PROXY Administration with PowerShell

This guide provides information about using the Microsoft Windows PowerShell scripting platform to automate administrative tasks to be performed on the PROXY Pro Gateway Server and PROYX Pro Host products.

This document last updated on Tuesday, May 10, 2016 for Proxy v9.0.1 hotfix#1. Information is applicable to PROXY v8.0 through PROXY v9.0.

Table of Contents

Introduction

Administrators often request ways to automate many of the routine tasks they perform in managing a Proxy Gateway Server -- e.g. moving Hosts between groups, adding or removing security permissions for users, etc.

These tasks are usually performed by hand using the Proxy Gateway Administrator MMC snap-in (GWA). The GWA, in turn, performs its actions by making calls to the Proxy SDK runtime controls. These controls are installed by default on any system on which any other Proxy software component is installed, e.g. Gateway, Host, or Master.

Prior to Proxy v8.0, these controls were not accessible via scripting due to license restrictions. In Proxy v8.0 and later, both the Proxy Gateway control (PrxGW) and the Host Administration control (PrxHA) are now accessible via scripting. As a result, it is now possible to automate any formerly manual administrative process by assembling the right sequence of commands into a PowerShell script.

And by adding event handlers to your scripts, it is now possible to augment the built-in behavior of the Gateway Server with a customized, fully-autonomous Gateway management process.

The Proxy PowerShell Scripting package contains a set of PowerShell modules comprising a set of powerful yet easy-to-use routines you can call from your own PowerShell scripts, or directly from the PowerShell console. These routines simplify the process of making Proxy SDK calls to perform low level actions like moving Hosts between groups or modifying Gateway security settings.

Once you have loaded the Proxy PowerShell modules into a PowerShell command console, you're ready to start calling the high level routines from your PowerShell scripts, or directly from the command line. You also will have access to the complete low-level object model presented by the Proxy SDK.

This document gives a quick introduction to the prerequisites, the process of loading the Proxy PowerShell modules, and some common usage patterns. It also provides an overview of the modules provided in this package (that can be used as-is) and the sample scripts also provided (which you modify, and use as the basis for building more complex scripts).

Quick Start

If you're impatient to get started, or are comfortable with PowerShell and want to dive right in, follow these steps *exactly* to get going. The explanation & rationale for these steps is explained in the sections below – if you want to know why these are the right steps, or run into trouble and need to diagnose what went wrong, please read the later sections *very carefully* to get the details.

1. Copy the contents of distribution ZIP file, "Modules" directory, to a directory named "%USERPROFILE%\Documents\WindowsPowerShell\Modules". Create that directory if it doesn't already exist.
2. Copy the contents of the distribution ZIP file, "Scripts" directory, to your desktop, keeping them in a folder named "Scripts".
3. Create a shortcut on your desktop to launch PowerShell with the correct configuration. On Windows x64 editions, this command line is

```
C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe
-executionpolicy remotesigned -mta
```

   On Windows x86 editions, this command line is:

```
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
-executionpolicy remotesigned -mta
```

   Set the "Startup Directory" of the shortcut to be "%USERPROFILE%\Desktop\Scripts".
4. Launch the new shortcut to open the PowerShell environment. If it current directory is *not* your desktop "Scripts" directory, change to that directory.
5. Enter the command ".\ReportGatewayInfo.ps1 -gateway <dns-name-of-gateway>", filling in the DNS name of your Gateway Server.
   a. This script assumes the Gateway Server accepts connections on TCP protocol port 2303; to specify a different protocol and port, specify the "-protocol" option with the protocol and port number separated by a vertical bar, e.g. "-protocol "SSL|8443"".
   b. This script uses the logged-in user identity by default; to specify alternate credentials for authentication, specify the "-credentials" option with the username and password separated by a vertical bar, e.g. "-credentials "domain\administrator|P@ssword"".

Common Error Scenarios

Error Message: STA Apartment mode

```
WARNING: You are running in STA Apartment mode.
This occurs in the PowerShell ISE, or by default in PowerShell v3 and later.
The Gateway Client's COM object will not function properly when running in
the STA. Specifically, if the network connection drops or fails, that will
not be reflected in the Gateway object state.
```

To resolve this issue, run the PowerShell environment in the COM MTA model. This can be specified on the PowerShell command line with the "-mta" switch.

Error Message: 80040154 Class not registered

```
New-Object : Retrieving the COM class factory for component with CLSID
{D1290A58-156A-11D4-8356-00105A6BC646} failed
due to the following error: 80040154 Class not registered (Exception from
HRESULT: 0x80040154 (REGDB_E_CLASSNOTREG)).
```

This error indicates either that the PROXY Pro SDK Runtime controls are not installed on this machine, *or* that you're running the 64-bit version of PowerShell. The PROXY SDK controls

are 32-bit, in-process COM objects, and must be used from the x86 version of PowerShell. Be certain to run PowerShell from the "C:\Windows\SysWOW64\WindowsPowerShell\v1.0\" directory.

Error Message: Import-Module cannot load

```
import-module : File
C:\Users\Administrator\Documents\WindowsPowerShell\Modules\proxy-
hostadmin\proxy-hostadmin.psm1 cannot
be loaded because running scripts is disabled on this system. For more
information, see about_Execution_Policies at
http://go.microsoft.com/fwlink/?LinkID=135170.
```

This error indicates that the execution policy prevents the script from running. Follow the provided link for more information. Use the "set-executionpolicy" cmdlet as discussed in the "Prerequisites" section of this document to enable script execution.

Prerequisites

Using any of these scripts requires:
- Microsoft Windows PowerShell v2.0, v3.0, v4.0, or v5.0, x86 edition. Note well that on a Windows x64 operating system, you must use the x86 version of the PowerShell scripting environment, because the Proxy components are COM components that are 32-bit only.
- Proxy SDK Runtime v8.0 or later installed. Note that these runtime components are installed along with the Host, Master, Gateway Server, and Deployment Tool product components, so if any of these items is installed on the computer, the Proxy SDK Runtime is there too.

PowerShell v3.0 is available from Microsoft in the Windows Management Framework 3.0. This is available for Windows 7 and Server 2008 R2 at http://www.microsoft.com/en-us/download/details.aspx?id=34595.

PowerShell v4.0 is available from Microsoft in the Windows Management Framework 4.0. This is included in Windows 8.1 and Server 2012 R2, and is available for Windows 7, Server 2008 R2, and Server 2012 at http://www.microsoft.com/en-us/download/details.aspx?id=40855.

PowerShell v5.0 is available from Microsoft in the Windows Management Framework 5.0. This is included in Windows 10 and Server 2016, and is available for Windows 7, Server 2008 R2, Windows 8.1, Server 2012, and Server 2012 R2 at https://www.microsoft.com/en-us/download/details.aspx?id=50395.

To confirm what version of PowerShell you have:
1. Open the "Windows PowerShell (x86)" command prompt.
2. Issue the command "$Host.Version".
3. The output is the version of the PowerShell environment. The important number is the "Major" version number, which corresponds to the PowerShell major version (2 through 5).

To make the Proxy administration modules available to PowerShell, they either need to be copied to a directory on the PowerShell path, or placed in a location that is added to the path. To find out what the current PowerShell path is, issue the command "`$ENV:PSModulePath`". On Windows 7, this typically includes a per-user directory like "c:\users*username*\Documents\WindowsPowerShell\Modules"; placing the modules in this location is recommended. Alternately, you can add any directory to this path with a command like "$ENV:PSModulePath = "c:\Proxy\Powershell;" + $ENV:PSModulePath".

Finally, the default execution policy is to not allow scripts to run in general. Please see "`get-help about_execution_policies`" and "`get-help about_signing`" for more details about execution policy and code signing.

For initial testing and development, we recommend changing the execution policy to RemoteSigned. This allows local scripts to run, but provides protection against (potentially) malicious scripts from the internet. The command to set this for just the current PowerShell command prompt window is "`Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope Process -Force`".

Finally, be certain to read the next section, "PowerShell Threading Models", for important information about starting PowerShell v3 or later in the MTA threading model.

PowerShell Threading Models

Because the Proxy SDK components are COM objects, their operation is dependent on the COM threading model of the process that they're used in. Proxy supports both the single threaded apartment model (STA) as well as the multi-threaded apartment model (MTA). However, PowerShell implements the STA in a non-standard way, and use of the MTA is **strongly recommended**.

In PowerShell v2.0, the PowerShell command prompt defaults to the MTA model, and the SDK controls work reliably here. The Integrated Scripting Environment (ISE) uses the STA model, but does not consistently/promptly process Windows messages. This leads to Proxy components failing to recognize and process events like disconnect, and therefore this mode is less reliable than the MTA model.

In PowerShell v3.0 and later, both the PowerShell command prompt and the ISE default to the STA model for consistency. Because PowerShell doesn't consistently/promptly process Windows message in this mode, like PowerShell v2.0, this is less reliable than the MTA model.

PowerShell v3.0 or later command prompt should be started in MTA mode using the command line switch "-mta". You can set this by creating a shortcut with a command line like:
>   *path*\powershell.exe -executionpolicy remotesigned -mta
where '*path*' is the correct path to the PowerShell executable, e.g. "%SystemRoot%\system32\WindowsPowerShell\v1.0" on Windows x86 or "%SystemRoot%\syswow64\WindowsPowerShell\v1.0" on Windows x64.

References

PowerShell is a Microsoft technology in the Windows platform, and Proxy Networks cannot answer questions about generic PowerShell scripting. For information about PowerShell, please see the following Microsoft references:

Windows PowerShell (on MSDN)
> https://msdn.microsoft.com/en-us/library/dd835506(v=vs.85).aspx

Writing a Windows PowerShell Module (on MSDN)
> https://msdn.microsoft.com/en-us/library/dd878310(v=vs.85).aspx

PowerShell's Security Guiding Principles (Blog post by the PowerShell Team)
> http://blogs.msdn.com/b/powershell/archive/2008/09/30/powershell-s-security-guiding-principles.aspx

Getting Started

To see the Proxy PowerShell Scripting at work, execute the following sequence of commands after following the instructions in "Prerequesites".

```
PS> Import-Module Proxy-Gateway
PS> Connect-Gateway 'gatewayspecifier'
PS> $gws = Get-GatewayObject
PS> $gws.getHosts() | where { $_.machine -ne 'virtual' }
PS> Disconnect-Gateway
PS> $gws = $null
```

These steps do the following, in order:
1. Import the "Proxy-Gateway\Proxy-Gateway.psm1" module contents into this PowerShell command prompt.
2. Connect to the specified Gateway Server. The arguments for "`Connect-Gateway`" are:
   a. "`-station stationname`" to specify the Gateway to connect to. This can be a machine name, DNS name, or IP address. This is the default argument.
   b. "`-protocol protocolname`" to specify the protocol, either "TCP", "UDP", or "SSL".
   c. "`-credentials credentialsstring`" to specify explicit credentials (instead of using the logged-in username). This string must be quoted, and is in the form "*domain\username|password*".
3. Retrieve the connected Gateway object into the local variable `$gws`.
4. Call the `getHosts()` function on the Gateway object, and filter the results to eliminate items where the "`machine`" property is the value "virtual".
5. Disconnect the connected Gateway object. This invalidates, but does not clear, the local variable `$gws`.
6. Clear the local variable `$gws`.

Note very well that the design of the Proxy PowerShell Scripting module is such that there can be only one connection to a Proxy Gateway Server per instance of PowerShell. In order to connect to a different Gateway Server, the current connection must be disconnected first (using the `Disconnect-Gateway` cmdlet).

This is also true of the Host Administration connection, available in the Proxy-HostAdmin module.

Proxy PowerShell Script Modules

The Proxy PowerShell Scripting package includes a number of modules to provide access to Proxy features. These modules should be placed in one of the directories from the $ENV:PSModulePath path, or a new directory should be added to that path, in order to locate the modules.

The modules included in this release of the package are as follows. Each module is in a directory with the module name, and the file is the module name with extension ".PSM1".

Module	Description
Proxy-Gateway	Main functionality for accessing Gateway Server. Creates and manages Gateway connection, subscribes to Gateway events, and includes example event handlers.
Proxy-HostAdmin	Administration of Host Settings via Gateway. Includes commands to connect to the Host Administration object via a Gateway.
Proxy-Constants	Defines security constants. Should be imported to have definitions of the specific security rights in Proxy-GatewaySec and other security modules.
Proxy-GatewaySec	Security-related cmdlets. Find specific user rights across all Gateway objects. Remove user from Gateway security. Includes functions to work with different objects to add and remove user rights.
Proxy-Utils	Defines the "Wait-EventWithMessageLoop" function, which is required to wait for an event when running in the COM STA. This module is automatically included by other modules that need it.
Proxy-Gateway-Utils	Helper functions. This module is automatically included by other modules that need it.
Proxy-Gateway-ADs	Helper functions for accessing Active Directory Services. This module is automatically included by other modules that need it.
Proxy-ConnectionEvents	Defines constants used within the PowerShell scripts. This module is automatically included by other modules that need it.
Proxy-ObjectSec	Helper functions for manipulating security objects. These functions work with all Proxy components (Host and Gateway). This module is automatically included by other modules that need it.

Proxy Sample PowerShell Scripts

In addition to the modules that provide subroutine library functionality, and which can be used as-is, this package includes several sample scripts designed to be modified to meet your needs.

The scripts included in this release of the package are as follows. Each script is in the "Scripts" directory, with extension ".PS1".

Script	Description
AddGatewayToHost	Script to add a Gateway configuration to a Host that's already connected-to for administration. Use with RemoveGatewayFromHost and ConfigureAllHostsOnGateway scripts.
Add-NewGWSUser	Script for adding access rights for a new user to the Gateway security model. This script should be changed to accommodate specific organizational policies and processes.
ConfigureAllHostGroupingViaRules	Script to manage Host group memberships based on AD groups. A CSV file lists AD organizational units and the corresponding Gateway-based groups machines in those OUs should belong to. The script enumerates machines in AD, locates them in the Gateway, and adjusts their group memberships appropriately. Note that a CSV file for your local environment must be created; the script does not need to be modified.
ConfigureAllHostsOnGateway	Script to enumerate all Hosts managed by a Gateway, connect to each one for Host administration, and the update the Gateway configurations at that Host. Uses AddGatewayToHost and RemoveGatewayToHost scripts. Note that this script must be edited before use to specify the desired changes.
Delete-MatchingGatewayGroups	Script to find Gateway groups that fit a pattern, and delete them. This script was developed by a customer that had originally used a Host Grouping Rule with the default "OU=" prefix, then decided to change that, so new groups with the new naming convention were created. The script was used to delete the no-longer-needed "OU=*" groups.
Get-HostSecurity	Script to print a Host's security settings, including the Service, Settings, and Administration security descriptors.
Get-HostSettings	Script to read many of the Host settings and return name/value pairs of information as a hash table or PSObject.
Get-HostSettingsToXML	Script to read many of the Host settings and return an XML document containing them.
Invoke-HostTask	Script to invoke a PowerShell script block against a connected PrxHA control instance for the specified Host.

orgHostsByOU	Script to organize Hosts in a Gateway by looking up the Host machines in Active Directory, and assigning them to groups named for the Organizational Unit that they belong to.
RemoveGatewayFromHost	Script to remove a Gateway configuration from a Host that's already connected-to for administration. Use with AddGatewayToHost and ConfigureAllHostsOnGateway scripts.
ReportAllHostsOnGateway	Script to enumerate all Hosts managed by a Gateway, connect to each one for Host administration, and report whether the Host is configured for simple password, and enumerate all of the Gateway configurations in the Host settings.
ReportGatewayInfo	Script to count different objects in the Gateway Server and report the counters. This is a very simple script and is a good way to test that the PowerShell environment is working.
Set-DefaultHostSettings	Script to reset many Host settings to their installation defaults.
Set-Owner	Script defining a function to take ownership of a security descriptor.
taskWatchGatewayEvents	Script demonstrates how to run in the background and monitor for specific Gateway event notifications.

Test, Get and Disconnect existing Gateway connection

Import the module from script, not knowing if it's already loaded

```
if (-not (Get-Module Proxy-Gateway))
{
      Import-Module Proxy-Gateway
}
```

Test if Proxy-Gateway module is connected to a Gateway Server

```
Test-Connection
```

This cmdlet returns $true if a connection exists and is connected, or $false otherwise.

Retrieve the COM object that is the root of the Proxy object model

```
$gws = Get-GatewayObject
```

This cmdlet returns the COM object "ProxyGWLib.ProxyGWClass", which is the root object of the Proxy PrxGW Gateway Administration object model.

Disconnect the current connection to the Gateway Server

```
Disconnect-Gateway
```

Working with Groups

Proxy-Gateway-Utils module has some utility functions to aid in Group membership management. Using them, administrator can schedule regular Group membership maintenance, for example copying and deleting Hosts from Groups.

Copy-HostToGroupsByPolicy cmdlet

Iterates through all Managed and Unmanaged Hosts on the Gateway and calls $policy for each Host with Host Object as a parameter. Next, it gets list of Group's Names returned from the $policy function and makes sure this Host belongs to all those Groups, making it a Managed Host first (if the list is not empty, and the Host is not already Managed).

Parameters:
- $prxGWS : Gateway Object
- $policy : Script block which receive ProxyGWHost Object as a parameter and returns a list of Group Names where this Host belongs
- $forceGroupCreation : Force creation of Groups if they doesn't exists on the Gateway

Example: The sample script "Copy-Hosts-Example.ps1" creates Gateway Groups with the names of the Active Directory Organizational Units that the Host machines belong to, and adds the Hosts to those groups.

Remove-HostFromGroupsByPolicy cmdlet

Removes Host from groups by given policy (works the same as Copy-HostToGroupsByPolicy but removing instead of copying).

Parameters:
- $prxGWS : Gateway Object
- $policy : Script block which receive ProxyGWHost Object as a parameter and returns a list of Group Names from where this Host should be removed

Sync-HostInGroupsByPolicy cmdlet

Iterates through all Managed and Unmanaged Hosts on the Gateway and calls $policy for each Host with Host Object as a parameter. Next, it gets list of Group's Names returned from the $policy and makes sure this Host belongs to all those Groups which are in the list, and doesn't belongs to the groups which are not in the list but exists on the Gateway. This effectively does the "Remove" operation (above) to remove the Host from groups other than the Group with the same name as the Organization Unit the Host machine belongs to, and the "Copy" operation (above) to put the Host into the right Group.

Parameters:
- $prxGWS : Gateway Object

- $policy : Script block which receive ProxyGWHost Object as a parameter and returns a list of Group Names where this Host belongs
- $forceGroupCreation : Force creation of Groups if they doesn't exists on the Gateway

Example: The sample script "Sync-Hosts-Example.ps1" creates Gateway Groups with the names of the Active Directory Organization Units that the Host machines belong to, and ensures that each Host belongs to only the one group with the name of the OU that it belongs to. Note that this is similar to the "Copy-Hosts-Example.ps1" script, but that script merely ensures the Host belongs to the right Group, without removing it from any other group(s) it may belong to. This script ensures the Host belongs to the right Group, and also ensures it does not belong to any other groups (except "All Hosts").

Proxy Gateway events: examples and templates

Proxy Gateway Server is using .NET wrapped COM events to notify about changes in different collections, like Group Membership, Unmanaged Hosts, Workstations etc. (ProxySDK documentation has a list of all Gateway events in 'ProxyGW Events' section.)

To subscribe for events in the PowerShell script, you need to enable notifications for the specific event group (because they're off by default), and also to register an action for the event. For example, the following script lines associate the script block $action with changes to the Unmanaged Hosts collection:

```
# Enable Group membership notifications on the Gateway Server
$gws.enableChangeNotifications([ProxyGWLib.GatewayCollection]::colle
    ctUnmanagedHosts, -1, "", -1)
# Enable group membership event's handling in script
$newEvent = Register-ObjectEvent $gws onUnmanagedHostDataChanged -
    Action $action
```

There are several helpers to register actions for the most common events along with the Action's examples and templates. Event registration helpers:
- Register-GroupMembershipChange - Group membership change notifications. (Fired when Host is moving from group to group except "Unmanaged" and "All Hosts" which are not real groups). Template Action: $template_event_group_membership_action
- Register-UnmanagedHostChange - Unmanaged Host collection change notifications. (Fired when new host is added to "Unmanaged" Hosts). Template Action: $template_event_unmanaged_action. Example Action: $example_event_unmanaged_action.
- Register-WorkstationsDataChange - Property of a workstation-type Host has changed notification. (Can be used to find when new Host is added to "Managed" Hosts). Template Action: $template_event_workstation_data_action. Example Action: $copy_new_workstation_to_OUs_action

Examples:
```
Register-UnmanagedHostChange $example_event_unmanaged_action
Register-WorkstationsDataChange $copy_new_workstation_to_OUs_action
```

Security management scripts and modules

The Proxy object model provides full access to the security configuration in the Gateway (restricted by access rights – only authorized users can read or change the security settings). Scripting security policies and changes can avoid errors that may be made when security is edited "by hand".

To use the security management modules, you need to "Import-Module" both "Proxy-Constants" and "Proxy-GatewaySec". Having done that, there are some powerful cmdlets available.

Find-AllUserACEs cmdlet

List everyplace the specified user appears across all securable objects in the Gateway Server

Parameters:
- -prxGWS
- -domain
- -name

Example:
```
Find-AllUserACEs -prxGWS $gws -Domain "boston.com" -Name
    "ivan_drinkwater"
```

Remove-AllUserACEs cmdlet

Remove the specified user from all securable objects in the Gateway Server. Note well that this only removes entries (ACEs) that name the specified account explicitly. The user account may retain some access to the Gateway via group membership.

Parameters:
- -prxGWS
- -domain
- -name

Example:
```
Remove-AllUserACEs -prxGWS $gws -Domain "boston.com" -Name
    "ivan_drinkwater"
```

Add-UserTo* cmdlets

This collection of cmdlets add the specified user, with the specified rights, to the specified object in the Gateway Server. There are different cmdlets for each type of object:
- Add-UserToGWSDataServicesSecurity
- Add-UserToGWSSettingsSecurity
- Add-UserToGWSOperationSecurity

- Add-UserToGWSGroupSecurity
- Add-UserToGWSHostSecurity

The security access masks can be specified by referencing the appropriate constants from the Proxy-Constants module.

Example:
```
Add-UserToGWSGroupSecurity -prxGWS $gws -Domain "boston.com" -Name
    "ivan_drinkwater" -GroupName "Support"` -HostAccessMask
    $hostFullUser -SessionAccessMask $sessionPlayRecordedSession
```

Remove-UserFrom* cmdlets

This collection of cmdlets removes the specified user from the specified object in the Gateway Sever. There are different cmdlets for each type of object:
- Remove-UserFromGWSDataServicesSecurity
- Remove-UserFromGWSSettingsSecurity
- Remove-UserFromGWSOperationSecurity
- Remove-UserFromGWSGroupSecurity
- Remove-UserFromGWSHostSecurity

Example:
```
Remove-UserFromGWSGroupSecurity -prxGWS $gws -Domain "boston.com" -
    Name "ivan_drinkwater"` -GroupID "{76837334-DA68-42F2-835F-
    7910661285B5}"
```

Security example script

The sample script "Add-NewGWSUser.ps1" demonstrates how to script adding a user account to a number of different securable objects in the Gateway Server, with access rights that are hard-coded into the script. This ensures consistent application of access rights.

The access rights can be changed in the script to suit your particular organizational needs.

Script paramters:
- -prxGWS: Gateway object
- -Domain
- -Name

Some common tasks

List all Gateway-based groups
```
$gws.getGroups()
```

Add a new Group
```
$gws.addGroup('QA', 'QA Hosts')
```

List "Active Gateway Data Services" collection (from Gateway Administrator "Active Status" folder)

```
$gws.getActiveClientConnections()
```

List "Active Master Connection Services" collection (from Gateway Administrator "Active Status" folder)

```
$gws.getActiveMasters()
```

List "Active Hosts" collection (from Gateway Administrator "Active Status" folder)

```
$gws.getActiveHosts()
```

List "Active Recordings" collection (from Gateway Administrator "Active Status" folder)

```
$gws.getActiveRecordings()
```

List "Reverse Connections" and "Pending Host Status Updates" collections (from Gateway Administrator "Active Status" folder); note that the Gateway Administrator segregates these items based on the "IsReverseConnection" property)

```
$gws.getActiveHostCtrlConnections()
```

Explore the methods of the "Active Host Control Connection" object

```
@($gws.getActiveHostCtrlConnections())[0] | Get-Member
```

Disconnect a Host (that is connected through the Gateway for remote control & other services)

```
@($gws.getActiveHosts())[0].disconnectHost()
```

Find group named "QA"

```
$gws.getGroups() | ? { $_.groupName -eq 'QA' }
```

List the Hosts that are members of the Group "QA"

```
($gws.getGroups() | ? { $_.groupName -eq 'QA' }).getHosts()
```

Add Host named "WKS-123" to group "QA"

```
($gws.getGroups() | ? {$_.groupName -eq
    'QA'}).addHost(($gws.getHosts() | ? {$_.name -eq 'wks-123'}))
```

Get list of all Unmanaged workstations

```
$gws.settings.getUnmanagedWorkstations()
```

Manage the unmanaged Host named "WKS-123" (this moves it from "Unmanaged" to "All Hosts")

```
($gws.settings.getUnmanagedWorkstations() | ? {$_.name -eq 'wks-
    123'}).isManaged = -1
```

Add the Host named "WKS-123" to the Group named "QA"

```
($gws.getGroups() | ? {$_.groupName -eq
    'QA'}).addHost(($gws.settings.getUnmanagedWorkstations() | ?
    {$_.name -eq 'wks-123'}))
```

Running "Service" scripts

Running and managing scripts as Scheduled Jobs

With the release of PowerShell 3.0 it becomes very convenient to schedule periodic maintenance jobs from the PowerShell environment. Jobs can be used to enforce policy rules, perform maintenance tasks and give real time notifications about events from Proxy Gateway Server. PowerShell Scheduled Jobs are managed by Windows Task Scheduler and can use all options available in it. In fact it is possible to use Windows Task Scheduler manually with PowerShell 2.0, but there is no built-in job management and reporting and you need to configure each task from the Task Scheduler's UI or to write your own configuration scripts. We will give descriptions on how to use Windows Task Scheduler with PowerShell 2.0 at the end of this chapter.
For some IT tasks you may prefer to use Task Scheduler UI after you create a new Scheduled Job from PowerShell.

"You can view and manage the jobs in Task Scheduler, enable and disable them as needed, run them or use them as templates, establish a one-time or recurring schedules for starting the jobs, or set conditions under which the jobs start." [http://technet.microsoft.com/en-us/library/hh847802.aspx]

Also, there is no place here to fully discuss such a big feature as PowerShell's Job Scheduling, but we will give all necessary information on how to use it, specific to running Proxy PowerShell scripts, and also include some pointers to further reading at the end of this section.

NOTE: You should run PowerShell with elevated privileges in order to register and manage Scheduled Jobs.

Usual way to schedule a new job is to create its "trigger" and "options" objects first. "Trigger" is what tells the job when to run and it is a mandatory parameter. For example:

```
PS> $trigger = New-JobTrigger –Daily –At '4 am' #runs task daily at 4am

or

PS> $trigger = New-JobTrigger -Once -At (Get-Date) -RepetitionInterval (New-TimeSpan -
Minutes 2) -RepetitionDuration ([TimeSpan]::MaxValue) #runs task every two minutes for
unlimited period of the time starting from now
```

"Trigger" can be created in any way "*New-JobTrigger*" command allows you, and has no specific restrictions related to Proxy PowerShell scripting.
"Options" object is mandatory for running Proxy PowerShell scripts and have advanced options for a scheduled job. The mandatory part is "*-RunElevated*" switch. For example:

```
PS> $opt = New-ScheduledJobOption -RunElevated # minimum set of
options you need to run Proxy PowerShell scripts.
```

When the "Options" and "Trigger" objects created, we can register a new scheduled job:

```
PS> Register-ScheduledJob -Name hostsToOU -FilePath
"C:\Proxy\Scripts\orgHostsByList.ps1" -Trigger $trigger -
ScheduledJobOption $opt -RunAs32 -Credential "username" # creates new
job with given name, trigger and options, running script from the file
system
```

When using *Register-ScheduledJob* there is only one parameter which is mandatory for Proxy scripts: - "*-RunAs32*". If this parameter is omitted PowerShell will run script in 64 bit environment which is not compatible with Proxy SDK 32 bit. But if you have Proxy SDK 64 bit installed you should omit this parameter from the command.

After new Scheduled Job is successfully registered, PowerShell has commands to manage it and its instances. For example you can view scheduled job(s):

```
PS> Get-ScheduledJob
```

, or you can run it on-demand in the current shell's background:

```
PS> Start-Job -DefinitionName hostsToOU
```

"Jobs that are started by using the Start-Job cmdlet are standard Windows PowerShell background jobs, not instances of the scheduled job. Like all background jobs, these jobs start immediately – they are not subject to job options or affected by job triggers – and their output is not saved in the Output directory of the scheduled job directory."
[http://technet.microsoft.com/en-us/library/hh847802.aspx]

Here how you can force the scheduler to run Scheduled Job by adding a new trigger to it which will run job once after 1min delay:

```
PS> $trigNow = New-JobTrigger -Once -At ((Get-Date) +
[timespan]"00:01:00")
PS> Add-JobTrigger -Name hostsToOU -Trigger $trigNow
```

NOTE: In order to use Get-Job, Receive-Job and other job related commands to work with Scheduled Job's instances from PowerShell, you either should use one of the Scheduled Job commands first or manually load "PSScheduledJob" module with *Import-Module cmdlet*:

```
PS> Import-Module PSScheduledJob
```

Once Scheduled Job has run, you can run *Get-Job* to fetch a list of available Job instances. Every individual job result will appear on the list and have the same name as the scheduled job that ran it (but different time and ID). Notice that there is a difference between Job and Scheduled Job. Scheduled Job is an entry in Windows Task Scheduler which can be viewed by calling to *Get-ScheduledJob* and it is effectively just a job description, not an actual running job instance. Once the "Trigger" criteria was met a real job instance will run creating new PowerShell background job which is not tied to any interactive PowerShell instance but can be managed from it with *Get-Job* and other related commands. For example you can retrieve all outputs from every run of Scheduled Job from any PowerShell instance with:

```
PS> Get-Job | ? { $_.name –eq "hostsToOU" } | % { Receive-Job -id
$_.id -Keep }
```

Output for scheduled jobs is kept on the disk of the computer where the job exists. Usually it's in your user profile directory in *\AppData\Local\Microsoft\Windows\PowerShell\ScheduledJobs*, with a subfolder for each job name. The resulting output will remain on disk after being "received" with *Receive-Job*, even if you didn't use the "*-Keep*" parameter. You can use *Remove-Job* to delete job's results from the disk or "*-MaxResultCount*" parameter of *Register-ScheduledJob and Set-ScheduledJob* to set how many of saved results to retain on the disk (default is 32). Every new result will delete the oldest one when limit is reached.
So, we covered here simple set of three cmdlets which you need in order to run Proxy Script as a Scheduled Job: - *New-JobTrigger*, *New-ScheduledJobOption* and *Register-ScheduledJob.* Please, take a look at links below and at PowerShell help for related commands before going to the next section where we will discuss what type of Proxy Scripts you can run as Scheduled Jobs.

Links:
- PowerShell 3.0: http://www.microsoft.com/en-us/download/details.aspx?id=34595
- Scheduled_Jobs : http://technet.microsoft.com/en-us/library/hh847802.aspx
- Scheduled_Jobs_Troubleshooting: http://technet.microsoft.com/en-us/library/hh849674.aspx
- Windows PowerShell Scheduled Job Cmdlets: http://technet.microsoft.com/en-us/library/hh849778.aspx
- PSScheduledJob Module ("about_*" sections has a lot of info): http://technet.microsoft.com/en-us/library/hh847863.aspx
- MSDN blog about Job Scheduling: http://blogs.msdn.com/b/powershell/archive/2012/03/19/scheduling-background-jobs-in-windows-powershell-3-0.aspx

Managing Scheduled Jobs
- *Add-JobTrigger* – add a new trigger to an existing scheduled job
- *Disable-JobTrigger* – turn off a scheduled job's triggers, but don't delete them
- *Enable-JobTrigger* – enable a previously disabled scheduled job trigger
- *Get-JobTrigger* – gets the triggers of scheduled jobs
- *New-JobTrigger* – create a new job trigger

- *Remove-JobTrigger* – remove a job trigger from the scheduled job
- *Set-JobTrigger* – reconfigure a trigger on a scheduled job
- *Get-ScheduledJobOption* – gets the options for a scheduled job
- *New-ScheduledJobOption* – creates a new option set
- *Set-ScheduledJobOption* – reconfigures a scheduled job's options
- *Disable-ScheduledJob* – disable, but do not delete, a scheduled job
- *Enable-ScheduledJob* – re-enable a previously disabled scheduled job
- *Get-ScheduledJob* – get the scheduled jobs on a computer
- *Register-ScheduledJob* – creates and registers a new scheduled job
- *Set-ScheduledJob* – reconfigure an existing scheduled job
- *Unregister-ScheduledJob* – removes a scheduled job entry

Managing Scheduled Job instances

- *Get-Job* – get background jobs
- *Receive-Job* - gets the results of background jobs
- *Remove*-Job - delete background jobs

Script types

Now, when we know how to run Proxy scripts with the help of PowerShell 3.0 scheduler, let's take a look at different kind of scripts we can run to help with Proxy Gateway Server management and maintenance. There are two types of scripts which we will cover here: short running scripts for periodic maintenance jobs and long running scripts for event notifications.

Short running scripts are just the same scripts as we already covered in other parts of this document and have no special properties. If you have some maintenance work to do, like moving Hosts around or changing their properties according to some policy (like in *Scripts\orgHostsByOU.ps1*), just follow the guidelines from previous section and apply it to the same script you will run from the interactive prompt. Simplest form of such script will be: import Proxy modules → connect to Gateway → do the job → disconnect.
Something like this:

```
Import-Module Proxy-Utils
Import-Module Proxy-Gateway
Import-Module Proxy-Gateway-Utils


Connect-Gateway 'localhost'
if(Test-Connection)
{
    $gws = Get-GatewayObject


    # Do the maintenance job here


    Disconnect-Gateway
}
```

Long running scripts are for the cases when you want to receive real-time notifications on something happening in Proxy Gateway Server (GWS). Such script will be scheduled to run at the system's startup and should have two distinct properties; It should subscribe for GWS event notifications and it should handle occasional disconnects.

Subscribing for GWS notifications is covered in "Proxy Gateway events: examples and templates" chapter and you can add your own events handling to your scripts following it's guidelines.

Handling of disconnects is something new and you only need it for long running scripts to survive GWS going down or network problems. "*Scripts*\taskWatchGatewayEvents.ps1" is a template for such a script. It uses an example action *$template_event_group_membership_action* from *Proxy-Gateway* module and cmdlet *Register-GroupMembershipChange* to register this action to be fired on the "Group Membership Change" event from GWS. When creating "trigger" and "options" for long running job, use following code as a guideline:

```
PS> $trigger = New-JobTrigger -AtStartup
PS> $opt = New-ScheduledJobOption -MultipleInstancePolicy StopExisting
-RunElevated
```

Or some other combinations which fit a profile of long-running script. The only mandatory parameter is "-*RunElevated*" in the "options" object.

It is a good idea to run your script from PowerShell interactive environment first, to make sure it's working, before assigning it to Scheduled Job.

NOTICE: Scheduled Script's output is not available until script is finished. This makes "Long running script" output invisible to `Receive-Job` command when it still running. Use some other mechanism to monitor long running script: - log files, email, Windows Event Log, etc. The "*Scripts*\taskWatchGatewayEvents.ps1" **example is using PSLog module which can be found at: http://gallery.technet.microsoft.com/scriptcenter/PSLog-Send-messages-to-a-db389927**

Running Scheduled Jobs with PowerShell 2.0

Note: If you are using 32bit version of Proxy SDK, all script and command invocations should use appropriate PowerShell binary:
. On 64bit Windows: "Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe"
. On 32bit Windows: "Windows\System32\WindowsPowerShell\v1.0\powershell.exe"

Here we explain how to run Scheduled Jobs using PowerShell 2.0 and Windows Task Scheduler's UI. This method is lucking two main properties of Scheduled Jobs from PowerShell 3.0, - managing job instances and retrieving job results. However you should be able to use file system to create a custom logs of your own (we don't discuss it here).

Running PowerShell scripts from the Task Scheduler is pretty straightforward, with a few caveats. One of them is the "script's running policy" for the PowerShell scripts on local computer. It should be setup in advance before adding new tasks to Task Scheduler. There are number of ways to setup your script execution policy and you can read PowerShell help on the

matter: "*get-help about_signing*" and "*get-help about_Execution_Policies*". We recommend using: "*Set-ExecutionPolicy RemoteSigned -scope CurrentUser -Force*" which will set policy for the current user to run local scripts "unrestricted", and downloaded scripts "signed". To set this up invoke next line from the command prompt:

```
> C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe -command
"&{Set-ExecutionPolicy RemoteSigned -scope CurrentUser -Force}"
```

After script execution policy is set it will be a good test to try to run your script from the command line first, to see if it actually working and can complete the task. We will use an example script which assigs Hosts to Gateway's groups by their Organizational Unit in the Active Directory: *Scripts\orgHostsByOU.ps1*, but you should use your own script unless this is exactly what you want to do with Hosts on the Gateway. So, try to invoke script from the command line and see if it got invoked with the new policy we set, and it's doing the right things on the gateway:

```
> C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe -command
"& 'C:\Proxy\PowerShell\Scripts\orgHostsByOU.ps1'"
```

Now, we can setup the actual task for the script. Open Task Scheduler from the `Computer\Computer Management\System Tools\Task Scheduler`. Right-click on the Task Scheduler `Library/Create Task...`:

On the property pages of "Create Task":
General:
- *Name*: Organize Hosts by OU
- *Description*: Assign new Hosts by their Active Directory Organizational Unit to Gateway groups with the same names (as units).

Security Options (same property page):
- *When running the task, use the following user account*: Choose User's identity for the script to run under. It should be the same User which identity will be used for the Gateway connection and who has rights on the Gateway to perform the operation. Also, it should be the same user as the one you have used to set PowerShell script's execution policy(otherwise you should use `-scope LocalMachine` in *Set-ExecutionPolicy*).
- *Run with highest privileges*: True.

Set other properties on the page according to your system and task preferences.

Trigger:
Choose when you want the script to run

Action:
- *Action*: Start a program
- *Program/script*: C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe
- *Add arguments*: *-command "& 'C:\Proxy\PowerShell\Scripts\orgHostsByOU.ps1'"*

To make sure your task is working, you can try to run it once manually, by right-click on it - "Run".

Links:

- Use Scheduled Tasks to Run PowerShell Commands on Windows: http://blogs.technet.com/b/heyscriptingguy/archive/2011/01/12/use-scheduled-tasks-to-run-powershell-commands-on-windows.aspx