QA Checks
Developers Guide v4

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

It is assumed that you have a good working knowledge of PowerShell v4 and above and are familiar with the existing QA checks. A good understanding of various tools and utilities of getting information out of the Windows operating system via WMI, the registry or other means as required.

Please make sure you fully read and understand the document and the Getting Started guide, as well as the information posted in the Wiki location of the GitHub repository.

This document refers to version 4 of the scripts. It can be used for other versions however some of the screen shots and/or wording may be different.

Contents

Overview

The QA checks came about as a need to verify the build of new servers for our various customers and their environments. All new server builds are done with a custom gold image; however this image still lacks many of the additional tools and configuration settings needed before it can be accepted into support.

Most of this extra tooling and configuration should be automated, however checks are still needed to make sure each customer or environment has their specific settings in place.

Technical details

The scripts are written in the Microsoft PowerShell language, with version 4.0 in mind. Version 4 was used as a base as it is available on most modern Windows operating systems and as some significant enhancements over version 2 (as used in version 3 of these scripts).

It also requires WinRM to be configured in order to scan remote servers. WinRM can be configured either with or without SSL connections.

Supported operating systems

- Windows Server 2008 R2 — PowerShell 4 is not installed by default
- Windows Server 2012
- Windows Server 2012 R2
- Windows Server 2016

Unsupported operating systems

- Windows 2003 Server
- Windows Server 2008 R1
- Any non-server operating system
- Any non-Windows operating system

The Checks

There are over 100 different checks split over 10 separate sections. These are executed whenever the QA script is executed against a server and can usually take anywhere between 30 seconds and a couple of minutes to complete. If you are checking multiple servers then this time is per server. The script, by default, is set to run up to 5 checks concurrently.

Each check is written to be as efficient as possible; however due to the nature of some of them they may take a little longer than normal. With this in mind, each individual check has a timeout of 60 seconds. This should give them plenty of time to complete their task.

For a full list of checks and sections, check the GitHub Wiki pages:
https://github.com/My-Random-Thoughts/Server-QA-Checks-v4/wiki/Sections

Layout Overview

Folder Layout

The QA scripts are laid out in the following way:

```
\-- Server-QA-Checks
    \-- checks
        \-- accounts
        \-- compliance
        \-- …
    \-- documentation
    \-- engine
    \-- i18n
    \-- settings
```

Root Folder

The root folder contains the compiled QA scripts as well as the compiler and QA Settings Configuration Tool. Do not make any changes to the compiled scripts unless you are doing quick testing in order to fix an issue, they will be over written every time a compile is run.

Checks

The `checks` folder has several subfolders, one for each of the sections. Each of these subfolders contains the actual scripts.

Documentation

Any user or developer documentation lives here, as well as links to the GitHub Wiki sections

Engine

This folder holds separate function files that are required by the checks as well as the main script engine. The functions have been separated out in order increase code-reuse and maintaining version control. For a full list and description of each one see Appendix A. The main QA engine script does all the work of making sure the checks execute correctly and collates the results.

i18n

This folder is used for any language specific strings that are used throughout the script or tool. Instructions exist for how to translate the checks or the tool into various different languages, if you can help translate, please let me know.

Settings

This holds the `default-settings.ini` file as well as any customer or environment specific settings for your organisation. There can be any number of settings files located here.

Creating New Checks

Before creating a new check, first make sure what you want to do isn't already implemented. If you are duplicating a check, I may be able to help by modifying the existing one to help suit your needs. This is may not be possible in all cases, but it's worth asking!

Important

Version 3 of these scripts stipulated that PowerShell version 2 be used. This has been updated for these scripts and PowerShell version 4 can be used. However, since Windows 2008 R2 is still the lowest supported operating system, your checks need to be fully tested on this platform.

Script File Names

When creating a new check, the file name is important. Files should be named according to the following guide:

```
xxx-yy-zzzz.ps1
```

Where:

- `xxx` The three letter category label for the check,
- `yy` A unique zero-padded number that is next in sequence for the category,
- `zzzz` A short name for the check, all lowercase and any spaces replaced with hyphens (-).

Example:

```
acc-01-local-users.ps1
```

The compiler will automatically pick up any new scripts that are created.

General Standards

When writing a new QA check, there are several guidelines that should be followed. While some of these guides are not set in stone, they will help to keep all checks to a certain standard level.

1. Each and every check should be tested on a clean Windows 2008 R2 server with PowerShell v4 installed. Additional checks on other Windows operating systems must produce the same pass/fail results (where applicable),

2. Almost everything must be wrapped in a Try/Catch block. The Catch can either be ignored or captured correctly as a script error, or part of the check as needed.

3. There should be no screen or code output from the check. All output must be suppressed and only a fully formed PSObject (`$results`) should be returned. Use `[void]` instead of `Out-Null`. The script must always return a fully formed PSObject result regardless of pass, fail or a Try/Catch issue.

4. Each check should perform one function only. An example of this is the two event log checks. One for the Application log and one for the System log. There is the odd exception like `net-07` where is checks several items, but all related to the check itself. Another is `sec-18`.

Details on the returned PSObject format are explained later in this document.

Layout Standards

Each and every check tries to follow the same standard layout; this helps with debugging and organisation. The standard layout helps with reading each check to quickly see what it does and how it does it.

Comment Header

Every check has the same comment header block at the very top. This must be included in your check in order for the compiler and QASCT to pick up the required information for the various parts of the script.

Open any of the existing checks and read through some of the headers. You should notice a pattern in the layout and formatting. Try to keep this going forward in your checks.

Description

This gives the full description of the check, explaining its purpose and any other information that might help the end user running the script.

Required-Inputs

This lists any inputs that the check is looking for, along with the descriptions for each input. Validation rules can also be added for when the GUI configurator is being used. The format for these entries is:

```
{name} - "{type/options}" - {description}|{validation}
```

The `name` and `description` are mandatory, the rest is optional. See Appendix B for details

Default-Values

The default values of all the check inputs specified above. Single quotes must be used.

```
{name} = ''
{name} = @('', '')
```

Default-State

If your check should be enabled by default, set this value to "Enabled". Any other value will mark it to be skipped. This is used when the settings INI files have no record of this check.

Input-Description

This allows checkboxes and/or drop-down lists to have short description for each value. For example: an input value may just be '0' or '1'. This might not mean much, so descriptions can be added. See Appendix D for example screen shots. `net-13` has an example of this.

Results

A list of the messages returned from your check for each of the result states (pass, warning, fail, etc.). These are used in the HTML hover help display.

Applies

The type of servers this check applies to. The current values are listed below. Used in the HTML hover help display and the QA Settings Configuration tool

- `ALL` All Servers
- `VMW` Virtual Servers
- `PHY` Physical Serves
- `HVH` Hyper-V Hosts
- `DCS` Domain Controllers
- `RDP` Terminal Servers

Required-Functions

A list of any required functions that your script relies on. Available functions are listed in Appendix A. If your check requires a function not listed, simply add it to the bottom of your script and set this to "None"

Copying Existing Scripts

The quickest way of getting up and running with a new check is to copy an existing one and changing its details. Several scripts are very similar in what they do and could be adapted to suit your needs.

For example, several checks look for a single registry key and value. Some look for a particular WMI setting. These can be used as templates for new checks.

Testing Your Script

Once you have created your script and want to test its output, you can do this in one of two ways.

1. Run the script as a separate standalone check,
2. Compile the entire QA checks into one QA script.

Option 1

This is the quicker option to perform but does require a small change to your check that must be removed before using as a final version.

Add the following code to the very top of your script:

```
1.  Function newResult { Return ( New-Object -TypeName PSObject -Property @{'server='';
    'name='''; 'check='''; 'datetime'=(Get-Date -Format 'yyyy-MM-dd HH:mm');
    'result'='Unknown'; 'message='''; 'data='''; blob=''} ) }
2.  $script:lang      = @{}
3.  $script:chkValues = @{}

4.  $script:chkValues ['xxx'] = ('')
5.  $script:chkValues ['xxx'] = ('')

6.  $script:lang['Error']          = 'Error'
7.  $script:lang['Fail']           = 'Fail'
8.  $script:lang['Manual']         = 'Manual'
9.  $script:lang['Not-Applicable'] = 'N/A'
10. $script:lang['Pass']           = 'Pass'
11. $script:lang['Script-Error']   = 'SCRIPT ERROR'
12. $script:lang['Warning']        = 'Warning'
```

Change lines 4 and 5 to suit your needs. It should contain any settings variables that you may require in your script. Add as many as needed. Next, add the following code the very end of your check, after the closing bracket '`}`' of the function:

```
xxx-yy-zzzz
```

Where `xxx-yy-zzzz` is the name of your script function

Option 2

This option doesn't require any extra code, but is slower as you need to compile the entire QA script and wait for it to execute.

To do this, simply follow the instructions in Appendix D for compiling the scripts

Returned PSObject Result

Every check returns the same result object regardless of outcome or errors. This helps to make sure all the results are captured correctly. The format is as follows:

server	Name of the server this check is being executed on
name	The language independent name of this check
check	The internal name of this check
datetime	The date and time this check was executed (yyyy-MM-dd HH:mm)
result	Pass/Warning/Fail/Manual/NA
message	Result message
data	Check data that may be relevant to the result
blob	Special data used to generate extra report files (exported event logs, for example)

With your script, the result object is created and partially filled in at the top of the function:

```
$result        = newResult
$result.server = $env:ComputerName
$result.name   = $script:lang['Name']
$result.check  = 'acc-01-local-users'
```

The first three lines will be the same for every single check script. The last of the four will change to show the internal name of the particular script.

As your script progresses and the results come in from various checks you are performing, the `result`, `message` and `data` entries will be filled in.

Result

The `result` entry will be set to one of the following results

```
$script:lang['Pass']
$script:lang['Warning']
$script:lang['Fail']
$script:lang['Manual']
$script:lang['NotApplicable']
```

This will ensure that the result is set to the correct language specific string.

Message / Data

The `message` and `data` entries can be set to basic text while you are developing your check and later converted into language neutral entries. If you require a newline character in your strings, use a comma followed by a hash (`,#`) in the text. For example from `acc-02`, the `data` field has the following string:

```
Administrator: {0},#Guest: {1}
```

This will display as two lines in the HTML report.

Message strings are generally short descriptions on why the check got the result it did.

Data strings are made up of any data that is generated from the check. For example, when checking drive space on all drives, why not output all the drives and available space to the user, instead of just saying “Pass” or “Fail”. Give the user as much information as you can in the `data` field.

Blob

The `blob` entry is a special one that is made up of another object. Checks `sys-05` and `sys-06` are good examples of its usage. It allows data to be returned from the check to the engine and saved as a file. In the above two checks, exported event log data is returned and saved in the results folder.

The code from `sys-05` is shown below

```
    $blob = (New-Object -TypeName PSObject -Property @{filename=''; subpath=''; type ='';
data='';} )
    $blob.filename = "$($script:lang['dt00'])_Event-Log.csv"
    $blob.subpath  = 'Event-Logs'
    $blob.type     = 'CSV'
    $blob.data     = ($gWINe | Sort-Object -Property 'TimeCreated')
    $result.blob = $blob
```

(In this example, `$gWINe` is a `[System.Collections.ArrayList]` created from the `Get-WinEvent` cmdlet.)

First, we create a `$blob` object with the correct fields, as follows:

filename	The name of the file to create
subpath	The subfolder to store the file created. Will be created under the results folder
type	Either "`CSV`" or "`TXT`" format currently
data	The object to export

When using the `CSV` type, the data object is saved using the `Export-CSV` cmdlet with the `-NoTypeInformation` flag set. Any other value will be saved with the `Out-File` cmdlet with a utf8 encoding. More output formats may be added later if it's requested.

Once all the fields have been completed, we set the `$result.blob` value to `$blob`.

This is a rarely used return value; there are currently only three checks that need to return data as files.

Appendix A - Custom Required Functions

There are several functions that are used by various checks, and have been separated out to help reduce bugs. They are listed below...

Check-IsDomainController

This function returns `$true` or `$false` if the server is a domain controller or not. This is used to determine if local groups should exist or not (c-acc-04-local-groups) or if specific software should be installed (c-sec-13-rsa-authentication).

Check-IsHyperVGuest

This function checks if the server being checked is a Microsoft Hyper-V Host server. This is used for all Hyper-V-Host (hvh-) checks. This returns a `$true` or `$false` value.

Check-IsPortOpen

This function tries to open a connection to a server name on a given port. This returns a `$true` or `$false` value.

Check-IsTerminalServer

This function checks to see if the Terminal Services namespace exists on the server you are checking. It automatically includes the above `Check-NameSpace` function (the only duplication). This returns a `$true` or `$false` value.

Check-IsVMwareGuest

This function determines if the server is a VMware ESX(i) guest by looking at the serial number value of the server. This returns a `$true` or `$false` value.

Check-NameSpace

This checks looks to see if a specific WMI namespace exists and returns `$true` or `$false`. This is useful to use before calling a custom namespace or one that may not exist on a particular operating system version.

Check-Software

This function searches two specific registry keys for the given software name and returns the version number. If no version number exists for the software it returns a version of '0.1'. If the software is not found, then a `$null` value is returned.

Appendix B - Input Types And Validation

Adding comments to the Required-Inputs comment section at the top of your check helps the GUI Configurator inform the user what this particular input is required for. There are some additional controls that can be added to the comments to help shape the input given. These are explained below:

Input Valuation

Most input options will just be a simple string value and therefore will not require any special control codes. Simply add a comment to describe what this input is for. The GUI Configurator will present a single textbox that can be used to enter the details.

Adding one of the following to the end of a comment will provide the input validation shown...

`|AZ`	Letters A-Z only (case insensitive)
`|Numeric`	Numerical validation (integer numbers, as well as floating)
`|Integer`	Integer (whole) only number validation (1, 42, 538)
`|Decimal`	Decimal (floating) number validation (1.23, 42.743, 538.293)
`|Symbol`	Symbols validation (Anything except A-Z or 0-9)
`|File`	File or folder name validation
`|URL`	Web address validation, protocols include http, https, ftp, sftp or ftps
`|Email`	Email address validation
`|IPv4`	IPv4 address format validation
`|IPv6`	IPv6 address format validation

Examples

No validation

```
Enter your name
```

IPv4 address validation

```
DNS IP addresses that you want to check|IPv4
```

Number validation (integers only)

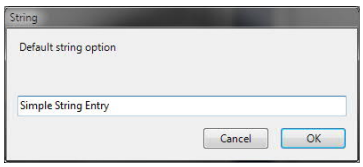
```
Enter the number of days|Integer
```

Input Types

When asking for users to enter data, there are a variety of different input options that can be used. Each one has its own comment control code.

Simple

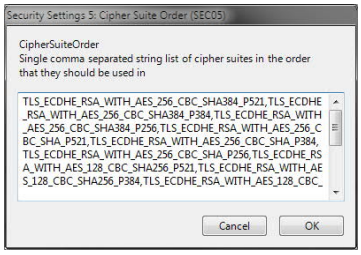
Without any control code or validation, this is the default input box that a user will see



Large

This is similar to the Simple input control; however this shows a larger input window. This allows for longer strings of text to be entered. This type ignores all line breaks and returns the entered text as one line.
Note: Input validation is ignored for this type.

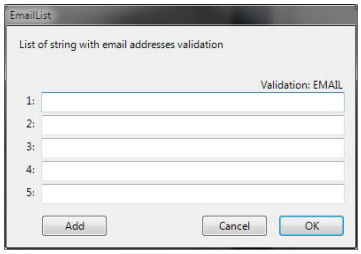
```
"LARGE" – Enter a long single line of text
```



List Of Strings

To get the user to enter one or more items as a list, add “LIST” to the start of the comment, separates with a hyphen (-). The will show 5 input boxes (with optional validation) with which the user can enter the requested values. A maximum of 30 boxes can be shown. This input form checks for and rejects duplicates.

```
"LIST" – Email address to use|Email
"LIST" – Enter all DNS servers to check|IPv4
```

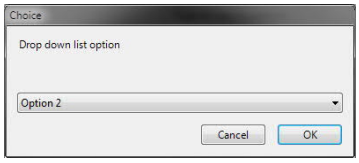


Dropdown List

To have the user select one option from a list of options, add a pipe (|) separated list of items to the front of the comment. Must be separated from the comment with a hyphen (-). See Appendix C for details on providing more information to the user.

```
"High|Normal|Low" – Email priority
"True|False" – Send email using SSL
"Option 1|Option 2|Option 3|Option 4|…" – Please select an option
```

As many options as required can be listed.

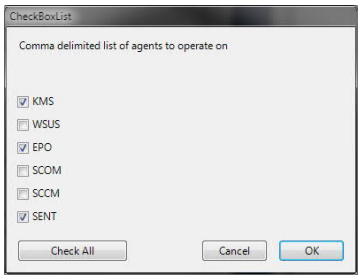


Checkbox List

If more than one option is required, check boxes can be used to choose one or more of the available items. To use, add a comma (,) separated list of items to the front of the comment. Must be separated from the comment with a hyphen (-). See Appendix C for details on providing more information to the user.

```
"KMS,WSUS,EPO,SCOM,SCCM,SENT" – Select one or more items to install
"Option 1, Option 2, Option 3, Option 4" – Select all that apply
```

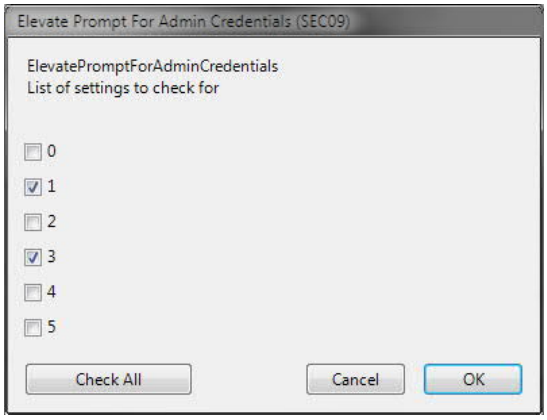
Try to keep the selection list to a maximum of about 10-15 items.



Appendix C - Input Description

When entering data into the GUI front end, there are times when a list of options is presented, but they hold very little information. This optional script header item can be used to add short descriptions to these values.

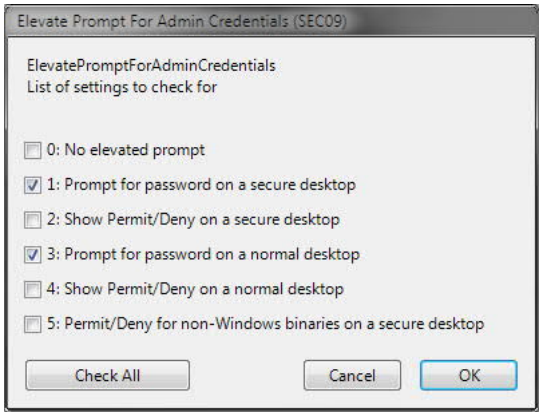
For example, the check `sec-09` asks for a list of options to choose from when a user is requesting elevated rights. The check looks for one or more of the input values that are in the range 0 to 5. This would mean very little to most people when they are configuring their environment settings.



To add a description to one or more of these items, simply add to your scripts comment header the following information:

```
INPUT-DESCRIPTION:
    0: No elevated prompt
    1: Prompt for password on a secure desktop
    2: Show Permit/Deny on a secure desktop
    3: Prompt for password on a normal desktop
    4: Show Permit/Deny on a normal desktop
    5: Permit/Deny for non-Windows binaries on a secure desktop
```

The format is `{entry}: {description}`, where `entry` is the value that is being asked for.
Once you have added this, the following screen will be shown:



These descriptions can be added to checkbox selections (as above) or drop down list selections as `sec-15` shows.

Appendix D - Recompiling the QA scripts

Whenever a change is made to the settings file or any of the individual checks, you will need to recompile the QA script.

The reason for compiling into a single file is to make the completed script portable without having 100 separate files all over, potentially being of different versions.

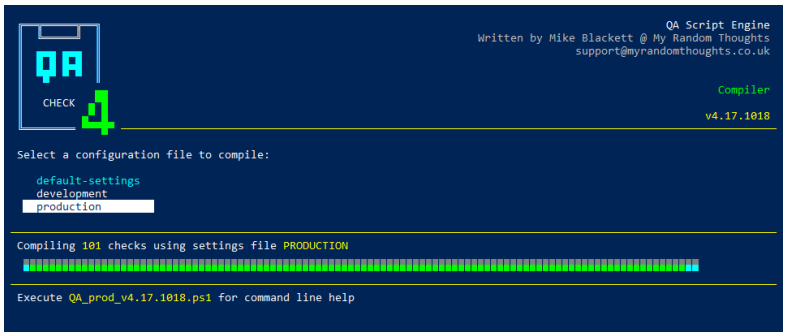
To start, open a normal PowerShell window and change to the folder containing all the script files.
Typing the following command will show a list of possible configuration files. If only the default-settings file is found, it will automatically compile that one.

```
.\compiler.ps1
```

To tell the compile script to use a specific settings files type:

```
.\comiler.ps1 –Settings {name-of-file}
```

The screen shot below shows the completed compile process for a selected settings file called production.



As you can see, the QA script filename contains the short code that is used for this settings file.