RSA | NetWitness

EPL Essentials

Authored by: Lee Kirkpatrick

Table of Contents

1 Overview 3
2 Esper Processing Language 4
2.1 The Basics 4
2.2 Using timestamp in Events 8
2.3 Comments 9
2.5 Variables 10
2.6 Time-Based Contexts 11
2.7 Output Suppression 13
2.8 Ignoring Case 14
2.9 Named Windows 15
2.10 Event Patterns 16
2.11 Java Lang String Methods 17
2.12 Joins 18
2.13 Arrays 20
2.14 Split 22
3 ESA-Client 23
3.1 ESA Data Feeds 27
4 Enrichment 30
5 Use Cases 33
5.1 Same user VPN followed by RDP to specific network 33
5.2 Device Down 33
5.3 Increase in network traffic by 50% 34
5.4 RDP traffic from same source to multiple destinations 35
5.5 Five Failed Logins from the Same User 35
5.6 Same user failing to logon via VPN from two separate locations 35
5.7 Alert on previously denied connections now being allowed 36
5.8 One machine doing excessive Port 25 Outbound connections 37

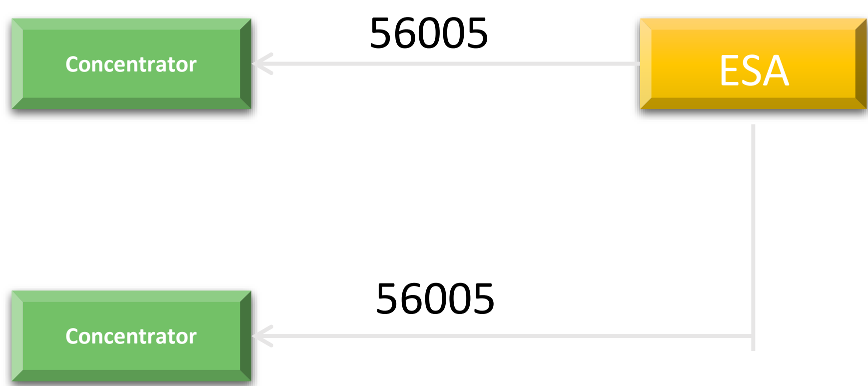
1 Overview

This document is intended to give an overview of how to utilise the Esper EPL language within the NetWitness ESA component.

2 Esper Processing Language

2.1 The Basics

Esper is what allows us within NetWitness to perform advanced correlation of metadata. Said Meta data is consumed by the ESA appliance from one or more Concentrators:



This data is all fed through a stream, which is just a sequence of events. Within NetWitness, this stream is called the "Event" stream.

EPL looks similar to that of SQL, an example of an EPL rule (**that would alert on everything**) is below:

```
SELECT * FROM Event;
```

You can see that we are selecting everything (*) from our stream mentioned earlier, called "Event".

We can also specify to **filter for specific Meta** from our **stream**:

```
SELECT * FROM Event(user_dst = 'Lee')
```

Or **multiple pieces of Meta**:

```
SELECT * FROM Event(user_dst = 'Lee' AND event_cat_name =
'User.Activity.Failed Logins');
```

This can be extended to only alert on a **specific number of events** being seen and **threading (grouping) on a specific variable**:

```
SELECT * FROM Event(user_dst = 'Lee' AND event_cat_name =
'User.Activity.Failed Logins') GROUP BY user_dst HAVING COUNT(*)
> 4
```

We can also add a **time window** for these events to be seen within:

```
SELECT * FROM Event(user_dst = 'Lee' AND event_cat_name =
'User.Activity.Failed Logins').win:time(5 min) GROUP BY user_dst
HAVING COUNT(*) > 4
```

The important item to note with Esper is that the time window we specified above is based upon when the Esper engine sees the events and not the time within the event itself.

There are a variety of these data window types that can be utilised (**see pages 6 – 7).**

Please see the table below for a list of possible data window view types:-

View	Syntax	Description	Example
Length Window	win:length(size)	Sliding length window extending the specified number of elements into the past	SELECT * FROM Event(user_dst IS ‘JohnDoe’).win:length(5) GROUP BY user_dst HAVING COUNT(*) = 5;
Length Batch Window	win:length_batch(size)	Tumbling window that batches events and releases them when a given minimum number of events has been collected	SELECT * FROM Event(user_dst IS ‘JohnDoe’).win:length_batch(5) GROUP BY user_dst HAVING COUNT(*) = 5;
Time Window	win:time(time period)	Sliding time window extending the specified time interval into the past	SELECT * FROM Event(user_dst IS ‘JohnDoe’).win:time(10 sec) GROUP BY user_dst HAVING COUNT(*) = 5;
Externally-timed Window	win:ext_timed(timestamp expression, time period)	Sliding time window, based on the millisecond time value	SELECT * FROM Event.win:ext_timed(timestamp, 10 min) WHERE user_dst = 'Lee'

		supplied by an expression	
Time-Length Combination Batch Window	win:time_length_batch(time period, size)	Tumbling multi-policy time and length batch window	SELECT * FROM Event(user_dst IS 'JohnDoe').win:length_batch(10 sec, 5) GROUP BY user_dst HAVING COUNT(*) =5;
Time-accumulating Window	win:time_accum(time period)	Sliding time window accumulates events until no more events arrive within a given time interval	SELECT * FROM Event(user_dst IS 'JohnDoe').win:time_accum(10 sec);
Keep-all-window	win:keepall()	Simply retains all events	SELECT * FROM Event(user_dst IS 'JohnDoe').win:keepall()
Unique Window	std:unique(unique criteria(s))	Only retains distinct values for a given criteria in a sepearet window for each variable	Create window users.std:unique(user_dst).win:time(20 min) (user_dst string);

Table 1 - Data Windows

2.2 Using timestamp in Events

A useful window to note in the above table is the "**externally-timed window**". This window allows us to specify the time window based on an external timestamp, such as a timestamp stored in a variable from an event:-

```
EPL Statement
SELECT * FROM Event.win:ext_timed(timestamp, 10 min)
WHERE user_dst = 'Lee'
GROUP BY user_dst HAVING COUNT(*) > 1;
```

From the above example, the statement is using an externally-timed window based on the timestamp variable supplied to Esper and looking for those two timestamps to be between 10 minutes. Now, let's say we had the following two events injected into the Esper engine 30 minutes apart, these events contain a timestamp variable that actually states they happened within five minutes:-

```
Events
// Mon, 15 Dec 2014 20:01:00 GMT
Event={user_dst='Lee', timestamp=1418673660000}
// Send next event within 30 minutes
t=t.plus(30 min)
// Mon, 15 Dec 2014 20:06:00 GMT
Event={user_dst='Lee', timestamp=1418673960000}
```

The EPL statement above would render **true** because the events (according to the supplied timestamp fields) where within 10 minutes even though the events according to the engine where seen 30 minutes apart; this is useful for events that arrive in bulk from a file transfer for example.

2.3 Comments

Comments can appear anywhere in the EPL or pattern statement text where whitespace is allowed. Comments can be written in two ways: slash-slash (// ...) comments and slash-star (/* ... */) comments.

Slash-slash comments extend to the end of the line:

```
// This comment extends to the end of the line.
Select * from Event // this is a slash-slash comment
```

Slash-star comments can span multiple lines:

```
/* This comment is a "slash-star" comment
that spans
multiple lines.
*/
```

2.5 Variables

Within EPL variables can be used for a variety of different purposes, below are some examples of how to create variables:

```
create variable string myvalue = 'Lee';
create variable boolean KEEP = true
```

The above variables are only capable of holding one value or state, below is an example of how to create an array and subsequently query it:

```
create variable string[] mylist =
{
'Peter',
'Lee',
'Julie',
'George'
};

SELECT * FROM Event(user_dst IS NOT ALL(mylist));
```

2.6 Time-Based Contexts

Contexts are declared using the 'create context' and can be utilized for a variety of purposes, in the following example, a **context is declared to specify working hours (9am – 5pm):**

```
create context BizHours start (0, 9, *, *, *) end (0, 17, *, *, *);
```

The context can then be invoked by specifying 'context <name>':

```
context BizHours
select * from Event(event_cat_name = 'Firewall Change');
```

The statement above will only render to true if "event_cat_name = Firewall Change" event is seen and is within the hours of 9am -5pm.

The context time is defined as follows:-

```
(minutes, hours, days of month, months, days of week [, seconds])
```

Field Name	Mandatory?	Allowed Values	Additional Keywords
Minutes	yes	0 - 59	
Hours	yes	0 - 23	
Days Of Month	yes	1 - 31	last, weekday, lastweekday
Months	yes	1 - 12	
Days Of Week	yes	0 (Sunday) - 6 (Saturday)	last
Seconds	no	0 - 59	

An alternative method to utilizing a context would be to utilize a variable that changes state at specified time intervals, and then use this variable in a query to check its state. An example of this is given below:

```
create variable string var_on_off;
on pattern[Every(timer:at(*, 9, *, *, *))] set var_on_off = 'true';
on pattern[Every(timer:at(*, 17, *, *, *))] set var_on_off =
'false';
select * from Event(var_on_off='true' AND event_cat_name='Firewall
Change')
```

In the above example, a **variable is being created called 'var_on_off'**. Then the second statement is specifying **at 9am, to change the variable value to 'true'**. The third statement is specifying **at 5pm, to change the variable value to 'false'**. Then within our filter, we can perform a **check to confirm the state of our variable**, which is only true or false during specified hours.

2.7 Output Suppression

In order to achieve output suppression we can utilise the GROUP BY and OUTPUT statements together. The OUTPUT statement is working in tandem with the GROUP BY and suppressing based on the given variables for the time supplied.

```
SELECT      ip_src, dst_port
FROM        Event
GROUP BY    ip_src,dst_port
OUTPUT      first every 5 min
```

RSA

2.8 Ignoring Case

EPL is case sensitive, so any matches that do not specify the exact case of the metadata being evaluated will return false. In order to prevent this from happening with Meta you may not know the case, you can use either of the following java.lang.string methods to either ignore case, or to make everything lower case:

```
select * from
Event(event_cat_name.equalsIgnoreCase('user.activity.failed
logins'))
select * from Event(event_cat_name.toLowerCase() =
'user.activity.failed logins')
```

2.9 Named Windows

A named window is a global data window that can take part in many statement queries, and that can be inserted-into and deleted-from by multiple statements.

The create window clause declares a new named window. The named window starts up empty unless populated from an existing named window at time of creation. Events must be inserted into the named window using the insert into clause. Events can also be deleted from a named window via the on delete clause.

The **create window** statement **creates a named window by specifying a window name** and **one or more data window views**, as well as the **type(s) of event to hold in the named window**.

The example below creates a window to hold user names and IP addresses for one hour:

```
CREATE WINDOW ActiveUsers.win:time(1 hour) (user_dst string,
event_computer string);
```

It is then possible to insert values into the Window by utilizing the 'insert into' clause:

```
INSERT INTO ActiveUsers
SELECT user_dst,event_computer FROM Event(user_dst IS NOT NULL AND
event_computer IS NOT NULL);
```

It is also possible to delete from named windows when a specific event is seen via the on delete clause:

```
ON Event(user_dst IS NOT NULL AND event_computer IS NOT NULL) DELETE
FROM ActiveVPNUsers;
```

2.10 Event Patterns

Event patterns can be used within Esper to perform more complex matching. When employing event patterns within your EPL, you must be sure to wrap it within the following 'pattern[]'. An example of this is below:

```
SELECT * FROM pattern[Event (user_dst IS NOT NULL)]
```

When employing patterns, you have the ability to utilize the followed-by operator. The followed by '->' operator specifies that first the **left hand expression** must turn true and only then is the **right hand expression** evaluated for matching events. An example use of followed by is below:

```
SELECT * FROM pattern[Event(user_dst IS NOT NULL and
event_cat_name='User Created') -> Event(user_dst IS NOT NULL AND
event_cat_name='User Deleted') WHERE timer:within(5 min)]
```

Employing patterns also allows you to take advantage of cache variables and use them for comparisons in other statements:

```
SELECT * FROM pattern[s1=Event(user_dst IS NOT NULL and
event_cat_name='User Created') -> Event(user_dst=s1.user_dst AND
event_cat_name='User Deleted') WHERE timer:within(5 min)]
```

The statement above is **specifying a name for the first filter**, and in the second filter after the followed-by, we are specifying that this events **'user_dst' variable must match that of the first.**

NOTE: When a pattern successfully matches, it will not start matching again. To ensure that the pattern evaluates to true more than once, you must utilise the '**Every'** operator.

The 'Every' operator can be employed like the following:

```
SELECT * FROM pattern[Every (s1=Event(user_dst IS NOT NULL and
event_cat_name='User Created')) -> Event(user_dst=s1.user_dst AND
event_cat_name='User Deleted') WHERE timer:within(5 min)]
```

This will ensure that the pattern will thread on the first statement and continue matching. Take note of the brackets.

2.11 Java Lang String Methods

Within EPL we have the ability to utilize the following java.lang.string.methods. These can help us write EPL by ignoring case or performing other functions:-

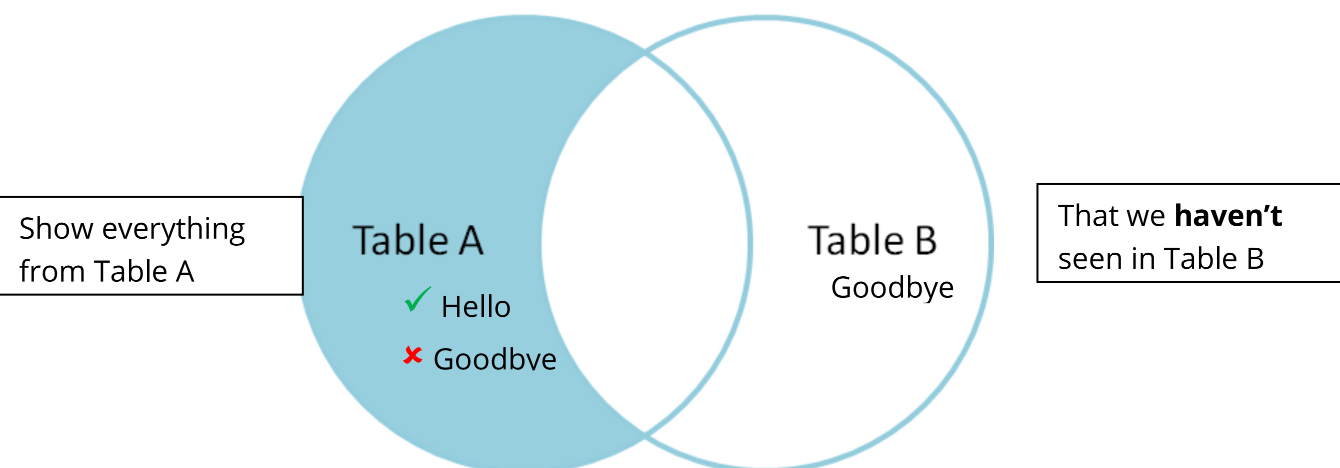
- .toLowerCase() =
- .endsWith('Login')
- .startsWith('User')
- .contains('Activity')
- .equalsIgnoreCase('user.activity.failed login')

These can be used individually or can be used in conjunction with each other like below:-

```
SELECT * FROM Event(event_cat_name.toLowerCase().contains('failed'))
```

2.12 Joins

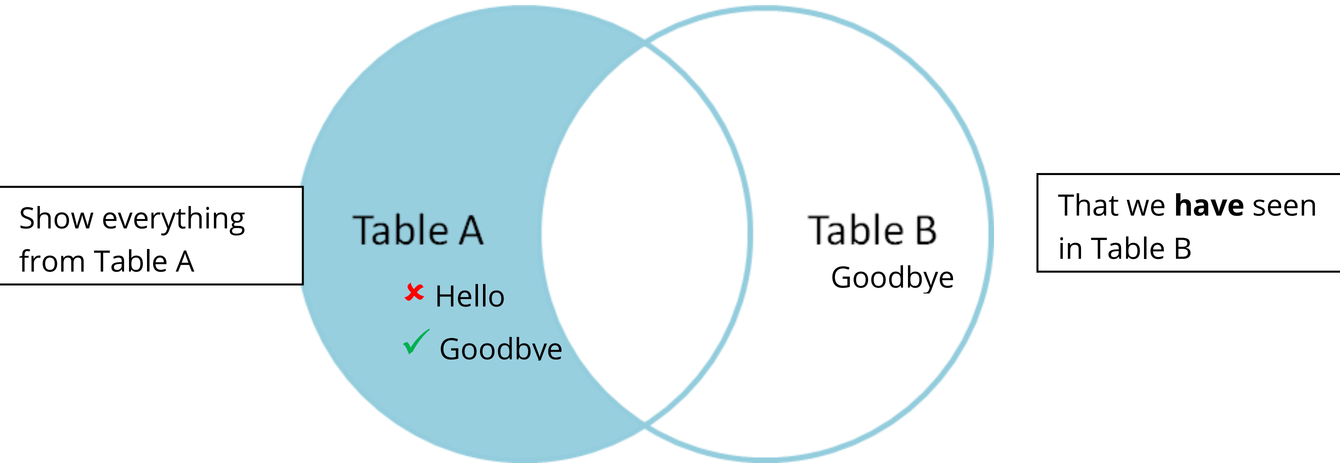
A left outer join produces a complete set of records from Table A, with the matching records (where available) in Table B. If there is no match, the right side will contain null.



For example, if we only wanted to be alerted when we see something in Table A, we **haven't** yet seen in Table B.

```
create schema Event(ip_src string, dst_port int, action string);

create window HoldThis.win:time(20 min) (ip_src string, dst_port
int);

INSERT INTO HoldThis
SELECT ip_src, dst_port FROM Event(ip_src IS NOT NULL AND dst_port
IS NOT NULL AND action='drop');

SELECT Event.ip_src,Event.dst_port
FROM  Event.win:time(20 min)
LEFT OUTER JOIN HoldThis
ON Event.ip_src = HoldThis.ip_src AND
Event.dst_port=HoldThis.dst_port
WHERE HoldThis.ip_src IS NULL AND action='allow';
```

This could also be changed to only show you events from Table A, **we have** previously seen in Table B by changing the WHERE clause.




```
create schema Event(ip_src string, dst_port int, action string);


create window HoldThis.win:time(20 min) (ip_src string, dst_port
int);

INSERT INTO HoldThis

SELECT ip_src, dst_port FROM Event(ip_src IS NOT NULL AND dst_port
IS NOT NULL AND action='drop');


SELECT Event.ip_src,Event.dst_port

FROM  Event.win:time(20 min)

LEFT OUTER JOIN HoldThis

ON Event.ip_src = HoldThis.ip_src AND
Event.dst_port=HoldThis.dst_port

WHERE HoldThis.ip_src IS NOT NULL AND action='allow';
```

2.13 Arrays

Arrays in EPL are treated differently to strings and thus the syntax also differs. The following details how to interact with arrays within EPL.

How to check if an array variable equals 'value'

```
SELECT * FROM Event(action(0) = 'POST')
```

```
SELECT * FROM Event(action(1) = 'POST')
```

How to check if an array variable does not equal 'value'

```
SELECT * FROM Event(action(0) != 'POST')
```

```
SELECT * FROM Event(action(1) != 'POST')
```

How to check if any of the array variables equals ‘value’

```
‘deny’ = ALL( action )
```

How to check if an any of the array variables do not equal ‘value’

```
 ‘deny’ != ALL( action )
```

Using contains and lower case against an array

```
SELECT * FROM Event WHERE action.anyOf(i =>
i.toLowerCase().contains("deny"))
```

Using Regex against an array

```
SELECT * FROM Event where action.AllOf(a=> a regexp '.*GET*.*');
```

Compare multiple values against array variables and ignore case

```
SELECT * FROM Event((isOneOfIgnoreCase(action,{ 'monitor' ,
'session' }))
```

Compare length of variables in array to 'value'

```
alias_host.anyOf(i => i.length()>50)
```

RSA

2.14 Split

Sometimes the metadata that is consumed by the ESA can be in a format that makes correlation difficult, e.g. ‘domain-user’, or maybe we only want to store a specific part of the metadata in a named window or table for comparison purposes.

Using the example above, we can split the ‘domain’ and ‘user’ by using the Esper function **split**:-

```
SELECT * FROM pattern[a=Event(user_dst IS NOT NULL) ->
Event(user_dst.split("-").get(1)=a.user_dst) WHERE timer:within(30
seconds)]
```

Which would then take the value:

- ‘admin-johndoe’ and only return ‘johndoe’

3 Alerting Brief

EPL rules written inefficiently can have a detrimental impact on how the ESA appliance functions – therefore it is important to write effective EPL rules. The following section outlines essential information to keep in mind when writing EPL rules.

3.1 Alerting

Alerts will not be generated by default within the ESA appliance unless the RSA specific annotation is used, this must be added before the EPL statement(s) that are designated to alert, i.e.:

```
@RSAAlert
SELECT * FROM Event(user_dst IS NOT NULL)
```

3.2 Boundaries

All EPL rules that contain windows or grouping should be bounded, either by a time window or event count (unless they are only matching on a single event). Boundaries ensure that the EPL rules do not consume excessive amount of memory over time and will clean up old data that is no longer required. This can be achieved by using EPL views as follows:

```
.win:time(30 min)
.win_time_length_batch(30 min, 10)
```

3.3 Testing

All rules should be tested on Esper's EPL try-out website prior to using in a production environment:

- http://esper-epl-tryout.appspot.com/epltryout/mainform.html

Subsequently, all rules should be put into trial mode on the ESA prior to enabling in production:

- http://sadocs.emc.com/0_en-us/088_SA106/50_Alrt/20_WrkTrRles

3.4 Pattern Matching

When using pattern matching, a new thread will be created for every 'a' event in the first statement below. This means that multiple 'a' events will match with the same 'b' event.

This could result in unexpected and undesirable number of alerts for the same user during the time window. It is recommended to use the hint **@SuppressOverlappingMatches** with the PATTERN syntax using every.

```
SELECT * FROM PATTERN  [
every a = Event(device_class='Web Logs'
AND host_dst = 'icanhazip.com')
-> b = Event(category LIKE '%Botnet%' AND device_class='Web Logs'
AND user_dst=a.user_dst)
where timer:within(300 seconds)
```

3.5 Rule Order

EPL rules will be loaded in the Esper engine based on the time that they have been deployed - first deployed means first loaded in the Esper engine.

There are some scenarios which are based on multiple rules and it is important to define the loading order if there are dependencies between them. You can use the EPL statement "`uses <module_name>`" so that it will force the pre-loading of the required rules, i.e.:

Rule 1

uses createcontext;

Look for login in not working hours

Rule 2

module createcontext;

Create context workinghours

5 ESA-Client

The esa-client is available on all 10.4 and above ESA components and allows you to connect to the jmx-console of the ESA. It can be found at the following location:

```
/opt/rsa/esa/client/bin/esa-client
```

When run, the following prompt will be displayed:

```
localhost:com.rsa.netwitness.esa:/>
```

Typing **help** will give you a list of commands you can use. The main ones I utilize are:

- jmx-ls
- jmx-cd
- jmx-dump
- jmx-invoke

This tool is useful for a quickly viewing statistics, performing configuration changes and interrogating windows. Some examples of these are shown later in the document. For now I will detail some useful locations and commands for information.

How many and what values exist in my window(s)?

Let's say, for example, I created the following window:

```
CREATE WINDOW ActiveUsers.win:time(1 hour) (user_dst string);

INSERT INTO ActiveUsers
SELECT user_dst FROM Event(user_dst IS NOT NULL);
```

And wanted to know how many values are stored in the window, I can perform the following from the esa-client:

```
jmx-cd /CEP/Engine/windows

localhost:com.rsa.netwitness.esa:/CEP/Engine/windows>jmx-invoke
getWindowSize --param ActiveUsers

2
```

If I wanted to see what these 2 values were, I could run the following:

```
localhost:com.rsa.netwitness.esa:/CEP/Engine/windows>jmx-invoke
query --param "SELECT * FROM ActiveUsers"

[{

  "ActiveUsers": {

    "user_dst": "root"

  }

}

, {

  "ActiveUsers": {

    "user_dst": "root"

  }

}

]
```

RSA

5.1 ESA Data Feeds

It is possible to use custom data feeds with Esper (basically a new stream). These allow you to consume information from a CSV file and push it into the Esper engine as if they were real events. This can be useful if you have a CSV file of data you would like to consume into a window for comparison agiant metadata; this could be a list of users or assets for example.

Open the JMX console:

```
/opt/rsa/esa/client/bin/esa-client
```

CD to the following:

```
jmx-cd /Workflow/Source/fileFeedSource
```

And add a **feed source directory** and **stream name**:

```
jmx-invoke addFileSource --param file:///root/test?type=LeeStream
```

If successful you should see the following in the esa.log:

```
2015-02-26 12:51:49,699 [fs-watch-/root/test] INFO
com.rsa.netwitness.core.workflow.worker.source.DirectoryWatcher -
/root/test{Type=LeeStream, Format=csv, Enabled=true, NoDelete=false,
Recursive=true}: Started watching directory /root/test for *.csv
```

Now we must edit the Esper configuration to allow us to use our stream and variables we will specify in out CSV:

```
vi /opt/rsa/esa/conf/esper-config.xml
```

Add the following at the end of the file:

```
<event-type name="LeeStream">
    <java-util-map>
        <map-property name="myuser" class="string"/>
    </java-util-map>
</event-type>
```

Restart the ESA service:

```
service rsa-esa start
```

Ensure the fileFeeds pipeline has been loaded from esa.log:

```
2015-02-26 12:57:14,334 [pool-1-thread-2] INFO
org.springframework.beans.factory.xml.XmlBeanDefinitionReader -
Loading XML bean definitions from file
[/opt/rsa/esa/workflow/pipeline-fileFeeds.xml]
```

Create a CSV file to inject into our stream:

```
myuser string
lee
bob
```

Copy it to our watch directory we specified earlier “/root/test/”:

```
cp /root/users.csv /root/test
```

After moving the file, it will be deleted automatically from our watch directory. You should also see something similar to the following in the esa.log:

```
2015-02-26 12:59:33,596 [pipeline-fileFeeds-0] INFO
com.rsa.netwitness.core.workflow.worker.ESPERFeeder - 2 events in
144 seconds (0 EPS) at minute 2/26/15 12:57 PM forwarded for
correlation.
```

This log shows us that our two events from the CSV have been injected into the Esper engine.

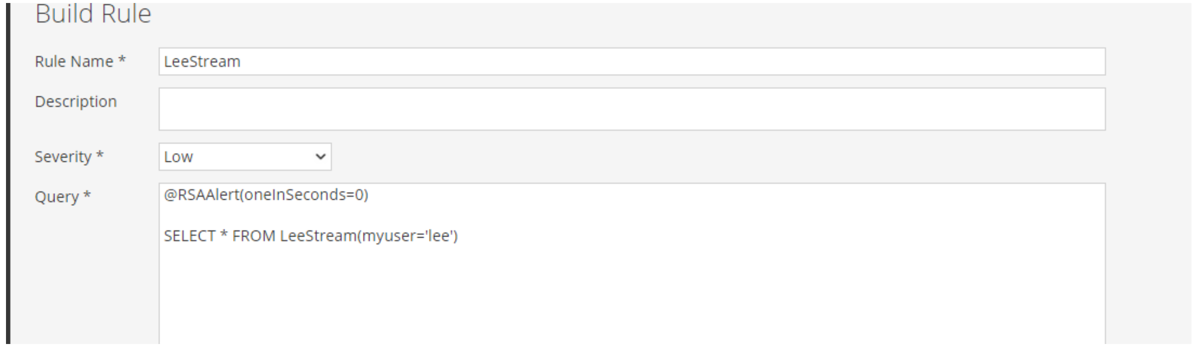
You can also see the number of events offered to Esper has increased:

```
/opt/rsa/esa/bin/dumpJmx.sh |grep -i numeventsoffered
```

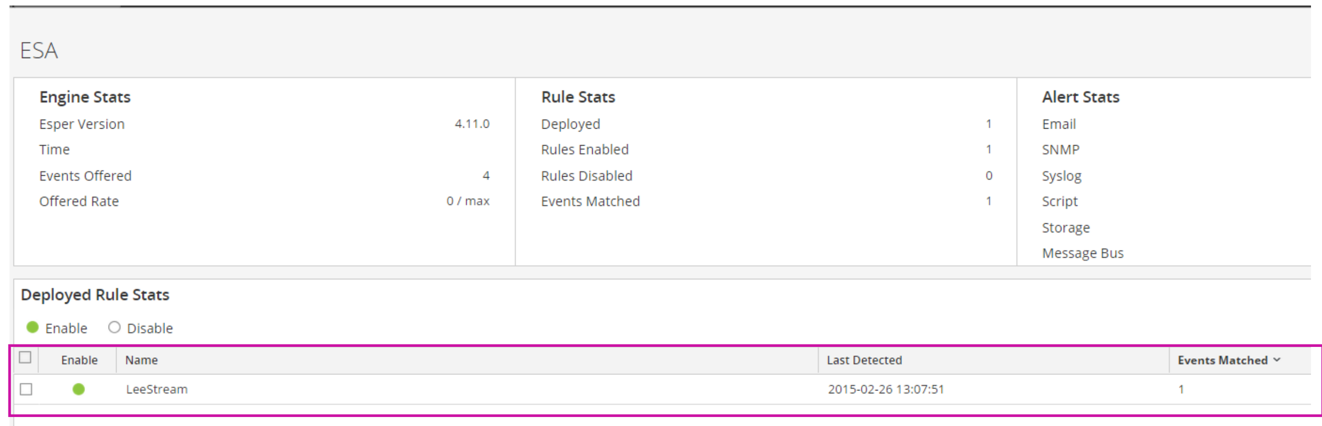
If we wanted to create an alert on this information, we could specify the following:

```
SELECT * FROM LeeStream(myuser='lee');
```

Now if I add my rule to ESA:



Copy the CSV to the watch directory and I can see my alert match:

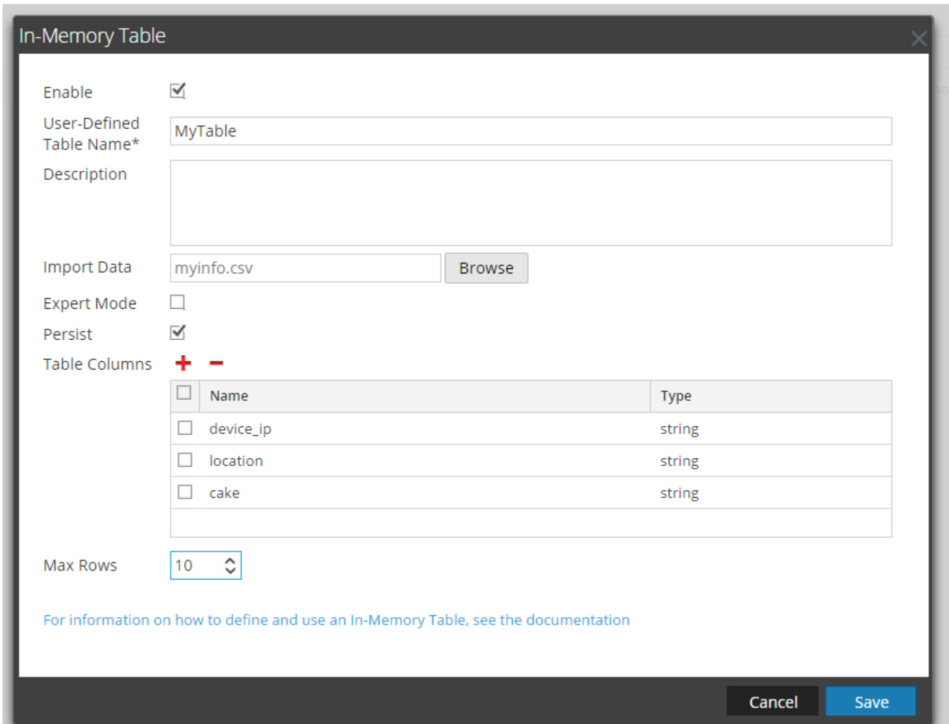


6 Enrichment

Alerts from ESA can be enriched using In-Memory Tables. When an alert fires, it can check if a value is common from the alert and the In-Memory Table and then subsequently enrich the alert with additional Meta.

To configure this, navigate to “Enrichment Sources” from the ESA configure page.

1. Click + and select “In-Memory Table”. The below window will appear:



2. Give the table a name, description and specify a **csv** for the import of the data. The csv should be constructed like the following:

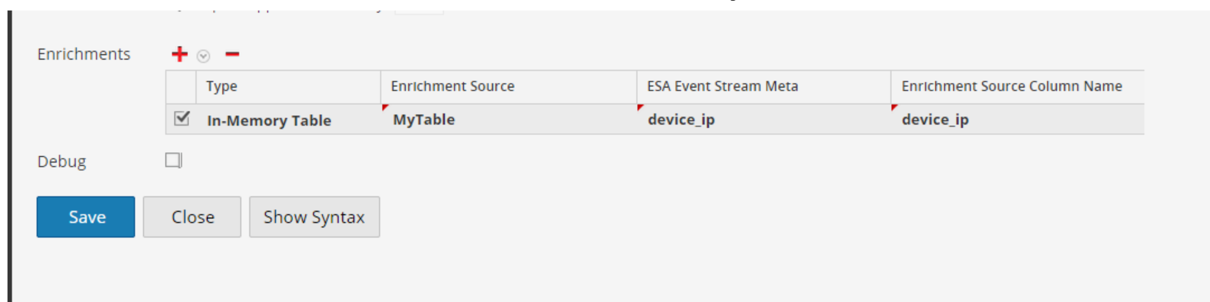
```
device_ip string,location string,cake string
192.168.183.12,The Moon,yes please
```

3. Persist will persist the Window to disk (/opt/rsa/esa/temp/esa-window)
4. Configure the max rows this table can store.
5. When completed click “Save”.
6. Create a “Basic Rule”
7. Under “Enrichments” select + and then “In-Memory Table”

Enrichment Source = the table we created earlier with our values from the csv

ESA Event Stream Meta = the Meta we want to join with the table

Enrichment Source Column Name = the column to join with the Event stream



8. Save and push the rule
9. Look for a log similar to the following to confirm the enrichment has been set:

```
015-03-03 06:36:32,754 [Carlos@68bedd6c-
15(run(SetEnrichmentConnectionRequest))(admin)] INFO
com.rsa.netwitness.core.enrichment.EsperNamedWindowEnrichmentDataSto
re - Prepared an enrichment connection between statement
54f55637e4b07df15d9df78c and source Module_54f554c0e4b07df15d9df78b:
select * from MyTable where device_ip = ?
```

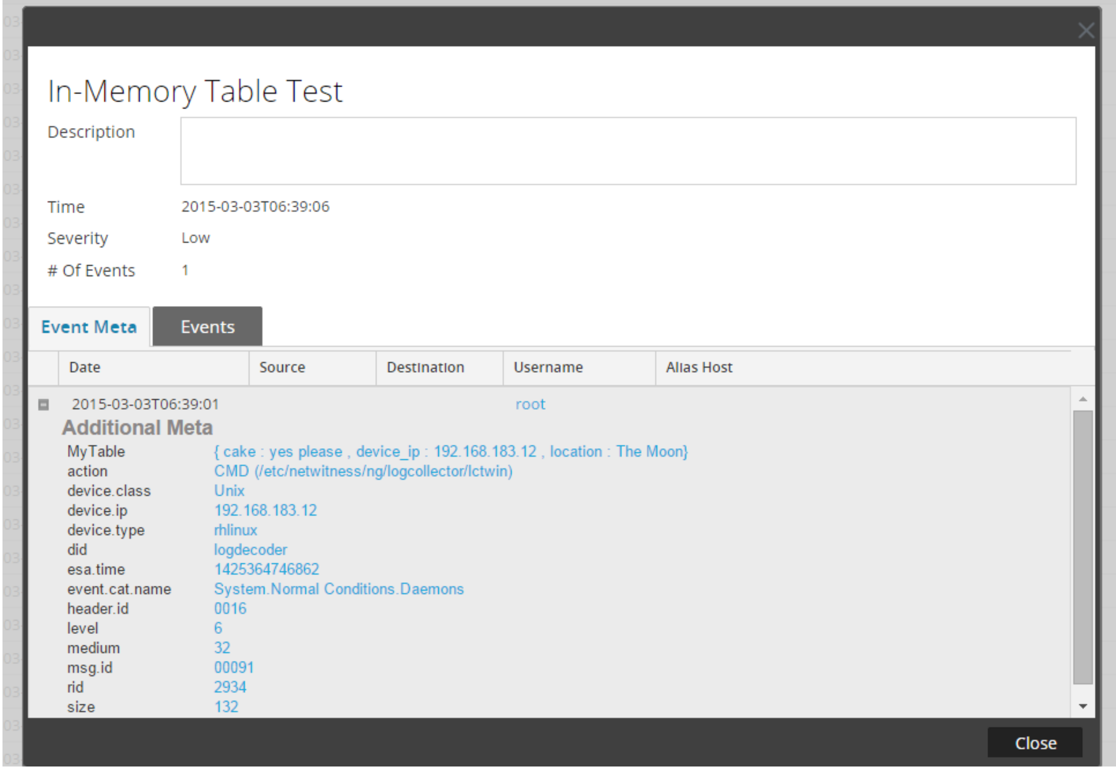
10. From the esa-client, you can also see the enrichment:

```
cd /CEP/Engine/windows

dump

"/CEP/Engine/windows" : {
    "Dirty" : false,
    "Enabled" : true,
    "ManagedWindows" : [ {
      "key" : "Module_54f554c0e4b07df15d9df78b.MyTable",
      "value" : "MyTable=/opt/rsa/esa/temp/esa-window/esa-
Module_54f554c0e4b07df15d9df78b-MyTable-
32f0270c174a79733bbce0ff44aa5aa9031a3161-system-snapshot.obj"
```

11. You can also see the values in the table:

```
localhost:com.rsa.netwitness.esa:/CEP/Engine/windows>invoke query --
param "SELECT * FROM MyTable"
[{
  "MyTable": {
    "device_ip": "192.168.183.12",
    "location": " The Moon",
    "cake": "yes please"
  }
}
]
```

12. When viewing alerts, if a join was successful between your In-Memory Table and Event stream column selection, the enriched Meta can be seen:



7 Use Cases

7.1 Same user VPN followed by RDP to specific network

User connects to VPN and is assigned a temporary IP and the same user RDP's to a specific network range with the same temporary IP:

```
// Create Window to store users and IP assignments
CREATE WINDOW ActiveVPNUsers.win:time(7 days) (user_dst string,
event_computer string);

// Insert into the Window, user and IP values where connected
INSERT INTO ActiveVPNUsers
SELECT user_dst,event_computer FROM Event(user_dst IS NOT NULL AND
event_computer IS NOT NULL AND device_ip='192.168.1.12' AND result =
'connected to gateway' AND device_type='checkpointfw1');

// Remove users from Window when they disconnect
ON pattern[s1=Event(user_dst IS NOT NULL AND event_computer IS NOT
NULL AND device_ip='192.168.1.12' AND result = 'disconnected from
gateway' AND device_type='checkpointfw1')] DELETE FROM
ActiveVPNUsers WHERE (s1.event_computer=(SELECT event_computer FROM
ActiveVPNUsers));

// Check to see if IP in Window RDP’s to a network range
SELECT event_computer FROM Event(event_cat_name=’RDP Connection’ AND
ip_dst LIKE ‘192.168.1.%’) WHERE event_computer IN (SELECT
event_computer FROM ActiveVPNUsers);
```

7.2 Device Down

Heartbeat rule to check that a device has not sent a log for one hour:

```
SELECT * FROM pattern [every(s1=Event(device_ip IS NOT NULL) )->
(timer:interval(1 hour) and not Event(device_ip = s1.device_ip))];
```

7.3 Increase in network traffic by 50%

```
// Create context to start at the top of each hour and end after 60
mins
@Name('context')
create context Hourly start (0,*,*,*,*,*) end after 60 minutes;

// Create Window to store sum bytes and time information
@Name('window')
create window ByteCount.win:time(31 days) as (theHour int,
theWeekDay int, theSum long);

// Insert each hour, the sum of the bytes along with day of week and
hour, each hour
@Name('Insert Values')
context Hourly
insert into ByteCount
select current_timestamp.getDayOfWeek as the WeekDay,
current_timestamp.getHourofDay as theHour, sum(bytes) as theSum from
Event(bytes IS NOT NULL)
output snapshot when terminated;

// Fire alert if traffic is 50% more than baseline
@Name('Fire Alert')
@RSAAlert(onceInSeconds=0)
select * from ByteCount as New
where theSum >= 1.5 * (
select avg(theSum)
from ByteCount as Old
where Old.theWeekDay = hbc.theWeekDay and Old.theHour =
New.theHour);
```

7.4 RDP traffic from same source to multiple destinations

This rule looks for the same IP source connecting to three different destinations within 3 minutes or 3 events:

```
SELECT * FROM Event(
      ip_src IS NOT NULL
      AND ip_dst IS NOT NULL
      AND service =
      ‘3389’).std:groupwin(ip_src).win:time_length_batch(180
      seconds, 3).std:unique(ip_dst) GROUP BY ip_src HAVING COUNT(*)
      = 3;
```

7.5 Five Failed Logins from the Same User

This rule looks for the same user failing to login five times within 5 minutes:

```
select * from pattern[Every
(s1=Event(event_cat_name='User.Activity.Failed Login')) ->
[4]Event(user_dst=s1.user_dst AND
event_cat_name='User.Activity.Failed Login') WHERE timer:within(5
min)];
```

This same rule can also be written without using pattern:

```
SELECT * FROM Event(user_dst IS NOT NULL
      AND event_cat_name=’User.Activity.Failed Login’)
      .std:groupwin(user_dst).win:time(5 min);
```

7.6 Same user failing to logon via VPN from two separate locations

```
select * from pattern[Every (s1=Event(device_class = 'vpn' AND
event_cat_name='User.Activity.Failed Login')) ->
Event(user_dst=s1.user_dst AND event_cat_name='User.Activity.Failed
Login' AND country_src IS NOT s1.country_src) WHERE timer:within(10
min)];
```

7.7 Alert on previously denied connections now being allowed

```
// Window to hold IP and ports
@Name('Window')
create window HoldThis.win:time(1 day).std:unique(ip_src,dst_port)
(ip_src string, dst_port int);


// Insert IP and port pairs into window
@Name('Insert Values')
INSERT INTO HoldThis
SELECT ip_src, dst_port FROM Event(ip_src IS NOT NULL AND dst_port
IS NOT NULL AND action='drop');


// Fire alert if previously denied connection is now allowed
@Name('Fire Alert')
SELECT Event.ip_src,Event.dst_port
FROM  Event.win:length(1)
LEFT OUTER JOIN HoldThis
ON Event.ip_src = HoldThis.ip_src AND
Event.dst_port=HoldThis.dst_port
WHERE HoldThis.ip_src IS NOT NULL AND action='allow';
```

7.8 One machine doing excessive Port 25 Outbound connections

```
SELECT * FROM Event(ip_src IS NOT NULL and dst_port =
25).std:groupwin(ip_src).win:time_length_batch(1 min, 100) HAVING
COUNT (*) > 9;
```