chuco

Chuco Gives a FAQ 2024: PowerShell Sensor Authoring Guide

Version: 1.0

Last Update: Oct 22nd, 2024

Table of Contents

When to use PowerShell for sensor code

PowerShell is useful for more advanced operations on Windows platforms only. While it can be run on NIX systems, Tanium does not directly support it. Keep in mind the native PowerShell interpreter is used. If the sensor code is written for a version higher than what exists on the system, you will encounter errors.

PowerShell is relatively efficient on the endpoint, especially when using .NET methods directly instead of relying on cmdlets. It is currently best practice to use .NET directly in your script.

Testing your code

Test!Test!Test!
It is important to remember that your script will be running in the 32-bit interpreter on the endpoint. Therefore, you will be subject to 32-bit path redirection. This applies to registry, file, and environment paths.

One way to run your code in 32-bit is to wrap the entire script in a script block to run in a job.

Unset
```
$script = Start-Job -ScriptBlock {

#PowerShell commands here

} -RunAs32
$script | Wait-Job | Receive-Job
```

You will also be running as the SYSTEM user, which has differences compared to running as a domain or local user. PSExec can allow you to open any application as SYSTEM. Refer to their documentation for current commands.

Writing PowerShell Sensors

Below are examples of writing PowerShell sensors, including how to return results, handle parameters, and access some Tanium object.

Basic Structure

This is the basic structure of a PowerShell sensor. Note that we should always use Write-Output to return data to the console.

We also cleanly handle exceptions and return them to the console as a TSE-Error.

Unset
```
$ErrorActionPreference = "Stop"
function Get-Main {
   # This is where your code goes.
   $result = "Do Something"
   return $result
}

try {
   $r = Get-Main
}catch {
   $r = "TSE-Error: $($_.Exception.Message.replace("`n", "->"))"
}finally {
   Write-Output $r
}
```

Multiple Results

Sometimes, we need to return numerous lines to the console. To achieve this, we only need to modify our boilerplate a little. We will add our lines to an array list. Unlike arrays in PowerShell, we can easily add to the array list without the entire array being recreated every time. This happens when we run the command $array += “string”.

```
Unset
function Get-Main {
    # You can return a list.
    $result = New-Object System.Collections.ArrayList
    [void]$result.add("Do Something")
    [void]$result.add("Do Another Thing")
    return result
}
```

Notice we used “[void]” on every line where we added to the list. This prevents the return of the add method from making its way back to the console.

To split results into multiple columns, just like any other language, use a delimiter:

```
Unset
function Get-Main {
    # Using | as a delimiter.
    columnData = @("data_a", "data_b", "data_c")
    return columnData.join("|")
}
```

Parameterized Sensors

Sensor parameter strings must be URL decoded. For more details, see Tanium's Documentation. We have a few ways to decode the sensor parameters in PowerShell sensors. There is a simple .NET one-liner, and the $Tanium object has a method for this. Both methods are in the example below.

Let's say we have four parameters with the types of: text, number, checkbox, and date time. These parameters will be substituted into the scripts as URL encoded strings, and the values below are enclosed in double pipes. (e.g. `||text||`) Often, we want to convert or cast those parameters into different types. In most cases, we can just do a simple type conversion.

Unset
```
function Get-Main {
    $text1 = [System.Uri]::UnescapeDataString('||text1||')
    $text2 = $Tanium.UnescapeFromUTF8('||text2||')
    [bool]$check = [int]'||check||'
    [int]$number = '||number||'
    $dateTime = [DateTimeOffset]::FromUnixTimeMilliseconds('||dateTime||')
}
```

Getting the Tanium Directory Path

You can get the path to the Tanium Client directory from the $Tanium object using `$Tanium.GetConfig("Path")`

Tanium subdirectories

Unlike the Python module, we do not have a quick built-in way to pull back Tanium subdirectories. Nevertheless, we can keep a simple function in our back pocket to achieve the same effect.

```
Unset
function Get-TaniumPath{
  param(
    $subDir
  )
  $path = $Tanium.GetConfig("Path")
  foreach($sub in $subDir){
    $path = Join-Path $path $sub
  }
  return $path
}
Get-TaniumPath
  #Example output: C:\Program Files (x86)\Tanium\Tanium Client
Get-TaniumPath -subDir @("Tools","StdUtils")
  #Example output: C:\Program Files (x86)\Tanium\Tanium Client\Tools\StdUtils
Get-TaniumPath -subDir "Tools\SoftwareManagement"
  #Example output: C:\Program Files (x86)\Tanium\Tanium
Client\Tools\SoftwareManagement
```

Query Sensor Results (Sensorception)

Sometimes, you need to modify or further process the results of an existing sensor. You might copy the sensor and modify the code. This method of copying sensors maintained by Tanium has some drawbacks, most notable is that they can break if the sensor relies on Tanium specific functionality. A good example of this is Custom Tags. The location of custom tags has changed in the past, breaking custom content. When appropriate, you can query the results of a Tanium sensor within a PowerShell sensor using `$Tanium.EvaluateSensorResultByName()`.

Multi-row

The new line character (`n) can split multi-row data. This allows you to iterate through each row using a loop.

```
Unset
function Get-Main {
    # This is where your code goes.
    $customTags = $Tanium.EvaluateSensorResultByName("Custom Tags")
    $customTags = $customTags.split("`n")
    foreach($tag in $customTags){
        # Do Something
    }
}
```

Multi-row and Multi-column

Let's say you want to get the first column of data from a sensor. We will first split the rows using a new line, then split each row by the column delimiter. The example will return only the application names that are installed.

```
Unset
function Get-Main {
    $installedApplications = $Tanium.EvaluateSensorResultByName("Installed
Applications").split("`n")| Foreach-Object {$_.split("|")[0]}
    return $installedApplications
}
```

Parameterized

Parameterized sensors can be called by passing the sensor parameters as a hash literal.

```
Unset
$tagExists = $Tanium.EvaluateSensorResultByName("Custom Tag Exists", @{tag =
"test"; exactMatch = "0"})
```

$Tanium Object

The $Tanium object will be available for use inside your PowerShell sensors. There is no need to import anything.

Name	**Type**	**Return Type**	**Use**
ClearSensorQuarantine	Method	void	ClearSensorQuarantine ()
CXMailboxRequest	Method	string	CXMailboxRequest (string, int, string)
EvaluateSensorResultByHash	Method	string	EvaluateSensorResultByHash (int, Variant)
EvaluateSensorResultByName	Method	string	EvaluateSensorResultByName (string, Variant)
GetArguments	Method	SAFEARRAY(Variant)	GetArguments ()
GetConfig	Method	Variant	GetConfig (string)
GetConfigKeys	Method	SAFEARRAY(Variant)	GetConfigKeys (Variant)
GetQuarantinedSensors	Method	SAFEARRAY(Variant)	GetQuarantinedSensors ()
GetSensorHistory	Method	SAFEARRAY(Variant)	GetSensorHistory (int)
GetSensorStatisticsByHash	Method	Variant	GetSensorStatisticsByHash (int)
GetSensorStatisticsByName	Method	Variant	GetSensorStatisticsByName (string)
Log	Method	void	Log (int, string)
RemoveConfig	Method	void	RemoveConfig (string)
SetConfig	Method	void	SetConfig (string, Variant)
SetConfigProtected	Method	void	SetConfigProtected (string, string)
UnescapeFromUTF8	Method	string	UnescapeFromUTF8 (string)
UnquarantineSensorByName	Method	bool	UnquarantineSensorByName (string)
Result	Property	string	Result () {set}
Version	Property	string	Version () {get}