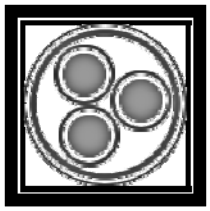
11/25/2012

MALTEGO SCRIPTING LANGUAGE (1.1)



Contents

Introduction 4
Concepts 4
Pipelines 4
The basics 4
Parallel paths 6
Serial paths 7
Transform settings, slider values 8
Filters 9
User filters 9
User filter options 10
Pipeline filter 11
Making filters global 14
Deleting nodes 14
Debug, logging and status messages 15
Perpetual machines 16
Complete reference guide 18
Service Pack 2 18
Actions 19
bookmark 19
clearBookmark 19
delete 19
deleteBranch 20
exportImage 20
log 20
run 21
saveAs 22
setLayout 22
status 23

userFilter 23
Filters 25
age 25
bookmarked 25
degree 26
incoming 26
outgoing 27
property 27
type 28
value 28

Introduction

The Maltego scripting language allows you to automate the running of transform in Maltego. This document tries to explain the language. No faffing around.

Concepts

- Pipeline – a set of transforms and filters that are executed in sequence. Think a macro.
- Trigger – a graph condition and a transform. Think when this happens on the graph, run this transform.
- Feeder – a mechanism to feed entities into Maltego.
- Machine – a combination of pipelines, triggers and feeders.

At the time of writing feeders and triggers are not implemented yet. As such we'll only look at pipelines in this document.

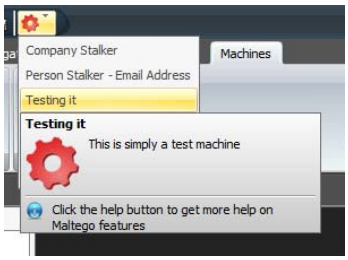
Pipelines

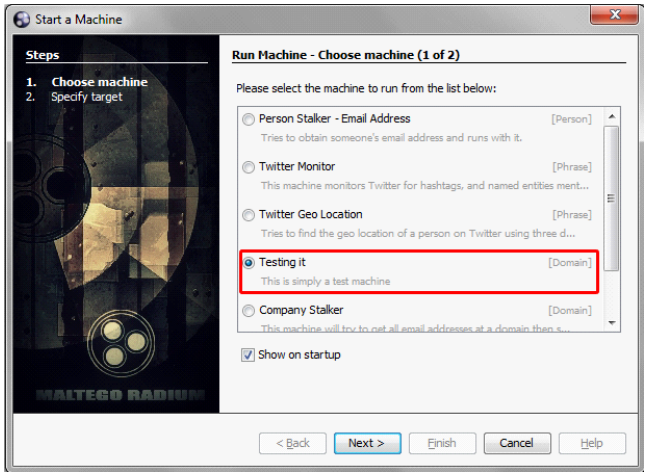
The basics

A pipeline is a sequence of filters and transforms. Looks at the following example:

```
//simple machine
machine("maltego.test",
    displayName:"Testing it",
    author:"Roelof Temmingh",
    description: "This is simply a test machine") {

    start{
       run("paterva.v2.DomainToMXrecord_DNS")
       run("paterva.v2.DNSNameToIPAddress_DNS")
       run("paterva.v2.IPAddressToNetblock_Cuts")
    }
}
```

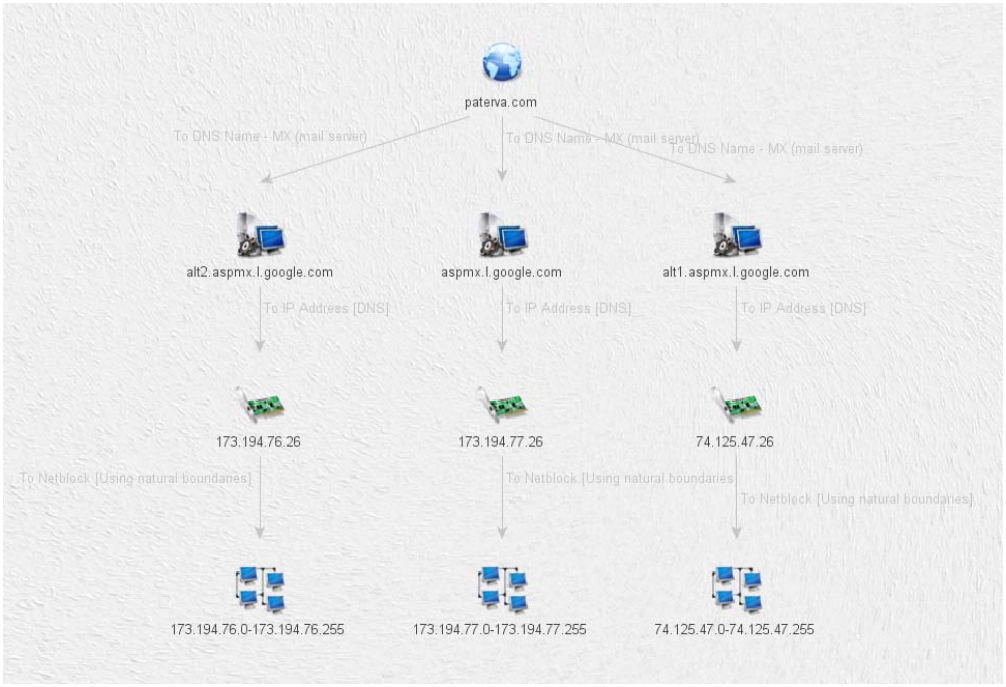
From the example we can quickly see that '//' is used to comment code. You will also see that each machine has a unique name, display name, author and a description. These are used to describe the machine and translate it showing up as follows:



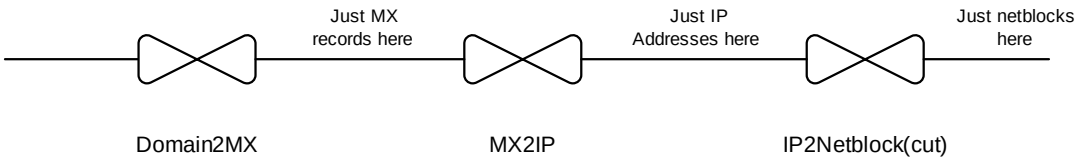


In the example there are three transforms that are executed – top to bottom. The first one takes a domain name as input and generates an MX record. At that stage in the pipeline we have an MX record, so the next transform takes the MX record to an IP address which in turn is transformed into a netblock.

The resultant graph looks like this (we used 'paterva.com' as input):



The pipeline looks like this (always left to right):



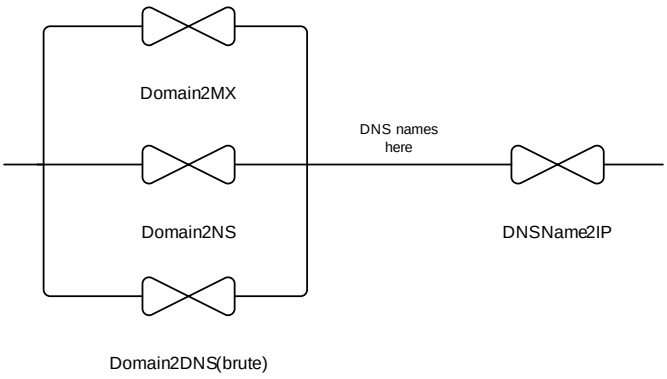
It's important to know that the pipeline does not accumulate entity types. It ONLY contains what the previous step has generated. This is also true for filters which we'll get to later.

Parallel paths

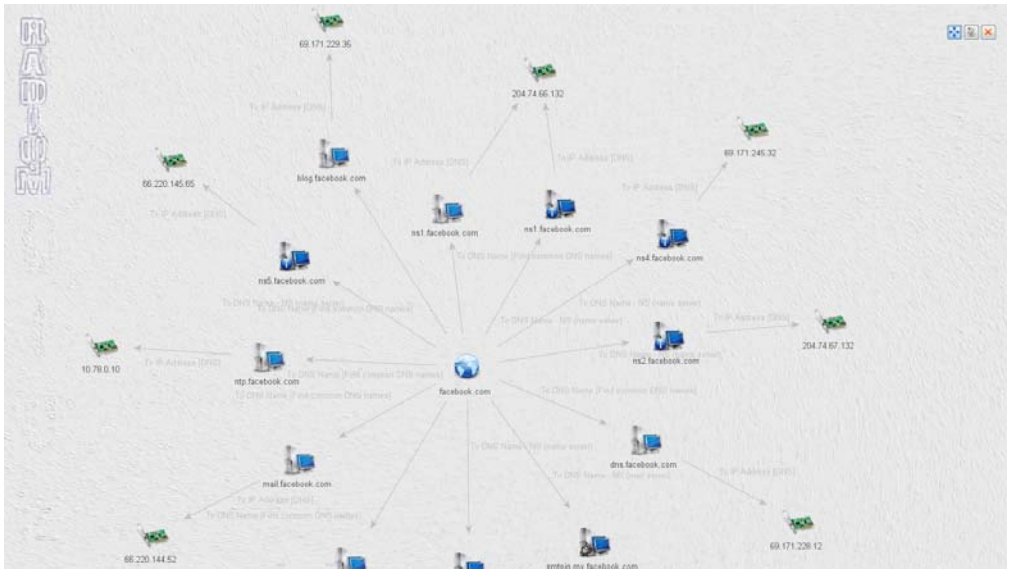
To run transforms in parallel we'll use the 'paths'. Again, let's look at an example:

```
start{
   paths{
         run("paterva.v2.DomainToMXrecord_DNS")
         run("paterva.v2.DomainToNSrecord_DNS")
         run("paterva.v2.DomainToDNSName_DNSBrute")
   }

   //now resolve these to IP addresses
    run("paterva.v2.DNSNameToIPAddress_DNS")
}
```

The pipeline for this looks as follows:



The resultant graph (running on 'facebook.com') looks like this:



Note that there are MX, NS and DNS Name records on the graph.

It's important to note that because the MX record entity and the NS record entity inherit from the DNSName entity all of them can be resolved to IP with one transform.

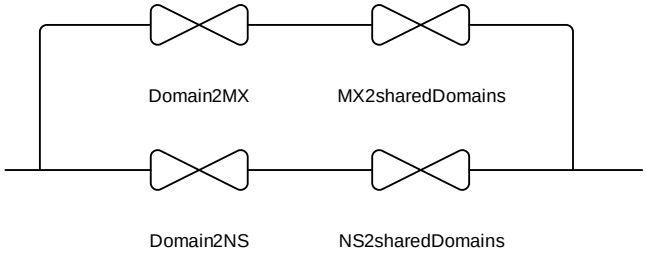
Whenever you want to split a pipe into two or more sections the syntax is as follows:

```
paths{
	Parallel-1
	Parallel-2
	..
	Parallel-N
}
```

Commands inside the curly brackets will be executed in parallel.

Serial paths

This is all fine but what when we want to do something like this:



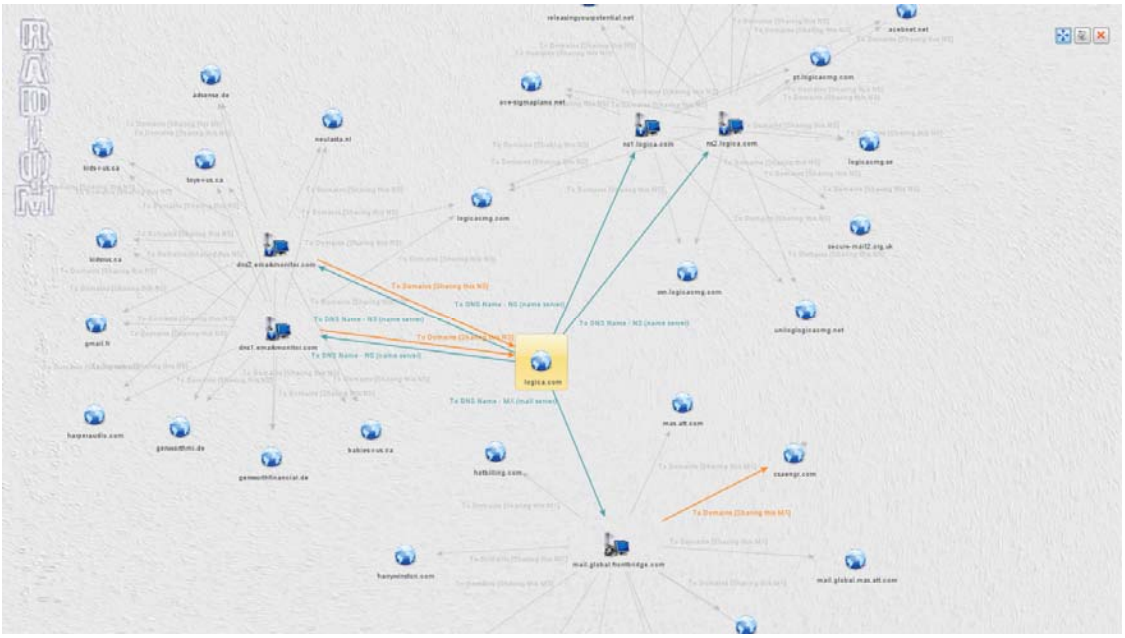
The problem here that there is a sequence of transforms that's within their own pipeline but that has to be executed in parallel. In Maltego scripting language this can be achieved as follows:

```
start{
	paths {
```

```
                path {
                    run("paterva.v2.DomainToMXrecord_DNS")
                    run("paterva.v2.MXrecordToDomain_SharedMX")
                }
                path {
                    run("paterva.v2.DomainToNSrecord_DNS")
                    run("paterva.v2.NSrecordToDomain_SharedNS")
                }
        }
```

From this example you can clearly see that each 'sub pipeline' is in a 'path' clause, but that these run in parallel.

The resultant graph when run on 'logica.com' looks like this:



From this graph is becomes apparent that some form of user filtering is needed before we proceed with next steps. In this example the NS records ns1.logica.com and ns2.logica.com yields good info, but Logica also uses MarkMonitor as a NS and looking at shared domains there is pretty useless. We cover filters in subsequent sections – keep reading!

Transform settings, slider values

Transforms can optionally be passed extra information. Slider value (how many results to return) as well as transform settings can be specified:

```
run("transformname",slider:N)
//will run transform and <= N number of results will be returned
```

Let's look an example:

```
run("paterva.v2.MXrecordToDomain_SharedMX",slider:255)
```

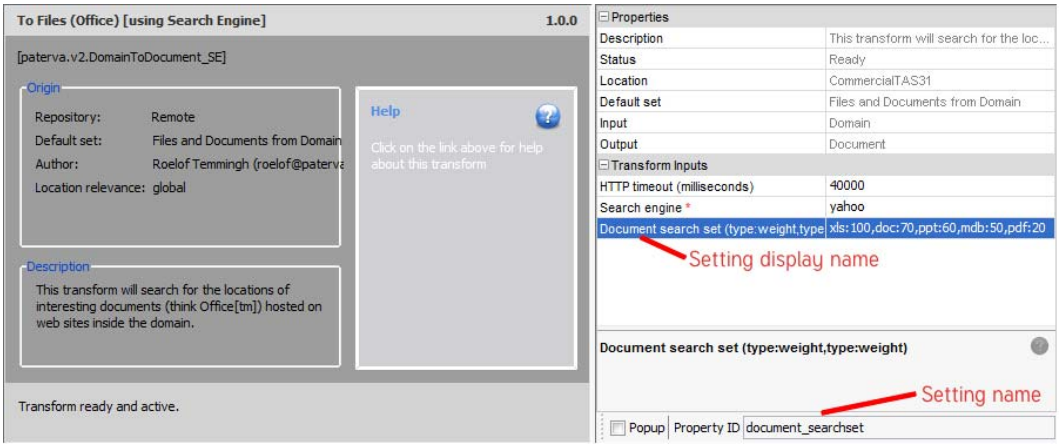
Transform settings can we specified like so:

```
run("transform", "setting_name":"value")
//will run transform with certain transform settings
```

As an example:

```
run("paterva.v2.DomainToDocument_SE","engine":"google",slider:50)
```

The setting's actual name (e.g. not the display name) can be found in the transform manager:



Filters

There are two types of filters – user filters and pipeline filters. A user filter will give the user the chance to interact with the information in the pipe, effectively stopping the flow until the user has decided what to keep in the pipe and what to delete. A pipeline filter (or just filter) will filter the information based on certain parameters and requires no user interaction.

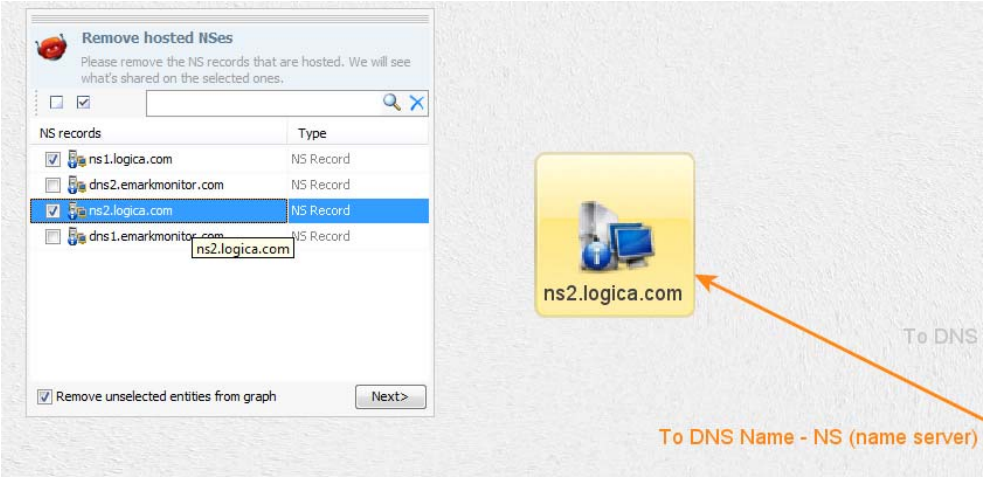
User filters

To stop the flow and send the output to the user for manual filtering, simple insert the command 'userFilter()'. Let's use it in an example. We'll use the same pipeline as before, but this time give the user the option of editing the MX and NS records before passing them along to the sharedMX/NS transforms. The script thus looks like this:

```
start{
    paths {
        path {
            run("paterva.v2.DomainToMXrecord_DNS")
            userFilter()
            run("paterva.v2.MXrecordToDomain_SharedMX")
        }
        path {
            run("paterva.v2.DomainToNSrecord_DNS")
            userFilter()
            run("paterva.v2.NSrecordToDomain_SharedNS")
        }
    }
```

```
}
```

When running this inside of Maltego it looks as follows:



In this case we're telling Maltego that we should not pass the MarkMonitor entities onto the next transform. Once this user filter has been completed (the user clicked on Continue), the pipeline flows again.

User filter options

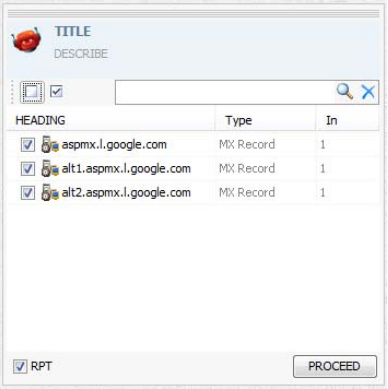
There are many ways to customize the user filter. The filter in the screenshot above was created as follows:

```
userFilter(title:"Remove hosted NSes",heading:"NS records",description:"Please
remove the NS records that are hosted. We will see what's shared on the
selected ones.",proceedButtonText:"Next>")
```

The following options are available:

- title (string) – the title of the dialog
- heading (string) – the heading of the first column
- proceedButtonText (string) – the text on the proceed button
- icon (string) – the name of an icon to display
- removePromptText (string) – the text on the "remove unselected entities" checkbox
- removePromptChecked (boolean) – the default value of said checkbox
- showIncomingLinks (boolean) – display the incoming links column
- showOutgoingLinks (boolean) – display the outgoing links column
- selectEntities (boolean) – default selection state of entities

Consider the following screen shot:



The user filter for above is as follows:

```
userFilter(title:"TITLE",heading:"HEADING",description:"DESCRIBE",proceedButton
Text:"PROCEED",removePromptText:"RPT",removePromptChecked:true,showIncomingLink
s:true)
```

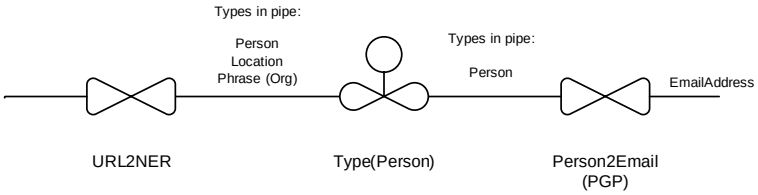
Remember that strings should be enclosed in quotes (") while Booleans are either true or false and are not enclosed in quotes.

Pipeline filter

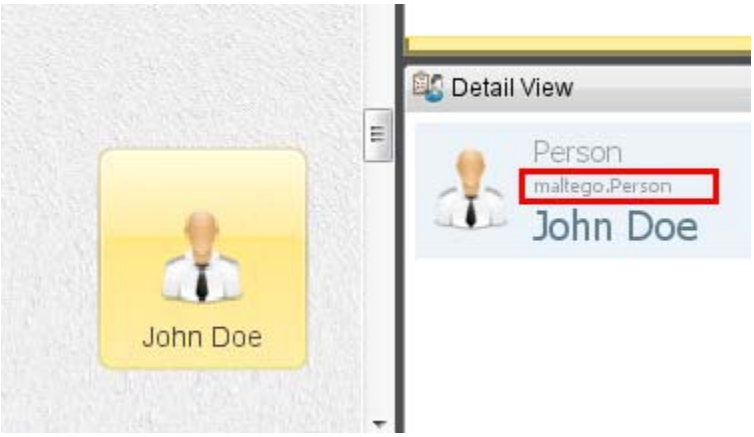
Pipeline filters stack next to each other. Each line in a filter makes the filter more specific. Consider the following (slightly useless) script:

```
start{
    run("paterva.v2.URLToPerson_NLP")

    //filter just person entities
    type("maltego.Person")
    run("paterva.v2.PersonToEmailAddress_SamePGP")
}
```

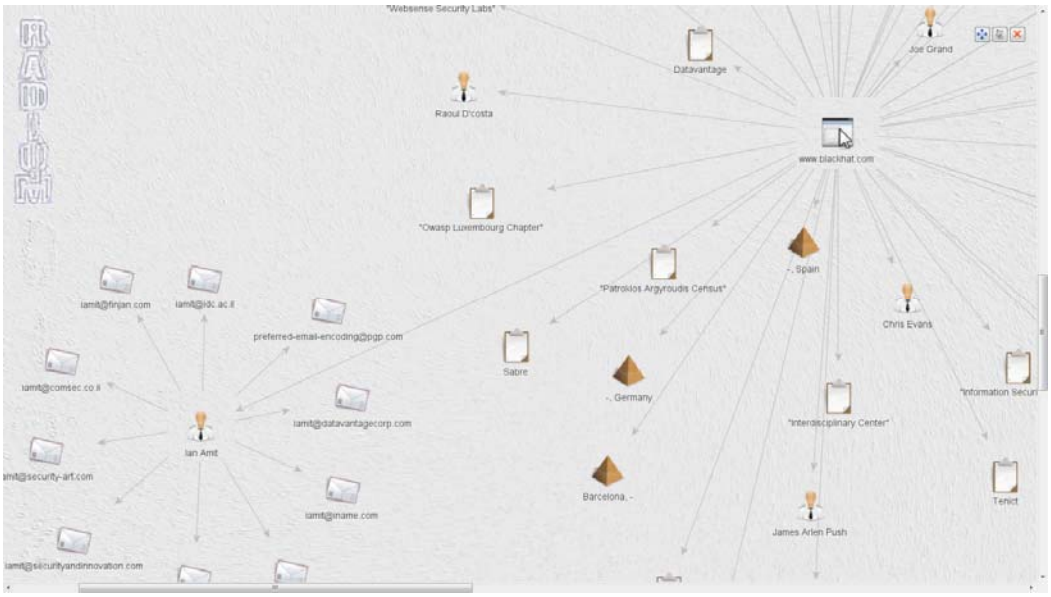
The pipeline for this looks as follows:



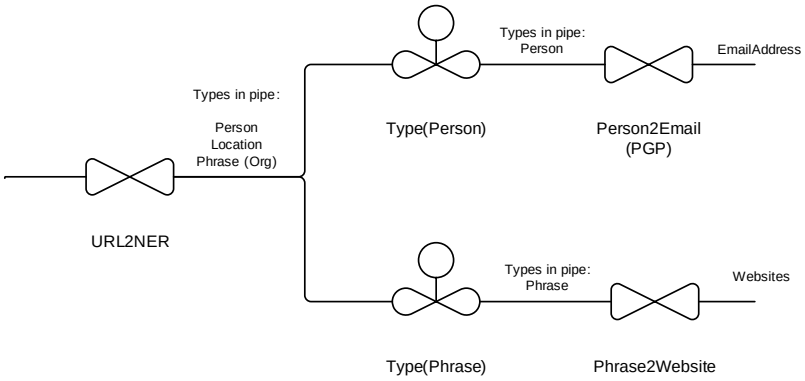
Note that the NER transform will return three types of entities – Person, Location and Phrase (which is the organization / company). To get the proper name for the entity you can look in the detail view:



When we feed this transform the BlackHat Europe 2010 speaker lineup (http://www.blackhat.com/html/bh-eu-10/bh-eu-10-speakerbios.html) the result is as follows:



If we needed to do something with the other types in the pipe (e.g. the phrases) the pipeline should look like this:



The code for this looks as follows:

```
start{
    run("paterva.v2.URLToPerson_NLP")

    paths{
        path{
            //filter just person names
            type("maltego.Person")
            run("paterva.v2.PersonToEmailAddress_SamePGP")
        }
        path {
            //filter on phrase
            type("maltego.Phrase")
            run("paterva.v2.PhraseToWebsite_SE")
        }
    }
}
```

The following can be used to filter in pipelines:

- Type. *Use **type()***
- Number of incoming links. *Use **incoming()***
- Number of outgoing links. *Use **outgoing()***
- Value of the entity. *Use **value()***
- Age of the entity – when used with perpetual machines. *Use **age()***
- Property of an entity. *Use **property(propertyname,value)***

When working with strings – for instance in value or property value filter the following applies

```
value("frikkie")
//only matches "frikkie"

value(like: "frikkie", ignoreCase:true)
//matches "frikkiesbeer", "Its me Frikkie" and "befrikkied"

property("ipaddress.internal", equalTo:true)
//matches all internal IPs

property("ipaddress.internal", like: "true", ignoreCase:true)
```

```
//matches all internal IPs but uses a like query
```

When working with numbers – for incoming, outgoing, age etc. the following applies:

```
outgoing(2)
//matches exactly 2 outgoing links

incoming(moreThan:0)
//matches if the entity has any incoming links

outgoing(lessThan:5, moreThan2)
//matches if the entity 3 or 4 links (in or out)

age(moreThan:400)
//matches nodes older than 400 seconds
```

Making filters global

In many cases you want to collect nodes from all over the graph and not just from the current location in the in the pipeline. To do this you can add the argument 'scope:"global"' to any filter. Consider the following script:

```
//prune leaf nodes
    start{
       incoming(1,scope:"global")
       outgoing(0)
       delete()
    }
```

When you run this machine on any node this script will simply prune leaf nodes on a graph. Leaf nodes have no outgoing links and just one incoming link – which is precisely what the filter combination does. Another way of writing the filter would be:

```
outgoing(0,scope:"global")
incoming(1)
delete()
```

Here we are simply reversing the order of the filters but it ends up to be the same thing.

Keep in mind that adding the global scope to your filter will access the ENTIRE graph everything. Therefore the second call will basically remove the first filter - this would be wrong…

```
outgoing(0,scope:"global")
incoming(1,scope:"global")
delete()
```

…as the second filter line also contains the global scope and negates the first filter.

Deleting nodes

You can instruct Maltego to delete nodes. This is very useful when using perpetual machines (else the graph grows out of control over time). In certain scenarios you would want to be able to delete not just

the node but perhaps the parents (and/or) children of the node too. The delete command will ALWAYS delete the node it's running on. The command to delete nodes is … delete().

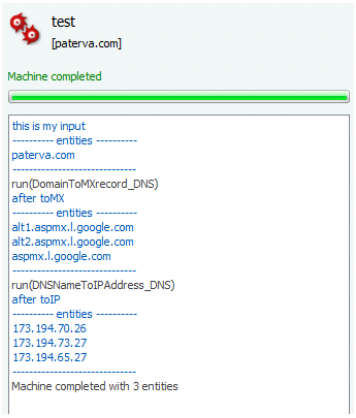
```
delete()
//deletes the current nodes in the pipeline

delete(parents:2)
//delete myself plus all my parents and grandparents –e.g. two levels up

delete(children:2)
//deletes myself plus all children and grandchildren – e.g. two levels down
```

Debug, logging and status messages

To help debugging machines you can use the 'log' command. Whenever you call this command the Maltego GUI will tell you what's in the pipe at that stage. Consider the following script:

```
start {
    log("this is my input",showEntities:true)
    run("paterva.v2.DomainToMXrecord_DNS")
    log("after toMX",showEntities:true)
    run("paterva.v2.DNSNameToIPAddress_DNS")
    log("after toIP",showEntities:true)
  }
```

Inside of Maltego you'll see the following in the output screen:



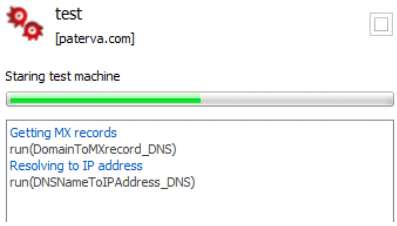
The debug command automatically shows which entities are in the pipeline at the stage when the debug command was called.

When called with `showEntities:false` the command acts as a scrolling status update in the text window.

To set the status at the top you may use the `status` command. This sets the top label on the machine window. Keep in mind that when setting the status label in a parallel path only one of the label will be

visible – as the commands are executed in parallel and whatever last label is set will stick. Here is an example of the status command and the corresponding output:

```
        start {
          status("Staring test machine")
          log("Getting MX records",showEntities:false)
          run("paterva.v2.DomainToMXrecord_DNS")
          log("Resolving to IP address",showEntities:false)
          run("paterva.v2.DNSNameToIPAddress_DNS")
        }
}
```



Perpetual machines

It is possible to have machines that run until terminated. Think of this more as a monitoring machine. Use onTimer(X) {} instead of start{} in your script. This will start the machine every X seconds. Consider the following script:

```
machine("maltego.twitter.monitor",
    displayName:"Twitter Monitor",
    author:"Roelof Temmingh",
    description: "This machine monitors Twitter for hashtags, and named
entities mentioned around a certain phrase") {

    //run every minute and a half
    onTimer(90) {
        run("paterva.v2.PhraseToTwit_Search",slider:30)
        paths{
            run("paterva.v2.pullHashTags")
            run("paterva.v2.toEntitiesNERTwitter")
            run("paterva.v2.pullURLs")
        }

        //delete Tweets with no info, after one round
        age(moreThan:100, scope:"global")
        type("maltego.Twit")
        outgoing(0)
        delete()

        //delete Tweets as they get older than 5 minutes
        age(moreThan:300, scope:"global")
        type("maltego.Twit")
        delete()

        //after a while, when nothing links to it, remove the orphans
        age(moreThan:500, scope:"global")
        incoming(0)
        outgoing(0)
        delete()

    }
}
```

As you can see from the machine – running the transforms are fairly straightforward. The more interesting code is deciding when and which nodes to delete to ensure that the graph is always up to date.

When a machine is still running it will not start another instance of the same machine.

Complete reference guide

The following document is the reference guide for MSL – this covers everything up to Maltego Radium Service Pack (SP) 2.

Service Pack 2

Service pack 2 has the following additional scripting features:

- **Bookmarking**. Set and clear bookmarks within the machine. Can also be used in filters.
- **Export image.** Saving the graph as an image – useful for perpetual machines
- **SaveAs.** Saving the graph as an MTGX – see above.
- '**ignoreSeenEntities**' option on running transforms. This is useful in perpetual machines when you want the transform to always run on entities even if they are not new.
- **Setting the layout**. Gives you the ability to set the layout within a machine.
- **'Invert'** option on filters. Gives you the ability to negate filters.

Actions

Actions are tasks that can be performed. Actions might apply to entities in the current pipe or to the graph as a whole.

bookmark

Bookmarks an entity

Example

`bookmark()`

Bookmarks an entity with the default bookmark colour (cyan)

Optional parameters

`<value>`	Integer value of the bookmark index/colour to tag the entity with
`overwrite`	Indicates whether to overwrite the bookmark if the entity is already bookmarked. Defaults to true.

More examples

`bookmark(4, overwrite:false)`

Tag the entity with a red bookmark but only if it does not have any kind of bookmark yet

clearBookmark

Removes any bookmark an entity might have

Example

`clearBookmark()`

Removes the entity's bookmark

delete

Deletes entities

Example

`delete()`

Deletes the selected entities from the graph

deleteBranch

Deletes entities and their children unless their children have a parent that is not part of the deleted branch

Example

```
deleteBranch()
```

Deletes all branches of selected entities

exportImage

Exports the current graph as an image

Example

```
exportImage("C:\\Temp\\myimage.png")
```

Exports the graph as a PNG file to the given location

Required parameters

`<value>`	The path of the image to create. The image format is indicated by the file extension. Backslashes need to be escaped with double backslashes (\\)

Optional parameters

`path`	Same as <value>
`suffixDate`	Boolean indicating whether date and time should be appended. Default is true.
`dateFormat`	Specifies the format in which the date and time should be appended. Default is yyyy`MMdd-HHmmss`SSS.

More examples

```
exportImage("C:\\Temp\\myimage.png", suffixDate:false)
```

Exports the graph as a PNG file without any date/time suffix

```
exportImage("C:\\Temp\\myimage.jpg", dateFormat:"HHmmss")
```

Exports the graph as a JPEG file and appends the time of day

log

Writes a log message to the Machine Run Window

Example

```
log("this is a log message")
```

Prints the given text in the machine run window

Required parameters

`<value>`	The message to print

Optional parameters

`status`	Update the status field with the given text
`showEntities`	Print the entities in the current pipe. Useful for debugging. Default is false.

More examples

```
log("Resolving IP addresses", status:"IP address resolution phase...")
```

Prints a log message and updates the status message in the Machine Run Window

```
log("DNS names are:", showEntities:true)
```

Prints a log message and lists the entities that are currently in the pipe

run

Runs a transform

Example

```
run("paterva.v2.DomainToMXrecord_DNS")
```

Runs the To DNS Names [MX] transform on the selected entities

Required parameters

`<value>`	The ID of the transform to run

Optional parameters

`slider`	A value from 12 to 10000 overriding the current slider setting
`ignoreSeenEntities`	Will not run the transform again on an entity that it already has been run on (default is true)
`<transform setting>`	Provides a transform setting programmatically

More examples

```
run("paterva.v2.PersonToEmailAddress_SE", additional:"engineering")
```

Searches for a person's email address on sites containing the term "engineering"

```
run("paterva.v2.DomainToDomain_TLD ", slider:20, ignoreSeenEntities:false)
```

Find top level domains for this domain, returning a maximum of 20 results and running this transform, even if it has run on this entity before.

saveAs

Saves the current graph to a file

Example

```
saveAs("C:\\Temp\\mygraph.mtgx")
```

Saves the current graph to the given file

Required parameters

`<value>`	The path of the graph file. Backslashes need to be escaped with double backslashes (\\).

Optional parameters

`path`	Same as <value>
`suffixDate`	Boolean indicating whether date and time should be appended. Default is true.
`dateFormat`	Specifies the format in which the date and time should be appended. Default is `yyyyMMdd-HHmmssSSS`.

More examples

```
saveAs("C:\\Temp\\mygraph.mtgx ", suffixDate:false)
```

Saves the graph without any date/time suffix

```
saveAs("C:\\Temp\\mygraph.mtgx ", dateFormat:"HHmmss")
```

Saves the graph and appends the time of day

setLayout

Sets the current layout mode and performs a layout of the graph

Example

```
setLayout("Organic")
```

Switches to an organic layout

Required parameters

`<value>`	The name of the layout mode. Possible values are: - Block - Organic - Circular - Hierarchical - Interactive Organic

Optional parameters

`layout`	Same as <value>

`scope`	Use a local or global scope. Global scope will apply the layout to the whole graph, local scope will try to layout only the entities currently in the pipe. Default is global (i.e. layout the whole graph)

More examples

```
setLayout("Interactive Organic")
```

Switches to interactive organic mode

```
setLayout("Circular", scope:"local")
```

Lays out the returned entities using a circular layout

status

Updates the status in the Machine Run Window

Example

```
status("Resolving IP addresses")
```

Sets the staus message to the given text

Required parameters

`<value>`	The new status message

userFilter

Shows a user filter to allow the user to make a manual selection

Example

```
userFilter()
```

Shows a user filter that lists the entities currently in the pipe

Optional parameters

`<value>`	The title text of the user filter
`title`	Same as <value>
`description`	The description below the title
`icon`	A name of an icon to display instead of the default machine icon. This icon needs to have been imported and named using the Icon Manager.
`heading`	The heading text of the first column
`proceedButtonText`	The text on the Proceed button
`removePromptText`	The text of the "Remove unselected entities from graph" checkbox
`removePromptChecked`	A Boolean indicating whether the remove prompt should be selected by default. Default is true.

`selectEntities`	A Boolean indicating whether the entities in the list should be selected or unselected by default. Default is true (selected).
`showIncomingLinks`	A Boolean indicating whether a column with the incoming link count should be shown in the grid.
`showOutgoingLinks`	A Boolean indicating whether a column with the outgoing link count should be shown in the grid.

More examples

```
userFilter("Behold the following entities!!")
```

Displays a user filter with the given title

```
run("Attention", icon:"Virus", removePromptText:"Kill 'em",
proceedButtonText:"Forward unto dawn!", showIncomingLinks:true)
```

Displays a customized user filter with a virus icon and number of incoming links in the grid

Filters

Filters act on the entities in the pipeline and (typically) reduce them based on the type and parameters of the filter.

age

Filters entities based on their age on the graph as measured in seconds since their first appearance

Example

`age(3)`	Matches entities are between 3.0 and 3.9 seconds old

Optional parameters

`<value>`	Matches entities aged the supplied number of seconds
`equalTo`	Same as value
`moreThan`	Matches an entity whose age is more than the given value
`lessThan`	Matches an entity whose age is less than the given value
`invert`	Boolean value indicating whether this filter should be inverted. Default is false.
`scope`	Either global or local. Global scope uses all the entities on the graph, local scope only the entities in the pipe for matching. Default is local.

More examples

```
age(moreThan:3, lessThan:5)
```

Matches entities older than 3 but younger than 5 seconds

```
age(moreThan:3, lessThan:5, invert:true, scope:"global")
```

Puts all the entities on the graph in the pipe that follows this filter if they are younger than three or older than five seconds.

bookmarked

Matches entities based on their bookmark

Example

`bookmarked()`	Matches entities that are bookmarked (with any bookmark)

Optional parameters

`<value>`	The index of the bookmark to match against
`index`	Same as value
`invert`	Boolean value indicating whether this filter should be inverted. Default is false.
`scope`	Either global or local. Global scope uses all the entities on the graph,

	local scope only the entities in the pipe for matching. Default is local.

More examples

`bookmarked(4)`

Matches all entities tagged with a red bookmark

`bookmarked(invert:true)`

Matches all entities that are not bookmarked

degree

Filters entities based on the number of links

Example

`degree(1)`	Matches entities that have one link (incoming or outgoing)

Optional parameters

`<value>`	The number of links
`equalTo`	Same as value
`moreThan`	Matches an entity with more than the given number of links
`lessThan`	Matches an entity with less than the given number of links
`invert`	Boolean value indicating whether this filter should be inverted. Default is false.
`scope`	Either global or local. Global scope uses all the entities on the graph, local scope only the entities in the pipe for matching. Default is local.

More examples

`degree(moreThan:3)`

Matches entities with 4 or more links

incoming

Filters entities based on the number of incoming links

Example

`degree(1)`	Matches entities that have one incoming link

Optional parameters

`<value>`	The number of links
`equalTo`	Same as value
`moreThan`	Matches an entity with more than the given number of incoming links
`lessThan`	Matches an entity with less than the given number of incoming links

`invert`	Boolean value indicating whether this filter should be inverted. Default is false.
`scope`	Either global or local. Global scope uses all the entities on the graph, local scope only the entities in the pipe for matching. Default is local.

More examples

`incoming(lessThan:3)`

Matches entities with two or less incoming links

outgoing

Filters entities based on the number of outgoing links

Example

`degree(1)`	Matches entities that have one outgoing link

Optional parameters

`<value>`	The number of links
`equalTo`	Same as value
`moreThan`	Matches an entity with more than the given number of outgoing links
`lessThan`	Matches an entity with less than the given number of outgoing links
`invert`	Boolean value indicating whether this filter should be inverted. Default is false.
`scope`	Either global or local. Global scope uses all the entities on the graph, local scope only the entities in the pipe for matching. Default is local.

More examples

`outgoing(lessThan:3, invert:true)`

Matches entities with three or more outgoing links

property

Filters entities based on the value of a property

Example

`property("title", equalTo:"test")`	Matches entities whose title property exactly match the string "test"

Required parameters

`<value>`	The name of the property

Optional parameters

`equalTo`	The value to match

`like`	Matches a string anywhere in the value
`ignoreCase`	Ignores case when matching strings. Default is false.
`invert`	Boolean value indicating whether this filter should be inverted. Default is false.
`scope`	Either global or local. Global scope uses all the entities on the graph, local scope only the entities in the pipe for matching. Default is local.

More examples

```
property("name", like:"Brown", invert:true)
```

Matches entities whose name property does not contain the string Brown

type

Filters entities based on type name

Example

`type("maltego.Domain")`	Selects entities of type Domain

Required parameters

`<value>`	The type name of the entity

Optional parameters

`invert`	Boolean value indicating whether this filter should be inverted. Default is false.
`scope`	Either global or local. Global scope uses all the entities on the graph, local scope only the entities in the pipe for matching. Default is local.

value

Filters entities based on their value

Example

`property("test")`	Matches entities whose value exactly match the string "test"

Optional parameters

`<value>`	The value to match
`equalTo`	The value to match
`like`	Matches a string anywhere in the value
`ignoreCase`	Ignores case when matching strings. Default is false.
`invert`	Boolean value indicating whether this filter should be inverted. Default is false.
`scope`	Either global or local. Global scope uses all the entities on the graph,

local scope only the entities in the pipe for matching. Default is local.

More examples

```
value(like:“Frik”, ignoreCase:true)
```

Matches Frik, Frikkie, Dear Frikkie and frikkie