Monitoring Your Network with PRTG - Custom Sensors Part 1

Joe Pruitt, 2012-29-08

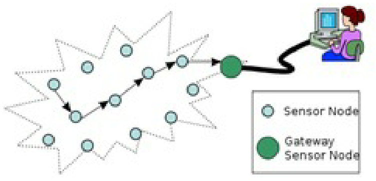
Articles in this series:

- Monitoring Your Network with PRTG - Overview, Installation, and Configuration

In my first article in this series, I talked about the PRTG Network Monitor from Paessler and how we decided to use it for our DevCentral migration from our IT datacenter into a cloud provider. In this article, I'm going to talk about the core of PRTG's monitoring: it's sensors.

What is a Sensor?

As the PRTG folks state: A sensor is the "heart" of the PRTG installation. In it's essence, a sensor is the term PRTG uses for a monitor. When you think of how you would monitor a device or service, that is a sensor. The list of available sensor types is extensive covering everything from simple ping and port checks to HTTP, Web Service, SNMP, Windows/WMI, Linux/Unix/OS X, Virtual Server, Mail Servers, SQL, File, VoIP, QoS, and Hardware.   A full list can be found here: Available Sensor Types.



The sensor type I find most interesting is the "Custom Sensor" type.   If, for some reason, there is something you need to monitor that is not in their list of built-in sensors, It is here that you can create custom scripts or executables to do whatever you want.   Their native support for PowerShell was a big plus for me.

For the most part, we were able to make use of the built-in sensors for our network devices and application servers but this series wasn't meant to be a tutorial on how to configure all of PRTG, so I'm not going to go into the configuration for all our basic sensors. What I would like to get into discussing how we developed our own set of custom sensors to monitor our F5 BIG-IP devices and our applications.

Custom Sensors

Before I dig into the sensors we created, I'll talk a little bit about how the custom sensors work.   You can create either a console based executable or you can write a script with one of their supported scripting languages. At the time of this article, these included BAT, CMD, VBS, and PowerShell. Being the PowerShell geek that I am, I went for the PowerShell option as it allowed me to easily test and deploy the scripts.

There are two modes the scripts can run under: Standard and Advanced. Standard scripts support a single channel (or data point). Since I wanted to create an aggregate sensor, I opted for the Advanced mode.

You return the status of the Advanced monitor in an XML format in the form of

```
<prtg>
  <result>
    <channel>Channel Name</channel>
    <value>xxx</value>
    ... other data
  </result>
  <result>
  <channel>Channel Name #2</channel>
    <value>yyy</value>
    ... other data
  </result>
  ...
</prtg>
```

or an error in this form:

```
<prtg>
  <error>1</error>
  <text>your error message</text>
</prtg>
```

The values you can supply for the “other data” are dependent on the data type and of the channel data and how you want it interpreted. The following table lists the available data values

<Tag>	Description	Possible Content
<Channel>	Name of the channel as displayed in user interfaces. This parameter is required and must be unique for the sensor.	Any string
<Unit>	The unit of the value. Default is Custom. Useful for PRTG to be able to convert volumes and times.	BytesBandwidth BytesMemory BytesDisk Temperature Percent TimeResponse TimeSeconds Custom Count CPU (*) BytesFile SpeedDisk SpeedNet TimeHours
<CustomUnit>	If Custom is used as unit this is the text displayed behind the value.	Any string (keep it short)
<SpeedSize> <VolumeSize>	Size used for the display value. E.g. if you have a value of 50000 and use Kilo as size the display is 50 kilo #. Default is One (value used as returned). For the Bytes and Speed units this is overridden by the setting in the user interface.	One Kilo Mega Giga Tera Byte KiloByte MegaByte GigaByte TeraByte Bit KiloBit MegaBit GigaBit TeraBit
<SpeedTime>	See above, used when displaying the speed. Default is Second.	Second Minute Hour Day
<Mode>	Selects if the value is a absolut value or counter. Default is Absolute.	Absolute Difference
<ShowChart>	Init value for the Show in Chart option. Default is 1 (yes).	0 (= no) 1 (= yes)
<ShowTable>	Init value for the Show in Table option. Default is 1 (yes).	0 (= no) 1 (= yes)
<Warning>	If enabled in at least one channel, the entire sensor is set to warning status. Default is 0 (no).	0 (= no) 1 (= yes)
<Float>	Define if the value is a float. Default is 0 (no). If set to 1 (yes), use a dot as decimal seperator in values. Note: In the sensor's Channels tab, you can define the number of decimal places shown in tables. Default is All.	0 (= no, integer) 1 (= yes, float)
<Value>	The value as integer or float. Please make sure the setting matches the kind of value provided. Otherwise PRTG will show 0 values.	Integer or float value
<Text>	Text the sensor returns in the Message field with every scanning interval. There can be one message per sensor, regardless of the number of channels. Default is OK. Note: This has to be provided outside of the element.	Any string

So, all that’s left is to determine what you want to monitor, create a response with each data point having a result element in the output from the script and you are good to go.

In our setup, I created 2 custom health monitors. The first one was for monitoring our F5 devices and the second was for our application stack.

F5-HealthCheck

The usage for the script is as follows:

```
PS> F5-HealthCheck.ps1 -Server server -Community community -Type type
```

The parameters are:

- server - The IP address for the management port of the F5 device.
- community - The community string for the SNMP requests.
- type - The server type we want metrics for. Possible values of egw, asm, or gtm.

I chose to use SNMP for this monitor instead of iControl as I already had the OIDs needed and there were some system metrics that iControl did not provide so instead of creating a hybrid script, I chose to go all SNMP.

The script itself has a bunch of internal data in the OIDs that I can’t share but I’ll throw in some of the logic of how the script was implemented.

The main script logic checks for the validity of the input parameters and then will call the Get-ChannelData function which I will describe below. Otherwise usage messages are displayed which are mainly helpful when debugging or running the script manually.

```
if ( ($Server -eq $null) -or ($Community -eq $null) -or ($Type -eq $null) )
{
  Write-Host "Usage: F5-Healthcheck.ps1 -Server server -Community community -Type <type>";
  Write-Host "Type";
  Write-Host "------------";
  $OIDHASH.Keys | Sort-Object Exit;
}
if ( Verify-ServerAndType -Server $Server -Type $Type )
{
  $OIDList = $OIDHASH[$Type];
  if ( $null -ne $OIDList ) {
    $s = Get-ChannelData -Server $Server -Community $Community -OIDList $OIDList;
    $Res = @"
<prtg>
  $s <Text>$($script:STATUS)</Text>
</prtg>
"@;
  $Res;
  } else {
    Write-Host "Invalid Type. Valid types are:";
    Write-Host $OIDHASH.Keys;
  }
} else {
  Write-Host "Invalid Type '$Type' for Server $Server of type '$($SERVERHASH[$Server])'"
  Write-Host "Valid Types:"
  $OIDHASH.Keys | Sort-Object | Where-Object { $_.StartsWith($SERVERHASH[$Server]) }
}
```

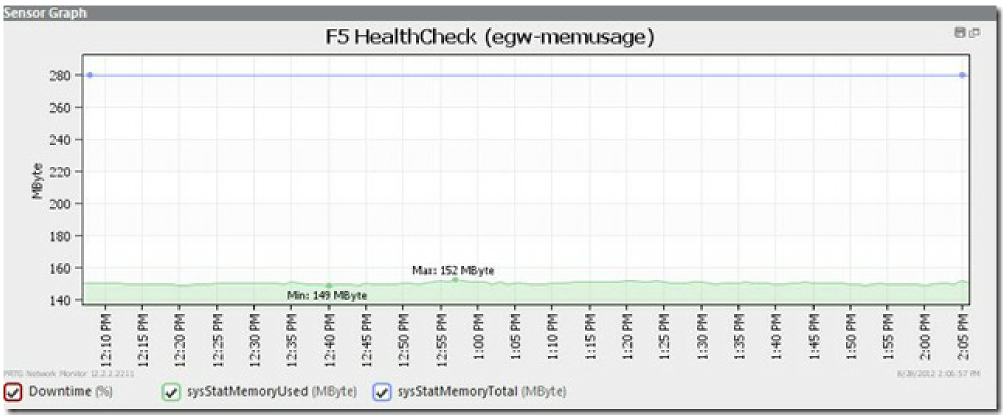
The Get-ChannelData function will take a list of OIDS, query their values, and then produce an output of results for PRTG to interpret as channels.

```
function Get-ChannelData() {
  param(
    $Channel = $null,
    $Server = $null,
    $Community = $null,
    $OIDList = $null
  );
  $s = "";
  foreach($OIDEntry in $OIDList) {
    $Name = $OIDEntry[0];
    $Unit = "Custom";
    $CustomUnit = $OIDEntry[1];
    $OID = $OIDEntry[2];
    $Mode = "Absolute";
    if ( $CustomUnit -eq "##" ) {
      $Mode = "Difference";
    }
    $Warning = 0;
    $v = Get-SNMPValue -Server $Server -Community $Community -OID $OID;
    if ( $v -eq "" ) {
      $Warning = 1;
      $Value = 0;
      $script:STATUS = "Warning: Error in SNMP request"
    }
    $Value = $v.Value;
    if ( $CustomUnit -eq "Megabytes" ) {
      $Value = $v.Value / 1000000.0;
    }
    elseif ( $CustomUnit -eq "Megabits" ) {
      $Value = 8 * (($v.Value)/ 1000000.0);
    }
    elseif ( $CustomUnit -eq "Bytes" ) {
      $Unit = "BytesBandwidth"
      $Mode = "Difference"
    }
    elseif ( $CustomUnit -eq "BytesMemory" ) {
      $Unit = "BytesMemory"
    }
    elseif ( $CustomUnit -eq "Requests" ) {
      $CustomUnit = "Req" $Mode = "Difference"
    }
    $s += @"
  <result>
    <channel>${Name}</channel>
    <unit>${Unit}</unit>
    <CustomUnit>${CustomUnit}</CustomUnit>
    <mode>${Mode}</mode>
    <showChart>1</showChart>
    <showTable>1</showTable>
    <warning>${Warning}</warning>
    <float>0</float>
    <value>$($Value)</value>
  </result>`r`n
"@;
  }
  $s;
}
```

The Get-SNMPValue function is just a wrapper around an SNMP GET returning the value as the output.
So, running the script for ASM memory usage against one of our Edge Gateway devices results in the following output.

```
PS C:\Program Files (x86)\PRTG Network Monitor\Custom Sensors\EXEXML> .\F5-HealthMonitor.ps1 -Server 10.50.4.13 -type egw-memusage
<prtg>
  <result>
    <channel>sysStatMemoryUsed</channel>
    <unit>BytesMemory</unit>
    <CustomUnit>BytesMemory</CustomUnit>
    <mode>Absolute</mode>
    <showChart>1</showChart>
    <showTable>1</showTable>
    <warning>0</warning>
    <float>0</float>
    <value>158022008</value>
  </result>
  <result>
    <channel>sysStatMemoryTotal</channel>
    <unit>BytesMemory</unit>
    <CustomUnit>BytesMemory</CustomUnit>
    <mode>Absolute</mode>
    <showChart>1</showChart>
    <showTable>1</showTable>
    <warning>0</warning>
    <float>0</float>
    <value>293601280</value>
  </result>
  <Text>OK</Text>
</prtg>
```

And PRTG will read this in and generate the following sensor with the two defined channels for the used and total memory for the device:



Conclusion

The PRTG Network Monitor makes it very easy for you to thoroughly monitor your applications and network with it's vast array of built-in sensors. But, the real power lies in the ability to build your own - plus it's fun!

In the next article in this series, I'll discuss the second custom sensor I created that will monitor the health of our application stack.