QA Checks
Developers Guide

Informational Documentation

This document is intended for developers who want to extend the functionality of the QA Checks.

It is assumed that you have a good working knowledge of PowerShell and are familiar with Windows scripting languages as well as how to get information out of the Windows operating system via WMI, the registry and other means.

Note: This document refers to version 3.0 and above.

Contents

Overview

The QA Checks (aka scripts) came about as a need to verify the build of new servers into the ASC, AHS or other environments. All servers should be built from a standard gold build image; however this image still lacks many of the additional tools and configuration settings that are needed before a server can be taken in to BAU support.

The core list of checks consists of all the ASC QA requirements. Additional checks are available for other environments. For example, there are 6 extra checks for the AHS servers. Each environment has its own script. The core script is called QA.ps1 with customer specific ones having a three letter abbreviation: QA-AHS.ps1. The rest of this document will only refer to the QA.ps1 script.

Technical Details

The scripts are written using the Microsoft PowerShell scripting language, with a minimum version of 2.0. This is due to Windows Server 2008 R2 (the lowest supported operating system) having this version installed by default.

The scripts can be run on any Windows operating system as long as PowerShell version is installed, and the PowerShell command window is run with administrative privileges. The only supported operating systems are listed below...

Supported Operating Systems

Windows Server 2008 R2
Windows Server 2012
Windows Server 2012 R2

Unsupported Operating Systems (but still work)

Windows 2003 Server
Windows Server 2016 Technical Preview

Script Overview

This section will detail the scripts and their usage and function within the overall QA Script

Folder Layout

The image below shows the current folder layout for the QA scripts.

```
\-- QA
   \-- checks
   \-- engine
   \-- functions
   \-- settings
```

The files and folders shown are all explained in the next section.

Root (QA) Folder

The QA script file is automatically generated from all the other files by using the compiler script; therefore it is not recommended that you make any changes to this file as they will be lost the next time a compile is run. The QA script file will have the compiled date appended to it, for example: qa_v3.16.0902.ps1

The compiler.ps1 file is an important one. It brings together all the files in the engine folder, as well as the checks scripts into the QA files. It works by searching the checks folder and any subfolders for all .ps1 file starting with c-. See the section on compiling new scripts for more details.

Checks

The checks folders are the ones that hold the actual scripts that perform each check or fix. The file name layout in each of these folders is explained below:

a-bbb-cc-dddd.ps1

a. Starts with 'c' for checks. Fixes (when implemented) will start with 'f'
b. Category label:
- **acc** : Accounts
- **com** : Compliance
- **drv** : Drives
- **net** : Network
- **reg** : Regional
- **sec** : Security
- **sys** : System
- **vmw** : Virtual

c. A unique zero-padded number per category
d. Short name of the check or fix

Example: **c-acc-01-local-users.ps1**

Since the compile script will search all subfolders under Checks you could create application specific folders for additional checks: Exchange, SQL, IIS, etc. This would help keep scripts organised and manageable. Just

make sure that every script has a unique number, regardless of their folder location – fixes should have the same number as the corresponding checks.

Engine

The **help.ps1** file holds the basic text information for the HTML reports. Any text entered in here will be shown to the engineer viewing the report. The format of this file is a simple XML layout.

Please keep to the same layout and format when adding or changing the text in these files.

The **main.ps1** file is the main engine script for the QA process. This script handles the entire process of running and collecting results from the checks and fixes. It is also responsible for generating the two output files for the servers (HTML and CSV).

Functions

This folder holds all the custom functions that are used by various checks. Putting them here makes the scripts more modular and helps with version changes. See **Appendix A** for more details on these functions.

Settings

The **default-settings.ini** file is the one that is customised for my specific environment. While this will work quite happily in your environment, you may have a more relaxed security and build configuration so most of the checks could fail.

Create a copy of the default file and customise it to suit your needs. Once done, compile a new QA script using your settings.

Editing Or Creating Scripts

This section of the documentation is for developers or people with a good working knowledge of Microsoft PowerShell and other scripting/programming languages.

> IMPORTANT: Since the lowest supported operating system is Windows Server 2008 R2 which ships by default with PowerShell version 2.0, all scripts must be written for this level of PowerShell. None of the newer commands can be used unless you have an equivalent command for version 2. An example of this is shown in **c-sys-05-system-event-log** where the PowerShell v4 method of collecting event logs is much quicker than the v2 method.

General Standards

When writing a new QA check, or editing an existing one, there are several guidelines that should be followed. These are standards that I have tried to follow and develop as I have written these scripts.

1. Every check must be tested on a clean Windows 2008 R2 and 2012 R2 server. The results should be identical between the two,

2. Everything must be wrapped in a Try/Catch statement, with the catch either being ignored or captured correctly, either as a script error, or part of the check,

3. There should be zero output from the script, all output is handled outside of each check,

4. One check per script. Do not put more than one check into a single script file. Exceptions are where you are checking multiple registry keys under the same root key. (See c-sec-01-schannel and c-sys-17-pagefile). An example of separation is the two checks for the event logs; one for System and one for Application,

5. Capitalisation is important in names and descriptions. It's just good grammar,

6. Mostly always the case, each check should be split into two sections:
   I. Try/Catch for getting the information required
   II. If/Else for working on the information

7. ...

Each and every check tries to follow the same standard and layout; this helps with debugging and organisation of them.

Layout Standards

If you compare all the checks, you will quickly see there are a number of set layout standards that have been used. This makes reading the scripts a bit easier. They might seem a little pedantic or restrictive, but this is to help future developers quickly see what is happening in the script.

- The first 10 lines must be a commented out description of what the check is designed to do. This includes a line for any required functions that are needed – See **Appendix A**.
- The function name (line 12) and `$result.check` value (line 21) must have identical names.
- The Function must start on line 12.
- Lines 14-21 are identical in all scripts, except for the values on 20 and 21.
  Line 20: The name of the check as shown in the HTML report.
  Line 21: This must match the name of the function (Line 12).
- Line 23 signifies the start of the function.
- Line 25 should always start with a `Try` command. There are a few exceptions however.
- Lines 26+ are up to you, following the various layout standards set in the existing scripts.
- The last two lines of the function must be **`Return $result`** and the final closing curly bracket '**`}`**'.

Creating New Checks Or Fixes

Whenever a new check or fix is required, it's a good idea to read though several of the existing scripts first to see how they are done. It is often quicker to copy an existing script and modify it than it is to start from scratch.

Check Methods

The following table will help you with identifying which script executes in which way. For example, if you want to check for a single registry key then use **X**. If you want to check for multiple registry keys under the same root key, script **Y** will do this, as well as many others.

Some checks may be in several methods.

Script	Checking Method
	Single registry key
Todo	Multiple registry keys (under same root key)
	Simple WMI query
	WMI query, looking for single entry
	WMI query, looking for multiple entries
	WMI query, excluding multiple entries from results
	Multiple checks depending on different results
	Execute a separate EXE and check for returned result
	Using external COM object
	Using custom Win32_Product function (see Appendix E)
	Manual Only Checks

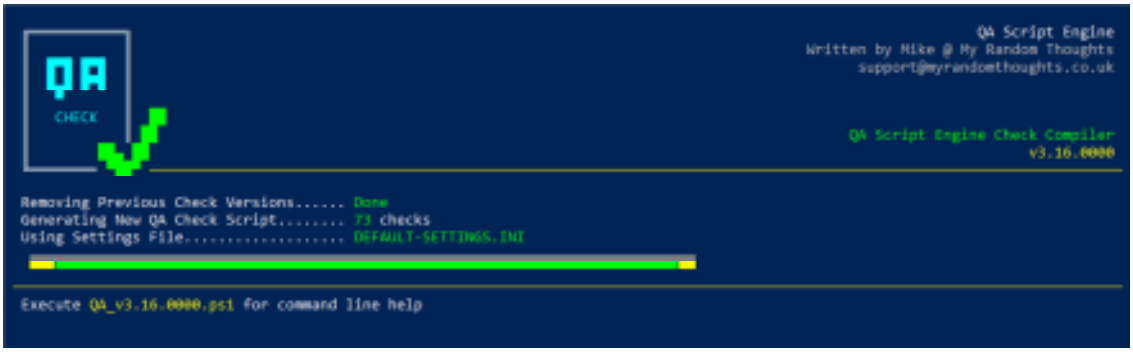
Compiling New QA.PS1 File

Once you have created a new check, or changed an existing one, you will need to compile the scripts into a new QA file for people to use.

In order to generate new master QA scripts, simply open a PowerShell window and execute the compiler script as you would any other PowerShell script.

```
.\compiler.ps1
.\complier.ps1 –Settings ‘myCustomSettings.ini’
```

The output of the compiler is shown below:



The green blocks are the checks being incorporated into the final script, the yellow blocks are the support files. You will notice that the version number will automatically change to the date when the script was compiled. This saves you changing the version yourself.

Appendix A – Custom Functions

There are a few custom functions used throughout the checks that help reduce the code duplication. They are stored here so that a change to any single function doesn't need to be replicated to lots of checks.

Calling Functions

Before you can use a function within a check, you need to make sure its referenced. You can do this by adding to the commented line at the top of the script called `REQUIRED-FUNCTIONS`. The line below shows an example.

```
REQUIRED-FUNCTIONS: Win32_Product, Test-Port
```

Function List

Win32_Product

I found that when running a WMI query against the Win32_Product class, it could take at least 10 minutes to return a value. It would also run verify checks against every single MSI file installed on the particular server. This is a known issue and reported widely on the internet.

Because of this, I created a function that took the name of the application that was being looked for, and instead of searching the WMI classes, it searches the two uninstall keys of the registry. If the application name is found, the version number is return. The two registry keys are:

```
HKLM\Software\Microsoft\Windows\CurrentVersion\Uninstall
HKLM\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninstall
```

Instead of taking 10 minutes, and potentially modifying the system, this function takes seconds at most, and changes nothing. Its usage is as follows:

```
[string]$verCheck = Win32_Product -serverName [server] -displayName [application]
```

This will populate the variable $verCheck with either a $null value, or the version of the application. If the application is found, but no version is given, a value of '0.1' is returned.

Check-VirtualMachine

This is a simple check to see if the server we are running a check against is a virtual machine or not. This same function is used when checks for physical servers only are needed:

It runs a quick WMI query to get the serial number of the remote server. If it returns a string containing 'VMware', then it's clearly a virtual server.

Check-HyperV

Similar to the above Virtual Machine function, however this is for Hyper-V guests. It's not currently used, but it may come in handy later.

Check-DomainController

This function checks to see if the remote server is a domain controller. This executes a WMI query and returns `$true` or `$false`.

Check-TerminalServer

As the Domain Controller check, this determines if Terminal Services is installed.

Check-NameSpace

This function is used to check that a specific WMI NameSpace exists before trying to query it. This helps reduce error messages and will help speed up processing time.

To call it, use the following example:

```
If ((Check-NameSpace -serverName $serverName -namespace 'MicrosoftIISv2') -eq $true)
```

The namespace "**MicrosoftIISv2**" is checked to see if it exists or not. This is a good way to check if IIS is installed as an application or not. Other examples could include "Cimv2\TerminalServices" as a sub-namespace