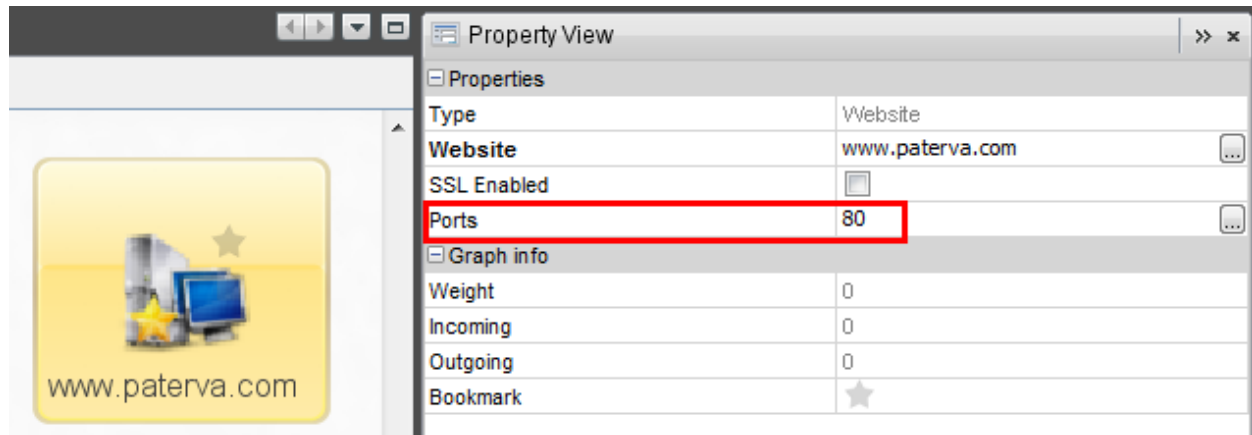
In some cases properties are calculated from other properties. In the case of a person the property FullName is created from FirstName and LastName. This does not mean you cannot set the property, but you don't need to. For a full list of entity names, properties and calculated properties refer to the Entity definition reference at the end of this document.

Property types

The current version of the Python library only supports string types but subsequent releases might extend this to include all the other types of properties that Maltego support. When dealing with entities that have property types other than strings, make sure you format the string value properly so that the Maltego client can parse the correct type from the string value.

Reading properties

This is fairly straightforward. Consider the Website entity. It has a static property called 'ports' with a default of 80 (see Entity definition reference for a list of static entity properties). Inside of Maltego this is really defined as a list of integers but we'll see how we deal with it in TRX.



Let's start a new transforms. We'll call it 'Mangle', register it on the TDS (running on a website entity) and in our TRX.wsgi stub.

Our transform code looks like this:

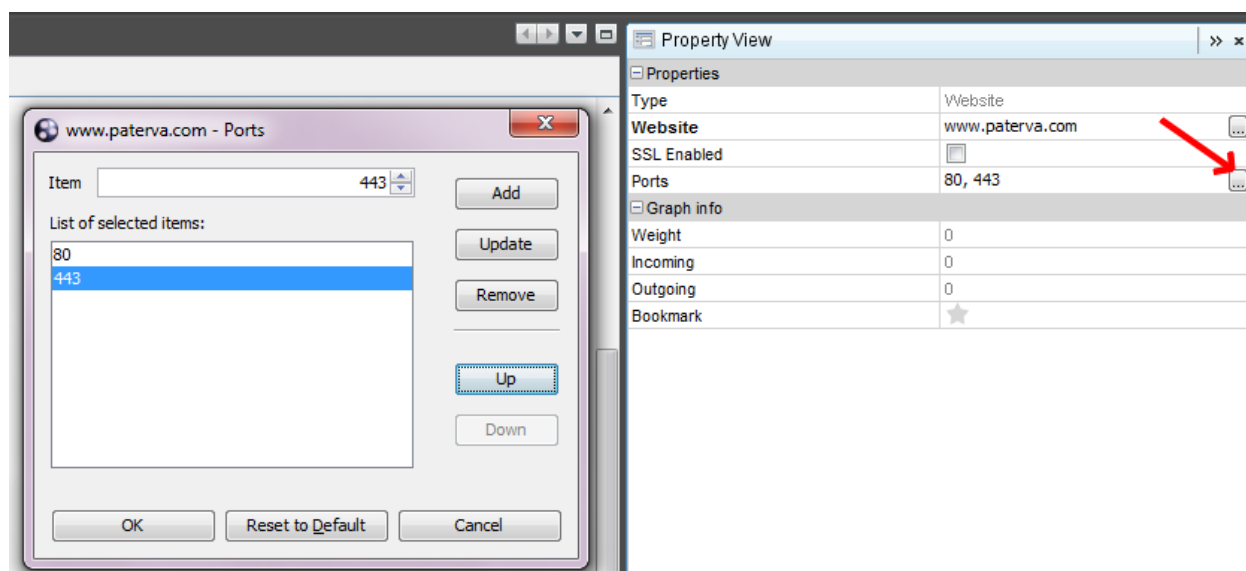
```
def trx_Mangle(m) :  
    #define response  
  
    TRX = MaltegoTransform()  
  
    TRX.addUIMessage("Property value is:"+m.getProperty("ports"))
```

```
return TRX.returnOutput()
```

We use the '**getProperty**' method to get the property's value. You will see that we do not return any entities (that's 100% OK), we simply write a message to the output with the Output screen. On a default website entity the output looks like this:

```
Running transform Mangle on 1 entities.  
Property is:80  
Transform Mangle returned with 0 entities.  
Transform Mangle done
```

If we add a port to the list...



...and run the transform again we get:

```
Running transform Mangle on 1 entities.  
Property is:80, 443  
Transform Mangle returned with 0 entities.  
Transform Mangle done
```

In other words – the list is returned as a comma delimited list.

Adding dynamic properties

Let's see how we can write properties using TRX. Each property has

- Displayname : how it's rendered in Maltego
- Name : the actual name used in code
- Value : the value of the property
- Matching rule : if entities with the same value, type and property value should be merged

Let's assume we want to create a transform that runs on an IP address and generates a netblock. It will always assume a class C netblock (255 IPs) but it will add a dynamic property to the netblock to indicate if it's a private netblock or not. We'll do this with a dynamic property called 'private' and it will be set to 'yes' or 'no'.

We start by building a stub (in `TRX.wsgi` for Apache deployment or `debugTRX_Server.py` when debugging) and registering our transform on the TDS. We begin our transform (without input validation – it's not the point) like this:

```
def trx_NetblocksRUs(m):
    TRX=MaltegoTransform()

    start=m.Value[0:m.Value.rfind('.')]
    netblock=start+".0-"+start+".255"

    Ent = TRX.addEntity('maltego.Netblock')
    Ent.setValue(netblock)

    return TRX.returnOutput()
```

Note that at this point we haven't added any dynamic properties to the netblock. We need to first have something that checks if an IP address is private or not. A quick search on the Internet finds this useful piece of code:

```
>>> from IPy import IP
>>> ip = IP('10.0.0.1')
>>> ip.itype()
'PRIVATE'
```

This function relies on a Python module called `IPy`. On Ubuntu installing this is as simple as:

```
sudo apt-get install python-ipy
```

This seems perfect for our transform. We add a line importing this library to our `DNSTRANSFORMS.py`:

```
from IPy import IP
```

And the transform now looks like this:

```
def trx_NetblocksRUs(m):
    TRX=MaltegoTransform()

    start=m.Value[0:m.Value.rfind('.')]
    netblock=start+".0-"+start+".255"

    Ent = TRX.addEntity('maltego.Netblock')

    ip = IP(m.Value)
```

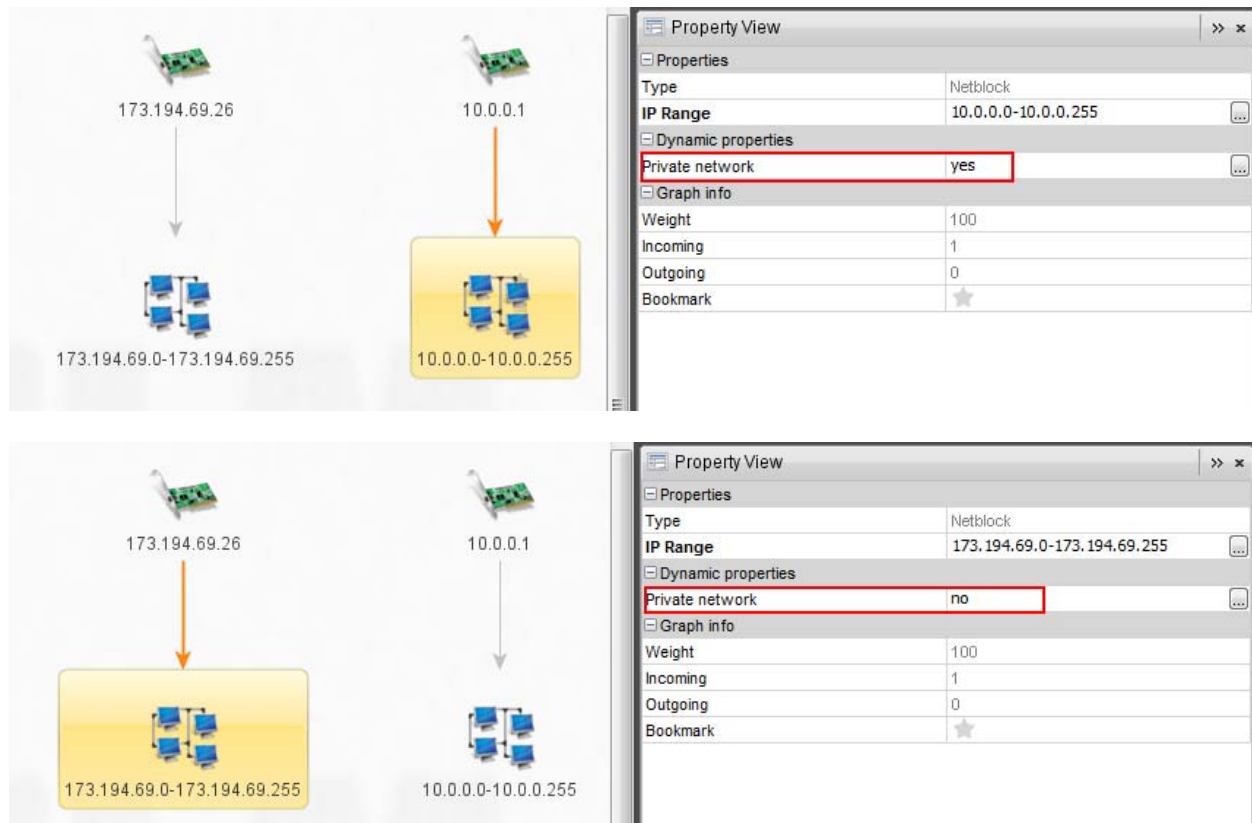
```

if (ip.iptype()=='PRIVATE'):
    Ent.addProperty('private','Private network','strict','yes')
else:
    Ent.addProperty('private','Private network','strict','no')

Ent.setValue(netblock)
return TRX.returnOutput()

```

Now running the transform on two different IP addresses we get:



Perfect! Of course we can create a transform that uses netblocks as input and simply outputs a phrase 'public' or 'private' (this is not very useful, but hopefully will show how to use additional properties).

```

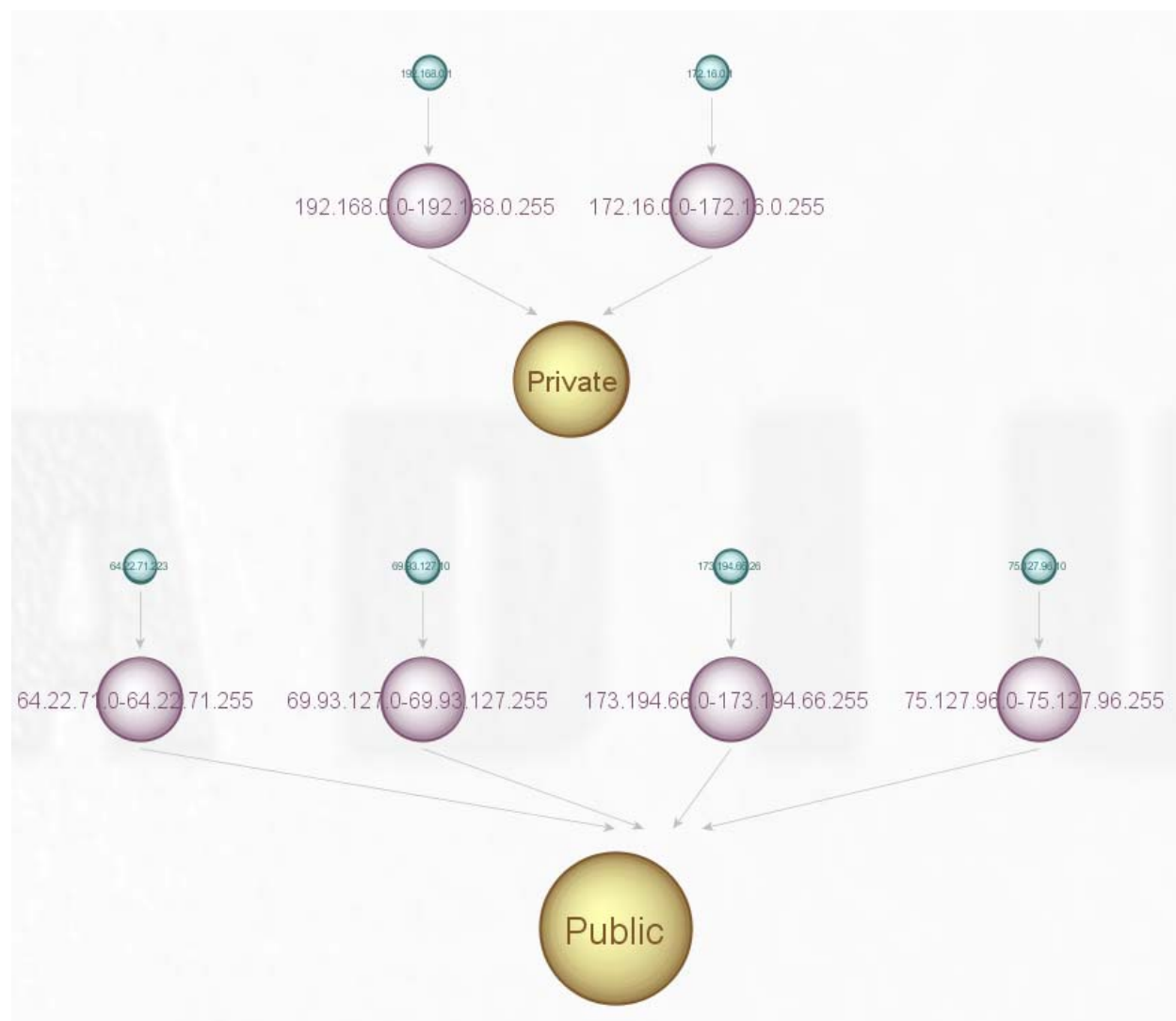
def trx_PublicPrivate(m):
    TRX=MaltegoTransform()
    Ent = TRX.addEntity('maltego.Phrase')

    if m.getProperty('private')=='yes':
        Ent.setValue('Private')
    else:
        Ent.setValue('Public')

    return TRX.returnOutput()

```

The resultant graph (when populated with some IPs->Netblock->Public/Private and graph set to Bubble view block layout):

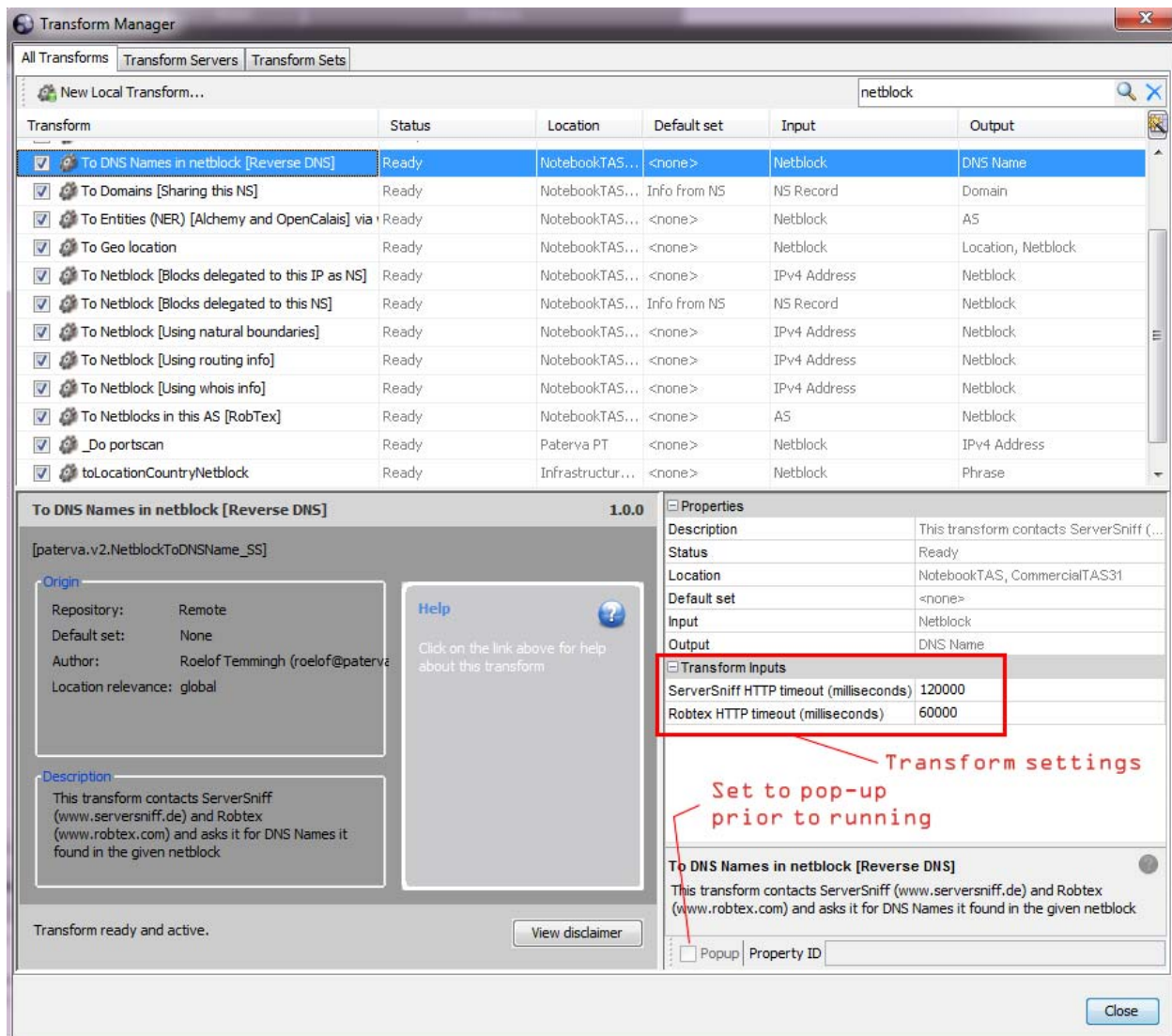


Using transform settings

Introduction

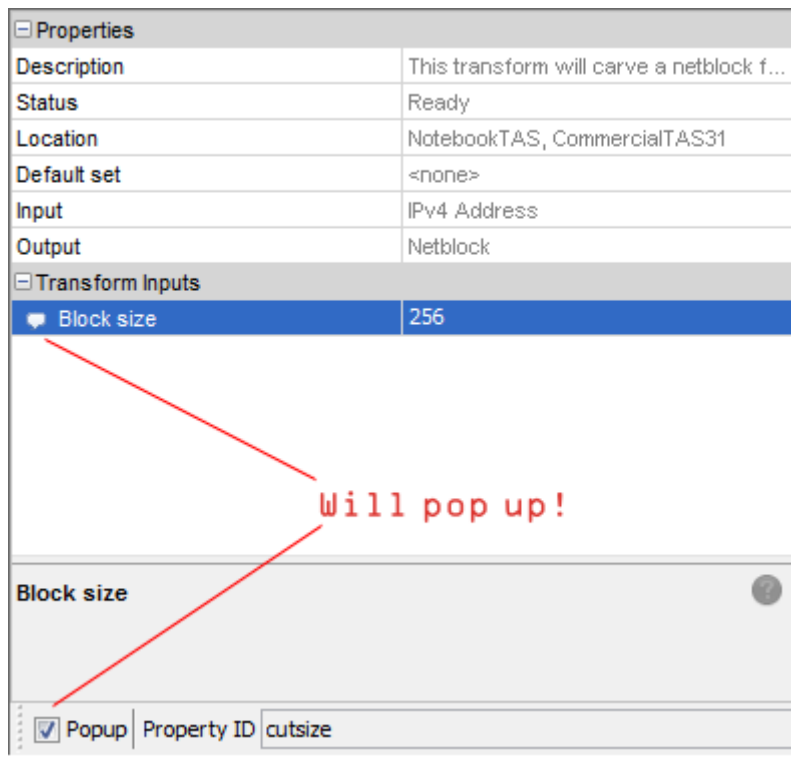
Sometimes you want the user to be able to send you information about how you should run your transform. These could be settings that could change every time the transform is executed, or could be settings that the user wants to customize and have Maltego remember. We call these transform inputs or settings.

Almost all of the standard transforms have settings – most of the time users don't really bother to change them because the defaults are working as expected. A good example of this can be seen in the reverse DNS transform where the HTTP timeout to Serversniff and Robtex can be configured:

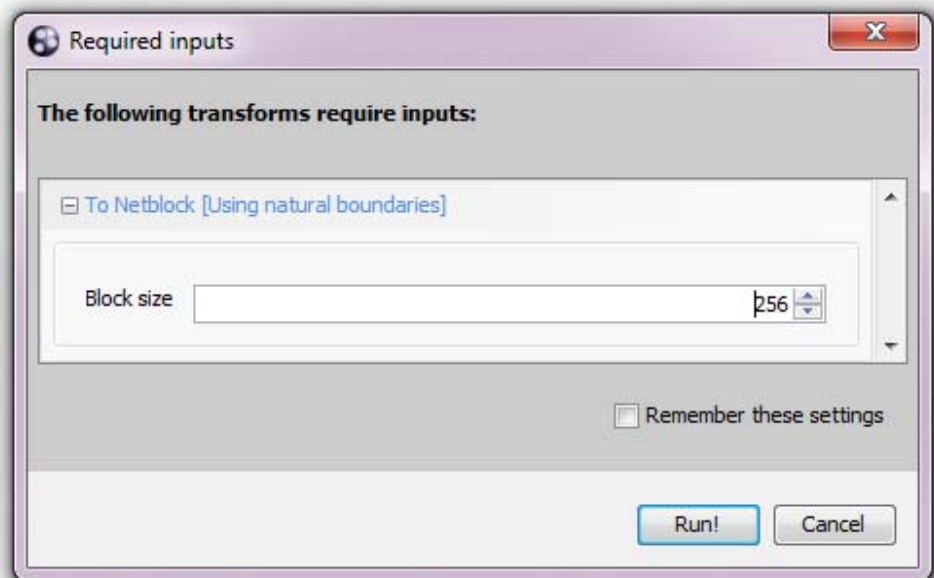


To get to the transform manager click on the 'Manage' ribbon and go to 'Manage transforms'. Any transform setting can be configured to pop-up. This is useful when the user would want to change the setting prior to the transform running. Such is the case with IP to Netblock (natural boundaries). The user might choose to select class C (256) or sub class Cs (16,32,64) or perhaps even class Bs (65535).

This transform setting looks as follows in the manager:

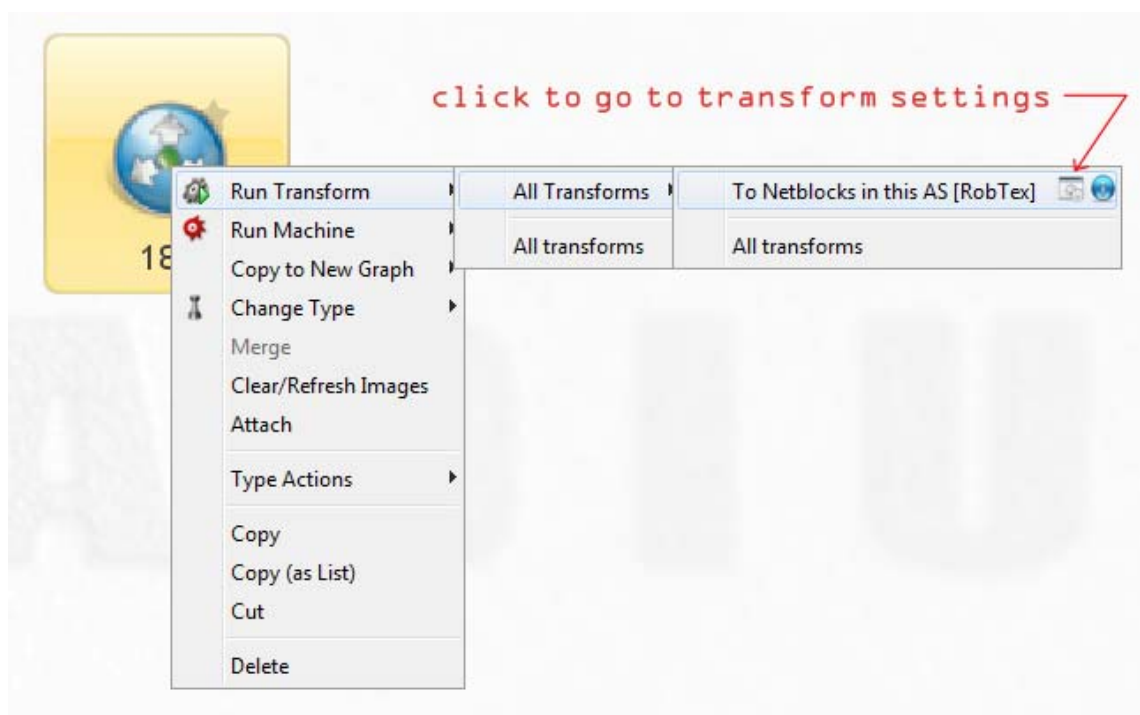


The small dialog icon next to the setting indicates that this transform will pop up a setting prior to executing the transform:



The user might choose to fix the value for subsequent transforms – by checking the ‘Remember these settings’ checkbox. In order to get the transform to pop up the setting dialog again the ‘Popup’ checkbox needs to be selected in the transform manager.

To quickly get to each transform's settings you can click on the small gear in the context menu:



Transform setting types

The Maltego GUI client supports many different pop-up types (integer, dates, lists of strings and integers) but the TDS currently only support the use of strings. This will be extended in the NTDS.

Using transform settings in TRX

Now that we know what transform settings are, let's see how to use them with custom transforms.

Because transform settings are prompted for prior to the transform running their definitions are stored in the transform itself they are loaded when the transforms are discovered. This means that, should you change transform settings, you need to rediscover from the seed so that their definitions are updated. Luckily background discovery makes this really painless:



Transform settings are defined on the TDS and can be re-used across any number of transforms. The TDS only supports string based transform settings, but this might change in the near future.

Basically a transform setting requires a

Name	:	As used in the code
Display Dialog	:	How the user is asked for it
Default value	:	Initially setting will have this value (choose carefully)
Pop-up state	:	Will it pop up?
Optional	:	Is the setting optional or mandatory?

The current specification says that a transform setting will only pop up if it's mandatory and there is no default value. This is something Paterva is aware of and will fix in the NTDS.

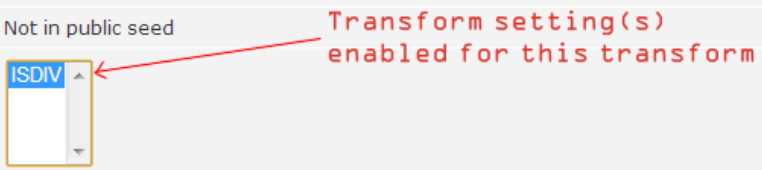
Let's go back to our initial transform (the one with the phrase as a number and AS-es as results). We want to add a transform setting that will obtain a number prior to the transform running and only return AS numbers that are divisible by that setting (if that sounds silly, it is, but again – hopefully it shows the concept). First off we need to define a transform setting on the TDS. We go to the root, click on Settings and click 'Add setting'. We fill out the form as such:

Add Transform Setting

Fill in the following form to add a new setting.

Name	<input type="text" value="ISDIV"/>
Display	<input type="text" value="Number that result should be divisible by"/>
Default Value	<input type="text" value="2"/>
Optional?	<input type="button" value="False"/> ▼
Popup?	<input type="button" value="Yes"/> ▼
<input type="button" value="Add Transform Setting"/>	

Our variable will be called **ISDIV**. Once you've added the transform setting click on 'Add Transform Setting'. Navigate to transforms, click on the EnumAS transform (which we created at the start of this document) and scroll down. At 'Transform Settings' highlight the 'ISDIV' item (you can shift click to select multiple settings):

Author	iTDS@Paterva.com
Debug	<input checked="" type="checkbox"/>
Public	Not in public seed
Transform Settings	<div>ISDIV</div> 
Seeds	<div>MyTDSSeed</div>

This tells the TDS that you wish to use the ISDIV transform setting in your transform. Transforms could have multiple settings and the same settings can be used on multiple transforms.

Next – let’s look what’s needed in the code. Our previous EnumAS ended making a graph that looked like pea soup – so we’ll start with fresh code:

```
def trx_EnumAS(m):
    TRX = MaltegoTransform()

    #read the value
    if (not m.Value.isdigit()):
        TRX.addUIMessage('Sorry but [' + m.Value + '] is not a whole
number', UIM_PARTIAL)
        return TRX.returnOutput()

    #read the setting
    isdiv = m.getTransformSetting('ISDIV')
    if (not isdiv.isdigit()):
        TRX.addUIMessage('Silly! We need a number', UIM_FATAL)
        return TRX.returnOutput()

    ###here we know we're good to go.
    howmany = int(m.Value);

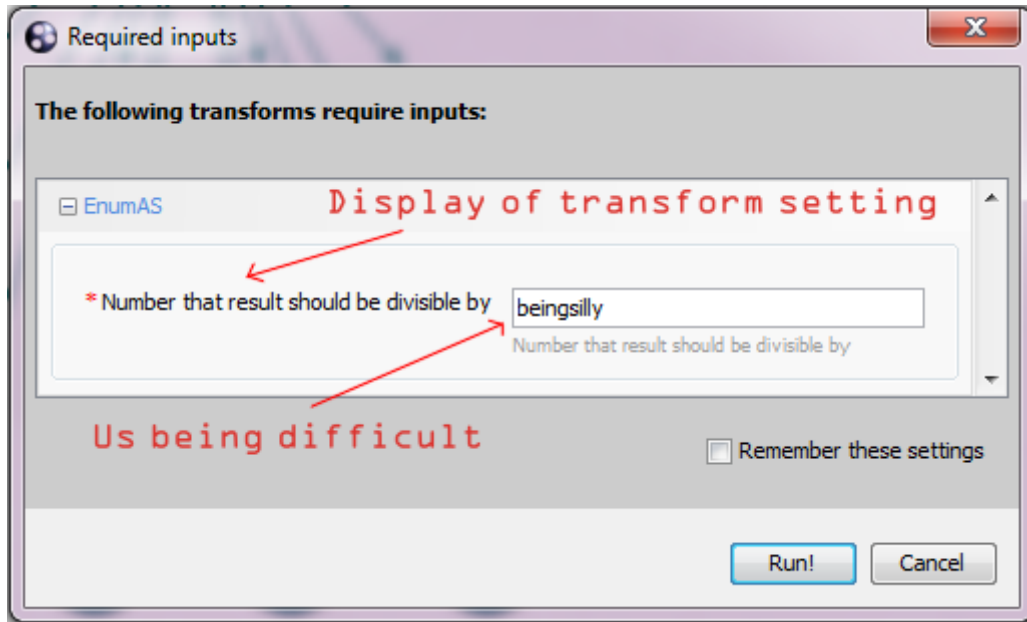
    # how many have accumulated?
    accum=0;

    for i in range(1,howmany+1):
        if (i % int(isdiv) == 0):
            Ent = TRX.addEntity('maltego.AS', str(i))
            Ent.setWeight(howmany-i)

            accum=accum+1;
            if accum>=m.Slider:
                break
```

```
return TRX.returnOutput()
```

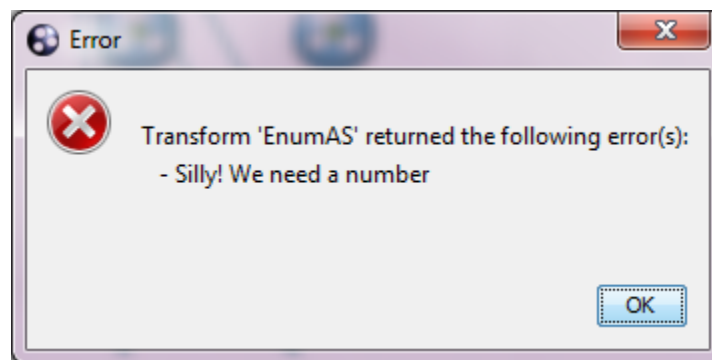
Let's see what happens when we run the transform. First up we get a pop-up asking us some questions:



In the code we do:

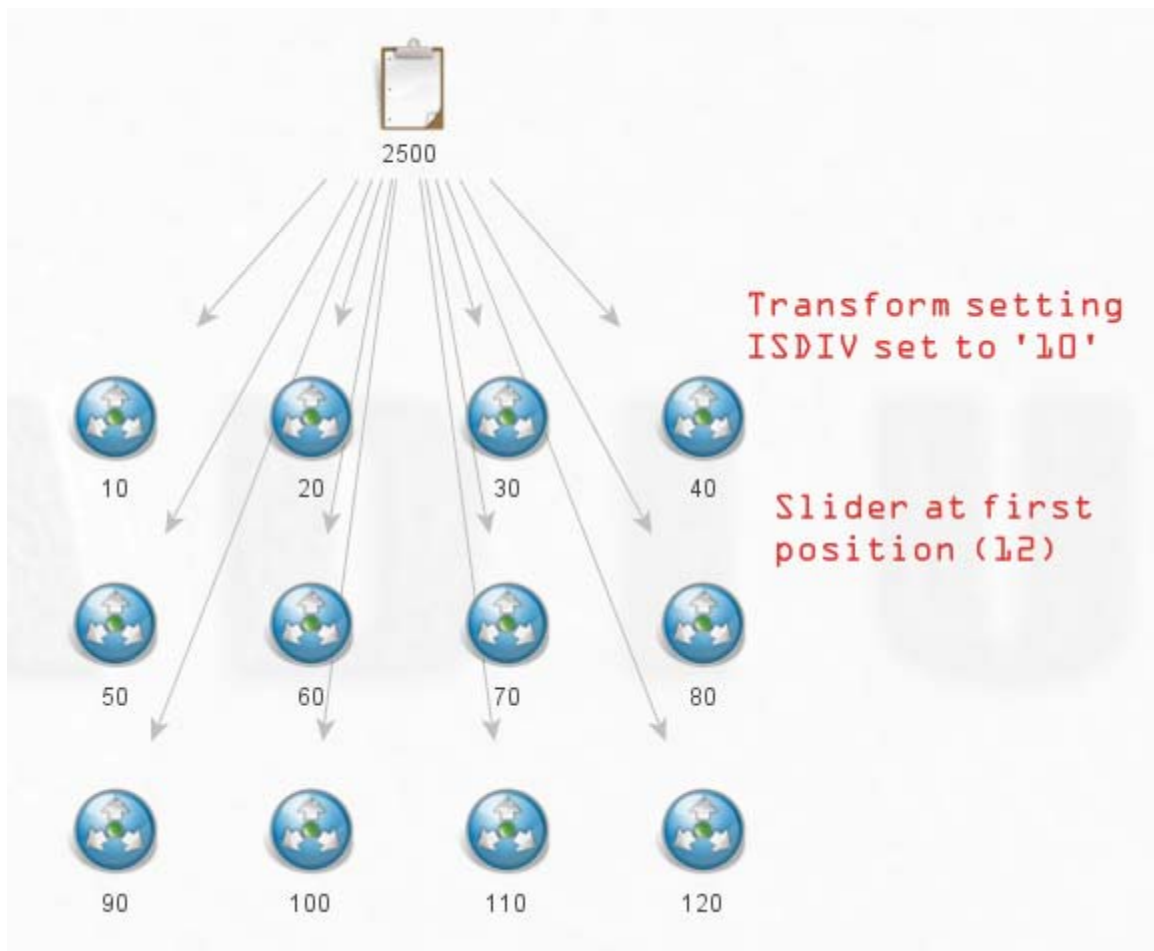
```
#read the setting
isdiv = m.getTransformSetting('ISDIV')
if (not isdiv.isdigit()):
    TRX.addUIMessage('Silly! We need a number',UIM_FATAL)
    return TRX.returnOutput()
```

This checks if the value is indeed a digit – if it's not we complain bitterly (with UIM_FATAL):



It's important to remember that the TDS only supports strings. As such we need to convert our string to an integer (after checking that it really is one!) and make sure we give the right amount of nodes back to the user.

The output of the transform is as expected:



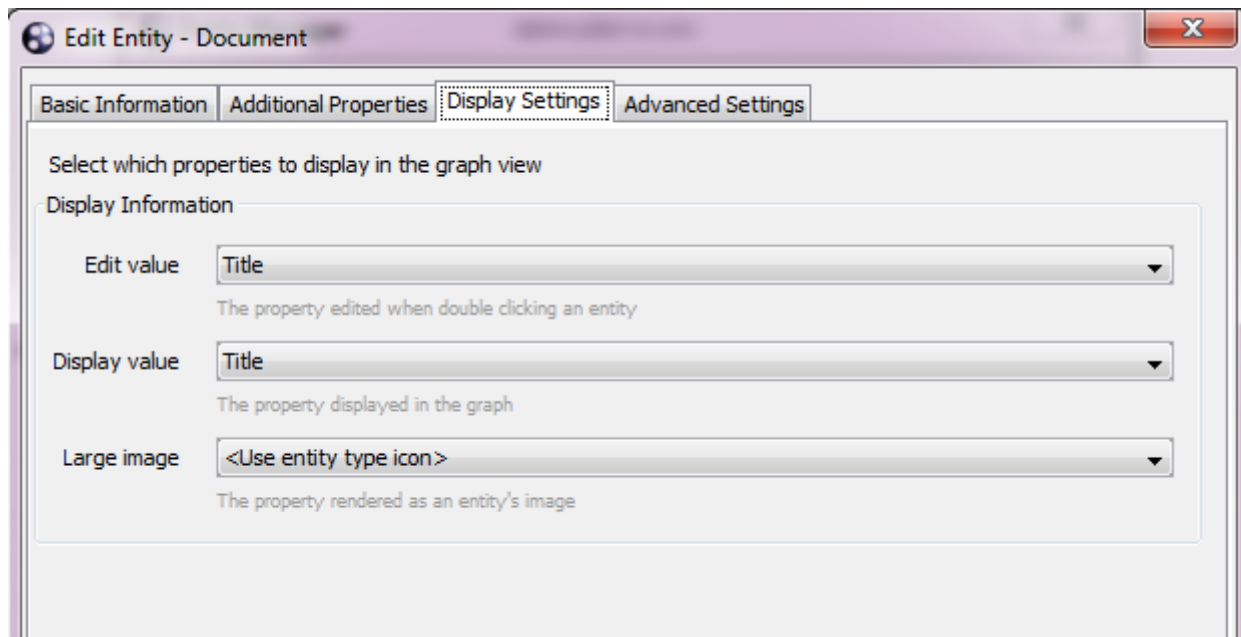
Conclusion

The combination of all of these components makes Maltego very flexible and powerful in the hands of a crafty developer. TRX makes it very easy to wield this type of power and the developer is properly shielded from any of the Maltego internal complexities.

Entity reference

V2/V3 Confusion

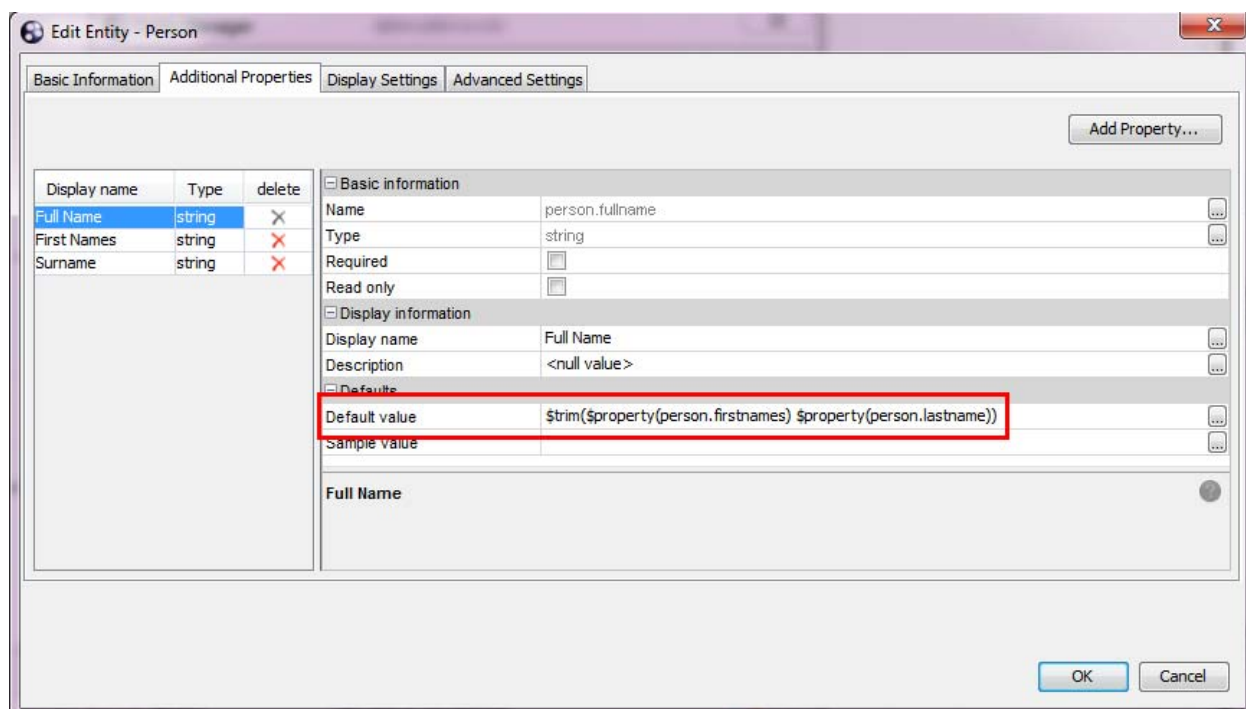
With the release of Maltego version 3 (around 2010) a new data model was developed. This was done to ensure scalability of entities. Where Maltego v2 just knew about a 'Person' we started to understand that other users would want to perhaps create their own Person entity. Thus - we moved to `maltego.Person`. The same principle was applied to properties. Where we had `'firstname'` in V2 we now understood that we should have `'person.firstname'`. Furthermore the model was built in such a way that entities could have any number of properties but that there would be a mapping between any of these properties and what's displayed underneath the entity in the graph (the display value). A good example of this is the `maltego.Document` entity - it makes more sense to display the title of the document rather than the URL where the document is located. As such the display value for a Document entity is the title. This can be easily seen on the 'Display Settings' when editing an entity within the Maltego client. Here the `maltego.Document` entity's Display Settings is shown:



In the same way the value that's edited when the user clicks on the entity and the IconURL can be mapped to any property of the entity.

Calculated properties

Another concept that was introduced in version 3 was the use of calculated properties. A person's `fullname` for instance is calculated by the concatenation of the `firstnames` and the `lastname`. This is exposed in the Maltego client:



The only entities that use calculated properties are

- maltego.Person
- maltego.Location
- maltego.PhoneNumber

Inheritance

CaseFile offered many more entities than Maltego. In CaseFile you can have a Judge, Criminal and Officer that are essentially all Persons. When importing a graph made in CaseFile into Maltego you would want to be able to run the Person transforms on all of these but the early data model did not support it.

We added the concept of inheritance – for the standard Maltego installation this meant that the MXRecord, NSRecord and Website entities were really just specialized DNSNames. The upside of it is that one transform (DNSName 2 IPAddress) worked on all of them – this saved a lot of transform configuration. For example - if you specify on the TDS that a transform will run on a DNSName it will also run on all entities down the ‘family tree’ – MXRecord, Website and NSRecord.

At the top of the tree is ‘maltego.Unknown’. This means that if you configure a transform to run on this base entity type – it will be available when you right click ... on any entity.

Why are we stuck with V2 properties?

The trouble with this new data model was that the server was still expecting the old property names for incoming values and was still supplying the old property names as results. With many different versions of Maltego in the wild changing the server to use the new properties would mean that many clients would have a non-functional Maltego client.


As such a conversion layer was implemented that mapped the old property names internally to the new names. It meant that when setting the value of a property it would internally be setting a specific property of the entity. It also meant that when the GUI supplied entities to the server it would convert the properties to the old names.


This worked fine until other developers started to code transforms. The Maltego GUI exposed the internal property names and there was no way to know what these properties translated to when it was supplied to the transforms. That is – unless you followed the specification - that was last updated in 2009.


As such we provide here both the internal (V3) property names as well as their V2 equivalents. When coding transforms it is best to stick with the V2 equivalents. It would be trivial to change the server (CTAS) to use the new properties but it will almost certainly result in some clients stuck with a non-functional GUI. It seems that these legacy properties are here to stay.

When working with your own custom entities this legacy problem is a non-issue of course. It only comes into play when you want to leverage the transforms that are located on the CTAS – then you need to make sure you are mapping to the right V2 equivalent properties.


Entity reference guide


maltego. Domain	Inherits from: maltego.Unknown			 paterva.com
Property Name	Type	Display name	V2 equivalent	
fqdn	string	Domain name	Value	
whois-info	string	WHOIS info	whois	


maltego. DNSName	Inherits from: maltego.Unknown			 alpine.paterva.com
Property Name	Type	Display name	V2 equivalent	
fqdn	string	DNS Name	Value	


maltego. MXRecord	Inherits from: Maltego.DNSName			 mail.paterva.com
Property Name	Type	Display name	V2 equivalent	
fqdn	string	MX Record	Value	


mxrecord.priority	integer	Priority	mxrecord.priority
-------------------	---------	----------	-------------------

maltego. NSRecord	Inherits from: Maltego.DNSName	 ns1.linode.com	
Property Name	Type	Display name	V2 equivalent
fqdn	string	MX Record	Value


maltego. IPv4Address	Inherits from: maltego.Unknown	 74.207.243.85	
Property Name	Type	Display name	V2 equivalent
ipv4-address	string	IP Address	Value
ipaddress.internal	boolean	Internal	ipaddress.internal


maltego. Netblock	Inherits from: maltego.Unknown	 74.207.243.0-74.207.243.255	
Property Name	Type	Display name	V2 equivalent
ipv4-range	string	IP Range	Value


maltego. AS	Inherits from: maltego.Unknown	 188	
Property Name	Type	Display name	V2 equivalent
as.number	integer	AS Number	Value


maltego. Website	Inherits from: maltego.DNSName	 www.paterva.com	
Property Name	Type	Display name	V2 equivalent
fqdn	string	Website	Value


website.ssl-enabled	boolean	SSL Enabled	website.ssl-enabled
ports	int []	Ports	ports


maltego. URL	Inherits from: maltego.Unknown		 URL Title
Property Name	Type	Display name	V2 equivalent
short-title	string	Short Title	Value
url	URL	Value	theurl
title	string	Title	fulltitle


maltego. Phrase	Inherits from: maltego.Unknown		 Some phrase
Property Name	Type	Display name	V2 equivalent
text	string	Text	Value


maltego. Document	Inherits from: maltego.Unknown		 Some Document
Property Name	Type	Display name	V2 equivalent
url	string	URL	link
Title	String	Title	Value
document.meta-data	String	Meta-Data	metainfo


maltego. Person	Inherits from: maltego.Unknown		 John Doe
Property Name	Type	Display name	V2 equivalent
person.fullname ⁺	string	Full Name	Value
person.firstnames ⁺	string	First Names	firstname
person.lastname ⁺	string	Surname	lastname


maltego. EMailAddress	Inherits from: maltego.Unknown	 info@paterva.com	
Property Name	Type	Display name	V2 equivalent
email	string	Email Address	Value


maltego. Location	Inherits from: maltego.Unknown	 Pretoria, South Africa	
Property Name	Type	Display name	V2 equivalent
location.name*	string	Name	Value
country*	string	Country	country
city*	string	City	city
location.area	string	Area	area
countrycode	string	Country Code	countrysc
longitude	float	Longitude	long
latitude	float	Latitude	lat

maltego. PhoneNumber	Inherits from: maltego.Unknown	 27 12 667 0000	
Property Name	Type	Display name	V2 equivalent
Phonenumber*	string	Phone Number	Value
phonenumber.countrycode*	string	Country Code	countrycode
phonenumber.citycode*	string	City Code	citycode
phonenumber.areacode*	string	Area Code	areacode
phonenumber.lastnumbers*	string	Last Digits	lastnumbers

maltego. Alias	Inherits from: maltego.Unknown	 Mr. T	
Property Name	Type	Display name	V2 equivalent
alias	string	Alias	Value

maltego. Image	Inherits from: maltego.Unknown			 Image
Property Name	Type	Display name	V2 equivalent	
description	string	Description	Value	
url	URL	URL	url	

maltego. Twit	Inherits from: maltego.Unknown			 Over 10,000 Maltego Community Edition users!
Property Name	Type	Display name	V2 equivalent	
twit.name	string	Twit	value	
id	string	Twit ID	id	
author	string	Author	author	
author_uri	string	Author URI	author_uri	
content	string	Content	content	
img_link	string	Image Link	img_link	
pubdate	string	Date published	pubdate	
title	string	Title	title	

maltego.affiliation. Twitter	Inherits from: maltego.Unknown			 Paterva
Property Name	Type	Display name	V2 equivalent	
person.name	string	Name	value	
affiliation.network	string	Network	network	
affiliation.uid	string	UID	uid	
affiliation.profile-url	string	Profile URL	affiliation.profile-url	
twitter.number	int	Twitter Number	twitter.number	
twitter.screen-name	string	Screen Name	twitter.screen-name	
twitter.friendcount	int	Friend Count	twitter.friendcount	
person.fullname	string	Real Name	person.fullname	

* Calculated properties

+ Used to calculate property

The TRX Maltego.py API

MaltegoMsg class

This is used to read the Maltego request. It is passed along to each transform.

Member name	Description
Value (String)	The value of the node as displayed on the graph. Note that this is not necessarily the value you want work with (see URL entity).
Weight (Integer)	The weight of the node.
Slider (Integer)	The slider's value – e.g. how many results should be returned.
Type (String)	The type of the input node. See entity definitions for possible values.
Properties (List)	A list of properties of the node. Name, value pairs as strings
TransformSettings (List)	The settings for the transform. A list of name, value pairs as strings.

The following methods are defined to read entity properties and transform settings. You can read it straight out of the `Property` and `TransformSettings` list, but it's nicer to use these functions as they do some error checking:

Method name	Description	Returns
<i>getProperty</i> (String key)	Returns the value of the key, None if not defined	Value
<i>getTransformSetting</i> (String key)	Returns the value of the key, None if not defined	Value

MaltegoTransform class

This is used to construct the reply to the TDS. All values are strings.

Method name	Description	Returns
<i>addEntity</i> ([String Type , String Value])	Adds an entity to the return vessel with type 'Type' and value 'Value'. Note that these can be set using functions in MaltegoEntity library.	MaltegoEntity
<i>addUIMessage</i> (String msg , Const type)	Shows a message 'msg' in the Maltego GUI. Types could be <ul style="list-style-type: none">• UIM_FATAL (pop up window)• UIM_PARTIAL (yellow)• UIM_INFORM (default)• UIM_DEBUG (light gray)	none
<i>addException</i> (String msg)	Throws a transform exception	none

returnOutput()	Returns the XML of the vessel	none
-----------------------	-------------------------------	------

MaltegoEntity class

This is the object that defines a single entity within Maltego. An entity can be created using the 'addEntity' method in the MaltegoTransform class.

Method name	Description
setType (String Type)	Sets the type of the entity – see list of Entity definitions for possible values
setValue (String Value)	Sets the value of the entity
setWeight (Integer weight)	Sets the weight of the entity
addDisplayInformation (String Value, [String Label])	Adds display information for label named 'Label'. This field is rendered as HTML within Maltego. Default label is 'Info'.
addProperty (String propertyname, String displayname, String matchingrule String value)	Adds a property to the entity. Each property has a name, value and a display name. The display name is how it will be represented within Maltego. The matching rule determines how entities will be matched and could be 'strict' (default) or 'loose'
setIconURL (String URL)	If set it will change the appearance of the icon. The URL should point to a PNG or JPG file. Maltego will size to fit but lots of large files will drain resources.
setLinkColor (String HexColor)	Sets the color of the link to the node. Colors are in hex –for example '0xff00ff'
setLinkStyle (Const Style)	Sets the style of the link to the node. The following constants should be used: <ul style="list-style-type: none"> • LINK_STYLE_NORMAL • LINK_STYLE_DASHED • LINK_STYLE_DOTTED • LINK_STYLE_DASHDOT
setLinkThickness (Integer thickness)	Sets the thickness of the link to the node. Value is in pixels.
setLinkLabel (String value)	Sets the label of the link to the node.
setBookmark (Const Color)	Sets the bookmark color of the node. Keep in mind that these are chosen from a set number of colors. Use the following constants: <ul style="list-style-type: none"> • BOOKMARK_COLOR_NONE • BOOKMARK_COLOR_BLUE • BOOKMARK_COLOR_GREEN • BOOKMARK_COLOR_YELLOW • BOOKMARK_COLOR_ORANGE • BOOKMARK_COLOR_RED
setNote (String note)	Creates an annotation to the node with value 'note'. If a subsequent transform sets an annotation on the node it will appended to the note.

