Alerts - Using models to generate custom events

Basic Alerts

An alert is a custom event generated by testing the value of an OID or custom variable and producing a boolean result (true or false). If the test returns true, an event is raised and it will run through the escalation system, false will not raise an alert. Later on, when the test that was returning true once again returns false the event will be cleared.

The values required for the alert are:

```
test => this is a boolean operation using $r to determine if the alert should be raised
event => the name of the event when it is raised
level => the level of the event when it is raised
```

Test

This can be any Perl expression, and its evaluation result will be interpreted like perl does booleans (i.e. empty string, 0, undef means false, anything else means true).
$r holds the value of the oid and you need to use `$r` to test for the condition you want to raise the alert for. All of perl's operators are available, the most likely suspects are:

```
# Numbers
"==" returns true if the left argument is numerically equal to the right argument.
"!=" returns true if the left argument is numerically not equal to the right argument.
"<" returns true if the left argument is numerically less than the right argument.
">" returns true if the left argument is numerically greater than the right argument.
"<=" returns true if the left argument is numerically less than or equal to the right argument.
">=" returns true if the left argument is numerically greater than or equal to the right argument.

#strings
"eq" returns true if the left argument is stringwise equal to the right argument.
"ne" returns true if the left argument is stringwise not equal to the right argument.
"lt" returns true if the left argument is stringwise less than the right argument.
"gt" returns true if the left argument is stringwise greater than the right argument.
"le" returns true if the left argument is stringwise less than or equal to the right argument.
"ge" returns true if the left argument is stringwise greater than or equal to the right argument.
```

A quick note on stringwise versy numerical comparison: in numeric mode, the expressions will be converted to numbers (i.e. string "0003" becomes the number 3 for comparison). In string mode the expressions' characters are compared one by one. If $r is "0003", `$r == 3` is true, but `$r eq 3` is false.

Where to add the Alert

The alert can be added to current variables being polled from the devices, or a new section can be added. For example a new section in Model->system->sys could be added which might look like the example below (the "--snip--" indicates that extra model code has been removed for clarity):

```
%hash = (
  --snip--
  'system' => {
     --snip--
    'sys' => {
      'standard' => {
        --snip--
      },
      'alerts' => {
        'snmp' => {
          'tcpCurrEstab' => {
            'oid' => 'tcpCurrEstab',
            'title' => 'TCP Established Sessions',
            'alert' => {
              'test' => '$r > 250',
              'event' => 'High TCP Connection Count',
              'level' => 'Warning'
            }
          }
        }
      },
       --snip--
    }
    --snip--
  }
  --snip--
);
```

Adding the alert also adds the information to the "Device Details" panel, so you get the last polled value displayed all the time. Note that when you add such a basic alert its variable is collected independently of any other variables that your model might collect.

TCP Established Sessions	106

Example

The following is an example of the layout of an alert (in this example serialNum is taken from Model-CiscoRouter.nmis) and uses a string based (stringwise) comparison:

```
'serialNum' => {
  'oid' => 'chassisId',
  'title' => 'Serial Number',
  'alert' => {
    'test' => '$r ne "SomeSerialNumber"',
    'event' => 'Serial Number Invalid',
    'level' => 'Critical'
  }
}
```

$r in this case is the value of the OID chassisId. This will raise the event "ALERT: Serial Number Invalid" when the oid chassisId is not equal to "SomeSerialNumber.

A numeric based comparison can also be used, in this case when the value is 0 the alert is raised, when the value is not 0 the alert is cleared:

```
'cipSecGlobalActiveTunnels' => {
  'oid' => 'cipSecGlobalActiveTunnels',
  'title' => 'Global Active Tunnels',
  'alert' => {
    'test' => '$r == 0',
    'event' => 'No tunnels present',
    'level' => 'Critical'
  }
}
```

More Advanced Alerts

Alerts can also be created in the 'alerts' section of the model. lerts created in this section have the advantage of being able to use values from a whole section of data to determine if the alert should be triggered or not; however, such alerts **can NOT access** variables collected/modelled in the 'system' section and as such are mostly useful for systemHealth modelling.

If the model does not have such an 'alerts' section, it can be added at the outermost level of the model, e.g. along side '-common=' and 'system'.

A concrete example always makes things more clear.

```
%hash = (
 '-common-' => {
   -- snip --
 },
 'system' => {
   -- snip --
 },
 'storage' => {
   -- snip --
 },
 'alerts' => {
   'services' => {
     'HighProcessMemoryUsage' => {
       'type' => 'test',
       'test' => 'CVAR1=hrSWRunPerfMem;$CVAR1 > 300000',
       'value' => 'CVAR1=hrSWRunPerfMem;$CVAR1 * 1',
       'unit' => 'KBytes',
       'element' => 'hrSWRunName',
       'event' => 'High Process Memory Usage',
       'level' => 'Warning'
     }
   }
 }
);
```

Let's break down the above example.

- 'services' defines what section the values being used for the alert are taken from. In this case services won't be found in the model because it is a special section just for servers. Normally you will not need to worry about special sections. Please note that you **CANNOT** use the 'system' section for advanced alerts!
- 'HighProcessMemoryUsage': this creates a label/id for the alert
- 'type' => 'test': this means the alert will test a single condition. The options are ['test', 'threshold-rising', 'threshold-falling']
- 'test' => 'CVAR1=hrSWRunPerfMem;$CVAR1 > 300000' defines a custom variable and then uses that variable to perform a boolean test. See the paragraph below regarding custom variables.
- 'value' => 'CVAR1=hrSWRunPerfMem;$CVAR1 * 1': this defines how the value that triggered the alert should be reported and displayed when the alert is shown in the GUI
- 'unit' => 'KBytes': the unit that the above value will be displayed with
- 'element' => 'hrSWRunName': which OID/value that has the problem, a descriptor or identifier. In this case it is showing the name of the process that has high memory usage.
- 'event' => 'High Process Memory Usage': sets the name of the alert event
- 'level' => 'Warning': the level the event will be triggered with. When using thresholding this is not used as the thresholds define the level.

Custom Variables

Please note that in NMIS versions before 8.6 you can only use one custom variable in a `test` expression, namely `CVAR`. This limitation has been removed in NMIS 8.6, and the limitation also never applied to `value` or `control` expressions.

As described in more detail on the Advanced Modelling page, your `test`, `control` and `value` expressions may contain custom variable definitions and accesses. The syntax is straightforward:
`CVAR=snmp_var_name;` or `CVAR3=other_snmp_var;` looks for the named snmp object value (which must be listed for collection in the model!) and saves it in one of eleven custom variables (`CVAR` and `CVAR0` to `CVAR9`). When you want to access that saved value, you use `$CVAR`, `$CVAR3` etc.

Please note that the substitution is performed on a purely textual basis before perl is given the expression to evaluate; if you want to use string variable contents you should double-quote your access expressions, e.g.

```
CVAR2=ifAlias; "$CVAR2" =~ /some description/
```

Thresholds

While boolean tests are nice it is often much more useful to specify levels of acceptance instead of just on or off, thresholds allow us to do this. Another example:

```
'alerts' => {
 'storage' => {
  'HighDiskUsage' => {
   'type' => 'threshold-rising',
    'threshold' => {
     'Normal' => '70',
     'Warning' => '75',
     'Minor' => '80',
     'Major' => '95',
     'Critical' => '98',
     'Fatal' => '99',
    },
   'test' => '',
   'value' => 'CVAR1=hrStorageSize;CVAR2=hrStorageUsed;$CVAR2 / $CVAR1 * 100',
   'element' => 'hrStorageDescr',
   'unit' => '%',
   'event' => 'High Disk Usage',
   'level' => '',
   'control' => 'CVAR=hrStorageType;$CVAR =~ /Fixed Disk/',
  },
 }
```

I will just outline the differences here.

- 'type' => -- is set to 'threshold-rising', this means that test will be ignored and the value in 'value' will be compared against the threshold provided and will define the level of the alert.
- 'threshold' => – defines a set of threshold values , the values in this hash must make sense when compared against the value defined below
- 'value' => – this defines a single value that will determine the level based threshold. Custom variables are supported and may be used to calculate the value. They can hold any OID from within the same section defined in the model.
- 'test' => '' – notice that this is blan (or can be omitted altogether) as it does not make sense to ask for a boolean value when we want a graduated, non-boolean result.
- 'control' => – defines a boolean, when true the threshold is run against the specific item, in this case only "Fixed Disk" items should match this alert, if you look in the net-snmp model you will see another alert for Memory that defines different threshold values. Again, custom variables are supported.

Also notice that 'level' is missing, threshold determines this value.